SAT-Based Optimization with Don’t-Cares Revisited

Alan Mishchenko Robert Brayton

Department of EECS, UC Berkeley

{alanmi, brayton}@berkeley.edu

Ana Petkovska Mathias Soeken

Ecole Polytechnique Federale de Lausanne (EPFL)

{ana.petkovska, mathias.soeken}@epfl.ch

Abstract

The paper describes a SAT-based framework for logic optimization with don’t-cares aimed at reducing delay and area after LUT mapping. While individual components of the framework are known, its novelty is in synergistically combining the following aspects of SAT-based optimization for the first time: a) improved computation of delay and area criticality, b) novel reconvergence-driven windowing and divisor selection, c) the use of complete don’t-cares, and d) SAT-based generation of new useful cut-points in the network. Experimental results show that a preliminary implementation improves delay after LUT mapping at the cost of some area increase, compared to previous methods.

1. Introduction

Logic synthesis and technology mapping are often charged with the task of reducing delay in a logic network representing a hardware design. Delay optimization is important in FPGA synthesis because reduced delay correlates well with improved maximum clock frequency that can be achieved for the design. When delay optimization succeeds in reducing delay, area often increases, making area optimization under delay constraints another important goal of logic synthesis for FPGAs.

In this paper, we treat delay and area optimization uniformly. However, some methods differ depending on whether the goal of optimization is delay or area. It is noted when a notion or a heuristic is delay- or area-specific.

Most optimization methods for logic networks developed in the last few decades fall into one of the two categories:

- Those applied on a technology-independent level before mapping without taking into account a specific LUT size used during mapping.
- Those applied after technology mapping when the structure of the LUT network is fixed and can be modified only incrementally.

This paper generally follows the second approach, but it also compatible with the first approach, because it allows candidate divisors to be selected among those subject graph nodes that are not exposed as outputs of individual LUTs.

The paper also proposes a novel notion of delay/area criticality, which allows for better selection of target nodes during optimization, and a novel reconvergence-driven windowing and divisor selection, which improve runtime.

In addition to the above, the proposed method employs several known techniques in a way, which enhances the generality and expressive power of the optimization.

Figure 1 compares the proposed method (command &*mfsd* in ABC) against three other techniques, a) AIG-rewriting [10] (*dc*2), b) structural LUT mapping [11] (*if*), and c) don’t-care-based optimization [12] (*mfs*2). The following criteria are used for comparison:

- *Use of AIGs as an underlying representation* – this is preferred because it provides more candidate divisors for re-expressing the target node.
- *Accounting for the target LUT size* – this is important for making goal-oriented changes; otherwise structural bias hinders the mapper in finding a good mapping.
- *Use of dynamic programming* – this is desirable because it helps minimize delay at the target node by getting the best possible delay at its fanins.
- *Use of SAT-based methods* – these tend to explore a larger search space whereas other methods are often limited to fewer structural optimization opportunities.
- *Use of don’t-cares* – these allow for more aggressive transformations since the equivalence is only maintained on the care-set. It also reduces the runtime of QBF based functional evaluation.
- *Creating cut-points that are not in the current netlist* – these additional nodes synthesized by the algorithm can result in more optimization opportunities.

Feature for comparison	AIG rewriting (*dc2*)	Structural mapping (*if*)	DC-based optimizat. (*mfs*2)	This method (*mfsd*)
AIG based	Yes	Yes	No	Yes
LUT size aware	No	Yes	Yes	Yes
Uses dyn. program.	No	Yes	No	Yes
SAT-based	No	No	Yes	Yes
Uses don’t-cares	No	No	Yes	Yes
Create new cut-points	Yes	No/Yes	No	Yes

Figure 1. Comparing optimization methods.

Note that structural mapping creates new cut-points only when the target technology is LUT-structures [15] rather than single LUTs.

Some traditional LUT mapping is delay-optimal for a *fixed* subject graph [2][11]. However, the proposed algorithm can *change* the graph structure, thereby enabling mapping with even shorter delays.

The proposed method can be extended to standard cells, technology-independent AIGs, and logic structures composed of known primitives, such as MUXes or majority gates. The main difference, compared to the LUT networks, in the synthesis phase, when the result of synthesis is not a LUT structure, but a complex programmable cell [13] or a combination of gates from a standard-cell library [14].

The rest of the paper is organized as follows. Section 2 contains necessary background. Section 3 gives a top-level view of the optimization framework. Section 4 describes the components of the framework. Section 5 shows experimental results. Section 6 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which can influence the output value of f. The *support size* is denoted by $|X|$.

Expressions $\bar{x}$ and x are the *negative* and *positive literals* of variable x, respectively. "Negative" and "positive" are the *polarities* of variable x in the literals.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node n has zero or more fanins, i.e., nodes driving n, and zero or more fanouts, i.e., nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes delivering the results to the environment. A transitive fanin (fanout) cone (TFI/TFO) of node n is a subset of the network nodes, reachable through the fanin (fanout) edges of the node.

Internal flexibilities of a node in a network arise because of a limited controllability and observability. Lack of controllability occurs because some combinations of values are never produced at the fanins of the node. Lack of observability occurs because the node's value does not have an impact on the values of the POs under some values of the PIs. Examples can be found in [9].

These internal flexibilities result in *don't-cares* at the node n. The complement of the don't-cares is 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

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

where $z_i'(x)$ are the PO functions in a copy of the network with node n complemented, as shown in Figure 3 [9].

2.3 And-Inverter Graph

An *And-Invertor Graph* (AIG) is a combinational Boolean network composed of two-input AND gates and inverters. An AIG can be derived by factoring the functions of the logic nodes found in the network. The AND/OR gates in the factored forms can be converted into node AIGs using DeMorgan's rules. The AIG of the network is then constructed in a topological order by composing each node AIG into a network AIG.

A *cut* C of a node n is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to n passes through at least one leaf. Node n is called the *root* of cut C. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if cut size does not exceed K.

2.4 Boolean satisfiability

A *satisfiability problem* (SAT) takes a propositional formula representing a Boolean function and decides if the formula is satisfiable or not. The formula is *satisfiable* (SAT) if there is an assignment of variables that evaluates the formula to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver*. SAT solvers provide a satisfying assignment when the problem is satisfiable.

Modern SAT solvers can accept assumptions, which are single-literal clauses holding for one call to the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*.

2.5 Conjunctive Normal Form (CNF)

To represent a propositional formula in the SAT solver, important aspects of the problem are encoded using Boolean *variables*. The presence or absence of a given aspect is represented by a positive or negative *literal* of a variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a *CNF*. CNFs can be processed efficiently by mainstream CNF-based SAT solvers, such as MiniSAT [3].

Deriving a CNF for a subset of nodes of the Boolean network is performed by putting together CNFs obtained by converting each node. A CNF for a node is derived by deriving SOPs of the on-set and off-set of the Boolean function of the node, and converting these SOPs into CNF using the De Morgan rule.

3. Framework overview

The proposed optimization framework includes several components whose interaction is illustrated in Figure 2.

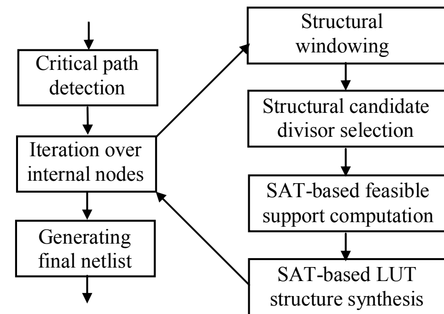

Figure 2: Overview of optimization framework.

In the case of delay optimization, processing begins by selecting delay-critical nodes. If an initial mapping is provided, this mapping is used to detect the critical region and compute a sequence of critical nodes. If no mapping is provided, a delay-oriented mapping is computed on the fly assuming that the critical paths contain the entire network.

Nodes are ordered by a heuristic, which selects a node that can benefit from delay optimization more than others.

For this, a priority queue is maintained during computation, which ranks nodes based on the number of critical paths passing through them, as discussed in Section 4.1.

For each node that is targeted for delay or area optimization, the following steps are performed:

- A topological *window* and candidate divisors are computed by a novel reconvergence-driven windowing and divisor selection method introduced in Section 4.2.
- A subset of candidate divisors is selected and proved to be feasible as a functional support of the target node, or proved that no feasible subset exists, using a novel divisor selection procedure, described in Section 4.3.

Finally, a new LUT structure, which by construction is guaranteed to reduce area or delay at a target node, is synthesized using QBF, as shown in Section 4.4. The main difference, compared to previous work [6][4][13] on QBF-based LUT-structure evaluation, is that don't-care are used, which improves the quality and at the same time reduces the runtime of the QBF solver.

For example, consider the following scenario that may appear during delay optimization. If the best delay of a target node in the preliminary mapping is 5, the framework attempts to change the circuit structure of the target node and its fanins, so that delay 4 is achieved. It is assumed here that each LUT has a delay of 1. Thus, divisors with delay 3 or less are selected. If it is possible to realize the target node using a LUT whose fanins have delay 3, then the node can be realized with delay 4, and the goal of delay optimization is achieved. If the node cannot be realized as a single LUT whose fanins have delay 3, the framework may check the existence of a LUT-structure composed of two LUTs, such that inputs to the fanout LUT of the structure have delay 3 or less, while inputs of the fanin LUT have delay 2 or less. If that structure exists, the target node can be realized with delay 4. If not, the computation may try a larger LUT-structure, or move to another node.

At the end of the computation, or if a timeout occurs, the final restructured LUT network is generated and returned. The network is guaranteed to have the delay projected by the framework during delay optimization.

4. Framework components

4.1 Detection of critical nodes

While most of the paper considers delay and area optimization uniformly, this section treats them differently.

SAT-based delay optimization is based on the notion of the *delay criticality* of a node. Delay criticality of a node shows how much reducing delay at the node may reduce the maximum delay of the network, measured as the largest number of LUTs on any path from a PI to a PO.

SAT-based area optimization is based on the notion of the *area criticality* of a node, which shows how much area could be potentially saved by changing the node. The area is measured as the number of LUTs.

Area criticality

We consider area criticality first. The *maximum fanout free cone* (MFFC) of a node in the network, is the set of all internal nodes that would be removed if the node were to be removed. The larger is the MFFC of a node, the more area-critical is the node. For example, consider a node whose all fanins have some other fanouts besides the given node. The MFFC of this node contains only the node itself. Even if the node is proved redundant or replaced by another node already present in the network, only one LUT is saved. If, on the other hand, the node has a large MFFC, it may be possible to reduce the MFFC size or reexpress the function of the node in such a way that it does not depend on some LUTs in its MFFC, but instead depends on the LUTs that are currently outside of the node's MFFC.

When resynthesis for area is performed, the nodes are considered in the decreasing order of their MFFC sizes. If an area-critical node cannot be improved, the node's fanins are considered in the decreasing order of their MFFC sizes. An attempt is made to replace the most area-critical fanin(s) by the nodes that are currently outside of the MFFC of the fanin(s). If this is possible, the transformation is accepted, thereby reducing the total area.

Delay criticality

To characterize delay criticality of a node, it is helpful to introduce the notion of a critical edge. An edge between a node and its fanin is *critical*, if increasing the delay of this edge results in increasing the delay at a PO. By extension, an internal node is *critical*, if it has at least one incoming or outgoing critical edge. The *critical region* of the network is the set of all critical nodes.

It is helpful to observe that two nodes may be critical and yet an edge between them, if it is exists, may be not critical, because the nodes are critical due to some other edges.

An *input-to-output critical path* (IOCP) is a sequence of critical edges originating at a PI and terminating at a PO. Similarly, an *output-to-input critical path* (OICP) is a sequence of critical edges originating at a PO and terminating at a PI. Two critical paths (CPs) are *different* if they differ in at least one edge.

Next, we introduce a method of computing how many different IOCPs (OICPs) pass through a given critical edge. To compute the number of IOCPs, we assign IOCP count of a PI to be 1, if the PI appears on a critical path, and 0, otherwise. Next, we traverse the network in a topological order from PIs to POs and set the IOCP count of an internal node to be equal to the sum of IOCP counts of the fanins. The OICP counts of the nodes are computed similarly, by exchanging the roles of PIs and POs. The *delay criticality* of a node is defined as the total number of IOCPs and OICPs passing through the node.

This sum is computed at the beginning and updated incrementally during delay optimization. Each time the delay of a node is reduced, the delay change is propagated to the TFI and TFO. The edges that were critical before, may be no longer critical, and as a result, the number of IOCPs and OICPs will be reduced. Both the edge delays and the CP counts are updated incrementally, by traversing the affected part of the node's TFI and TFO. Only if the global delay of the network has changed after updating the node, new critical edges are created, in which case the edge delays and the CP counts are recomputed from scratch.

In our implementation, the critical nodes are stored in the priority queue ordered by their delay criticality. The most critical node is repeatedly extracted from the queue and an attempt is made to improve delay of the node. If the attempt succeeds, the network is changed and the delay criticalities

of the nodes are incrementally updated, which is reflected in the ordering of the nodes in the queue.

It should be noted that the notion of delay criticality, as defined above, elegantly captures the node's role in the critical region of the network. If the sum of CP counts is 0, the node is outside of the critical region. The larger is the sum of CP counts, the more critical paths go through the node and the more likely improving delay at the node may reduce the maximum delay of the network.

To avoid making unnecessary changes, the network is backed up each time after its maximum delay has changed. If the maximum delay of the network is not reduced after a number of incremental delay optimization steps when a resource limit is reached, the network is restored to the state it was the time of the last backup.

4.2 Windowing and divisor selection

This section describes an improved windowing algorithm, which leads to better runtime and quality of results, compared to the traditional windowing.

Window computation

The traditional *structural window* for a node contains a fixed number of TFI/TFO levels of logic centered at the node, plus all paths originating in the limited TFI and terminating in the limited TFO [9]. Thus, by construction, this window contains all recovergences, such that at least one of the reconvergent paths goes through the node.

The main idea of the new windowing, is that the non-reconvergent paths do not provide don't-cares, even if they are included in the window. One possibility is to compute a new window by starting from the traditional window and removing all the nodes found on the paths without reconvergence in the current window. Another possibility, is to collect the recovergent paths directly, by performing a dedicated traversal starting from the node, as shown below.

For this traversal, we consider the netlist as an undirected graph, that is, we traverse fanins and fanouts of each node uniformly, without distinguishing them. Each node accessed during the traversal is marked as "visited once". If the same node is accessed again, it is marked as "visited more than once". All the nodes that are "visited more than once" along with the path connecting them with the target node, are collected and included in the window.

Candidate divisor selection

Once the window is computed, we need to determine the set of *candidate divisors*, that is, the set of nodes that may be used to reexpress the function of the target node while trying to optimize ir for delay or area. When the traditional windowing [9] is used, the set of candidate divisors is composed of two sets of nodes: (a) the window nodes in the TFI of the target node, and (b) the window nodes that are not in the TFI and the TFO of the target node, but whose support in terms of the window inputs, is a subset of the support of the target node.

In the previous subsection, we introduced an improved reconvergence-driven windowing algorithm. Here we similarly improve candidate divisor selection, by making it reconvergence-driven. To this end, we group the divisors using their topological information. Two divisors are *topologically related* if they share a path to the target node or if they are located on overlapping reconvergent paths. In both cases, the divisors share some support variables and may be may replace each other as fanins of the node.

For example, suppose we are building a new functional support of the target node, and at some point we found a *feasible* set of divisors, as defined in the next section. Furthermore, suppose one divisor does not meet the optimization requirements, in particular, its arrival time is too high, thereby precluding a delay improvement at the target node. In this case, we can remove this late-arriving divisor from the feasible set, temporarily resulting in an infeasible set. To derive another feasible set, there is no need to try divisors that are not topologically related to the removed one. Instead, we consider adding only divisors that are topologically related, but have smaller delay than the removed one. If adding one or more of them makes the set feasible, we achieved the optimization goal. Otherwise, we conclude that the target node's delay cannot be improved and the algorithm moves on the next target node.

Characterizing don't-cares

The window computed for node n is used to derive a Boolean relation defining the local care set of n, as shown in Figure 3. The circuit representation of the Boolean relation contains two copies of n's TFO, one of which has an inverter at n's output. A comparator network is added to compare the pairs of corresponding outputs of the window. The output of the comparator is set to 1; hence any satisfying assignment complementing n's function causes a difference to be seen by at least one of the window outputs. Thus, the constructed circuit represents *care set* of the node in the given window.

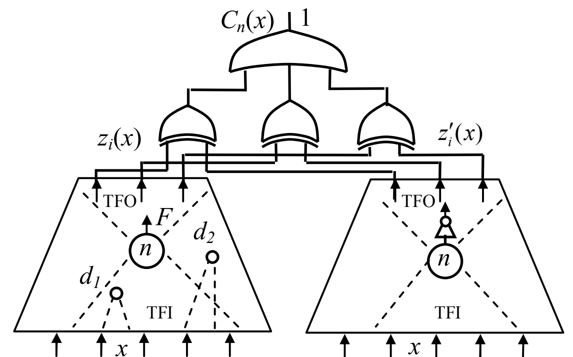

Figure 3. Deriving Boolean relation containing all possible implementations of the node in terms of candidate divisors.

The Boolean relation represented by the circuit structure in Figure 3 relates the function of n with functions of candidate divisors whose support is a subset of the support of the node's TFI, illustrated as divisors d_1 and d_2, shown in Figure 3. Note that candidate divisors may be contained in n's TFI, such as d_1, or just having its support contained in its TFI, such as d_2.

4.3 Checking feasibility of a set of divisors

A set of candidate divisors is *feasible* for implementing a target node, if the node's function can be expressed in terms of these candidates only. Obviously, the set of the

node's fanins is feasible but some fanins may be late-arriving and not useful for reducing delay of the node.

The following SPFD Theorem [12] is used check the feasibility of a set of divisors: *The set of divisors is infeasible iff there exist two care set minterms (m_1, m_2) such that the node has different values in the two minterms while each divisor has the same value in the two minterms*.

In this case, the set of divisors does not have enough expressive power to distinguish the two minterms, but the target node should be able to distinguish them.

There are two ways of checking whether a set of divisors is feasible. The first one, based on a resubstitution miter, was introduced in Section 4.4 of [12]. It is faster for large sets of divisors, but the drawback is that it requires the use of interpolation to derive the new function at the node.

The second way to check the divisor set feasibility is based on cube enumeration. In this case, we construct a SAT instance representing the node's care set, as shown in Figure 3. Note that the window output is set to 1. This SAT instance is used to perform three tasks:

- compute a care onset minterm m_1 of n,
- compute a special care offset minterm m_2 of n,
- expand m_1 against the offset if m_2 does not exist.

To perform the first task, n's output is assumed to be 1, and the SAT solver is used to find a satisfying assignment.

To perform the second task, n's output is assumed 0 while each candidate divisor is assumed to have the same value it had for m_1. If this problem is SAT, we have found a care offset minterm, m_2, while having the same values at the divisors as for m_1. According to the above theorem, this set of divisors is infeasible.

If the problem is UNSAT, the SAT solver's method, called *analyze_final*() in MiniSAT, is used to return a subset of candidate divisor literals, which should be present in a cube for it to have no overlap with the offset, that is, to belong to the onset plus the don't care set. Thus m_1 has been expanded to a cube having no overlap with the care offset. The resulting cube is used later as a blocking clause to make sure that future care on-set minterms are not covered by this cube. The process is repeated until all care onset minterms are covered by at least one cube, or until the infeasibility of the set of divisors has been proved.

The advantage of checking the divisor set feasibility using cube enumeration is that it derives the new function at the node as a by-product. However, since large functions often have more cubes, this method can be slower for functions with more than 10 inputs.

4.4 Selecting a set of divisors

In the case of delay optimization, the above procedure checks the existence of a feasible subset of divisors that can reduce the delay of n from D to D-1. This procedure is called for a set of all divisors of the node having delay D-2 or less. If the set is not feasible, we know that the node cannot be implemented with delay D-1, given the circuit structure used to generate the candidate divisors.

If the given set of divisors is feasible, the procedure returns a reduced subset selected by calls to *analyze_final*(). If the subset contains no more than K divisors, the target node can be implemented using one K-LUT. If the resulting subset contains more than K divisors, we try to find a LUT-structure implementing the node in terms of more than one K-LUT, such that the delay of the target node does not exceed D-1.

For this, we consider the topology of a selected LUT-structure and the delays of the set of divisors. For example, if the LUT-structure contains two LUTs in tandem, then for the LUT-structure to be delay-reducing, the fanins of the top LUT should have delay D-2 or less, while the fanins of the bottom LUT should have delay D-3 or less.

4.5 Synthesizing a LUT-structure

SAT-based synthesis of LUT-structures for the given Boolean function was pioneered in [6]. Our implementation uses a dedicated Quantified Boolean Formula (QBF) solver in ABC [1], which is based on iterative SAT solving.

Computing a LUT-structure $S(x, p)$ that can implement a Boolean function $F(x)$, is done by checking satisfiability of the QBF formula: $\exists p \forall x\ [S(x, p) = F(x)]$. If a satisfying assignment for the parameters p exists, it shows how to configure the LUT structure to realize $F(x)$.

Unlike the case when $F(x)$ is completely specified and is given explicitly as a truth table or as a circuit, the formulation in this paper uses don't-cares. To extend the above approach for incompletely specified functions, Boolean function $F(x)$ and the care-set $C(x)$ are represented implicitly as a SAT instance similar to the one constructed in Section 4.3 to relate the value of the node with those of the candidate divisors. The difference in the new SAT instance is that, instead of comparing a window containing the original node and the same window containing the complemented node, as shown in Figure 3, we are comparing the node's original implementation with the node's implementation as a LUT-structure, $S(x, p)$.

For a specific minterm, say $p = p_0$, the above SAT instance checks whether the LUT-structure $S(x, p_0)$ initialized with values p_0 implements the target node on the care-set. If it does, the SAT instance is UNSAT, and the desired LUT structure is represented by minterm p_0. If it is SAT, we get a care-set minterm x_0, for which parameter values p_0 fail to implement the given function.

The minterm x_0 is substituted into $S(x, p)$, and the resulting function, $W_0(p) = S(x{=}x_0, p)$, is used as an additional constraint for $\exists p[S(x, p) == F(x)]$. If the resulting formula is SAT, we have another set of specific values of p, p_1, which is handled in the same way as p_0, generating another constraint $W_1(p)$.

If at some point the formula is UNSAT, the QBF instance has no solution, meaning that the desired LUT structure does not exist. An upper bound on the number of iterations is $2^{|x|}$, but in practice convergence is often achieved much faster. An additional speedup can be achieved by adding symmetry-breaking clauses [4] and solving multiple QBF instances concurrently [13].

5. Experimental results

A preliminary version of the proposed SAT-based optimization framework specialized for delay optimization has been implemented as command *&mfsd* in ABC [1]. This command differs from two SAT-based commands: *mfs2* [12] for LUT-mapped networks and *mfs3* [13] for standard-cell-mapped networks.

The implemented version is limited in the following way:

- Only delay optimization is considered
- All nodes are targeted for delay optimization rather than some nodes on the critical paths.
- The proposed reconvergence-driven techniques are not enabled and, instead, the traditional windowing and candidate divisor selection are used [12].

The method implemented in *&mfsd* was compared against delay optimization by mapping into LUT structures composed of two 4-input LUTs (called "*S44*") [15]. These two methods are similar in that they use 1) dynamic programming, 2) AIGs as the underlying data-structure, and 3) generation of new delay-oriented cut-points, which are outputs of the bottom LUT feeding into the top LUT.

The methods differ in that *S44* considers multiple cuts at each node and uses Boolean decomposition implemented with the truth tables to target the 2-LUT structures. The proposed method uses SAT-based LUT-structure synthesis on the care-set while limiting the support to a set of feasible divisors. The number of sets considered is fewer than the number of cuts in *S44* but the expressive power of SAT-based synthesis with don't-cares is higher.

Both *S44* and the proposed method have been applied to a suite of 60 industrial designs ranging in size from 1K to 50K 4-LUTs. Three runs were performed: (1) only *S44*, (2) only *&mfsd*, (3) *S44* followed by *&mfsd*.

The results are:

1) *&mfsd* improved delays, expressed in terms of 4-LUT levels, by 4.0% compared to *S44* alone,
2) when both *S44* and *&mfsd* are used, the delays improved by 5.3%, compared to *S44* alone.

The runtimes of the proposed method are reasonable. For the largest examples, it takes about 3-5x longer than *S44* and did not exceed 5 min for any of the designs.

In these preliminary runs, the area increased by more than 10%. It is expected that the area increase will be less when area recovery under delay constraints is performed.

6. Conclusions

The paper proposes a new powerful method for optimization of networks mapped for LUT-based FPGAs. The method is more general than the known methods and improves most of the aspects of SAT-based delay and area optimization with don't-cares:

- windowing and candidate divisor computation are reconvergence-driven, which reduces the size of logic considered and the runtime, while improving the expressive power and the quality of results;
- divisor selection is modified to find a minimal set of divisors using a new efficient SAT-based procedure, which efficiently tries different divisor combinations to find the one with a minimum cost;
- a new flavor of QBF based evaluation procedure allows for updating several LUTs at a time and works for multi-output windows, while the previous SAT-based optimization with don't-cares [12] can change only one LUT at a time, while that changing more than one LUT [4] does not use don't-cares.

Future work in this area will include:

- fine-turning SAT and QBF engines to reduce runtime
- extending these methods to work for standard cells
- developing SAT-based methods for topologically constrained LUT architectures, such as 2D and 3D meshes, in which each LUT can communicate with a limited number of neighbors.

7. Acknowledgements

This work is supported in part by NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" and SRC contract 2710.001 "SAT-based methods for scalable synthesis and verification".

8. References

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

[2] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", IEEE TCAD'94, Vol. 13(1), pp. 1-12.

[3] N. Een and N. Sörensson, "An extensible SAT-solver", *Proc. SAT'03*, LNCS 2919, pp. 502-518.

[4] Y. Hu, V. Shih, R. Majumdar, and L. He. "Efficient SAT-based Boolean matching for heterogeneous FPGA technology mapping", *Proc. ICCAD'07*.

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

[6] A. Ling, D. Singh, and S. Brown. "FPGA PLB evaluation using Quantified Boolean Satisfiability", *Proc. FPGA'05*. http://www.eecg.toronto.edu/~brown/papers/fpl05-ling.pdf

[7] A. Malik, R. K. Brayton, A. R. Newton, and A. Singiovanni-Vincentelli, "Two-level minimization of multi-valued functions with large offsets", *IEEE TCAD'91*, Vol. 10(4), pp. 413-424.

[8] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Cutless FPGA mapping", *ERL Technical Report*, EECS Dept., UC Berkeley, 2007.

[9] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *DATE '05*, pp. 418-423.

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

[11] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

[12] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. "Scalable don't-care-based logic optimization and resynthesis", *ACM TRETS*, Vol. 4(4), April 2011, Article 34.

[13] A. Mishchenko, R. Brayton, W. Feng, and J. Greene, "Technology mapping into general programmable cells", *Proc. FPGA'15*.

[14] A. Mishchenko, R. Brayton, T. Besson, S. Govindarajan, H. Arts, and P. van Besouw, "Versatile SAT-based remapping for standard cells", *Proc. IWLS'16*.

[15] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", *Proc. DATE'12*, pp. 1579-1584.