# A Robust Algorithm For Approximate Compatible Observability Don't Care (CODC) Computation

Nikhil Saluja
Dept ECE, University of Colorado
nikhil.saluja@colorado.edu

Sunil P. Khatri
Dept ECE, University of Colorado
spkhatri@colorado.edu

## ABSTRACT

Compatible Observability Don't Cares (CODCs) are a powerful means to express the flexibility present at a node in a multi-level logic network. Despite their elegance, the applicability of CODCs has been hampered by their computational complexity. The CODC computation for a network involves several image computations, which require the construction of global BDDs of the circuit nodes. The size of BDDs of circuit nodes is unpredictable, and as a result, the CODC computation is not robust. In practice, CODCs cannot be computed for large circuits due to this limitation.

In this paper, we present an algorithm to compute approximate CODCs (ACODCs). This algorithm allows us to compute compatible don't cares for significantly larger designs. Our ACODC algorithm is scalable in the sense that the user may trade off time and memory against the accuracy of the ACODCs computed. The ACODC is computed by considering a subnetwork rooted at the node of interest, up to a certain topological depth, and performing its don't care computation. We prove that the ACODC is an approximation of its CODC.

We have proved the soundness of the approach, and performed extensive experiments to explore the trade-off between memory utilization, speed and accuracy. We show that even for small topological depths, the ACODC computation gives very good results. Our experiments demonstrate that our algorithm can compute ACODCs for circuits whose CODC computation has not been demonstrated to date. Also, for a set of benchmark circuits whose CODC computation yields an average 28% reduction in literals after optimization, our ACODC computation yields an average 22% literal reduction. Our algorithm has runtimes which are about 25× and memory utilization which is 33× better that of the CODC computation of SIS.

## Categories and Subject Descriptors

B.6.3 [**Logic Design**]: [Automatic Synthesis, Optimization, Design Aids]

## General Terms

Algorithms, Design, Performance


*DAC 2004,* June 7–11, 2004, San Diego, California, USA


## Keywords

Compatible Observability Don't Cares (CODC), Multi-level Logic Optimization, Logic Synthesis

## 1. INTRODUCTION

Technology independent logic optimization of a multi-level network is an important part of logic synthesis. During such optimization, one of the operations involves the computation of multi-level don't cares of the circuit. These don't cares can take the form of Satisfiability Don't Cares (SDCs), Observability Don't Cares (ODCs) or External Don't Cares (XDCs). These are described further in [7].

After computing these don't cares for any node in a network, we find the *image* of the don't cares of the node on its local fanin space. Thus the node function can be considered as an incompletely specified function (ISF) and traditional 2-level logic minimization of this ISF yields a reduction in the number of literals in the node. This is the typical figure of merit during technology independent logic optimization.

Although ODCs are extremely powerful in their optimization ability, the minimization of a node $n$ with respect to its ODCs can potentially change the ODCs at all other nodes in the circuit, resulting in a need to re-compute ODCs for all circuit nodes. As a result, Compatible ODCs (CODCs) - a subset of ODCs - are utilized. If a node $n$ is minimized with respect to its CODCs, then the CODCs of all other circuit nodes are still valid (and therefore do not need to be recomputed).

The CODC computation does not eliminate the need to perform image computation to map the don't care information back to the local fanin space of the node being optimized. This computation is highly memory intensive, since it is typically done using ROBDDs [4]. ROBDDs can exhibit highly irregular memory requirements, with unexpected blowup. As a result, the CODC computation, though very elegant in its conception, is typically not feasible for large circuits. Additionally, the computation is not robust for medium-sized circuits either.

For these reasons, CODC based optimization is rarely applicable for large industrial circuits. Our goal in this paper is to extend the applicability of CODCs by introducing an approximate variant which is easier to compute. Our algorithm for approximate CODC (ACODC) computation allows the user to trade off the accuracy of the computed don't cares against computational resources.

The rest of this paper is organized as follows. In Section 2, we provide definitions along with a brief background on CODC computation. Section 3 contains a summary of pre-

vious work in this area, while Section 4 details our ACODC computation methodology. In Section 5, we report results of our experiments comparing the ACODC and CODC methods. Finally we conclude in Section 6.

## 2. PRELIMINARIES AND TERMINOLOGY

DEFINITION 1. *A Boolean network $\eta$ is a directed acyclic graph (DAG) in which every node has a Boolean function $f_i$ associated with it. Also, $f_i$ has a corresponding Boolean variable $y_i$ associated with it, such that $y_i = f_i$.*

There is a directed edge $e_{ij}$ from $y_i$ to $y_j$ if $f_j$ depends explicitly on $y_i$ or $\overline{y_i}$.

A node $y_i$ is a *fanin* (FI) of a node $y_j$ if there is a directed edge $e_{ij}$. Node $y_i$ is a *fanout* (FO) of $y_j$ if there is a directed edge $e_{ji}$.

A node $y_i$ is in the *transitive fanin* (TFI) of a node $y_j$ if there is a directed path from $y_i$ to $y_j$. Node $y_i$ is in the *transitive fanout* (TFO) of node $y_j$ if there is a directed path from $y_j$ to $y_i$.

DEFINITION 2. **Reduced Ordered Binary Decision Diagrams (ROBDD's)** *[4] are a means to represent a Boolean function $f$. They are modified Shannon decompositions of $f$ in which any path from the root to the leaves obeys the same variable ordering, and isomorphic nodes are deleted from the decomposition.*

For a given variable ordering, ROBDDs are canonical. In other words, the ROBDDs of equivalent functions are identical.

DEFINITION 3. *The* **consensus operator** *or* **universal quantification** *of a function $f$ with respect to a variable $x_i$ is*

$$C_{x_i} f = f_{x_i} \cdot f_{\overline{x_i}}$$

DEFINITION 4. *The* **existential quantification** *of a function $f$ with respect to a variable $x_i$ is*

$$\exists x_i f = f_{x_i} + f_{\overline{x_i}}$$

DEFINITION 5. *The Observability Don't Care (ODC) of node $y_j$ (in a multi-level Boolean network) with respect to output $z_k$ is*

$$ODC_{jk} = \{x \in B^n \ s.t. \ \ z_k(x)|_{y_j=0} = z_k(x)|_{y_j=1}\}$$

In other words $ODC_{jk}$ is the set of minterms of the primary inputs for which the value of $y_j$ is *not observable* at $z_k$ [13]. This can also be denoted as

$$ODC_{jk} = (\overline{\frac{\partial z_k}{\partial y_j}})$$

where

$$\frac{\partial z_k}{\partial y_j} \quad = \quad z_k(x)|_{y_j=0} \oplus z_k(x)|_{y_j=1} \qquad (1)$$

$\frac{\partial z_k}{\partial y_j}$ is also known as the Boolean Difference of $z_k$ with respect to $y_j$.

Once a node function is changed by minimizing it against its ODCs (using Espresso [3]), the ODCs of the other nodes must be recomputed. To avoid re-computation of ODCs during optimization, Compatible Observability Don't Cares (CODCs) [13] were developed. The CODC of a node is a subset of the ODC for that node. Unlike ODCs, CODCs have a property that one can *simultaneously* change the function of all nodes in the network as long as each of the modified functions are contained in their respective CODCs. Compatibility is achieved by ordering the don't care computation in a manner that if node $n$ precedes node $m$ in the order, then node $n$ gets the most don't cares. Also, while computing the don't cares of $m$, compatibility is maintained with the don't cares already assigned to node $n$.

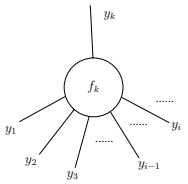


**Figure 1: Node $y_k$ and its fanins**

The computation of CODCs [13] in SIS [15] is performed in two phases.

- The first phase computes the CODC of a node using node operations. The resulting CODC is a function of arbitrary nodes in the network. However, we desire the support of the CODC to be the support of the node itself. For this reason, we need a second phase in the computation to achieve this.
- In the second phase, image computation needs to be performed to map the CODC points to the local fanin space of each node. This image computation uses BDDs.

In the first phase, CODC computation for the network $\eta$ starts from the primary outputs and proceeds towards primary inputs in a reverse topological order. The CODC at each primary output is initialized to the external don't care (XDC) at that node.

The CODC at node $y_i$ (denoted as $CODC_i^\eta$) is found by using the CODC for each fanout edge $e_{ik}$ of $y_i$. This compatible don't care of edge $e_{ik}$ is denoted by $CODC_{ik}^\eta$. The CODC for $y_i$ is obtained by intersecting the CODCs computed for its fanout edges.

Suppose, as shown in Figure 1, that we have a node with function $y_k = f_k$, and ordered fanins $y_1 < y_2 < \cdots < y_i$. Given $CODC_k^\eta$, we can compute the CODCs of the fanin edges of $y_k$ as follows.

$$CODC_{ik}^\eta \equiv (\frac{\partial f_k}{\partial y_1} + C_{y_1})...(\frac{\partial f_k}{\partial y_{i-1}} + C_{y_{i-1}})\overline{\frac{\partial f_k}{\partial y_i}} + CODC_k^\eta \quad (2)$$

Note that in the above computation, we assume that $CODC_j^\eta$ have already been computed (for $j < i$). Also, $C_{y_j}$ is the consensus operator. For the first input in the ordered list of fanins, we have $CODC_{1k}^\eta = (\overline{\frac{\partial f_k}{\partial y_1}}) + CODC_k^\eta$, indicating that this node obtains maximum flexibility. The intuition behind the correctness of the above computation in general is that the new edge $e_{ik}$ should have its don't care condition as the conjunction of $\overline{\frac{\partial f_k}{\partial y_i}}$ with the condition that other inputs $j < i$ are not insensitive to the input $y_j$ ($\frac{\partial f_k}{\partial y_j}$), or are independent of $y_j$ ($C_{y_j}$) indicating that the node $i$ is free

to use such terms independently of how $y_j$ was simplified. Finally, the CODCs of the fanout node $y_k$ are also CODCs of the edge $e_{ik}$. As a result, the computation of $CODC^{\eta}_{ik}$ using the formula above, performed in the specified order, results in the Compatible set of ODCs of the edges $e_{ik}$.

$$CODC^{\eta}_{i} \equiv \prod_{k \in FO_i} CODC^{\eta}_{ik} \quad (3)$$

The intuition of the above method of computing the CODC of $y_i$ based on its edge CODCs ($CODC^{\eta}_{ik}$) is that $CODC^{\eta}_{i}$ must not be greater than any $CODC^{\eta}_{ik}$. Note also that all terms in $CODC^{\eta}_{ik}$ from Equation 2 except $CODC^{\eta}_{k}$ have a support which is the support of $y_k$. However, $CODC^{\eta}_{i}$ has a support which includes its fanout's fanins, and in general, we have to do an image computation to convert this CODC to a function which is on the support of $y_i$.

In the second phase, we perform this image computation. Using the ordering heuristics of [10], global BDDs at each node of the Boolean network are computed in terms of the primary inputs. BDDs are also built for each primary output in the external don't care network using this same ordering. We next compute the CODC at $y_i$ in terms of primary inputs, using a BDD based computation. This is done by substituting each literal of the CODC of $y_i$ by its global BDD function (which was computed earlier). From this we find all the points that are reachable in the local space of $y_i$ by a BDD based image computation. The functions used for image computation are the global functions at the fanins of $y_i$. In most cases the number of primary outputs is much less than the number of primary inputs; therefore the recursive image computation method [17] is used to do the computation.

# 3. PREVIOUS WORK

Previous work in the area of computing approximate don't cares has been somewhat scarce. In [9], the authors describe a method to compute don't cares using overlapping subnetworks, computed using a varying window size. Their method does not optimize wires, but only gates in a design, in contrast to our technique, which frequently removes wires in a circuit. Further, this technique uses [8] to optimize a single subnetwork. In [8], optimization is done by manipulating a cover of the subnetwork explicitly. The authors indicate that this requires large amounts of runtime for small networks. As a consequence, the technique of [9], in many examples, requires run-times which are dramatically larger than MIS [2]. In [6], the authors partition the circuit into subnetworks, each of which is flattened and optimized using ESPRESSO [3]. Our technique uses implicit ROBDD methods to control the runtime and memory utilization of the computation. We instead perform CODC computation on overlapping subnetworks, and demonstrate that our technique is *significantly* faster than the full CODC computation with comparable literal reductions.

In [11, 5], the authors compute subsets of the ODC, but compatibility is not guaranteed. In [16], techniques to compute the projection of ODCs on some subset of nodes are reported, but compatibility is not guaranteed. Finally in [12] the authors discuss the simplification of non-deterministic MV networks and their internal nodes using internal flexibilities. They provide a new scheme to compute the complete (maximum) don't care in an MV network, however, once again, the don't care computed is not compatible. Although compatibility is sometimes not useful (eg. when nodes are processed one at a time), it is required in many instances (eg. performing any logic optimization step using the don't cares of the network). In contrast, our method computes a subset of the CODC [13], and therefore the don't-cares we compute may be used in an optimization setting without the need for re-computation of the don't cares for node $y_k$ after optimizing the node $y_j$ using its don't cares.

In [1], the CODC computation of [14] was shown to be dependent on the current implementation of a node, and an implementation-independent computation was proposed. These local don't cares are orthogonal to our proposed approach, and it would be an interesting exercise to demonstrate them in tandem with our method.

# 4. APPROXIMATE CODC COMPUTATION

Consider a Boolean network $\eta_{XZ}$ with $X$ as primary inputs and $Z$ as primary outputs. Let us also define two cuts of this network, with cut variables $W$ and $V$. These cuts define new subnetworks $\eta_{XW}$, $\eta_{VZ}$ and $\eta_{VW}$. Let the CODC of a node $y_k$, as computed by equation 3, be denoted as $CODC^{\eta_{PQ}}_{k}$ where $P$ can be either $X$ or $V$. Likewise, $Q$ can be either $Z$ or $W$. Let the CODC of node $y_k$, mapped back to its support after image computation, be denoted as $CODC^{\eta_{PQ},FI}_{k}$. In this context, $CODC^{\eta_{XZ},FI}_{k}$ is the traditional CODC of node $y_k$.

In Section 4.1 we prove that a CODC computation that utilizes a cut $W$ as the primary outputs yields $CODC^{\eta_{XW},FI}_{k} \subseteq CODC^{\eta_{XZ},FI}_{k}$. Next, in Section 4.2 we prove that a CODC computation that utilizes a cut $V$ as the primary inputs for the don't care computation yields $CODC^{\eta_{VZ},FI}_{k} \subseteq CODC^{\eta_{XZ},FI}_{k}$. Finally, we show, in Section 4.3, that an ODC computation that utilizes a cut $V$ as the primary inputs and a cut $W$ as the primary outputs for the don't care computation yields $CODC^{\eta_{VW},FI}_{k} \subseteq CODC^{\eta_{XZ},FI}_{k}$.

Figures 2, 3 and 4 illustrate the corresponding circuits.

## 4.1 Case 1: Cutset used as Primary Output

Suppose we partition the network as shown in Figure 2.

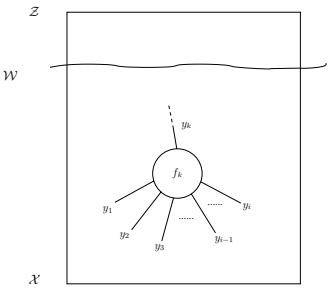


**Figure 2: Partition network 1**

THEOREM 4.1. *If the CODC of a node $y_k$ is computed in terms of cut nodes $W$ in the TFO of $y_k$, then $CODC^{\eta_{XW},FI}_{k} \subseteq CODC^{\eta_{XZ},FI}_{k}$*

PROOF. Consider the case of computing the full CODC of $y_k$. The computation proceeds in a reverse topological order from the POs. For any PO $z$, we have $CODC^{\eta_{XZ},FI}_{z} = \varnothing$.

Computing CODCs as described in Section 2, suppose we reach a node $w \in W$. In general $CODC_w^{\eta_{XZ},FI} \neq \varnothing$.

Now consider the case that $W$ nodes are considered POs. The CODC computation proceeds in reverse topological order from nodes in $W$. As a result, $CODC_w^{\eta_{XW},FI} = \varnothing$.

In the CODC computation for $y_k$, both the above cases yield identical functions for Equation 2 (except for the last term). The last term for a node in the first case in general contains the last term for the same node in the latter case since $CODC_w^{\eta_{XZ},FI} \supseteq CODC_w^{\eta_{XW},FI} = \varnothing$. Hence, for an arbitrary $y_k$, we have $CODC_k^{\eta_{XW},FI} \subseteq CODC_k^{\eta_{XZ},FI}$ □

Note that in general, for a PO $z$, we may have $CODC_z^{\eta_{XZ}} = XDC^Z(X)$ (the external don't care of the output $z$). The proof in that case is similar. We still have $CODC_w^{\eta_{XW},FI} = \varnothing$, whereas in general $CODC_w^{\eta_{XZ},FI} \neq \varnothing$.

## 4.2 Case 2: Cutset used as Primary Input

Consider the network as partitioned in Figure 3. Let $R(V,X) = \prod_{i \in V} v_i \overline{\oplus} v_i(X)$. In other words, $R(V,X)$ is the relation between the $X$ and $V$ cuts. Let us define also $I = \exists_X(R(V,X))$, the image of the PIs on the $V$ cut. Note that $I \neq \varnothing$ in general.

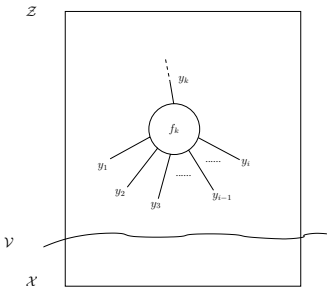


**Figure 3: Partition network 2**

THEOREM 4.2. *If the CODC of a node $y_k$ is computed in terms of cut nodes $V$ in the TFI of $y_k$, then $CODC_k^{\eta_{VZ},FI} \subseteq CODC_k^{\eta_{XZ},FI}$*

PROOF. In the case of computing the ACODC at $y_k$, we first compute $CODC_k^{\eta_{VZ}}$ as discussed in Section 2. Now we compute the image $I_1$ of this on the $V$ space and project this image back to the fanin space of $y_k$.

Now consider the case of computing the full CODC at $y_k$. We first compute $CODC_k^{\eta_{XZ}}$ in the normal manner, and then compute the image $I_2$ of this on the PI space. Projecting this back to the fanin space of $y_k$, we obtain $CODC_k^{\eta_{XZ},FI}$. Consider the projection $I_3$ of $I_2$ on $V$. We can write $I_3 = I_1 + \exists_X((R(V,X))(I_2(X)))$. Therefore, $I_3 \supseteq I_1$, which yields the result. □

## 4.3 Case 3: Cutset used as Primary Output and Primary Input

Consider a network with two cuts as shown in Figure 4.

THEOREM 4.3. *Suppose the CODC of a node $y_k$ is computed in terms of cut nodes $W$ in the TFO of $y_k$ and cut nodes $V$ in the TFI of $y_k$. Then $CODC_k^{\eta_{VW},FI} \subseteq CODC_k^{\eta_{XZ},FI}$*

PROOF. This follows directly from Theorems 4.1 and 4.2 since their proofs are orthogonal. □

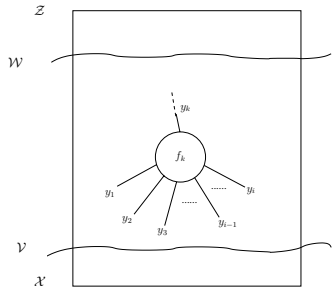


**Figure 4: Partition network 3**

In our experiments, we checked for the containment of our ACODC in the CODC for each node in the design, and verified the correctness of our implementation.

## 4.4 Implementation

In our implementation, we generate ACODCs by creating dynamic cutsets of the given Boolean network. We ensure that for any node in the computation, ACODCs are computed *exactly once*. The ACODC algorithm is shown in Algorithm 1.

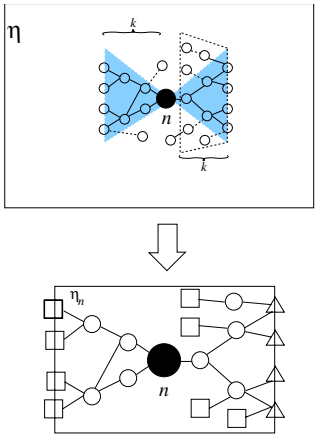


**Figure 5: Dynamic cutsets**

**Algorithm 1** Compute ACODC using dynamic cutset method

*Traverse $\eta$ in reverse topological order*
**for** (*each node $n \in$ network $\eta$*) **do**
  $\eta_n = extract\_subnetwork(n,k)$
  $ACODC(n) = compute\_acodc(\eta_n, n)$
  $optimize(n, ACODC(n))$
**end for**

For every node $n$ in the network $\eta$, $extract\_subnetwork(n,k)$ constructs a subnetwork $\eta_n$ rooted at $n$ such that if node $m \in TFO(n,k)$, then $m \in \eta_n$ and if node $p \in TFI(n,k)$, then $p \in \eta_n$. Here $k$ is a user-specified parameter, and $TFO(n,k)$ includes all nodes in the transitive fanout of node $n$ up to a forward depth $k$, and $TFI(n,k)$ includes all nodes in the transitive fanin of node $n$ up to a backward depth of $k$. Node $m \in \eta_n$ is designated as primary input of $\eta_n$ if

$FI(m \in \eta_n) = \varnothing$, and primary output if $FO(m \in \eta_n) = \varnothing$. $FI(m)$ and $FO(m)$ represent the immediate fanins and immediate fanouts of $m$ respectively. Next we include in our subnetwork all nodes $m \in TFI(v,k)$, where $v$ is a primary output of the subnetwork $\eta_n$. A sample subnetwork is shown in Figure 5. Note that we could also include in $\eta_n$, nodes $m \in TFO(v,k)$, where $v$ is a primary input of $\eta_n$. This is currently not implemented.

After constructing $\eta_n$, we run *compute_acodc*$(\eta_n, n)$. This subroutine performs the CODC computation for node $n$ using Equations 2 and 3. Note that the computation of CODCs of nodes $m \in \eta_n \mid m \in TFO(n)$ is not required, since a prior call to *compute_acodc*$(\eta_m, m)$ performed this computation. Therefore the only computations that are performed are the computations of edge CODCs for edges $e_{nm}$ where $m \in FO(n)$. We find the image of the resulting CODC of $n$ on the space of the primary inputs of $\eta_n$, and project this image back to the fanins of $n$ to get the ACODC of $n$. Note that in this method, ACODCs of any node are computed *exactly once*. Further these computations are on small subnetworks, resulting in a fast, robust computation. The ACODC of $n$ so obtained is a subset of the actual CODC of $n$ (by the result of Section 4.3) since the primary inputs and primary outputs of $\eta_n$ are nodes on some two cuts[1] of $\eta$.

The function $optimize(n, ACODC(n))$ runs Espresso [3] on the node $n$ with respect to its ACODC, after which the node $n$ is replaced by its minimized version.

The entire process is iterated over all the nodes $n$ of the network $\eta$ (in reverse topological order). In this way we obtain the ACODCs for all nodes in the network $\eta$. Note that the size of each subcircuit $\eta_n$ is limited by the depth parameter $k$. Hence by suitably choosing $k$, we ensure that the subcircuits are never too large and ACODCs can be computed quickly and with low memory utilization for extremely large circuits.

# 5. EXPERIMENTAL RESULTS

We have implemented our technique in SIS [15]. For our experiments we have used circuits from the *mcnc91* and *itc99* benchmarks. Our experiments consisted of reading in a design and computing ACODCs for the circuit using our dynamic cutset approach. Next we optimized the circuit nodes with respect to their ACODCs and then ran *sweep* to remove the redundant nodes after optimization. We compared our results with the CODCs generated in SIS by running *full_simplify* followed by *sweep*. In both cases, we used the -snocomp option. The experiments were run on an IBM IntelliStation running Linux with a 1.7 GHz Pentium-4 CPU and 1 GB of RAM.

Table 1 reports results for large circuits on which *full_simplify* failed to complete due to memory blowup resulting from large BDDs. Column 1 lists the circuits used while columns 2 and 3 list the number of nodes and literals in each circuit (before optimization) respectively. Columns 4 and 6 list the nodes and literals respectively after they have been optimized with our method (with $k = 4$). Column 5 lists the percentage reduction in the number of nodes after optimization and column 7 lists the percentage reduction in the number of literals. Columns 8 and 9 report the runtime and the peak number of ROBDD nodes for the computation respectively. On average, a healthy 19% node reduction and about 10% literal reduction is achieved with our technique. The runtime never exceeded 2 minutes, and the memory utilization was below 100K ROBDD nodes. We expect that with larger values of $k$, the quality of these results may show improvement.

| Circuit | Original | | Our Method | | | | | |
|---|---|---|---|---|---|---|---|---|
| | nodes | lits | nodes | % | lits | % | time (s) | mem |
| C6288 | 2416 | 4800 | 2310 | 4.39 | 4623 | 3.69 | 3.65 | 1022 |
| C7552 | 3466 | 6098 | 2056 | 40.68 | 4506 | 26.15 | 6.11 | 9198 |
| b14 | 9768 | 18917 | 8093 | 17.15 | 17001 | 10.12 | 117.60 | 105582 |
| b14_1 | 6570 | 12886 | 5254 | 20.03 | 11654 | 9.56 | 19.04 | 50078 |
| b20 | 19683 | 38213 | 16014 | 18.64 | 34895 | 8.68 | 65.39 | 69496 |
| b20_1 | 13900 | 27074 | 11265 | 18.96 | 24652 | 8.95 | 39.34 | 34748 |
| b21 | 20028 | 38993 | 16440 | 17.91 | 35689 | 8.48 | 66.37 | 65408 |
| b21_1 | 13899 | 27164 | 11458 | 17.56 | 24595 | 9.45 | 38.32 | 34748 |
| AVG | | | | 18.77 | | 9.49 | | |

**Table 1: Results for Large Circuits**

Table 2 provides a comparison between our ACODC method and *full_simplify* (which is implemented in SIS [15]). Column 1 lists the experimental circuits used. Two measures of effectiveness are defined to compare our results with those generated by *full_simplify*. Effectiveness measure #1 compares the number of minterms in the CODCs generated by our method with those in *full_simplify*.

$$effectiveness\ \#1 \equiv \frac{\sum_n |ACODC(n)|}{\sum_n |CODC(n)|}$$

Here $|ACODC(n)|$ represents the number of minterms in the CODC of node $n$ generated by our method, and $|CODC(n)|$ represents the number of minterms in the CODC of node $n$ generated by *full_simplify*.

Effectiveness measure #2 compares the number of nodes for which ACODCs and CODCs are identical

$$effectiveness\ \#2 \equiv \frac{\#equal}{\#total}$$

where $\#equal$ represents the number of nodes for which the ACODCs were identical as the CODCs produced by SIS, and $\#total$ represents the total number of nodes in the circuit.

Both measures of effectiveness are expressed as percentages in Table 2. Columns 2 and 3 report the value of the effectiveness measure #1 for depth $k = 4$ and 6 respectively, whereas columns 4 and 5 provide the value of effectiveness measure #2 for the same levels of depth. Column 6 gives the original number of literals in the circuits considered. Column 7 gives the number of literals after running *full_simplify* whereas column 8 gives the percentage reduction in literal count after *full_simplify*. Columns 9 and 11 give the number of literals after optimizing the circuits with our method for depth $k = 4$ and 6 respectively. Finally, columns 10 and 12 give the percentage reduction in literal count for the same levels of depth. Columns 13, 14 and 15 compare the run-times for *full_simplify* against our method for $k = 4$ and 6 respectively, while columns 16, 17 and 18 compare peak number of ROBDD nodes for *full_simplify* against our method with $k = 4$ and 6 respectively.

When compared to the circuits on which *full_simplify* does complete, we get a 23% reduction in literal count as compared to 29% by *full_simplify*. The effectiveness measures

[1]Actually, these nodes belong to some subnetwork in general, but the addition of nodes (to the PIs and POs of the subnetwork) to construct cuts would not change the computation.

| circuit | eff # 1 | | eff #2 | | literals | | | | | | | run-time (s) | | | memory | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 6 | 4 | 6 | orig | fs | % | new4 | % | new 6 | % | fs | 4 | 8 | fs | 4 | 6 |
| C1355 | 98.04 | 98.04 | 98.34 | 98.34 | 1032 | 984 | 4.65 | 992 | 3.88 | 992 | 3.88 | 39.28 | 1.66 | 1.80 | 312732 | 3066 | 3066 |
| C1908 | 81.56 | 84.69 | 87.13 | 88.89 | 1497 | 941 | 37.14 | 1038 | 30.66 | 1026 | 31.46 | 54.68 | 2.40 | 2.50 | 106288 | 3066 | 3066 |
| C2670 | 94.13 | 94.13 | 86.79 | 86.79 | 2043 | 1240 | 39.30 | 1370 | 32.94 | 1370 | 32.94 | 11.77 | 4.20 | 4.66 | 172718 | 4088 | 4088 |
| C432 | 71.43 | 71.43 | 92.81 | 92.81 | 372 | 298 | 19.89 | 322 | 9.95 | 322 | 9.95 | 4.91 | 1.25 | 1.45 | 289226 | 6132 | 6132 |
| C499 | 98.56 | 98.56 | 97.34 | 97.34 | 616 | 568 | 7.79 | 572 | 6.50 | 572 | 6.50 | 2.41 | 1.20 | 1.31 | 99134 | 8176 | 8176 |
| C880 | 80.00 | 84.44 | 94.56 | 95.77 | 703 | 625 | 11.10 | 635 | 9.67 | 630 | 10.38 | 2.05 | 0.70 | 0.72 | 73584 | 2044 | 3066 |
| C3540 | 85.43 | 97.81 | 84.15 | 97.51 | 2934 | 1943 | 33.78 | 2145 | 26.89 | 2100 | 28.42 | 835.64 | 25.25 | 27.45 | 321746 | 8176 | 8176 |
| dalu | 78.00 | 79.86 | 75.78 | 79.55 | 3588 | 2164 | 39.68 | 3240 | 9.70 | 3240 | 9.70 | 210.09 | 6.23 | 7.12 | 499758 | 11242 | 12264 |
| i10 | 99.34 | 99.34 | 85.45 | 85.45 | 5376 | 3787 | 29.55 | 3899 | 27.47 | 3899 | 27.47 | 332.22 | 8.56 | 9.21 | 507934 | 4088 | 4088 |
| b01_C | 92.68 | 92.68 | 83.33 | 83.33 | 80 | 44 | 45.00 | 45 | 43.75 | 45 | 43.75 | 0.03 | 0.05 | 0.05 | 1022 | 1022 | 1022 |
| b03_C | 68.89 | 75.56 | 87.23 | 89.43 | 254 | 101 | 60.00 | 154 | 39.37 | 149 | 41.34 | 0.19 | 0.20 | 0.20 | 1022 | 1022 | 1022 |
| b04_C | 63.42 | 63.42 | 85.35 | 85.35 | 1267 | 862 | 31.96 | 904 | 28.65 | 904 | 28.65 | 12.15 | 1.47 | 1.66 | 117530 | 2044 | 3066 |
| b05_C | 74.70 | 84.85 | 76.38 | 88.02 | 1858 | 1007 | 45.80 | 1580 | 14.96 | 1580 | 14.96 | 24.50 | 2.43 | 2.50 | 25550 | 5110 | 5110 |
| b06_C | 92.11 | 92.11 | 87.10 | 87.10 | 83 | 40 | 51.80 | 45 | 45.78 | 45 | 45.78 | 0.04 | 0.04 | 0.04 | 1022 | 1022 | 1022 |
| b07_C | 69.52 | 81.90 | 91.02 | 95.21 | 749 | 660 | 11.88 | 666 | 11.08 | 666 | 11.08 | 4.04 | 0.84 | 0.86 | 26572 | 2044 | 2044 |
| b08_C | 98.33 | 98.33 | 96.09 | 96.09 | 306 | 276 | 9.80 | 277 | 9.48 | 277 | 9.48 | 0.30 | 0.30 | 0.30 | 2044 | 2044 | 2044 |
| b09_C | 79.00 | 95.00 | 83.04 | 90.18 | 277 | 108 | 61.00 | 155 | 44.04 | 150 | 45.85 | 0.37 | 0.26 | 0.27 | 1022 | 1022 | 1022 |
| b10_C | 80.65 | 83.87 | 92.90 | 94.19 | 353 | 309 | 12.39 | 312 | 11.05 | 312 | 11.05 | 0.40 | 0.30 | 0.32 | 2044 | 1022 | 1022 |
| b11_C | 83.44 | 85.94 | 89.50 | 91.65 | 1378 | 1065 | 22.71 | 1180 | 14.36 | 1180 | 14.36 | 9.97 | 0.26 | 0.34 | 22484 | 2044 | 2044 |
| b12_C | 67.10 | 79.04 | 87.17 | 90.92 | 1967 | 1494 | 24.05 | 1853 | 5.80 | 1853 | 5.80 | 81.72 | 3.23 | 3.55 | 15330 | 4088 | 4088 |
| b13_C | 65.08 | 65.08 | 91.53 | 91.53 | 558 | 453 | 18.81 | 499 | 10.57 | 499 | 10.57 | 0.28 | 0.20 | 0.21 | 2044 | 1022 | 1022 |
| AVG | 81.97 | 85.52 | 87.85 | 89.52 | | | 28.36 | | 22.34 | | 22.82 | | 0.0365 | 0.0412 | | 0.0283 | 0.0322 |

**Table 2: Results for Medium Circuits**

show an increase with $k$, especially measure #1. However, the well established figure of merit of technology independent optimization, literal count, shows little variation with $k$. As a result, our recommendation is to use smaller values of $k$. Our method, with a value of $k = 4$ gives about 80% of the literal reduction of *full_simplify*, with an average runtime and memory utilization which is 25× and 33× better than that of *full_simplify*.

# 6. CONCLUSIONS

We have presented a technique to compute approximate CODCs (ACODCs) for any network by dynamically extracting substantially smaller sub-networks from the original one and computing the CODCs on the smaller sub-networks. We formally showed that the ACODCs thus obtained are subsets of the actual CODCs. Since the size of the extracted sub-network can be controlled by the user-specified depth parameter $k$, we can compute CODCs for arbitrarily large circuits in our technique. By keeping $k$ relatively small, we can ensure that the BDDs which are built during the CODC computation (for image computation) never exceed a certain size and do not cause memory overflow. Our method computes the ACODC of any node *exactly once*, resulting in a fast and efficient don't care computation.

Our results show that we can obtain very reasonable optimization of large circuits *for which CODCs can't be computed by the resident method in SIS, full_simplify*. This optimization results in a reduction of 19% in the node count and a reduction of 9.5% in literal count. Also, for smaller circuits which can be optimized by the resident technique in SIS (*full_simplify*), our technique obtains an average reduction of 23% in literal count as compared to a 29% reduction by *full_simplify*. Our technique respectively demonstrates average runtimes and memory utilization about 25× and 33× better than $full\_simplify$.

# 7. REFERENCES


[1] R. Brayton. Compatible output don't cares revisited. In *IEEE/ACM International Conference on Computer Aided Design*, pages 618–623, Nov 2001.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: A multiple-level logic optimization system. *IEEE Trans. on CAD/ICAS*, CAD-6(6):1062–1082, Nov 1987.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[4] R. E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, August 1986.

[5] M. Damiani and G. D. Micheli. Observability don't care sets and boolean relations. In *IEEE/ACM International Conference on Computer Aided Design*, Nov 1990.

[6] S. Dey, F. Brglez, and G. Kedem. Circuit partitioning and resynthesis. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 29.4/1 –29.4/5, May 1990.

[7] S. Hassoun, editor. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Nov 2001.

[8] J. C. Limqueco and S. Muroga. SYLON-REDUCE: an MOS network optimization algorithms using permissible functions. In *Proceedings, IEEE International Conference on Computer Design*, pages 282–285, Sept 1990.

[9] J. C. Limqueco and S. Muroga. Optimizing large networks by repeated local optimization using windowing scheme. In *IEEE International Symposium on Circuits and Systems, ISCAS*, volume 4, pages 1993–1996, May 1992.

[10] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *IEEE International Conference on Computer-Aided Design*, pages 6–9, November 1998.

[11] P. C. McGeer and R. K. Brayton. The observability don't care set and its approximations. In *IEEE International Conference on Computer Design*, Sept. 1990.

[12] A. Mishchenko and R. Brayton. Simplification of non-deterministic multi-valued networks. In *IEEE/ACM International Conference on Computer Aided Design*, pages 557–562, Nov 2002.

[13] H. Savoj and R. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990.

[14] H. Savoj, R. Brayton, and H. Touati. Extracting local don't cares for network optimization. In *IEEE/ACM International Conference on Computer Aided Design*, pages 514–517, Nov 1991.

[15] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[16] T. Stanion and C. Sechen. Maximum projections of don't care conditions in a boolean network. In *IEEE/ACM International Conference on Computer Aided Design*, pages 674–679, Nov 1993.

[17] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *IEEE International Conference on Computer-Aided Design*, November 1990.