Efficient Don't Care Computation for Hierarchical Designs

Kanupriya Gulati[‡] Matthew Lovell[*] Sunil P Khatri[‡]
[*] Hewlett-Packard Company, Fort Collins, CO 80528
[‡] Department of EE, Texas A&M University, College Station TX 77843.

Abstract

In this paper, we describe a BDD-based hierarchical don't care computation algorithm. In contrast to traditional don't care computation techniques, our method retains the hierarchy in the design netlist during the don't care computation. Although this may reduce some of the flexibility inherent in the optimization process, it allows our technique to handle large designs. Our method computes don't cares at input and output interfaces of different modules in the hierarchy by an image computation process. In case an exact image image cannot be computed, our method computes the largest approximate image. Once the don't cares at the input and output interfaces are computed, the hierarchical instances are optimized separately using a traditional optimization flow. Experimental results demonstrate that our technique can achieve a 36% reduction in literal count for large hierarchical designs, with reasonable runtimes. Our method can complete for several examples in which flattened optimization fails.

1. Introduction

Logic synthesis typically begins by producing a flat netlist for a circuit, collapsing hierarchy in the process. SIS [1], for example, does not retain hierarchy in its internal data structures even though the *blif* input format for SIS can represent hierarchy. Even VIS [2], which retains the hierarchy in a network, has as its central data structure a multi-level network constructed by flattening hierarchy. Flattening an entire circuit may result in a representation which is too large to work with effectively, even if the original hierarchy had been carefully partitioned to avoid such a problem. Without such flattening, synthesis may be somewhat constrained since some don't care conditions are not exposed. However, in practice, we find that this is not a major problem.

A synthesis flow which retains hierarchy during the optimization flow can be useful in several ways.

- It can enable existing tools to handle larger designs, increasing their effectiveness.
- The retention of hierarchy can enable the designer to focus their attention in optimizing those modules in the hierarchy which are critical in terms of the optimization objective. If the hierarchy was flattened, the absence of the separate information on the relative effect of the optimization on each hierarchical module makes this hard to do.
- A design may consist of several instances of the same hierarchical module. In the absence of separate don't care information for the different instances, a designer would be forced to optimize them as a unit, with no don't cares at the I/O interfaces. However, with hierarchical don't care information, available separately for each instance, the designer could optimize each module separately, obtaining a better optimized design. In case there is a requirement that a single module be used after optimization, the don't cares at the input interfaces of the different instances can be intersected, resulting in a better optimized design.

In our approach, we extract don't care conditions from a hierarchical circuit while leaving that hierarchy intact. Since a recursive flattening is not performed, there is no need to build a representation of the *entire* circuit. At any given level of the circuit, don't care minterms are extracted for each subcircuit by performing an image computation starting from that level's primary inputs. This resulting image is then utilized recursively by other levels of the design hierarchy. Since subcircuit outputs may feed other subcircuits, don't care conditions on outputs are also computed, and can be used by the the other subcircuits. The don't cares are recorded in blif files associated with each subcircuit, and used to optimize each subcircuit separately.

Of course, if the design has been poorly partitioned, the don't care image computations may end up building a representation of the majority of the circuit anyway. We assume that the design hierarchy has been created with care, and that this situation is therefore avoided. Also, by separating the hierarchy into separate files, some optimality may be sacrificed. This is a consequence of no longer being able to extract satisfiability and observability don't cares across hierarchical boundaries. However, we find that the loss in optimality due to the use of hierarchical design optimization is minimal.

The remainder of this paper is organized as follows. Section 2 describes previous work in this area, while Section 3 describes our approach. Section 5 describes our experimental results. Finally, in Section 6, we make concluding comments and discuss further work that needs to be done in this area.

2. Previous Work

To the best of our knowledge, little research has been written on this practical technique for logic optimization. The only relevant article seems to be [3], which at this point is nearly 14 years old. In this work, a hierarchical optimization technique was proposed, but the effectiveness of this approach was not shown. Other research has concentrated on optimization in the presence of subcircuits treated as "black boxes" [4]. Our work, in contrast, exploits the functionality of the subcircuits in the hierarchy to infer don't cares at the input and output interfaces of other subcircuits in the design.

Hierarchy has been exploited to compute don't cares in very restrictive (mux-based) networks in [5]. In this work, the authors extract don't cares from the datapath portion of a design, and use them to optimize the control portion. The circuit model in this case is therefore also restricted. In contrast, our approach works for arbitrary hierarchies, with arbitrary logic networks in the subcircuits of the hierarchy. In [6], the authors report a technique for sequential optimization, which exploits the partitioning between the datapath and control portions of a design.

3. Our Approach

3.1 Formulation

The core of the paper is a recursive image computation to compute the don't cares at the interfaces of blocks in the hierarchy. Let us begin by considering the acyclic, combinatorial circuit shown in Figure 1. It depicts a circuit A containing a child circuit B, which in turn has a child circuit C. It is acyclic in the sense that no loop, either sequential or combinatorial in nature, exists among subcircuits. Note that if any node in the hierarchy has latches, we make its latch outputs as subcircuit primary inputs, and latch inputs as

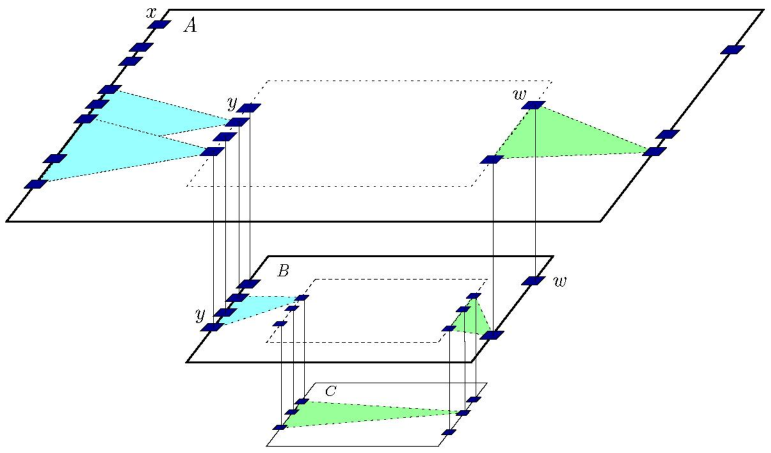

Figure 1: Computation of Interface Don't Cares

subcircuit primary outputs. Let A's primary inputs be denoted by $x = x_1, x_2, \ldots, x_n$ (note that some of these may correspond to outputs of B). Let $y = y_1, y_2, \ldots, y_m$ denote B's primary inputs. Suppose, for the moment, that the transitive fanin for y consists solely of x. Then, each y_i is given by some function $f_i(x)$. We may capture the fact that y_i never differs from $f_i(x)$ by writing $y_i \overline{\oplus} f_i(x)$. The characteristic function of B's primary inputs can be found by conjoining such terms over all i,

$$T(x,y) = \prod_i y_i \overline{\oplus} f_i(x) \tag{1}$$

Let $D(x)$ denote the external don't care set for x, i.e., the minterms which do not appear on the inputs of A. The minterms which *will* appear can be denoted $C(x) = \overline{D(x)}$. Conjoining $C(x)$ with $T(x,y)$ yields the characteristic function for y given constraints on the values that x can assume. Finally, existentially quantifying (smoothing out) the variables in x from this characteristic function yields an expression $C(y)$ which represents the care points on the input interface of B.

$$C(y) = Img[C(x)] = \exists x. \left[\left(\prod y_i \overline{\oplus} f_i(x)\right) \wedge C(x)\right] \tag{2}$$

Equation (2) characterizes the minterms which can appear at the input interface of subcircuit B. It is also precisely the same as the forward image computation used in formal verification [7, 8]. Applying this formulation recursively yields the input care set, and hence the don't cares at the input interfaces of all subcircuits in the hierarchy. In our example, the next step in the recursion yields the don't cares on the inputs of C. When the recursion reaches the leaves of the hierarchy, a similar image computation yields the care (don't care) set at the output interfaces of the subcircuits in the design. Specifically, if y denotes the leaf circuit's inputs, w denotes its outputs, and $g_i(y)$ the output functions, we have

$$C(w) = Img[C(y)] = \exists y. \left[\left(\prod w_i \overline{\oplus} g_i(y)\right) \wedge C(y)\right] \tag{3}$$

Here $C(y)$ is the care set image on the input interface of the input interface of the leaf block being operated upon.

Using a similar notation as in Equation 1, we denote

$$T(y,w) = \prod_i w_i \overline{\oplus} g_i(y) \tag{4}$$

As the recursion unwinds, we combine the care sets of different subcircuits at the same level of the hierarchy, by computing their conjunction. We make appropriate variable substitutions to change the support of the don't cares from the formal variables of a block to its actual variables (or vice versa). In this way, we determine the care sets of the output interfaces all the way up the hierarchy. These output care sets are necessary since some subcircuit outputs feed parent subcircuits; they thus become care sets at the input interface of parent subcircuits, for further recursion.

We extended this technique to handle sequential circuits, by considering each latch output as another free input for the logic it feeds, and a latch input as an output of the logic that drives it. Alternatively, one could perform limited sequential analysis on the latch outputs and use those results.

Finally, it should also be possible to handle circuits with loops between subcircuits. We conjecture that a fixpoint-like computation of the care sets of the design should be able to handle the presence of loops correctly. Our goal in this paper was to handle loop-free circuits, and to break the loops induced by memory elements.

3.2 Approximate Computation

The formulation described in Section 3.1 does not lend itself to robust ROBDD [9, 10] based computations. The characteristic functions $T(x,y)$ and $T(y,w)$ described in Section 3.1 may have large ROBDDs, causing the computation to fail.

To alleviate this problem, we compute approximate characteristic functions $T'(x',y)$ and $T'(y',w)$ in our approach. Note that there may be common variables between x and x', and also between y and y'. Our approximate computation computes the characteristic functions such that the size of the ROBDDs of these functions is bounded by a value S.

Our approximate computation is schematically illustrated in Figure 2. Consider the task of computing the input interface care set for subcircuit B. From node $y_i \in y$, we perform a reverse topological traversal until we reach outputs of latches or outputs of subcircuits of A, or primary inputs of A. The variables found in this manner comprise x. We then attempt to compute the characteristic function $T(x,y)$. If any step in this computation results in a ROBDD which has a size greater than S, we abort the computation. Then, from nodes $x_i \in x$, we compute the fanouts f_j. The set of fanout nodes now comprise the set x'. We again try to construct $T(x',y)$, and if this attempt fails as well, we update the set x' as described above. At the end of this exercise, we obtain the characteristic function of the input interface variables of subcircuit B.

A similar approximate computation is used for the computation of $T(y,w)$. These approximate computations are integrated into the recursive computation of care points for subcircuit input and output interfaces, described in Section 3.1.

The approximate computation of $C'(y)$ and $C'(w)$ based on the use of $T(x',y)$ and $T(y',w)$ is conservative.

LEMMA 3.1. $C'(y) = \exists x'. [T(x',y) \wedge C(x')] \supseteq C^{exact}(y) = \exists x. [T(x,y) \wedge C(x)]$

PROOF. Let $C^{exact}(x') = \exists x. [T(x,x') \wedge C(x)]$. Using $C^{exact}(x')$, we can compute the exact care set image (RHS) as: $C^{exact}(y) = \exists x'. \left[T(x',y) \wedge C^{exact}(x')\right]$. However, in the expression for the LHS, $C(x') \supseteq C^{exact}(x')$, yielding the result.

□

In a similar manner, we can prove that the computation of the care image based on the use of $T(y',w)$ is conservative.

In most cases, the variable sets x and x', as well as y and y' have common variables, allowing the care functions found during the input interface care set computation to be utilized during the output interface care set computations, even though the computation is approximate.

In actual designs, the presence of input don't cares is less likely on datapath elements such as adders and multipliers. While our implementation correctly identifies this situation, it could be expensive to run the algorithm on such designs. It thus seems more appropriate for random control logic. The examples we have used to test our approach are based on datapath primitives, since it was not possible to obtain true hierarchical random logic blif benchmarks in the public domain. The benchmarks we did obtain were from the VIS-2.0 benchmark suite, and had a significant datapath component in them.

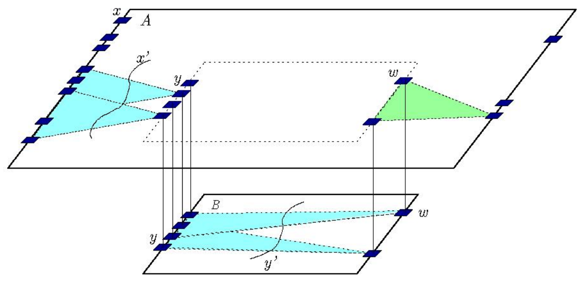

Figure 2: Approximate Computation of Interface Don't Cares

3.3 An Example

Consider the circuit shown in Figure 3. It consists of a top level block P with two subcircuits Q_1 and Q_2, both instances of a module Q.

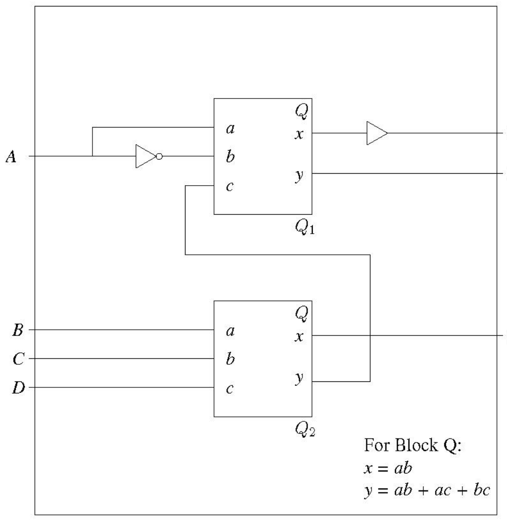

Figure 3: Example - Computation of Interface Don't Cares

Our computation first does a topological sort of blocks Q_1 and Q_2. Since Q_2 has an output y which is an input to Q_1, we first process Q_2. The don't care set on the input interface of Q_2 is empty. The don't care set on the outputs (x,y) of Q_2 is { 10 }. Then we compute the don't care set on the inputs (a, b, c) of the block Q_1, and this gives us { 00-, 11- }. Finally, we compute the output image of Q_1, and get the don't cares at the top level of the circuit, on the space (Q1_y, Q1_x, Q2_y, Q2_x) as (0001, 100-, -10-, -11-).

4. Implementation

The work described in this paper was implemented in VIS [2]. Since VIS retains hierarchy information, and since it contains packages for image computation, the don't care computations were implemented as a VIS extension. SIS was then used, without modification, to optimize the resulting blif files, each with attached external don't cares specified.

While VIS retains hierarchy information and makes it accessible to the programmer, many of VIS' algorithms still concentrate on flattened designs. In our approach, we construct non-flattened data structures within VIS, in order to compute input and output interface don't cares of subcircuits within the hierarchical design.

The algorithms we used to construct the interface don't cares are described in Section 3. We compute the output care set (henceforth called the outputImage) of a given level of the design based on the output care sets of all subcircuits at that level. Of course, to determine those output care sets, one must first determine the input care set for each subcircuit. Since each subcircuit may itself contain hierarchy, the procedure is recursive. It is invoked at the current node of the design with tautology as the initial careSet.

Algorithm 1 Function ComputeDontCares(hrcNode,careSet)

```
Create network for hrcNode, i.e., the current level
Order children
for each child do
  inputImage ← Compute child's input care set given careSet
  outputpreImage ← outputpreImage ∧ ComputeDontCares(child,inputImage)
end for
outputImage ← Compute this level's output care set given outputpreImage
write out blif file
return outputImage
```

Network creation is accomplished via almost the same route as in VIS. The key difference is that all recursion has been removed. Additionally, some network restructuring is performed. All signals which has subcircuit inputs become primary outputs of that level, and all subcircuit outputs become primary inputs. A similar idea is used for latches in the design. Special care is taken for signals which do nothing but connect a subcircuit directly to its parent's primary inputs or outputs. In such cases, we create "buffers", so that some logic exists within the parent network.

The computation of input and output care sets for each child subcircuit also takes advantage of the constructed network. The care set or image computation is performed by calling functions in the VIS image package. By using the image package, all of the algorithms and flexibility constructed into VIS for image computation is utilized. All constructed `ntk_network_t`'s utilize the same MDD manager, which is passed down through the recursion. This enables the results of the image computation to be utilized meaningfully throughout the design.

It is also worth noting how the constraints of the computed care sets are communicated to each subcircuit's logic. When the recursion encounters a new subcircuit, some manipulation of MDD ID's is performed. Each formal input of the subcircuit is given the same MDD ID as its corresponding actual input. The connectivity is establish like this, rather than using `bdd_substitute`, since the mapping from parent signals to subcircuit inputs is highly flexible. It is possible, after all, to instantiate a subcircuit and connect one signal to *every* input. Output connectivity, in contrast, is established using `bdd_substitute`, since there is a guaranteed one-to-one signal correspondence in that direction. The connectivity between formal inputs and outputs on one level and the actual signals in the parent are made via name.

5. Experimental Results

We performed several experiments to compare our technique (hierarchical, using interface don't cares) with a flattened optimization approach (no hierarchy used) and a hierarchical technique using no interface don't cares. We used benchmark designs from the VIS-2.0 [2] distribution. Of the benchmarks in the VIS-2.0 distribution, we selected those which had hierarchy, and did not have non-deterministic variables in any subcircuit of the design. In general, these benchmarks had a good fraction of arithmetic circuits in them. We expect better results with random-logic style hierarchical benchmarks, but unfortunately it is difficult to obtain such designs in the public domain. We also performed experiments on randomly generated examples. Starting with flat blif files from the MCNC benchmark suite, we generated hierarchy by instantiating the designs in different files at any level of hierarchy. Half the instantiated files were collapsed, resulting in a hierarchical design. This was repeated, to create deeper hierarchies. The randomly generated examples are named with a prefix "rex".

During hierarchical don't care computation we set the threshold S to be 50,000 ROBDD nodes. Also, in case the cover of the ex-

Ckt	Hierarchical Opt.						Flat Opt		
	Lits before	With DC			Without DC		Lits before	Lits after	Time(ms)
		Lits after	VIS Time (s)	SIS Time (s)	Lits after	SIS Time (s)			
dp32	575	0.74	0	2	0.75	0	1177	0.74	19
dp8	143	0.71	0	0	0.75	0	222	0.88	0
gcd16	1706	0.88	8	7	0.91	1	2017	1	spaceout
gcd12	1042	0.85	1	3	0.89	0	1269	1	spaceout
gcd8	863	0.88	0	1	0.91	0	1009	1	timeout
gcd4	439	0.86	0	1	0.91	0	505	0.67	0
Vcordic	7149	0.47	40	11	0.48	38	9624	1	spaceout
Huff	8543	0.32	112	38	0.34	107	8600	0.28	295
rex1	2046	0.635	48	2	0.743	3	2054	0.48	40
rex2	4293	0.608	4	3	0.745	2	4301	1	spaceout
rex3	1812	0.48	0	0	0.582	0	1814	0.32	2
rex4	4527	0.580	58	2	0.67	3	4545	1	spaceout
rex5	2046	0.563	39	2	0.583	3	2054	0.48	40
rex6	3908	0.570	118	2	0.67	3	4545	1	spaceout
rex7	1812	0.512	2	1	0.939	1	1814	0.32	2
AVG	2726.9	0.64	28.7	5	0.72	10.7	3036.7	0.74	

Table 1: Comparison of Hierarchical and Flattened Don't Care Computation

ternal don't cares was too large, we limited the number of cubes written out to the blif files to 500. For flattened SIS optimization, the timeout value was set to 10 minutes, which is quite high compared to the hierarchical runtimes. All experiments were run on a Compaq Presario 2100, with a 2.4GHz Celeron CPU and 512MB of RAM, running the Linux operating system.

Our optimization approach was implemented in VIS [2]. After traversing the hierarchy of the design and computing the interface don't cares as described in Section 3, we wrote out the blif files corresponding to each subcircuit. These files were then read into SIS [1] and optimized. These results are indicated in Table 1 under the column **Hier. with DC**.

We also wrote out blif files corresponding to each subcircuit, but without any interface don't cares. These results are indicated in Table 1 under the column **Hier. without DC**.

For hierarchical optimization, the run-time reported consists of the sum of the SIS run-times for each subcircuit in the design. In case of the *Hier. with DC* method, we also include the time taken to compute the interface don't cares.

Finally, we performed a flattened optimization in SIS, and these results are indicated in Table 1 under the column **Flattened**. Note that for flattened SIS optimization, some examples failed due to timeout or spaceout conditions, as indicated in Table 1. Also, the total number of literals before optimization was larger than in the case of hierarchical optimization, since the flattened design may have several copies of the same subcircuit of the hierarchy.

In all cases, we assumed the design was a loop-free combinational circuit. Sequential designs were optimized as well, but this was done by making their latch inputs as circuit primary outputs, and the latch outputs into circuit primary inputs.

In Table 1, the first column reports the circuit being optimized. The second column reports the number of literals (in the factored form) in the hierarchical design. This is computed as a sum over all subcircuits in the hierarchy. The third column reports the literals after optimization using our method (as a fraction of the literals before optimization). The fourth column reports the run-time in VIS (spent computing the interface don't cares) while the fifth column reports the total SIS run-time (over all sub-circuits of the design). The sixth and seventh columns represent the literals after optimization and the SIS run-time, in case hierarchical interface don't cares are not used.

The last three columns represent the literals before optimization, literals after optimization, and run-time for SIS in a flattened optimization approach. Literals after optimization are represented as a fraction of the literals before optimization.

We observe that our method achieves a better literal count reduction (about 36%) compared to flattened optimization. This is because the flattened optimization method either times out or spaces out on several large examples. Our literal count reduction is increased by 12.5% over the the *Hier. without DC* method, with a nominal increase in run-time. Additionally, we can complete the optimization for some examples in which flattened optimization is not possible due to timeout or spaceout conditions.

6. Conclusions

In this paper, we describe a BDD-based don't care computation algorithm, which is applicable even to large hierarchical combinational designs. Our method retains the hierarchy in the design netlist during the don't care computation, unlike existing methods. Although this may reduce some of the flexibility inherent in the optimization process, this reduction is minimal and so the approach allows our technique to handle large designs.

Our method computes don't cares at input and output interfaces of different modules in the hierarchy by an image computation process. In case an exact image image cannot be computed, our method computes the largest approximate image. Once the don't cares at the input and output interfaces are computed, the hierarchical instances are optimized separately using a traditional optimization flow. Experimental results demonstrate that our technique can achieve a 36% reduction in literal count for large hierarchical designs, with reasonable runtimes. Our method completes for some examples in which flattened optimization is infeasible.

References

[1] E. M. Sentovich, K. J. Singh, L. Lavagno, *et al.*, "SIS: A system for sequential circuit synthesis," Memorandum UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, May 1992.

[2] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, *et al.*, "VIS: A system for verification and synthesis," in *International Computer-Aided Verification Conference* (R. Alur and T. A. Henzinger, eds.), vol. 1102, (New Brunswich, NJ), pp. 428–432, Springer-Verlag, July–August 1996.

[3] G. D. Hachtel and M. R. Lightner, "Don't care conditions in top down synthesis," in *IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 316–319, IEEE Computer Society Press, November 1987.

[4] T.-H. Liu, K. Sajid, A. Aziz, *et al.*, "Optimizing designs containing black boxes," in *Proceedings, Design Automation Conference*, (Anaheim, CA), pp. 113–116, June 9–13, 1997.

[5] S. Bhattacharya, F. Brglez, and S. Dey, "Transformations and resynthesis for testability of RT-level control-data path specifications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, pp. 304–318, Sept 1993.

[6] H. Eikerling and J. Rosenstiel, "Automatic structuring and optimization of hierarchical designs," *Proceedinsgs, European Design Automation Conference with EURO-VHD '96 and Exhibition*, pp. 134–139, Sept 1996.

[7] K. McMillan, *Symbolic Model Checking.* Boston, MA: Kluwer Academic Publishers, 1994.

[8] E. Clarke, O. Grumberg, and D. Peled, *Model Checking.* Cambridge, MA: MIT Press, 1999.

[9] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[10] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proc. of the Design Automation Conf.*, pp. 40–45, June 1990.