Scalable Logic Rewriting Using Don't Cares

Alessandro Tempia Calvino, Giovanni De Micheli
Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—Logic rewriting is a powerful optimization technique that replaces small sections of a Boolean network with better implementations. Typically, *exact synthesis* is used to compute optimum replacement on-the-fly, with possible support for Boolean *don't cares*. However, exact synthesis is computationally expensive, rendering it impractical in industrial tools. For this reason, optimum structures are typically pre-computed and stored in a database, commonly limited to 4-inputs. Nevertheless, this method does not support the use of don't cares. In this paper, we propose a technique to enable the usage of don't cares in pre-computed databases. We show how to process the database and perform Boolean matching with Boolean don't cares, with negligible run time overhead. Logic rewriting techniques are typically very effective at optimizing majority-inverter graphs (MIGs). In the experiments, we show that the usage of don't cares in logic rewriting on MIGs offers an average size improvement of 4.31% and up to 14.32% compared to state-of-the-art synthesis flow.**

I. Introduction

Multi-level logic optimization [1] is a key step in the realization of efficient digital systems. State-of-the-art logic synthesis tools initially describe a Boolean network using a technology-independent representation of simple primitives, such as the *and-inverter graph* (AIG) [2], that is optimized for size and depth. Then, they map the network to the target technology, and lastly, they further optimize the technology-dependent circuit representation.

Logic rewriting is a powerful optimization technique that iteratively rewrites small sections of a Boolean network with better implementations, typically evaluated in terms of size or depth. Many varieties of logic rewriting methods have been proposed in the literature, such as DAG-aware rewriting [3], cut rewriting [4], LUT-based rewriting [5], and mapping-based rewriting [6]. SAT-based *exact synthesis* [7] is typically used to compute optimum replacements for the network. However, exact synthesis is computationally expensive and generally limited to synthesizing networks up to 4 inputs. In industrial applications, on-the-fly computation of replacements using exact synthesis is generally run time prohibitive, even for small logic blocks. Hence, structures are typically pre-computed and saved in a database. Logic rewriting with databases employs Boolean matching [8] to retrieve implementations from the database given a Boolean function. However, while Boolean don't cares are supported by on-the-fly exact synthesis, they are not supported by logic rewriting using a pre-computed database. In the academic tools ABC [9] and Mockturtle [10], size-optimum databases for logic rewriting contain up to 4-input networks. In [11], a delay-optimum database has been constructed using exact synthesis for up to 4-input networks.

Logic rewriting is very effective at optimizing many graph representations, such as the *majority-inverter graph* (MIG) [12], composed of three-input majorities and inverters, with many applications in standard-cells flows, and majority-based emerging technologies [13], [14].

This paper presents improvements to logic rewriting by enabling the use of Boolean don't cares (DCs) with pre-computed databases. First, we present the notion of *don't care class* that is used to classify a database based on don't cares and permissible functions. This computation typically takes less than half of a second for databases containing up to 4 input functions. Then, we present a Boolean matching approach to access the database while leveraging Boolean don't cares. Finally, we propose an efficient integration of don't care computation and matching in cut-based logic rewriting.

In the experiments, we show that mapping-based rewriting with DCs reduces up to 13.21% and 0.62% on average the size compared to the state-of-the-art MIG flow over the EPFL benchmarks. Notably, in this experiment, we compare one algorithm against a flow of 3 algorithms composed of the mapping-based rewriting itself and variants of Boolean resubstitution. Moreover, we achieve even more reductions in size up to 14.32% and 4.31% average after integrating logic rewriting with DCs in the state-of-the-art flow.

II. Background

In this section, we introduce the basic notations and the necessary background related to logic networks, equivalence classes, and reconvergences.

A. Notations and Definitions

A *Boolean function* is a mapping from a k-dimentional Boolean space into a 1-dimentional one: $\{0,1\}^k \rightarrow \{0,1\}$.

A *truth table* representation of a k-input Boolean function $f : \{0,1\}^k \rightarrow \{0,1\}$ is a bit string $b = b_{l-1} \dots b_0$, i.e., a sequence of bits, of length $l = 2^k$. A bit $b_i \in \{0,1\}$ at position $0 \leq i < l$ is equal to the value taken by f under the input assignment $\vec{a} = (a_0, \dots, a_{k-1})$ where

$$2^{k-1} \cdot a_{k-1} + \dots + 2^0 \cdot a_0 = i.$$

A truth table t_1 is said to *imply*, or *cover*, another truth table t_2 if each bit of t_1 is true also in t_2. This relationship is denoted as $t_1 \leq t_2$. Similarly, t_2 is said to be *implied* by t_1, denoted as $t_2 \geq t_1$. For instance, $1000 \leq 1001$.

A *Boolean network* is modeled as a directed acyclic graph (DAG) 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). For any node n, the *fanins* of n is a set of nodes driving n, i.e. nodes that have an outgoing edge towards n. Similarly, the *fanouts* of n

is a set of nodes that are driven by node n, i.e., nodes that have an incoming edge from n. If there is a path from node a to node b, then a is in the *transitive fanin* (TFI) of b, and b is said to be in the *transitive fanout* (TFO) of a. The transitive fanin of b includes node b and the nodes in its transitive fanin, including the PIs. The *transitive fanout* of b includes b and all the nodes in its transitive fanout including the POs.

A *cut* C of a Boolean network is a pair (n, $\mathcal{K}$), where n is a node called *root*, and $\mathcal{K}$ is a set of nodes, called *leaves*, such that 1) every path from any PI to node n passes through at least one leaf and 2) for each leaf $v \in \mathcal{K}$, there is at least one path from a PI to n passing through v and not through any other leaf. The *size* of a cut is defined as the number of leaves. A cut is k-feasible if its size does not exceed k. A cut *covers* all the nodes encountered on the paths between the leaves and the root, including the root and excluding the leaves.

B. $\mathcal{NPN}$-equivalence classes

Two functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are $\mathcal{NPN}$-equivalent if there exists an inversion of the inputs $\mathcal{N}_i : (x_i \rightarrow \overline{x}_i)$, a permutation of the inputs $\mathcal{P}_i : (x_i x_j \rightarrow x_j x_i)$, and an inversion of the output $\mathcal{N}_o : (f \rightarrow \bar{f})$ such that f and g can be made Boolean equivalent [8].

For n-inputs, 2^{2^n} different Boolean functions exist. Boolean functions can be partitioned into $\mathcal{NPN}$ classes that are orders of magnitude smaller than the number of functions. In particular, n-input Boolean functions can be classified into 14, 222 and 616126 classes for $n = 3, 4, 5$, respectively.

C. Reconvergence

In a Boolean network, a path is a finite sequence of connected nodes $v_0 \rightarrow \cdots \rightarrow v_l$ where (v_i, v_{i+1}) are connected with an edge. Two paths are *reconvergent* if they start at the same node v_0 and end at the same node v_l arriving from two different fanins of v_l. Identifying reconvergent paths is crucial in logic optimization because reconvergences enable don't care optimizations. A *reconvergence-driven cut* is a type of cut constructed to include reconvergences. This type of cut is used in Boolean methods such as resubstitution [15] to leverage don't cares. A reconvergence-driven cut with multiple outputs is referred to as a *reconvergence-driven window*.

III. Extracting don't care classes

Due to controllability or observability don't cares (DCs) in a logic network, often a Boolean function f can be changed into another one, f', without affecting the intended behavior of the circuit at the primary outputs. Such a function f' is called a *permissible function*, and the functional flexibility is described by its don't care set dc. All the permissible functions of f are referred to as the *maximum set of permissible functions* (MSPF) [16].

In this section, we present a method to represent a database of structures and compute permissible functions reachable given Boolean don't cares. First, the database is classified into $\mathcal{NPN}$-equivalence classes. Then, it is processed to compute all the don't care sets that lead to a permissible function with better cost. Finally, we present our algorithm for Boolean matching.

A. Database

The database is internally represented as a compact data structure that facilitates fast Boolean matching. The database is classified into $\mathcal{NPN}$-equivalence classes to limit the number of entries (e.g., 222 for 4-input functions). Each class functionality is expressed by a representative truth table, which is computed by finding the lexicographically smallest truth table in the class. A class may list several implementations (Boolean networks), each realizing the class representative function and described by its area cost and pin-to-pin delay.

B. Don't care classes

Given a database classified into $\mathcal{NPN}$-equivalence classes, we compute minimal don't care sets that support moving from an $\mathcal{NPN}$ class into a function in a different $\mathcal{NPN}$ class. The definition of minimal is given later in the text at Definition 3. Informally, this problem can be seen as the construction of a directed graph where nodes are $\mathcal{NPN}$ classes and edges are don't care sets. This idea is similar to the work in [17], where vertices are $\mathcal{NPN}$ classes, but edges link functions that differ by one minterm. Thus, our approach differs in the size of the graph, the number of edges, and the Boolean matching technique. The graph creation is achieved by enumerating and storing don't care sets for each class.

Don't care sets are represented as truth tables. For a function f, an entry b_i in its don't care set dc is '1' if the bit in position i of f can be flipped. This information introduces flexibility in the functionality potentially leading to a better implementation.

Example 1: Let c be a 2-input cut in an *and-inverter graph* (AIG), composed two-input ANDs and inverters, with function $f = 1001$ and don't care set $dc = 0001$. The cut represents an XNOR function that needs 3 AND nodes to be implemented. The don't care set introduces flexibility to flip bit b_0 of f to obtain $f' = 1000$, which is an AND function that needs only one AIG node, improving the AIG size.

We define a *don't care class* that belongs to an $\mathcal{NPN}$ class as a set of don't cares that supports Boolean transformations into better permissible implementations. Generally, for a function f of k variables, there exist 2^{2^k} possible don't care sets. Moreover, for each don't care set dc, there are 2^p possible permissible functions, where p is the number of *minterms* in dc, i.e., the number of bits at 1 in dc. Therefore, filtering mechanisms are necessary to enable the computation and limit the search space during Boolean matching. To enable Boolean don't cares, we use two assumptions that limit the number of matching possibilities, stored don't care sets, and permissible functions.

Assumption 1. *We use the best area of implementations in the maximum set of permissible functions (MSPF) to evaluate the benefit offered by a don't care set.*

If a database contains the size optimum implementations, the best area coincides with the area optimum. This assumption prioritizes the area over other metrics for multiple reasons.

First, logic rewriting is typically area-oriented. Second, the area is usually independent of the context of the rewriting, whereas propagation delay depends on the arrival time, and it cannot be evaluated offline. Third, often better area implementation offer also better delay (especially in the context of technology-independent optimization). This assumption is used as a filter. In other words, we only store don't care sets for which there exist a function in the MSPF that offers a better implementation in terms of area cost.

Assumption 2. *For each don't care set, we select only one permissible function that minimizes the area.*

Given a don't care set for a function f, the size of the MSPF can be pretty large, offering many implementation options. Evaluating all of them during logic rewriting may significantly increase the run time without offering a considerable advantage. Hence, for each don't care set, our method stores a single permissible function that minimizes the area cost.

To further filter the number of saved don't care sets, we employ the definition of *dominance*.

Definition 3. *For a function f, a don't care set t_1 is said to dominate a don't care set t_2 if $t_1 \leq t_2$ and the best area cost in the MSPF of f for t_1 is not worse than the one for t_2. The set t_2 is said to be dominated by t_1.*

Informally, we refer to a non-dominated don't care set as *minimal*. Non-minimal sets are redundant to store since they are implied and don't offer better implementations.

Algorithm 1 shows the procedure to compute don't care (DC) classes and permissible functions. The algorithm takes a database classified into $\mathcal{NPN}$-equivalence classes and the maximum number of input variables in the database, which is typically 4, as inputs. The procedure starts by iterating through each $\mathcal{NPN}$ class, assigning to f_i the class representative function. At line 4, the DC class for f_i is set to empty. Then, from line 5 to 14, the procedure iterates to all the other classes f_j with a better area cost than f_i. At this step, all the possible don't care sets that link f_i and f_j are computed. To achieve that, all the negations and permutations configurations of f_j are enumerated to capture all the functions g in the $\mathcal{NPN}$ class of f_j. Along with g, the enumeration generates the input permutation vector *perm* and input/output negation vector *neg* that store the information to transform g into f_j. The don't care set dc, which links f_i and g, is computed using the Boolean difference at line 10. Then, dc is checked for dominance following Definition 3. If the don't care set is currently minimal, previously computed dominated sets are removed, and the new one is inserted in dc_class. The set is inserted together with the input permutations and input/output negations to apply to f_i under don't care set dc to obtain f_j. Finally, dc_class is sorted by implementation area in ascending order. If a database is *partial*, i.e., it does not contain implementations for each $\mathcal{NPN}$ class (not *complete*), the best area of missing classes is assumed to be infinite.

Example 2: Let us consider the $\mathcal{NPN}$-4 class $f_i = 033\text{c}$, with the bit string represented in hexadecimal format, having best area 4. First, let us consider the class $f_j = 0000$ ($\perp$) that represents constants, of cost 0. The two possible don't care sets that link the two classes are $dc_1 = 033\text{c}$ and $dc_2 = \text{fcc3}$, since $f_i \wedge \neg dc_1 = \perp$ and $f_i \vee dc_2 = \top$. The two DC sets are minimal and are found by taking the Boolean difference between f_i and f_j for dc_1, and f_i and $\bar{f}_j$ for dc_2.

Example 3: Let us consider the previous class $f_i = 033\text{c}$ and a class $f_j = 003\text{c} = \bar{x}_3 \wedge ((x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2))$ of cost 3. Along with the trivial DC set $dc_3 = 0300$, there exist another one $dc_4 = 000\text{c}$ with permutations $\mathcal{P}_i : (x_0 x_1 x_2 x_3 \rightarrow x_0 x_3 x_1 x_2)$ and no negation. If we flip bits using the don't care we obtain $g = f_i \wedge \neg dc_4 = 0330 = \bar{x}_1 \wedge ((x_2 \wedge \bar{x}_3) \vee (\bar{x}_2 \wedge x_3))$, which is a permutation $\mathcal{P}_i^{-1}$ of class f_j.

Regarding the scalability of Algorithm 1, the computation of don't care classes takes less than half a second for databases up to 4-inputs and very low memory. For larger databases, this method would experience limitations due to the double exponential increase in the number of Boolean functions. Hence, it may necessitate restricting the computation to only *practical classes* for functions of more than 4 variables. Practical classes are a subset of $\mathcal{NPN}$ classes that are highly observed in common designs and tend to be much less than $\mathcal{NPN}$ classes. For instance, common practical functions are the fully- and partially-decomposable functions. In [5], the authors found only 286 unique $\mathcal{NPN}$ classes for 6-input functions when mining the EPFL benchmarks [18].

Algorithm 1: Extracting don't care classes

```
1  Input  : Database data, Number of variables k
2  Output: Don't care classes dc_class
3  foreach function f_i in NPN(k) do
4      dc_class(f_i) ← ∅;
5      foreach function f_j in NPN(k) do
6          s_j ← best_area(f_j, data);
7          if s_j ≥ best_area(f_i, data) then
8              break;
9          foreach {g, perm, neg} in npn_enumeration(f_j) do
10             dc ← f_i ⊕ g;
11             if is_dominated(dc, dc_class(f_i), s_j) then
12                 continue;
13             remove_dominated(dc, dc_class(f_i), s_j);
14             dc_class(f_i).add(dc, f_j, perm, neg);
15     sort_dc_class(dc_class(f_i));
16 return dc_class;
```

C. Boolean matching using Boolean don't cares

Given a Boolean function and its don't care set, as truth tables, Boolean matching returns a list of implementations in the MSPF class with minimal cost computed by Algorithm 1.

The Boolean matching procedure is shown in Algorithm 2. Compared to standard Boolean matching over $\mathcal{NPN}$ classes, our algorithm adds the steps from line 4 to 10. The algorithm takes a function f, its don't care set dc, the database, and the don't care classes as inputs. First, function f is canonized by computing the lexicographically smallest truth table in its $\mathcal{NPN}$ class using fast enumeration [19]. The class

Algorithm 2: Boolean matching with don't cares

```
1  Input : Function f, Don't care set dc, Database data, Don't
       care classes dc_class
2  Output: {Matches M, Permutations perm, Negations neg}
3  {f_c, perm, neg} ← npn_canonization(f);
4  dc ← apply_permutations(dc, perm);
5  foreach {t, f_i, p, n} in dc_class(f_c) do
6      if t ≤ dc then
7          perm ← apply_permutations(perm, p);
8          neg ← apply_permutations(neg, p);
9          neg ← neg ⊕ n;
10         return {data(f_i), perm, neg};
11 return {data(f_c), perm, neg};
```

representative f_c is returned along with its permutation and negation vectors. Then, the permutations are applied to the don't care set such that its bits respect the new permutation in f_c (line 4). Input and output negations are not applied since they don't affect the don't cares. Then, the don't care class of f_c is accessed to retrieve a better implementation. Each entry is accessed in order, from the smallest area implementations to the largest. Each entry is composed of its don't care set t, its $\mathcal{NPN}$ class representative f_i, the permutation vector to apply p, and the negation vector to apply n. The entry is a permissible function if $t \leq dc$, i.e., the don't care set t implies dc. As soon as this is true, the algorithm returns the implementations for the best permissible function. Before returning, the previously computed permutation and negation vectors are adjusted to match the new $\mathcal{NPN}$ class and its representative (from line 7 to 9). This is required to match the functionality of the new class, as shown by Example 3. If no entry matching the given don't care set is found, the algorithm returns the implementations from f_c.

IV. Logic rewriting with don't cares

This section describes the integration of Boolean matching with don't cares into classical logic rewriting algorithms. The classification of the database and the computation of the don't care classes presented in Section III are independent of logic rewriting, are computed offline, and are not addressed in this section. Algorithm 3 reflects the implementation of DAG-aware rewriting [3] with an extension to support Boolean don't cares. Similarly, this method can be integrated into alternative rewriting or mapping techniques.

Algorithm 3 tries to replace small sections of the network defined by cuts with a better implementation. The algorithm processes the nodes in topological order and searches for the best replacements that locally improve the area. Compared to the standard rewriting, Algorithm 3 adds the steps between line 4 and 13. For each gate g, the k-feasible cuts rooted in n are computed using a cut enumeration procedure [20]. Then, a reconvergence-driven window of l inputs, having $l > k$, is extracted around gate g. The window can be single-output (a cut) in case only controllability don't cares are used, or multi-output, expanded over the transitive fanout of g if observability don't cares are used. Then, the window is

Algorithm 3: Logic rewriting with Boolean don't cares

```
1  Input : Network N, Database data, Don't care classes
       dc_class, Cut size k, Cut size l
2  foreach gate g ∈ N in topological order do
3      C ← compute_cuts(N, g, k);
4      W ← reconvergence_driven_window(N, g, l);
5      S ← simulate_window(W);
6      R ← Λ;
7      best_gain ← 0;
8      foreach cut c ∈ C do
9          f ← truth_table(c);
10         dc ← ⊥;
11         if c ⊂ W then
12             dc ← compute_dont_cares(c, W, S);
13         {M, p, n} ← bool_matching(f, dc, data, dc_class);
14         gain ← evaluate_gain(N, g, c, M, p, n);
15         if gain > best_gain then
16             R ← candidate_replacement(N, g, c, M, p, n);
17             best_gain ← gain;
18     if best_gain > 0 then
19         replace(N, g, R);
```

simulated to extract complete truth tables for each covered node. The truth tables are on l variables and computed with respect to the inputs of the window. Next, for each cut, the best matches are computed and evaluated. First, for a cut c, its function f is extracted. Then, if the cut fits in the window, i.e., all its leaves are contained in the window, its don't care set is computed from the window. If it doesn't, don't cares are ignored for the cut. Alternatively, a window may be computed to guarantee containment at the cut at the cost of additional run time. However, experimental results have shown that many cuts tend to be included in the window. Next, Boolean matching is performed according to Algorithm 2, and candidate replacements are evaluated. Finally, the candidate with the best area gain is used as a replacement.

The most runtime-intensive process of logic rewriting with Boolean don't cares is the computation of DCs. Boolean DCs for a cut are extracted starting from a window of logic that includes it. First, the window is simulated over its input to collect complete simulation patterns. Given simulation patterns, controllability don't cares (CDCs) are computed by checking which combinations of patterns appear at the leaves of the cut. Not-appearing patterns are CDCs for the cut. This process is called *projection* of the don't cares and its complexity is exponential in the number of leaves of the window. Observability don't cares (ODCs) at a gate g are instead computed by checking for which patterns the Boolean difference between the function of the gate g and its inverse is observable. Let $S(g)$ be the simulation pattern in the window for gate g and let O be the set of output of the windows. First, the window is re-simulated fixing the simulation of gate g to $\neg S(g)$ and obtaining the simulation patterns S' at the outputs. The ODCs are then computed as follows:

$$ODC_g = \neg \bigvee_{o \in O} S(o) \oplus S'(o).$$

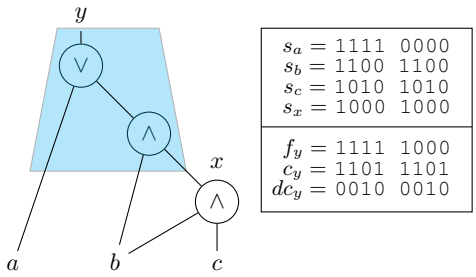

Fig. 1: Example of projection of CDCs on a cut

The ODCs need to be projected over the cut leaves as for CDCs. Despite being the run time bottleneck of logic rewriting, projections for multiple cuts can be computed in parallel, notably reducing the impact over run time.

Example 4: Figure 1 shows an example of CDC computation and projection on an AIG. In this example, the window covers the entire circuit. First, the window is simulated, obtaining the patterns s_a, s_b, s_c, and s_x. Patterns s_a, s_b, and s_c are the input patterns of the window and are used to simulate all the input combinations (each bit b_i in every input pattern represents a combination). The section in blue represents a cut to optimize with function $f_y = a' \vee (b' \wedge x')$, computed considering a', b', and x' as inputs[1]. The care set c_y, of node y, i.e., the complement of the don't care set, is computed starting from the simulation patterns. Each combination of input patterns of the window s_a, s_b, and s_x is used to set bits in c_y. For instance, taking b_0, the input pattern of the cut is 000. Hence, b_0 of c_y is set to one. Taking b_6, the pattern is 110 setting b_6 of c_y to 1. At the end of this process, the care set has a "1" for occurring patterns. Finally, $dc_y = \neg c_y$ is computed to express the CDCs of the cut. Consequently, f_y can be simplified into 11111010, which corresponds to $a' \vee x'$. This transformation removes a node while preserving the correct functionality.

V. Experiments

In this section, we present experimental results on logic rewriting with Boolean don't cares. For our experiments, we use the EPFL combinational benchmark suite [18] containing several circuits provided as *and-inverter graphs* (AIGs).

The construction of the database, the generation of the don't care classes, and Boolean matching with don't cares have been implemented in C++17 and used to extend the algorithms in the open-source logic synthesis framework *Mockturtle*[2]. The database of structures used in the experiments is available in the library and contains 4-input size-optimum implementations obtained using exact synthesis. Up to 10 structures are available for each $\mathcal{NPN}$ class. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in *ABC*[3].

[1]Variables a', b', and x' are "virtual" variables input of the cut. The link between a and a' is not visible by the cut.

[2]Available at: https://github.com/lsils/mockturtle

[3]Available at: https://github.com/berkeley-abc/abc

In this experiment, we test logic rewriting with don't cares to optimize *majority-inverter graphs* (MIGs) [12], which have many applications in standard-cells flows [12], and majority-based emerging technologies [13], [14]. We compare our approach against the state-of-the-art flow published in [21], which is based on the most effective MIG methods known. The baseline flow carries the optimization by running the command *compress2rs* in ABC, 3 times the mapping-based logic rewriting algorithm in [6], the MIG Boolean resubstitution in [13] until no more improvement, and the improved MIG resubstitution presented in the paper itself [21]. These results have been reproduced on our machine.

In our implementation, logic rewriting operates on a database of 4-input size-optimum MIG implementations, the same one used in [6]. The classification of the database and the computation of don't care classes took less than half a second on our machine. We extended the implementation of two logic rewriting algorithms to support don't cares. In particular, we improved the mapping-based logic rewriting algorithm in [6], referred to as *map*, and the DAG-aware rewriting algorithm in [3], referred to as *rw*. Both algorithms include the don't care computation as shown in Algorithm 3 for controllability don't cares. In the experiments, we don't use observability don't cares for two reasons: 1) ODCs are generally not compatible, i.e., not safe to use in parallel optimization (like *map* does) [1]. Hence, additional run time is required to compute compatible ODCs (CODCs); 2) experimental results using ODCs (and CODCs) in logic rewriting have not shown significant benefits in quality. In our implementation, don't care projections have not been parallelized.

Table I shows the experimental results. To test our approach, we implemented three flows with increasing optimization effort. All three flows are applied to initial results obtained by executing the optimization script *compress2rs* in ABC, like for the state-of-the-art flow. Our first flow, named "map with DCs", consists of 3 iterations of *map* with CDCs computed from 12-input cuts. Our second flow, named "map + rw with DCs", adds to flow one 3 iterations of *rw* with CDCs computed from 8-input cuts. Finally, flow three, named "MIG flow with DCs", adds Boolean resubstitution [21] to flow two.

Our first flow reduces the size up to 13.21% and 0.62% on average compared to the state-of-the-art. This is a major result considering that the comparison is between a single command and a flow. Our flow includes the column T_{BM}, which reports the total time taken by Boolean matching with don't cares. The matching time is a small fraction of the total time, which is mainly dominated by the computation and projection of CDCs. Our second flow further reduces the number of MIG nodes improving up to 14.06% and 2.17% on average the state-of-the-art. Almost every result is already significantly better before employing Boolean resubstitution, which is the standard algorithm to leverage Boolean don't cares. Notably, logic rewriting is very effective at optimizing arithmetic benchmarks (the first 10 benchmarks). Finally, the third flow uses also Boolean resubstitution to obtain superior results for every benchmark reducing the size up to 14.32%

TABLE I: Comparison between state-of-the-art MIG results and multiple MIG flows using logic rewriting with don't cares.

Benchmark	**Flow in [21]**		***Map* with DCs**				***Map* + *rw* with DCs**			**Flow with DCs**		
	Size	Time (s)	Size	Red. (%)	T_{BM} (s)	Time (s)	Size	Red. (%)	Time (s)	Size	Red. (%)	Time (s)
adder	384	0.18	384	0.00%	0.02	0.16	384	0.00%	0.20	384	0.00%	0.22
bar	2588	0.82	2597	-0.35%	0.06	0.73	2445	5.53%	1.68	2433	5.99%	1.72
div	12532	4.54	12551	-0.15%	0.40	6.58	12498	0.27%	12.27	12462	0.56%	16.30
hyp	124177	58.73	115856	6.70%	3.41	54.10	115628	6.88%	91.12	115541	6.95%	118.51
log2	23109	36.22	22714	1.71%	0.49	12.59	22430	2.94%	24.69	22010	4.76%	45.64
max	2210	0.99	2202	0.36%	0.06	1.08	2191	0.86%	2.28	2190	0.90%	2.63
multiplier	18440	6.80	17474	5.24%	0.47	7.10	17155	6.97%	10.08	17112	7.20%	12.65
sin	3967	4.20	4005	-0.96%	0.13	2.50	3929	0.96%	5.69	3870	2.45%	8.55
sqrt	12423	10.38	12450	-0.22%	0.34	7.06	12388	0.28%	10.60	12357	0.53%	16.08
square	9498	2.72	8243	13.21%	0.23	3.30	8163	14.06%	4.59	8138	14.32%	5.33
arbiter	6719	4.34	6996	-4.12%	0.17	5.27	6869	-2.23%	7.70	6711	0.12%	9.81
cavlc	533	2.60	525	1.50%	0.01	0.09	517	3.00%	0.17	492	7.69%	1.72
ctrl	79	0.62	84	-6.33%	0.00	0.01	81	-2.53%	0.02	74	6.33%	0.30
dec	304	0.23	304	0.00%	0.00	0.03	304	0.00%	0.03	304	0.00%	0.06
i2c	932	0.37	898	3.65%	0.02	0.18	893	4.18%	0.37	871	6.55%	0.49
int2float	181	0.24	180	0.55%	0.00	0.03	178	1.66%	0.04	172	4.97%	0.11
mem_ctrl	34777	17.18	35218	-1.27%	0.82	14.59	34727	0.14%	33.91	32097	7.71%	43.75
priority	431	0.30	426	1.16%	0.01	0.15	420	2.55%	0.29	406	5.80%	0.35
router	151	0.17	155	-2.65%	0.00	0.04	154	-1.99%	0.08	147	2.65%	0.11
voter	4561	1.74	4819	-5.66%	0.12	1.95	4564	-0.07%	3.56	4555	0.13%	4.45
Average				0.62%				2.17%			4.31%	

and 4.31% on average.

Furthermore, we tested this approach on AIG optimization. While rewriting with DCs helps reduce the number of AIG nodes, the improvement is less significant compared to MIGs since AIG-resynthesis methods are much more mature. In particular, the area reduction compared to standard rewriting [3] is up to 6.8% and 0.42% on average over the EPFL benchmarks, previously optimized using the script *compress2rs* in ABC.

VI. Conclusion

In this paper, we proposed a scalable technique to enable Boolean don't cares in logic rewriting with pre-computed databases. Typically, the don't cares can be leveraged only when using exact synthesis on-the-fly, which is runtime-prohibitive for industrial synthesis tools. We presented the notion of *don't care class*, which is used to classify a database on minimal don't care sets and permissible functions. Then, we proposed a method for Boolean matching with don't cares and an efficient integration in a logic rewriting algorithm. The experiments showed a significant reduction in size in MIG optimization. Compared to the state-of-the-art MIG flow, enabling don't cares in rewriting achieves an average size improvement of 4.31% and up to 14.32%.

Acknowledgments

This research was supported by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981, and Synopsys Inc.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. CAD*, 2002.

[3] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *DAC*, 2006.

[4] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *DATE*, Mar 2019.

[5] W. Haaswijk, M. Soeken, L. Amarù, P. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. ASP-DAC*, 2017.

[6] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *ASP-DAC*, 2022.

[7] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. CAD*, 2020.

[8] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, July 1997.

[9] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, 2010.

[10] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, and G. D. Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. arXiv:1805.05121v3, 2022.

[11] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *Proc. ICCAD*, 2017.

[12] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. CAD*, 2016.

[13] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. D. Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. NANOARCH*, 2018.

[14] G. Meuli, V. Possani, R. Singh, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, P. Vuillod, L. Amaru, S. Chase, J. Kawa, and G. D. Micheli, "Majority-based design flow for AQFP superconducting family," *DATE*, p. 6, 2022.

[15] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, 2006.

[16] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method-design of logic networks based on permissible functions," *Trans. on Computers*, vol. 38, no. 10, 1989.

[17] F. Mailhot and G. De Micheli, "Technology mapping using boolean matching and don't care sets," in *Proc. EDAC*, 1990.

[18] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.

[19] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast boolean matching based on npn classification," in *International Conference on Field-Programmable Technology*, 2013.

[20] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.

[21] S.-Y. Lee and G. De Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *Trans. CAD*, 2023.