IBAW: An Implication-Tree Based Alternative-Wiring Logic Transformation Algorithm

Wangning Long, Yu-Liang Wu*, and Jinian Bian
Dept. Computer Sci. & Tech., Tsinghua University, Beijing, 100084, China
*Dept. Computer Sci. & Eng., The Chinese University of HK, Shatin, NT, Hong Kong

Abstract - The well-known ATPG-based alternative wiring technique, RAMBO, has been shown to be very useful because of its proven powerfulness and flexibility in attacking many design automation problems (e.g. logic optimization, circuit partitioning, and post-layout logic transformation .. etc). Since the ATPG based alternative wire locating procedure is the center engine for all its applications, speeding up of this process should be very crucial and useful.

We observe that the bottleneck of the technique lies in the costly redundancy tests among a large number of candidate alternative wires. In this paper, we develop a so-called implication-tree data structure which stores implication relationship between nodes with determined logic values, and propose a new ATPG-based alternative-wiring algorithm to speed up the engine. The algorithm, Implication-Tree Based Alternative-Wiring (IBAW), differs from other ATPG-based algorithms in terms that it selects source node of alternative wires from the implication-tree, which makes IBAW be able to trim out many unnecessary redundancy checking quite easily without calling for complicated procedures. Hence, it produces a steady speeding up of around 3.6 times faster while maintains the same rewiring capability of the original RAMBO. Our experimental results show that the overall circuit area optimized by IBAW can be slightly better than that by RAMBO, while the runtime is just one-half of the latter.

1 Introduction

Traditional multi-level logic synthesis systems, such as SIS [1], usually adopt algebraic and Boolean methods and so on to do logic transformations. In recent years, some ATPG (Automatic Test Pattern Generation) based alternative-wiring logic transformation algorithms [2-8] were proposed. Unlike SIS, this kind of methods use ATPG techniques, such as logic implication [9] and recursive learning [10] etc, to add a new wire to substitute the target-wire without changing the network's logic behavior. RAMBO (Redundancy Addition-and-removal for Multilevel Boolean Optimization) [2,3] is such an ATPG-based alternative-wiring algorithm.

When logic optimization is concerned, some algorithms adopt an add-first scheme, instead of the target-first scheme used by original RAMBO. They first add a redundant wire into the network and then look up the wires that become redundant after adding the new wire [6]. This scheme is good for logic optimization purpose, because adding a new wire (maybe with an additional gate) may cause several wires being removed. Several methods are proposed to quickly identify the redundant wires caused by the new added wire [7-8]. By excluding some wires that are essential for the redundancy of the new added wire, reference [7] diminishes the search space of the possible redundant wires, so as to speed up the program. In [8], a new direct RID method is proposed to speed up the redundancy identifying process. However, we notice that the add-first scheme is not suitable for post-layout logic transformation, because a post-layout logic synthesis tool would like to first select a target wire and then look up the alternative wires to substitute it. In this case, we have to use the original target-first RAMBO.

RAMBO has found wide applications in the area of logic optimization [6-7], post-layout logic restructuring [11-12] and circuit partitioning [13], etc. The latter two applications are for physical design automation of VLSI circuits. With the development of sub-micron VLSI technology, there is a wide appeal for the merge of logic synthesis and physical design. RAMBO is a logic synthesis tool that can be used for bridging the gap with physical design tools.

However, the main problem of the current ATPG-based alternative-wiring scheme is that it runs somewhat slowly, mainly because of the time consuming property of the ATPG procedure. The RAMBO algorithm first selects a wire as the target wire, and then look for a new redundant wire that will make the target-wire redundant, so as to substitute the target wire. If the new wire is redundant, it is a feasible alternative wire. Otherwise, adding the new wire will change the network's logic behavior and it should be discarded. As there is commonly quite a large number of candidate alternative wires whose redundancy need to be verified, the most CPU costly procedure in RAMBO is therefore lies on such kind redundancy check processes.

To improve the speed, we must reduce the times of calling the redundancy check.

We have noticed that the implication relationship between nodes can help us accelerate the ATPG-based alternative-wiring algorithm. For two candidate-wires that share a common destination node, assume the two source nodes are $g_1$ and $g_2$, which have determined logic value $a$ and $b$ respectively. If $g_1 = a$ implies $g_2 = b$ and the candidate-wire from $g_1$ is redundant, then the candidate-wire from $g_2$ must also redundant. Utilizing this relationship, we can accelerate the ATPG-based scheme dramatically, because many infeasible alternative wires can be discarded without calling the time consuming redundancy identification procedure.

In this paper, we propose a data structure, implication-tree, to store the implication relationship. We will then propose a new ATPG-based alternative-wiring algorithm, IBAW, the implication-tree based alternative-wiring algorithm. Differing from other methods, IBAW selects source node of alternative wire from the implication-tree, which makes IBAW skip many unnecessary (unsuccessful) redundancy checking. As a result, IBAW can run much faster than the original RAMBO.
IBAW is also a general-purposed alternative-wiring algorithm. Like RAMBO, it can also be applied in logic optimization. Experimental results show that for the purpose of finding alternative wires for target-wires, IBAW runs 3.6 times faster than the complete RAMBO algorithm. Besides, when logic optimization is concerned, the overall circuit area optimized by IBAW is a little better than that by the optimization-oriented RAMBO [6], while the runtime is just one-half of the latter.

2 Background and Definitions

A Boolean network is a Directed Acyclic Graph (DAG) whose nodes are primary inputs (PI), primary outputs (PO), and internal nodes (logic gates). A PI has only out-going edges, while a PO has only an in-coming edge. An internal node has at least two in-coming edges and one out-going edge, and is associated with a Boolean function. Note that inverters are not considered as an internal node here. Instead, it is considered as an edge’s polarity, which is defined below. For convenience, the gate types of internal nodes in this paper are assumed to be simple gates, i.e., AND, OR, NAND, or NOR.

A connection, or a wire, is an edge connecting two nodes. We denote a wire by a triplet, \( <S, D, P> \), where \( S \) is the source node, \( D \) the destination node, and \( P \) the polarity \((1 \text{ for inverted}, 0 \text{ for non-inverted})\). In some cases, we do not care about the polarity of a wire. In such cases, \( <S, D, P> \) can be simply denoted as \( S \rightarrow D \).

The logic value of a wire \( <S, D, P> \), represented by \( f(S, D, P) \), is the logic value at the end of the wire. Note that \( f(S, D, P) \) may be different from the value of \( S \). Suppose \( P = 1 \), for example, \( f(S, D, P) = 1 \) if \( S = 0 \).

A logic value is a controlling value of gate \( G \), \( cv(G) \), if and only if anyone of \( G \)'s inputs having the value will determine \( G \)'s output. The controlling value of AND (NAND) gate is 0, and that of OR (NOR) gate is 1. Conversely, the non-controlling value of a gate is 1 for AND (NAND) gate, and 0 for OR (NOR) gate.

For a node on the path from a non-PO node to a PO node, the input right on the path is called the on-input of the node, while the other inputs are called the side-inputs. In Fig. 1, for example, when the path from \( g_5 \) to \( O_1 \) is concerned, wire \( g_5' \rightarrow g_7 \) is the on-input of \( g_7 \), while \( g_4' \rightarrow g_7 \) is the side-input.

Node \( D \) is a dominator of another node \( G \) if and only if all the paths from \( G \) to any POs go through \( D \). Node \( D \) is a dominator of wire \( g_1 \rightarrow g_2 \), if \( D \) dominates \( g_2 \). The fanout-cone of node \( G \) is defined as the sub-circuit from \( G \) to all of the reachable PO nodes. The fanout-cone of wire \( g_1 \rightarrow g_2 \) is defined as the fanout-cone of \( g_2 \).

A connection is redundant if and only if the logic behavior of the network is independent of the connection, i.e. if it is removed from the network, the logic behavior of the network remains unchanged. The redundancy of a wire can be identified by ATPG technique. Let \( w_i(s-a-x) \) \((x = 0 \text{ or } 1)\) denote the stuck-at-\( x \) fault on wire \( w_i \). If \( w_i \) is an input of an AND (NAND) gate, it is redundant if and only if \( w_i(s-a-1) \) is untestable. If \( w_i \) is an input of an OR (NOR) gate, it is redundant if and only if \( w_i(s-a-0) \) is untestable.

For a target-wire \( w_i \), a candidate alternative wire \( <g_1, g_2, p> \) is a virtual connection from \( g_1 \) to \( g_2 \) with polarity \( p \), which makes \( w_i \) redundant. However, it can be added into the network only if it is redundant. Otherwise, it has to be discarded.

In Fig. 1, for example, let wire \( w_i = g_3 \rightarrow g_7 \) be the target-wire, which is irredundant. During the redundancy checking process for \( w_i \), \( s-a-1 \), \( g_5 \) has a mandatory assignment of 0, which is the controlling value of \( g_5 \). Hence, the candidate wire \( w_a = <g_5, g_6 \), 0> will make \( w_i \) redundant. After redundancy identification for \( w_a \), we see that \( g_5 \) is redundant. Therefore, \( w_j \) is a feasible alternative wire for \( w_i \).

Mandatory-assignment and logic implication is the center ATPG procedure exercised. There are two kinds of mandatory-assignments. One is for path sensitization, another is for fault driving.

By Path sensitization, the side-inputs of the dominators of the target-wire are set to be non-controlling value, so that the signal on the target-wire can be observed. In some papers, path sensitization is also called observability mandatory-assignment.

By fault driving, the source node of the target-wire is set to be the logic value that results in the destination node to have different values when the network is in good and faulty status.

In Fig. 1, for example, to check if \( w_i = g_3 \rightarrow g_7 \) is redundant, we need to check if \( w_i \) \((s-a-1) \) is untestable. To test this fault, the path sensitization requires \((g_5 = 1, g_4 = 0, h = 1)\), while the fault driving requires \( g_5 = 0 \).

A node with a determined logic value \((0 \text{ or } 1)\) is called a determined node. Some determined nodes result in other nodes to have some determined logic values. This process is called logic implication. For example, for the AND gate shown in Fig 2, \( a = 0 \) implies \( f = 0; f = 1 \) implies \( a = 1 \) and \( b = 1 \).

The implication process should be taken throughout the network for good and faulty status of the target-wire. Note that only the nodes in the output-cone of the target-wire may have different logic values during the mandatory assignment and logic implication process in good and faulty status. However, in an alternative wiring logic transformation system, the source node of the candidate alternative wire could not be in the output-cone of the wire \([6]\), and the selection of the destination node can be done without the
faulty-status process. Therefore, we just use the good-status
mandatory assignment and logic implication in the following
sections.

During the logic implication process, a node’s logic value
may be inconsistent with another node’s logic value. This is
called logic conflict. In Fig 2, for example, if two conditions
require that \( f = 1 \) and \( a = 0 \) respectively, then a logic conflict
occurs. If there is a logic conflict during the test generation
process for a fault, the fault is untestable.

The above logic implication process is called direct logic
implication. The other implication methods, such as recursive
learning [10], are more complex and powerful. However,
direct logic implication is able to identify most redundant
faults with a high speed. Therefore, in this paper, we only use
mandatory assignment and direct logic implication.

3 Implication Relationship and Implication-tree

In a network after mandatory assignment and logic
implication process, many nodes have a determined logic
value. We define an implication relationship between two
determined nodes as following.

Definition 1 (Implication relationship between two
determined nodes): Suppose \( g_0 \) and \( g_1 \) are two determined
nodes in the network under consideration, \( g_0 = b_0 \) and \( g_1 = b_1 \).
If \( g_0 = b_0 \) results in \( g_1 = b_1 \), then we say \( g_0 = b_0 \) implies \( g_1 = b_1 \),
denoted as \( g_0 \Rightarrow b_0 \Rightarrow g_1 = b_1 \); or for brevity, \( g_0 \) implies \( g_1 \),
denoted as \( g_0 \Rightarrow g_1 \).

Obviously, the implication relationship between
determined nodes possesses transitivity property, i.e., if \( g_0 \Rightarrow g_1 \), \( g_1 \Rightarrow g_2 \), then \( g_0 \Rightarrow g_2 \). We say \( g_0 \) directly implies \( g_1 \) if \( g_0 \)
implies \( g_1 \) without through any transition.

In Fig 1, for example, \( g_5 \) directly implies \( g_2 \), \( d \), and \( g_6 \),
while \( g_1 \) and \( g_4 \) through transition.

Notice that the implication relationship defined above is
different from the logic implication process, although the
former is based on the results of the latter. In Fig 1, for
example, the logic implication process on node \( g_7 \) is that \( g_7 = 0 \)
and \( g_6 = 0 \) result in \( g_7 = 0 \). However, according to definition 1,
we say \( g_7 \) directly implies both \( g_7 = 0 \) and \( g_6 = 0 \).

Theorem 1: Given two determined nodes \( g_0 \) and \( g_1 \), \( g_0 = b_0 \)
and \( g_1 = b_1 \). Suppose \( g_0 \Rightarrow g_1 \). Let \( <g_0, D, p_i> \) \( (i = 0, 1) \) be
two candidate wires. \( D \) is an internal node whose controlling
value is \( d \). \( p_i \) is the correspondent polarity that makes \( \beta(g_0, D, p_i) = d \).
If \( <g_0, D, p_i> \) is irredundant, then \( <g_1, D, p_i> \) is also
irredundant.

Proof. Firstly, as the two candidates are connected to the
same node, the fault propagation paths for the two candidates
are the same. Hence, the path sensitization mandatory
assignment is the same for the two candidates.

Secondly, let us consider the fault driving assignment when
the redundancy checking is carried out for wire \( <g_0, D, p_i> \) \( (i = 0, 1) \). Given \( cv(D) = d \), the fault that may cause \( <g_0, D, p_i> \)
redundant is \( (s-a-d') \) \( (d' \) is the complement of \( d \). Therefore
the fault driving assignment on \( <g_0, D, p_i> \) requires \( \beta(g_0, D, p_i) = d \). Recall that when \( g_i = b_i \) is given, \( p_i \)’s value is defined to
make \( \beta(g_0, D, p_i) = d \). Hence, the above fault driving
assignment equals to \( g_i = b_i \). As \( g_0 \) implies \( g_1 \), after
the mandatory assignment and logic implication for
\( <g_0, D, p_i> \) \( (s-a-d') \), both \( g_0 \) and \( g_1 \) become determined, i.e.
\( g_0 = b_0 \) and \( g_1 = b_1 \). While for
\( <g_1, D, p_i> \) \( (s-a-d') \), only \( g_1 \)
becomes determined, i.e. \( g_1 = b_1 \) and \( g_0 = X \) (\( X \) is don’t care).
Obviously, the former condition is stricter than the latter
condition. As all the other conditions are the same for both
faults, if the mandatory assignment and logic implication for
\( <g_0, D, p_i> \) \( (s-a-d') \) causes no logic conflict, so does for \( <g_1, D, p_i> \) \( (s-a-d') \). Therefore, if \( <g_0, D, p_i> \) is irredundant, then
\( <g_1, D, p_i> \) is also irredundant. QED

Observation tells us that most candidate alternative wires
are not redundant. Hence, utilizing theorem 1, many
irredundant candidate wires can be easily kicked out without
calling the time-consuming ATPG procedure. Thus the whole
logic transformation process can be much faster than before.

To store the implication relationship, we propose an
implication-tree as following.

Definition 2 (Implication-tree): Given a determined-node
set \( V_1 = \{v_1, v_2, ..., v_k\} \), an implication-tree is a tree whose
vertex set is \( V = \{v_0, v_1, v_2, ..., v_k\} \), where \( v_0 \) is root, and it does
not correspond to any node in the network. The sons of
\( v_k \) are the determined nodes that are not implied by any other
nodes. A leaf vertex corresponds to a node in the network that
does not imply any other nodes. Other vertices are called
internal vertices. There are two kinds of edges in an
implication-tree, father-son edge and brother edge. The
father-son edge, denoted by a vertical solid line, links a father
and its first son. The brother edge, denoted by a horizontal
dotted line, links the two close brothers. Linked by brother
dges, a father’s sons form a son array. An internal vertex
directly implies all of its sons. All the vertices except \( v_k \) have
a logic value marked beside the vertex. Besides, we restrict
that every vertex in the tree has at most one father. If two or
more nodes imply a vertex, we just keep the vertex as a son
of one of them.

For example, Fig 3 gives an implication-tree corresponding
to the results shown in Fig 1. The target-wire is \( w_i = g_5 \rightarrow g_6 \).
Let the determined-node set under consideration be \( V_1 = \{a, f, d, g_1, g_2, g_3, g_4, g_5\} \). Then, the vertex set of the implication-
tree is \( V' = \{v_0, a, f, d, g_1, g_2, g_3, g_4, g_8\} \), where \( v_8 \) is the root.
\( g_1 \) and \( g_8 \) are \( v_0 \)’s son, which are not implied by any other
nodes. The edge from \( g_3 \) to \( a \) is a father-son edge, with \( a \)
being the first son of \( g_3 \). The edge from \( a \) to \( f \) is a brother edge, which means that \( a \) and \( f \) are two close brothers. Both \( a \)
and \( f \) are leaf vertices. And so on.
In the above example, \( g_6, g_7 \) and \( g_8 \) are excluded from \( V_{\text{f}} \), because according to [6], the connections from the nodes in the fanout-cone of the target-wire can not be a feasible alternative wire. Therefore we only use the determined nodes outside the output-cone of the target-wire to establish an implication-tree.

After the mandatory assignment and logic implication for a target-wire, the correspondent implication-tree can be established in the following two steps: (Let the node under consideration outside of the output-cone of the target-wire)

Step 1) For each determined node, save the implication relationship between its inputs and the output. For example, let \( i_1 \) be an input of node \( n \). Suppose \( n \) implies \( i_1 \) that has no fathers until now, then \( i_1 \) is put into the son-array of node \( n \), while \( n \) is set as \( i_1 \)'s father.

Step 2) Put all the determined nodes that have no fathers into the son-array of \( v_k \), and set \( v_k \) as their father.

After these two steps, the tree rooted by \( v_k \) is the implication-tree we want.

4 Selection of Alternative wire Based on Implication-Tree

A candidate alternative wire is required to make the target-wire redundant. On the other hand, it should be redundant after it is added into the network. Selecting a candidate alternative wire can be divided into two steps: (i) select a destination node, (ii) select a source node. In [6], how to select a destination node of a candidate wire is discussed in details, which can be concluded briefly as that (1) the destination node should be a dominator of the target-wire. (2) The logic value of the on-input of the dominator at good status should be the node’s controlling value. Otherwise the candidate wire can’t be redundant after it is added into the network. In this paper, we also adopt this method to select destination nodes.

Then, the rest work is how to select a source node when a destination node is given. In RAMBO, this process is quite time-consuming. However, with implication-tree, the process becomes much faster.

Given destination node \( D \), a node \( S \) is called a feasible source node if the candidate wire \( <S, D, P> \) is a feasible alternative wire. The following algorithm describes how to select a source node from the implication-tree.

Select-a-source-node-from-implication-tree \( (R_i) \) \n\( R_i \) is Success or Fail; */ (\( R_i \)=Success) means the node visited last time is a feasible source node. */

Begin

\( * P_i \) is a global variable that points to the last visited node. \( P_i \)'s initial value is Null. After the tree has been traversed, \( P_i \) is set as \( v_k \), the root of the implication-tree. */

\( * P_i \) is a global variable that points to the last visited node. \( P_i \)'s initial value is Null. After the tree has been traversed, \( P_i \) is set as \( v_k \), the root of the implication-tree. */

if \( (P_i = \text{Null}) \) \{ 
  \( P_i := v_k \)'s first son;
  return \( P_i \);
\} else if \( (R_i = \text{Success}) \) \{ 
  if \( (P_i \) has at least one son) \{ 
    \( P_i := P_i \)'s first son;
    return \( P_i \);
  \} else if \( (P_i \) has unconsidered brothers) \{ 
    \( P_i := P_i \)'s close brother that has not been considered;
    return \( P_i \);
  \} else \{ /* \( P_i \) has no sons and no unconsidered brothers. */

while \( (P_i \) has no unconsidered brothers && \( P_i \neq v_k) \) 
  \( P_i := P_i \)'s father;
if \( (P_i = v_k) \) 
  return NULL; /*No more nodes */
else \{ 
  \( P_i := P_i \)'s brother;
  return \( P_i \);
\} 
\} else /* if \( R_i \) = Fail. */
if \( (P_i \) has unconsidered brother) \{ 
  \( P_i := P_i \)'s close brother that has not been considered yet;
  return \( P_i \);
\} else /* \( P_i \) has no other brothers. */
while \( (P_i \) has no unconsidered brothers && \( P_i \neq v_k) \) 
  \( P_i := P_i \)'s father;
if \( (P_i = v_k) \) 
  return NULL; /*No more nodes */
else \{ 
  \( P_i := \) the brother of \( P_i \)'s father;
  return \( P_i \);
\} 
\} 
End

Every time, except the first time, before the procedure is called, it is required that the node selected last time has been checked if it is a feasible source node. If the node selected last time is feasible, the node-pointer is modified to its left over son. Otherwise, the node-pointer is modified to its close brother or its father’s brother, etc. In the end of the procedure, if the node-pointer points to a non-root vertex of the implication-tree, the vertex is returned. Otherwise, NULL is returned.

In the procedure, the parameter \( R_i \) is a flag that marks whether or not the node selected last time is a feasible source node. First time when the procedure is called, \( P_i \) is Null. So \( P_i \) is modified to the first son of the root \( v_k \), which is then returned. In the rest calls, \( R_i \) is checked first. If \( R_i \) is Fail, \( i.e. \) the node selected last time is a feasible source node, then its sons are also feasible, and \( P_i \) is modified to \( P_i \)'s brother or its father’s brother and so on. If \( R_i \) is Success, its son should be visited, and \( P_i \) is modified to \( P_i \)'s first son if it has any children, or to its next brother or its father’s next brother and so on if it has no children. On the other hand, if \( R_i \) is Fail, \( P_i \) is modified to its next brother or its father’s next brother and so on. If \( P_i \) points to a non-root vertex of the tree, the vertex is returned. Otherwise, NULL is returned, which means no more nodes can be selected.

Take the implication-tree in Fig 3 as an example. Suppose \( g_6 \) in Fig. 1 is selected as the destination node of the candidate wire. The above procedure is called for several times as following. (1) First time when the procedure is called, \( P_i = \text{NULL} \). So \( P_i \) is modified as \( v_k \)'s first son, \( g_3 \), which is then returned. (2) As \( g_3 \) is an infeasible source node, \( R_i = \text{Fail} \), and \( P_i \) is modified as \( g_3 \)'s next brother \( g_5 \), which is then returned. (3) As \( g_5 \) is a feasible node, \( R_i = \text{Success} \). So \( P_i \) is modified as \( g_5 \)'s first son, \( g_3 \), which is then returned. (4) As \( R_i = \text{Fail} \), \( P_i \) is modified as \( g_3 \)'s next brother, \( d \), which is then returned. (5) As \( R_i = \text{Fail} \), \( P_i \) is modified as \( v_k \). Then, NULL is returned, which means no more nodes can be selected. Finally, there is only one feasible node, \( g_5 \), in the network. The correspondent
feasible alternative wire is \(<g_5, g_6, 0>\). Note that our method needs only 4 trials, while the original RAMBO needs 8 trials.

5 IBAW Algorithm

The following pseudo-codes gives the frame of IBAW, the implication-tree based alternative-wiring logic synthesis algorithm.

\[ \text{IBAW}(net) \]
\[ \text{net is the network under consideration;} \]
\[ \text{Begin} \]
\[ \quad \text{for each node } (net, n_i) \]
\[ \quad \quad \text{for each fanout } (n_i, o_i) \{ \]
\[ \quad \quad \quad w_i := n_i \text{o}; \]
\[ \quad \quad \text{IBAW-transform } (net, w_i); \]
\[ \quad \} \]
\[ \quad \text{return Success; } \]
\[ \} \]
\[ v_b := \text{Generate-implication-tree}(net, w_i); \]
\[ v_b \text{ is the root of implication-tree. } \]
\[ \text{Put } w_i \text{'s dominators into } S_{\text{dom}}/ * S_{\text{dom}} \text{ is an array. } \]
\[ \text{Sort-dominator-array } (S_{\text{dom}}); \]
\[ \text{for } (j_1 = 0; j_1 < \text{length of } S_{\text{dom}}; j_1++) \{ \]
\[ \quad X := S_{\text{dom}}[j_1]; \]
\[ \quad D := \text{Insert-buffer-node-after } (X); \]
\[ \quad P_i := \text{NULL; } / * P_i \text{ is a global node pointer. } / * \]
\[ \quad S := \text{Select-a-source-node-from-implication-tree } (\text{Success}); \]
\[ \text{while } (S \neq v_b) \{ \]
\[ \quad \text{Determine } P; / * P \text{ is the polarity of the candidate alternate wire. } / * \]
\[ \quad w_i := n_i \text{i} / * S, D, P \text{ is the candidate alternative wire. } / * \]
\[ \quad \text{if } (w_i \text{ is redundant}) \{ / * w_i \text{ is a feasible alternate wire. } / * \]
\[ \quad \quad \text{Put } w_i \text{ into } \text{alt_array}; \]
\[ \quad \quad R_i := \text{Success}; \]
\[ \quad \text{else } R_i := \text{Fail}; \]
\[ \quad S := \text{Select-a-source-node-from-implication-tree } (R_i); \]
\[ \} \]
\[ \text{Delete-buffer-node } (D); \]
\[ \text{if } (\text{alt_array} \text{ is not empty}) \}
\[ \quad \text{break; } / * \text{break from the for loop. } / * \]
\[ \} \]
\[ \text{if } (\text{alt_array} \text{ is not empty}) \{ \]
\[ \quad \text{Choose a wire } w_i \text{ from } \text{alt_array}; \]
\[ \quad \text{Add } (net, w_i); / * \text{Add } w_i \text{ into net. } / * \]
\[ \quad \text{Remove } (net, w_i); / * \text{ Remove } w_i \text{ from net. } / * \]
\[ \quad \text{return Success; } \]
\[ \} \]
\[ \text{else return Fail; } \]
\[ \text{End} \]

At the beginning of the process, the redundancy identification for \(w_i\) is carried out. If \(w_i\) is redundant, it should be removed. Otherwise, an implication-tree is built for the results of the logic implication process. For each dominator \(X\), a temporary buffer node \(D\) is added right after \(X\), where \(D\) is assumed as the destination node of the candidate alternative wire. Note that a buffer node can be converted into an AND gate or OR gate according to the requirement when the candidate wire is introduced. Then the implication-tree is traversed to find a feasible source node \(S\), such that wire \(S \rightarrow D\) is a feasible alternative wire. Note that by now the candidate alternative wire has not been added into the network. There are 4 parameters in alt_array: \(S, X, P\) and \(T\), to denote a candidate alternative wire. \(S\) is the source node, \(X\) is a dominator with a temporary buffer node \(D\) added behind, \(P\) is the polarity of the candidate wire, and \(T\) is the gate type required for \(D\).

After the implication-tree has been traversed, the temporary buffer node is removed to keep the original circuit unchanged. If alt_array is not empty by now, i.e., the target-wire has at least one alternative wire, the other dominators will be ignored. Of course, if we want to find as many alternative wires as possible, line 24 and 25 can simply be removed.

At the end of the procedure, if alt_array is not empty, a candidate wire is chosen from the array to substitute the target-wire. ADD(net, \(w_i\)) is a procedure to add the alternative wire \(w_i\) into net. As mentioned above, we just use \(S, X, P\) and \(T\) to denote a candidate alternative wire. If \(X\) has only one fanout node whose gate type is \(T\), then the alternative wire is directly connected from \(S\) to \(X\)'s fanout node with polarity \(P\). Otherwise, a new gate \(D\) with type \(T\) is added right after \(X\) as the destination node of the alternative wire. Note that the possibility of the existence of alternative wires for the target-wire becomes higher by adding a new gate behind a dominator.

6 Heuristic to Accelerate IBAW

[7] gives the following theorem:

**Theorem 2:** In an irredundant circuit, any node that has an observability mandatory assignment obtained during the process of checking the redundancy for the target-wire can not be a feasible source node of the alternative wire.

The observability mandatory-assignment nodes are those nodes that are assigned a logic value when path sensitization is carried out. From theorem 1 and 2, the following corollary can be immediately obtained:

**Corollary 1:** In an irredundant circuit, any node that is implied by an observability mandatory-assignment node can not be a feasible source node of the alternative wire.

Using implication-tree, we can apply theorem 2 and corollary 1 easily. If the current node is an observability assignment node, then the node pointer simply jumps over its sons to point to its brother or father’s brother and so on.

It is worth to note that it is very time consuming to check whether the circuit is irredundant after each change has been made to the circuit. However, even in a circuit that may have some redundant wires, theorem 2 and corollary 1 can also be used as an approximate heuristic to accelerate IBAW. In [14], we have carried out experiments for 22 benchmark circuits, and the results show that only 0.43% of the total alternative
wires are from the observability assignment nodes. Therefore, the heuristic is applicable for the real circuits.

7 Some Considerations for Logic Optimization

For logic optimization purpose, we hope that the gate number can be decreased after the wire substitution is carried out. This can be achieved by carefully selecting the target-wire and the destination node of the alternative wire.

7.1 Selection of the Target-wire

Suppose the target-wire is \( w_i = g_{s \rightarrow g_d} \) where \( g_s \) is the source node, \( g_d \) is the destination node. To simplify the circuit, we hope that deleting the target-wire may cause at least one gate being removed. If one of the following three conditions is satisfied, then at least one gate can be removed.

1. \( g_s \) has only one fanout. In this case, \( g_s \) can be removed if \( w_i \) is removed.
2. \( g_s \) has only 2 inputs. In this case, \( g_s \) can be removed if \( w_i \) is removed.
3. \( g_s \) has two fanouts, but after one of them is removed, the other becomes redundant. In this case, \( g_s \) can also be removed.

Therefore, if \( g_s \) has two fanouts, at least one of them should be checked.

7.2 Selection of the destination node of the candidate alternative wire

In previous sections, we have mentioned that the destination node of the candidate alternative wire should be the dominator of the target-wire. Sometimes, we have to add a new dominator into the network to be the destination node of the alternative wire. However, for optimization purpose, we add a new gate only if there are no alternative wires for the target wire if we do not add a gate.

In IBAW-transform, we use Sort-dominator-array to sort the dominators. Firstly, the dominators are put into two arrays: array1 and array2. In array1, each dominator has only one fanout whose type is consistent with the requirement by the logic value of its on-inputs, so that no new gate is added. Other dominators are put in array2. Secondly, the dominators in each array are sorted according to their distance from the target-wire. The dominator with less distance from the target-wire is put in the front of the array. This is easy to be understood. If a wire is close to the target-wire, the relationship between them is relatively strong, and the chance of the wire’s being a feasible alternative wire is higher. Hence, the wire close to the target-wire should be checked first. Lastly, the two arrays are put back into \( S_d \), in which the elements in array1 is in front of that in array2.

8 Experimental Results

We have implemented IBAW on SunE450 Workstation Server. The circuits used in this paper are mapped by using “msu1.genlib” in SIS [1], which is a sub-set of “msu.genlib” in SIS that keeps all gates simple with maximum input count of 4.

The first experiment is to compare the speed of IBAW and RAMBO in finding alternative wires for target wires. The results are shown in Table 1, where C-IBAW is the complete IBAW, and C-RAMBO is the complete RAMBO. Both methods try to find as many alternative wires as possible for every wire in the circuit. We implement C-RAMBO by simulating the original RAMBO algorithm [2] which tries to find all possible alternative wires for all target-wires in the network. C-RAMBO differs a little from the original RAMBO [2] in that C-RAMBO can create a new node after a dominator when necessary. The target wires used by C-IBAW and C-RAMBO are exactly the same. The alternative wires found by both programs are the same, so we do not list them in the table.

Table 1 shows that the overall runtime of C-IBAW is only 1/3.6 of C-RAMBO. IBAW is especially powerful for larger circuits. For example, for des, the largest among the 22 circuits, the runtime of C-IBAW is 1/4 of that of C-RAMBO. For C3540, the ratio is 1/5.4.

We then apply IBAW for doing logic optimization. The circuits are first optimized by “script.boolean” of SIS, which uses Boolean method to do logic transformation. Then, the circuits are mapped using msu1.genlib. After that, IBAW1, IBAW2 and UCSB RAMBO [6] are applied to the new circuits, whose results are also mapped using “msu1.genlib” for comparison. Both IBAW1 and IBAW2 are optimization-oriented IBAW. The difference between them is that IBAW2 uses the heuristic described in Section 6, while IBAW1 does not. The UCSB RAMBO is obtained from UC Santa Barbara, in which the optimization-oriented part is used. The comparison among them is shown in Table 2, where “area” means the total cell size of the corresponding circuit under library “msu1.genlib”.

As shown in Table 2, overall, RAMBO improves the results obtained by “script.boolean” by 3.6%, while both IBAW1 and IBAW2 improve by 4.3%. However, the runtime of IBAW1 and IBAW2 are only 59.1% and 49.1% of RAMBO. Hence, IBAW is similarly powerful but faster in doing logic optimization.

9 Conclusion

In this paper, we proposed a data structure, implication-tree, to store the implication relationship of the determined nodes, which was built when the redundancy check was carried out for the target wire. We then proposed an implication-tree based alternative-wiring logic transformation algorithm, IBAW.

IBAW differs from other ATPG-based algorithms in that it selects source node of alternative wires from the implication-
tree, which helps IBAW trim out many useless redundancy checking. Hence, the speed is much improved.

Experimental results show that the runtime of IBAW is only 1/3 of the complete RAMBO in finding alternative wires for target-wires. With our heuristics, IBAW is able to reduce the circuit area that is optimized by “script.boolean” of SIS for 4.3% in average, which is a little better than that obtained by UCSB RAMBO codes [6], while the runtime of the former is only one-half of the latter. Basically, IBAW is a fast and general-purpose-alternative-wiring logic transformation tool, which should also be useful to many other known EDA applications.

Table 2: Comparison of three methods

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Area</th>
<th>CPU (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sxp1</td>
<td>2512</td>
<td>4.49</td>
</tr>
<tr>
<td>Sypm-hdl</td>
<td>2532</td>
<td>2.19</td>
</tr>
<tr>
<td>C75A0</td>
<td>1636</td>
<td>501.22</td>
</tr>
<tr>
<td>C7515</td>
<td>1636</td>
<td>302.95</td>
</tr>
<tr>
<td>C7528</td>
<td>1636</td>
<td>712.26</td>
</tr>
<tr>
<td>C7552</td>
<td>1636</td>
<td>480.68</td>
</tr>
<tr>
<td>alu2</td>
<td>1576</td>
<td>93.90</td>
</tr>
<tr>
<td>alu4</td>
<td>1576</td>
<td>450.03</td>
</tr>
<tr>
<td>apex6</td>
<td>1636</td>
<td>40.36</td>
</tr>
<tr>
<td>b9_n2</td>
<td>1636</td>
<td>2.68</td>
</tr>
<tr>
<td>comp</td>
<td>1636</td>
<td>9.01</td>
</tr>
<tr>
<td>des</td>
<td>1636</td>
<td>1342.34</td>
</tr>
<tr>
<td>duke2</td>
<td>1636</td>
<td>82.69</td>
</tr>
<tr>
<td>f51m</td>
<td>1636</td>
<td>5.73</td>
</tr>
<tr>
<td>miscx3</td>
<td>1636</td>
<td>102.03</td>
</tr>
<tr>
<td>my_adder</td>
<td>1636</td>
<td>2.85</td>
</tr>
<tr>
<td>pcbl08</td>
<td>1636</td>
<td>1.72</td>
</tr>
<tr>
<td>rnt</td>
<td>1636</td>
<td>9.01</td>
</tr>
<tr>
<td>sa02-hdl</td>
<td>1636</td>
<td>16.90</td>
</tr>
<tr>
<td>term1</td>
<td>1636</td>
<td>9.56</td>
</tr>
<tr>
<td>ttk</td>
<td>1636</td>
<td>9.30</td>
</tr>
<tr>
<td>x3</td>
<td>1636</td>
<td>30.75</td>
</tr>
</tbody>
</table>

| Total       | 372408| 4143.40 |
| Ratio       | 1     | 19.4%   |

References