A Coupling Area Reduction Technique Applying ODC Shifting

Yi Diao, Tak-Kei Lam, Xing Wei
Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
{ydiao, tklam, xwei}@cse.cuhk.edu.hk

Yu-Liang Wu
Easy-Logic Technology Ltd.
Hong Kong
ylw@easylogic.hk

Abstract—**Circuit size reduction is a basic problem in today's integrated circuit (IC) design. Besides yielding a smaller area, reducing circuit size can also provide advantages in many operations throughout the design flow, including technology mapping, verification and place-and-route. In recent years, some node based logic synthesis algorithms have been proposed for this purpose. Node Addition and Removal (NAR) and Observability Don't Cares (ODCs) based node merging were found to be quite effective in reducing the number of nodes in a netlist. However, both methods do not address the effect of re-distributing ODCs and the results are virtually fixed after one iteration run. We study the implications of redistributing ODCs and propose a node-based and wire-based coupling synthesis scheme that can effectively find better solutions with the application of ODC shifting operations. Experimental results show that this approach can produce area reductions nearly double of the pure node-based algorithms.**

Keywords—Node merging, Rewiring, Area Reduction, Logic synthesis

I. INTRODUCTION

Area reduction is very important in Integrated Circuit (IC) design. Many operations throughout the whole design flow, including technology mapping, verification and placement, can consequently become less complicated when the netlists are smaller.

Node merging is an important technique for area reduction. It reduces the circuit area by merging the logically equivalent nodes and thus reduces the number of nodes in a netlist. Rewriting [1] and sweeping [2] are the earliest node merging algorithms known. They are effective and can be easily scaled for large circuits. But the constraint of these algorithms is that two nodes must be logically equivalent to be merged. This requirement is too strict and prevents us from finding more merging solutions.

Recently, some improved node merging algorithms [3][4][5][6] have been proposed. Their advantage over the earlier node merging methods is that, these algorithms explored the concepts of ODCs. By utilizing the information of ODCs, two nodes that are not logically equivalent can still be merged if their difference is included in one of the node's ODCs.

In [4], the authors proposed a Boolean satisfiability-based (SAT-based) node merging algorithm, which computes the approximate ODCs of each node and applies SAT to check the correctness. NAR [5] is an Automatic Test Pattern Generation (ATPG) based algorithm, which replaces a target node by another existing node inside the netlist. In [6], NAR was improved to allow replacement of a target node by an additional new node when no suitable existing nodes can be found for substitution.

The two different methods in [4] and [6] have shown comparable area reduction capability when tested under the same benchmarks. With regard to speed, [6] runs faster because ATPG-based techniques are intrinsically more efficient than SAT-based techniques.

We observe that both algorithms highly depend on the initial ODC distributions of the circuits and the results can be further improved if ODC information can be redistributed following certain guidelines. Suppose there is a target node having no substitute nodes. If we can change the logic function of the target node, or enlarge the Boolean space of its ODCs without affecting any function of the primary outputs (POs), we may have a higher chance of replacing that target node.

To explore the above idea, in our work, we propose a new area reduction algorithm in which node-based and wire-based logic synthesis algorithms are coupled. When the node-based algorithm fails to find any merging solution for a target node, we use the wire-based algorithm to modify its logic function or ODC toward the direction of enhancing it chance of being merged while maintaining the functionality of the entire circuit. In this work, we choose NAR [6] as the basic node-based method, and ECR [12], which is a rewiring technique, as the wire-based algorithm.

The effectiveness of this scheme is justified by empirical results which demonstrate nearly double area reduction over all other previous works.

II. BACKGROUND

In this section, we will introduce some background knowledge about our work. We first explain the definition of ODC, and then discuss the node merging algorithm NAR. At last we will explain the principle of the rewiring engine ECR.

A. Observability Don't Cares (ODCs)

ODCs occur when the value of an internal node does not affect the outputs of the circuit because of limited observability. For example, in Figure 1, $a = 0$ implies $n_2 = 0$. Since $n_2 = 0$ is an input-controlling value of n_4, it prevents the value of n_3 from being observed at n_4. Then, the output value of node n_3 is a don't care under such situation.

Suppose the logic function of the target node n_t and substitute node n_s are $F(n_t)$ and $F(n_s)$ respectively. The ODCs of n_t is represented as $ODC(n_t)$. With the flexibility of ODCs, $F(n_t)$ and

F(n_s) are not required to be equivalent, so long as Equation (1) is satisfied.

$$F(n_t) \oplus F(n_s) \subset ODC(n_t) \quad (1)$$

For example, in Figure 1, n_1 and n_3 are not functionally equivalent. However, the values of n_1 and n_3 only differ when $a = b$. Additionally, $a = b$ implies $n_2 = 0$. As discussed above, the output value of node n_3 is a don't care under such situation. Thus, replacing n_3 with n_1 does not change the overall functionality.

We can infer from Equation (1) that, the larger the set of ODCs, the easier the target node can be replaced. This is because having more ODCs allows more logic difference between the target and the substitute nodes.

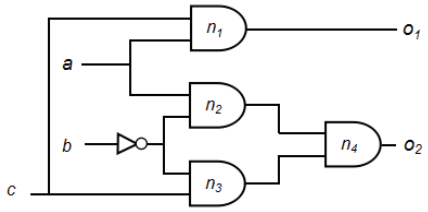

Figure 1 Example of ODCs and node merging

Computing exact ODCs explicitly is very time consuming. Hence, most algorithms use approximate ODCs instead. For example, ATPG techniques approximate ODCs by exploring compatible logic difference that won't be observed under any test patterns.

B. Node Merging Algorithm

As described in the previous section, different kinds of node merging algorithms were proposed in previous works. Among them, ATPG-based Node Addition and Removal (NAR) [6] is one of the most efficient. It replaces the target node with an existing substitute node or an additional new node identified by ATPG techniques.

Mandatory assignments (*MAs*) are the values that have to be assigned to some nodes so as to activate and propagate the specific stuck-at fault. The NAR algorithm assumes there are stuck-at faults (*stuck_at_0* and *stuck_at_1*) occurring at the target node. It then calculates the mandatory assignments for each of the *stuck_at_0* fault (*s_a_0*) and *stuck_at_1* fault (*s_a_1*), which are denoted by MA_0 and MA_1 respectively. A node having different values in MA_0 and MA_1 and is not in the transitive fanout cone of the target node, no matter being an existing node in the circuit or an additional new node, is considered as a qualified substitute node for the target node.

Though ODCs are not explicitly computed, NAR utilizes the flexibility of ODCs. In NAR, the target node and the substitution node do not have to be exactly equivalent, so long as the difference between their logic functions will never be propagated to any PO under any test pattern.

C. Rewiring Algorithm

Rewiring, which was originally proposed in [7-8], is a powerful technique for combinational optimization. It transforms the circuit through replacing certain wires by adding some extra wires to the circuit, while keeping the logic function of the circuit unchanged. The wires being removed are called target wires (TWs), while the additional wires to be added are called alternative wires (AWs). The appropriate target wires and alternative wires can be selected via suitable cost functions to achieve different optimization objectives.

The most commonly used rewiring technique is ATPG-based. It converts the problem of finding target-alternative wire pairs into a problem of seeking undetectable stuck-at faults where ATPG techniques are applied.

ECR [12] is one of the typical ATPG-based rewiring techniques. Similar to NAR, the removal of an existing wire w_t from the circuit can be considered as the occurrence of the wire's stuck-at fault, which induces an error w_t error. The addition of a wire w_a which does not exist in the circuit originally introduces an error w_a error. If the w_t error can be cancelled before it is propagated to any primary output by adding w_a error, w_a is an alternative wire for w_t. In other words, w_t and w_a mutually cancel each other completely before reaching primary outputs. Different from traditional rewiring techniques such as RAR [9][10] and IRRA [11], ECR does not require w_a to be redundant. It is therefore more flexible.

Figure 2 describes an example of rewiring. After the transformation where the target wire $b \rightarrow n_1$ has been replaced with the alternative wire $n_2 \rightarrow n_6$, the function of every PO remains unchanged, and the size of the circuit is not increased.

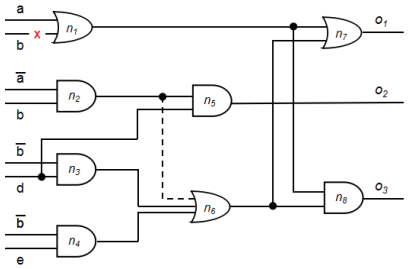

Figure 2 Example of rewiring

III. THEORETICAL EXPLANATION OF OUR METHODOLOGY

As discussed in Introduction (Section 1), traditional node merging techniques have some limitations in area reduction. In our methodology, we employ rewiring, the wire-based logic synthesis tool, to lift the limitation. Before describing the details of our algorithm, we first explain the theoretical ground on why we choose rewiring and how it can help to solve the problems in this section.

A. Rewiring for Further Optimization

Though node merging is a well-known technique for area reduction, rewiring can perform the same function too. In fact, some earlier works [13] already applied rewiring technique to simplify netlists.

In practice, node merging can be regarded as a special kind of rewiring techniques. It removes several target wires (all fanout wires of the target node) simultaneously, and has a constraint that all alternative wires must share the same source node (the substitute node).

Unlike node merging which removes all fanout wires simultaneously, rewiring replaces the fanout wires one by one. The target node can be removed from the netlist too when all its fanout wires are successfully removed. In terms of node removal, rewiring is not as efficient as node merging, as we cannot determine whether the target node can be successfully removed until one of its fanout wires is proven to be unremovable. But in some cases, rewiring can find a solution while node merging fails. In fact, rewiring can also be viewed as a special and extended

node merging technique that replaces the target node with multiple existing nodes. Figure 3 shows the difference between node merging and rewiring when removing the target node.

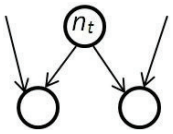

Figure 3.1
Original netlist

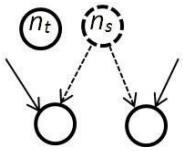

Figure 3.2
Removing n_t with n_s by node merging

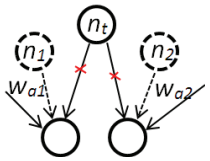

Figure 3.3 Removing node n_t by rewiring

B. Rewiring for ODCs Shifting

From the previous section we know that, the degree of difficulty of replacing a node is directly related to its observability don't cares. Intuitively, it is easy (difficult) to replace a node when the don't cares set is large (small). Node merging techniques do not change the ODCs of nodes intentionally. In the following text, we will explain how rewiring can shift ODCs on the netlist to lead to a better structure for subsequent optimization.

Intuitively, a node closer to PIs, or more distant from POs, is likely to have a larger set of ODCs, as it tends to be less observable at the POs. Typically, if a gate has a fanout being a primary output, the gate is fully observable and has no observability don't cares.

Therefore, during rewiring process, we can intentionally push some critical nodes to be closer to PIs or further away from POs to increase their ODCs. We have the following heuristics.

Heuristic 1: Suppose node n_a is a PI or a node very close to PI. Adding a new wire from n_a to another node n_b can reduce the distance from n_b to PIs, and consequently the ODCs of n_b and the nodes in its transitive fanin cone are increased.

Heuristic 2: Suppose $n_a \rightarrow n_b$ is an existing wire, and n_b is very close to POs. If we can disconnect n_a from n_b by replacing $n_a \rightarrow n_b$ with an alternative wire, the distance of n_a to POs may be increased. In that case, the ODCs of n_a and nodes in its fanin cone may likely be increased.

Typically, if a gate has more fanouts, it is easier to be observed. This is because there are more paths to the primary outputs. As a result, the size of the ODCs associated with this node decreases and the same applies to the nodes in its fanin cone. However, it is not always true because multiple fault effects may cancel each other due to the reconvergence of the node's fanouts. We have the following heuristic.

Heuristic 3: Suppose $n_a \rightarrow n_b$ is a fanout wire of n_a, whose fanout cone does not intersect with the fanout cones of n_a's other fanouts. Then, deleting $n_a \rightarrow n_b$ increases the ODCs of n_a and the nodes in its fanin cone. The purpose of the restriction on the fanout cone of wire $n_a \rightarrow n_b$ is to ensure there is no fanout reconvergence.

In the experiments in [6], every benchmark was processed with six iterations of optimization. The authors show that even running the optimization flow with the traditional node merging algorithm NAR for five more iterations, the average improvement on area reduction can increase only 0.9% (from 5% to 5.9%). This implies that node merging algorithms reach local minimum rapidly after the first iteration. As discussed earlier, rewiring techniques can escape the local minimum by introducing perturbations on the netlist.

The perturbations introduced by rewiring can alter the circuit structure by replacing a wire with another wire without increasing the circuit size. Applying such unintrusive rewiring transformations repeatedly by proper guidance can lead to a better circuit structure for subsequent optimization processes. The heuristics guiding the transformation process will be discussed in the next section.

Figure 4.1 and 4.2 give a small example to show how rewiring and node merging are applied together to achieve a better optimization result. Originally, no merge solution can be found in Figure 4.1. But if we can remove wire $n_2 \rightarrow n_4$, or replace it with an alternative wire, then node n_1, which is in the fanin cone of n_2, can be replaced by n_6. Figure 4.2 shows the optimized netlist.

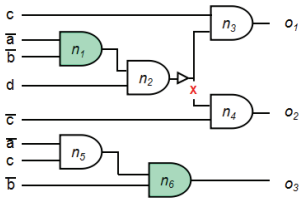

Figure 4.1
Rewiring for ODCs shifting

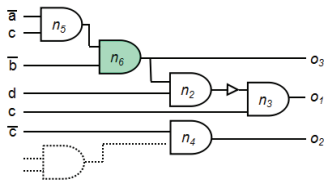

Figure 4.2
Netlist after rewiring and node merging

Rewiring has been proven to be efficient. In [12] the authors demonstrated that, the complexity of replacing a wire by ECR is very close to merging a node, and the rewiring rate (the rate of successfully replacing a target wire with its alternative wires) is nearly 40%. The number of nodes in the circuit can be kept from being increased during the rewiring process.

IV. DETAILS OF OUR APPROACH

In this section, we will describe our new approach for area reduction in details. We will focus on how the node merging technique and rewiring technique are coupled.

In an iteration, our approach is mainly consisted of two stages, the optimization stage and the perturbation stage. In the optimization stage we try to reduce the number of nodes by both node merging and rewiring techniques. In the perturbation stage, rewiring is applied to perturb the netlist so as to escape from a local minimum.

A. Optimization Stage

In [4] and [6], only node merging technique is applied to reduce the number of nodes in the netlist. However, in some cases, rewiring can find a solution to remove the target node while node merging fails.

In our approach, for each node, we first try to replace it with an existing node or an additional new node by node merging technique. If that fails, rewiring is then performed on the fanout wires of the target node one by one. If all fanout wires can be successfully removed, the target node can be removed from the netlist. If we cannot remove a target node, reducing its number of fanouts will likely bring an effect of increasing the ODCs of the nodes in the target node's fanin cone, as well as the ODCs of the target node.

Before rewiring, we assign a priority to every fanout wire of the target node according to its potential for ODCs optimization. Wire

with higher priority will be treated earlier. According to the discussion in the previous section, the fanout wires with following features will be assigned with higher priority:

(1) Wires whose fanout cones do not intersect with the fanout cones of other fanout wires. According to Heuristic 3, deleting such wires can increase the ODCs of the target node.

(2) Wires that are close to POs. Deleting them can probably push the target node further away from POs according to Heuristic 2.

During rewiring, according to Heuristic 1, the target wire is replaced by the alternative wire that is the closest to PIs In our scheme.

As discussed above, performing rewiring on the target node can help to increase the ODCs of the target node and also the ODCs of all the nodes in the target node's fanin cone. Obviously, a node nearer to the POs has a larger fanin cone. Hence, target nodes are selected in the reverse topological order from POs to PIs in the optimization flow so that more nodes can potentially be benefited from subsequent optimization.

Algorithm 1: Optimization algorithm

```
input : circuit C, max_fanout_limit
output : optimized circuit C'
1  begin
2    C' ← C;
3    foreach node n_t in C in the DFS order from POs to PIs do
4      N_s ← Find substitute node by node merging algorithm NAR ;
5      if success then
6        C' ← Replace n_t with a substitute node n_s ∈ N_s
           That is the closest to PIs
7      else if fanout number of n_t < max_fanout_limit then
8        Sort fanout wire by priority;
9        foreach fanout wire w_t do
10         W_a ← Find alternative wires by rewiring engine ECR;
11         if success then
12           C' ← Replace w_t with an alternative wire
               w_a ∈ W_a whose source node is the closet to PIs ;
13   Return C'
14 end
```

Figure 5 Optimization algorithm

Algorithm 1 describes the flow of the optimization stage. In practice, we choose NAR as the node merging technique, and ECR as the rewiring engine, as they are both efficient and powerful. The parameter *max_fanout_limit* is used to decide whether rewiring should be performed. If the number of fanout wires of a node exceeds *max_fanout_limit*, rewiring is not performed because it is practically hard to optimize such nodes.

B. Perturbation Stage

As the local optimum will be reached gradually, we perform a perturbation stage between two optimization stages. We extend the method in [13] as follows.

We try to transform the netlist into the structure suitable for optimization by performing rewiring repeatedly. Unlike the optimization stage, target nodes are selected in the topological order from PIs to POs in the perturbation stage. For each target node, we treat each of its fanout wire as the target wire for rewiring. Since estimating the change of ODC is very time consuming, some heuristics are adopted to choose the best alternative wire to be added for a target wire.

For a target wire $n_a \rightarrow n_b$, we follow the rules derived from Heuristic 1 to choose the best alternative wire. Suppose $n_c \rightarrow n_d$ is a candidate alternative wire. In order to optimize the ODCs, the destination node of the alternative wire n_d has to be a dominator of the target wire, and the source node of the alternative wire n_c must be a PI or closer to PIs than the source node of the target wire n_a.

In Figure 6, Cone I contains n_a and its fanin cone. After a fanout wire of n_a is removed, we assume that the ODCs of the nodes in Cone I are increased. Cone II contains n_c and its fanin cone. If an additional fanout wire is added to n_c, we consider that the ODCs of the nodes in Cone II are decreased.

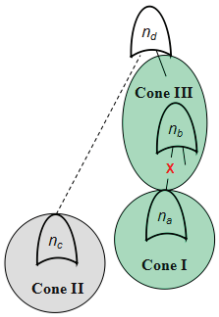

Figure 6 Rewiring for perturbation

Cone III contains all the nodes in the intersection of the fanin cone of n_d and the fanout cone of n_b. As demonstrated in [13], the ODCs of the nodes in this area will be increased after the transformation.

The circuit will be transformed with an alternative wire $n_c \rightarrow n_d$ if the following equation is satisfied.

$$Size(Cone\ I) + Size(Cone\ III) > Size(Cone\ II) \quad (2)$$

Equation (2) implies that we accept the transformation if the number of nodes that have their ODCs increased is larger than the number of nodes that have their ODCs decreased.

Algorithm 2: Perturbation algorithm

```
input : circuit C
output : perturbed circuit C'
1  begin
2    C' ← C;
3    foreach node n_t in C in the topological order from POs to PIs do
4      foreach fanout wire w_t do
5        W_a ← Find alternative wires by rewiring engine ECR ;
6        Sort alternative wires W_a in the order of the distance
           between their source node and PIs ;
7        foreach alternative wire w_a ∈ W_a do
8          Calculate Size (Cone I), Size (Cone II), Size (Cone III);
9          if size (Cone I) + size (Cone III) > size (Cone II) then
10           C' ← Replace w_t with an alternative wire w_a ;
11           break;
12   Return C'
13 end
```

Figure 7 Perturbation algorithm

Algorithm 2 describes the flow of the perturbation stage. As in optimization stage, ECR is adopted as the rewiring engine.

C. The Main Flow

In our flow, considering the tradeoff between performance and runtime, we choose to run the optimization stage for 3 times and a perturbation stage between 2 runs of the optimization stage

V. EXPERIMENTAL RESULT

We implemented our algorithm in C++. Our algorithms were tested against the IWLS 2005 benchmark suite [14], which was also used by [4][6]. The tests were run on a 2.8 GHz 1 GB RAM Linux OS system. We only consider the combinational parts of the benchmarks. For a fair comparison, we synthesized each benchmark into an And-Inverter Graph (AIG) with the *resyn2* script using the ABC [15] package (which performs local circuit rewriting optimization) and count the number of and-inverter nodes before and after optimization. The same procedure was also used in [4][6]. The correctness of our optimization method was verified by the equivalence checking tool in the ABC package.

Table 1 summarizes the experimental results. As we cannot obtain five of the benchmarks used in [4][6], we only tested the 18 available cases. For most cases, the initial synthesized circuit sizes are comparable to those reported in [6]. The difference may be due to the difference of the ABC version (we used the latest version of ABC in our work). Two cases have more than 30% difference.

The benchmarks are listed in the first column of the table. Columns N and Nr list the number of and-inverter nodes before and after optimization respectively. Column % shows the percentage of circuit size reduction. And Column T(s) shows the CPU time. The statistics of our approach, NAR [6] and the SAT-based node merging approach [4] are presented side by side. Since the authors of [4] did not report the node number after optimization, we only list the node reduction percentage and runtime. Due to the version difference of ABC for pre-optimization, the original number of nodes N is a little bit different to [6].

The experiments show that, by coupling rewiring and node merging techniques, the results can yield improvement nearly double of applying the node merging algorithm alone. We believe the improvement is mainly due to the effective ODC shifting algorithms proposed by this new scheme.

In our flow, we implemented our own version of the NAR and ECR algorithms. As we built the algorithms atop a complicated integrated database supporting operations on both logical and physical EDA operations, in a similar single NAR run, our NAR takes 10 times of CPU time of the original implementation in [6]. The CPU time spent in our flow is about 10 times of one NAR run, which is due to the 3 iterations of optimizations with a perturbation stage inserted in between. Besides, rewiring is intrinsically more complicated than node merging.

There are three iterations in our approach. In Table 2, we list the optimization details of each iteration. The statistics shows that, our approach already outperformed the previous node merging algorithm in iteration 1. The advantage of our approach over previous approaches is that the results can still be improved after a few iterations for perhaps an effective circuit perturbation capability. Nearly 3% further improvement could be achieved after 2 more iterations.

VI. CONCLUSION

Node merging algorithms have been a principal approach for area reduction in recent years. The drawback of such node-based algorithms is lacking the capability in maneuvering ODC shifting toward a desired direction for our needs. In this paper, we propose an approach that complements the node-based by wire-based logic synthesis algorithms for area reduction by adding the operations of effective ODC shifting. The scheme can find more node merging solutions and produce a double area reduction compared to those approaches employing node-based algorithms only.

We observe that, though not explicitly described in [4] and [6], the circuit depths would mostly increase while the area is reduced in both methods. However, without the related data published of these two methods, unfortunately we cannot make a fair comparison on the depth issue here.

In fact, rewiring can also be used to optimize circuit depth and timing attributes [16]. In our future work, we will extend our coupling flow to use node merging to reduce area while using rewiring to optimize delay at the same time.

REFERENCES

[1] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *DAC '06*, pp. 532-536.

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

[3] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't cares", *DAC '06*, pp. 229-234.

[4] 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.

[5] Y. C. Chen and C. Y. Wang, "Fast Detection of Node Mergers Using Logic Implications," in *Proc. Int. Conf. on Computer-Aided Design*, 2009, pp. 785-788.

[6] Y. C. Chen and C. Y. Wang, "Node Addition and Removal in the Presence of Don't Cares," in *Proc. DAC*, 2010, pp. 505-510.

[7] S.C. Chang, L.P.P.P. van Ginneken, and M. Marek-Sadowska, "Fast Boolean Optimization by Rewiring", in *Proc. Int'l Conf. Computer-Aided Design*, Nov. 1996, pp. 262–269.

[8] Y.M. Jiang, A. Krstic, K.T. Cheng, and M. Marek-Sadowska, "Post-layout Logic Restructuring for Performance Optimization", in *Proc. of Design Automation Conf*., 1997, pp. 662–665.

[9] K.T. Cheng and LA Entrena. Multi-level logic optimization by redundancy addition and removal. In *Design Automation, 1993, with the European Event in ASIC Design. Proceedings.[4th] European Conference on*, pages 373–377, 1993.

[10] C.W.J. Chang and M. Marek-Sadowska. Single-pass redundancy addition and removal. In *Proc. of the 2001 IEEE/ACM international conference on Computer-aided design*, page 609. IEEE Press, 2001.

[11] C.C. Lin and C.Y. Wang. Rewiring Using Irredundancy Removal and Addition. In *Proc. of Design Automation and Test in Europe*, pages 324–327, 2009.

[12] X. Yang, T.K. Lam, and Y.L. Wu, "ECR: A low complexity generalized error cancellation rewiring scheme," in *Proc. Design Automation Conference*, pp. 511–516, 2010.

[13] S.C. Chang, M. Marek Sadowska, and K.T. Cheng, "Perturb and simplify: Multilevel Boolean network optimizer," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*., vol. 15, pp. 1494–1504, Dec 1996.

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

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

[16] X. Wei, et al. "Mountain-mover: An intuitive logic shifting heuristic for improving timing slack violating paths." *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. IEEE, 2013.

Table 1. The experimental results of area reduction by using approaches in [4] and [6] and our approach

benchmarks	Our approach				[6]				[4]	
	N	Nr	%	T(s)	N	Nr	%	T(s)	%	T(s)
s9234*	809	776	4.1	12	1353	1323	2.2	0.2	1.2	8
pci_spoci_ctrl	824	591	28.3	221	878	757	13.8	0.4	9.2	6
s13207*	848	803	5.3	14	2108	2043	3.1	0.8	1.8	17
i2c	935	839	10.3	66	941	894	5	0.2	3.2	3
systemcdes	2442	2178	10.8	227	2641	2580	2.3	1.2	4.7	9
spi	3208	3045	5.1	415	3429	3383	1.3	5.6	1.3	84
tv80	7190	6445	10.4	7191	7233	6813	5.8	20.3	7.1	1445
s38584	7331	7156	2.4	1695	9990	9836	1.5	15.1	0.8	223
s38417	8045	7891	2	620	8185	8105	1.0	1.5	1	275
mem_ctrl	8578	6756	21.2	1914	8815	7287	17.3	13.8	18	738
systemcaes	9930	9664	2.7	3693	10585	10386	1.9	30.7	3.8	360
ac97_ctrl	10287	10185	1	728	10395	10364	0.3	3.1	2	188
usb_funct	13339	12537	6	2508	13320	12868	3.4	11.4	1.4	681
pci_bridge32	16423	16163	1.6	3589	17814	17599	1.2	19.7	0.1	1134
aes_core	20472	19412	5.2	5282	20509	20195	1.5	22.7	8.6	1620
b17	33219	30114	9.3	9096	34523	33204	3.8	205.5	1.6	5000
wb_conmax	41230	36293	12	8101	41070	38880	5.3	48.4	6.2	5000
des_perf	74546	67600	9.3	9372	71327	69421	2.7	84.7	3.7	5000
average			8.2				4		4.2	
total				54744				485.3		21796

* Benchmarks with large different size with [6] after pre-optimization

Table 2. Iterations of optimization in our approach

benchmarks	N	iteration 1		iteration 2		iteration 3	
		Nr	%	Nr	%	Nr	%
s9234	809	791	2.2	786	2.8	776	4.1
pci_spoci_ctrl	824	657	20.3	619	24.9	591	28.3
s13207	848	808	4.7	805	5.1	803	5.3
i2c	935	875	6.4	856	8.4	839	10.3
systemcdes	2442	2259	7.5	2207	9.6	2178	10.8
spi	3208	3156	1.6	3118	2.8	3045	5.1
tv80	7190	6644	7.6	6473	10	6445	10.4
s38584	7331	7190	1.9	7160	2.3	7156	2.4
s38417	8045	7937	1.3	7909	1.7	7891	2
mem_ctrl	8579	6889	19.7	6787	20.9	6756	21.2
systemcaes	9930	9712	2.2	9698	2.4	9664	2.7
ac97_ctrl	10287	10230	0.6	10193	1	10185	1
usb_funct	13339	12679	4.9	12561	5.8	12537	6
pci_bridge32	16423	16218	1.2	16174	1.5	16163	1.6
aes_core	20472	19766	3.4	19568	4.4	19412	5.2
b17	33219	31459	5.3	30432	8.4	30114	9.3
wb_conmax	41230	38385	6.9	36576	11.3	36293	12
des_perf	74546	71471	4.1	68574	8	67600	9.3
average			5.6		7.3		8.2