SAT-Based Complete Don't-Care Computation for Network Optimization

Alan Mishchenko
Department of EECS
University of California, Berkeley
alanmi@eecs.berkeley.edu

Robert K. Brayton
Department of EECS
University of California, Berkeley
brayton@eecs.berkeley.edu

Abstract

This paper describes an improved approach to Boolean network optimization using internal don't-cares. The improvements concern the type of don't-cares computed, their scope, and the computation method. Instead of the traditional compatible observability don't-cares (CODCs), we introduce and justify the use of complete don't-cares (CDC). To ensure the robustness of the don't-care computation for very large industrial networks, a windowing scheme is implemented, which computes substantial subsets of the CDCs in reasonable time. Finally, we give a SAT-based don't-care computation algorithm, which is more efficient than BDD-based algorithms. Experimental results confirm that these improvements work well in practice. Complete don't-cares allow for a reduction in the number of literals compared to the CODCs. Windowing guarantees robustness, even for very large benchmarks, to which previous methods cannot be applied. SAT reduces the runtime, making don't-cares affordable for a variety of other Boolean methods applied to the network.

1 Introduction

Optimization of Boolean networks using don't-cares plays an important role in technology independent logic synthesis and incremental re-synthesis of mapped netlists. Traditionally, only satisfiability don't-cares (SDCs) and compatible observability don't-cares (CODCs) have been used [19]. The classical algorithm to compute CODCs [21] implemented in SIS [22] became the method of choice for many industrial tools. Later improvements to this algorithm concerned a more robust implementation [20], independence from the local function representation [2], and generalization to multi-valued networks [7].

CODCs form a subset of the complete don't-cares (or complete flexibility) [12] projected onto a node by its context in the multi-level network. It was shown experimentally [11] that the computation of CDCs is comparable in runtime and memory requirements, while, as expected, the amount of don't-cares computed is larger than for CODCs. The presentation in [11][12] considers the most general case of non-deterministic multi-valued networks, leaving questions about its efficiency when applied to purely binary networks.

The *first contribution of this paper* is in developing a specialized version of the multi-valued don't-care computation algorithm [11][12], to work on binary networks, and in showing that this algorithm leads to an increase in optimization quality, due to the additional freedom provided by the CDCs, compared to CODCs.

The traditional don't-care optimization in SIS is performed using the whole network as the context for each node. This restricts the use of don't-cares to small or medium-sized networks. To apply the same method to larger networks, the network can be partitioned with the scope of computation limited to one partition at a time. Such methods have not been published, but might be part of some industrial tools. We suspect that partitioning for don't care computation is difficult, ad hoc, and implementation dependent.

The *second contribution of the paper* concerns the computation of don't-cares in large industrial designs. We propose a non-partitioning scheme, called *windowing*, which efficiently trades quality for runtime in network optimization. Windowing guarantees that the maximal flexibility, within a context limited by a fixed number of logic levels, is captured. Windowing is fast because construction of a window for a node involves only a small number of surrounding nodes, making it unnecessary to traverse the whole network. Windowing is not a partitioning scheme because each node has its own window, which may overlap with windows computed for other nodes. Finally, windowing is dynamic and can be performed "on the fly", without the need to duplicate or otherwise modify the network or its parts. The latter quality makes windowing useful for applications that frequently update the network, e.g. decomposition-mapping [13].

The *third contribution of the paper* concerns the use of Boolean satisfiability [9][15], rather than BDDs or SOPs, for the computation of don't-cares. We show that SAT is responsible for speed-ups in the computation, making CDCs easy to compute and affordable enough, so that many procedures, which previously relied on algebraic

methods, can now be extended to Boolean methods based on don't-cares.

In combination, these contributions provide improved efficiency, quality, and ruggedness for technology independent logic synthesis.

The paper is structured as follows: Section 2 establishes the background. Section 3 defines CDCs and compares them with CODCs. Section 4 presents windowing. Section 5 describes and compares BDD-based and SAT-based approaches to the CDC computation. Section 6 gives experimental results, and Section 7 concludes the paper.

2 Background

Definition. A *completely specified Boolean function* (CSF) is a mapping from n-dimensional ($n \geq 0$) Boolean space into a single-dimensional one: $\{0,1\}^n \rightarrow \{0,1\}$.

A *don't-care* for a logic function allows it to have either 0 or 1 as a possible value. If, for some input combinations, the output of the function is a don't-care, this function is called an *incompletely specified Boolean function* (ISF).

An assignment of n Boolean variables is called a *minterm*. A CSF has *negative* (*positive*) *minterms*, which correspond to the assignments, for which it takes values 0 (1). The positive and negative minterms are called the *care minterms*. An ISF additionally has *don't-care minterms*, which correspond to the assignments, for which the function is flexible and can be either 0 or 1.

A CSF is *compatible* with an ISF (*implements* the ISF), if the CSF can be derived from the ISF by assigning either 0 or 1 to each don't-care minterm.

Given several ISFs, the *largest* ISF is the one that has the largest number of don't-care minterms.

Definition. A *Boolean network* is a directed acyclic graph with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs).

Typically, the nodes and their output signals are named the same. The output of a node may be an input to other nodes called its *fanouts*. The inputs of a node are called its *fanins*. If there is a path from node A to B, then A is said to be in the *transitive fanin* of B and B in the *transitive fanout* of A. The transitive fanin of B, TFI(B), includes B and all nodes in its transitive fanin, including the PIs. The *transitive fanout* of B, TFO(B), includes B and all nodes in its transitive fanout including the POs.

The functionality of a node in terms of its immediate fanins is called the *local function* of the node. The functionality in terms of the primary inputs of the network is called the *global function*.

3 Complete don't-cares

Consider an individual node represented by its CSF. It is not possible to change the node's function without changing the node's behavior. However, the situation is different when the node is considered in its context in the network. In this case, the node's function can often be substantially modified, without changing the behavior of the network. This is because other nodes prevent some combinations of inputs from reaching the node as well as hiding the node's output from the POs under some conditions.

The flexibility allowed in the implementation of a node can be represented as an ISF. A don't-care minterm of the ISF represents a combination of the node's input variables, for which the value of the node's output is not required for the POs of the network to produce the correct values.

Definition. The *complete don't-cares* (CDCs), or *complete flexibility* (CF), of a node in the binary network, is the largest ISF, whose don't-care minterms represent conditions when the output of the node does not influence the values produced by the POs of the network.

The CDCs are important for network optimization because replacing the node's function by any CSF compatible with the ISF representing a node's CDCs, does not change the functionality of the POs of the network.

A key observation about CDCs is that they are *not* compatible. That is, some POs of the network may produce incorrect values if CDCs are derived for several nodes and used independently. In this sense CDCs differ from CODCs [21]. However, if CDCs are computed and used immediately to optimize a node before moving on to another node, compatibility is not required. In this case, the CDCs computed and used for each node reflect all prior changes to the nodes.

The CDCs include two major parts, the satisfiability don't-cares (SDCs), which arise because some combinations are not produced as the inputs of the node, and the observability don't-cares (ODCs), which arise because under some conditions the output of the node does not matter. Figure 1 shows a situation when node F has SDCs in the local space ($x = 0$, $y = 1$) due to limited controlability, while node G has ODCs in the global space ($a = 1$, $b = 1$) due to the limited observability.

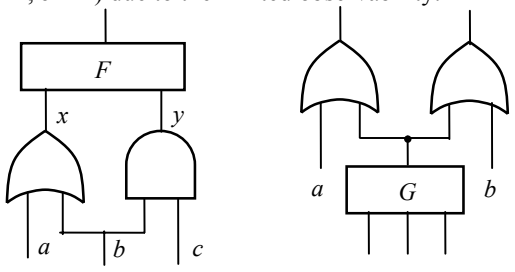

Figure 1. Example of SDCs and ODCs.

Don't-care computations are traditionally performed in the context of the whole Boolean network, as exemplified by SIS [22]. In the case of CDCs, this approach guarantees that the don't-cares computed are the largest don't-cares possible for a node.

However, in many cases, the network is too large, and the computation becomes slow or impossible. In such cases, a method of limiting the scope is needed to restrict the computation to a relatively small sub-network. The don't-cares computed for the node in this sub-network should be complete with respect to the sub-network, but will not be complete in general, i.e. considering a larger sub-network could result in more don't-cares with a likely increase the runtime.

Not only the size but also the "shape" of the sub-network is critical to get sufficiently large don't-cares. If a sub-network is large but does not include the nodes responsible for producing most of the don't-cares, the computation will be ineffective.

We developed a *windowing* method, which limits the scope of the don't-care computation to only a few logic levels on the fanin/fanout side of the node. An important observation is that reconvergence is responsible for don't cares; hence along with the near TFI and TFO of the node, a window should contain all *re-convergent paths* that begin and terminate in these nodes.

For the special case, when the inputs to the window have disjoint supports in terms of the PIs, while all outputs of the window belong to the POs of the network, the CDCs computed for a node in the window are equal to the CDCs when the whole network is considered.

4 Windowing

This section contains a detailed discussion of the windowing algorithm introduced in [13].

Definition. Given a directed acyclic graph and two non-overlapping subsets of its nodes, one set is called the *leaf set* and the other the *root set*, if every path from any node in the root set towards the sources of the graph passes through some node in the leaf set.

Definition. Given a directed acyclic graph and two subsets of its nodes, which are in the leaves/roots relationship, a *window* is a subset of nodes of the graph, which contains the roots and all nodes between the root set and the leaf set. The nodes in the leaf set are delimiters and do not belong to the set of nodes included in the window.

Definition. A path between a pair of nodes is *distance-k* if it spans exactly *k* edges between the pair.

Definition. Two nodes are *distance-k* from each other if the shortest path between them is distance-*k*.

The pseudo-code in Figure 2 and the example in Figure 3 describe the flow of the window computation algorithm. Procedure *Window* takes a node and two integers, which define the number of logic levels on the fanin/fanout sides of the node to be included in the window. It returns the leaf set and the root set of the window. With minor modifications, this procedure can compute a window for a set of nodes, which, in general, can be neither adjacent nor in the fanin/fanout relationship.

```
nodeset Window( node N, int nFanins, int nFanouts )
{
    nodeset  I1 = CollectNodesTFI( {N}, nFanins );
    nodeset  O1 = CollectNodesTFO( {N}, nFanouts );
    nodeset  I2 = CollectNodesTFI( O1, nFanins + nFanouts );
    nodeset  O2 = CollectNodesTFO( I1, nFanins + nFanouts);
    nodeset S = I2 ∩ O2;
    nodeset L = CollectLeaves( S );
    nodeset R = CollectRoots( S );
    return (L, R);
}
```

Figure 2. Computation of a window for a node.

The procedure *CollectNodesTFI* takes a set *S* of nodes and an integer number *m*, $m \geq 0$, and return a set of nodes on the fanin side, which are distance-*m* or less from the nodes in *S*. An efficient implementation of this procedure for small *m* (for most applications, $m \leq 10$) iterates through the nodes that are distance-*k* ($0 \leq k \leq m$) from the given set. The distance-0 nodes are the original nodes. The distance-(*k*+1) nodes are found by collecting those fanins of the distance-*k* nodes, which were not visited before. The procedure *CollectNodesTFO* is similar.

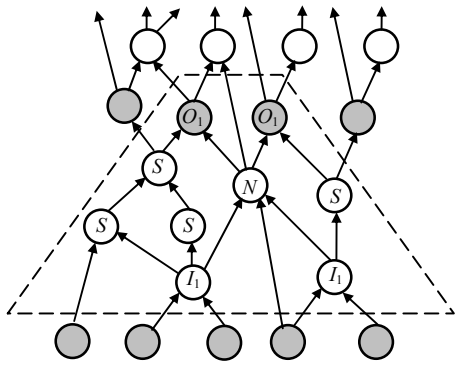

Figure 3. Example of a 1 x 1 window.

Procedures *CollectLeaves* and *CollectRoots* take a set of window's internal nodes and determine the leaves and roots of this window. The leaves are the nodes that (a) do not belong to the given set, and (b) are fanins of at least one of the node in the set. Similarly, the roots are the nodes that (a) belong to the given set, and (b) are fanins of at least one node not in the set. Note that some of the roots computed in this way are not in the TFO cone of the original node(s), for which the window is being computed, and therefore can be dropped without violating the

definition of the window and undermining the usefulness of the window for the don't-care computation.

We typically refer to the window constructed for a node by including *n* TFI logic levels and *m* TFO logic levels as an *n* x *m* window. For example, Figure 3 shows a 1 x 1 window for node *N* in a network. The nodes labeled I_1, O_1 and *S* are in correspondence with the pseudo-code in Figure 3. The window's roots (top) and leaves (bottom) are shaded. Note that the nodes labeled by *S* do not belong to the TFI and TFO cones of node *N*, but represent the reconvergent paths in the vicinity of node *N*. The left-most and right-most roots can be dropped, as explained above.

5 Don't-care computation

The network optimization discussed in this paper iterates through all the nodes of the network. For each node, the CDCs are computed and used to simplify the node before optimizing the next node. The computation of CDCs for a node can be performed in the context of the whole network, if the network is small; otherwise, a window is constructed for the node. Without limiting the generality of the CDC computation methods, we discuss these methods as applied to a node in the whole network. If a window is used, the network is the sub-network defined by the window containing the node.

The general approach to computing the CDCs of a node in the non-deterministic multi-valued network [11][12] relies on the use of an additional variable *z* for the output of the node, and the computation of Boolean relations in terms of the PI variables, the PO variables, and variable *z*.

This approach can be simplified for a node in a deterministic binary network; the computation can be performed without variable *z* or Boolean relations. In both BDD-based and SAT-based implementations, we consider two instances of the same network, which only differ in an inverter at the output of the given node in the second copy of the network (Figure 4). This duplication of the network is an imaginary construction, done for the sake of the presentation and not actually implemented in software.

The first network represents the original behavior, while the second represents the behavior of the network that produces the opposite value at the node. The functionalities of these networks are compared in order to detect when the change in the node's behavior influences the values at the POs. To this end, the two networks are transformed into a miter [1] derived by combining the pairs of PIs with the same names and feeding the pairs of POs with the same names into EXOR gates, which are ORed to produce the only output of the miter (Figure 4).

5.1 Computation using BDDs

The BDD-based CDC computation begins by deriving the global functions of the primary outputs of the two networks, $\{f_i(x)\}$ and $\{f_i'(x)\}$, where the index *i* varies over the POs. Next, the function of the output of the miter, $C(x)$, is derived, representing the care set in the global space:

$$C(x) = \sum_i [f_i(x) \oplus f_i'(x)].$$

The ODCs of the node in the global space is the complement of the care set:

$$ODC(x) = \overline{C(x)} = \prod_i [f_i(x) \equiv f_i'(x)],$$

The local CDCs are computed by imaging the global ODCs into the local space. To this end, mapping $M(x,y)$ is used, which relates the global and local spaces:

$$CDC(y) = \forall_x [M(x,y) \Rightarrow ODC(x)].$$

This computation adds the SDCs, $\overline{M(x,y)}$, to the already computed ODCs. It requires that the don't-care minterm *y* was a don't-care for *all* assignments of the PI variables *x*. If external don't-cares are present, they are added to the observability don't-cares.

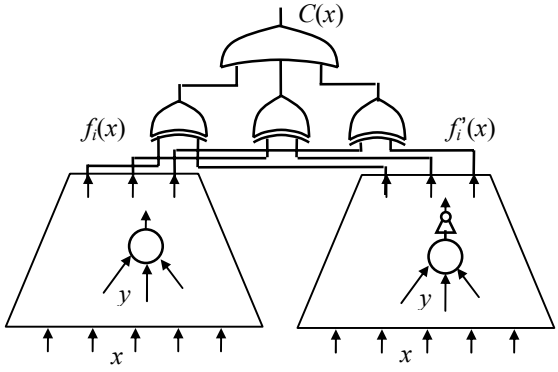

Figure 4. Illustration of CDC computation.

5.2 Computation using SAT

The use of SAT [9][15] in the CDC computation is similar to the use of SAT in combinational equivalence checking [4]. A solution of the SAT problem represented by the miter in Figure 4 gives satisfying assignments of all signals producing value 1 at the output of the miter. The values of variables *y* (the inputs of the node) in this solution form a *care set minterm* in the local space of the node. This is because, for them, we know the values of the PI variables *x*, such that at least one pair of POs produces different values.

All the care set minterms in terms of variables *y* are collected by enumerating through the satisfying assignments of the SAT problem and adding breaking clauses for each of them. A similar method of generating the satisfying assignments is described in [10], except that we do not undo the implication graph when a new satisfying assignment is found. We treat satisfying assignments similar to conflicts. In both cases, non-chronological backtracking is performed to the highest level determined using the new clause.

The SAT-based CDC computation is summarized in Figure 5. The top-level procedure *CompleteDC* takes node N and its context S given by the network, or by a window constructed for node N. Procedure *ConstructMiter* applies structural hashing [8] to the miter of the two copies of S shown in Figure 4. The resulting compact AND-INV graph G is constructed in one DFS traversal of the nodes in S, without actual duplication.

Random simulation of G reduces the runtime of the SAT solver. Indeed, each assignment of the PIs variables x, such that the output of the miter is 1, detects a care minterm of the node in terms of variables y. Only unique care minterms are collected. In practice, simulation is performed until "saturation" when, after a fixed number of rounds of bit-parallel simulation (typically, 5-10 rounds), the simulator did not turn up a new care minterm.

The CNF P is the conjunction of clauses derived from G and the complement of F_1, the part of care set derived by random simulation. The CNF of G is derived using a well-known technique, which adds three CNF clauses for each AND gates. For example, the clauses added for the gate $ab = c$ are: $\bar{c} + a$, $\bar{c} + b$, $\bar{a} + \bar{b} + c$. The only other clause added to the CNF is the clause asserting that the PO of the miter is equal to 1.

The SAT solver enumerates through the satisfying solutions, F_2, of the resulting problem representing the remaining part of the care set. In practice, it often happens that the SAT problem has no solutions ($F_2 = 0$). In such cases, SAT is only useful to prove the completeness of the care set derived by random simulation.

```
function CompleteDC( node N , context S )
{
    aig G = ConstructMiter( S, N );
    function F1 = RandomSimulation( G );
    cnf P = CircuitToCNF( G ) ∧ FunctionToCNF( F1' );
    function F2 = SatSolutions( P );
    return (F1 + F2)' ;
}
```

Figure 5. Pseudo-code of SAT-based CDC computation.

This approach solves the SAT problem by enumerating through the satisfying assignments, which represent *minterms* of the care set of the given node. Therefore, it is limited to nodes with roughly 10 inputs or less, which is typically the case for most Boolean networks. To make the approach work for networks nodes with a larger number of inputs, the implementation of the SAT solver should be further modified to return incomplete satisfying assignments, which correspond to *cubes* rather than minterms of the care set.

6 Experimental results

The methods for computing CDCs of a node in the context of both a window and the whole network are implemented in the MVSIS environment [18].

The SAT-based part is implemented using MiniSat [3], an "extensible SAT solver". Despite its small size (600 lines of C++ code written without STL), MiniSat is very efficient. In our experiments, it outperformed several popular SAT solvers. Moreover, the implementation of MiniSat is easy to understand and modify, in complete agreement with the original intention of its developers.

The experiments are divided into several parts, in correspondence with the contributions of the paper. All measurements are made on a Windows XP computer with a 1.6GHz CPU and 1Gb RAM, although less than 256Mb of RAM are needed for the largest benchmarks in Table 4.

The resulting networks are verified using a SAT-based verifier in MVSIS designed along the lines of [4][6].

6.1 Experiment 1: Comparing CODCs vs. CDCs

First, we compared the optimization potential of CODCs and CDCs. The BDD-based don't-care computation flow was used in both cases. We considered the largest MCNC benchmarks [23], for which BDDs could be constructed. Table 1 compares the runtime and the number of literals of the CODC-based command *full_simplify* from the distribution of SIS, and the new CDC-based command *mfs* implemented in MVSIS and later ported to SIS. The SIS version was used in this experiment. Both *full_simplify* and *mfs* perform Boolean resubstitution followed by the SOP minimization as part of don't-care-based optimization. Network *sweep* in SIS, which eliminates constants and single-input nodes and removes internal nodes without fanouts, is performed before and after both commands.

The first column in Table 1 lists the benchmark names, followed by five columns containing the number of literals: (1) after initial sweeping only ("sweep") (which is the starting point of the other columns), (2) after *full_simplify* ("fs"), (3) after *mfs* without the "advanced features" ("mfs"), (4) after *mfs* with 2 x 2 windowing without the "advanced features" ("mfsw"), and (5) after *mfs* with the advanced features enabled ("MFS"). The advanced features include on-the-fly merging of nodes with functionality equivalent up to complementation and phase-assignment, performed as part of optimization. In columns (2) and (3) these features are disabled to have a fair comparison with *full_simplify*. Some benchmarks could not be processed by *full_simplify* because of the large BDD sizes (indicated by the dash in the table).

The last three columns give the runtimes in seconds. The bottom line shows the average of the ratios of the improvements in the number of literals, achieved by each command, compared to the number of literals in the

original (swept) benchmarks. The asterisk in Table 1 indicates that, to compare against *fs*, the averages of the ratios are taken only over the 11 examples where *fs* could complete.

Table 1. Comparing CODCs vs. CDCs.

Name	Literals in factored forms					Runtime, sec		
	sweep	*fs*	*mfs*	*mfsw*	*MFS*	*fs*	*mfs*	*mfsw*
dalu	2976	2140	1741	2250	1747	64.8	2.1	0.8
des	6101	5677	5616	5920	5334	8.1	3.7	3.7
frg2	2010	1454	1440	1477	1409	5.1	0.6	0.5
i10	4355	-	3809	3853	3694	-	82.2	1.2
k2	2928	2889	2663	2878	2641	6.2	3.9	3.3
Pair	2420	2179	2143	2151	2139	3.5	2.9	0.4
c1355	992	984	992	992	992	22.8	86.7	0.2
c1908	1058	869	870	869	754	12.4	10.9	0.3
c2670	1570	1189	1215	1411	1195	4.9	2.8	0.3
c432	335	298	288	299	288	2.2	0.9	0.3
c499	576	568	576	576	576	1.0	13.0	0.1
c5315	3531	3184	3168	3176	2951	31.5	7.3	0.9
c7552	4750	-	4057	4079	3594	-	50.0	1.4
c880	648	625	624	625	624	1.2	7.2	0.1
Ave	**1.00**	**0.88***	**0.86**	**0.90**	**0.83**	**1.00**	**0.87**	**0.07**

The noticeable improvement in runtimes from *full_simplify* to *mfs* are because they are implemented in a different way and use different BDD variable orders (in general *fs* should be faster). Comparing literals, Table 1 shows that the CDCs outperform CODCs in the context of the whole network (columns "fs" vs. "mfs"). Although not included in the tables, other experiments have shown that, on average over all considered benchmarks, the CDCs typically contain 20% more don't-care minterms in the local spaces of the nodes, compared to the CODCs.

For CDCs with windowing (column "mfsw"), Table 1 shows that the literal count is almost as good as in the case of CODCs in the context of the whole network (column "fs"), but the runtime is only 7% of that of "fs". Additionally, window-based optimization (*mfsw*) is applicable to very large circuits well beyond the scope of *full_simplify* in SIS or *mfs* .

6.2 Experiment 2: The effect of windowing

The second experiment demonstrates the use of windowing for trading optimization quality for runtime in the BDD-based don't-care computation flow in MVSIS. Table 2 compares the number of literals in the factored forms of the original benchmarks after sweeping ("sweep") with the number of literals after optimization, which includes SOP minimization, Boolean resubstitution, and phase-assignment. Identification of nodes with equivalent global functionality was not enabled in this experiment. The optimization was applied (1) without don't-cares, which corresponds to window 0x0 ("*mfs –w* 00"), (2) with don't-cares derived using window 2 x 2 ("*mfs –w* 22"), and (3) with don't-cares computed in the scope of the entire network, i.e. infinite window ("*mfs*"). The columns "time" lists the runtime in seconds for each of the cases. The bottom line shows the averages of the ratios of all the cases.

Table 2. Performance depending on window size.

Name	*sweep*	*mfs –w* 00		*mfs –w* 22		*mfs*	
	lits	lits	time	lits	time	lits	time
dalu	2976	2272	0.5	2250	0.8	1724	2.6
des	6101	6065	2.6	5920	4.0	5920	9.6
frg2	2010	1687	0.4	1477	0.6	1429	4.3
i10	4355	4110	0.8	3851	1.1	3703	290.3
k2	2928	2878	3.1	2878	3.6	2715	9.9
pair	2420	2187	0.4	2151	0.5	2143	7.7
c1355	992	992	0.1	992	0.2	985	156.6
c1908	1058	870	0.3	869	0.4	861	36.0
c2670	1570	1453	0.4	1370	0.4	1167	12.2
c432	335	335	0.1	299	0.3	288	1.9
c499	576	576	0.1	576	0.2	568	51.5
c5315	3531	3224	0.8	3176	1.1	3174	9.2
c7552	4750	4181	0.8	4079	1.3	3906	54.6
c880	648	625	0.1	625	0.1	625	117.4
Ave	**1.00**	**0.92**	**1.00**	**0.89**	**1.55**	**0.86**	**279.2**

Table 2 demonstrates that windowing is very efficient in trading the quality of optimization for runtime, by setting the scope for the don't-care computation. Using 2 x 2 windows gives, on average, intermediate results in terms of quality between not using don't-cares, on the one hand, and using the entire network as the context, on the other hand. The runtime of 2 x 2 windowing is only 55% longer than the runtime without don't-cares, while considering the whole network as the context increases the runtime more than two orders of magnitude.

The difference in the number of literals after running *mfs* in SIS (Table 1, column "MFS") and in MVSIS (Table 2, column "mfs/lits") is because in Table 2, sweeping of nodes with equivalent functionality was disabled. In general, minor variations in the performance of different implementations is because (a) the amount of don't-cares computed for the nodes depends on the order that the nodes are considered for optimization and (b) employing similar resource limits (timeouts in BDD computation and image computation) often leads to computations being aborted at different moments, resulting in different subsets of CDCs.

The runtime difference between *mfs* in SIS and *mfs* in MVSIS has two reasons: (a) computations in MVSIS use a general multi-valued approach [11][12], and therefore they are more complex, compared to an efficient specialized approach described in this paper for binary benchmarks,

and (b) the resource limits are currently better fine-tuned in the SIS version.

6.3 Experiment 3: BDDs vs. SAT for CDC computation

The third experiment compares the speed of don't-care computation only, using BDDs and SAT for windows of different sizes. The benchmarks in Table 3 are the largest ITC'99 benchmarks [5] (*b*-files), the largest sequential circuits from the MCNC benchmarks [23] (*s*-files), and the combinational logic extracted from the cores of the PicoJava microprocessor [16] (*pj*-files). Table 4 gives the number of inputs, outputs, and latches in the selected benchmarks.

Three window sizes were considered (1x1, 2x2, and 4x4). In each case, the runtimes in seconds of the BDD-based computation ("BDDs") and the SAT-based computation ("SAT") are reported. It was formally verified that the complete don't-cares computed in each case by BDDs and SAT using the same window are identical. The bottom line in Table 3 shows the average of the ratios in all cases.

The measurements in Table 3 are not exactly comparable due to different pruning techniques employed by the two computation flows. One pruning technique uses *window rescaling*, which reduces the scope of a window if its size exceeds a predefined limit. For example, if a 4x4 window turns out to be too large, it is automatically replaced by a 3x3 window. For BDDs, the window is rescaled if it has more than 30 leaves and 15 roots, while for SAT the window is rescaled if it contains more than 500 AND-gates after structural hashing.

Table 3. BDD vs. SAT for CDC computation.

Name	Window 1 x 1		Window 2 x 2		Window 4 x 4	
	BDDs	SAT	BDDs	SAT	BDDs	SAT
b14	1.47	0.67	3.50	0.84	12.29	1.24
b15	0.84	0.99	3.11	1.20	26.70	5.30
b17	2.97	1.33	6.69	3.21	48.59	4.37
b20	2.98	2.18	6.19	2.19	20.18	2.23
b21	3.42	2.13	6.48	2.79	18.34	2.42
b22	4.50	3.18	9.62	4.86	27.80	3.24
s15850	0.17	0.26	0.39	0.28	4.14	0.30
s35932	0.28	0.20	0.44	0.28	1.10	0.53
s38417	1.16	0.50	3.40	0.55	18.78	1.15
pj1	1.69	1.58	5.75	1.38	15.26	2.35
pj2	0.20	0.20	0.28	0.26	3.66	0.28
Ave	**1.00**	**0.80**	**1.00**	**0.46**	**1.00**	**0.14**

Table 3 indicates that the SAT-based computations are faster and scale better than the BDD-based ones. Thus, for 1x1 windows, SAT is on average 20% faster; for 2x2 windows, it is over 2 times faster while for 4x4 windows, it is over 7 times faster. This ratio increases further with the window size.

6.4 Experiment 4: The cumulative effect of improvements

Table 4 shows the results of network optimization using the SAT-based flow for the benchmarks from Table 3. These benchmarks are relatively large. As a result, BDD-based methods, *full_simplify* in SIS and *mfs* in MVSIS without windowing, cannot be applied.

Table 4. Network optimization using CDCs, windowing and SAT.

Name	In/Out/Latch	Literals in factored forms			Runtime, s	
		sweep	mfsw	script	mfsw	script
b14	32 / 54 / 245	17388	10664	7911	3.9	18.0
b15	36 / 70 / 449	16244	15056	10948	6.1	22.9
b17	37 / 97 / 1415	57311	49067	37877	35.7	104.8
b20	32 / 22 / 490	35149	21826	16813	7.6	55.0
b21	32 / 22 / 490	35908	22312	16932	9.3	51.1
b22	32 / 22 / 735	52276	33017	25174	13.5	59.8
s15850	14 / 87 / 597	7303	6350	4033	1.2	4.0
s35932	35 / 320	24408	20248	10986	4.2	16.7
s38417	28 / 106	18699	17327	13640	4.5	15.5
pj1	1769 / 1063/0	34828	30547	18076	9.5	37.0
pj2	690 / 429/0	7422	6464	3457	1.1	4.0
Ave		**1.00**	**0.79**	**0.54**	**1.00**	**4.36**

The first column of Table 4 lists the benchmark names. The second column shows the number of inputs, outputs, and latches. The next three columns contain the number of literals in the factored forms in (1) the original benchmark after sweeping ("sweep"), (2) after applying *mfs* with 2x2 windowing ("mfsw"), and (3) as part of a script ("script"). The last two columns show the runtime in seconds for the two optimization options.

The script used in this experiment was *mvsis.rugged*, which is similar to *script.rugged* from the SIS distribution, except that *mvsis.rugged* is implemented in MVSIS, except that whenever the CODC-based command *full_simplify* is used in SIS, the CDC-based *mfs* with 2x2 windows (*mfs –w* 22) and SAT are used in MVSIS.

Table 4 shows that the proposed don't-care-based optimization flow can be applied to large circuits. This is because the don't-care computation is performed in a window, and therefore is local and does not depend on the circuit size. The overall runtimes scale well with the problem size and is predictable; a rule of the thumb is for *mfs –w* 22, the computation takes about 1 second per 3000 literals in the original netlist.

7 Conclusions

This paper contributes several improvements to the optimization of logic networks using don't-cares:

- Complete don't-cares are used instead of compatible don't-cares. Abandoning compatibility does not lead to any problems in runtime while increasing the scope of the don't-cares computed.
- To ensure robust computation of don't-cares windowing is used. This technique noticeably reduces the runtime while computing a substantial subset of complete don't-cares for each node.
- A new implementation of the don't-care computation using Boolean satisfiability is used, taking advantage of the recent improvements in the performance of SAT solvers [15]. The same set of don't-cares is computed as in the corresponding BDD-based algorithm, but several times faster.

The experiments described in the paper show that the proposed improvements enhance the optimization quality and reduce the runtime. The overall effect is that the computation of internal don't-cares becomes very affordable, even for very large industrial networks.

We believe that such ideas can be applied to other Boolean logic optimization methods and will reduce the computational cost and improve optimality. As a result, these Boolean methods will become more affordable and may eventually replace the sub-optimal algebraic methods for a variety of tasks in logic synthesis.

Acknowledgements

The authors gratefully acknowledge the support of the California MICRO program and our industrial sponsors, Intel, Fujitsu, and Synplicity. The authors thank Jordi Cortadella for many helpful discussions.

References

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

[2] R. K. Brayton. Compatible observability don't-cares revisited. *Proc. IWLS '01*. pp. 121-126.

[3] N. Eén, N. Sörensson. An extensible SAT-solver. *Proc. SAT '03.* http://www.cs.chalmers.se/~een/Satzoo/An_Extensible_SAT-solver.ps.gz

[4] E. Goldberg, M.Prasad, R.K.Brayton. Using SAT for combinational equivalence checking. *Proc. DATE '01*, pp. 114 -121. http://eigold.tripod.com/

[5] ITC '99 Benchmarks http://www.cad.polito.it/tools/itc99.html

[6] F. Lu, L. Wang, K. Cheng, R. Huang. A circuit SAT solver with signal correlation guided learning. *Proc. DATE '03*, pp. 892-897.

[7] Y. Jiang, R. K. Brayton. Don't-cares and multi-valued logic network optimization. *Proc. ICCAD'00*, pp. 520-525. http://www-cad.eecs.berkeley.edu/Respep/Research/mvsis/

[8] A. Kuehlmann, V. Paruthi, F. Krohm, M. K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. CAD*, Vol. 21, No. 12, December 2002, pp. 1377-1394.

[9] J. P. Marques-Silva, K. A. Sakallah, GRASP: A search algorithm for propositional satisfiability, IEEE Trans. Comp, vol. 48, no. 5, pp. 506-521, May 1999.

[10] K. McMillan. Applying SAT methods in unbounded symbolic model checking. *Proc. CAV '02, LNCS, vol. 2404*, pp. 250-264.

[11] A. Mishchenko, R. K. Brayton. Simplification of non-deterministic multi-valued networks. *Proc. ICCAD '02*, pp. 557-562.

[12] A. Mishchenko, R. K. Brayton. A theory of non-deterministic networks. *Proc. ICCAD '03*, pp. 709-716.

[13] A. Mishchenko, X. Wang, T. Kam. A new enhanced constructive decomposition and mapping algorithm. *Proc. DAC '03*, pp. 143-147.

[14] A. Mishchenko. *EXTRA library of the DD procedures*. http://www.ee.pdx.edu/~alanmi/research/extra.htm

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

[16] SUN Microelectronics. *PicoJava Microprocessor Cores*. http://www.sun.com/microelectronics/picoJava/

[17] F. Somenzi. *BDD package "CUDD v. 2.3.0."* http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[18] MVSIS Group. *MVSIS.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[19] H. Savoj, R. K. Brayton. The use of observability and external don't-cares for the simplification of multi-level networks. *Proc. DAC' 90*. pp. 297-301.

[20] H. Savoj. Improvements in technology independent optimization of logic circuits. *Proc. IWLS '97.*

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

[22] E. Sentovich et al. SIS: A system for sequential circuit synthesis. Tech. Rep. UCB/ERI, M92/41, ERL, Dept. of EECS, UC Berkeley, 1992.

[23] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.