Fast Boolean Optimization by Rewiring

Shih-Chieh Chang
National Chung-Cheng University,
Jay-Yi, Taiwan R.O.C.

Lukas P.P.P. van Ginneken
Synopsys Inc.
Mountain View, California

Malgorzata Marek-Sadowska
University of California at
Santa Barbara

This paper presents a very efficient Boolean logic optimization method. The boolean optimization is achieved by adding and removing redundant wires in a circuit. Our algorithm applies the reasoning of Automatic Test Pattern Generation (ATPG) which can detect redundancy efficiently. During the ATPG process, mandatory assignments are assignments which must be satisfied. Our algorithm analyzes different characteristics of mandatory assignments during the ATPG process. New theoretical results based on the analysis are presented which lead to significant performance improvements. The fast run time and the excellent scaling to large problems make our Boolean optimization method practical for industrial applications. Experiments show that the optimization results are comparable to those of [11] while the run time is two orders of magnitude faster (average 126x speed up). Furthermore, we report optimization results for several large examples, which were previously thought to be too large to be handled by Boolean optimization methods.

1 Introduction

Logic synthesis is a step that realizes a set of logic expressions using cells from a technology library. Usually, the objectives in logic synthesis are to optimize area, delay, power and testability. Among logic optimization algorithms, Automatic Test Pattern Generation (ATPG) based optimizations [2] [3] [4] [5] [11] [15] [16] [17] are becoming very popular because of the following advantages. First of all, ATPG based algorithms require little memory to process large circuits. Although the running time of a ATPG algorithm may be exponential, the memory requirement is linear in the size of the circuit. The amount of effort spent in running fault tests can be accurately controlled. ATPG based algorithms have good failure characteristics; when the algorithm aborts, this does not imply that the entire algorithm needs to be aborted, as is the case with most BDD based algorithms. In addition, ATPG optimizations can implicitly use circuit observability and controllability don't cares without the need to explicitly calculate them.

One of ATPG's strengths is the ability to detect efficiently redundant wires in a circuit. As a result, most ATPG based algorithms achieve optimization by adding and removing redundant connections in circuits. For example, [11] adds one redundant wire and then removes redundant wires caused by the previous change. Area optimization can be achieved by adding one and removing many wires in a circuit. Another ATPG based algorithm [3] targets some particular wire. The algorithm tries to remove a target wire by adding to the circuit another set of wires. Removing some critical wire can be very useful for many applications. For example, one may remove a wire in the critical paths to improve delay. [6] removes a wire to improve partitioning. In FPGA, [3] removes unroutable wires after routing. It adds routable wires to correct an unroutable FPGA circuit.

The motivation of this paper is to establish a theoretical background for the single wire addition and removal [3][4][5]. We investigate some necessary conditions for a wire to be redundant. Using these conditions, we are able to improve the results and speed up the run time of many ATPG based algorithms. Two essential issues are addressed in this paper:

1. Which wires can be removed after adding a new redundant wire?
2. Which new redundant wire, when added, will make the existing target wire redundant?

We study the characteristics of mandatory assignments [1], which must be satisfied for every test vector. We discuss two very important concepts, "forced" and "observability" mandatory assignments, which are used to distinguish among mandatory assignments. These two attributes of mandatory assignments can be computed along with the calculation of mandatory assignments. Very little computational overhead is required to determine these two additional attributes. Based on these, we derive theorems that improve the results of ATPG based optimizations.

2 An example

In this section, we illustrate some basic concepts while walking through an example from [11]. This paper first identifies that there is a redundant wire *e->m* (dotted in the FIGURE 1b) which can be added to the circuit. Adding this wire *e->m* causes two originally irredundant wires *e->g* and *a->f* (dotted) to become redundant. To detect the redundancy of *a->f* and *e->g*, [11] applies redundancy removal to the entire circuit. The drawbacks of this technique are two fold. First, running the redundancy removal on the entire circuit requires enormous CPU time for big circuits. In addition, there is no hint of which wire should be added since no information is available.

In this paper, our first set of theorems provide necessary conditions for wires to be redundant after adding another wire. For example, in FIGURE 1b, suppose we consider to add a redundant wire e->m. With proper analysis of mandatory assignments when detecting the redundancy of e->m, we can quickly determine that {h->z, d->z, g->h, f->h, c->g, b->e, b->f} cannot be redundant. This leaves only {a->f, a->e, e->g} as possible candidates for redundancy checking if e->m is added. Finally, we need only to perform the redundancy test on these three wires. The redundancy information about these wires not only can reduce the run time but also can direct the optimization to choose a good redundant candidate for adding.

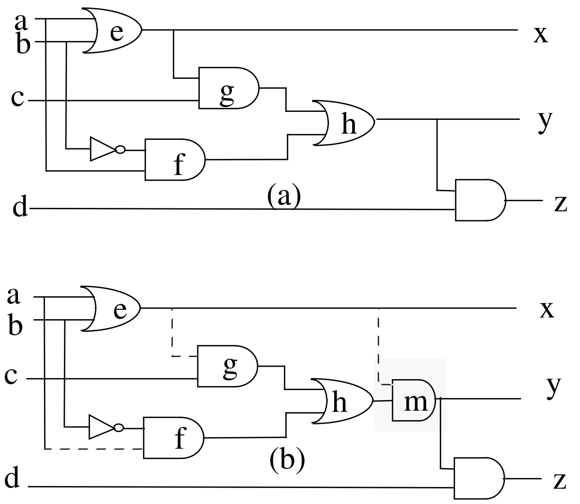

FIGURE 1 Redundant wires caused by adding anothe

Another contribution of this paper is to speed up and improve the results of finding "alternative wires" [3][4]. A single alternative wire of a target wire is a wire whose addition can make the target wire redundant. For example, in the FIGURE 2, after adding the wire g_5-> g_9 and removing the wire g_1->g_4, the circuit's function remains the same. g_5->g_9 is a single-alternative wire for g_1->g_4.

In this paper, we will develop methods that allow us by performing the stuck-at fault test of g_1->g_4 to conclude that any among wires, c->g_9, c->g_8, c->g_9, g_2->g_4, g_2->g_8, g_2->g_9, g_7->g_4, g_7->g_9, f->g_4 and f->g_8, cannot possibly be an alternative for g_1->g_4. Therefore, we can skip the redundancy test for those wires. This will substantially reduce the cpu run time in comparison to the algorithms in [4].

This paper is organized as follows: Section 3 reviews some concepts in testing that we will be using in this paper. Section 4 introduces two attributes of mandatory assignment,

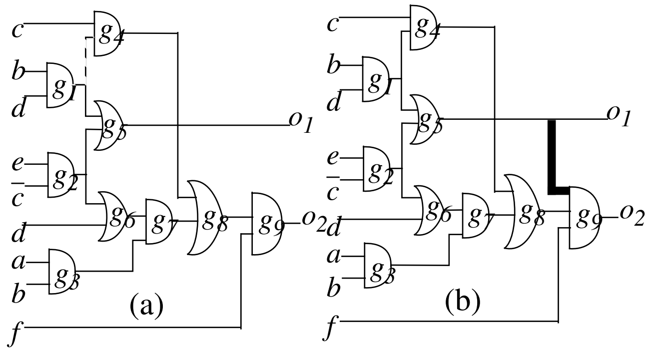

FIGURE 2 An example for single alternative wire.

namely "forced" MAs and "observability" MAs. Section 5 derives some necessary conditions for redundancy of a wire. Section 6 gives necessary and sufficient conditions for a wire to be an alternative wire. Section 7 describes an efficient implementation of the algorithm of [4], using the theorems of the previous two sections. Finally results and conclusions are presented.

3 Background and Definitions

In the following, we review some standard logic synthesis terminology and ATPG related concepts which will be used throughout the paper. Here we only consider circuits consisting of AND, OR and INV gates. Complex gates can be handled by decomposing them into AND, OR and INV gates.

A Boolean network is a directed acyclic graph where each node is associated with a Boolean function f_i, and a Boolean variable y_i. There is a *wire* directed from a node n_i to a node n_j if the function f_j depends on the variable y_i. The *dominators* [10] of a wire *W* is a set of gates *G* such that all paths from *W* to any primary output have to pass through all gates in *G*. The value of an input to a gate is said to be *controlling* if it determines the value of the gate's output regardless of the values of the other inputs; the controlling value is 1 for an OR or a NOR gate, and 0 for an AND or a NAND gate. The inverse of the controlling value is called the *non-controlling* value or *sensitizing* value.

Consider the dominators of a wire *W*. The *side inputs* of a dominator are its inputs not in the transitive fanout of the wire *W*. To generate a test for a stuck-at fault at wire *W*, all side inputs of the wire *W*'s dominators must be assigned their sensitizing values. For a wire stuck-at-1 {0} fault test, a test vector must generate a 0 {1} at the source node of the wire. We refer the 0 {1} at the source node of the wire as an *activating* value for the test.

Let w_r be a wire being tested for a stuck-at 0 {1} fault; a *faulty* circuit is the circuit in which w_r is replaced by a constant 0 {1}. An input combination *v* is a *test vector* if an output of the good circuit and faulty circuit are different when applying *v*. If no such a test vector exists, then the wire under *stuck-at fault* test is redundant.

The *mandatory assignments* (MA) are the value assignments to nodes required for a test to exist and must be satisfied by any test vector. The process of computing these mandatory assignments and checking their consistency is referred to as *implication* [1].

The process of implication is as follows. The MAs on the side inputs of a dominator are set to sensitizing values and the MA on the source node of the target wire is set to the activating value. These MAs can then be propagated by using some simple rules such as if the output of AND {OR} gate is 1 {0}, the inputs are 1 {0}. If all the inputs of an AND {OR} gate are 1 {0}, the output is 1 {0} etc. [1]. This process is called *direct implication*. More MAs can be found by more complicated approaches such as recursive learning [12].

If the mandatory assignments of a stuck-at fault test cannot be consistent, the fault is untestable and therefore, the wire is redundant. A wire to be removed is referred to as the *target wire*. The corresponding stuck-at fault is called the *target fault*.

4 Forced mandatory assignments and observability mandatory assignments

In this section, we discuss two very important concepts, "observability" mandatory assignments and "forced" mandatory assignments. These two concepts are used to form the backbone of our theorems. As mentioned in section 3, a MA of a stuck-at fault can be derived from MAs that activate the fault or sensitize a fault propagating path to one primary output.

Definition 1: During a stuck-at fault test for $w_r = n_s$->n_d, we define a MA to be an *observability* MA if the MA must be set to sensitize a fault propagating path to one primary output.

The observability MAs are mandatory assignments which are necessary to make the fault observable at a primary output. The observability MA is derived from MAs that sensitize the dominators but excluding the effect of the activating MA. The observability MAs are a subset of all the MAs. Note that since the activating value does not play a role, the observability MAs do not depend on the source node n_s of w_r

For example, in the Fig 2(a), the observability MAs for g_1->g_4 are {c=1, g_2=0, g_7=0, f=1}. Note that MAs {g_1=0, g_5=0} are not observability MAs because they need to be derived from the activating MA.

Now we define the "forced" MA. Let *n* be a node in the circuit *C*. Suppose after a stuck-at fault test, *n* has a MA. In the case of five-valued logic [1], the MA can be 0, 1, D or $\overline{\text{D}}$. We build the new circuit *C'(n)* as follows. If the MA is 1 or D, disconnect *n* from its fanouts and connect those fanouts to a constant 0. If the MA is 0 or $\overline{\text{D}}$, we connect those fanouts to a constant 1. See FIGURE 3. (The reason why D and $\overline{\text{D}}$ are inserting different value can be found in [3].)

Definition 2: Suppose a node *n* has a MA after performing a stuck-at fault test in C. We say that *n* has a *forced* MA in C if when we perform the same stuck-at fault test in *C'(n)*, the fault becomes untestable.

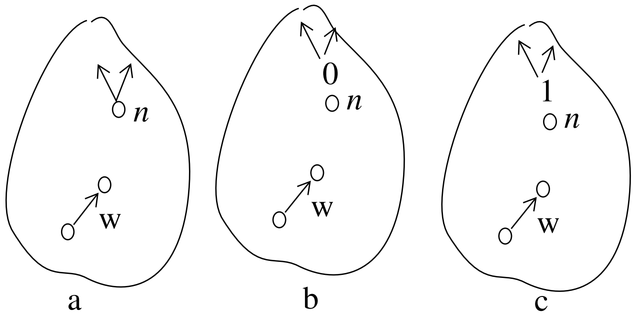

FIGURE 3 (a). *w* is the wire on which the stuck-at fault test is performed on. (b) shows C'(n) if *n* has an MA=1 or d. (c) shows C'(n) if the node has an MA=0 or $\overline{\text{d}}$.

Forced MAs can be seen as MAs which are required for the fault to be testable. Modifying the circuit structure to change a forced MA will cause a conflict and will make the fault untestable. Non-forced MAs are due to an incidental consequence of the test, but changing the circuit structure to change a non-forced MA will not make the original fault untestable.

For example in FIGURE 2(a), consider the g_1->g_4 stuck-at-1 test. We have MA={g_1=0, c=1, g_2=0, g_5=0, g_7=0, f=1}. c=1 is a forced MA because disconnecting c with g_4 and inserting a 0 at an input of g_4 will make the stuck-at fault untestable. The MA g_5=0 is non-forced. It is so because after disconnecting g_5 and o_1 and connecting o_1 to 1, g_1->g_4 is still testable.

According to the definition of forced MA, the MAs on the dominators are all forced. This is because putting a constant value at a dominator makes the target fault untestable. This definition suggests a precise way of finding forced mandatory assignments in a stuck-at fault test. However, in reality, computing whether a MA is forced applying the definition directly is very time-consuming. In the following, we discuss how these forced MAs can be calculated in practice.

During the process of direct implication, if the output of an AND {OR} gate is 1 {0}, all the inputs must be 1 {0}. We refer to this process as backward implication.

Lemma 1: The MAs obtained by setting side inputs of dominators to non-controlling values and the activating MA on the source node of the target fault are forced. In addition, the MAs obtained by backward propagation are also forced.

Proof. This theorem follows directly from the definition of a forced MA. QED

Lemma 1 suggests that whether a MA is forced can be determined while performing direct implication. Therefore, no additional test is required to decide whether a MA is forced or not. For example, in FIGURE 2(a), let us consider g_1->g_4 stuck-at-1. We have {c=1, g_7=0, f=1} as forced MAs because they are the side inputs of dominators, and {g_2=0, g_5=0} as not forced MAs because they are obtained from forward propagating other MAs.

MAs can be also derived from recursive learning which applies direct implication recursively. If a MA is obtained from recursive learning, one can also use the notion of backward implication to decide whether it is a forced MA. In this paper, we do not discuss finding forced MAs in recursive learning.

Intuitively, a forced MA is the MA that must be maintained for the fault to be testable. If a forced MA is changed, the fault becomes untestable. For example in FIGURE 2(b), g_9 (a dominator) has a forced MA in the g_1->g_4 s-a-1 test. If a redundant wire g_5->g_9 is added to the circuit, the MA of g_9 is changed to 0. As a result, g_1->g_4 is redundant after adding g_5->g_9.

5 Wires cannot possibly be redundant after adding one redundant wire

A redundant wire is a wire that we can add/remove from the circuit without changing the circuit's behavior. Adding a redundant wire to a circuit may result in redundancy of other wires. In the following, we explain how to use the concepts of forced MA and observability MA to identify whether a particular wire can be redundant after adding another redundant connection.

Two assumptions are made in this paper: First, the circuit under consideration is irredundant, that is, no wire in the circuit is redundant. This assumption of irredundancy is very important in our proofs. We will be using these theorems as filters to screen out wires that are not possible to be redundant. As a result, when the circuit in consideration contains some redundant wires, applying these theorems may fail to screen out some wires, causing unnecessary work to be performed. The second assumption is that we only consider adding a redundant wire and checking whether another wire can be removed individually. We do not consider a possibility of finding "two simultaneously redundant wires [4]." Two wires are simultaneously redundant if we can add one and remove the other simultaneously but we cannot add or remove either of them individually.

Without losing generality, let us consider adding a redundant wire $w_r = n_s$-> n_d to an AND gate in an irredundant circuit C in FIGURE 4(a). Since w_r is redundant, computing the MA of the w_r s-a-1 test is inconsistent. Because of this inconsistency, the MAs in the circuit are meaningless. The following theorems and lemmas show which wires cannot become redundant after adding a redundant wire.

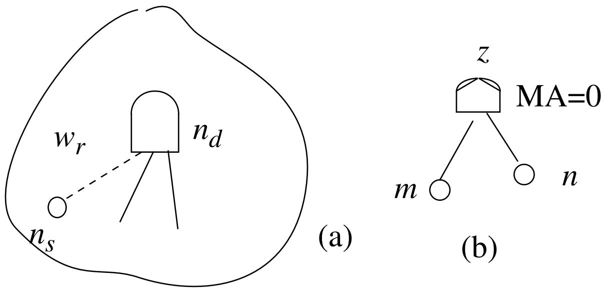

FIGURE 4 Adding a redundant wire w_r in an irredundant circuit.

Lemma 2: The *observability* MAs for the redudant wire w_r=n_s->n_d must be consistent in the circuit C.

Proof. Let n_d be an AND {OR} gate. Suppose the observability MAs are inconsistent. Any new connection which does not exist in C that fanins to n_d is a redundant wire. Therefore, a constant 0 {1} that fanins to n_d is also redundant. We can then conclude that n_d can be replaced by constant 0 {1} which contradicts our irredundancy assumption. QED.

Lemma 3: Let n_d be an AND {OR} gate. The wire w_r=n_s-> n_d is redundant, if and only if n_s has an **observability** MA=1 {0} for w_r stuck-at fault test. (A similar theorem is shown in [11].)

Proof. Let us compute observability MAs for w_r first. Based on Lemma 2, these observability MAs are consistent. Then, assign 0 {1}, the activating value, at the n_s. Since w_r is redundant, the MAs are inconsistent. Therefore, when computing observability MAs, n_s must have had an observability MA=1 {0}. QED

Note that observability MAs don't depend on the source of the target fault. In FIGURE 1, the observability MAs for (*any node)->m* stuck-at-1 test are {*e=1, h=1*}. Since e=1, e->m is a redundant wire. For another example, in FIGURE 2, suppose we know that g_5->g_9 is a redundant wire. When computing observability MAs for *(any node)->*g_9, since assigning g_5=0 will cause conflict (g_5->g_9 is redundant), g_5 must have a 1.

Let (C ∪ w_r) denote a circuit C with an added wire w_r and (C\w_r) be a circuit C from which the wire w_r has been removed. Suppose an irredundant wire w_t in C becomes redundant after adding a redundant wire $w_r = n_s$-> n_d.

Lemma 4: w_r becomes irredundant after removing w_t in (C∪w_r\w_t).

Proof. If w_r is still redundant after removing w_t, w_t is redundant without adding w_r. This conflicts with our original assumption that C is irredundant. QED.

Lemma 5: For the w_r stuck-at fault test, the observability MA at n_s in C is different from $(C \cup w_r \backslash w_t)$.

Proof. If the n_s has the same observability MA, w_r is still redundant after removing w_t. This contradicts Lemma 4 QED.

For example, in FIGURE 1b, observability MAs for e->m are {*e=1, h=1*}. Suppose we remove e->g. The observability MA is {*h*=1}. The observability MA at *e* is different. Therefore, after removing e->g, the wire e->m is no longer redundant. As we mentioned in the definition, we consider the computation of MAs from either direct implication or recursive learning.

Theorem 6: Suppose observability MAs are computed for a redundant wire w_r. A wire in $(C \cup w_r)$ is not redundant if the wire is not visited during the process of computation of observability MAs for w_r,

Proof. Removing those wires will not change the observability MA at n_s since without those wires we can still have the same observability MA at n_s. QED

Example 1: in FIGURE 1(b), let us consider to add a redundant wire *e->m* (*m* is a highlighted AND gate). When computing observability MAs for *e->m*, we first set *h*=1 to propagate a fault D from *m* to any output. Since *h*=1, either node *g*=1 or node *f*=1. If *g*=1, {*c*=1, *e*=1}. If *f*=1, {*a*=1, *b*=0, *e*=1}. As a result, we conclude that *e*=1 and *e->m* is a redundant wire. During the computation, the wires *d->z* and *m->z* are never visited. Therefore, we also conclude that adding any redundant wire to node *m*, the wires *d->z*, and *m->z* will not be redundant.

In this example, we need to apply recursive learning to obtain e=1 from h=1. Since h=1, we try g=1 or f=1. All the wires {g->h, f->h, a->f, b->f, c->g, e->g, b->e, a->e} that are traversed from either of the choices {g=1, f=1}. Our theorems in the followings are applied for each choice separately and the results are accumulated to obtain information on all possible redundant wires. For example, when g=1, we traverse {g->h, c->g, e->g}. when f=1, we traverse {f->h, a->f, a->e, b->f, b->e}

Theorem 7: All the direct input wires of an AND {OR} gate are not redundant in $(C \cup w_r)$ if the AND {OR} gate has an **observability and forced** MA=0 {1} for w_r in C.

Proof. Without loss of generality, let us consider a two-input AND gate which has a forced MA=0. See Figure 4b. There are three possibility of MAs at the inputs of an AND gate. Those are (m, n) = (1, 0), (0, 0), (X, X). We discuss them separately and prove that removing either one of the input wires on the AND gate will not change the observability MA at n_s.

Case 1: (m, n) = (0, 0). Removing either m->z or n->z will not change any MAs in the circuit so m->z and n->z are not redundant after adding w_r.

Case 2: (m, n) = (1, 0). m->z is not redundant because of the same as case 1. Consider computing the observability MAs for w_r after removing n->z. There will be a conflict since *z* has a forced MA 0 and *m* has a value of 1. Therefore, after removing n->z, w_r is still redundant. This violates our original irredundancy assumption.

Case 3: (m, n) = (x, x). The MA z=0 implies (m, n) = (0, x) or (x, 0). Assuming (m, n) = (0, x) or (x, 0) will not change other MAs in the circuits. Therefore, removing m->z or n->z will not change observability MAs. The wires m->z and n->z are not redundant. QED.

Let us return to Example 1. Since the *h* has observability forced MA=1, the wires g->h and f->h are not redundant. The possible redundant wires are {a->f, b->f, c->g, e->g, b->e, a->e}

Theorem 8: A wire $w=(n_x, n_z)$ is not redundant in $(C \cup w_r)$ if n_z is AND {OR} gate and n_x has an observability MA of 1 {0} for w_r.

Proof. The proof is similar to the above theorem because observability MAs will not change after removing the wire. QED.

Again, let us look at Example 1. When considering f=1, we have a=1 and b=0. According to the above theorem, we know that b->e cannot be redundant. The possible redundant wires are {a->f, b->f, c->g, e->g, a->e}.

Theorem 9: Let node n_z be an AND {OR} gate and one of its input wires is (n_x, n_z). If n_x has an observability MA of 0 {1} for w_r, all other input wires of n_z are not redundant in $(C \cup w_r)$.

Proof. It is the same as the above. QED

Theorem 10: During recursive learning, if a wire does not contribute to any MA, the wire cannot be redundant.

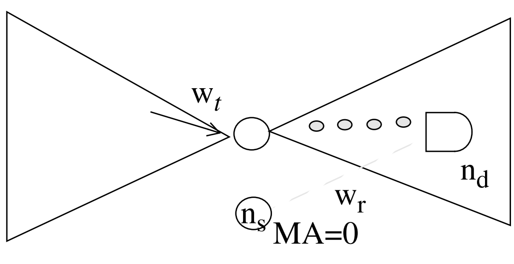

FIGURE 5 Find single alternative wire for w_t

In Example 1, since b->f and c->g do not cause that e=1. Therefore they cannot be redundant. As a result, the possible redundant wires are {a->f, e->g, a->e}.

In summary, in FIGURE 1b, consider to add the redundant wire e->m: 1. From theorem 6, the wires {d->z, m->z} cannot be redundant. 2. From theorem 7, the wires {g->h, f->h} cannot be redundant. 3. From theorem 8, the wires {b->e} cannot be redundant. 4. From theorem 10, the wires {b->f, c->g} cannot be redundant.

6 Single alternative wire

Single alternative wire is a concept proposed in [3]. A single *alternative* of a target wire is a redundant wire whose addition can make the target wire redundant. For example, in FIGURE 2, if we add the wire g_5->g_9, then, the wire g_1->g_4 becomes redundant. In this case, we say the g_5->g_9 is a single alternative wire for g_1->g_4. In this section, we first review the procedure that finds single-alternative wires for a wire. Then, we show the necessary and sufficient condition for a redundant wire to be a single-alternative wire for the target wire. In addition, we also propose an improved version of the procedure. This new procedure improves the quality as well as the run time of finding single-alternative wires.

Let us consider removing a target wire w_t by adding a redundant wire w_r to the circuit. The way of removing the target wire w_t is to make w_t stuck-at fault test to become untestable [3]. For example, in Fig. 5, suppose we would like to remove w_t and w_r=n_s-> n_d is not present in the circuit now. Let us consider w_t stuck-at fault test. n_s is a node that has a MA=0, and n_d is a dominator of w_t. If the wire w_r is present in the circuit, the value of 0 will force MA=0 on n_d. The MA=0 on n_d blocks the fault propagation and makes the w_t stuck-at fault untestable and redundant. Therefore if w_r is present in the circuit, w_t is redundant. However, adding w_r may change circuit's function so we need to make sure that w_r is a redundant wire. In summary, the procedure of finding alternative wires for w_t is as follows. First, we compute the MAs for the w_t stuck-at fault test. Secondly, collect a set of candidate connections that can block the fault propagation. Finally, check if a candidate connection is redundant.

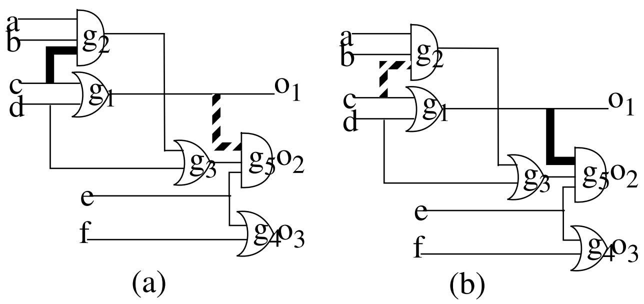

FIGURE 6 Another example of single-alternative wire. c->g_2 is a single alternative wire for g_1->g_5.

For example, in Figure 6, to remove c->g_2, we first compute the stuck-at-1 fault for c->g_2. We have {c=0, b=1, a=1, d=0, g1=0, e=1, g_4=1}. Then, we find that g_1->g_5 is a candidate connection to block the fault propagation. Finally, we check the redundancy of g_1->g_5. Since it is redundant, we say g_1->g_5 is an alternative wire for c->g_2.

The above procedure shows a way to find an alternative wire for the target wire w_t. There is no information about whether we are finding all possible alternative wires for a wire or not. Note that our goal is to make the target wire untestable. Blocking the fault propagation is not the only way of achieving this. Here, we show a necessary and sufficient condition (Theorem 13) for a redudant wire to be an alternative wire for the target wire w_t.

Theorem 11: If w_r=n_s-> n_d is an alternative wire for w_t, n_s must have a mandatory assignment 0 {1} for the stuck-at fault test of w_t and n_d is an AND {OR} gate.

Proof. Suppose n_s does not have a MA. Some test vectors for w_t stuck-at fault will generate a 0 and other test vectors generate a 1 at the n_s. Suppose n_d is an AND {OR} gate. After adding w_t, a test vector that generates 1 {0} at the n_s is still a test vector for w_t after adding w_r. As a result if n_s does not have an MA, then after adding w_r, w_t is still not redundant. Therefore, w_r is not an alternative wire for w_t. Using the same argument, we can also show that n_d must be an OR gate when n_s=1 and AND gate when n_s=0. QED

Theorem 12: If w_r= n_s-> n_d is an alternative wire for w_t, an AND {OR} gate n_d must have a forced mandatory assignment 1 or D {0 or $\overline{D}$} for the stuck-at fault test of w_t in C.

Proof. According to Theorem 11, n_s must have a MA=0 {1} when n_d is an AND {OR} gate. Since w_t is redundant in (C $\cup$ w_r), the MAs of w_t stuck-at fault is inconsistent. Therefore, n_d must have a MA. According to the definition of forced MA, this MA must be forced. QED.

Theorem 13: A redundant wire w_r=n_s-> n_d is an alternative wire for w_t, if and only if an AND {OR} gate n_d has a forced MA=1 or D {0 or $\overline{D}$} and n_s has a MA=0 {1} for the stuck-at fault test of w_t.

Proof. The if part can be proved by the definition of a forced MA. The only if part follows directly from the previous two theorems. QED

For example in Fig.6. since g_1->g_5 is an alternative wire for c->g_2. c->g_2 must also be an alternative wire for g_1->g_5. We now consider the circuit in Fig.6(b) and try to remove g_1->g_5 by adding c->g_2. After computing g_1->g_5 s-a-1 test, we have MA={g_1=0, c=0, d=0, g_3=1, e=1, o3=1, g_2=1, a=1, b=1}. Since g_2=1 is a forced MA, we can try to add a wire to violate the MA. Then, in the second step, we find that c->g_2 is a candidate connection to remove g_1->g_5. Since c->g_2 is redundant, c->g_2 is an alternative wire for g_1->g_5.

The above procedure improves the quality of the original finding single alternative wires procedure [3]. In the following, we show a way to improve the efficiency. Note that

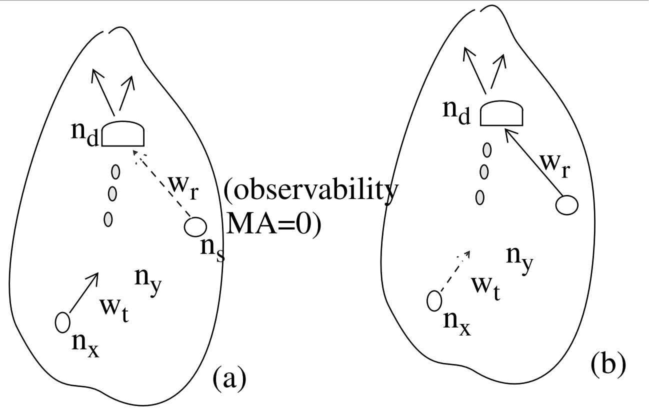

FIGURE 7 Observability MA is not useful to find single alternative wires.

in [3] there are several theorems related to efficiency improvement. The discussion here is quite different from those in [3]. According to Theorem 8, a source node n_s of w_r=n_s -> n_d must have a MA. In the following, we show the observability MAs are not useful. Note that observability MAs are subset of the MAs.

Theorem 14: If n_s has an observability MA for w_t stuck-at fault test, w_r=n_s -> n_d **cannot** be an alternative wire for w_t =n_x -> n_y.

Proof. Prove by contradiction. Without losing generality, let us assume that n_d is an AND gate in FIGURE 7a. Suppose w_r is an alternative wire for w_t and n_s has an observability MA. Since observability MAs are independent to the activating value, adding w_r, we can replace w_t with a constant 1 or 0 because of s-a-1 and s-a-0 are untestable.

Let us consider adding w_r and replace w_t with a constant 1. According to Lemma 4, w_r becomes an irredundant wire after replacing w_t with a constant 1 in ($C \cup w_r \backslash w_t$) in FIGURE 7. In addition, in ($C \cup w_r \backslash w_t$), w_t is an alternative wire for w_r. Let us consider adding w_t to remove w_r. According to Theorem 11, n_x has a MA=0 and n_y has a forced MA 1 or D. Now if we insert a "0" to the input of n_d, the stuck-at fault test for w_r will become redudant because n_d has a forced MA 1 or D. On the other hand, since w_r is an alternative wire for replacing w_t by a constant 0, w_r becomes irredundant after the replacement. This conflicts with the previous argument. Therefore, n_s should not have an observability MA. QED.

For example in FIGURE 6, to find single alternative for c->g_2, the MAs for c->g_2 stuck-at-1 test are {a=1, b=1, d=0, g_3=1, e=1, g_4=1, c=0, g_1=0}. Among the MAs, {a=1, b=1, d=0, g_3=1, e=1, g_4=1} are the observability MAs. Therefore, to find single alternative wire that fanins to g_5, we only need to consider the candidate connection of {c->g_5, g_1->g_5}.

7 Implementation

The implementation of our algorithm is based on the algorithm derived in [4]. This algorithm does Boolean optimization based on alternative wires. We used our theorems to further speed up this algorithm. See FIGURE 8 for pseudo-code of our algorithm.

Our algorithm traverses all nodes of the network. Each node is considered as a destination node n_d of a newly added redundant wire w_r. Our goal is to find a source node n_s such that the wire w_r = n_s->n_d can remove a set of wires w_t of maximum cost. See [4] for details on this optimization. The goal of this paper is to make the search for n_s more efficient.

```
foreach node n_d in the circuit {
    find the observability MA of n_d;
    /* the observability MA don't depend on the source*/
    for each wire w_t in the fanin and fanout cone of n_i {
        Use Theorem 7, 8, 9, and 10 and to skip wires w_t which
            cannot possibly become redundant;
        Calculate the MA of w_t
        Use Theorem 12 to check if a wire can be added to n_d
            to make w_t redundant
        Fill source_array with nodes which have an MA
            using Theorem 11
        Use Theorem 14 to prune source_array
        optimize the circuit as in [4]
```

FIGURE 8 The algorithm

The algorithm starts with calculating the observability MAs for n_d. Note that this can be done, even if n_s is not yet known. From these observability MAs we can determine that some wires cannot possibly be redudant, using Theorem 7, Theorem 8, Theorem 9, and Theorem 10. Subsequently we calculate the MAs for all w_t which pass this test. Now, using Theorem 12 we check if n_d is suitable to make w_t redundant.

To find all possible source nodes n_s for the alternative wire n_s->n_d to replace w_t, here is a simple procedure. After computing the w_t stuck-at fault MA's, we put a node which has an MA into source_array. We can then use Theorem 14 to check whether a candidate wire is an alternative wire by checking whether it is redundant, and whether the MA's are conflicting. Since the proofs does not make an assumption that all implications should be found, all theorems hold even partial implications are used. The actual implementation only uses those implications which are easy to compute.

8 Experimental Results

Table 1 compares the optimization of SIS1.1, HANNIBAL and ours (termed rewire) for some benchmark circuits. Since our circuits are in the form of AND and OR gate, we post-process those circuits with "el; sweep; el; simplify" (which are commands in SIS) and compare the factored literal count with SIS and HANNIBAL. Column 2 shows the run time for Hannibal and column 3 shows the run time for our implementation. Column 4, 5 and 6 shows the results of SIS, HANNIBAL and ours in terms of literal count. Among those circuits in the table, our algorithm is 126 times faster than the algorithm of HANNIBAL. The literal counts are about the same between ours and Hannibal's. The first section of the table lists and compares the circuits reported in [11]. The remainder of the table lists some additional circuits.

9 Conclusions

As demonstrated in the table, our implementation is much faster then the optimization algorithm of [11], with competitive results. Further speed improvements are possible, because our implementation did not use Theorem 6. From the results on some very large circuits, such as s38417, we can see that our approach scales well. The above theorems are applicable to other ATPG based optimizations as well. It seems likely that the algorithms of [2] [3] [5] [11] [15] [17] can take advantage of our theorems to speed up run time.

10 References

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

[2] C. L. Berman and L. H. Trevillyan. "Global Flow Optimization in Automatic Logic Design," IEEE Trans. CAD 10, pp. 557-564, May 1991.

[3] S. C. Chang, K. T. Cheng, N.S. Woo and M. Marek-Sadowska "Layout Driven Logic Synthesis for FPGA," *Proc. Design Automation Conf.* pp. 308-313, June 1994

[4] S. C. Chang, M. Marek-Sadowska "Perturb and Simplify, Multi-level logic optimizer", *Digest Int. Conf. on Computer Aided Design*, pp.2-5, Nov. 1994.

[5] K. T. Cheng and L. A. Entrena, "Multi-Level Logic Optimization by Redundancy Addition and Removal," in *Proc. European Conference On Design Automation*, pp. 373-377, Feb. 1993.

[6] D. I. Cheng, C. C. Lin and M. Marek-Sadowska, "Circuit Partitioning with Logic Perturbation," in *Proc. Int. Conference on Computer Aided Design,* pp., 650-655, Nov. 1995.

[7] M. Damiani, J. C. Y. Yang and G. De Micheli, "Optimization of Combinational Logic Circuits Based on Compatible Gates", *Proc. Design Automation Conf.*, pp. 631-636, June 1993.

[8] L. A. Entrena and K. T. Cheng, "Sequential Logic Optimization By Redundancy Addition and Removal", *Proc. Int. Conf. on Computer Aided Design,* Nov. 1993.

[9] M. Higashida, J. Ishikawa, M. Hiramine, K. Nomura, "Multi-level Logic Optimization Based on Pseudo Maximum Sets of Permissible Functions," *European Design Automation Conf.***,** pp. 386-391, 1993.

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

[11] W. Kunz and D.K. Pradhan, "Multi-Level Logic Optimization by Implication Analysis", *Digest Int. Conf. on Computer Aided Design,* pp. 6-13, Nov.1994.

[12] 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.*, pp.816-825, Oct. 1992.

[13] S. Muroga et al, "The Transduction Method-Design of Logic Networks Based on Permissible Functions," IEEE Transaction. on Computer C38(10). pp. 1404-1423 Oct. 1989.

[14] M. Schulz and E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," *Proc. Fault Tolerant Computing Symp.*, pp. 30-34, June 1988.

[15] B. Rohfleisch, B. Wurth, K. Antreich "Logic Clause Analysis for Delay Optimization", Proc. DAC, 1995, pp. 668-672.

[16] M. R. C. M. Berkelaar, L. P. P. P. van Ginneken: "Efficient Orthonormality Testing for Synthesis with Pass-Transistor Selectors", *Digest Int. Conf. on Computer Aided Design*, pp. 256-263, Nov. 1995.

[17] M. Yuguchi, Y. Nakamura, K. Wakabayashi, T. Fujita "Multi-Level Minimization based on Multi-Signal Implications", proc. DAC, 1995, pp. 658-662.

TABLE 1

Circuits	**Hannibal cpu (sec)**	**Rewire cpu (sec)**	**SIS boolean (lits)**	**Hannibal (lits)**	**Rewire (lits)**
C3540	6815	85.6	1299	1154	1127
C432	95	2.0	240	161	171
C2670	1782	31.7	759	718	697
C880	269	3.2	427	417	415
C5315	15611	65.9	1815	1697	1687
C1355	555	17.5	554	544	552
C6288	13704	89.0	3550	3240	3251
C1908	935	16.0	552	517	512
C499	543	8.4	554	544	550
subtotal	**40309**	**319.3**	**9750**	**8113**	**8107**
relative	**126.24**	**1**	**1.19**	**1.001**	**1**
s13207		541.7	2957		2719
s38417		1746.6	12301		10434
s5378		77.8	1490		1351
s9234		126.1	1944		1724
alu2		50.3	446		324
alu4		152.1	800		623
term1		5.0	237		145
too_large		31.9	437		301
ttt2		5.2	223		179
z4ml		0.5	48		36
f51m		4.8	135		105
frg2		85.1	933		761
total		**3146.4**	**31701**		**27704**
relative			**1.14**		**1**