An Efficient Algorithm for Local Don't Care Sets Calculation

Shih-Chieh Chang, **Malgorzata Marek-Sadowska and Kwang-Ting Cheng**

Synopsys Inc. **Electrical and Computer Engineerimg Department**

University of California at Santa Barbara

Abstract

Local don't cares of an internal node expressed in terms of its immediate inputs are usually of interest. One can directly apply any two-level minimizer on the on-set and the local don't cares set to simplify an internal node. In this paper, we propose a memory efficient technique to calculate local don't cares of internal nodes in a combinational circuit.

Our technique of calculating local don't cares makes use of automatic test pattern generation (ATPG) approach which allows us to identify quickly whether a cube in the local space is a don't care or not. Unlike other approaches which construct an intermediate form of don't cares in terms of the primary inputs, our technique directly computes the don't care cubes in the local space. This gives us a significant advantage over the previous approaches in memory usage. Experimental results on MCNC benchmarks are very encouraging.

1 Introduction

Don't care sets represent the degree of freedom in transforming a digital circuit into another equivalent one. Logic synthesis algorithms use don't cares to improve various circuit's aspects such as area, timing, testability or power dissipation. In particular, during a multilevel Boolean network optimization, local don't cares associated with internal nodes are computed over and over again. In this paper, we propose a memory efficient technique to calculate local don't cares for combinational circuits. For example, our new technique requires only 4 Mbyte of memory to calculate don't cares for C7255 in ISCAS benchmarks and after simplification, the resulting literal count is reduced from 3269 to 2980.

Local don't cares of an internal node expressed in terms of the immediate inputs are usually of interest. One can apply any two-level logic minimizer on the on set and local don't care sets to simplify an internal node. The local don't cares contain the information of a circuit's structure and are composed of observability and satisfiability don't cares. It was shown that the observability don't cares can be calculated exactly by either flattening the network [1] or by using the chain rule [3]. Both methods require enormous amount of cpu time. In addition, when a modification takes place, the observability don't cares of the modified circuit have to be recomputed. As a result, techniques have been proposed to compute sub-sets of observability don't cares. One such effort has lead to the study of compatible observability don't cares. The compatible observability don't cares are less time-consuming to compute and their recalculation is not necessary when the circuit is modified. For example, in [13] a technique to calculate the compatible observability don't cares through effective use of image computation techniques has been proposed.

The techniques [11] [13] of computing the compatible observability don't cares still have disadvantages. First, for a large circuit, these approaches, which are Binary Decision Diagram(BDD) based, demand a lot of memory. This is because the intermediate results of compatible observability don't cares computations need to be expressed in terms of primary inputs. Therefore, in the worst case, the memory requirement may be $O(2^n)$ (n is the number of primary inputs). Once the memory is used up, partial don't cares or intermediate results cannot be retained for optimization. Secondly, the techniques of calculating compatible observability don't cares are very inefficient when only a part of the circuit is to be optimized while the rest is to remain intact. For example, in applications such as hierarchical design, design reuse, engineering change, etc., some parts of a circuit may have already been optimized, mapped or even placed and routed, therefore should not be modified. But current approaches to determine compatible observability don't cares of internal nodes require that the don't cares of all nodes in the transitive fanout are calculated. Therefore, cpu time is wasted when computing don't cares of those nodes which are not to be modified.

One simple way to tackle the above disadvantages is to partition a circuit. For example in [15] a window around a node in question is considered. The size of the window is determined such that the size of an intermediate BDD does not exceed certain bounds. When calculating local don't cares for the target node, the nodes outside the window are not taken into account.

Here we propose a memory efficient and time consistent approach to compute the local don't cares. A distinct feature of our approach is that when calculating the local don't cares of a particular node, don't cares of any other node need not be computed. The don't cares are determined through the use of automatic test pattern generation (ATPG) approach which allows us to identify quickly whether a cube in the local space is a don't care or not. Let us discuss the intuition how local don't cares can be computed through ATPG approach. For example, in Fig. 1, $a\bar{b}$ is a local don't care of gate g_5. This is because when a=1 and b=0, g_2 becomes 1. Since g_2 is 1, the primary output O1 is always 0 which blocks the observability of gate g_5. Unlike other approaches which construct an intermediate form of don't cares in terms of the primary inputs, our technique directly computes the don't care cubes in the local space. This gives us a significant advantage over the previous approaches in memory usage. We have applied our technique to all MCNC and ISCAS benchmarks and the results are very encouraging.

The rest of this paper is organized as follows. Section 2 shows our approach. Sections 3 and 4 present experimental results and give conclusions.

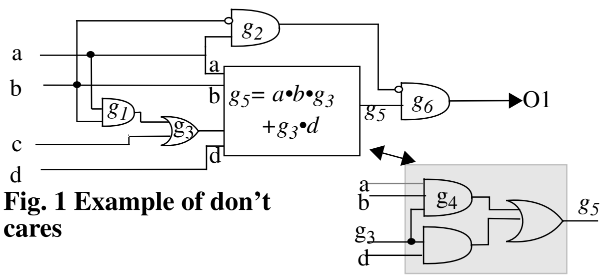


Fig. 1 Example of don't cares

2 An efficient local don't care computation algorithm based on implication

Intuition behind the relation between local don't cares and ATPG is as follows. In a circuit composed of AND and OR gates, if a wire is stuck-at fault untestable, the wire is redundant and can be removed from the circuit. Consider the example (from [5]) shown in Fig. 2. The stuck-at-1 fault at wire g_5->g_9 (dotted in the figure) is untestable so it can be deleted. Another view of the fact that stuck-at-1 at g_5->g_9 is untestable is that (g_5, g_8, f)=*(0,1,1)* constitutes a local don't care for the gate g_9. Therefore, from the fact that a wire is stuck-at fault untestable, we can obtain certain information about the local don't cares for an internal gate. Section 2.1 reviews some commonly used ATPG definitions. Section 2.2 shows our main result for calculating local don't cares. Section 2.32.3 describes a search structure for the don't cares. Section 2.4 analyzes the time complexity of our algorithm and a discussion on optimality is presented in section 4.5.

2.1 Redundancy identification

In this section, we review the procedure [5] that identifies redundancies (stuck-at faults) using the concept of mandatory assignments.The *absolute dominators* (dominators) [9] of a wire *W* is a set of gates *G* such that all paths from the wire *W* to any primary output have to pass through all gates in *G*. The *mandatory assignments* are the value assignments required for a test to exist and must be satisfied by any test vectors. Given a fault *f*, we compute the set of mandatory assignments [5] SMA(f). All mandatory assignments can be computed via implication (9 value implication) [9] [14] and recursive learning [10]. If the mandatory assignments of a fault cannot be consistently justified, the fault is redundant.

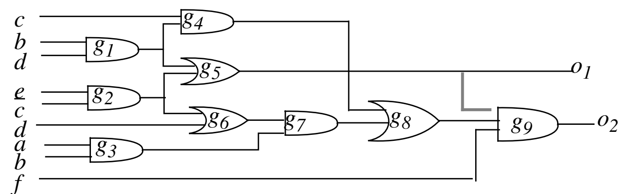


Fig. 2 Examples of testing wire stuck-at fault

2.2 Identification of the local don't cares using implications

Now, we show a technique to find local don't cares for an internal node *f: B^r-> B*. We used an ATPG based approach to determine whether a cube in the local space B^r is a don't care or not. Theoretically, a complete ATPG based approach may be very expensive in some cases. For this reason we adopt a compromising heuristic [7] described in the previous section. This compromising heuristic uses the idea of mandatory assignment to check redundancy. Since a complete ATPG method is not used, our technique of computing don't cares may lose optimality. However, a better quality result can be obtained if a more sophisticated implication approach for example [10] is applied. In the subsequent context, we will discuss cases where we lose optimality when applying our heuristic. Note that the techniques of computing compatible observability don't cares which are the sub-sets of observability don't cares also lose optimality. The following theorem describes the main concept of our technique.

Theorem: Consider an internal node n_i computing a function *f(V): B^r-> B*, and a cube $c0 \in B^r$. A *local_cube* test for a cube *c0* is to check whether there exists a test vector that can both satisfy *V=c0* and propagate an (imaginary) fault from n_i to any primary output. If there is no test vector for a *local_-cube* test for *c0*, the cube *c0* is a local don't care for the node n_i.

Proof: omitted

The intuition behind this theorem is as follows. If a cube is a local don't care, this cube must be contained in the satisfiability or observability don't cares. The activation condition V=*c0* checks whether *c0* is in the satisfiability don't care set and the condition of propagating the fault effect checks whether *c0* is in the observability don't care set.

For example, in Fig. 1, since there is no test vector that can generate (a, b, g_3) =(1,1,0), the cube $a*b*\bar{g}_3$ is a local don't care for g_5. In addition, all the test vectors that generate (a, b)

= (1, 0) cannot propagate the fault effect from the g_5 to any primary output so $a*\overline{b}$ is also a don't care for g_5.

Corollary: Let n_i be an internal node computing a function $f(V): B^r \text{-}> B$. If during the procedure of finding *local_cube* test for $c0 \in B^r$ the mandatory assignments cannot be consistently justified, the cube $c0$ is a local don't care for node n_i.

For example, in Fig. 1, to perform $(a, b, g_3)=(1,1,0)$ local_cube test, we have mandatory assignments $a=1$, $b=1$, $g_3=0$, $g_2=0$(a side input to the dominator g_6), $g_1=1(a=1$ and $b=1)$. Since $g_1=1$, we have $g_3=1$ which conflicts with the original assignment $g_3=0$. Therefore, we conclude that $(a, b, g_3) =(1,1,0)$ is a local don't care for g_5. For another $(a, b) = (1, 0)$ local_cube test, we have a conflicting mandatory assignment on g_2, because $g_2=0$ from the side input to the dominator g_6 and $g_2=1$ from $a=1$and $b=0$. Therefore, $(a, b) = (1, 0)$ is a local don't care for g_5.

The above theorem and corollary show that we can identify whether a cube $c0$ is a don't care by doing a local_cube test. A straighforward technique of calculating all local don't cares for $f: B^r \text{-}> B$ can be developed by applying the local_cube test for all the minterms in B^r. Therefore, for a given node, this straightforward technique has complexity of O(2^r * complexity(one cube_test)) where r is the number of immediate inputs. Note that any algorithm that computes local don't cares for a node with r immediate inputs has complexity of at least $O(2^r)$. It is because the representation of don't cares itself may be $O(2^r)$ in the worst case. In the following, we discuss an organized search for local don't cares. Though in the worst case, the algorithm is still $O(2^r)$, the efficiency can be improved a lot.

2.3 The organized search and essential cubes

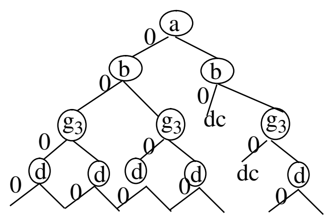
The basic idea of this organized search makes use of the fact that when a cube is a don't care, any minterm contained in it is also a don't care. Therefore, instead of checking whether the individual minterms $x_1x_2,...x_{r-1}x_r$ and $x_1x_2,...x_{r-1}\overline{x}_r$ are don't cares, we check whether $x_1x_2,...x_{r-1}$ is a don't care cube first. If $x_1x_2,...x_{r-1}$ is a don't care then, we can conclude that $x_1x_2,...x_{r-1}x_r$ and $x_1x_2,...x_{r-1}\overline{x}_r$ are both don't cares and need not to be tested. Our algorithm is shown in Fig. 3a. The algorithm assigns an order to the input variables first. Then, it starts from the first variable and performs a depth first search. Once a cube is detected to be a don't care, no further search in this branch is necessary.

For example, in Fig. 1, we will find the local don't cares for g_5. Assume the order of g_5's inputs is $a<b<g_3<d$. The search tree is shown in Fig. 3b. When the don't care for $(a, b) =(1, 0)$ is recognized, no further checking on this branch is necessary. We also like to mention that all the information (mandatory assignments) calculated at one level can be further used in the next level. Therefore, we can perform an incremental updating of the mandatory assignments from one level to the next level.

The algorithm described in Fig. 3a however, in the worst case still has the complexity of O(2^r* N)· where r is the number of immediate inputs for the target node and N is the time for each *local_cube* test. When r is large, the algorithm may not be feasible. In the following, we discuss the method that only checks the essential cubes.

Find_local_dont_care(ni)
  {dc =∅.   Order = Assign_order(faninof (ni));
    check_dont_care(Order, 0, dc, 1);
    return (dc)   }
check_dont_care(Order, current_var, dc, oldcube) {
  newVar = Order[current_var];   newcube = oldcube ∩ $\overline{\text{newVar}}$;
  if (cube_is_dont_care(newcube)) { dc = dc ∪ newcube; return; }
  else check_dont_care(Order, current_var+1, dc, newcube);
  newcube = oldcube ∩ newVar;
  if (cube_is_dont_care(newcube)) { dc = dc ∪ newcube; return; }
  else check_dont_care(Order, current_var+1, dc, newcube);
}

(a) The search algorithm



(b)Graphical interpretation

Fig. 3 algorithm of organized search and an example

The idea of essential cube search is to identify those don't cares which are of interest for optimization. When minimizing the literal count, one simple heuristic is to consider those cubes that are one Hamming distance away from the on-set cubes in the current expression. For example, in Fig. 1, the node g_5 computes a Boolean function $a*b*g_1+g_1*d$. The one-distance-away cubes from $a*b*g_1$ are $\{\overline{a}*b*g_1,\ a*\overline{b}*g_1,\ a*b*\overline{g}_1\}$ and from g_1*d are $\{\overline{g}_1*d,\ g_1*\overline{d}\}$. Each time when a one-distance-away cube is found, the literal count for the node is reduced by one.

Suppose that an internal node has been decomposed into a two-level network of AND and OR gates. Checking don't cares for one-distance-away cubes is equivalent to checking if the wires (inputs to the two-level AND/OR circuit) are stuck-at fault testable. For example, in Fig. 1, if the one-distance-away cube $\overline{a}*b*g_1$ is a don't care, then, the wire $a\text{-}>g_4$ (dotted in the figure) is stuck-at-1 untestable. Therefore, these one-distance-away cubes can be easily recognized by performing redundancy check for the stuck-at faults at the corresponding wires. Essential cubes are very useful in some cases. For example, if there is an AND or an OR gate with large number of fanins, the essential cube test is very efficient.

2.4 Discussion on optimality

If a complete ATPG approach is used, we can check exactly whether a cube is a local don't care. Due to the exponential nature of a complete ATPG for some cases, we adopt a compromising approach which is based on the idea of mandatory assignments. In this compromising approach, only simple implications are used during the local_cube test. When there is a violation of mandatory assignments, we conclude that the tested cube is a don't care. In the following, we discuss an example when the simple implication technique loses optimality.

In the Fig. 4, $(g_1, g_2, g_3) = (0, 1, 1)$ never appears in the input of g_4 so $(g_1, g_2, g_3) = (0, 1, 1)$ is a local don't care for g_4. However, with the simple implication approach, we cannot detect that $(g_1, g_2, g_3) = (0, 1, 1)$ is a don't care. During the process, no further implications can be determined for the primary inputs *a* or *b* when $g_1=0$ alone. The same situation happens when g_2=1 and $g_3=1$. Therefore, in the simple implication technique, we cannot conclude that (g_1, g_2, g_3) = (0, 1, 1) is a don't care. According to our experiments, we feel that when there are many XOR or XNOR gates in a circuit, the simple implication technique may not perform well. As we will show in our experimental results, for the ALU type circuits like alu2 and alu4, which contain many XOR, XNOR gates, the simple implication technique does not work as well as the CODC [13] technique. The results for these examples can be improved if a more sophisticated implication approach [10] is used. Here, we only briefly discuss how [10] works using again the example in Fig. 4. After simple implication, first, the primary input gate *a* is assumed to be 0. From $a=0$ and $g_2=1$, we have $b=1$. Since $a=0, b=1$ and $g_3=1$ are in conflict so we conclude that *a* must be 1. However, the implication which starts from $a=1$ still leads to a conflict. Therefore, $(g_1, g_2, g_3) = (0, 1, 1)$ is a don't care. This learning technique [10] is more powerful than the simple implication technique but requires more CPU time.

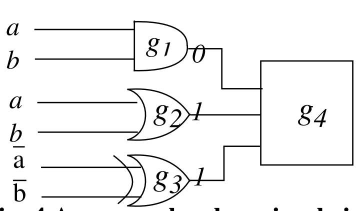


Fig. 4 An example when simple implications lose optimality

3 Experimental results

In this section, we present experimental results for combinational benchmark circuits. We have implemented the algorithm shown in FIG. 5 on DEC 5000. In the algorithm, when the number of direct inputs for an internal node is greater than 11, we check only the essential cubes.

```
For each node n_i in a circuit {
    update_dominators();
    dc = find_local_dont_care(n_i);
    simplify_node(function(n_i), dc);    }
```

FIG. 5 The local don't care computation

(a) scirpt.rugged	(b) script.local_cube
sweep; eliminate -1 simplify -m nocomp eliminate -1	sweep; eliminate -1 simplify -m nocomp eliminate -1
sweep; eliminate 5 simplify -m nocomp resub -a	sweep; eliminate 5 simplify -m nocomp resub -a
fx resub -a; sweep	fx resub -a; sweep
eliminate -1; sweep	eliminate -1; sweep
full_simplify	**local_cube_simplify**

Fig. 6 SIS script.rugged and our algorithm imbedded in it.

We have compared the result of our algorithm with the result in [13] which calculates CODC in BDD form and applies image computation to find the local don't cares. Note that the don't cares obtained by both algorithms are not exact. Using the don't cares obtained by both methods, we can simplify the literal count for nodes in a circuit.

We have performed experiments to justify the usefulness of our approach. In the experiment, we run *script.rugged* [13] listed in Fig. 6a and our script, *script.local_cube* listed in Fig. 6b on the same MCNC benchmarks. The results of *script.rugged* and *local_cube* are shown respectively in the third and forth column of the TABLE 1. Several MCNC circuits C2670, C3540, C5315, C6288, and C7552 cannot be finished by the script.rugged (full_simplify) procedure due to the excessive memory requirements. For such unfinished examples, we have listed the literal count of the original circuit. The results on all the benchmarks suggest that our technique is very memory and time efficient. For example, one large circuit C7552 requires less than 4Mbyte of memory and takes less than 5 minutes to complete. Another circuit C5315 needs only 3 Mbyte of memory and takes less than 3 minutes. Note that the total cpu time is not listed because some circuits were aborted during computation. In TABLE 1., we also list the results of [15] in the second column. However, we would like to mention that the comparison between max_proj [15] and the other two is not quite fair. The reason is that the initial circuits for the max_proj [15] are different from the initial circuits for full_simplify and local_cube.

4 Conclusion

In this paper, we propose a memory and timing efficient algorithm to compute local don't cares of internal nodes. Our algorithm is based on implication which allows us to quickly identify whether a cube in the local space is a don't cares or not. Instead of checking whether all the minterms in the local space are don't cares or not, we develop a structured search technique that can efficiently trim down the searching space. Our experimental results have demonstrated the usefulness of our approach.

TABLE 1. results from script.rugged and script.local_cube in Fig. 5.

circuit	proj (lits)	rugged (lits)	local_cube (lits)	CPU(sec)full_simp	CPU(sec) local_cube
5xp1-hdl	93	77	77	1.9	0.7
9sym-hdl	102	93	88	3.8	2.2
9sym	--	341	313	54.1	27.5
C1355	-	560	560	167.3	7.0
C1908	-	606 (space out)	560	162.3	12.9
C2670	-	938 (space out)	901	365.2	85.6
C3540	-	2169 (space out)	1483	265.9	1089.8
C432	-	338	329	1261.1	110.4
C499	-	560	560	164.9	6.9
C5315	-	1872	1877	156.9	116.3
C6288	-	3600 (space out)	3500	397.0	122.4
C7552	-	3269 (space out)	2980	327.2	289.7
C880	-	467	467	49.4	30.0
alu2	-	395	415	470.1	140.2
alu4	-	766	845	2099.6	178.9
alupla	182	144	143	5.7	14.7
b9	-	163	161	4.1	11.5
duke2	-	487	492	59.6	33.2
example2	-	356	355	12.2	39.4
frg1	-	212	213	35.5	10.8
lal	-	117	115	6.6	3.3
misex3	561	112	113	276.1	262.9
misex3c	-	544	546	19449.0	76.6
pair	-	1851	1825	176.6	312.7
sao2	-	192	206	14.6	12.9
ttt2	-	290	267	13.9	30.6
unreg	-	102	102	2.2	3.7
x3	-	946	889	44.0	104.7
x4	-	399	399	18.4	33.1
total		23420	22216		

"-": not available

Acknowledgment

This work was supported in part by NSF grant MIP91-17328 and in part by Xilinx through the California MICRO Program. The authors would like to thank the anonymous reviewer for bringing to their attention reference [16].

5 Reference

[1] K.A. Barlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, R. Rudell, A. Sangiovannni-Vincentelli, and A. Wang, "Multilevel Logic Minimization Using Implicit Don't Cares", IEEE Trans. Computer-Aided Design, vol. CAD-7, pp. 723-739, June 1988.

[2] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: Multi-level Interactive Logic Optimization System," IEEE Trans. on CAD, CAD-6(6), pp. 1062-1081, Nov. 1989.

[3] A.C. L. Chang, I, S. Reed, A.V. Banes, "Path Sensitization, Partial Boolean Difference and Automated Fault Diagnosis," IEEE Trans. Computers, vol. C-21, pp. 189-194, Feb. 1972.

[4] S-C Chang and M. Marek-Sadowska, "Perturb and Simplify: Multi-level Boolean Network Optimizer," to be published in ICCAD 94.

[5] K.-T. Cheng and Luis A. Entrena, "Multi-Level Logic Optimization by Redundancy Addition and Removal," in *Proc. European Conf. on Design Automation*, pp. 373-377, Feb. 1993.

[6] M.Damiani and G.De Micheli, "Don't Care Set Specifications in Combinational and Synchronous Logic Circuits", IEEEE Trans. on CAD, Vol 12, no. 3, pp.365-388, March 1993.

[7] L.A. Entrena and K.T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," Proc. Int. Conf. on CAD, November 1993.

[8] G. D. Hachtel, R. M. Jacoby, P. H. Moceyunas, "On Computing and Approximating the Observability Don't Care Set," in Proc. Int. Workshop on Logic Synthesis, Research Triangle Park, May 1989.

[9] T. Kirkland and M.R. Mercer, "A Topological Search Algorithm For ATPG," *Proc. 24th Design Automation Conf.*, pp. 502-508, June 1987.

[10] W. Kunz and D. K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits", in *Proc. Int'l Test Conference*, pp. 816-825, October 1992.

[11] S.Muroga, Y. Kambayashi, H. Lai and J. Culinary, "The Transduction Method-Design of Logic Networks Based on Permissible Functions," IEEE Trans. Computers, vol. 38, pp. 1404-1424, 1989.

[12] H. Savoj and R. K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multiple-level Networks," in Proc. Design Automation Conference, June 1990.

[13] H. Savoj, R. K. Brayton and H. Touati, "Extracting Local Don't Cares for Network Optimization," in Proc. Int. Conf. on Computer Aided Design 1991.

[14] M. Schulz and E.Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," *Proc. Fault Tolerant Computing Symposium*, pp. 30-34, June 1988

[15] T. Stanion and C.Sechen, "Maximum Projections of Don't Care Conditions in Boolean Network," Proc. Int. Conf. on Computer Aided Design, pp. 674-679, 1993.

[16] A. Zemva F. Brglez, k,Kozminski, and B. Zajc, " A Functionality Fault Model: Feasibility and Applications," Proc. EDAC, pp. 152-158, 1994