On Removing Multiple Redundancies in Combinational Circuits

Shih-Chieh Chang
National Chung Cheng Univ.
Chia-Yi, Taiwan, R.O.C.

David Ihsin Cheng
Exemplar Logic Inc.
San Jose, CA 95131

Ching-Wei Yeh
National Chung Cheng Univ.
Chia-Yi, Taiwan, R.O.C.

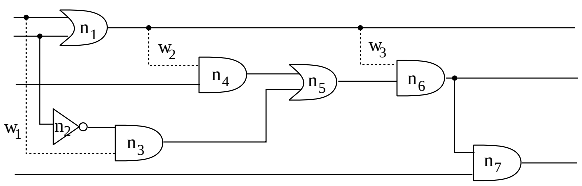


Figure 1: 3 redundant wires

Abstract[1]

Redundancy removal is an important step in combinational logic optimization. After a redundant wire is removed, other originally redundant wires may become irredundant, and some originally irredundant wires may become redundant. When multiple redundancies exist in a circuit, this creates a problem where we need to decide which redundancy to remove first. In this paper, we present an analysis and a very efficient heuristic to deal with multiple redundancies. We associate with each redundant wire a Boolean function that describes how the wire can remain redundant after removing other wires. When multiple redundancies exist, this set of Boolean functions characterizes the global relationship among redundancies.

1 Introduction

A redundant wire in a circuit is a wire whose removal does not change the circuit's functionality. Although not affecting the behavior of a circuit's primary outputs, removing a redundancy does change the functionalities of internal nodes. As a result, after removing a redundancy, other originally redundant wires may become irredundant, and some originally irredundant wires may become redundant. When multiple redundancies exist in a circuit, this creates a problem where we need to decide which redundancy to remove first. For example, consider the circuit in Fig. 1, where wires w_1, w_2, and w_3 are redundant. If w_3 is removed first, w_1 and w_2 are no longer redundant and hence cannot be further removed. On the other hand, if w_1 is removed first, w_2 is still redundant in the new circuit and hence can be further removed. In this example, removing w_1 and w_2 would give us a smaller circuit than if we remove w_3 alone.

Multiple redundancies exist not only unintentionally but also intentionally. Many logic optimization algorithms (e.g.: [3][4][5]) use the philosophy of first adding some redundancies and then removing other redundancies elsewhere, with the goal that the removed ones give us more "gains" than the added ones. In this type of intentionally introduced redundancies, removing multiple redundancies in a good order is very important to the final quality of the circuits.

In this paper, we present both a theoretical analysis and a very efficient heuristic to deal with multiple redundancies. In this paper we tackle the redundancy removal problem when multiple redundancies are present. We associate a Boolean function, termed *redundancy assurance function*, with each redundant wire. The redundancy assurance function of a redundant wire describes how the redundant wire can remain its redundancy when some other wires are removed. For example, consider again the redundant wires w_1, w_2, and w_3 in Fig. 1. We say that the redundancy assurance function of wire w_3 is $R_{w_3}=p_{w_1}p_{w_2}$, where variable p_{w_i} ($\overline{p_{w_i}}$) represents the presence (absence) of wire w_i. The meaning of this redundancy assurance function R_{w_3} is that, if w_1 and w_2 are both present in the circuit, w_3 remains redundant. In contrast, the redundancy assurance functions of w_1 and w_2 are $R_{w_1}=R_{w_2}=p_{w_3}$, meaning that w_1 and w_2 are both redundant as long as w_3 is kept in the circuit. We can see that after each redundant wire's redundancy assurance function is calculated, we have a global view of the correlation among all the redundant wires.

2 Background review

For simplicity, throughout this paper we only consider circuits with AND, OR, and INV gates. There are two kinds of redundancies, the stuck-at-1 redundancy and the stuck-at-0 redundancy. We say that a wire w is *stuck-at-1(0) redundant*, or simply *redundant* when the context is clear, if w's stuck-at-1(0) fault is untestable[1]. Let w be a wire in a given circuit. When wire w is stuck-at-1(0) redundant, we say we can *remove* w because we can replace w with a constant 1(0), in which case we also say w is absent or not present.

2.1 A precise algorithm

Given a circuit, let $X = \{x_1, x_2, \cdots, x_p\}$ be the set of primary inputs and $F = \{f_1, f_2, \cdots, f_q\}$ be the set of primary outputs. We denote $n_i(X)$ as the function of an internal node n_i in terms of primary inputs, and denote $f_i(X, w_j)$ as the function of primary output f_i in terms of the primary inputs and wire w_j. Also let $B_{v_i}f$ be the *Boolean difference* operator of function f with respect to variable v_i. In other words, $B_{v_i}f = f_{v_i=1} \oplus f_{v_i=0}$, where $f_{v_i=1}$ and $f_{v_i=0}$ are the *cofactor* operator of function f with respect to v_i=1 and v_i=0, respectively.

[1] Supported in part by a grant from the National Science Council of R.O.C. under contract no. NSC-87-2215-E-194-008.

Assuming n is the driving node for a wire w_j, it is well known that wire w_j is stuck-at-0 redundant if and only if

$$n(X) \cdot [\sum_{i=1}^{q} B_{w_j} f_i(X, w_j)] = 0. \tag{1}$$

And similarly, w_j is stuck-at-1 redundant if and only if

$$\overline{n(X)} \cdot [\sum_{i=1}^{q} B_{w_j} f_i(X, w_j)] = 0. \tag{2}$$

In equation 1, the first term $n(X)$ characterizes the primary input combinations for activating the stuck-at-0 fault, and the second term $\sum_{i=1}^{q} B_{w_j} f_i(X, w_j)$ characterizes the primary input combinations for observing the fault at any primary output. The AND of these two terms characterizes all the test vectors at the primary inputs that can detect this fault. For stuck-at-1 fault, we just need to complement the first term, as in Equation 2, since only the activating condition needs to be inverted as compared to a stuck-at-0 fault.

The above equations form an algorithm for determining if a given wire is redundant. In terms of a whole circuit, if we construct and check these equations for every wire, we are guaranteed to find all redundancies. Note that the above equations are a necessary and sufficient condition for a wire to be redundant. We therefore say this algorithm is *precise*, i.e., any wire identified by the algorithm as redundant is indeed redundant and any redundant wire is guaranteed to be found by the algorithm. In practice, however, we rarely use such an algorithm because we usually cannot afford constructing such equations due to the space/time explosion problem of Boolean functions.

2.2 Heuristic

We review a well-known heuristic [1] for identifying redundancy and also define some terminology for later discussion. The *dominators* of a wire w is a set of nodes D such that all the paths from w to any primary output have to pass through all the nodes in D. Given a dominator n of a wire w, the *side inputs* of dominator n are n's immediate inputs not in the transitive fanout of wire w. The value v of an input to a node is said to be *controlling* if v determines the value of the node's output regardless of the values of the other inputs. The controlling value is 1 for an OR gate and 0 for an AND gate. The inverse of the controlling value is called *noncontrolling* or *sensitizing* value.

To generate a test vector for a stuck-at-0 (stuck-at-1) fault at a wire w, we must assign a value 1 (0) at the driving node of w to activate the fault. Furthermore, for the fault to be observable at any primary output, we also must assign the sensitizing values to all the side inputs of all the dominators of wire w. To check for the redundancy of wire w, we then check if these assignments are consistent, the process of which is called *implication*.

3 Conceptual model

Let $W = \{w_1, w_2, \cdots, w_n\}$ be the set of redundant wires in a given circuit C. Without loss of generality, we only define the redundancy assurance function for redundant wire w_1. The *redundancy assurance function* of w_1, denoted by R_{w_1}, is a Boolean function defined in terms of variables $p_{w_2}, \cdots, p_{w_n}$, where p_{w_i} ($\overline{p_{w_i}}$) represents the presence (absence) of wire w_i in circuit C. A minterm $p_{w_2} \cdots p_{w_n}$ is in the on-set of R_{w_1} if w_1 is still redundant after removing all w_i's in $\{w_i \mid p_{w_i}{=}0\}$ and keeping all w_i's in $\{w_i \mid p_{w_i}{=}1\}$.

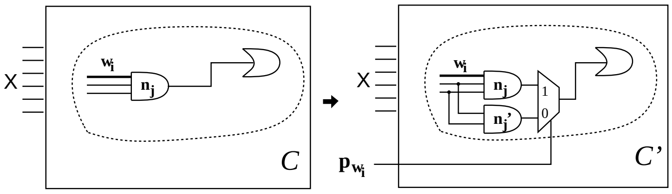


Figure 2: Local transformation

For example, in Fig. 1, the set of redundant wires are $W = \{w_1, w_2, w_3\}$. One can find that $R_{w_3}(p_{w_1}, p_{w_2}) = p_{w_1} p_{w_2} = \{11\}$, which means that w_3 is redundant if w_1 and w_2 are both not removed. As another example, $R_{w_1}(p_{w_2}, p_{w_3}) = p_{w_3} = \{01, 11\}$, which means that w_1 is redundant if w_3 is not removed. In this case, whether w_2 is present or not does not affect the redundancy of w_1. Similarly, one can find $R_{w_2}(p_{w_1}, p_{w_3}) = p_{w_3} = \{01, 11\}$. We can see that these redundancy assurance functions establish the relationship among redundancies.

3.1 A precise algorithm

Given a circuit C, let $X = \{x_1, x_2, \cdots, x_p\}$ be the set of primary inputs and $F = \{f_1, f_2, \cdots, f_q\}$ be the set of primary outputs. Also let $W = \{w_1, w_2, \cdots, w_n\}$ be the set of redundant wires in circuit C. We transform the given circuit C to a new circuit C' by adding n primary inputs $P = \{p_{w_1}, p_{w_2}, \cdots, p_{w_n}\}$. For each redundant wire w_i, we locally perform the transformation shown inside the dotted oval line in Fig. 2. The left side of Fig. 2 shows the original circuit C with redundant wire w_i under transformation and the right side shows the circuit C' after the transformation. We first duplicate node n_j, which is driven by wire w_i, to a new node n_j'. Then we remove redundant wire w_i only on node n_j' and keep n_j intact. Finally we add a 2-to-1 multiplexer to select between n_j and n_j'. The select line of the multiplexer is a new primary input p_{w_i}, with $p_{w_i}{=}1$ selecting n_j and $p_{w_i}{=}0$ selecting n_j'.

We can easily see that each combination on the new primary inputs $P = \{p_{w_1}, p_{w_2}, \cdots, p_{w_n}\}$ in the transformed circuit C' corresponds to a configuration on the original circuit C where all wires in $\{w_i \mid p_{w_i}{=}0\}$ are removed and all wires in $\{w_i \mid p_{w_i}{=}1\}$ are kept intact. In our formulation, some of these combinations on P may actually result in a different functionality on circuit C' compared with the original circuit C. When all the newly added primary input p_{w_i}'s are set to 1, the functionality of C' is guaranteed to be identical to that of the original circuit C because all the multiplexers are selecting the same connection as in the original circuit C. To make sure our formulation does not change circuit C's behavior, we express the function of the primary outputs F_i's of C' in terms of the original primary inputs X and the newly added primary inputs P, and we must have

$$\prod_{i=1}^{q} (F_i(X, P) \equiv f_i(X)) = 1, \tag{3}$$

where F_i's and f_i's are the primary outputs in circuit C' and C, respectively, and $\equiv$ is the equivalence (or exclusive NOR) operator. Since for all the combinations of the original primary inputs X Equation 3 must hold, we apply the concensus operator to Equation 3, and we have

$$L(P) = \forall_X \prod_{i=1}^{q} [F_i(X,P) \equiv f_i(X)], \tag{4}$$

where $\forall$ is the concensus operator, i.e., $\forall_{x_i} = f_{x_i=0} \cdot f_{x_i=1}$. $L(P)$ is a function in terms of only the newly added primary inputs P. Any minterm in $L(P)$ represents a configuration that makes circuit C equivalent to circuit C' under any combination of the original primary input X. Since $p_{w_i}=0$ means redundant wire w_i can be removed, the redundancy assurance function of a redundant wire w_i is then simply the cofactor with respect to $p_{w_i}=0$ of Equation 4. In other words,

$$R_{w_i} = L_{p_{w_i}=0}(P) \tag{5}$$

4 Heuristic

Like the case reviewed in Section 2.1, given a set of redundant wires in a circuit, it is impractical to precisely calculate the redundancy assurance functions. In this section we present a very efficient algorithm to approximate the problem. We say that our algorithm is an approximation in the sense that we will only find a subset of the on-set minterms in the redundancy assurance function. For each redundant wire w, this means that when the redundancy assurance function R_w found by our approximation is 1, w is indeed redundant, while when R_w is 0, we do not know if w is redundant and would simply claim it as not redundant. This is similar to the situation on redundancy identification of the heuristic reviewed in Section 2.2 versus the precise algorithm reviewed in Section 2.1.

We first need to identify as many redundancies as possible before we tackle the problem of multiple redundancies. We assume that some identification process is done a priori. For simplicity, we will present our multiple-redundancy algorithm by assuming that this identification process is exactly the heuristic reviewed in Section 2.2. Although presented with this particular redundancy identification technique in mind, the philosophy of our algorithm can be easily generalized to many other redundancy identification and/or implication techniques.

We first discuss in more detail the heuristic we use for identifying redundant wire. Let wire w_i be the wire that we want to perform stuck-at-v redundancy check. We first assign the fault activating value $\overline{v}$ to the node driving w_i and assign the sensitizing values on the side inputs of the dominators of w_i. These assignments are the starting point of the implication phase. Then we repeatly perform direct implication on nodes having some values assigned. Direct implication have four rules for AND gates, four rules for OR gates, and two rules for INV gates. In Table 1, the first column shows the rule names and the second column shows the rules. Take Rule A1 in Table 1 as an example. Node n_m has two fanins, node n_k through wire w_i and node n_l through wire w_j. If node n_k is somehow assigned value 0, then we imply that node n_m must also be assigned value 0, as indicated by the arrow in the figure inside the second column. Similarly we can easily derive all the other rules in the table. Due to space limit, we omit the simple case for INV gates. The implication process is finished either when we find a conflict on the assignments, or when all the gates with some assignments are all checked and exhausted. In the former case, we conclude that the wire w_i is redundant. In the latter case, all the value assignments are consistent and we do not know if w_i is redundant, in which case we simply claim that it is not redundant.

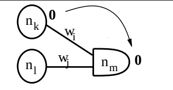
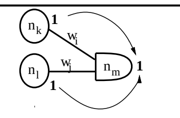
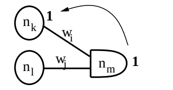
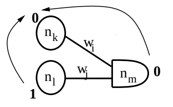
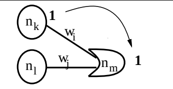
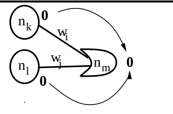
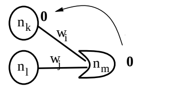
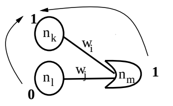
	implication rule	back propagation rule
A1		$C(n_m{=}0) = C(n_k{=}0)p_{w_i}$
A2		$C(n_m{=}1) = C(n_k{=}1)C(n_l{=}1)p_{w_i}p_{w_j} + C(n_k{=}1)\, p_{w_i}\overline{p_{w_j}} + C(n_l{=}1)\, \overline{p_{w_i}}p_{w_j}$
A3		$C(n_k{=}1) = C(n_m{=}1)p_{w_i}$
A4		$C(n_k{=}0) = C(n_m{=}0)C(n_l{=}1)p_{w_i}p_{w_j} + C(n_m{=}0)p_{w_i}\overline{p_{w_j}}$
O1		$C(n_m{=}1) = C(n_k{=}1)p_{w_i}$
O2		$C(n_m{=}0) = C(n_k{=}0)C(n_l{=}0)p_{w_i}p_{w_j} + C(n_k{=}0)p_{w_i}\overline{p_{w_j}} + C(n_l{=}0)\overline{p_{w_i}}p_{w_j}$
O3		$C(n_k{=}0) = C(n_m{=}0)p_{w_i}$
O4		$C(n_k{=}1) = C(n_m{=}1)C(n_l{=}0)p_{w_i}p_{w_j} + C(n_m{=}1)p_{w_i}\overline{p_{w_j}}$

Table 1: Implication rules and back propagation rules

After the redundancy identification process, we are given a set of redundant wires among which we want to determine which redundancies to remove first. Our goal is to calculate the redundancy assurance function for each redundant wire w_i. Recall that the redundancy assurance function R_{w_i} is expressed in terms of newly introduced Boolean variables p_{w_j}'s representing the presence or absence of other redundant wires w_j's. For ease of discussion, in the following we will say that every wire w_j, redundant or not, has an associated Boolean variable p_{w_j} representing w_j's presence. Since irredundant wires can never be removed, this little generalization is only for the convenience of notation and we have the understanding that p_{w_j} always equals to 1 if w_j is not redundant.

For each redundant wire w, we have a trace of implication steps that eventually leads to a conflict on some node. The philosophy of our algorithm is to find R_w, w's redundancy assurance function, by tracing back these implication steps. We recursively define the *constraint function* for each node n_j involved in the trace of the implication steps. Let node n be assigned value v somewhere in the implication steps. Intuitively, the *constraint function* $C(n{=}v)$ of the condition $n{=}v$ is a Boolean function, in terms of all the new variables p_{w_i}'s, to represent the configuration in which condition $n{=}v$ can be guaranteed. Since the starting point of the implication steps is the assignments either on the node driving wire w or on the side inputs of the dominators of wire w, we define the recursion basis as

$$C(n{=}v) = \begin{cases} 1 & \text{if } n \text{ is the node driving wire } w \text{ and is assigned value } v \\ p_{w_i} & \text{if } n \text{ is a side input connecting through wire } w_i \text{ to a dominator node of } w \text{ and } n \text{ is assigned value } v \end{cases} \tag{6}$$

The definition of $C(n{=}v)$ on all other nodes n's with value v assigned depends on where value v is implied from and is recursively defined. The recursive definitions are shown in the third column of Table 1. For convenience, we also call these recursive definitions as the *back propagation* rules. Take Rule A1 in Table 1 as an example. First we note that the implication rule in the second column is a situation for an AND gate where node n_m is assigned 0 because node n_k was assigned 0. Since we are formulating a problem where wires may be removed, the implication $n_m{=}0$ holds as long as the n_k is 0 and the wire w_i is present. We therefore in the third column have the back propagation rule as $C(n_m{=}0) = C(n_k{=}0)p_{w_i}$. A more complicated example is Rule A2. The implication rule in the second column shows that node n_m is assigned 1 because both inputs, n_k and n_l, were assigned 1. To guarantee implication $n_m{=}1$ holds under potential removals on wires, we can either have

1. $C(n_k{=}1)C(n_l{=}1)p_{w_i}p_{w_j}$, which means both w_i and w_j are present and both n_k and n_l are 1,
2. $C(n_k{=}1)p_{w_i}\overline{p_{w_j}}$, which means w_j is removed, w_i is present and $n_k{=}1$, or
3. $C(n_l{=}1)\overline{p_{w_i}}p_{w_j}$, which means w_i is removed, w_j is present and $n_l{=}1$.

ORing these three functions is what we have in the third column of Rule A2.

The rules in the third column of Table 1, together with the recursion basis in Equation 6, completes our definition of the constraint function $C(n{=}v)$ for a given assignment $n{=}v$. Note that in our formulation we say values can be assigned on nodes but not on wires. We always say that wires are to be present or absent, and never to assume a value. Also note that in our definition, the constraint function is associated with an implication rule. In other words, the constraint functions are defined only when an implication occurs, and are undefined for other cases, such as a node without a value assigned.

Once we understand the definition of the constraint functions, we are ready to present our algorithm. Our algorithm starts at the node that has conflicting assignments. Let n be the node where the conflict occurs, our algorithm back traces the constraint functions $C(n{=}1)$ and

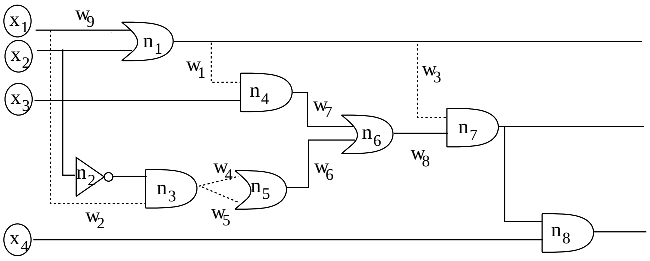


Figure 3: A circuit with 5 redundant wires

$C(n{=}0)$ separately. This can be best explained with an example. Fig. 3 shows a circuit with 5 redundant wires, w_1, w_2, w_3, w_4, and w_5. Let us focus on wire w_3. The trace of implication steps for finding w_3 stuck-at-1 redundant are

$$\begin{array}{lcl} n_6 = 1 \text{ (side input to } n_7) & & \\ n_1 = 0 \text{ (activating fault)} & \Longrightarrow & x_1 = 0 \\ x_1 = 0 & \Longrightarrow & n_3 = 0 \\ n_3 = 0 \text{ or } n_3 = 0 & \Longrightarrow & n_5 = 0 \\ n_6 = 1 \text{ and } n_5 = 0 & \Longrightarrow & n_4 = 1 \\ n_1 = 0 & \Longrightarrow & n_4 = 0 \text{ (conflict!)} \end{array}$$

Now we apply the back propagation rules of constraint functions shown in the third column of Table 1. Since n_4 is the node where the conflict occurs, we separately back trace $C(n_4{=}0)$ and $C(n_4{=}1)$. On the $C(n_4{=}0)$ trace, by Rule A1, we have

$$C(n_4{=}0) = C(n_1{=}0)p_{w_1}.$$

Since $n_1{=}0$ is the activating value assignment for w_3 stuck-at-1 fault, we reach our recursion basis in Equation 6 and have $C(n_1{=}0) = 1$. Hence,

$$C(n_4{=}0) = p_{w_1} \tag{7}$$

On the $C(n_4{=}1)$ trace, by Rule O4, we have

$$C(n_4{=}1) = C(n_6{=}1)C(n_5{=}0)p_{w_6}p_{w_7} + C(n_6{=}1)\overline{p_{w_6}}p_{w_7} \tag{8}$$

Since w_6 and w_7 are not redundant wires, we have $p_{w_6} = 1$ and $p_{w_7} = 1$. Furthermore, since $n_6{=}1$ is the assignment on the side input of w_3's dominator, we reach our recursion basis in Equation 6 and have $C(n_6{=}1) = p_{w_8}$. Since w_8 is again not a redundant wire, we have $C(n_6{=}1) = p_{w_8} = 1$. Simplifying Equation 8, we have

$$C(n_4{=}1) = C(n_5{=}0).$$

Applying Rule O2, $C(n_5{=}0)$, we have

$$\begin{aligned} & C(n_4{=}1) \\ = \; & C(n_3{=}0)C(n_3{=}0)p_{w_4}p_{w_5} + C(n_3{=}0)p_{w_4}\overline{p_{w_5}} \\ & + C(n_3{=}0)\overline{p_{w_4}}p_{w_5} \\ = \; & C(n_3{=}0)(p_{w_4} + p_{w_5}) \end{aligned}$$

Applying Rule A1, we have $C(n_3 = 0) = C(x_1 = 0)p_{w_2}$. Hence,

$$C(n_4{=}1) \;=\; C(x_1{=}0)p_{w_2}(p_{w_4} + p_{w_5})$$

Applying Rule O3, we have $C(x_1 = 0) = C(n_1 = 0)p_{w_9}$. Hence,

$$C(n_4{=}1) \;=\; C(n_1{=}0)p_{w_9}p_{w_2}(p_{w_4} + p_{w_5})$$

Since w_9 is not a redundant wire, we have $p_{w_9}{=}1$. Since $n_1{=}0$ is the fault activating value, we reach the recursion basis and have $C(n_1{=}0) = 1$. We therefore have

$$C(n_4{=}1) = p_{w_2}(p_{w_4} + p_{w_5}) \quad (9)$$

The above example illustrates the first step of our algorithm—to find the two constraint functions at the conflicting node. In the above example, we conclude with the two constraint functions in Equations 7 and 9 for the conflicting node n_4. The meaning of Equation 7 is that n_4 will have an implication value 0 if we keep wire w_1 present. Similarly, the meaning of Equation 9 is that n_4 will have an implication value 1 if we keep wire w_2 and one of wires $\{w_4, w_5\}$ present. Understanding this point, the second step of our algorithm is straightforward. Recall that w_3 is the wire we want to calculate the redundancy assurance function for. To make sure w_3 is redundant, all we have to do is to AND the two constraint functions found at the conflicting node n_4. ANDing the two constraint functions guarantees that there is still a conflict at node n_4 and therefore w_3 is still redundant. In the above example, we AND Equations 7 and 9, and we have the redundancy assurance function

$$\begin{aligned} & R_{w_3}(p_{w_1}, p_{w_2}, p_{w_4}, p_{w_5}) \\ = \; & C(n_4{=}0) \cdot C(n_4{=}1) \\ = \; & p_{w_1} p_{w_2} (p_{w_4} + p_{w_5}) \end{aligned}$$

Once we have all the redundancy assurance functions, which characterize the global relationship among redundancies, we can then easily build a redundancy removal algorithm that exploits this global relationship. Due to space limitation, we refer the details to our technical report in [2].

5 Experimental results

We test our algorithm on a set of MCNC benchmarks and compare the result to that obtained with the redundancy removal algorithm in SIS [6]. Table 2 shows the experimental results. In our experiment, an input circuit is first pre-processed by *script.boolean* [6] and then decomposed into AND and OR gates. The second column of Table 2 shows the initial literal count after this pre-processing. Some redundant wires are then added to each circuit in a similar way to the method in [3] [4] [5]. The resulting circuits serve as our initial circuits for comparison. The third and fourth columns show the results obtained with SIS and our algorithm, respectively. The last two columns show the CPU time spent on SIS and our algorithm.

At the last row of Table 2, we compare the percentage of improvements. We normalize all the results so that the result produced by our algorithm is 1. As shown in the table, our result is on average 8% better than the results obtained with SIS. One surprise shown in Table 2 is that in a few cases our algorithm not only produces better result but also runs faster than SIS's algorithm. This surprise may be due to two reasons. First, without a global consideration of multiple redundancies, SIS algorithm has to restart a whole new round of redundancy identification process after removing the first encountered redundancy in a given circuit. As a result, there may be many rounds of such identification process. In contrast, our algorithm tries to remove as many identified redundancies as possible at one shot, and therefore the number of new rounds is much smaller than that of SIS. Second, after filtering out some easy-to-detect irredundant wires by random fault simulation, the algorithm in SIS tries very extensively to determine if a wire is redundant. Since the check has to be done for all the remaining wires, this slows down the process.

		result		CPU	
circuit	initial	SIS	ours	SIS	ours
9symml	360	354	338	2.1	2.6
alu2	639	547	505	12.2	13.44
apex	1126	1113	1063	19.5	54.0
b9	164	160	148	0.8	0.6
cm82a	40	38	32	0.1	0.28
cm85a	82	76	58	0.4	0.47
cmb	62	60	54	0.1	0.19
cordic	108	108	89	0.2	0.25
comp	218	199	152	0.9	1.13
count	233	225	208	1.9	1.3
f51m	212	192	179	1.4	1.28
my_adder	336	284	270	4.2	1.73
pm1	65	63	57	0.2	0.28
t481	1105	943	812	29.1	71.10
term1	352	228	216	3.9	4.38
too_large	613	595	547	11.6	24.60
ttt2	305	281	263	2.0	1.3
z4ml	60	54	48	0.3	0.35
C432	316	252	224	3.0	2.52
C6288	4296	4242	3817	870.3	313.3
C7552	3617	3218	3071	316.3	92.0
C880	639	621	699	4.9	2.18
Total	13822	12740	11787	1265.9	535.28
%	1.17	1.08	1.00		

Table 2: Experimental results

6 Conclusion

In this paper, we first formulate a conceptual model and discuss the precise solution for the multiple-redundancy problem. For each redundant wire, we define the redundancy assurance function to model the global relationship between the redundant wire with other redundancies. We then present a very efficient heuristic to solve the multiple-redundancy problem for practical circuits. The experimental results are very encouraging.

References

[1] M. Abramovici, M.A. Breuer, A.D. Friedman, "Digital Systems Testing and Testable Design," *IEEE Press,* 1994.

[2] S.C. Chang, D.I. Cheng, C.W. Yeh, "On Removing Multiple Redundancies in Combinational Circuits," *Tech. Report 97-004 C.S. Dept., National Chung Cheng University.*

[3] S.C. Chang, L. VanGinneken, and M. Marek-Sadowska, "Fast Boolean Optimization by Rewiring," *Proc. ACM/IEEE ICCAD-96, pp. 262-269,* Nov. 1996.

[4] L. Entrena and K.T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," *Proc. ACM/IEEE ICCAD-93,* Nov. 1993.

[5] W. Kunz and D.K. Pradhan, "Multi-Level Logic Optimization by Implication Analysis," *Proc. ACM/IEEE ICCAD-94, pp. 6-13,* Nov. 1994.

[6] E.Sentovich etc., "SIS: A System for Sequential Circuit Synthesis" *Memorandum No. UCB/ERL M92/41, University of California, Berkeley.*