Multi-Level Logic Minimization based on Multi-Signal Implications

Masayuki Yuguchi Yuichi Nakamura Kazutoshi Wakabayashi Tomoyuki Fujita
C&C Research Laboratories, NEC Corporation
Miyazaki, Miyamae-ku, Kawasaki 216, Japan

Abstract— **This paper presents a novel method for logic minimization in large-scale multi-level networks. It accomplishes its great reductions on the basis of multi-signal implications and the relationships among these implications. Both are handled on a transitive implication graph, proposed in this paper, which realizes high-speed, high-quality minimization. This proposed method holds great promise for the achievement of an interactive logic design environment for large-scale networks.**

I. Introduction

With recent rapid progress in VLSI computer design technology, highly efficient algorithms for logic minimization in large-scale multi-level networks have become increasingly important. While the potential for using a network's internal don't care set to transform intermediate logic functions in such a way as to reduce network size has long been apparent, use of all of the elements of a set has proven too expensive, and minimization techniques employing subsets have been difficult to perfect [1, 2].

The Transduction method [3] is one example of a subset approach. Permissible functions (intimately related to observability don't care (ODC) sets) enable network-size reductions. (RENO [4] is an example of a good application of Transduction) In large-scale networks, however, permissible functions are too complex for practical handling.

Another example of an internal don't care subset approach is the Global Flow method [5, 6]. Global Flow collects underlying implications among signals in a network and, with these implications, iterates a connection/redundancy-removal transformation. The collected implications are, in fact, satisfiability don't care (SDC) subsets. While Global Flow can be used in the minimization of large-scale networks, its minimization capacity is insufficient: each transformation uses only the implications from a single signal. That is to say, only very limited elements of the internal don't care subset are used at any one time for transforming intermediate logic functions, which makes it difficult to achieve dramatic size reductions.

32nd ACM/IEEE Design Automation Conference ®

In this paper, we propose a method for high-speed, high-quality minimization in large-scale networks, which applies an entirely new approach to the use of implications: it uses the implications from multiple signals for each transformation, and, most particularly, it makes use of the relationships among those implications. (We refer here to implications from multiple signals as multi-signal implications) The fact that individual transformations are performed on the basis of multi-signal implications and that these transformations take into consideration the relationships among those implications gives our method the capacity to reduce the size of large-scale networks drastically.

In the sections which follow, we first present a concept of the transitive implication graph, which plays a key role to our method. This kind of graph efficiently represents implications underlying a given network and facilitates the use of the relationships among these implications. Moreover, this graph can be generated for large-scale networks. Second, we introduce a new minimization method based on those relationships as they are represented in the transitive implication graph. Our method iterates a subnetwork-addition/redundancy-removal transformation. A subnetwork can be generated with multi-signal implications and can then be added to the original network, which will then contain a removable redundancy significantly larger than the added subnetwork. When this redundancy has been removed, total network size will be significantly reduced. Use of information (contained in the transitive implication graph) regarding the relationships among implications allows for simple reduction in the size of the subnetwork that will produce a given size redundancy. This characteristic is especially important to the high-quality minimization achieved with our method. Further, since with this method it is necessary only to determine the relationships among implications within individual subsets (i.e. subgraphs of the transitive implication graph), as opposed to having to determine all the relationships among all the implications, minimization can be accomplished at significantly increased speed.

II. Preliminaries

This paper represents networks in Boolean form [1]. A *Boolean network* is a directed acyclic graph, consisting of nodes and arcs. Each node i is either an *(primary) input node* ($i = 1, 2, \cdots, p$) or an *intermediate node* ($i = p+1, p+2, \cdots, p+q\ (=n)$). Each input node i ($i = 1, 2, \cdots, p$) is assigned with an *input variable* y_i. Each intermediate node i ($i = p+1, p+2, \cdots, p+q\ (=n)$) is assigned an *intermediate variable* y_i and an *intermediate logic function* F_i. Some intermediate nodes are designated as the primary output nodes of a network. An arc from

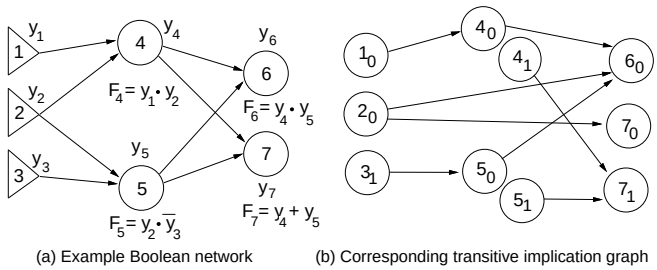

Figure 1: Transitive implication graph

node i to node j means that node j uses variable y_i in F_j. If there is an arc from node i to node j, node i is a *fan-in* of node j, and conversely, node j is a *fan-out* of node i. If there is a path from node i to node k, node i is a *transitive fan-in* of node k, and conversely, node k is a *transitive fan-out* of node i. In this paper, the terms *variable* and *signal* are used interchangeably, and each arc in a Boolean network is also called a *connection*.

A simple example of a Boolean network is shown in Fig.1(a). In this Boolean network, the input nodes are $\{1,2,3\}$ and the intermediate nodes are $\{4,5,6,7\}$. We assume that nodes $\{6,7\}$ are designated as the primary output nodes of the network. The fan-ins of node 4 are $\{1,2\}$ and the fan-outs of node 4 are $\{6,7\}$. The transitive fan-outs of node 1 are $\{4,6,7\}$ and the transitive fan-ins of node 6 are $\{1,2,3,4,5\}$.

In a Boolean network, an *implication* indicates that if $y_i = a$, then $y_j = b$ $(a,b \in \{0,1\})$, which is denoted by $y_i = a \Rightarrow y_j = b$ $(a,b \in \{0,1\})$.

III. Transitive Implication Graph

In this section, we present an efficient new representation of implications, the transitive implication graph, which can efficiently represent both implications and their relationships.

Before defining the structure of this graph, let us first define a transitive implication.

Definition 1 If an implication $y_i = a \Rightarrow y_j = b$ $(a,b \in \{0,1\})$ satisfies the following conditions, it is called a *transitive implication*:

1) A node j is a transitive fan-out of node i.
2) Either no transitive fan-in of node i has an implication to $y_i = a$ or at least one fan-out of node i does not have an implication from $y_i = a$.
3) For every node k on the paths from node i to node j, except for nodes i and j themselves, all fan-outs of node k have implications from $y_i = a$.
4) At least one fan-out of node j has no implication from $y_i = a$.

This transitive implication is denoted by $y_i = a \stackrel{*}{\Rightarrow} y_j = b$.

With this transitive implication, a *transitive implication graph* is defined as follows.

Definition 2 A transitive implication graph is a directed acyclic graph which consists of vertices and edges:

vertex: A vertex i_a represents $y_i = a$ $(a \in \{0,1\})$.

edge: An edge represents a transitive implication. For example, the edge from vertex i_a to vertex j_b represents $y_i = a \stackrel{*}{\Rightarrow} y_j = b$ $(a,b \in \{0,1\})$.

The edge from vertex i_a to vertex j_b $(a,b \in \{0,1\})$ is denoted by (i_a, j_b). If there is an edge (i_a, j_b), vertex j_b is called a *head* vertex of this edge and vertex i_a, and conversely, vertex i_a is called a *tail* vertex of this edge and vertex j_b. The set of all head vertices of vertex i_a is called a *head set* of vertex i_a.

Fig.1(b) shows the transitive implication graph which corresponds to the network in Fig.1(a). Vertex 2_0 indicates $y_2 = 0$. If $y_2 = 0$, then $y_4 = 0$ and $y_5 = 0$, and moreover, $y_6 = 0$ and $y_7 = 0$. These transitive implications are represented by the edges $(2_0, 6_0)$, $(2_0, 7_0)$, respectively, in this graph.

A transitive implication graph can be generated by searching through the paths from each node to transitive fan-outs of this node. This searching process requires $O(n^3)$ time complexity (n is the number of nodes in a network) in the worst case which is equivalent to the one in which a transitive closure of the network is found. However, this worst case hardly ever occurs, because only transitive implications are searched for. This search will be fast in most cases. Similarly, a graph can be generated for a large-scale network since the size of a transitive implication graph is in practice much smaller than that of a transitive closure of the network.

Our method uses the relationships among edges, each of which represents an implication, in a transitive implication graph. Specifically, our method handles an intersection of the head sets of multiple vertices. Such intersections of large size are very useful for our method. A transitive implication graph is defined with transitive implications so as to increase the number of edges from or to each vertex and, as a result, to generate large-size intersections.

There are some cases where, if one edge is used in a minimization approach, another edge represents a wrong implication. Therefore, we make a transitive implication graph in the following way: a transitive implication graph will ever include only one or the other of them, so that such a case never occurs.

IV. Conventional Approach

To help understand the method we will introduce in Section V, we first show a conventional approach to network-size reduction, one which is similar to Global Flow and which uses only for the information a transitive implication graph contains about implications, not for the other information it contains about the relationships among those implications.

First, let us consider the following theorem about signal connection.

Theorem 1 In a transitive implication graph, if there is an edge (i_a, j_b) $(a,b \in \{0,1\})$, then F_j can be replaced by

$$\begin{cases} y_i F_j & (\textbf{if } a=0 \textit{ and } b=0) \\ \overline{y_i} + F_j & (\textbf{if } a=0 \textit{ and } b=1) \\ \overline{y_i} F_j & (\textbf{if } a=1 \textit{ and } b=0) \\ y_i + F_j & (\textbf{if } a=1 \textit{ and } b=1) \end{cases}$$

Thus, node i becomes a fan-in of node j.

Proof: Similar to that of Theorem 3 in [6].

This connection (i,j) is called a *signal connection*. It is important to note that such signal connections may

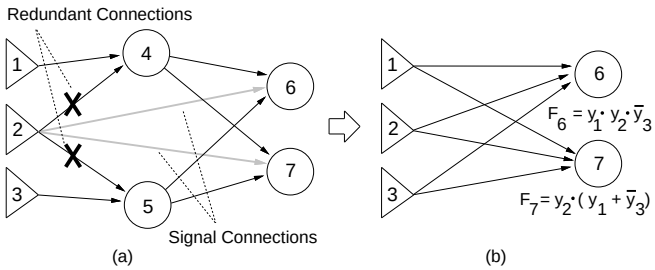

Figure 2: Example conventional approach

make some other connections redundant, and redundant connections can be detected with the following theorem.

Theorem 2 Suppose that there are edges $(i_a, j^0{}_{b_0}), (i_a, j^1{}_{b_1}), \cdots, (i_a, j^r{}_{b_r})$ $(a, b_0, b_1, \cdots, b_r \in \{0,1\}, r \geq 0)$ in a transitive implication graph, and that the signal connections for these edges have been made in accord with Theorem 1. A connection from node i to node j becomes redundant if every path from node j to the primary outputs that are reachable from node j includes at least one node from $\{j^0, j^1, \cdots, j^r\}$. In such a case, then, F_j can be replaced by F_{jy_i} (if $a = 0$), or $F_{j\overline{y_i}}$ (if $a = 1$). (F_{jy_i} is a cofactor of F_j with respect to y_i).

Proof: Similar to that of Theorem 4 in [6].

Fig.2(a) illustrates an application of a signal connection and redundancy removal transformation to the network originally introduced in Fig.1(a). Signal connections for the edges shown in Fig.1(b), $(2_0, 6_0)$ and $(2_0, 7_0)$, have been made in accord with Theorem 1. The connections from node 2 to node 4 and to node 5 become redundant, then, in accord with Theorem 2. For example, the stuck-at-1 fault on the connection from node 2 to node 4 can not be propagated to any primary outputs because the values of node 6 and node 7 are both 0. Thus, $F_4 = y_2 F_{4y_2} + \overline{y_2} F_{4\overline{y_2}} = 1 \cdot F_{4y_2} + \overline{1} \cdot F_{4\overline{y_2}} = F_{4y_2} = y_1$. The resulting network is shown in Fig.2(b). (This network is represented without any buffer or inverter.)

In our experience, however, a conventional approach based on signal connection and redundancy removal has very limited capacity to reduce network size. For example, let us consider Fig.3(a), which shows a part of a network. Each node has an AND logic function, and the network has 30 literals. Signal connection and redundancy removal for individual nodes can not reduce network size. Six signal connections (from node 1 to nodes 14 $\sim$ 19) are made for node 1, and two connections (from node 1 to nodes 6 and 8) will be removed. The resulting network, shown in Fig.3(b), has 34 literals, greater than the number in the original network.

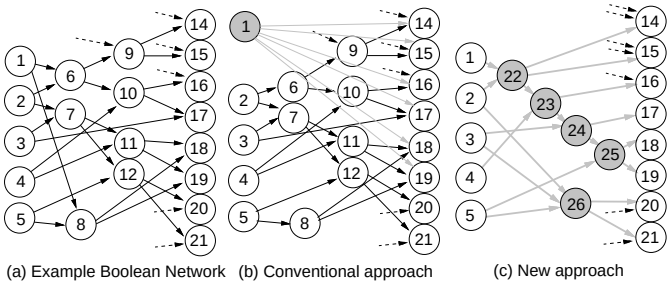

Figure 3: Conventional approach and new approach

V. Logic Minimization based on Multi-Signal Implications

We propose a new minimization method which can reduce network size drastically. It accomplishes its great reduction in network size on the basis of multi-signal implications, and, most particularly, on information regarding the relationships among those implications, and it achieves network-size reductions through the use of a transitive implication graph.

Let us first consider again here the example, shown in Fig.3(a). Our new approach performs a subnetwork-addition and redundancy removal with all implications from $\{y_1 = 0, y_2 = 0, y_3 = 0, y_4 = 0, y_5 = 0\}$. This subnetwork includes not only signal connections but also new nodes, like nodes 22 $\sim$ 26. The resulting network, shown in Fig.3(c), has 26 literals, smaller than the number of the original network.

Our proposed algorithm is shown in Fig.4. This algorithm iterates subnetwork-addition/redundancy-removal transformation. In Step 2, a subgraph is extracted from the transitive implication graph. This subgraph consists of the edges from multiple vertices and their tail and head vertices. These edges correspond to multi-signal implications. In Step 5, the subnetwork to be added is generated with this subgraph. If the subgraph has been previously transformed in Step 4, the subnetwork of smaller size can then be generated. It is important to note that this subgraph transformation utilizes the relationships among edges and allows for simple reduction in the size of the subnetwork which will produce a given redundancy size. In Step 6, this algorithm detects any redundancy which would be created in the original network if the subnetwork were to be added to the network. Such redundancy can be easily detected with a transitive implication graph. In Step 7, the algorithm examines whether the entire network size would be reduced by adding the subnetwork and removing the redundancy. In Step 9, this subnetwork is actually added into the network at this time and, in Step 10, the redundancy detected in Step 6 is removed. With subnetwork size reduction, redundancy removal can drastically reduce network size.

Logic Minimization Algorithm

1 Generate a transitive implication graph
2 Extract a subgraph from it
3 **if** (no more subgraph) end
4 Transform the subgraph
5 Generate a subnetwork with the transformed subgraph
6 Detect the redundancy created in the original network if the subnetwork were to be added to it
7 Determine whether network size would be reduced or not
8 **if** (the network size is not reduced) {
  Restore the subgraph
  Free the subnetwork and go to 2
  }
9 Add the subnetwork to the network
10 Remove the redundancy and go to 2

Figure 4: Logic minimization algorithm

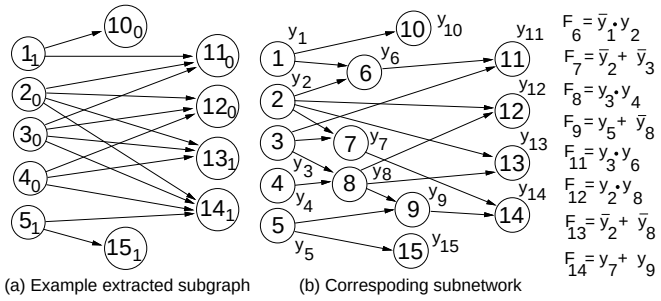

Figure 5: Subgraph extraction

In the following sections, we consider in great detail the processes of subgraph extraction, subgraph transformation, subnetwork addition, and redundancy removal.

A. Subgraph Extraction

For the subgraph transformation mentioned in Section V-B, it is desirable that each extracted subgraph contains an intersection of the head sets of multiple vertices. Our method extracts a subgraph for each vertex i_a in the transitive implication graph. For i_a, our method finds all vertices whose head set has an intersection with the head set of i_a (i_a itself is also included in these vertices) and extracts a subgraph which consists of these vertices, their head vertices, and the edges among them.

An example extracted subgraph is shown in Fig.5(a). It has been extracted for vertex 2_0. The vertices whose head sets have an intersection with the head set of vertex 2_0 are $\{1_1, 2_0, 3_0, 4_0, 5_1\}$. Fig.5(b) shows the part of the network which relates to the extracted subgraph. (The full network also has the other nodes, such as the transitive fan-ins of nodes $\{1, 2, 3, 4, 5, 10, 15\}$ and the transitive fan-outs of nodes $\{10, 11, 12, 13, 14, 15\}$.)

B. Subgraph Transformation

Let us first describe a case in which the subgraph is not transformed. If a subgraph were simply extracted (not transformed), a corresponding subnetwork consisting of only signal connections alone, can be added to the network in accord with Theorem 1. The redundancy created by adding this subnetwork can be detected in accord with Theorem 2. This process would be no more successful than the conventional approach described in Section IV. In our method, however, we first reduce the number of edges in the graph. This, as may be seen from Theorem 1, reduces the number of connections in the subnetwork to be added, which results in a reduction in the number of literals. Our heuristic subgraph transformation algorithm based on a search for the largest intersections, is shown in Fig.6.

Subgraph Transformation Algorithm

1 Find a set of vertices whose head sets have the largest size intersection. The vertices set is called an SRC set and the corresponding intersection is called an FRC set.
If the FRC set has only one vertex, exit
2 Generate a new vertex m_c ($c \in \{0,1\}$)
3 Remove all edges in $\{(i_a, j_b) | i_a \in SRC, j_b \in FRC\}$ from the subgraph
4 Generate edges (i_a, m_c) for all $i_a \in SRC$
5 Generate edges (m_c, j_b) for all $j_b \in FRC$ and go to 1

Figure 6: Subgraph transformation algorithm

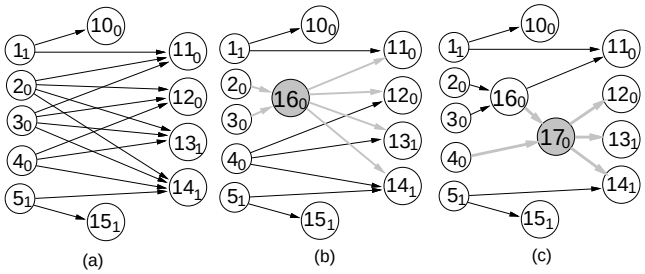

Figure 7: Subgraph transformation

Fig.7 shows how the transformation is applied to the subgraph in Fig.5(a). In Fig.7(a), the largest FRC set is $\{11_0, 12_0, 13_1, 14_1\}$ and the corresponding SRC set is $\{2_0, 3_0\}$. A new vertex 16_0 is made, and the edges among the SRC set and the FRC set are re-made as described in the algorithm. In Fig.7(b), the largest FRC set is then $\{12_0, 13_1, 14_1\}$ and the SRC set is $\{4_0, 16_0\}$. From this, the subgraph may be transformed to that of Fig.7(c).

This subgraph transformation requires $O(v^4)$ time complexity in the worst case. (v is the number of vertices in the full transitive implication graph) This is because, for each vertex, the entire subgraph may be searched for the largest size intersection. This search requires $O(v^3)$ time complexity. With subgraph transformation requiring a time complexity $O(v^4)$, the algorithm in Fig.4 requires $O(v^5)$ time complexity in the worst case. However, for a large-scale network, extracted subgraphs are hardly ever equivalent to the whole graph, and actual performance is much faster than the worst-case scenario in most instances.

C. Subnetwork Addition

Subnetworks consist of connections and nodes. Each connection corresponds to an edge in the subgraph, and each node corresponds to a new vertex generated in the subgraph transformation.

This subnetwork can be added to the original network. Let us consider a new vertex m_c ($c \in \{0,1\}$), its tail vertices ${i^0}_{a_0}, {i^1}_{a_1}, \cdots, {i^r}_{a_r}$ ($a_0, a_1, \cdots, a_r \in \{0,1\}$, $r \geq 0$), and its head vertices ${j^0}_{b_0}, {j^1}_{b_1}, \cdots, {j^s}_{b_s}$ ($b_0, b_1, \cdots, b_s \in \{0,1\}$, $s \geq 0$). With regard to the edges to vertex m_c, we can make a new node m ($F_m = 1$ (if $c = 0$) or $F_m = 0$ (if $c = 1$)), and the signal connections for the edges to vertex m_c can be made in accord with Theorem 1. With regard to the edge from vertex m_c to vertex ${j^t}_{b_t}$ ($0 \leq t \leq s$), let us consider the values of F_{j^t} and ${F_{j^t}}'(= y_m F_{j^t})$ in the case $c = 0$ *and* $b_t = 0$. If $y_m = 0$, then ${F_{j^t}}' = 0$, and at least one y_{i^k} ($0 \leq k \leq r$) of $y_{i^0}, y_{i^1}, \cdots, y_{i^r}$ is $y_{i^k} = a_k$. Since there ought to be the edge from ${i^k}_{a_k}$ to ${j^t}_0$ before transforming the subgraph with vertex m_0, there is the implication $y_{i^k} = a_k \Rightarrow y_{j^t} = 0$. Thus, $F_{j^t} = 0$. On the other hand, If $y_m = 1$, then ${F_{j^t}}' = 1 \cdot F_{j^t} = F_{j^t}$. Consequently, F_{j^t} can be replaced by ${F_{j^t}}'(= y_m F_{j^t})$, and the connection from node m to node j^t can be made.

Fig.8(a) illustrates the addition of a subnetwork generated from the transformed subgraph of Fig.7(c). The

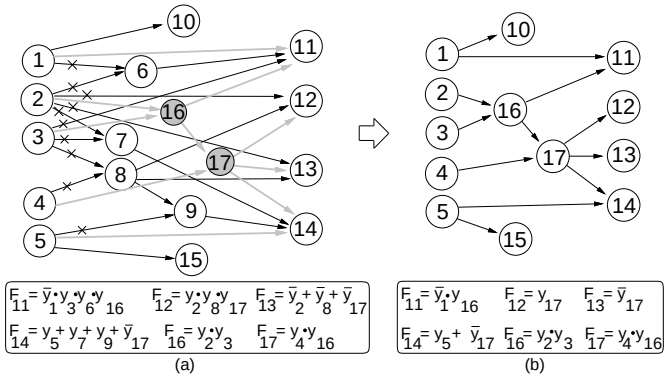

Figure 8: Subnetwork addition and redundancy removal

added subnetwork consists of these connections and the nodes which are shaded. The connections are signal connections for the edges of the transformed subgraph, and the nodes correspond to new vertices generated in the subgraph transformation.

D. Redundancy Removal for Subnetwork Addition

Redundancy is created in the original network when the subnetwork generated in Section V-C is added to the network. It is important to note that, whether the extracted subgraph has been transformed or not, the same redundancy can be created.

Let us consider the edges from vertex i_a, $(i_a, j^0{}_{b_0}), (i_a, j^1{}_{b_1}), \cdots, (i_a, j^r{}_{b_r})$, in the subgraph which is not transformed $(a, b_0, b_1, \cdots, b_r \in \{0,1\},\ r \geq 0)$. When the subgraph is transformed, each implication $y_i = a \Rightarrow y_{j^s} = b_s$ $(0 \leq s \leq r)$ is preserved. This is why such an implication can be represented by the edges $(i_a, k^0{}_{c_0}), (k^0{}_{c_0}, k^1{}_{c_1}), (k^1{}_{c_1}, k^2{}_{c_2}), \cdots, (k^t{}_{c_t}, j^s{}_{b_s})$ for the new vertices $k^0{}_{c_0}, k^1{}_{c_1}, k^2{}_{c_2}, \cdots, k^t{}_{c_t}$ $(c_0, c_1, \cdots, c_t \in \{0,1\},\ t \geq 0)$. Thus, the stuck-at-1 faults on the connections from node i, which are redundant when the subgraph is not transformed, are also redundant when the subgraph is transformed. This is because these faults can not be propagated to any primary outputs in both cases.

In Fig.8(a), when the subnetwork is added to the network, the redundant connections marked "×" can be detected and removed. The resulting network is shown in Fig.8(b).

VI. Experimental Results

We implemented our method on a SPARC Station 2 (28.5MIPS). All cpu times (seconds) were measured on this workstation.

Experimental results are shown in Table 1, including the resulting reductions in the number of literals (in factored form) for ISCAS '85,'89 benchmark large networks, from which buffers and inverters had been removed. We compared our proposed method (the column "Proposed method") with signal connection and redundancy removal (the column "Signal connection") which is closely similar to Global Flow. The column "ratio" shows the ratio of the network-size reduction. We can see that our proposed method produces about $1.5 \sim 16$ times larger reductions than signal connection and redundancy removal at much higher speed in most cases.

The simple iteration of our proposed method can give more reductions. The column "Iteration" shows the resulting reductions of three-time iteration.

Table 1: Experimental results

Circ.	Init.	Signal connection			Proposed method			Pro./Sig.		Iteration	
	literal	literal	ratio	time	literal	ratio	time	ratio	time	literal	time
C3540	2221	2142	3.60%	70.9s	1822	18.0%	66.0s	5.00	0.93	1810	191.2s
C5315	3528	3486	1.19%	43.9s	2847	19.3%	44.1s	16.2	1.00	2769	132.3s
C6288	4705	4225	10.2%	62.6s	3790	19.4%	58.7s	1.90	0.94	3762	267.4s
C7552	4740	4626	2.41%	154.2s	3789	20.1%	66.8s	8.34	0.43	3657	210.8s
s13207	5791	4991	13.8%	148.0s	4420	23.7%	125.8s	1.72	0.85	4137	353.0s
s15850	7299	6617	9.34%	245.1s	5954	18.4%	124.1s	1.97	0.51	5783	332.3s
s38417	18698	17625	5.74%	308.4s	16351	12.6%	258.4s	2.20	0.84	16232	792.0s
s38584	24348	22005	9.62%	1817.6s	20485	15.9%	791.3s	1.65	0.44	20144	1900.0s
clma	39927	38625	3.26%	212.5s	37999	4.83%	190.9s	1.48	0.90	37643	612.8s

VII. Conclusions

In this paper, we have presented the transitive implication graph as an efficient form for representing implications and the relationships among them, one which can be generated for large-scale networks. We have also proposed a new minimization method which makes use of multi-signal implications and the relationships among these implications in the graph to iterate subnetwork-addition and redundancy-removal transformation.

Our method achieved experimentally $1.5 \sim 16$ times larger reductions than the signal connection and redundancy removal approach for the large benchmark networks at much higher speed in most cases, and appears to hold great promise for the achievement of an interactive logic design environment for large-scale circuits which contain more than tens of thousands nodes.

References

[1] K.A.Bartlett, R.K.Brayton, G.D.Hachtel, R.M.Jacoby, C.R. Morrison, R.L.Rudell, A.Sangiovanni-Vincentelli, and A.R.Wang, "Multi-Level Logic Minimization Using Implicit Don't Cares", *IEEE Trans. on CAD*, Vol.7, No.6, pp.723-740, 1988.

[2] R.K.Brayton, R.L.Rudell, A.Sangiovanni-Vincentelli, "MIS: A Multiple-Level Logic Optimization", *IEEE Trans. on CAD*, Vol.6, No.6, pp.1062-1081, 1987.

[3] S.Muroga, Y.Kambayashi, H.C.Lai, J.N.Culliney, "The Transduction Method -Design of Logic Networks based on Permissible Functions", *IEEE Trans. on Comput.*, Vol.38, No.10, pp.1404-1424, 1989.

[4] K.C.Chen, M.Fujita, "Efficient Sum-To-One Subsets Algorithm for Logic Optimization", *IEEE DAC*, pp.443-448, 1992.

[5] C.L.Berman, L.H.Trevillyan, "Global Flow Optimization in Automatic Logic Synthesis", *IEEE Trans. on CAD*, Vol.1, No.5. pp.557-564. 1991.

[6] R.K.Brayton, E.M.Sentovich, F.Somenzi, "Don't Cares and Global Flow Analysis of Boolean Networks", *IEEE ICCAD*, pp.98-101, 1988.