On Removing Multiple Redundancies in Combinational Circuits

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

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

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

Abstract

Redundancy removal is an important step in combinational logic optimization. After a redundant wire is removed, other originally redundant wires may become redundant, 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 $p_{w_i}$ ($[p_{w_i}]$) 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 $X = \{x_1, x_2, \ldots, x_p\}$ be the set of primary inputs and $F = \{f_1, f_2, \ldots, f_s\}$ 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 $B_{v_i}$ be the Boolean difference operator of function $f$ with respect to variable $v_i$. In other words, $B_{v_i} = f_{v_i=1} \oplus f_{v_i=0}$, 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.
Assuming \( n \) is the driving node for a wire \( w_i \), it is well known that wire \( w_i \) is stuck-at-0 redundant if and only if
\[
 n(X) \cdot \sum_{i=1}^{n} B_{w_i} f_i (X, w_i) = 0.
\]  \tag{1}
And similarly, \( w_i \) is stuck-at-1 redundant if and only if
\[
 n(X) \cdot \sum_{i=1}^{n} B_{w_i} f_i (X, w_i) = 0.
\]  \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 \( \sum_{i=1}^{n} B_{w_i} f_i (X, w_i) \) 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 these equations for every wire, 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 \( W = \{ w_1, w_2, \ldots, w_n \} \) 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 \( w_i \). The redundancy assurance function of \( w_i \), denoted by \( R_{w_i} \), is a Boolean function defined in terms of variables \( p_{o_1}, \ldots, p_{o_k} \), where \( p_{o_i} \) represents the presence (absence) of wire \( w_i \) in circuit \( C \). A minterm \( p_{o_1} \cdots p_{o_k} \) is in the on-set of \( R_{w_i} \) if \( w_i \) is still redundant after removing all \( w_i \)'s in \( \{ w_i | p_{o_i}=0 \} \) and keeping all \( w_i \)'s in \( \{ w_i | p_{o_i}=1 \} \).

For example, in Fig. 1, the set of redundant wires are \( W = \{ w_1, w_2, w_3 \} \). One can find that \( R_{w_1}(p_{o_1}, p_{o_2}) = p_{o_1} p_{o_2} = \{ 11 \} \), which means that \( w_3 \) is redundant if \( w_1 \) and \( w_2 \) are both not removed. As another example, \( R_{w_1}(p_{o_1}, p_{o_2}) = p_{o_2} = \{ 01 \} \), which means that \( w_i \) is redundant if \( w_5 \) is not removed. In this case, whether \( w_5 \) is present or not does not affect the redundancy of \( w_i \). Similarly, one can find \( R_{w_1}(p_{o_1}, p_{o_3}) = p_{o_2} = \{ 01 \} \). We can see that these redundancy assurance functions establish the relationship among redundancies.

3.1 A precise algorithm

Given a circuit \( C \), let \( X = \{ x_1, x_2, \ldots, x_p \} \) be the set of primary inputs and \( F = \{ f_1, f_2, \ldots, f_k \} \) be the set of primary outputs. Also let \( W = \{ w_1, w_2, \ldots, w_n \} \) 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 \( P = \{ p_{o_1}, p_{o_2}, \ldots, p_{o_k} \} \). 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 \( n_i \), which is driven by wire \( w_i \), to a new node \( n_i' \). Then we remove redundant wire \( w_i \) only on node \( n_i \) and keep \( n_i \) intact. Finally we add a 2-to-1 multiplexer to select between \( n_i \) and \( n_i' \). The select line of the multiplexer is a new primary input \( p_{o_i} \), with \( p_{o_i}=1 \) selecting \( n_i \) and \( p_{o_i}=0 \) selecting \( n_i' \).

We can easily see that each combination on the new primary inputs \( P = \{ p_{o_1}, p_{o_2}, \ldots, p_{o_k} \} \) in the transformed circuit \( C' \) corresponds to a configuration on the original circuit \( C \) where all wires in \( \{ w_i | p_{o_i}=0 \} \) are removed and all wires in \( \{ w_i | p_{o_i}=1 \} \) 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 \( p_{o_i} \)'s are set to 0, 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
\[
 \prod_{i=1}^{k} (F_i(X, P) \equiv f_i(X)) = 1,
\]  \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 consensus operator to Equation 3, and we have

$$L(P) = \bigvee_{i=1}^{n} [F_i(X, P) \equiv f_i(X)],$$

where $V$ is the consensus operator, i.e., $\forall x \in f_{x_{n-1}} \cdot f_{x_{n-2}} \cdot \ldots \cdot f_{x_0} \cdot f_{x_1}$, and $L(P)$ 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 $p_{w_0} = 0$ means redundant wire $w_0$ can be removed, the redundancy assurance function of a redundant wire $w_0$ is then simply the cofactor with respect to $p_{w_0} = 0$ of Equation 4. In other words,

$$R_{w_0} = L_{p_{w_0}=0}(P)$$

### 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 $v$ 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 repeatedly 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

<table>
<thead>
<tr>
<th>implication rule</th>
<th>back propagation rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>$C(n_m=0) = C(n_k=0)p_{w_i}$</td>
</tr>
<tr>
<td>A2</td>
<td>$C(n_m=1) = C(n_k=1</td>
</tr>
<tr>
<td>A3</td>
<td>$C(n_k=1) = C(n_m=1)p_{w_i}$</td>
</tr>
<tr>
<td>A4</td>
<td>$C(n_k=0) = C(n_m=0</td>
</tr>
<tr>
<td>O1</td>
<td>$C(n_m=1) = C(n_k=1)p_{w_i}$</td>
</tr>
<tr>
<td>O2</td>
<td>$C(n_m=0) = C(n_k=0</td>
</tr>
<tr>
<td>O3</td>
<td>$C(n_k=0) = C(n_m=0)p_{w_i}$</td>
</tr>
<tr>
<td>O4</td>
<td>$C(n_k=1) = C(n_m=1</td>
</tr>
</tbody>
</table>

Table 1: Implication rules and back propagation rules
tion steps. We recursively define the constraint function for each node \( n_i \) 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 \( p_{v_i} ' s \), 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

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

(6)

The definition of \( C(n=v) \) on all other nodes \( n \) 's with value \( v \) assigned depends on where \( v \) 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 \( C(n_m=0) = C(n_k=0)\rho_{w_i} \). 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. \( C(n_k=1)C(n_l=1)\rho_{w_i}p_{w_i} \), which means both \( w_i \) and \( w_j \) are present and both \( n_k \) and \( n_l \) are 1,
2. \( C(n_k=1)\rho_{w_i}\overline{p_{w_i}} \), which means \( w_i \) is removed, \( w_j \) is present and \( n_k=1 \), or
3. \( C(n_l=1)\overline{p_{w_i}}p_{w_i} \), 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

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

\[
\begin{align*}
& n_k = 1 \text{ (side input to } n_j) \\
& n_l = 0 \text{ (activating fault)} \quad \Rightarrow \quad x_1 = 0 \\
& x_1 = 0 \quad \Rightarrow \quad n_3 = 0 \\
& n_a = 0 \text{ or } n_b = 0 \quad \Rightarrow \quad n_8 = 0 \\
& n_a = 1 \text{ and } n_b = 0 \quad \Rightarrow \quad n_4 = 1 \\
& n_1 = 0 \quad \Rightarrow \quad n_4 = 0 \text{ (conflict!)}
\end{align*}
\]

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 \( C(n_4=1) \).

As an example, the circuit in Figure 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

\[
\begin{align*}
& n_k = 1 \text{ (side input to } n_j) \\
& n_l = 0 \text{ (activating fault)} \quad \Rightarrow \quad x_1 = 0 \\
& x_1 = 0 \quad \Rightarrow \quad n_3 = 0 \\
& n_a = 0 \text{ or } n_b = 0 \quad \Rightarrow \quad n_8 = 0 \\
& n_a = 1 \text{ and } n_b = 0 \quad \Rightarrow \quad n_4 = 1 \\
& n_1 = 0 \quad \Rightarrow \quad n_4 = 0 \text{ (conflict!)}
\end{align*}
\]

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

\[
C(n_4=0) = C(n_4=0)p_{w_1}
\]

(7)

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,

\[
C(n_4=0) = p_{w_4}
\]

On the \( C(n_4=1) \) trace, by Rule O4, we have

\[
C(n_4=1) = C(n_4=1)C(n_5=0)p_{w_5}p_{w_6} + C(n_5=0)p_{w_4}p_{w_6}
\]

(8)

Since \( w_6 \) and \( w_7 \) are not redundant wires, we have \( p_{w_6} = 1 \) and \( p_{w_7} = 1 \). Furthermore, since \( n_1 = 1 \) is the assignment on the side input of \( w_3 \)’s dominator, we reach our recursion basis in Equation 6 and have \( C(n_5=1) = p_{w_4} \). Since \( w_5 \) is again not a redundant wire, we have \( C(n_5=1) = p_{w_5} = 1 \).

Simplifying Equation 6, we have

\[
C(n_4=1) = C(n_5=0)
\]

Applying Rule O2, \( C(n_5=0) \), we have

\[
C(n_4=1) = C(n_5=0)C(n_5=0)p_{w_4}p_{w_5} + C(n_5=0)p_{w_4}p_{w_5}
\]

(9)

Applying Rule A1, we have \( C(n_5=0) = C(x_1=0)p_{w_2} \). Hence,

\[
C(n_4=1) = C(x_1=0)p_{w_2}p_{w_4} + p_{w_4}
\]

Applying Rule O3, \( C(x_1=0) \), we have \( C(x_1=0) = C(n_1=0)p_{w_4} \). Hence,

\[
C(n_4=1) = C(n_1=0)p_{w_4}p_{w_5} + p_{w_4}
\]

Since \( w_5 \) is not a redundant wire, we have \( p_{w_5} = 1 \). Since \( n_1 = 0 \) is the fault activating value, we reach the recursion basis and have \( C(n_1=0) = 1 \). We therefore have
\[ C(n_4 = 1) = p_{w_2} (p_{w_4} + p_{w_5}) \]  

(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 \( w_3 \) 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 \( n_4 \). And the two constraint functions guarantee 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

\[ R_{w_3}(p_{w_1}, p_{w_2}, p_{w_4}, p_{w_5}) = C(n_4 = 0) \cdot C(n_4 = 1) = p_{w_1} p_{w_2} (p_{w_4} + p_{w_5}) \]

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.

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.

<table>
<thead>
<tr>
<th>circuit</th>
<th>initial</th>
<th>result</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIS</td>
<td>ours</td>
<td>SIS</td>
</tr>
<tr>
<td>weymdl</td>
<td>360</td>
<td>354</td>
<td>338</td>
</tr>
<tr>
<td>ah2</td>
<td>639</td>
<td>547</td>
<td>505</td>
</tr>
<tr>
<td>apex</td>
<td>1126</td>
<td>1113</td>
<td>1063</td>
</tr>
<tr>
<td>b9</td>
<td>164</td>
<td>160</td>
<td>148</td>
</tr>
<tr>
<td>cm82a</td>
<td>40</td>
<td>38</td>
<td>32</td>
</tr>
<tr>
<td>cm85a</td>
<td>82</td>
<td>76</td>
<td>58</td>
</tr>
<tr>
<td>cmb</td>
<td>62</td>
<td>60</td>
<td>54</td>
</tr>
<tr>
<td>comp</td>
<td>108</td>
<td>108</td>
<td>88</td>
</tr>
<tr>
<td>count</td>
<td>218</td>
<td>199</td>
<td>152</td>
</tr>
<tr>
<td>f5im</td>
<td>212</td>
<td>192</td>
<td>170</td>
</tr>
<tr>
<td>my_adder</td>
<td>336</td>
<td>284</td>
<td>270</td>
</tr>
<tr>
<td>pml</td>
<td>65</td>
<td>63</td>
<td>57</td>
</tr>
<tr>
<td>t185</td>
<td>1105</td>
<td>943</td>
<td>812</td>
</tr>
<tr>
<td>terml</td>
<td>352</td>
<td>228</td>
<td>216</td>
</tr>
<tr>
<td>too_large</td>
<td>613</td>
<td>595</td>
<td>547</td>
</tr>
<tr>
<td>t12</td>
<td>305</td>
<td>281</td>
<td>263</td>
</tr>
<tr>
<td>s4ml</td>
<td>60</td>
<td>54</td>
<td>48</td>
</tr>
<tr>
<td>C432</td>
<td>316</td>
<td>252</td>
<td>224</td>
</tr>
<tr>
<td>C6288</td>
<td>4296</td>
<td>4242</td>
<td>3817</td>
</tr>
<tr>
<td>C7552</td>
<td>3617</td>
<td>3218</td>
<td>3071</td>
</tr>
<tr>
<td>C880</td>
<td>639</td>
<td>621</td>
<td>699</td>
</tr>
<tr>
<td>Total</td>
<td>13822</td>
<td>12740</td>
<td>11787</td>
</tr>
<tr>
<td>%</td>
<td>1.17</td>
<td>1.08</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 2: Experimental results

References