Node Mergers in the Presence of Don't Cares

Stephen Plaza, Kai-hui Chang, Igor Markov, and Valeria Bertacco

CSE-TR-521-06

July 06, 2006

THE UNIVERSITY OF MICHIGAN
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2121
USA

Node Mergers in the Presence of Don't Cares

Stephen Plaza, Kai-hui Chang, Igor Markov, and Valeria Bertacco
{splaza, changkh, imarkov, valeria}@umich.edu

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2121

July 06, 2006

Abstract

Merging equivalent nodes in a circuit is a synthesis technique useful for reducing area and improving equivalence checking. Recently proposed algorithms that determine node equivalence are unable to exploit observability don't cares (ODC) and therefore miss several merging opportunities. Current strategies for extracting don't care information for identifying additional node mergers restrict the types of don't cares computed because of computational expense.

We develop an efficient framework that merges nodes by exploiting ODCs through simulation and SAT. Specifically, the framework integrates (1) a novel strategy for representing ODC information during logic simulation, (2) a fast ODC-aware simulator, (3) an ODC-aware equivalence checker, and (4) progressive refinement of simulation vectors. Our results indicate that ODCs enable many node mergers with up to 30% gate reduction on unoptimized circuits and 23% area reduction after synthesis.

1 Introduction

Merging equivalent nodes is a synthesis technique that directly results in the reduction of logic in the circuit. For networks that are constructed using BDDs [3], a candidate for merger can be identified and verified in $O(1)$-time. However, the size of BDDs is prohibitive for many practical designs and does not scale well.

To efficiently identify node-merging opportunities in larger designs, the work in [12, 15] uses a combination of SAT and simulation. Candidate nodes for merging can be identified by checking whether their outputs correspond to random simulation patterns applied to the design, and their equivalence can be verified using SAT [9]. To avoid related unsuccessful equivalence checks, the work in [15] reuses counterexamples produced by such checks. It refines node signatures by simulating these additional input patterns.

The work in [15] is useful in equivalence checking. By simplifying and proving the equivalence of internal nodes incrementally, the equivalence between the outputs of two designs can be more easily ascertained.

Using simulation and SAT to perform node mergers implicitly takes into account satisfiable don't cares (SDCs) because unsatisfiable combinations never arise during simulation. However, observability don't cares (ODCs) are not exploited in [15], where two nodes can be merged only when one of them has just been constructed but its downstream nodes have not. ODCs occur when node values for certain input patterns are irrelevant to the outputs of the design. By taking into account ODCs, additional node mergers will be possible.

Several algorithms for efficiently generating ODC information to be used in network simplification and improving the quality of SAT solvers have been developed [7, 16, 17, 19]. Techniques that determine ODCs often restrict the computation to a subset of them [19] or consider small windows [17] of the circuit. Also, these algorithms generate don't care information that may be unnecessary for finding node mergers.

In this paper, we introduce a framework that uses simulation vectors and downstream logic to identify potential mergers in the presence of ODCs. Unlike previous approaches, we avoid expensive computation unless fast simulation indicates a potential merger. In addition, SAT computation is limited to portions of the circuit that are necessary to verify a merger. By using simulation, we can find potential mergers in large circuits with high-accuracy and allow the complexity of the verification procedure to explicitly limit what is examined.

Our first insight involves maintaining signatures storing don't care values in a manner that allow potential mergers to be identified quickly through efficient signature manipulations. Second, we develop a fast simulator that identifies don't care situations to produce these signatures. Next, we verify a merger indicated by the signatures using a SAT engine that considers downstream logic. Instead of considering all downstream logic we examine only a fraction of it and incrementally increase the logic considered until the merger can be verified. Through incremental SAT-solving, this process avoids miters that are unnecessarily large. In many cases, only a few layers of logic are needed to prove or disprove equivalences. If the miter becomes too deep, the verification can be aborted and the merger opportunity rejected.

Because the identification of ODCs relies on the quality of signatures, we reuse all counterexamples generated by SAT calls to prevent spurious merger candidates. We note that this technique is only superficially similar to that used in [15] because we focus on downstream logic, which is unavailable in their approach.

The work closest to ours is that on and-inverter graphs (AIGs) [12] and functionally-reduced AIGS (FRAIGs) [15] used to efficiently represent logic networks. However, there are several key differences. In addition to handling ODCs by considering downstream logic, our work does not assume a technology-independent representation such as an AIG and can operate on an arbitrarily mapped netlist. In particular, it is applicable to post-placement logic optimization, whereas AIG-based techniques are most likely not.

Section 2 gives background on signatures, SAT, and recent advances in ODC computation. Section 3 explains a representation scheme for ODCs along with an efficient simulator. Section 4 describes the SAT engine that verifies the merger. Section 5 elaborates on our strategy for generating dynamic simulation patterns. Finally, in Section 6, results are given that show the number of ODC-based mergers performed for several benchmarks.

2 Background

The following discusses work in signature-based equivalence checking [12, 15] which we extend to handle ODC-based equivalence. Then, prior strategies for computing ODCs [16, 19] are described.

2.1 Simulation and Satisfiability

A given node F in a Boolean network can be characterized by its signature, S_F, for K input vectors $X_1 \cdots X_K$.

Definition 1 $S_F = \{F(X_1), \ldots, F(X_K)\}$ *where* $F(X_i) = \{0,1\}$ *indicates the output of F for input* X_i*.*

Typically, random simulation is used to generate vectors X_i and derive signatures for each node in a design. For a network with N nodes, the time complexity of generating signatures for the whole network is $O(NK)$. Nodes can be distinguished by the following formula: $S_F \neq S_G \Rightarrow F \neq G$.

Signatures can be easily created and manipulated by taking advantage of bit-parallel operations. Therefore, equivalent signatures can be used to efficiently identify potential node equivalences in a circuit [12]. A hash index can be derived for each signature and equivalent signatures can be discovered using an $O(1)$-time hash table lookup. Since $S_F = S_G$ does not imply that $F = G$, this potential equivalence must be verified using SAT. In [12], dynamic input vectors are generated from the counter-examples derived from SAT checks that prove $F \neq G$. The dynamic input vectors improve the quality of the signatures by limiting situations where $S_F = S_G$ despite $F \neq G$.

The efficiency of the frameworks in [12, 15] is dependent on the underlying engines for formally verifying the equivalence of nodes with equivalent signatures. Recent advances in SAT such as learning, non-chronological backtracking, and watch literals [18, 22] have made SAT a more scalable alternative to BDDs in applications like equivalence checking. The equivalence of two nodes, F and G, in a network can be determined by constructing an XOR-based miter [2] between them and asserting the output to 1 as shown in the following formula:

$$F = G \Leftrightarrow \forall i\, F(X_i) \oplus G(X_i) \neq 1 \tag{1}$$

where $\bigcup_i X_i$ is the set of all possible inputs.

2.2 Observability Don't Cares

Figure 1 shows examples of satisfiability don't cares (SDCs) and observability don't cares (ODCs). An SDC arises when certain inputs are not possible. For example, the combination of $x = 1$ and $y = 0$ cannot occur for the circuit shown on the left in Figure 1. SDCs are implicitly handled when using SAT for equivalence checking because this combination cannot occur for any satisfying assignment. ODCs occur when the value of an internal node does not effect the outputs of the design. For the circuit on the right, when $a = 0$ and $b = 0$, the value F is a don't care.

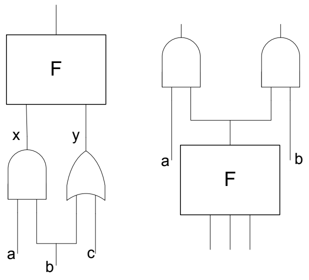

Figure 1: The left circuit shows examples of SDCs, and the right circuit shows examples of ODCs.

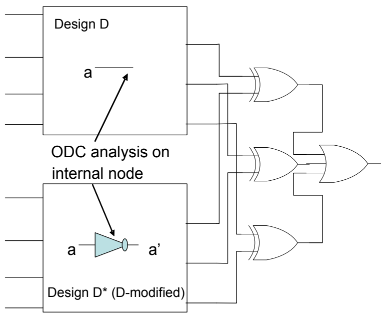

Figure 2: Identifying an ODC for an internal node a in a network by constructing a miter for each output and inverting a in a modified copy of the network. The set of inputs where the miter is 1 corresponds to the care-set of that node.

Figure 2 shows a strategy for identifying ODCs for a node a. First, the design D is copied and a is inverted in the design D^*. Then, miters are constructed between the outputs of the two designs and the care set, denoted as $C(a)$, can be derived as follows:

$$C(a) = \bigcup_i X_i \; if \; D(X_i) \neq D^*(X_i) \tag{2}$$

A SAT solver can derive C by adding constraints called **blocking clauses** that prevent the solver from re-deriving previous satisfying assignments to the miter in Figure 2 [16]. The ODC of a is therefore:

$$ODC(a) = \bigcup_i X_i - C(a) \tag{3}$$

This approach can be computationally expensive and therefore unscalable particularly when the miters are far from a. To address this problem, in [16], small windows of the circuit are examined that reduces computation but produces smaller don't care sets. Also, other strategies derive a subset of ODCs called compatibility ODCs (CODCs) that are easier to compute because of their convenient properties [19, 20]. CODCs have the nice property that optimizations involving one node's CODCs do not effect the CODCs of another. In our work, we achieve scalability by identifying ODCs by simulation without requiring small windows to reduce computation. Furthermore, we are not limited to CODCs because in our framework we examine one node at a time, making compatibility unnecessary.

3 Finding ODC Equivalences

Traditionally, a node merger can occur between a and b when they are functionally equivalent. We now define node mergers between a and b in the presence of ODCs when a is **ODC equivalent** to b.

Definition 2 *a is ODC equivalent to b if $ONSET(a) + ODC(b) = ONSET(b) + ODC(b)$.*

When a is ODC equivalent to b, a merger between a and b means that a can be substituted for b. Because the ODCs of only one node are considered, ODC equivalence is not symmetric as b might not be ODC equivalent to a.

Our strategy of merging nodes in the presence of ODCs first uses signatures to find candidate ODC equivalences. Unlike [15], we need to consider downstream logic. Also, identifying ODC merger candidates with signatures cannot be done with $O(1)$-time signature hashing because the candidate merger depends on which node's ODCs are considered.

In the following, we describe a strategy for encapsulating ODC information in signatures so that they can be sorted and searched efficiently to minimize the cost of identifying merging candidates. In addition, we develop a simulator that generates ODC information by considering downstream logic with complexity comparable to non-ODC signature generation without considering downstream logic.

3.1 Reasoning About ODCs in Signatures

Each node in the design maintains a signature S as defined in Definition 1. In addition to this, an **ODC-mask** S^* is maintained:

Definition 3 *For node f,*
$S^*_f = \{X_1 \notin ODC(f), \ \ldots \ , X_K \notin ODC(f)\}$. When an input vector X_i is in the set $ODC(f)$, that bit position is denoted by a 0.

Set operations can be efficiently executed on these signatures through bit-wise manipulations. The following shows how the $\subseteq$ relation is defined using the signatures of two nodes, S_a and S_b:

Definition 4 *$S_b \subseteq S_a$ if and only if $S_b | S_a = S_a$ where $|$ represents bit-wise OR.*

Figure 3 shows a circuit with signatures for each node and a mask for node c. Each ODC for a node is marked by a 0 in the ODC-mask. In our framework, we show the flexibility of a given node by maintaining **upper-bound** S^{hi} and **lower-bound signatures** S^{lo}.

Definition 5 *$S^{lo} = S \& S^*$ where & represents bit-wise AND and $S^{hi} = S | \neg(S^*)$.*

S^{lo} and S^{hi} of node f correspond to range of Boolean function $[f^{lo}, f^{hi}]$ that are ODC-equivalent to f because the differences between the range of functions are a subset of $ODC(f)$.

After simulation populates the different signatures, merger candidates can be found. In the example in Figure 3, after the first four simulation, node b is depicted as a **candidate** for ODC-equivalence with c.

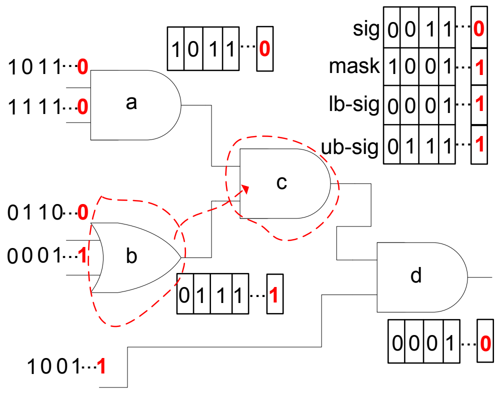

Figure 3: An example showing how ODCs are represented in a circuit. For clarity, the example only shows ODC information for node c. The other internal nodes show only signatures S. When examining the first four simulation, node b is a candidate for merging with c. Further simulation indicates that an ODC merger is not possible.

Definition 6 *Node b is a candidate for ODC-equivalence with node c if and only if $(S_b \oplus S_c) \subseteq \neg S_c^*$. This can be re-expressed as $S_b \subseteq [S_c^{lo}, S_c^{hi}]$, or S_b is contained within the signature range defined by S_c^{lo} and S_c^{hi}.*

Therefore, by simple application of S_c^*, it can be determined that b is an ODC-equivalent candidate with c. However, in this example, further simulation reveals that this candidate equivalence is incorrect. Similar to Definition 2, if b is an ODC-equivalent candidate with c, it does not imply that c is an ODC-equivalent candidate with b.

Unlike checking for equivalence with signatures, $O(1)$-time complexity hashing cannot be used to identify ODC-equivalence candidates. Each node needs to apply its mask to every other node to find candidates. The result is that for N nodes, finding all ODC-equivalence candidates for the design requires $O(N^2K)$-time complexity assuming that applying a mask is an $O(K)$-time operation. Although we do not implement a better worst-case complexity algorithm, we now discuss a technique that reduces computation in practice.

First, all of the signatures, S, in the design are sorted by the value obtained by treating each K-bit signature as a single K-bit number. This operation requires $O(NKlgN)$-time. Then, for a given node c, candidates can be found by performing two binary searches with S_c^{lo} and S_c^{hi} to obtain a lower and upper bound on the sorted S, an $O(KlgN)$-time operation. The following formula defines the set of signatures S_x that will be checked for candidacy: $\bigcup_x S_x \; if \; num(S_c^{lo}) \leq num(S_x) \leq num(S_c^{hi})$ where *num* represents the K-bit number of the signature. This set is linearly traversed to find any candidates according to Definition 6.

3.2 Generating ODC Information

Generating ODC masks S^* efficiently is integral to maintaining the scalability of our framework. Whereas each node's S can be computed from its immediate fanin, computing each node's S^* often requires all downstream logic.

The S^* can be computed for each node by determining the care-set using Equation 2 where the X_i are the random simulation vectors. This approach requires circuit simulation of each X_i for each node. For K simulation vectors and N nodes the time-complexity is $O(N^2K)$. Although the simulation can be confined to just the fanout cone of the node, the procedure is computationally expensive.

Approximate ODC Simulator: we now describe our approximate ODC simulator whose complexity is only $O(n'K)$ where n' is the number of nets or wires in the design. The algorithm for generating masks using the approximate simulator is shown in Figure 4.

```
void gen_odc_mask(Nodes $N$){
  set_output_$S^*$($N$);
  reverse_levelize($N$);
  for_each $node \in N$
  {
    $node.S^* = 0$;
    for_each $output$ in $node$.fanout
    {
      $temp\_S^*$ = get_local_ODC($node$, $output$);
      $temp\_S^* = temp\_S^*$ & $output.S^*$;
      $node.S^* \mid= temp\_S^*$;
    }
  }
}
```

Figure 4: Algorithm for efficiently generating ODC masks for each node.

The function `set_output_`S^* initializes the mask of nodes directly connected to the input of a latch or primary output to all 1s. The nodes are then ordered and traversed in reverse topological order as generated by `reverse_levelize`. The immediate fanout of each *node* is then examined. The function `get_local_ODC` performs ODC analysis for every simulation vector for *node* as defined by Equation 2 except only the sub-circuit defined by *node* and *output* is considered. This local ODC mask is bitwise-ANDed with the *output*'s S^* and is subsequently ORed with the node's S^*.

The algorithm requires only a traversal of all the nets given by the two for_each loops and the K simulation vectors considered for `get_local_ODC`, resulting in the $O(n'K)$ complexity. Intuitively, each fanout node is another constraint that reduces the number of ODCs in the mask hence the bitwise-OR. The flexibility is increased by the local ODCs that are generated across each fanout node hence the bitwise-AND.

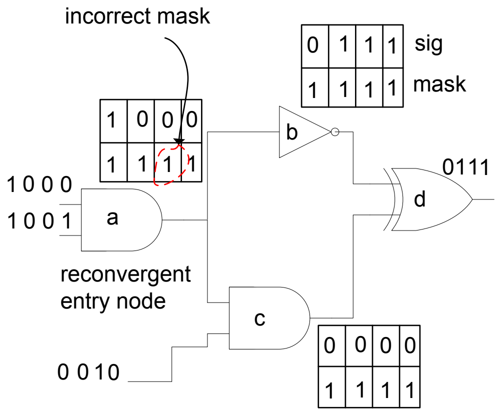

Figure 5: The ODC information produced by approximate ODC simulation. Sometimes reconvergence can cause ODC simulation to produce incorrect ODC masks. S^* and S are shown for the internal nodes, and only S are shown for the inputs and outputs.

Accuracy of Approximate Simulator: because of reconvergence in logic, it is possible for the algorithm in Figure 4 to incorrectly produce Os (false positives) or 1s (false negatives) in S^*. For the example in Figure 5, node a misses a don't care (false negative) in the third bit of S^*_a. Notice that node b and c do not have any ODCs and no local ODCs exist between a and b or a and c, resulting in no ODCs being detected

Table 1: Efficiency/quality of the approximate ODC simulator.

bench	runtime(s)			#cands	%pos	%neg
	sim	simodc	approx			
ac97_ctrl	1	6	1	63758	0.0	0.0
aes_core	2	79	1	315917	0.1	0.0
des_perf	9	410	7	296095	0.0	0.0
ethernet	4	76	2	8852009	0.3	0.8
mem_ctrl	1	119	1	867145	1.0	1.4
pci_bdge32	1	28	1	1158654	0.2	0.4
spi	0	39	0	156291	0.0	3.1
systemcaes	1	48	1	285189	0.2	0.2
systemcdes	0	24	0	5288	2.8	0.7
tv80	1	130	1	1348277	1.5	9.0
usb_funct	1	11	1	1685374	2.2	1.8
wb_conmax	3	69	4	1904773	0.0	0.0

by the approximate simulator. However the reconvergence of downstream logic makes the third value of node a a don't care. Our experiments show that these situations happen infrequently.

Performance of Approximate Simulator: in Table 1, the quality of the approximate ODC simulator is assessed. The first column indicates the benchmarks examined. The second column, *sim*, gives the time required to generate only signatures S for each node. We use this as a baseline to assess the cost of generating masks. The third column, *simodc*, shows the time required to generate S^* for each node using Equation 2. The fourth column, *approx*, shows the time to compute S^* using the approximate simulator. The last few columns show the number of ODC-equivalence candidates identified by the approximate simulator and the percent of **false positives** and **negatives** relative to the signatures generated by *simodc*. A **false positive** (**false negative**) in this context refers to a situation where *approx* confirms (rejects) an ODC-equivalence candidate that *simodc* rejects (confirms).

The results indicate that the approximate simulator is comparable to that of *sim* and is much faster than *sim_odc*. In addition, the number of false positives and negatives is typically a small percentage of the number of opportunities identified. These results were generated by running 2048 random simulation vectors.

The number of candidate equivalences is often much greater than the number of equivalences possible. In Section 5, a strategy that prunes the number of candidate equivalences significantly will be discussed.

4 Verifying ODC Equivalence

In the following sections, we discuss how to efficiently prove ODC-equivalence candidates. We first describe a simple technique for proving ODC-equivalence with SAT. We improve this technique with an adaptive SAT framework.

4.1 Output Equivalence Checker

Figure 2 shows how ODCs can be identified for a given node in a network. We can prove whether b is ODC equivalent to a in a similar manner. Instead of using a' in the modified circuit D^*, b is substituted for a and miters are constructed at the outputs. If the care-set determined by Equation 2 is null, b is ODC equivalent to a and a merging opportunity exists. The whole care-set does not need to be derived as a single satisfiable solution proves non-equivalence.

4.2 Moving-Dominator Equivalence Checker

For large designs, proving ODC equivalence could be prohibitive because all downstream logic is considered. We introduce a dynamic SAT framework that attempts to determine ODC-equivalence by considering a small portion of downstream logic.

Consider Figure 6 where b is a candidate of ODC equivalence with a. If a miter is constructed across a and b instead of the primary outputs as shown in part a), a set of differences between a and b that result in satisfying assignments is given by $DIFFSET(a,b) = ONSET(a) * OFFSET(b) + OFFSET(a) * ONSET(b)$. A satisfying solution here indicates the non-equivalence for the given section of logic considered. If the satisfying solution is simulated for the remaining downstream logic and discrepancies between the two circuits exist at the primary outputs, then non-equivalence for the whole design is proven. If the $DIFFSET$ is null, equivalence is proven.

However, if a and b are very different, the $DIFFSET$ could result in a prohibitive amount of simulation. To reduce the size of the $DIFFSET$, we construct miters further from the merger site while reducing the amount of downstream logic considered. We introduce the notion of a **dominator set** to define where we place the miters.

Definition 7 *The dominator set for node-a is a set of nodes in the circuit such that every path from node-a to a primary output contains a member in the dominator set and where for each dominator member there exists at least one path from node-a to a primary output that contains only that member. Multiple dominator sets can exist for a given node.*

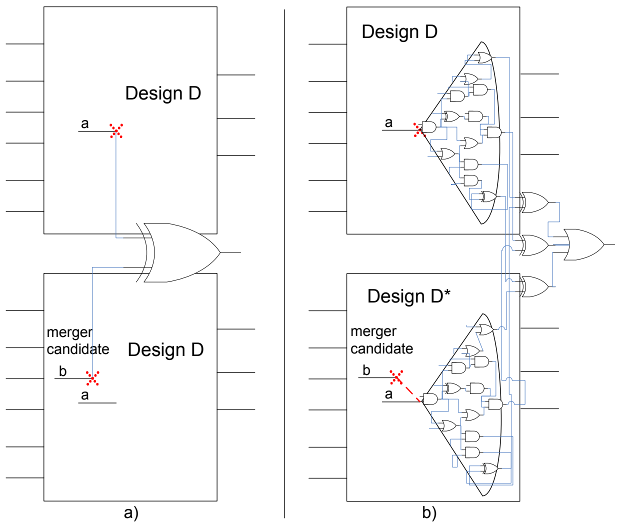

Figure 6: An example that shows how to prove that b is ODC-equivalent to a. a) A miter is constructed between a and b to find test vectors that are incompatible. b) A dominator set can be formed in the fanout cone of a and miters can be placed across the dominators.

In part b) of Figure 6, we show miters constructed for a dominator set of a. Dominator sets close to the candidate merger will result in less complex SAT instances but potentially more downstream simulation to check whether satisfying assignments prove non-equivalence. We devise a strategy that dynamically moves the dominator set closer to the primary outputs depending on the number of satisfying assignments generated. Our moving dominator algorithm is outlined in Figure 7.

In addition to the incremental SAT solver that finds satisfying assignments for a given dominator set, we also maintain a checker SAT instance with the dominator set being the primary outputs. The purpose of

```
bool odc_equivalent(a, b){
  current_dom = {a};
  while(dom_Sat(miter(current_dom, a, b)) == SAT){
    if(checker_Sat(satisfying_solution){
      return false;
    }
    else{
      add_blocking_clause;
      if(threshold_reached){
        current_dom = calculate_new_dom();
      }   }
  }
  return true;
}
```

Figure 7: Algorithm for determining the equivalence of two nodes in the presence of ODCs.

the checker SAT instance is to check whether satisfying assignments for a given dominator set propagate to the outputs. If the checker solver finds a conflict, a blocking clause is generated and sent to the other SAT instance to potentially prevent similar satisfying solutions from being derived.

The moving dominator algorithm starts by deriving a dominator set that is close to the merger site given by `current_dom`. Then, `dom_SAT` solves an instance where miters are placed across the current dominator set. An UNSAT solution means ODC equivalence and the procedure exits. Otherwise, the satisfying solution is checked in `checker_SAT` and an UNSAT solution here results in a blocking clause and potentially a new dominator set as determined by `calculate_new_dom`. `calculate_new_dom` extends the dominator set by one level of logic. The threshold for changing the dominator set was experimentally determined.

5 Improving Simulation Vectors

The previous sections were concerned with efficiently identifying ODC equivalence candidates and proving these candidate mergers. In this section, the focus is on ensuring that SAT checks required are minimized while maximizing the number of mergers that can be exploited in a design.

Table 1 indicates that better simulation is needed to reduce the number of potential merger candidates. To improve upon this, we generate an input vector based on the SAT call that proves non-ODC-equivalence. The satisfying assignment preventing the merger is added to the set of input vectors and new signature information is derived for each node. ODC equivalence candidates that are also false involving nearby nodes can benefit from the additional input vector and hence we immediately update every nodes' S and S^*.

When a merger is performed, the fanin and fanout cone of the merger site will be effected requiring that the corresponding signatures be updated. However, since signatures are used to find candidates that are later proven by a SAT solver, incorrect signatures can never lead to an incorrect merger and updates are not necessary. We have found experimentally that examining nodes for mergers in reverse topological order tends to reduce the flexibility in the circuit more than examining in topological order. Hence, we favor topological traversals.

Table 2: Gate reductions and performance of the ODC merging algorithm on unoptimized benchmarks. The algorithm has a timeout of 5000 seconds and is denoted by ☹.

benchmarks	#gates	time(s)	#merge	%area_reduct
i2c	1037	3	39	4.1%
pci_spoci_ctrl	1209	14	170	15.9%
alu4	2403	37	697	**29.9%**
dalu	2625	71	636	**28.2%**
i10	2676	40	257	10.6%
spi	3011	418	112	3.9%
systemcdes	3196	21	255	8.2%
C5315	3211	20	161	5.7%
C7552	4408	63	340	9.2%
s9234	5597	173	821	**20.9%**
tv80	6876	2599	658	10.8%
systemcaes	7453	307	658	9.0%
s13207	8027	119	300	5.5%
ac97_ctrl	10285	99	80	0.8%
mem_ctrl	10671	1887	2758	**27.9%**
usb_funct	11889	345	246	2.2%
pci_bridge32	15701	522	158	1.1%
s38584	19407	1053	2253	12.8%
aes_core	20280	1431	2072	10.8%
s38417	22397	1484	636	3.3%
wb_conmax	28409	2041	2313	9.7%
b17	30874	☹	614	2.5%
ethernet	37638	☹	370	1.0%
des_perf	97080	☹	2505	5.8%
average				**10%**

6 Experiments

We implemented our algorithms in C++. The SAT engine was developed by modifying MiniSAT [6]. Random simulation patterns are used to generate the initial ODC signatures. The test cases are from the IWLS 2005 suite produced from OpenCores designs generated by a quick synthesis run of the Cadence RTL Compiler [24]. The tests were run on Pentium-4 3.2 GHz machines.

For combinational simulation and equivalence checking, we treat the latches as inputs. Every internal node with an ODC-set that is not null is examined. If an ODC-equivalence is detected for the node, a merger is made. We ignore mergers that increase the number of logic levels in the design. The results were verified using the equivalence checking tool in ABC [1].

6.1 ODC Equivalence Results

In this section, we present results from a variety of benchmarks that indicate the existence of several merging opportunities due to ODCs leading to a reduction in circuit size. Table 2 shows the results of applying our ODC merging algorithm on unoptimized benchmarks. The timeout was set for 5000 seconds. The final column, *%gate_reduct*, indicates the decrease in the number of gates due to ODC merging. It is possible for a merger to cause other logic to become unnecessary thus allowing more reduction.

The results indicate that all benchmarks have merging opportunities. Several benchmarks show that 20% of the gates can be eliminated via ODC merging. Even for larger benchmarks that timed-out, we achieve favorable reductions by examining nodes earlier in logic that tend to have more ODCs.

Table 3: Gate reductions and performance of the ODC merging algorithm applied to benchmarks synthesized by running the ABC FRAIG algorithm.

benchmarks	#gates	abc(s)	odc(s)	#merge	%gate_reduct
dalu	1583	1	18	234	16.5%
C5315	1629	1	3	26	1.7%
i2c	1898	0	7	245	13.4%
s9234	2005	0	12	46	2.6%
C7552	2130	1	11	37	1.8%
pci_spoci_ctrl	2149	1	17	446	**23.1%**
i10	2229	0	18	89	4.7%
s13207	3289	0	23	54	1.7%
alu4	3540	0	73	931	**28.0%**
systemcdes	4419	1	53	812	18.9%
spi	6440	1	308	1091	17.3%
s38417	9696	1	241	78	0.9%
s38584	13217	1	247	152	1.4%
tv80	14130	4	2039	2464	18.2%
systemcaes	17488	3	1354	3532	**21.0%**
ac97_ctrl	24856	5	1101	3124	12.6%
mem_ctrl	26932	7	3166	8562	**34.9%**
b17	27887	4	⌚	243	1.2%
usb_funct	28432	4	2485	4141	15.0%
aes_core	30875	19	3164	5729	19.0%
pci_bridge32	44091	7	5045	1189	2.8%
wb_conmax	79455	47	⌚	2347	3.2%
des_perf	125759	348	⌚	2000	1.6%
ethernet	169593	103	⌚	279	0.2%
average					**10.9%**

6.2 Comparison to Other Synthesis Strategies

We have shown that ODC-based merging alone can lead to a significant reduction of area in a design. In this section, we show that ODC mergers allow for further optimizations even after optimizing the design with other strategies. Ultimately, our work could be used to enhance synthesis flows that do not utilize don't cares or replace unscalable/restrictive don't care analysis performed within a synthesis tool.

Previous synthesis tools do not take advantage of many possible ODC optimizations. One main reason for this is that previous strategies for optimization involving ODCs required too much computation. Below we show additional gains and also that the amount of additional time required for a post-synthesis ODC merging pass is often small especially for instances where few mergers exist.

In Table 3, our ODC merging algorithm is run on benchmarks that were converted to AIGs and simplified using FRAIGs in the synthesis package ABC [1]. The second column shows the number of gates after applying functional reduction to AIGs. The third and fourth columns give the runtime of ABC and the merging algorithm respectively. The last columns show the improvement achieved by doing ODC merging on the optimized design. The results indicate similar reductions and times to Table 2 and that the optimizations done to construct FRAIGs are orthogonal to the ones done here. Also, performing ODC merging on an AIG network results in only a minor increase in gate reduction over the results in Table 2. This shows that we do not require a technology-independent representation like FRAIGs to find several node mergers.

In Table 4, we evaluate our ODC merging algorithm after synthesizing the circuit using local rewriting through the resyn2 script defined in the synthesis package ABC [1, 14]. Because the rewriting strategy is competitive to other synthesis tools [14] and does not explicitly compute don't care information, we use ABC as a platform to check for potential improvements from ODC mergers. The second and third columns give the runtime of ABC and the merging algorithm respectively. The column, `area_reduct`, gives an

Table 4: Area reductions achieved by performing the ODC merging algorithm after the ABC rewriting algorithm.

benchmarks	abc(s)	odc(s)	#merge	%area_reduct
dalu	0	6	131	**12.3%**
C5315	0	2	8	1.0%
i2c	0	2	30	2.8%
s9234	0	5	10	1.1%
C7552	1	4	30	3.2%
pci_spoci_ctrl	0	3	106	9.7%
i10	1	9	39	1.6%
s13207	1	11	18	1.3%
alu4	1	37	479	**23.2%**
systemcdes	1	7	122	4.5%
spi	1	78	24	1.2%
s38417	2	189	34	0.7%
s38584	2	187	156	0.9%
tv80	3	785	639	7.3%
systemcaes	4	298	655	3.9%
ac97_ctrl	3	140	186	2.0%
mem_ctrl	5	518	1557	**16.5%**
b17	6	☹	336	1.9%
usb_funct	5	471	192	1.2%
aes_core	9	1330	2161	8.6%
pci_bridge32	6	878	89	0.2%
wb_conmax	19	☹	2404	6.2%
des_perf	50	☹	3281	3.7%
ethernet	28	☹	37	0.1%
average				**4.8%**

Table 5: Gate reductions and performance of the ODC merging algorithm performed on benchmarks synthesized by running script.rugged in MVSIS.

benchmarks	#gates	mvsis(s)	odc(s)	%over	#merge	%gate_reduct
dalu_opt	215	12	6	50%	5	2.3%
i2c	282	7	3	42.9%	2	0.7%
pci_spoci_ctrl	292	4	5	125%	3	1.0%
C5315_opt	356	8	3	37.5%	0	0%
s9234_opt	526	19	6	31.6%	1	0.2%
C7552_opt	535	65	2	3.1%	2	0.4%
i10_opt	539	191	7	3.7%	3	0.6%
spi	722	244	26	10.7%	4	0.6%
systemcdes	803	30	14	46.7%	6	0.7%
alu4_opt	902	19	35	184.2%	108	12%
s13207_opt	1101	17	10	58.8%	2	0.2%
tv80	2065	6881	125	1.8%	53	2.6%
systemcaes	2109	7022	42	0.6%	8	0.4%
s38417_opt	3012	4455	43	1%	0	0%
ac97_ctrl	3106	567	31	5.5%	1	0%
s38584_opt	3127	448	42	9.4%	4	0.1%
usb_funct	3937	331	72	21.8%	16	0.5%
pci_bridge32	4995	3252	103	3.2%	5	0.6%
aes_core	6078	379	197	52%	167	2.7%
des_perf	28545	1159	786	67.8%	354	1.2%
average				**37.9%**		**1.3%**

Table 6: Gate reductions and performance of the ODC merging algorithm performed on benchmarks synthesized by running high effort compilation in DesignCompiler. The ODC merging algorithm runtime for each benchmark is constrained to 1/3 of the corresponding runtime of DesignCompiler.

benchmarks	#gates	DC(s)	odc(s)	%over	#merge	%gate_reduct
pci_spoci_ctrl	281	15	0	0%	5	2.5%
dalu	315	11	2	18.2%	3	1.0%
s9234	375	23	1	4.3%	0	0.5%
systemcdes	437	33	0	0%	9	2.5%
s13207	487	44	1	2.3%	3	1.0%
i2c	544	17	1	5.9%	8	1.8%
alu4	806	18	6	33.3%	23	4.1%
spi	821	44	2	4.5%	4	0.7%
C5315	828	14	2	14.3%	6	0.7%
C7552	1046	17	2	11.8%	24	2.4%
i10	1185	18	4	22.2%	17	1.5%
aes_core	1758	293	3	1%	29	1.8%
tv80	1953	135	15	11.1%	16	1.1%
pci_bridge32	2079	488	23	4.7%	18	1.0%
ac97_ctrl	2119	284	12	4.2%	35	1.7%
systemcaes	2175	135	10	7.4%	10	0.6%
mem_ctrl	2560	258	23	8.9%	19	0.8%
s38417	2578	236	36	15.3%	28	1.2%
s38584	3922	207	20	9.7%	69	1.8%
ethernet	4163	3053	47	1.5%	25	0.6%
usb_funct	4718	293	44	15%	36	0.8%
wb_conmax	9833	885	203	22.9%	122	1.3%
b17	11133	1041	343	33.0%	87	0.8%
des_perf	12685	4719	216	4.6%	255	2.1%
average				**10.7%**		**1.4%**

estimated area reduction for ODC merging when technology mapping is performed after the rewriting and merging. Despite the quality of rewriting, we see that benchmarks can still be optimized with improvement over 10% in a few cases. These results indicate that ODC mergers can enhance successful synthesis flows.

Table 5 shows results of performing the ODC merging algorithm on a netlist generated MVSIS [8] using script.rugged. Several benchmarks were too large and could not complete on MVSIS and were therefore omitted from the table. Also, complete don't-care (CDC) computation [16] is done by MVSIS. Despite the use of CDCs, several mergers are still possible in the benchmarks. In addition, because the circuits have been reduced in size, the runtime has decreased accordingly. The column `%over` shows the percentage of overhead that is involved in calling the ODC merging algorithm. It is interesting to note, that the merging algorithm tends to be much faster than MVSIS. This indicates that ODC merging could be an efficient post-synthesis optimization. There are some benchmarks that achieve very few mergers. However, these instances have trivial runtime because the signatures are able to effectively eliminate several false candidates and minimize the computation required.

Table 6 gives results of performing the ODC merging algorithm on a netlist generated by Synopsys Design Compiler [25]. The benchmarks were synthesized with high effort and the result was mapped using the generic GTECH library. The results indicate that the simplifications done in Design Compiler also do not take advantage of all ODC information. One exception is *b17* where more computation time is required however gains are still found in this case within the timeout. Notice that the average overhead involved in ODC merging on a synthesized design is only 10.7%.

In general, more ODC-equivalences can be found in the designs by performing multiple topological traversals of the network. Also, we have observed that traversing the network in different orders greatly

Table 7: Statistics for the ODC merging algorithm on unsynthesized benchmarks that show the success rate of finding equivalences and number of false-candidates pruned by use of dynamic simulation vectors.

benchmarks	#merge	#SAT	%equiv	#dyn-sim	#prune
i2c	39	206	18.9%	167	36960
pci_spoci_ctrl	170	472	36%	302	34345
alu4	697	1306	53.4%	609	273497
dalu	636	1040	61.2%	404	25808
i10	257	580	44.3%	323	22029
spi	112	557	20.1%	445	78721
systemcdes	255	287	88.9%	32	153
C5315	161	192	83.9%	31	194
C7552	340	524	64.9%	184	107665
s9234	821	1959	41.9%	1138	514875
tv80	658	1781	36.9%	1117	832861
systemcaes	658	750	87.7%	88	8852
s13207	300	1007	29.8%	707	2208345
ac97_ctrl	80	256	31.3%	176	26803
mem_ctrl	2758	4356	63.3%	1580	2710618
usb_funct	246	1739	14.1%	1493	1206172
pci_bridge32	158	1189	13.3%	1031	2951017
s38584	2253	3610	62.4%	1357	3487613
aes_core	2072	2317	89.4%	245	2205
s38417	636	2416	26.3%	1780	11544973
wb_conmax	2313	5068	45.6%	2755	441002
b17	614	3588	17.1%	2974	21984143
ethernet	370	2084	17.8%	1509	2979472
des_perf	2505	2614	95.8%	109	1198
average			**47.7%**		

effects the number of mergers possible. Our strategy of topological traversal achieved the best results out of the strategies that we considered.

6.3 ODC Framework Analysis

In this section, we assess the quality of the ODC framework by providing statistics about the efficiency of the signatures in identifying ODC-equivalences. In Table 7, statistics are given for running the ODC merging algorithm on unoptimized benchmarks. The second column is the number of mergers performed. The third column is the number of times that the SAT solver is called. When the ODC equivalence candidates are found, the SAT solver is called and runs until the merger is validated, invalidated, or times out. The fourth column is the percentage of SAT calls that prove an ODC-equivalence. The fifth column reports the number of dynamic simulation vectors added when an ODC-equivalence candidate is proven false. The final column shows how many SAT calls are eliminated by adding the dynamic vectors.

The results indicate that, on average, around 50% of the SAT calls result in ODC merging. Also, the number of SAT calls pruned from dynamic simulation greatly effects this percentage. Reducing the number of SAT calls is integral, as we observe that SAT calls contribute to most of the runtime. Some benchmarks, like mem_ctrl, eliminated SAT calls by orders of magnitude. The number of vectors added is typically much smaller than the number of false positives due to the initial simulation that are pruned.

7 Conclusions

Modern industrial-grade synthesis tools integrate a plethora of logic optimizations. To this end, we show how to handle don't cares in simulation-guided synthesis environments, such as the FRAIG data structure [15]. In particular, our results indicate that ODC-equivalences and hence merging optimizations are available in realistic circuits even after extensive optimization by existing techniques. On circuits synthesized by the FRAIG-based ABC package from UC Berkeley [1], we achieve 5% average area reduction after rewriting. Moreover, our techniques never make area worse, and when few mergers are available, the runtime is small. We believe that our techniques may be able to replace slower, unscalable don't care algorithms in existing commercial tools.

A key insight in our work is to avoid the unnecessary computation of ODCs. This is accomplished through the use of simulation-based signatures and on-demand SAT-based equivalence checking which considers only as much downstream logic as necessary. To minimize the number of SAT calls, we reuse fresh counterexamples by adding them to the pool of simulation vectors.

Our techniques have a number of applications not explored in this paper. In particular, they could facilitate efficient handling of don't cares in post-placement rewiring [4]. By performing simulation on a placed design, a relaxed form of equivalence checking can be performed to enable more rewiring opportunities to reduce wirelength. Because generating signatures is generally inexpensive, the computation can be focused on particular areas of the design that require optimizations and other spurious don't care information can be ignored. The ODC framework also lends itself to future work involving sequential ODCs by considering a circuit that is unrolled several times.

References

[1] Berkeley Logic Synthesis and Verification Group, "ABC: a system for sequential synthesis and verification", `http://www.eecs.berkeley.edu/˜alanmi/abc/`.

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

[3] R. Bryant, "Graph based algorithms for Boolean function manipulation", *IEEE Trans. Comp '86*, pp. 677-691.

[4] K. H. Chang, I. Markov, and V. Bertacco, "Post-placement rewiring and rebuffering by exhaustive search for functional symmetries", *ICCAD '05*, pp. 56-63.

[5] G. DeMicheli and M. Damiani, "Synthesis and optimization of digital circuits", *McGraw-Hill '94*.

[6] N. Een and N. Sorensson, "An extensible SAT-solver", *SAT '03*, `http://www.cs.chalmers.se/˜een/Satzoo`.

[7] Z. Fu, Y. Yu, and S. Malik, "Considering circuit observability don't cares in cnf satisfiability", *DATE '05*, pp. 1108-1113.

[8] M. Gao, J. Jiang, Y. Jiang, Y. Li, S. Singha, and R. K. Brayton. MVSIS. *IWLS '01*. `http://embedded.eecs.berkeley.edu/Respep/ Research/mvsis/`

[9] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking", *DATE '01*, pp. 114-121.

[10] L. Kannan, P. Suaris, and H. Fang, "A methodology and algorithms for post-placement delay optimization", *DAC '94*, pp. 327-332.

[11] V. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization", *DAC '04*, pp. 438-441.

[12] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *IEEE Trans. CAD '02*, pp. 1377-1394.

[13] K. McMillan, "Applying SAT methods in unbounded symbolic model checking", *CAV '02*, pp. 250-264.

[14] A. Mischenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *DAC '06*.

[15] A. Mischenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report*, Berkeley. `http://www.eecs.berkeley.edu/ alanmi/abc/`.

[16] A. Mischenko and R. Brayton, "SAT-based complete don't care computation for network optimization", *DATE '05*, pp. 412-417.

[17] A. Mischenko et al, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *TCAD '06*.

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

[19] N. Saluja and S. Khatri, "A robust algorithm for approximate compatible observability don't care computation", *DAC '04*, pp. 422-427.

[20] H. Savoj and R. Brayton, "The use of observability and external dont-cares for the simplification of multi-level networks", *DAC '90*, pp. 297-301.

[21] E. Sentovich et al, "SIS: A system for sequential circuit synthesis", *ERL Technical Report '92*, Berkeley.

[22] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comp '99*, pp. 506-521.

[23] J. Marques-Silva and K. Sakallah, "Boolean satisfiability in electronic design automation", *DAC '00*, pp. 675-680.

[24] `http://iwls.org/iwls2005/benchmarks.html`.

[25] Synopsys. Design Compiler. `http://www.synopsys.com`.