Fast Detection of Node Mergers Using Logic Implications *

Yung-Chih Chen and Chun-Yao Wang
Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan
{ycchen, wcyao}@cs.nthu.edu.tw

ABSTRACT

In this paper, we propose a new node merging algorithm using logic implications. The proposed algorithm only requires two logic implications to find the substitute nodes for a given target node, and thus can efficiently detect node mergers. Furthermore, we also apply the node merger identification algorithm for area optimization in VLSI circuits. We conduct experiments on a set of IWLS 2005 benchmarks. The experimental results show that our algorithm has a competitive capability on area optimization compared to a global observability don't care (ODC)-based node merging algorithm which is highly time-consuming. Our speedup is approximately 86 times for overall benchmarks.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Optimization*

General Terms

Algorithms

Keywords

Logic implication, node merging, observability don't care

1. INTRODUCTION

Node merging is a popular and efficient logic restructuring technique. It replaces a node with another node by rewiring, and then removes the replaced node without changing the overall functionality of the circuit. A major application of the technique is to reduce the size of a logic circuit. As two nodes are merged, one of them can be removed from the circuit, and this merger may cause other redundancies in the circuit such that the resultant circuit is minimized. Circuit minimization also can be a pre-process before performing equivalence checking [3].

SAT sweeping [7] is a method that merges two functionally equivalent nodes. Firstly, it simulates the circuit by applying a set of random patterns. Next, node merger candidates are derived by searching two nodes that have the same simulation values. Finally, it uses a SAT solver to check if the nodes are actually equivalent. However, functional equivalence is not a necessary condition for node merging. In fact, if the functional differences of two nodes are never observed at any primary output (PO), these two nodes can be merged as well. Based on this observation, a node merging algorithm under *local observability don't cares* (ODCs) is proposed in [16]. The algorithm can identify additional node mergers that are not functionally equivalent to each other.

The local ODC-based algorithm [16] extends the SAT sweeping method by performing ODC analysis when deriving candidate node mergers. According to the simulation results, it computes the observability of each node and collects the pairs of nodes whose differences are not observable as candidates. Since full observability computation is very time-consuming, however, the method sets a k-bounded depth to extract local ODCs. With larger values of k, the method can identify more node mergers but spends more CPU time.

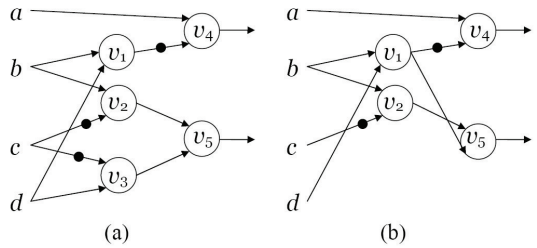

Figure 1: An example of ODC-based node merging. (a) The original circuit. (b) The resultant circuit of replacing v_3 with v_1.

To enhance the local ODC-based algorithm, the work in [11] proposes a node merging algorithm under *global* ODCs using the SAT sweeping technique as well. To reduce the complexity of full observability computation, the method computes approximate observability for each node instead of bounded-depth observability. Although the method detects certain node mergers that cannot be identified by the local ODC-based algorithm, it potentially misses some other node mergers as well. Additionally, since the approximate observability computation is global, the method expends a great amount of CPU time for large circuits.

Recently, a similar algorithm that merges nodes while considering sequential ODCs is proposed in [4]. It also employs SAT sweeping to identify node mergers under sequential ODCs for sequential circuit optimization.

Although both the works in [16] and [11] propose methods to decrease the complexity of observability computation, they cannot avoid performing ODC analysis in searching for a node merger. Additionally, they need to simulate a large amount of random patterns, collect candidate node mergers, and then perform SAT solving. This procedure can be very time-consuming due to a large number of SAT solving calls when the number of candidates is enormous.

Thus, in this work, we propose a new scheme − a sufficient condition for merging two nodes for node merger identification. In this condition, only two logic implications are required to find the substitute nodes for a given target node. The proposed approach is ODC-based, but does not need to perform observability computation, random pattern simulation, candidate collection, and SAT solving. As a result, it can globally and efficiently detect node mergers. Furthermore, we also apply the approach to area optimization in VLSI circuits.

We conduct experiments on a set of IWLS 2005 benchmarks [17]. The experimental results show that the proposed approach can efficiently identify node mergers within 175 seconds for each benchmark. In addition, an average of 6.52% of nodes can be replaced by other nodes in a circuit, and each replaceable node has an average of 2.79 substitute nodes. For area optimization, as compared to the state-of-the-art [11], the proposed approach has a speedup of 86 times for overall benchmarks while possessing a competitive capacity for circuit size reduction.

The rest of this paper is organized as follows: Section 2 uses an example to show the ODC-based node merging and presents the problem formulation. It also reviews the related concepts in VLSI testing used in this paper. Section 3 presents the proposed node merging algorithm. Its application for area optimization is introduced in Section 4. Finally, the experimental results and conclusion are presented in Sections 5 and 6.

*This work was supported in part by the National Science Council of R.O.C. under Grants NSC 98-2220-E-007-015 and NSC 98-2220-E-007-023.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICCAD'09, November 2–5, 2009, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-800-1/09/11...$10.00.

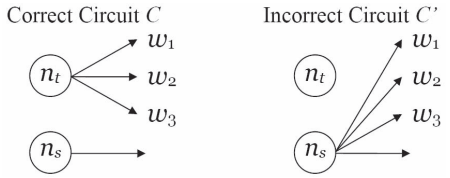

Figure 2: The misplaced wire error.

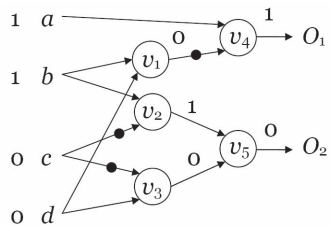

Figure 3: An example of a test for a replacement error.

2. PRELIMINARIES

2.1 An example

We use an example in Fig. 1 to demonstrate ODC-based node merging. The circuit in Fig. 1(a) is presented by using an And-Inverter Graph (AIG), which is an efficient and scalable representation for Boolean networks. Here, a, b, c, and d are primary inputs (PIs). $v_1 \sim v_5$ are 2-input AND gates. Their connectivities are presented by directed edges. A dot marked on an edge indicates that an inverter (INV) is in between two nodes.

In this circuit, v_1 and v_3 are not functionally equivalent, and thus, merging them potentially affects the overall functionality of the circuit. However, the values of v_1 and v_3 only differ when $d = 1$ and $b = c$. Additionally, $b = c$ implies $v_2 = 0$. Since $v_2 = 0$ is an input-controlling value of v_5, it prevents the value of v_3 from being observed at v_5. This situation makes the different value of v_3 with respect to v_1 never observed. Thus, replacing v_3 with v_1 does not change the overall functionality. The resultant circuit is shown in Fig. 1(b).

The problem formulation of this work is as follows: Given a target node v in a circuit, find other nodes called *substitute nodes* for v which can replace v without altering the functionality of the circuit.

For ease of discussion, we only consider circuits presented as AIGs, which is a popular and simple representation. Circuits having complex gates can also be handled by transforming them into AIGs first.

2.2 Background

An input of a gate g has an *input-controlling value* of g if this value determines the output value of g regardless of the other inputs. The inverse of the input-controlling value is called the *input-noncontrolling value*. For example, the input-controlling value of an AND gate is 0 and its input-noncontrolling value is 1. A gate g is in the *transitive fanout cone* of a gate g_s if there exists a path from g_s to g, and g_s is in the *transitive fanin cone* of g.

The *dominators* [6] of a gate g are a set of gates G such that all paths from g to any PO have to pass through all gates in G. Consider the dominators G of a gate g, the *side inputs* of G are the inputs of G that are not in the transitive fanout cone of g.

In VLSI testing, a *stuck-at fault* is a fault model used to represent a manufacturing defect within a circuit. The effect of the fault is as if the faulty wire were stuck at either 1 (stuck-at 1) or 0 (stuck-at 0). A stuck-at fault test is a process to find a test which can generate the different output values in the fault-free and faulty circuits. Given a stuck-at fault f, if there exists such a test, f is said to be testable; otherwise, f is untestable. To make a stuck-at fault on a wire w testable, a test needs to activate and propagate the fault effect to a PO. In a combinational circuit, an untestable stuck-at fault on a wire indicates that the wire is redundant and can be replaced with a constant value 0 or 1.

The *mandatory assignments* (MAs) are the unique value assignments to nodes necessary for a test to exist. Consider a stuck-at fault on a wire w; the assignments obtained by setting w to the fault-activating value and by setting the side inputs of dominators of w to the fault-propagating values are MAs. Then, these assignments can be propagated forward or backward to infer additional MAs by performing logic implications. Recursive learning [8], a learning method in automatic test pattern generation (ATPG), can be used to perform logic implications more completely. If the MAs of the fault are inconsistent, the fault is untestable, and therefore, w is redundant [13].

3. SUBSTITUTE NODE IDENTIFICATION

In this section, we first discuss the effect of merging two nodes regarding to the functionality of circuit. Next, we present a sufficient condition for correctly replacing one node with another node. Finally, according to the condition, an algorithm is presented to identify substitute nodes for a given target node.

3.1 Effect of merging two nodes

Suppose C is a combinational circuit. Let's consider the effect of replacing a node n_t with another node n_s in C regarding to the functionality of circuit. The behavior of the replacement is to use n_s to drive the wires originally driven by n_t. We can regard the replacement as setting an error in C, and it can be modeled as the misplaced wire error which is included in the typical design error models [1]. The design error models are very popular and widely used in the techniques about design error diagnosis and correction [12] [14] [15]. For example, in Fig. 2, the left and right figures indicate the correct (original) circuit C and incorrect (resultant) circuit C', respectively. The misplaced wire error is that the wires, $w_1 \sim w_3$, should be connected with n_t instead of n_s.

However, for some n_t and n_s nodes, the replacement error is undetectable when the error effect is never observed at any PO. Thus, n_t can be correctly replaced with n_s. Based on the observation, the problem of finding the substitute nodes for n_t can be transformed to finding n_s such that the replacement error is undetectable.

To check whether a replacement error is detectable or not, we can try to find an input pattern that can distinguish the functional difference between C and C'. We name such an input pattern a test for replacement error detection. Given a replacement error, if no test exists for it, the error is undetectable. As a result, to identify an undetectable replacement error, we can prove that no test exists for it. Theorem 1 below states the necessary and sufficient condition for an input pattern to be a test of a replacement error. Here, we consider an input pattern in C not in C'. This is because we do not first perform the replacement and then check if the replacement is detectable.

Theorem 1: Let f denote an error of replacing n_t with n_s. An input pattern t in C will be a test for f, if and only if t generates the different values for n_t and n_s and propagates the value of n_t to a PO.

Proof: Generating the different values for n_t and n_s is equivalent to activating the error effect of $n_t \neq n_s$, and propagating the value of n_t to a PO is equivalent to propagating the error effect to a PO. If an input pattern t can simultaneously activate and propagate the error effect, the error effect can be observed at a PO, and hence, t can detect f. Otherwise, t cannot detect f. □

For example, Fig. 3 shows that same circuit as Fig. 1(a), consider the error of replacing v_3 (n_t) with b (n_s). Here, the input vector ($a = 1$, $b = 1$, $c = 0$, $d = 0$) is a test, because it generates $v_3 \neq b$ ($v_3 = 0$, $b = 1$) and propagates $v_3 = 0$ to O_2.

According to the value of n_t, we can classify the complete test set of f into two subsets: The first one T_0 and the second one T_1 which consist of the input patterns having $n_t = 0$ and $n_t = 1$, respectively. Since a test for f propagates the value of n_t to a PO, each pattern in T_0 is exactly a test for the stuck-at 1 fault on n_t. Similarly, each pattern in T_1 is exactly a test for the stuck-at 0 fault on n_t. However, the converse relationship does not hold because a test for the stuck-at fault on n_t may not satisfy generating the different values for n_t and n_s.

Based on the relationship between the test sets for f and the stuck-at faults on n_t, we can divide the detection of f into the

Find_Substitute_Node(**Node** n_t)
1. Compute MAs($n_t = sa0$).
2. Compute MAs($n_t = sa1$).
3. *SubstituteNodes* $\longleftarrow$ nodes having different values in MAs($n_t = sa0$) and MAs($n_t = sa1$), and not in the transitive fanout cone of n_t.

Figure 4: The substitute node identification algorithm.

detection of two stuck-at faults. One is f_0: the stuck-at 0 fault on n_t under $n_s = 0$, and the other one is f_1: the stuck-at 1 fault on n_t under $n_s = 1$. Here, the constraints, under $n_s = 0$ and $n_s = 1$, are used to make a test have $n_t \neq n_s$. For example, consider the detection of f_0. Besides activating and propagating the fault effect of $n_t = 1$, a test must simultaneously cause $n_s = 0$. Therefore, T_1 is the complete test set for f_0 and T_0 is the complete test set for f_1. If f_0 and f_1 are simultaneously untestable, f is undetectable as well. Furthermore, a node n_s that renders f_0 and f_1 untestable is a substitute node of n_t.

Next, we introduce a sufficient condition as presented in Condition 1 for n_s that renders f undetectable based on the untestable f_0 and f_1. Then, we propose an algorithm to efficiently find n_s according to Condition 1.

Condition 1: Let f denote an error of replacing n_t with n_s. If $n_s = 1$ and $n_s = 0$ are MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t, respectively, f is undetectable.

Since generating $n_s = 0$ is necessary in a test for f_0, if $n_s = 1$ is also an MA for the stuck-at 0 fault on n_t, then f_0 is untestable due to the contradiction on the value of n_s. Similarly, if $n_s = 0$ is also an MA for the stuck-at 1 fault on n_t, f_1 is untestable. As a result, when Condition 1 is held, f is undetectable.

3.2 Proposed algorithm

Given a target node n_t, we can use the sufficient condition in Condition 1 to find its substitute nodes. Firstly, we compute the MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t, respectively. Then, we collect the MAs that satisfy Condition 1. Finally, the corresponding nodes are the substitute nodes of n_t. Based on Condition 1, we can identify more than one substitute nodes simultaneously by performing only two logic implications.

For example, consider finding the substitute nodes of v_3 in the circuit of Fig. 1(a). Firstly, we compute the MAs for the stuck-at 0 fault on v_3. To activate the fault effect, v_3 is set to 1. To propagate the fault effect, v_2 is set to 1. We then perform logic implications to derive additional MAs. They are $d = 1$, $c = 0$, $b = 1$, $v_1 = 1$, $v_4 = 0$, and $v_5 = 1$. Thus, the set of MAs for the stuck-at 0 fault on v_3 is $\{v_3 = 1, v_2 = 1, \boldsymbol{d = 1}, c = 0, b = 1, \boldsymbol{v_1 = 1}, v_4 = 0, \boldsymbol{v_5 = 1}\}$. Secondly, we use the same method to compute the MAs for the stuck-at 1 fault on v_3. They are $\{v_3 = 0, v_2 = 1, \boldsymbol{d = 0}, c = 0, b = 1, \boldsymbol{v_1 = 0}, \boldsymbol{v_5 = 0}\}$. Finally, d and v_1 are the substitute nodes of v_3 due to the satisfaction of Condition 1. Note that although v_5 also satisfies Condition 1, it is excluded from being a substitute node of v_3. This is because v_5 is in the transitive fanout cone of v_3, replacing v_3 with v_5 will result in a cyclic combinational circuit.

Furthermore, Condition 1 can be modified by reversing the value of n_s to find another kind of substitute node that replaces a target node with an additional INV. That is, if $n_s = 0$ and $n_s = 1$ are MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t, respectively, n_t can be replaced by n_s with an additional INV. Finding this kind of substitute node can increase the possibility of replacing a target node.

Fig. 4 shows the proposed algorithm for substitute node identification. Given a target node n_t, the algorithm computes MAs($n_t = sa0$) and MAs($n_t = sa1$). Then, nodes which have different values in MAs($n_t = sa0$) and MAs($n_t = sa1$), and are not in the transitive fanout cone of n_t are the substitute nodes. Therefore, only two logic implications are required to identify the substitute nodes for n_t: one is for the stuck-at 0 fault test on n_t and the other one is for the stuck-at 1 fault test on n_t.

4. AREA OPTIMIZATION

During optimization, each node of a circuit is visited and replaced if applicable. To determine the optimization order, we conducted some experiments on a set of IWLS 2005 benchmarks [17] and observed that iteratively selecting a target node from POs to PIs in the depth-first search (DFS) order and replacing it with a substitute node closest to PIs results in better simplification for most benchmarks. Thus, we use this order for circuit optimization in this work.

Area_Optimization(**Circuit** C)
For each node n in C in the DFS order from POs to PIs
1. Compute MAs($n = sa0$).
2. Compute MAs($n = sa1$).
3. *SubstituteNodes* $\longleftarrow$ nodes having the different values in MAs($n = sa0$) and MAs($n = sa1$), and not in the transitive fanout cone of n.
4. Replace n with a node which is in the set of *SubstituteNodes* and closest to PIs.

Figure 5: The overall algorithm for area optimization.

Table 1: The experimental results of substitute node identification.

benchmark	N	N_t	%	N_s	ratio	time (s)
C3540	1038	29	2.79	33	1.14	0.27
rot	1063	42	3.95	59	1.4	0.15
simple_spi	1079	26	2.41	125	4.81	0.11
i2c	1306	80	6.13	174	2.18	0.21
pci_spoci.	1451	170	11.72	890	5.24	0.62
dalu	1740	217	12.47	560	2.58	0.95
C5315	1773	33	1.86	113	3.42	0.16
s9234	1958	175	8.94	270	1.54	0.37
C7552	2074	60	2.89	104	1.73	0.41
C6288	2337	2	0.09	2	1	0.45
i10	2673	626	23.42	1076	1.72	1.35
s13207	2719	159	5.85	231	1.45	0.64
systemcdes	3190	147	4.61	301	2.05	1.51
i8	3310	1533	46.31	11622	7.58	3.84
spi	4053	65	1.6	91	1.4	3.35
des_area	4857	80	1.65	152	1.9	5.58
alu4	5270	206	3.91	236	1.15	54.87
s38417	9219	173	1.88	257	1.49	1.45
tv80	9609	496	5.16	3864	7.79	17.19
b20	12219	849	6.95	1640	1.93	17.28
s38584	12400	549	4.43	1109	2.02	17.02
b21	12782	1094	8.56	2066	1.89	19.34
systemcaes	13054	202	1.55	380	1.88	17.71
ac97_ctrl	14496	98	0.68	242	2.47	3.22
mem_ctrl	15641	1537	9.83	3588	2.33	98.8
usb_funct	15894	370	2.33	1271	3.44	6.33
b22	18488	1047	5.66	2127	2.03	24.95
aes_core	21513	452	2.1	1742	3.85	15.17
pci_bridge32	24369	309	1.27	621	2.01	21.69
wb_conmax	48429	5608	11.58	41996	7.49	28.18
b17	52920	1565	2.96	5515	3.52	174.49
des_perf	79288	2505	3.16	6195	2.47	51.37
average			6.52		2.79	
total						589.03

Fig. 5 shows the overall algorithm of applying node merging for area optimization. Given a circuit C, the algorithm iteratively selects a node n as a target node in the DFS order from POs to PIs, and then replaces n with one of its substitute nodes if applicable. Firstly, the algorithm computes MAs($n = sa0$) and MAs($n = sa1$), respectively. Then, the nodes which have the different values in MAs($n = sa0$) and MAs($n = sa1$), and are not in the transitive fanout cone of n are the substitute nodes of n. Finally, the algorithm selects one substitute node which is closest to PIs to replace n.

5. EXPERIMENTAL RESULTS

We implemented our algorithms in C language within an ABC [2] environment. The experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.6). The benchmarks are from the IWLS 2005 suite [17]. Each benchmark is initially transformed to an AIG format and we only consider its combinational portion. Additionally, in order to perform logic implications more completely with reasonable CPU time overhead, a recursive learning technique [8] is applied with the recursion depth 1 in our algorithms.

The experimental results consist of two parts: The first one is to show the efficiency and effectiveness of our approach for substitute node identification. The second one is to show the capability of our approach for area optimization as compared to the state-of-the-art in [11].

5.1 Substitute node identification

In the experiments, each node in a benchmark is considered

a target node. We identify its substitute nodes by using the proposed algorithm as shown in Fig. 4 and measure the CPU time in seconds.

Table 1 summarizes the experimental results of substitute node identification. Column 1 lists the benchmarks. Column 2 lists the number of nodes in each benchmark represented by AIG N. Columns 3 and 4 list the number of target nodes identified having substitute nodes N_t and its percentage with respect to N, respectively. Column 5 lists the number of pair of target node and substitute node N_s. It also represents the total number of substitute nodes identified. Column 6 lists the ratio of N_s with respect to N_t. Column 7 lists the CPU time.

According to Table 1, we can find the substitute nodes for an average of 6.52% of nodes in a benchmark, with 2.79 substitute nodes for each on average. Additionally, the experimental results also show that our approach can efficiently identify substitute nodes. The most time-consuming benchmark is *b17*, which costs only 174.49 seconds. All the other benchmarks cost less than 100 seconds.

However, the *alu4* benchmark takes much more CPU time, 54.87 seconds, than the other benchmarks of similar size such as the *des_area* and the *s38417* benchmarks. This phenomenon occurs due to the different logic structures. We find that an logic implication computes an average of 1680.41 MAs for the *alu4* benchmark, but only an average of 581.06 and 35.72 MAs for the *des_area* and the *s38417* benchmarks, respectively. Additionally, 44.53 out of 54.87 seconds are spent on performing recursive learning.

5.2 Area optimization

In the experiments, we compare our approach with that in [11] for area optimization. To have a fair comparison, we initially optimize each benchmark by using the *resyn2* script in the ABC package as performed by [11], which performs local circuit rewriting optimization [9]. After the initialization, we optimize the benchmarks by using the proposed algorithm as shown in Fig. 5. After the end of optimization, we also apply an equivalence checking tool, *cec* [10], in the ABC package to verify the correctness of our optimization. Among these benchmarks, the *b17* benchmark takes the most CPU time, 10.37 seconds, to perform *cec*.

Table 2 summarizes the experimental results. Columns 1 and 2 list the benchmarks and the number of nodes in each benchmark represented by AIG, respectively. Columns 3 to 6 list the results of our approach. These columns contain the number of node mergers identified, the number of nodes in the resultant benchmark N_r, the percentage of area reduction in terms of node count, and the CPU time, respectively. Columns 7 and 8 list the corresponding results reported in [11], the percentage of area reduction and the CPU time, respectively. The maximal CPU time in Column 8 is 5 000 seconds, which is the run time limit set by that work.

According to Table 2, our approach can obtain an average of 3.94% area reduction for the benchmarks. However, because our approach only focuses on area optimization, it may decrease or increase the number of logic levels of the benchmarks. From the aspect of quality comparison, the simplification capability of our approach is not as strong as that of the approach in [11], which obtains an average of 5.04% area reduction. The key reason behinds this result is the completeness of MA computation. If MAs are inferred more completely in our approach, more node mergers can be identified. Unfortunately, finding all MAs of a stuck-at fault requires exponential time complexity. It is equivalent to finding all patterns that can detect a fault [5]. Therefore, the recursive learning technique is used in our approach to infer more MAs for improving results. On the other hand, from the aspect of efficiency comparison, our overall CPU time is only 254.24 seconds, which is much less than 21 887 seconds required by the approach in [11]. Our approach has a speedup of 86 times.

Table 2: The experimental results of our approach and [12] for area optimization.

benchmark	N	our approach				[11]	
		# mergers	N_r	%	time (s)	%	time (s)
pci_spoci.	878	59	782	10.93	0.24	9.2	6
i2c	941	15	923	1.91	0.11	3.2	3
dalu	1057	37	985	6.81	0.3	12	10
C5315	1310	6	1304	0.46	0.09	0.7	2
s9234	1353	14	1331	1.63	0.16	1.2	8
C7552	1410	33	1371	2.77	0.28	3.4	8
i10	1852	72	1755	5.24	0.64	1.3	12
s13207	2108	36	2063	2.13	0.48	1.8	17
alu4	2471	164	1941	21.45	5.29	22.9	64
systemcdes	2641	33	2600	1.55	0.94	4.7	9
spi	3429	16	3411	0.52	2.71	1.3	84
tv80	7233	151	6960	3.77	10.6	7.1	1445
s38417	8185	41	8136	0.6	1.15	1	275
mem_ctrl	8815	258	7257	17.67	6.76	18	738
s38584	9990	99	9846	1.44	11.44	0.8	223
ac97_ctrl	10395	16	10379	0.15	1.96	2	188
systemcaes	10585	50	10521	0.6	13.09	3.8	360
usb_funct	13320	215	13026	2.21	5.89	1.4	681
pci_bridge32	17814	83	17729	0.48	12.04	0.1	1134
aes_core	20509	138	20371	0.67	13.23	8.6	1620
b17	34523	422	33979	1.58	72.4	1.6	5000
wb_conmax	41070	1199	39266	4.39	31.88	6.2	5000
des_perf	71327	1159	70081	1.75	62.56	3.7	5000
average				3.94		5.04	
total					254.24		21887
ratio					1		86.09

6. CONCLUSION

In this paper, we propose an ODC-based node merging algorithm that can quickly detect node mergers using logic implications. The algorithm is based on a sufficient condition for replacing a node with another node, and only two logic implications are required for finding substitute nodes of a given target node. Moreover, based on the node merging algorithm, we also propose an efficient algorithm for area optimization in combinational circuits. The experimental results show that the proposed algorithm has a competitive capability of area optimization and expends much less CPU time compared to the state-of-the-art.

7. REFERENCES

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 138-148, Jan. 1988.

[2] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/ alanmi/abc/.

[3] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression for Formal Verification", in *Proc. Int. Conf. on Computer-Aided Design*, 2004, pp. 42-49.

[4] M. Case, V. Kravets, A. Mishchenko, and R. Brayton, "Merging Nodes Under Sequential Observability," in *Proc. Design Automation Conf.*, 2008, pp. 540-545.

[5] C. W. Jim Chang, M. F. Hsiao, and M. M. Sadowska, "A New Reasoning Scheme for Efficient Redundancy Addition and Removal," *IEEE Trans. Computer-Aided Design*, vol. 22, pp. 945-952, July 2003.

[6] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," in *Proc. Design Automation Conf.*, 1987, pp. 502-508.

[7] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Propery Checking," in *Proc. Int. Conf. on Computer-Aided Design*, 2004, pp. 50-57.

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

[9] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. Design Automation Conf.*, 2006, pp. 532-536.

[10] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to Combinational Equivalence Checking, " in *Proc. Int. Conf. on Computer-Aided Design*, 2006, pp. 836-843.

[11] S. Plaza, K. H. Chang, I. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares," in *Proc. Asia South Pacific Design Automation Conf.*, 2007, pp. 414-419.

[12] I. Pomeranz and S. M. Reddy, "On Correction of Multiple Design Errors," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 255-264, Feb. 1995.

[13] M. H. Schulz amd E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," in *Proc. Int. Fault-Tolerant Computing Symp.*, 1988, pp. 30-35.

[14] A. Veneris and I. N. Hajj, "Design Error Diagnosis and Correction via Test Vector Simulation," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1803-1816, Dec. 1999.

[15] A. Veneris, J. B. Liu, M. Amiri, and M. S. Abadir, "Incremental Diagnosis and Correction of Multiple Faults and Errors," in *Proc. Design, Automation and Test in Europe Conf.*, 2002, pp. 716-721.

[16] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't Cares," in *Proc. Design Automation Conf.*, 2006, pp. 229-234.

[17] http://iwls.org/iwls2005/benchmarks.html.