An Timing Driven Pseudo Exhaustive Testing for VLSI Circuits

S. C. Chang and J. C. Rau

Department of Computer Science and Information Engineering
National Chung-Cheng University
Chiayi, Taiwan, R. O. C.

Abstract

Because of its ability to detect all non-redundant combinational faults, exhaustive testing, which applies all possible input combinations to a circuit, has become a very attractive test method. However, the test application time for exhaustive testing can be very large. To reduce the test time, pseudo exhaustive testing inserts some *bypass storage cells* (*bscs*) so that the dependency of each node is within some predetermined value. Though bsc insertion can reduce the test time, it may increase circuit delay. In this paper, our objective is to reduce the delay penalty of bsc insertion for pseudo exhaustive testing. We first propose a tight delay lower bound algorithm which estimates the minimum circuit delay for each node after bsc insertion. By understanding how the lower bound algorithm lose optimality, we can propose a bsc insertion heuristic which tries to insert bscs so that the final delay is as close to the lower bound as possible. Our experiments show that the results of our heuristic are either optimal because they are the same as the delay lower bounds or they are very close to the optimal solutions.

1 Introduction

As the feature size of VLSI fabrication is shrunk to deep sub-micron, the traditional stuck-at fault model can no longer precisely model some more complex faults such as cross talk faults and charge-sharing faults in domino circuits. By applying to a combinational circuit all possible input combinations as test patterns, exhaustive testing, one method of *Built-In Self-Test* (BIST) [1][2][3], becomes very attractive because the exhaustive testing method can guarantee the detection of all non-redundant combinational faults.

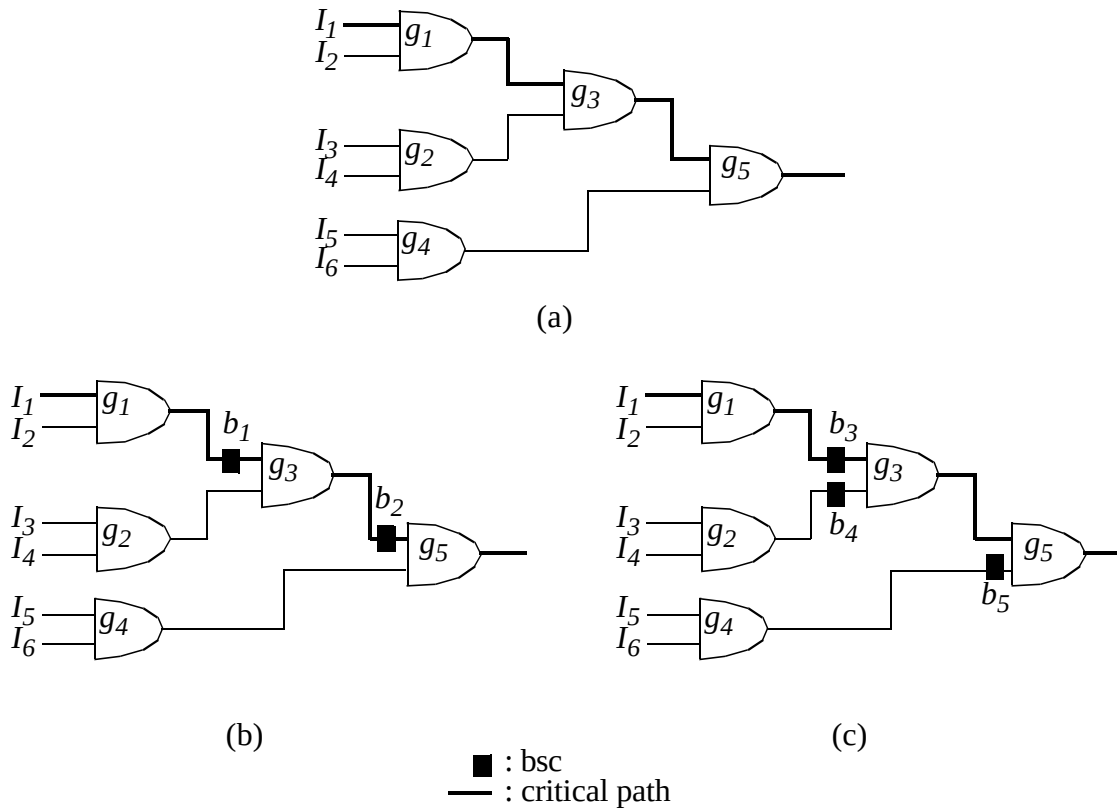

Fig. 1 : A circuit and its partition configurations.

Unfortunately, the test time of exhaustively testing a combinational circuit increases exponentially to the number of primary inputs (PIs) in the circuit, which makes this test method unpractical for circuits with large number of PIs. To tackle this problem, *pseudo exhaustive testing* [4][5][11][12][13][14][16][17] attempts to reduce test time without scarifying test quality. Among the techniques of pseudo exhaustive testing, [5][11][12][16][17] insert to the circuit some *bypass storage cells* (*bscs*) which are transparent in the normal mode and act as pseudo primary inputs and pseudo primary outputs (POs) in the test mode. It is said that a gate *depends* on a bsc (or a PI) if there is a path from the bsc (or the PI) to the node and there is no

other bscs in the path. After bsc insertion, the set of PIs and bscs which a node depends on is the *dependency* set of the node and the size is the *dependency* size. If the dependency size of a node is not greater than some value *k*, we can exhaustively test the node with 2^k test patterns. For example, the circuit of Fig. 1(a) originally has 6 PIs and 1 PO. In the test mode, we need 2^6 test patterns to exhaustively test the circuit. After inserting two bscs b_1 and b_2 into the circuit shown in Fig. 1(b), each (pseudo) PO depends on at most *3* PIs or bscs. Hence, in the test mode, we only need at most 2^3 test patterns to exhaustively test the circuit using an appropriate BIST configuration.

The test time to exhaustively test a node depends on its dependency size of the node. In order to exhaustively test a circuit using 2^k patterns, some bscs may be inserted so that the dependency of each node is less than or equal to some pre-determined value *k*, called the *dependency constraint*. On the other hand, adding bscs can increase the circuit size. Most previous work [5][11][16][17] attempts to reduce the number of bscs under a given dependency constraint. Basically, based on the area consideration, the bsc insertion methods for pseudo exhaustive testing can be divided into two categories. One uses the constrained partitioning strategy (CPS) which partitions a circuit into several disjoint sub-circuits [5][16][17]. The other uses the un-constrained partitioning strategy (UPS) which makes no such constraint [11]. In this paper, our strategy of bsc insertion is based on the first one. In addition to the area overhead of inserted bscs, the inserted bscs may be placed on critical paths and therefore can worsen the circuit delay. Consider the same circuit of Fig. 1(a) and assume the path I_1->*g1*->*g3*->*g5* is critical. Consider a solution for exhaustively testing the circuit in Fig. 1(b). Despite the inserted bscs do not affect circuit's original function in the normal mode, the circuit delay is increased due to the placement of two bscs on the critical path. Instead, if we insert bscs as in Fig. 1(c), where only one bsc is placed on the critical path, the critical path delay is less than the delay in Fig. 1(b). Note that the maximum dependency size of both BIST configuration is 3.

The purpose of this paper is to minimize the delay penalty resulted from inserting bscs under a given dependency constraint. Our algorithm consists of two phases. In the first phase, we discuss an algorithm which finds a *lower bound* of the minimum delay for a node after inserting bscs. This lower bound allows one to claim optimal if a delay result is the same as the lower bound. With the lower bound information and the understanding of how the optimality may not be achievable, we then develop a heuristic to guide bsc insertion so that the delay

result (after inserting bscs) is as close to the low bound as possible. Both the lower bound algorithm and the bsc insertion heuristic make use of the minimal-cut-maximal-flow algorithm. Our experimental results show that for many benchmark circuits, our heuristic achieves the same delay results as the lower bounds; i.e., optimal solutions.

This paper is organized as follows:Section 2 describes an implemenation of pseudo exhaustive testing and Section 3 describes the delay model with the associated delay computation. Section 4 describes an algorithm computing a delay lower bound for each node under the dependency constraint. The approximation algorithm to guide the bsc insertion process is developed in Section 5. Finally, the experimental results and conclusions are given in Sections 6 and 7.

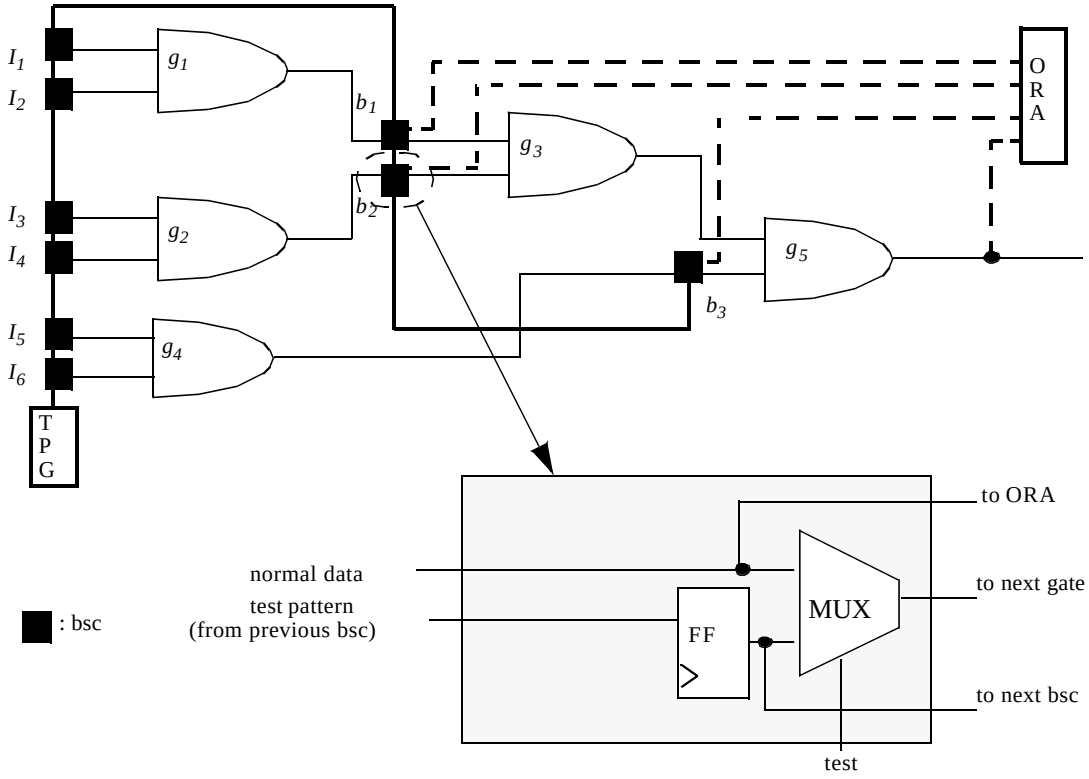

Fig. 2 : An implemenation of pseudo exhaustive testing and the structure of a bsc.

2 An Implementation of Pseudo Exhaustive Testing

We now describe an implementation of the pseudo exhaustive testing and the structure of a bsc by an example. Consider the circuit in Fig. 2. To implement the pseudo exhaustive testing, two components -- a test pattern generator (TPG) and an output response analyzer (ORA) are embedded into the circuit. Normally, The TPG is a linear feedback shift register [1] whose pur-

pose is to generate pseudo exhaustive test patterns and the ORA is a Multiple Input Signature Register [1] whose purpose is to collect and compress the output response. A bsc is inserted at the fanout of each PI. Three additional bscs, b_1, b_2, and b_3 are inserted to satisfy the dependency of *3*. All bscs are constructed as a scan chain and is connected to the TPG. The basic structure of a bsc contains a mux and a flip-flop in Fig. 2. During the normal mode, the signal, *test* is set to 0 and all bscs become transparent; that is, they do not affect the circuit' s function. In each cycle of the test mode (*test = 1*), a test pattern is serially shifted to the flip-flops of bscs, and the output responses is directed to the ORA which is analyzed later.

3 Delay Model and Delay Computation

In this section, we describe the delay model of bsc insertion and the associated delay calculation. Based on this model, we present in the subsequent sections an algorithm which attempts to find an optimal way of bsc insertion.

A combinational circuit can be represented as a directed acyclic graph $G = (V, E)$, where V consists of all the gates and E consists of directed edges such that a directed edge e **(***v->u)* is in E if v is an input of u. The edge *e (v-> u)* is a *fanin edge* of u and node v is a *fanin node of u*. In addition, node w or edge e is a *transitive* fanin of node u if there exists a path from w or e to u. The set of nodes which are transitive fanins of node v is referred to as the *input cone* of node v.

To fulfill the dependency constraint, some bscs may be inserted. Before inserting bscs, we assume that the circuit is technology mapped and wire delay information is available. Let the delay of edge e be $d(e)$ which contains the wire delay and the delay of its fanout gates. For simplicity, we assume that the delay of a node is added to the delay of its fanin edges. This will not affect the calculation of the circuit delay. In addition, each node n is assigned a value $A(n)$, called the *arrival time* or the delay, at which the signal it generates is stable [15]. Normally, the arrival time of each PI is set to zero. We can recursively calculate the arrival time $A(e)$ of edge e and the arrival time $A(n)$ of node n by the following two formulae:

$$A(e_i) = A(n_i)+d(e_i) \text{ where } n_i \text{ is the source gate of edge } e_i \quad \text{(EQ 1)}$$

$$A(n_j) = Max\{A(e_k)+d(e_k) \mid e_k \text{ is a fanin edge of } n_j\} \quad \text{(EQ 2)}$$

When a bsc is inserted on edge e, our delay model assumes that one *constant* delay penalty, $d_e(bsc)$, is added to the delay of edge e. Hence, EQ 1 can be rewritten as

$A(e) = A(n) + d(e) + d_e(bsc)$ where n is the source gate of edge e and $d_e(bsc)$ is the delay penalty of a bsc on e (EQ 3)

Example 1: Consider the circuit in Fig. 3(a). Assuming that edge delay $d(e)$ and bsc delay penalty $d(bsc)$ both are *1*, by EQ 1, EQ 2, and EQ 3, we can obtain the delay of each node shown outside the circle. In Fig. 3(a), without inserting any bscs, the delay of node g_5, $A(g_5)$ is *3* because the longest path from PIs is I_1->g_1->g_2->g_5 which contains 3 edge delays. In Fig. 3(b), after inserting bscs b_1, b_2, and b_3, the delay of g_5, $A(g_5)$ becomes *4* because of addition bsc delay.

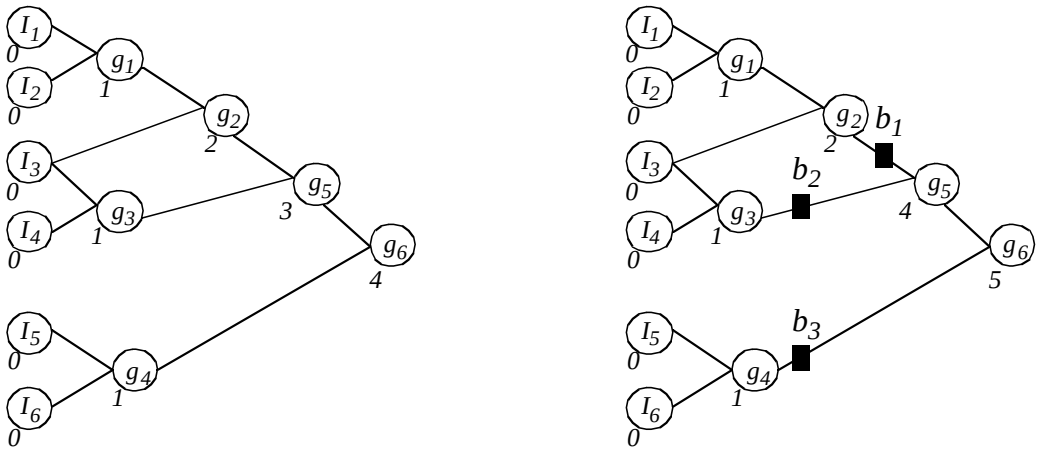

Fig. 3 : Delay model and delay computation.

The constant delay penalty, $d_e(bsc)$, may be different on various edge e where a bsc is inserted and can be obtained before applying the above computation. For simplifying the following discussion, we further assume that the delay penalty of inserting a bsc is the same on different edges, *i.e.*, $d_e(\text{bsc}) = d(bsc)$ for all edges. Our algorithm can be easily extended to the case when the delay penalty is different. The reason to assign one constant delay penalty to an inserted bsc is that in the linear delay model [9], where

$$delay = intrinsic\ delay + resistance * load, \quad \text{(EQ 4)}$$

the incurred delay penalty can be pre-computed since the values of all the variables in EQ 4 can be obtained in advance. Despite the fact that our delay model might not be accurate for dealing with other complex delay models, the presented algorithm still can give a good approximation solution to reduce the delay penalty of inserted bscs.

4 Delay Lower Bound Computation

As mentioned in Section 2, when a bsc is inserted on some edge e, the constant delay penalty $d(bsc)$ is added into the delay of every path passing through edge e. Our objective is to minimize the circuit delay (critical path delay) of the normal-mode circuit after inserting bscs

under the pre-determined dependency constraint k. The proposed algorithm consists of two phases. In the first phase, called the labeling phase, for each node, we compute a label which can accurately estimate the minimum delay under the dependency constraint. A node's label computed by our algorithm is always less than or equal to the minimum delay of the node, i.e., a lower bound. And, in the second phase, the bsc insertion phase, we insert bscs so that each node's final delay can be as close to its label as possible. We describe the labeling phase in this section, and the bsc insertion phase in the next section.

We first give some definitions used in our discussion. For simplicity, we assume that a bsc has been inserted on the fanout stem of each PI and let the *dependency set* of node n, denoted $dep(n)$, be the set of bscs on which node n depends and $|dep(n)|$ be the *size* of $dep(n)$. The bscs of $dep(n)$ can be viewed as a "cut set" which separates node n from the PIs. If $|dep(n)| \leq k$, the set of nodes on the fanout stems of which the bscs of $dep(n)$ are inserted are called a k-cutset of node n. For example, in Fig. 1(c), the bsc set $\{b_3, b_4, b_5\}$ is g_5's dependency set and the node set $\{g_1, g_2, g_4\}$ is g_5's *3*-cutset.

The label of node n, $l(n)$, is the (estimated) minimum delay of node n under dependency constraint k. The difference between the label of a node and the delay (arrival time) of a node is that the label is an estimated value which will be shown to be a lower bound of the optimal delay for the node. In order words, the label of a node is always less than or equal to the delay of the node. The initial labels of PIs are set to 0 and the labels are computed in the topological order from PIs to POs. Therefore, before calculating the label of node n, all the labels in node n's input cone have been obtained. In addition, when computing node n's label, all the labels in its input cone are assumed to have values. This assumption will not cause any error since a label is a lower bound estimation. The basic steps of computing node n's label are as follows. First, we determine a possible range of node n's label from the labels of its immediate fanins. If $d(bsc) = 1$, we show that there are only two possible values for node n's label, the smaller value of which is called *best_label* and the other is *best_label + 1*. Then, we try to find whether it is possible to have a k-cutset which can result in the *best_label*. If there is such a k-cutset, we set the label of node n to be *best_label*, otherwise, we set the label of node n to *best_label + 1*. In the following, we discuss an important property: the possible range of a node's label can be determined from the labels of its (immediate) fanin nodes.

Definition 1: The *best_label*, $bl(n)$, of node n is defined to be

$$bl(n) = \text{MAX}\ \{l(n_i)+d(e_i) \mid \text{where } e_i\ (= n_i\text{->}\ n) \text{ is a fanin edge of } n\}. \quad \text{(EQ 5)}$$

Lemma 1: If labels of node *n*'s immediate fanins are optimal delays, the label $l(n)$ of node *n* has the property:

$$bl(n) \le l(n) \le bl(n)+d(bsc). \quad \text{(EQ 6)}$$

Proof. We prove $bl(n) \le l(n)$ by contradiction. Because $bl(n)$ is the maximum value of $l(n_i) + d(e_i)$ for each fanin edge e_i, suppose edge $e_2(= n_2\text{->}n)$ is such a fanin edge of node *n* that $bl(n) = l(n_2) + d(e_2)$ in Fig. 4. Let us prove by contradiction. Assume that there exists an optimal *k*-cutset *Cut* of node *n* so that $bl(n) > l(n)$. Since n_2 is a fanin node of *n*, the *k*-cutset *Cut* must be also a *k*-cutset of n_2, as shown in Fig. 4. If we use *Cut* as a *k*-cutset of n_2, then

$$n_2\text{'s delay} = n\text{'s delay} - d(e_2) = l(n) - d(e_2) < bl(n) - d(e_2) = l(n_2)$$

Therefore, if *Cut* is a k-cutset for n_2, we have n_2's delay $< l(n_2)$ which contradicts the fact that the label of a node is the minimum value under the dependency constraint *k*.

It is easy to prove $l(n) \le bl(n)+d(bsc)$. If bscs are inserted on all the fanin nodes of node *n*, the delay of node *n* is equal to $bl(n)+d(bsc)$. **Q.E.D**.

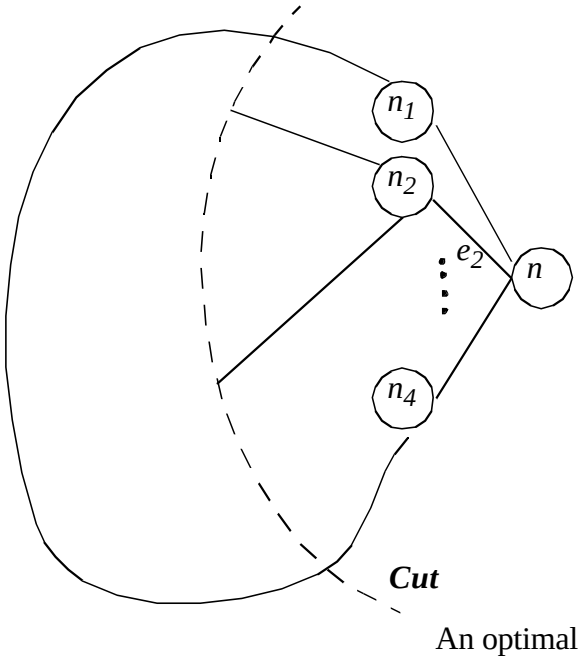

Fig. 4 : Bounds for the label of a node.

If the labels of all fanins are indeed optimal, $bl(n) + 1$ is always achievable. We now use an example to illustrate Lemma 1. Consider the same circuit as in Example 1. We duplicate the circuit in Fig. 5 where the number outside a node is the corresponding node' s label which we are about to compute. Again, let the delay of an edge, $d(e)$, and the bsc delay penalty, $d(bsc)$,

be both 1. We also assume the dependency constraint is *3*. Initially, the labels of all the PIs are set to *0*. Our labeling algorithm will start from PIs to POs. For node g_1, it is easy to see that there is no need to insert bscs for node g_1 so the label of g_1 is equal to *1*. Similarly, for g_2, g_3, and g_4, we have the labels, $l(g_2) = 2$, $l(g_3) = 1$ and $l(g_4) = 1$. The labels of these nodes, g_2, g_3, and g_4, are the same as their delay (arrival time) in Fig. 3(a) of Example 1. For node g_5 which has 4 PIs in its transitive fanins, we do need to insert bscs for g_5. Since g_5' s immediate fanin nodes are g_2 and g_3, according to Lemma 1, we have $bl(g_5) = \text{MAX}\{l(g_2)+d(e), l(g_3)+d(e)\} = \text{MAX}\{2+1, 1+1\} = 3$. Then we can find out that the range of g_5's label is $3 = bl(g_5) \le l(g_5) \le bl(g_5)+d(bsc) = 4$. There are only two choices for g_5' s label, i.e., either $l(g_5)=3$ or $l(g_5)=4$. In later discussion, we will show that there does not exist a solution which makes the delay of g_5 to be *3* under the dependency constraint *3*, so g_5's label must be *4*. Continue to consider g_6 which has *6* PIs in its transitive fanins. We also need to insert bscs for it. Since g_6' s immediate fanin nodes are g_4 and g_5 with labels *1* and *4*, respectively, we have $bl(g_6) = \text{MAX}\{l(g_4)+d(e), l(g_5)+d(e)\} = \text{MAX}\{1+1, 4+1\} = 5$. By Lemma 1, the range of g_6 can be determined to be $5 = bl(g_6) \le l(g_6) \le bl(g_6)+d(bsc) = 6$. We will show that there is one solution which makes g_6's delay to be *5*, so its label $l(g_6)$ is equal to *5* (= $bl(g_6)$). One of bsc insertion solution to achieve $l(g_6)$=5 is shown in Fig. 3(b) of Example 1.

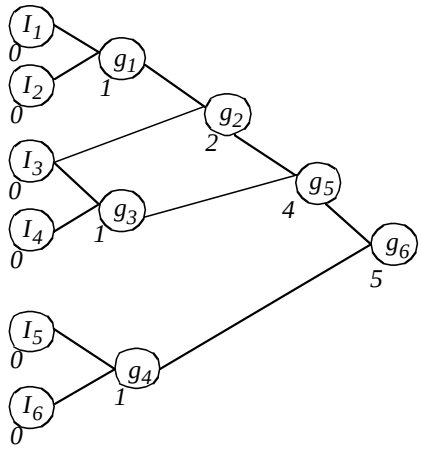

Fig. 5 : Computation of nodes' labels with the dependency constraint *3*.

From Lemma 1, we can obtain a possible range of node *n*' s label, $l(n)$ whose best value is $bl(n)$ and worst value is $bl(n)+d(bsc)$. Without losing generality, let us assume that the delay penalty $d(bsc)$ and the delay $d(e)$ of each edge e are integers. If the delay penalty $d(bsc)$ is *1*, according to EQ 6, we can have only two choices for node *n's* label $l(n)$: one is $bl(n)$ and the other is $bl(n)+1$. Therefore, to obtain *n*'s label, we only need to decide whether $bl(n)$ is achievable. If the answer is "yes", then *n*' s label is equal to $bl(n)$. Otherwise, *n*'s label is equal to $bl(n)+1$. When the delay penalty $d(bsc)$ is not equal to *1*, we can use the binary search method to gradually find $l(n)$. For example, suppose that $d(bsc)$ is *4*. We first check the feasibility of

$bl(n)+2$ which is the median between $bl(n)$ and $bl(n)+4$. If it is feasible, $l(n)$ must be between $bl(n)$ and $bl(n)+2$. Otherwise, $l(n)$ must be between $bl(n)+2$ and $bl(n)+4$. When $bl(n)+2$ is feasible, we can continue to check the feasibility of $bl(n)+1$ which is the median between $bl(n)$ and $bl(n)+2$. The binary search can continue until $l(n)$ is obtained.

Let us assume $d(bsc) = 1$ in the following discussion; now, the problem of finding node n's label has been changed to the yes/no problem of determining whether $bl(n)$ is achievable. Note that $bl(n)+1$ is always achievable by inserting bscs on the fanin edges of node n. Luckily, this yes/no problem has an approximate solution using the max-flow min-cut (MFMC) algorithm. Before describing the use of MFMC algorithm, we discuss how to exclude nodes which should not be selected into a k-cutset to achieve $l(n)=bl(n)$.

To compute the label of node n, suppose node c is selected into an optimal k-cutset. A bsc is inserted at the fanout of each node in the k-cutset. In Fig. 6, the longest path delay from a PI pi through node c to node n is $l(c)+d(c->n)+d(bsc)$, where $d(c->n)$ is the longest path delay from c to n without involving any bsc delay penalty. If node c is selected into an optimal k-cutset of node n, we have $l(c)+d(c->n)+d(bsc) \le l(n)$. Therefore, to achieve the *best_label* $bl(n)$ of node n, if node c (other than PIs) is selected into the k-cutset, its label, $l(c)$, must be less than or equal to the value of $bl(n)-d(c->n)-d(bsc)$. In addition, because a PI can be considered as a bsc, we do not need to insert bsc at the fanout of a PI. We define a node whose label satisfies EQ 7 and EQ 8 to be a *timing feasible* node for $bl(n)$.

$$l(c) \le bl(n)-d(c->n)-d(bsc) \text{ if } c \text{ is not a PI} \quad \text{(EQ 7)}$$

$$l(pi) \le bl(n)-d(pi->n) \quad \text{(EQ 8)}$$

In order to achieve $l(n) = bl(n)$, each node in the k-cutset must be a timing feasible node for $bl(n)$. Note that, for each node c in the input cone of node n, the values of $l(c)$, $d(c->n)$, $d(bsc)$ and $bl(n)$ can be pre-computed. Hence, we can easily determine whether node c is timing feasible for $bl(n)$.

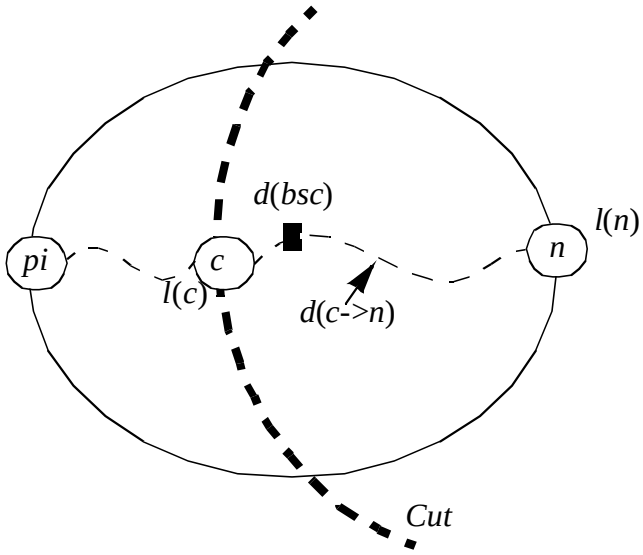

Fig. 6 : The delay of a path from *pi->c->n*.

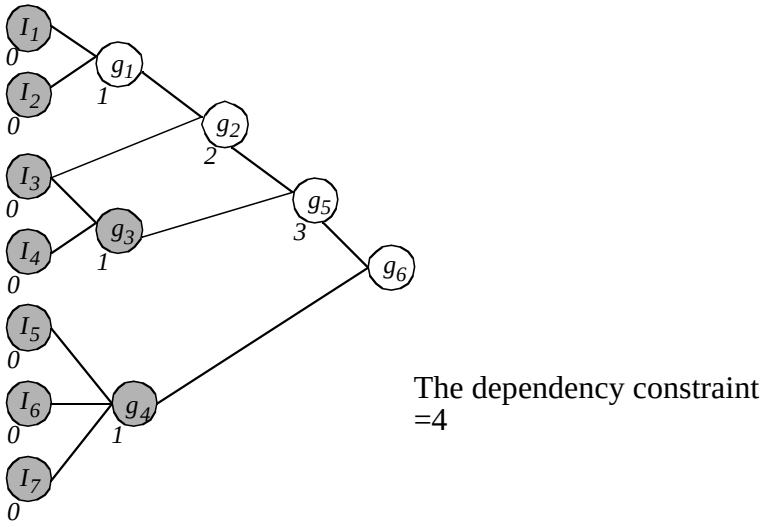

Fig. 7 : The timing feasible nodes for $bl(g_6)=4$.

For example, in Fig. 7, let gates g_1 to g_5 have been labeled and label of g_6 is under consideration. Since $l(g_4)=1$ and $l(g_5)=3$, according to EQ 5 and EQ 6, we have $bl(g_6) = 4$ and $4 \le l(g_6) \le 5$. In order to achieve $l(g_6)=bl(g_6)=4$, we can find that node g_4 is a timing feasible node because $l(g_4) \le bl(g_6)-d(g_4->g_6)-d(bsc)$, *i.e., 1≤4-1-1*. Similarly, node g_3 is also a timing feasible node for $bl(g_6) = 4$. On the other hand, node g_5 is not a timing feasible node for $bl(g_6)=4$ because $l(g_5) > bl(g_6)-d(g_5->g_6)-d(bsc)$, *i.e., 3>4-1-1*. Also, for PIs, we consider them as bscs. One can check that all PIs are timing feasible from EQ 8. In Fig. 7, all timing feasible nodes for $l(g_6)=4$ are marked.

Now, let us consider to check the feasibility of *bl(n)* for node *n*' s label. Assume all the labels of node *n*' s input cone *Cone(n)* except *n*, are already computed. Also, by applying EQ 7 and EQ 8, we can find all timing feasible nodes in *n*'s fanin cone. With these information, we are ready to use the MFMC algorithm to resolve the yes/no problem, *i.e.* to decide whether there is a *k*-cutset containing only timing feasible nodes for *bl(n)*.

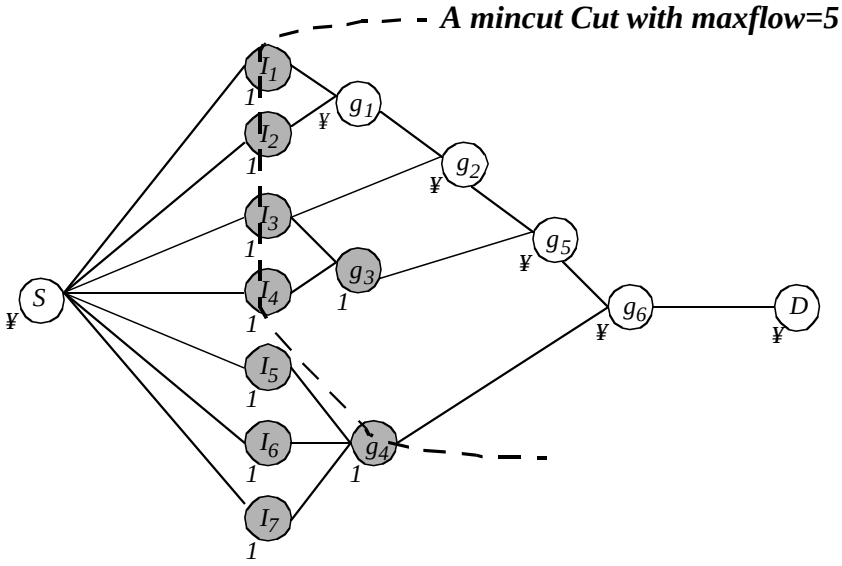

Fig. 8 : The weighted DAG for *Cone*(g_6) of Fig. 7 and a mincut *Cut* with maxflow 5.

Although the traditional MFMC algorithm performs the "cut" operation on edges of a graph, it can be easily transformed to perform the "cut" operation on nodes. To apply the MFMC algorithm, we transform *Cone*(*n*) into a weighted graph as follows. We first add two nodes, the source node *S* and the destination node *D*. We also add the edges connecting *S* to each PI and the edges connecting node *n* to *D*. Then, the weight of each timing feasible node for *bl(n)* is set to *1* and others to *infinite*. When applying the MFMC algorithm on this transformed graph, the MFMC algorithm tries to select a cutset whose total weight is minimum. The weights which we assign on the transformed graph cause the MFMC algorithm to select (cut) only timing feasible nodes because the weights of timing feasible nodes are much smaller than others. In addition, the MFMC algorithm will try to find the minimum number of possible timing feasible nodes to form a cutset. If the size of the selected cutset is greater than *k*, it means that there does not exist a *k*-cutset which contains only timing feasible nodes, *i.e.*, $l(n)=bl(n)$ is not feasible.

We now use an example to summarize the process of the labeling algorithm. Consider to label g_6 in Fig. 7. First, we would like to know whether it is possible to have $l(g_6) = bl(g_6) = 4$. Then, for nodes in the input cone of g_6, we find that only nodes {g_3, g_4, PIs} are timing feasible. After that, we transform the graph in Fig. 7 to the weighted graph in Fig. 8 where the weights of {g_3, g_4, all PIs} are assigned to 1 and others to infinite. The MFMC algorithm will then be applied on Fig. 8. Suppose a minimum weight cut shown as the dash line in Fig. 8 is

returned. Since the minimum weight is 5, it is not possible to have a 4-cutset to achieve the goal of $l(g_6)=bl(g_6)=4$. Since $l(g_6)=4$ is not achievable, we must have $l(g_6)=5$ according to Lemma 1 that $4 \le l(g_6) \le 5$. Therefore, the label $l(g_6)$ is set to 5.

During the labeling of node n, we use the MFMC algorithm to decide whether $l(n)=bl(n)$ is feasible or not. If the cut size returned from MFMC is greater than k, we conclude that there does not exist a k-cutset consisting of only timing feasible nodes so $l(n)=bl(n)$ must not be feasible. Therefore, the label of node n is set to $bl(n)+1$. On the other hand, if the cut size from MFMC is less than or equal to k, in the labeling algorithm, we set the label of node n to $bl(n)$. However, the assignment of $l(n)=bl(n)$ for the later case (cut size $\le k$) may be too optimistic. The reason is as follows. Even though each node in the k-cutset suggested by MFMC is timing feasible, all together they may not be timing feasible for $l(n)=bl(n)$. For example, consider finding the label of node g_9 in Fig. 9, where all highlighted nodes are timing feasible for $l(g_9) = bl(g_9) = 6$. Here, we let the dependency constraint be 4. For the example, the MFMC algorithm may return the 4-cutset $\{g_7, g_4, g_3, g_6\}$ for g_9 as shown in Fig. 9. Since the cutset size is equal to the dependency constraint, 4, the labeling algorithm sets $l(g_9)$ to 6. Note that, in the configuration of this 4-cutset, g_6 is in the fanout of g_3 so there are two bscs in the critical path I_1->g_1->g_3->g_6->g_8->g_9. In this special case, after inserting bscs for this 4-cutset, the delay of g_9 is 7 ($=d(I_1\text{->}g_1)+d(g_1\text{->}g_3)+d(g_3\text{->}g_6)+d(g_6\text{->}g_8)+d(g_8\text{->}g_9)+2*d(bsc) = 1+1+1+1+1+2$) which is larger than its label, 6. In this example, the 4-cutset has the configuration that g_3 is in the fanin of g_6. If the configuration of the k-cutset contains more than one node in a path as in the example, then the labeling algorithm may wrongly estimate the delay of node n. We call such situation the *bsc chain effect.* The details of the bsc chain effect and a method to alleviate the effect is discussed in Section 5.

Lemma 2: Suppose the labels of all the nodes in n's input cone are the same as the optimal delays under the dependency constraint, k. If the k-cutset of node n suggested by the MFMC algorithm has no bsc chain effect, then n's label $l(n)$ set by the labeling algorithm is optimal.

Proof. For any node n, we have $bl(n) \le l(n) \le bl(n)+1$ (Lemma 1). We finish the proof by two cases. Case 1: If the MFMC algorithm cannot suggest a k-cutset for $bl(n)$, then we insert bscs on the fanout stems of n's immediate fanin nodes. Hence, $l(n)$ is equal to $bl(n)+1$ which is the optimal delay under the dependency constraint k. Case 2: *Cut* is a k-cutset suggested by the MFMC algorithm for $bl(n)$, but it has no bsc chain effect. For any node c in *Cut*, there is at most one bsc on any path from c to n, hence, we have $bl(n) = \text{MAX}\{l(c) + d(c\text{->}n)+1\} \le \text{MAX}\{A(c) + d(c\text{->}n)+1\} = A(n)$. In this case, we set $l(n)$ to $bl(n)$ which is the optimal delay under the dependency constraint k. **Q.E.D.**

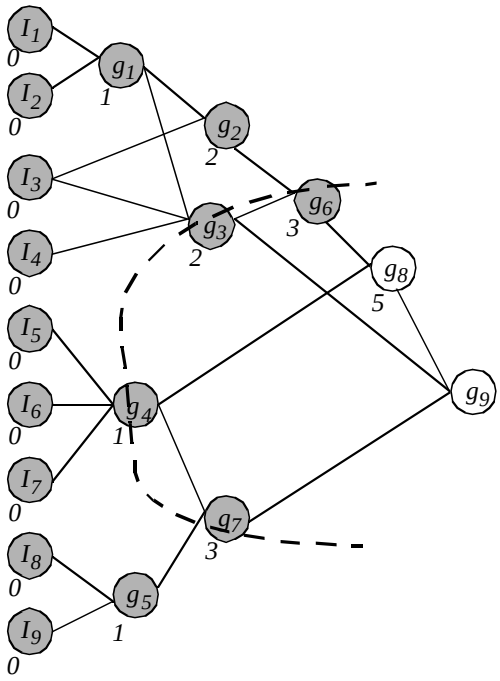

Fig. 9 : A *4*-cutset *Cut* with the bsc chain effect.

5 Bsc Insertion

Our final object is to insert bscs so that the whole circuit delay (the critical path delay) is as small as possible. Hence, after all labels of nodes have been determined, our basic strategy is to maintain the labels of nodes in the critical paths as inserting bscs. The bsc insertion process is iteratively performed from POs to PIs. Initially, we put all POs into a processing list. In each iteration, we select a node *n* with the maximal label from the list. After finding a *k*-cutset for node *n*, we insert bscs at the fanout stems of nodes in *k*-cutset. Then, node *n* is removed from the processing list and nodes in the *k*-cutset are added into the list. This process continues until there is no node in the processing list.

For example, consider the circuit of Fig. 10. Assume the dependency constraint is 3 and each node's label has been obtained and is shown outside the circle. First, we process the fanin cone of g_7 and the MFMC algorithm suggests the *3*- cutset $Cut_{g7} = \{g_3, g_4, g_6\}$. After inserting bscs on the fanouts of nodes in Cut_{g7}, we treat these nodes g_3, g_4, and g_6 as POs and put them into the processing list. Next, we process g_6's fanin cone, and obtain the 3-cutset $Cut_{g6} = \{g_2, g_5\}$. Similarly, we insert bscs for Cut_{g6} and put them into the processing list. Now, the processing list consists of g_5, g_3, g_4, and g_2 whose dependencies are all less than or equal to *3*. Therefore, we do not insert bscs for these nodes and the process ends.

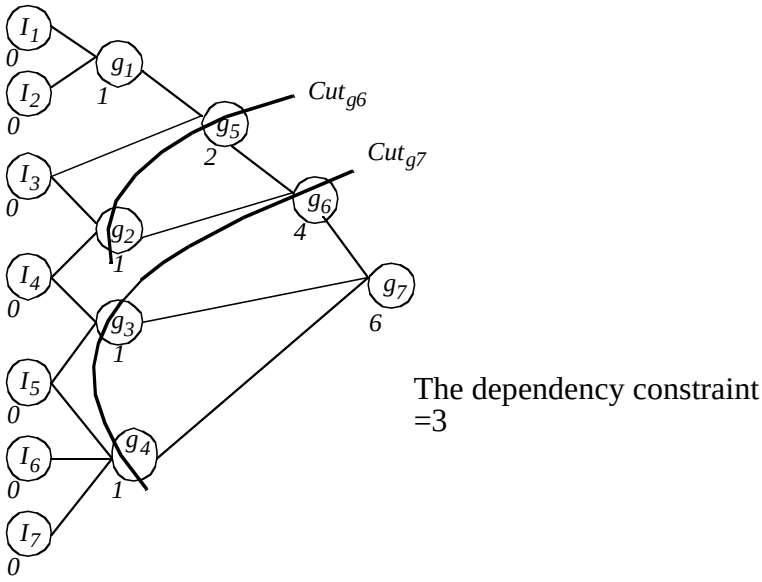

Fig. 10 *:* Insertion of BSCs.

While processing node *n*, we attempt to obtain a *k*-cutset for *n* so that the delay of node *n* is the same as its label. This *k*-cutset can be obtained by applying the MFMC algorithm as in the labeling phase. However, as mentioned in Section 4, after inserting bscs at the fanouts of nodes in the *k*-cutset, the delay of node *n* may be larger than its label *l*(*n*). In this section, we propose some heuristics to alleviate this inconsistency problem. When the *k*-cutset suggested by the MFMC algorithm cannot achieve the label, our heuristics try to find other *k*-cutsets which hopefully can achieve the label. Note that, since our final objective is to minimize the whole circuit (critical path) delay, we allow the delay of nodes in the non-critical paths to be larger than their labels. Note that, since our final objective is to minimize the whole circuit (critical path) delay, we allow the delay of nodes in the non-critical paths to be larger than their labels.

In the following subsections, we discuss two effects which may cause some node' s delay to be larger than its label after bsc insertion. These two effects are (1) *the bsc chain effect* and (2) *the interference effect.* In the subsequent subsections, these two effects are explained in detail and the heuristics to alleviate these two effects are also discussed.

5.1 The Bsc Chain Effect

Consider a *k*-cutset for node *n*, if the configuration of the *k*-cutset contains more than one bsc in a path, then the labeling algorithm may wrongly estimate the delay of node *n*. We call such situation the bsc chain effect, the reason of which is that the equation of EQ 7 always adds one bsc delay penalty between node *n* and another node in the *k*-cutset while there may be more than one bscs in between. However, there may exist other *k*-cutsets which can cause *n*'s delay

to be the same as its label. In this section, we propose heuristics to search other *k*-cutset solutions. Without going into the detail, we use an example to demonstrate our basic ideas.

Consider again Fig. 9, to fulfill the label of node g_9, the MFMC algorithm may suggest the *4*-cutset *Cut* = {g_3, g_4, g_6, g_7} based on the labeling algorithm. Instead of using the *4*-cutset *Cut* from MFMC, we may select another *4*-cutset *Cut'* = {g_2, g_3, g_4, g_7} as shown in Fig. 11. In the new configuration, there is no bsc chain effect and one can find the delay of g_9 is 6 which is the same as the label. This new *4*-cutset can be obtained by replacing g_6 with its fanin nodes g_2 and g_3. Such a way of replacing a node with its (immediate) fanin nodes in the *k*-cutset is called *the pushing-toward-input technique.*

However, the technique may not be applied to all nodes in the *k*-cutset. For a node *m* in the *k*-cutset, to apply the pushing-toward-input technique, two constraints must be satisfied. First, all fanin nodes of *m* are timing feasible nodes. Recall that a node *c* is a timing feasible node for *l(n)=bl(n)* if its label *l*(*c*) is less than or equal to "*bl*(*n*)-*d*(*c*->*n*)-*d*(*bsc*)". Secondly, the size of the resulting new cutset is not larger than *k*; i.e., it is still a *k*-cutset. Otherwise, applying arbitrarily the pushing-toward-input technique may incur larger delay or may not satisfy the dependency constraint. Return to the case of Fig. 9 where both g_2 and g_3 are timing feasible, we can apply the pushing-toward-input technique to replace g_6 with g_2 and g_3 and obtain another 4-cutset *Cut'* = {g_2, g_3, g_4, g_7} shown in Fig. 11. Because all nodes in *Cut'* are timing feasible and there exists no bsc chain effect, the delay of g_9 resulted form *Cut'* can achieve its label.

4-cutset *Cut'* of g_9

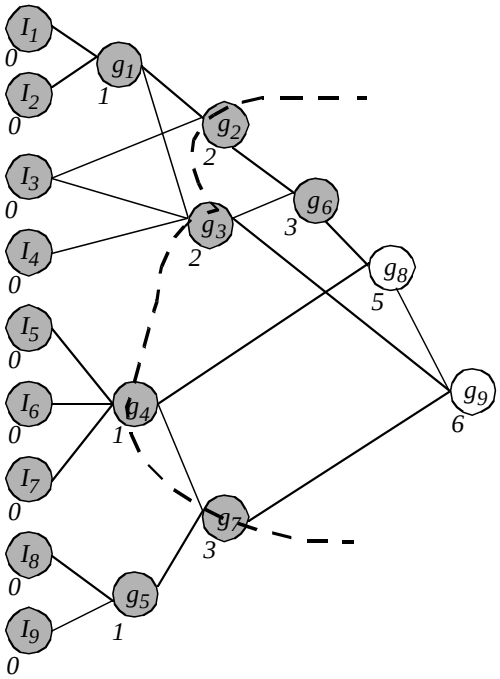

Fig. 11 : A *4*-cutset *Cut'* without the bsc chain effect.

5.2 The Interference Effect

In this section, we discuss the other effect, the *interference effect,* which may also cause some node' s solution from the MCMF algorithm to be larger than its labels. Basically, the interference effect comes from the sequential processing of nodes' labels. During the labeling phase, we evaluate a node' s label only based on the labels of nodes in its input cone. However, a node's label may be interfered by the *k*-cutsets of nodes outside its input cone. Consider two fanin cones of nodes n_1 and n_2 in Fig. 12(a). Suppose n_1 is processed first and we find a *k*-cutset C_1 for node n_1. Then, we process node n_2 and also find a *k*-cutset C_2 for n_2. Because of additional bscs from C_2 are inserted between n_1 and C_1, the delay of n_1 may be increased accidentally. Therefore, during the labeling phase, ignoring the effect of nodes outside n_1' s fanin cone may cause n_1' s label to be too optimistic.

Similar to the bsc chain effect, it is possible that there exist other *k*-cutsets which do not have the interference effect. For example, consider again the interference effect of Fig. 12(a). Assume that the *k*-cutset C_1 for node n_1 is chosen first. While finding a *k*-cutset C_2 for node n_2, there may be four possible configurations shown in Fig. 12. Arbitrarily selecting a *k*-cutset C_2 for n_2 as in (a) and (b) may cause the delay of n_1 to be larger than its label. However, if the configuration of a *k*-cutset C_2 for node n_2 is the same as in (c) and (d), there will be no interference effect for node n_1. Therefore, our algorithm will intelligently select *k*-cutsets so that there will be no interference effect, if possible.

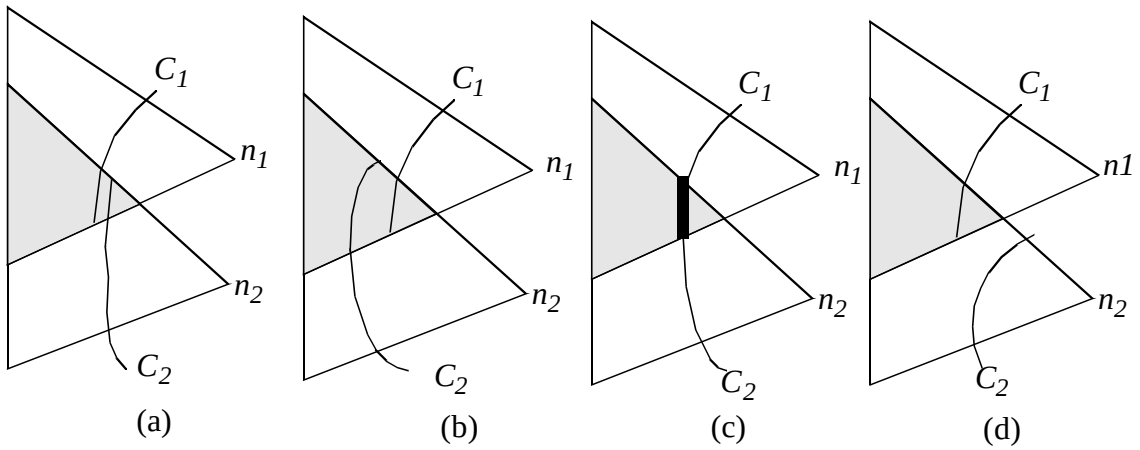

Fig. 12 : Interference effect between different cutsets.

Recall that our algorithm generates a weighted graph in which a timing feasible node is assigned weight 1 and others are assigned infinite; then, the MFMC algorithm selects a cutset from the weighted graph with minimum total weight. For leading MFMC to re-suggest another *k*-cutset solution for n_2, the heuristic **re-weights** some nodes in the fanin cone of node n_1 according to the following two re-weighting rules:

1. Re-assign a weight much less than the initial weight to each node in the *k*-cutset C_1.

2. Re-assign infinite weights for "critical" nodes which will cause the delay of n_1 to increase.

Again, suppose C_1 is already identified. The weight re-assignment of Rule 1 is to favor the nodes in C_1 when choosing a *k*-cutset C_2 for n_2 as in Fig. 12(c). The weight re-assignment of Rule 2 forbids selection of critical nodes in the input cone of n_1 so that the delay of n_1 will not be enlarged by C_2 as in Fig. 12(d).

The re-weight technique may not always be successful. However, since our final goal is to minimize the whole circuit (critical path) delay, we only need to minimize the delay of nodes in the critical paths. As a result, we may permit some nodes whose delays are greater than its label. For example, if node n_1 is more critical than node n_2, we will first process node n_1 and then force node n_2 not to interference with node n_1. In other words, we may increase the delay of n_2 by allowing n_1*'s* label not to be interfered.

5.3 Bsc Reduction under Delay Constraint

In this section, we propose a novel algorithm to reduce the number of bscs under some delay constraint. The basic idea is to add/remove some "timing redundant" bscs whose addition/ removal do not violate the previous timing and dependency constraint. The bsc addition step can be considered as a perturbation to help the optimization algorithm jump out of the local minimum. Then a modified bsc reduction algorithm [17] is applied to remove as many bscs as possible. Since the addition/removal of bscs does not increase delay, our delay result does not get worst than the previous result before bsc addition/removal, while the number of bscs can be reduced.

We say that a bsc is *timing redundant*, if the addition/removal of the bsc does not increase the circuit delay and violate the dependency constraint. For example consider Fig. 13(a) where there are *5* bscs $\{b_1, b_2, b_3, b_4, b_5\}$ to achieve the dependency constraint of *3*. Given $d(e) = d(bsc) = 1$, the circuit has the optimal delay of *5* and a critical path is shown in the graph. In this circuit, we say that bscs b_6 and b_7 in Fig. 13(b) are timing redundant because the addition does not violate the circuit timing and dependency constraint. However, the addition of timing redundant bscs may allow the subsequent bsc reduction algorithm to remove more bscs in the circuit. Consider the same example. After adding bscs b_6 and b_7, bscs $\{b_1, b_2, b_4, b_5\}$ become redundant and can be removed from the circuit in Fig. 13(c). Note that both circuits in Fig. 13(a) and Fig. 13(c) have the same delay but different number of bscs. By appropriately adding/removing timing redundant bscs, the number of bscs are reduced while the circuit timing is maintained.

During the bsc addition step, it should be noted that the addition of one timing redundant bsc may cause others to be non-timing-redundant. Our heuristic tries to order the selection in a way that bscs are inserted to those nodes which have larger fanouts or fanins. This is because that

the addition of those bscs has better chance to cause many bscs to become redundant and can be removed. Consider the same example in Fig. 13(c). The addition of bsc b_7 can cause bscs b_4 and b_5 to be removed.

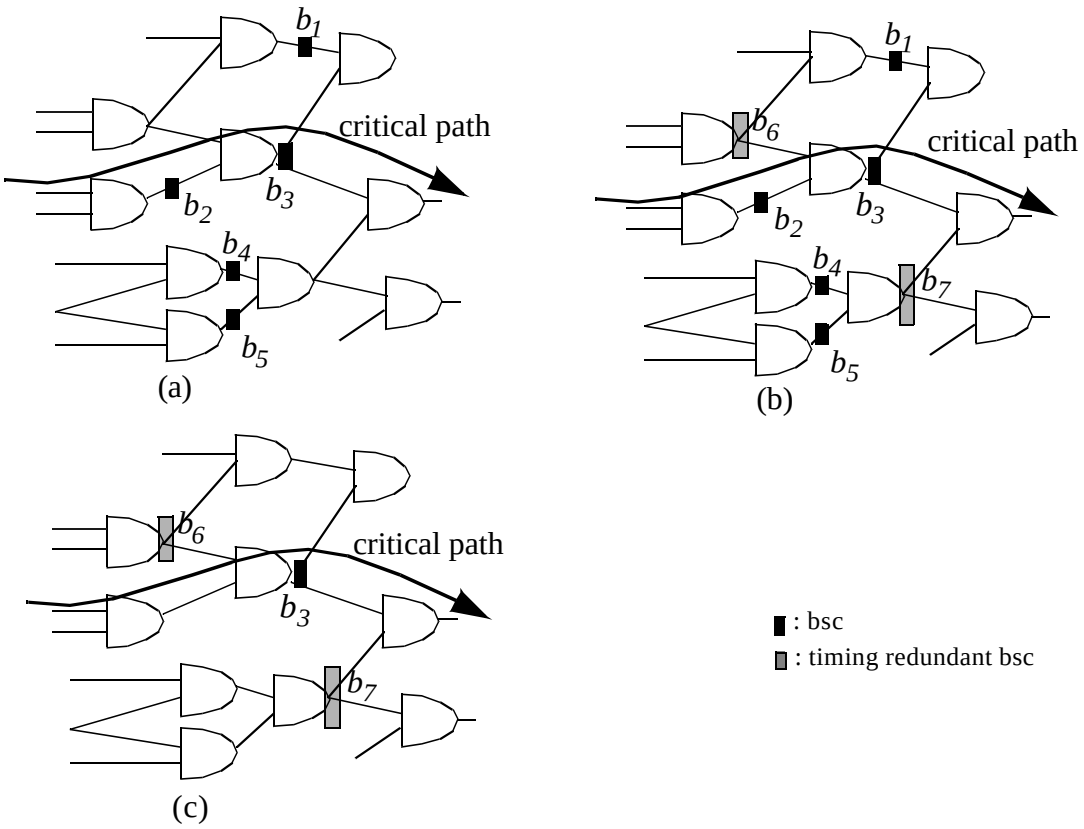

Fig. 13 : **Bsc addition and reduction.**

5.4 An Example to Summarize Our Bsc Insertion Algorithm

Fig. 14 shows our bsc insertion algorithm which integrates the heuristics to alleviate both the bsc chain effect and the interference effect. We use an example to summarize our algorithm.

```
bsc_insertion(network, k)
{
    compute label l(n) for each node n from PIs to POs;
    put all the POs into List_po;
    while (List_po is not empty) {
        remove the node n with the maximal label from List_po;
        weight the nodes of Cone(n) according to l(n);
        do{
            find a cutset Cut for n;
            if(the size of Cut is less than or equal to k){
                if (l(n) is maintaned) {
                    put all the nodes in Cut into List_po and the bsc set;
                }else{
                    modify some nodes' weights by the heuristics of Section 5.1and Section 5.2;
                }
            }else{
                l(n) = l(n) + 1;
                weight the nodes of Cone(n) according to l(n);
            }
        }until a feasible k-cutset for l(n) is found;
    }
    bsc_reduction();
}
```

Fig. 14 : The BSC insertion algorithm.

Let the dependency constraint be 3. Consider the circuit in Fig. 15(a) in which the label of a node is outside the corresponding node. Initially, the list $List_{po}$ consists of POs g_6 and g_8. Since the label of g_6 is larger than the one of g_8, we first try to insert bscs for g_6. After Applying the MFMC algorithm, we obtain the *3*-cutset $Cut_{g6} = \{g_1, g_3, g_4\}$ highlighted in Fig. 15(b). Because of the bsc chain effect in this 3-cutset, the delay of g_6 becomes *7* which exceeds its label, *6*. We then use the pushing-toward-input technique of Section 5.1 to obtain another *3*-cutset $Cut_{g6}' = \{g_1, g_2, g_3\}$ highlighted in Fig. 15(c). This new 3-cutset can cause the delay of g_6 to be the same as its label. Then, we remove g_6 and put nodes in Cut_{g6}' into $List_{po}$. After that, we process g_8 which has the maximal label currently. After applying the MFMC algorithm, we have one *3*-cutset $Cut_{g8} = \{g_4, g_7\}$ shown in Fig. 15(d). This 3-cutset interfere with the previous 3-cutset and cause the delay of g_6 to increase. By the re-weighting technique of Section 5.2, we can obtain another *3*-cutset $Cut_{g8}' = \{g_2, g_3, g_7\}$ which can satisfy our objective as shown in Fig. 15(e). Similarly, we put nodes in Cut_{g8}' to $List_{po}$. Now, since the dependency of each node of $List_{po}$ is less than or equal to the dependency constraint, *3*, the process

ends. And our algorithm reports that bscs should be inserted at the fanout stems of {g_1, g_3, g_7, g_2} shown in Fig. 14(f).

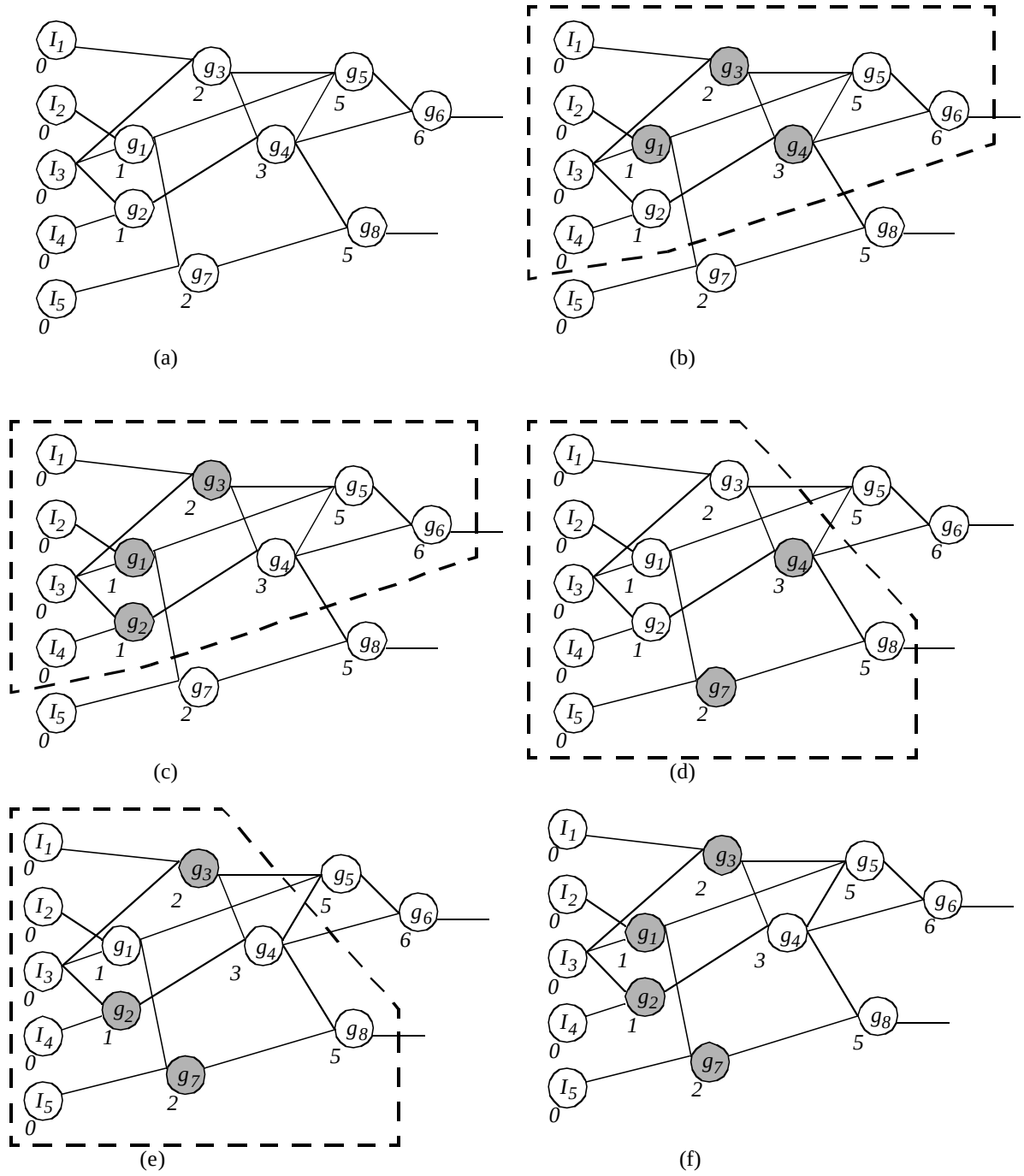

Fig. 15 : An example to illustrate the bsc insertion algorithm.

6 Experimental Results

We have implemented the algorithm in Fig. 14, and applied to the ISCAS85 combinational benchmark. In the experiment, we make two assumptions for the delays of bscs. One assumes

that the delay a bsc is *1*, $d(bsc) = 1$ (unit delay). The other assumes the delay of a bsc is *2*, $d(bsc) = 2$ (two delay), since a mux is a two-level combinational circuit. In addition, we also obtain technology mapping results using a commercial logic optimization tool with COMPASS 0.6-micron library. Our experimental results are shown in Table 1 to Table 4. Assuming $d(bsc) = 2$, Table 1 shows the results under the dependency constraint of *20* while Table 2 under the constraint of *15*. Assuming $d(bsc) = 1$, Table 3 shows the results under the constraint of *20* while Table 4 under the constraint of *15*.

Column one gives the name of each circuit. Column two shows the critical path delay without inserting any bsc. Column three shows the largest label among all POs. The largest label of a circuit is a lower bound of the critical path delay. After inserting bscs by the algorithm in Fig. 14, Column four shows the circuit delay and Column five shows the number of bscs inserted. Column six gives the CPU run time on Ultra Sparc II. After performing technology mapping with COMPASS 0.6-micro library, in the seventh and eighth columns, we show the delay and area results. (The area results do not include the *TPG* and *ORA*.) We also re-implement the algorithm [17] and Columns nine and ten show the results of delay and bscs needed while Columns eleventh and twelfth shows the results after technology mapping with the same library. For example, in Table 1, the critical path delay of C5315 is originally *53* without inserting bscs. Under the dependency constraint of *20* and the two delay assumption, the largest label is *55*. Our heuristic requires *39* bscs to achieve the delay of *55* which must be an optimal solution. After technology mapping, the results of our delay and area are *7.59* (ns) and 3997 while the results of [17] are *8.34*(ns) and 3840. We also highlight our results which are the same as the lower bounds, i.e. optimal solutions. For others, our results are optimal or very close to optimal solutions. On the average in Table 1, we obtain *9%* of delay improvement compared to the results of [17] with *4%* of area penalty.

7 Conclusions

In this paper, a timing-driven bsc insertion method for pseudo exhaustively testing VLSI circuits has been presented. We first presented an algorithm to estimate each node's label, *i.e.*, the lower bound of its delay. Since the bsc chain effect and the interference effect may occur during the bsc insertion process, the delay of some nodes may not achieve their labels by the formula. Then, we also explored the reasons which make the two effects occur. To alleviate these effects, we further proposed some heuristics on which an bsc insertion algorithm was developed based. Finally, the experimental results of ISCAS85 benchmark circuits show that our heuristic algorithm can achieve or be very close to the label (**optimal solution**).

Table 1: Dependency = *20* with *d*(*bsc*) = *2*.

circuit	initial delay	label	Ours					[17]			
			delay (unit)	#bsc	CPU time(sec)	technology mapping		delay (unit)	#bsc	technology mapping	
						delay (ns)	area			delay (ns)	area
C7552	*45*	*45*	*45*	*110*	*2752.7*	*6.76*	*5485*	*47*	*85*	*7.01*	*5345*
C6288	*124*	*130*	*132*	*66*	*2523.3*	*23.14*	*6381*	*142*	*55*	*25.19*	*6130*
C5315	*53*	*55*	*55*	*39*	*2127.0*	*7.59*	*3997*	*57*	*37*	*8.34*	*3840*
C3540	*56*	*60*	*60*	*61*	*955.9*	*9.07*	*2800*	*67*	*66*	*9.88*	*2786*
C2670	*41*	*45*	*45*	*21*	*266.2*	*4.14*	*2924*	*51*	*18*	*4.68*	*3040*
C1908	*45*	*47*	*47*	*19*	*227.8*	*7.25*	*1209*	*49*	*14*	*7.63*	*1103*
C1355	*27*	*29*	*29*	*8*	*84.6*	*5.63*	*1350*	*29*	*8*	*5.63*	*1350*
C880	*30*	*32*	*32*	*13*	*19.4*	*5.12*	*1287*	*42*	*13*	*6.68*	*1144*
C499	*15*	*17*	*17*	*8*	*18.1*	*4.91*	*1034*	*17*	*8*	*4.91*	*1034*
C432	*29*	*35*	*35*	*35*	*10.7*	*6.41*	*1207*	*42*	*19*	*7.52*	*850*
ratio						***0.915***	***1.044***			***1***	***1***

Table 2: Dependency = *15* with *d*(*bsc*) = *2*.

circuit	initial delay	label	Ours					[17]			
			delay (unit)	#bsc	CPU time(sec)	technology mapping		delay (unit)	#bsc	technology mapping	
						delay (ns)	area			delay (ns)	area
C7552	*45*	*45*	*45*	*124*	*2789.3*	*5.88*	*5485*	*47*	*118*	*6.95*	*5345*
C6288	*124*	*134*	*140*	*150*	*2521.7*	*24.12*	*6374*	*153*	*140*	*27.33*	*6130*
C5315	*53*	*55*	*55*	*63*	*2367.6*	*8.02*	*3997*	*57*	*58*	*8.58*	*3840*
C3540	*56*	*62*	*66*	*112*	*1105.5*	*9.59*	*2855*	*70*	*95*	*10.22*	*2786*
C2670	*41*	*45*	*47*	*65*	*304.2*	*5.19*	*3138*	*55*	*27*	*5.25*	*3040*
C1908	*45*	*47*	*48*	*27*	*176.9*	*7.17*	*1209*	*53*	*20*	*8.42*	*1103*
C1355	*27*	*29*	*29*	*8*	*84.4*	*5.63*	*1350*	*29*	*8*	*5.63*	*1350*
C880	*30*	*34*	*34*	*18*	*19.4*	*5.08*	*1239*	*38*	*15*	*5.68*	*1144*
C499	*15*	*17*	*17*	*8*	*18.1*	*4.91*	*1034*	*17*	*8*	*4.91*	*1034*
C432	*29*	*37*	*37*	*43*	*9.5*	*7.07*	*1061*	*44*	*30*	*7.64*	*850*
ratio						***0.846***	***1.046***			***1***	***1***

Table 3: Dependency = *20* with *d*(*bsc*) = *1*.

circuit	initial delay	label	Ours					[17]			
			delay (unit)	#bsc	CPU time(sec)	technology mapping		delay (unit)	#bsc	technology mapping	
						delay (ns)	area			delay (ns)	area
C7552	*43*	*43*	*43*	*110*	*2842.7*	*6.76*	*5714*	*44*	*85*	*7.01*	*5532*
C6288	*124*	*127*	*128*	*68*	*2569.4*	*23.45*	*6983*	*133*	*55*	*25.19*	*6881*
C5315	*49*	*50*	*50*	*39*	*2155.3*	*7.59*	*4177*	*51*	*37*	*8.34*	*4262*
C3540	*47*	*48*	*50*	*63*	*943.3*	*9.11*	*2999*	*51*	*66*	*9.88*	*2935*
C2670	*32*	*34*	*34*	*36*	*243.1*	*5.37*	*3638*	*38*	*18*	*4.68*	*2921*
C1908	*40*	*41*	*41*	*19*	*220.9*	*7.40*	*1191*	*42*	*14*	*7.63*	*1178*
C1355	*23*	*24*	*24*	*8*	*84.4*	*5.63*	*1350*	*24*	*8*	*5.63*	*1350*
C880	*23*	*24*	*24*	*13*	*24.5*	*5.30*	*1356*	*29*	*13*	*6.68*	*1202*
C499	*11*	*12*	*12*	*8*	*22.3*	*4.91*	*1034*	*12*	*8*	*4.91*	*1034*
C432	*17*	*19*	*20*	*35*	*10.5*	*7.06*	*1320*	*23*	*19*	*7.52*	*941*
ratio						***0.944***	***1.054***			***1***	***1***

Table 4: Dependency = *15* with *d*(*bsc*) = *1*.

circuit	initial delay	label	Ours					[17]			
			delay (unit)	#bsc	CPU time(sec)	technology mapping		delay (unit)	#bsc	technology mapping	
						delay (ns)	area			delay (ns)	area
C7552	*43*	*43*	*43*	*130*	*2802.3*	*5.94*	*5857*	*44*	*118*	*6.95*	*5532*
C6288	*124*	*128*	*130*	*156*	*2578.2*	*25.03*	*7000*	*138*	*140*	*27.33*	*6881*
C5315	*49*	*50*	*50*	*67*	*2378.7*	*8.27*	*4255*	*51*	*58*	*8.58*	*4262*
C3540	*47*	*49*	*50*	*123*	*986.4*	*9.98*	*3002*	*53*	*95*	*10.22*	*2935*
C2670	*32*	*34*	*35*	*60*	*278.0*	*5.54*	*3490*	*39*	*27*	*5.25*	*2921*
C1908	*40*	*41*	*41*	*28*	*169.6*	*7.43*	*1187*	*44*	*20*	*8.42*	*1178*
C1355	*23*	*24*	*24*	*8*	*83.6*	*5.63*	*1350*	*24*	*8*	*5.63*	*1350*
C880	*23*	*25*	*25*	*32*	*22.9*	*5.41*	*1324*	*27*	*15*	*5.68*	*1202*
C499	*11*	*12*	*12*	*8*	*22.2*	*4.91*	*1034*	*12*	*8*	*4.91*	*1034*
C432	*17*	*21*	*21*	*43*	*9.4*	*7.07*	*1320*	*24*	*30*	*7.64*	*941*
ratio						***0.855***	***1.056***			***1***	***1***

8 References

[1] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital System Testing and Testable Design*, Computer Science Press, New York, 1990.

[2] V. D. Agrawal and S. C. Seth, *Test Generation for VLSI Chips*, IEEE Computer Society, 1988.

[3] P. H. Bardell, W. H. McAnney and J. Savir, *Built-In Test for VLSI: Pseudo-random Techniques*, John Wiley and Sons, New York, 1987.

[4] Z. Barzilai, J. Savir, G. Markowsky and M. G. Smith, "The Weighted Syndrome Sums Approach to VLSI Testing", *IEEE Trans. on Computers*, vol. C-30, no. 12, pp.996-1000, Dec. 1981.

[5] S. N. Bhatt, F. R. K. Chung and A. L. Rosenburg, "Partitioning Circuits for Improved Testability", *Proc. of 4th MIT Conf. on Advanced Research in VLSI*, pp. 91-106, Apr. 1986.

[6] J. Cong and Y. Ding, FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs, *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 1, Jan. 1994.

[7] T. Cormen, C. Leiserson, and R. Rivest, *Algorithms*, Cambridge, MA: MIT press, 1990.

[8] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Inc., 1974.

[9] S. Devadas, A. Ghosh and K. Keutzer, *Logic Synthesis*, McGraw-Hill, Inc., 1994.

[10] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton, NJ: Princeton Univ. Press, 1962.

[11] W. B. Jone and C. A. Papachristou, "A Coordinate Circuit Partitioning and Test Generation Method for Pseudo-Exhaustive Testing of VLSI Circuits", *IEEE Trans. on CAD*, vol. 14, no. 3, pp. 374-384, Mar. 1995.

[12] K. T. Cheng, and C. J. Lin, "Timing-Driven Test Point Insertion for Full-Scan and Partial-Scan BIST", *Proc. of International Test Conf.*, pp. 506-514, Oct. 1995.

[13] E. J. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique", *IEEE Trans. on Computers*, vol. C-33, no. 6, pp. 541-546, Jun. 1984.

[14] E. J. McCluskey and S. Bozorgui-Nesbat, " Design for Autonomous Test", *IEEE Trans. on Computers*, vol. C-30, no. 11, pp. 866-875, Nov. 1981.

[15] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

[16] M. W. Roberts and P. K. Lala, "An Algorithm for the Partitioning of Logic Circuits", *IEE Proc.*, vol. 131, pt. E, no. 4, Jul. 1984.

[17] R. Srinivasan, S. K. Gupta and M. A. Breuer, "An Efficient Partitioning Strategy for Pseudo-Exhaustive Testing", *Proc. of Design Automation Conf.*, pp. 242-248, Jun. 1993.