Increasing Yield Using Partially-Programmable Circuits

Shigeru Yamashita

College of Information Science and Engineering
Ritsumeikan University
1-1-1 Noji Higashi, Kusatsu, Shiga 525-8577, Japan
ger@cs.ritsumei.ac.jp

Hiroaki Yoshida   Masahiro Fujita

VLSI Design and Education Center (VDEC)
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-8656, Japan
hiroaki@cad.t.u-tokyo.ac.jp   fujita@ee.t.u-tokyo.ac.jp

Abstract— This paper proposes to use a new circuit model called *Partially-Programmable Circuits* (PPCs) to increase the yield with very small overhead. PPCs are obtained from conventional logic circuits by replacing their sub-circuits with LUTs. If a connection in an PPC becomes redundant by changing the functionality of some LUTs, the connection is considered to be robust to defects because even if there are some defects at the connection, the circuit works properly by changing the functionality of some LUTs appropriately. To increase the number of such robust connections, we add some redundant connections to LUTs beforehand. We find such redundant connection by using functional flexibility represented by SPFDs and/or CSPFs. In other words, our proposed approach can increase the yield by only adding some redundant connections. From the result of our preliminary experiments, we consider our approach is promising.

I. Introduction

It has been considered that there will be a sharp increase in manufacturing defect levels in future electronic technologies [3, 11]. Thus, there has been a considerable interest in practical techniques to increase the yield of LSIs [4, 8, 6]. At the logic design level, a general method is to add *space* redundancy, e.g., Triple Modular Redundancy (TMR). If we are considering only manufacturing defects, another possible way to increase the yield is Double Modular Redundancy (DMR); we select a module without defect between the two identical modules after the LSI test. If we use FPGAs, we can have much more ways to bypass the defects after the LSI manufacture [5].

However, the above methods obviously have disadvantages: area overhead and/or performance degradation. In this paper, we propose a totally different approach to increase the yield with (possibly) lower overhead; we make a logic circuit from conventional logic circuits by replacing its sub-circuits with Look-Up Tables (LUTs) and Multiplexers (MUXs). Then, if we detect some defects (by the LSI test) in it, we reconfigure the functionality of some of LUTs and MUXs to bypass the defects. We call such circuits *Partially-Programmable Circuits* (PPCs) since some of their parts are programmable. Compared to the DMR scheme, our approach does not duplicate a target logic circuit entirely, but only replaces some parts of the circuit into LUTs. Therefore, the area and performance overhead could be small. Obviously we cannot use a PPC like an FPGA to reconfigure the functionality at any time; a PPC can only be reconfigured to bypass some (not all) defects, and thus the area/performance overhead would be lower than an FPGA.

In this paper, we suppose a single defect at a connection, e.g., stuck-at-0/1 fault. Then, we say a connection is *robust* to stuck-at-0 or stuck-at-1, if we can bypass the fault (by reconfiguring the circuit) to stuck-at-0 or stuck-at-1, respectively. If a connection can be made redundant by reconfiguring the functionality of some of LUTs, we call the connection simply robust (to any defect). This is because we can deal with any situation where the logic value at the connection becomes incorrect (unknown) value. Note also that we can bypass any defect at a logic gate if its fanout connections are all robust. In this paper, we mainly consider the ratio of robust connections in a designed circuit because the ratio is obviously related to the yield. The ratio is 100% for the above DMR scheme since one of the duplicated modules should work correctly if there is only a single fault in the circuit. On the other hand, since our scheme does not use as much overhead as the DMR scheme, we cannot expect such a high ratio; even so we can expect that our scheme achieve a reasonably high ratio with relatively low overhead as our preliminary experiments show.

To increase the ratio of robust connections in a designed PPC, our proposed method is to add some redundant connections in advance. To find appropriate redundant connections, our circuit transformation utilizes the notion of SPFD (Set of Pairs of Functions to be Distinguished) [12] and CSPFs (Compatible Sets of Permissible Functions) [7].

In the reminder of this paper, first we review the notion of SPFDs and CSPFs in Section II.. Next we provide an overview of our scheme in Section III. by using a motivational example. Then we propose a circuit transformation method in Section IV.. We then provide some preliminary experimental results in Section V. followed by our conclusions in Section VI..

II. Preliminaries

A. Basic Terminologies

We use the following notation in this paper.

- N_i represents a node in a network.
- $C_{[i,j]}$ represents the connection from N_i's output to one of N_j's inputs. If there is $C_{[i,j]}$, N_i is called a *fanin* of N_j, and N_j is called a *fanout* of N_i. Additionally, if there is a path from node N_i to node N_j, node N_i is called a *transitive fanin* of node N_j, and node N_j is called a *transitive fanout* of node N_i.
- f_i: represents the logic function at the output of N_i with respect to the primary inputs of the network. This is sometimes called the "global" function of the node.

We use the "functional flexibility" of logic functions:

- The condition in which an alternative function can replace the global function at the output of N_i is called the "*functional flexibility* of N_i."
- The condition in which an alternative function can replace the global function used for an input to N_j, which corresponds to $C_{[i,j]}$, is called the "*functional flexibility* of $C_{[i,j]}$."

Note that the functional flexibility of $C_{[i,j]}$ generally differs from that of $C_{[i,k]}$ although the global logic functions corresponds to $C_{[i,j]}$ and $C_{[i,k]}$ are the same as f_i.

To represent the functional flexibility for a conventional gate, we usually use an incompletely specified function whose ON-Set, OFF-Set and DC (Don't Care)-Set correspond to the input patterns, where the function must become 1, 0, and don't care. There are many calculation methods for such functional flexibility, e.g., *satisfiability don't cares (SDC)* [2], *observability don't cares (ODC)* [2], *compatible observability don't cares (CODC)* [9, 10], and *compatible sets of permissible functions (CSPFs)* [7]. In this paper, we use CSPFs among them, but our procedure can be modified to work with any one of methods to calculate functional flexibility.

To represent the functional flexibility for LUTs, we can have much more freedom because we can change the functionality of LUTs. To utilize such flexibility, we use *Sets of Pairs of Functions to be Distinguished (SPFDs)* [12].

The following notations are used for them:

- $SPFD_i$ ($CSPF_i$) is an SPFD (CSPF) that represents the functional flexibility of N_i, respectively.
- $SPFD_{[i,j]}$ ($CSPF_{[i,j]}$) is an SPFD (CSPF) that represents the functional flexibility of $C_{[i,j]}$, respectively.

Note that $SPFD_i$ and $SPFD_{[j,i]}$ implicitly assume that N_i is an LUT since we do not use SPFDs for input connections of conventional gates. $SPFD_i$ ($CSPF_i$) and $SPFD_{[i,j]}$ ($CSPF_{[i,j]}$) are sometimes referred to simply as "the SPFD (CSPF) of the node" and "the SPFD (CSPF) of the connection," respectively.

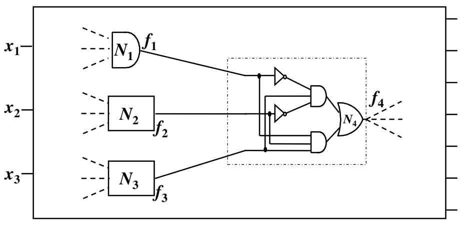
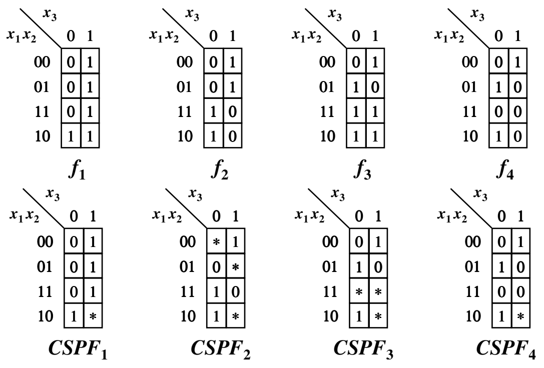


Fig. 1. A Boolean network to explain CSPFs/SPFDs.



x_3
x_1x_2 0 1
f_1: 00: 0 1; 01: 0 1; 11: 0 1; 10: 1 1
f_2: 00: 0 1; 01: 0 1; 11: 1 0; 10: 1 0
f_3: 00: 0 1; 01: 1 0; 11: 1 1; 10: 1 1
f_4: 00: 0 1; 01: 1 0; 11: 0 0; 10: 1 0
$CSPF_1$: 00: 0 1; 01: 0 1; 11: 0 1; 10: 1 *
$CSPF_2$: 00: * 1; 01: 0 *; 11: 1 0; 10: 1 *
$CSPF_3$: 00: 0 1; 01: 1 0; 11: * *; 10: 1 *
$CSPF_4$: 00: 0 1; 01: 1 0; 11: 0 0; 10: 1 *

Fig. 2. Functional flexibility by CSPFs.

In the following, we review the basic concept of CSPFs and SPFDs.

B. CSPF

Figure 2 shows an example for CSPFs [7] for the network shown in Figure 1. For simplicity, we suppose that the network has only three primary inputs x_1, x_2, and x_3. If f_1, f_2, and f_3 are shown as in Figure 2, internal logic of N_4 is f_4 is calculated by $(f_1 \cdot f_2 \cdot f_3 + \overline{f_1} \cdot \overline{f_2} \cdot f_3)$ as shown in Figure 2. Then, if the functional flexibility of N_4 is represented by $CSPF_4$ in Figure 2, the functional flexibilities of C_1, C_2, and C_3 are calculated as $CSPF_1, CSPF_2$, and $CSPF_3$ in Figure 2, respectively, by the method in the paper [7]. In the example, as long as f_1, f_2, and f_3 change within the flexibilities represented by $CSPF_1, CSPF_2$, and $CSPF_3$, it is guaranteed that f_4 changes within the flexibility represented by $CSPF_4$.

In this paper, we represent a CSPF as (f_{ON}, f_{OFF}) when its ON-Set and OFF-Set are f_{ON} and f_{OFF}, respectively.

C. SPFD

To discuss SPFDs, we need to define what it means **to distinguish a pair of functions**.

Definition 1 For two logic functions f and g, if $f(X)$ always becomes 1 for all primary input vectors X where $g(X)$ becomes 1, f is said to **include** g, also written $g \leq f$, or $g \Rightarrow f$. Note that "f includes g" $\Leftrightarrow g \cdot \overline{f} = 0$.

Definition 2 If either one of the following two conditions is satisfied, a function f is said to **distinguish** a pair of functions g_1 and g_2.

- f includes g_1, and $\overline{f}$ includes g_2. ($g_1 \leq f \leq \overline{g_2}$.)

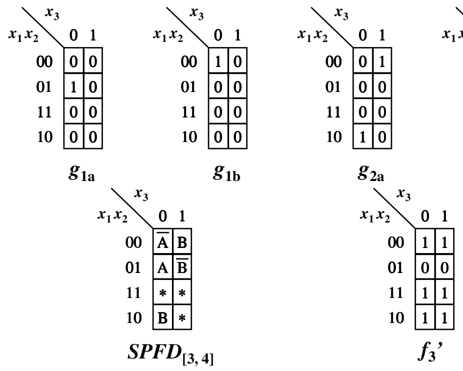


Fig. 3. Functional flexibility by an SPFD.

- f includes g_2, and $\overline{f}$ includes g_1. ($g_2 \leq f \leq \overline{g_1}$.)

In other words, f is said to distinguish g_1 and g_2 when one of ON-Set and OFF-Set of f includes g_1 and the other includes g_2.

Now, we can formalize an SPFD as follows.

Definition 3 SPFD (Set of Pairs of Functions to be Distinguished) is a set of pairs of functions that can be represented as $\{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\}$.

We also define the following terminology.

Definition 4 For an SPFD $= \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\}$, a function f is said to **satisfy the SPFD's condition** or simply **satisfy the SPFD** if f distinguishes g_{ia} and g_{ib} for all i.

Note that it is implicitly assumed that $(g_{ia} \cdot g_{ib} = 0)$ in the above SPFD; otherwise, no function can distinguish g_{ia} and g_{ib}.

In the example shown in Figures 1 and 2, suppose we implement the sub-circuit shown as the dotted rectangle into one LUT which is called N_4. Let also the functional flexibility of N_4 be the same as $CSPF_4$ in Figure 2. Then, the algorithm in [12] calculates $SPFD_{[3,4]}$ as $\{(g_{1a}, g_{1b}), (g_{2a}, g_{2b})\}$, where g_{1a}, g_{1b}, g_{2a} and g_{2b} are as shown in Figure 3.

The functions that satisfy $SPFD_{[3,4]}$ are functions whose intuitive truth tables are as follows.

- the values corresponding to A and $\overline{A}$ in the truth table shown as "$SPFD_{[3,4]}$" in Figure 3 must be different (one is 1 and the other is 0).
- the values corresponding to B and $\overline{B}$ in the truth table shown as "$SPFD_{[3,4]}$" in Figure 3 must be different (one is 1 and the other is 0).

Therefore, we can see that f_3' in Figure 3 satisfies $SPFD_{[3,4]}$. Since f_3' is not included in either $CSPF_{[3,4]}$ in Figure 2 or the simple negation of $CSPF_{[3,4]}$, we cannot replace f_3 with f_3' if we use $CSPF_{[3,4]}$ to represent the functional flexibility.

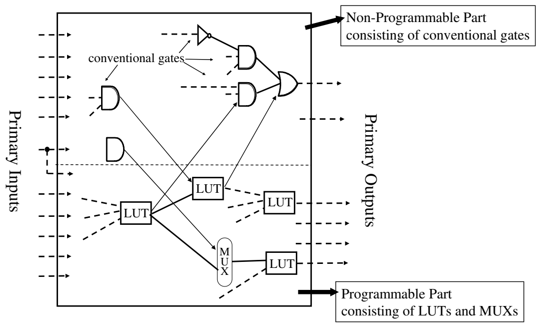

Non-Programmable Part consisting of conventional gates
conventional gates
Primary Inputs
Primary Outputs
LUT
LUT
LUT
LUT
MUX
Programmable Part consisting of LUTs and MUXs

Fig. 4. A partially-programmable circuit

D. Calculation of SPFDs and CSPFs

In our method, we calculate SPFDs and CSPFs by the original methods [12] and [7], respectively. They are calculated from the primary outputs toward the primary inputs as follows:

- For a primary output node N_i, set $SPFD_i$ as $\{(f_i, \overline{f_1})\}$, and $CSPF_i$ as $(f_i, \overline{f_1})$.
- For node N_i, we can calculate $SPFD_i$ ($CSPF_i$) by merging all the functional flexibility of the fanout connections of N_i. We refer the readers to the original papers [12] and [7] for the detail.
- From $SPFD_i$ ($CSPF_i$), we can calculate the SPFDs (CSPFs) of the fanin connections of N_i by the methods in [12] and [7], respectively.

III. Overview of our proposed scheme

A. A motivational example

In our proposed scheme, we implement a combinatorial logic circuit that has the following two parts: (1) Non-programmable part consisting of conventional gates, and (2) programmable part consisting of LUTs and MUXs. We call this kind of circuits specifically as *Partially-Programmable Circuits* (PPCs). Figure 4 shows an example. Some of the primary inputs may be connected to the both parts, and each primary output can be from either part. Also there may be connections from the programmable part to the non-programmable part, and/or vice versa.

A PPC can be viewed as a Boolean network (as shown in Figure 5) where we can have the following three types of nodes:

- Conventional logic gates, e.g., AND/OR/NOT.
- LUTs whose internal functionality can be reconfigured.
- MUXs whose select lines are controlled by programmable memory cells.

Suppose we can notice the following facts for the circuit (without dotted lines) in Figure 5. (We will explain how we can know that below.)

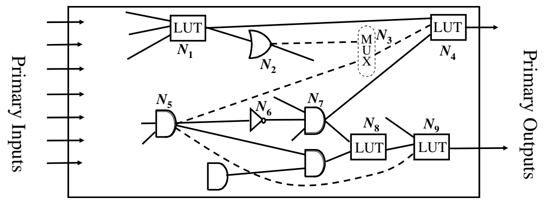


Fig. 5. A Boolean network to explain the model of PPCs.

- If we add a connection from N_5 to N_9, and then reconfigure the functionality of LUTs N_8 and N_9 appropriately, $C_{[7,8]}$ becomes redundant.
- If we add a connection from N_5 to N_4, and then reconfigure the functionality of LUT N_4 appropriately, $C_{[7,4]}$ becomes redundant.
- If we add a connection from N_2 to N_4, and then reconfigure the functionality of LUT N_4 appropriately, $C_{[1,4]}$ becomes redundant.

Therefore, in the above situation, we adds the dotted connections and MUX N_3 as shown in the figure. Then, we can bypass one of the following defects.

- If there is a defect at $C_{[7,8]}$, we can bypass the defect by reconfiguring the functionality of N_8 and N_9 appropriately.
- If there is a defect at $C_{[7,4]}$, we can bypass the defect by selecting the connection from N_5 by MUX N_3, and reconfiguring the functionality of N_4 appropriately.
- If there is a defect at any input of N_7 (or even a defect in N_7 itself), we can bypass the defect by the above two adjustments.
- If there is a defect at $C_{[1,4]}$, we can bypass the defect by selecting the connection from N_2 by MUX N_3, and reconfiguring the functionality of N_4 appropriately.

Accordingly, in the above circuit (with the added dotted lines), $C_{[7,8]}$, $C_{[7,4]}$, $C_{[1,4]}$ and the input connections to N_7, $C_{[5,6]}$ and $C_{[6,7]}$ as well as the added connections ($C_{[5,9]}$, $C_{[5,3]}$, $C_{[2,3]}$ and $C_{[3,4]}$) are all robust connections. A stuck-at-fault may occur at both $C_{[7,8]}$ and $C_{[7,4]}$ at the same time because they are connected. In the above example, we can also deal with the situation when there are defects at the two connections at the same time by reconfiguring the functionality of N_4, N_8 and N_9 appropriately.

Note that we do not need the two input connections of N_4 from N_2 and N_5 at the same time because we suppose a single defect in the circuit. Therefore, we do not increase the number of inputs of N_4 which results to an area penalty; we just put a MUX before N_4 to select a necessary input between the two inputs depending on a defect.

B. *Intuitive difference from DMR*

Note that all the robust connections in our scheme are not redundant at the same time, because we can bypass only a single fault at one of the robust connections at one time, i.e., we may not deal with multiple faults at the same time and thus they cannot be redundant at the same time. On the other hands, all the connections in one of the duplicated modules are all redundant at the same time in nature in the DMR scheme. This is a key observation of the difference between our scheme and the DMR scheme; the overhead of our scheme may be much lower because we consider only a single fault in a circuit. Even so we can expect the yield to increase greatly because the probability of having a single fault should be much larger than having multiple faults in standard LSI manufacturing. Thus, we expect our scheme may provide a better trade-off point between the area/performance overhead and the yield than the simple DMR scheme.

C. *Problem Formulation*

Now it should be clear what we want to do in this paper:
Our Problem.
Given a conventional combinatorial circuit, our problem is to design a PPC of the same functionality so that the ratio of connections that are robust to stuck-at-0 and/or stuck-at-1 is as high as possible.

IV. How to Increase robust connections

A. *Conversion between SPFDs and CSPFs*

Unlike the previous SPFD-based circuit transformation method [12], we need to calculate both SPFDs and CSPFs at the same time because our circuits contain LUTs and conventional gates. Thus we need to convert between SPFDs and CSPFs during the above calculation. This can be done easily as follows:

SPFD to CSPF. Suppose N_j is an LUT, and one of its fanin, N_i, is a conventional gate. Then, we can calculate the functional flexibility of $C_{[i,j]}$ as an SPFD, but in order to propagate it toward the inputs of N_i we need to convert it to a CSPF. Without loss of generality, let $SPFD_{[i,j]} = \{(g_{a_1}, g_{b_1}), (g_{a_2}, g_{b_2}), \cdots, (g_{a_m}, g_{b_m})\}$, where f_i includes g_{a_k} and $\overline{f_i}$ includes g_{b_k} for all k. Then, we convert the above SPFD to $CPFD_{[i,j]} = (f_{ON}, f_{OFF})$ where $f_{ON} = \sum_m g_{a_m}$ and $f_{OFF} = \sum_m g_{b_m}$. Note that any function included in $CPFD_{[i,j]}$ satisfies $SPFD_{[i,j]}$.

CSPF to SPFD. If N_j is a conventional gate, the functionality of each fanin connection of N_j is calculated as a CSPF. Then, if one of its fanin node, N_i, is an LUT, we need to convert $CPFD_{[i,j]} = (f_{ON}, f_{OFF})$ to an SPFD to proceed the above calculation of the functional flexibility in a circuit. This is easy; we just let $SPFD_{[i,j]} = \{(f_{ON}, f_{OFF})\}$.

B. *A proposed procedure for generating PPCs*

Now we are ready to explain our proposed procedure to increase robust connections in a given circuit.

In our procedure, we first generate an initial PPC from a given circuit as follows.

Generation of an initial PPC.

Step. 1 Map the given circuit into a $(k-l)$-input LUT networks. This can be done by any technology mapper. We assume to use k-input LUTs in the final implementation. This means that we may add at most l redundant connections to each LUTs in the later circuit transformation.

Step. 2 Select some of the LUTs by some heuristics. Then, restore the original conventional gates for unselected LUTs.

It is desirable that LUTs in a PPC have the following features:

Feature 1: An LUT should be useful to increase the number of robust connections, i.e., some connections should become redundant if we reconfigure the functionality of the LUT.

Feature 2: An LUT does not degrade the circuit performance too much.

Considering Feature 1, we consider it to be appropriate to select the following LUTs:

- An LUT which is near to the primary outputs.
- An LUT which is on a re-convergent point in the circuit.

Considering Feature 2, we may consider a heuristic that does not select an LUT which is on a critical path. In the following, we suppose that all the primary outputs are realized by LUTs in the initial circuit. We may consider another way of constructing initial circuits, which may be our future work.

After generating initial PPC (without MUXs), we add redundant connections to increase robust connections in the PPC by the procedure as shown in Figure 6.

In the procedure, we try to make each connection robust in the outer loop (lines 2 to 23). Suppose we are now considering $C_{[i,j]}$. If there is a defect at $C_{[i,j]}$, there are also some defects at connections that are connected to $C_{[i,j]}$ with high probability. For example, when we assume that $C_{[i,j]}$ is stuck-at-0, it is natural to assume that the other output connections from N_i also become stuck-at-0. However, for simplicity, we assume that only $C_{[i,j]}$ has a defect. Note that it is obviously easy to extend our procedure to treat the above error model where the connected connections have defects at the same time.

In the inner loop between lines 5 and 13, we select each LUT, N_m, in the fanout cone of $C_{[i,j]}$. Then, we try to find some redundant connections for the inputs of N_m so that it can satisfy its SPFDs even if the function realized at $C_{[i,j]}$ becomes unknown logic values. If all the LUTs in the fanout cone of $C_{[i,j]}$ can satisfy their SPFDs by adding some connections, $C_{[i,j]}$ is considered to be robust

```
1: Calculate the SPFDs of all the LUTs in the circuit.
2: for $C_{[i,j]}$ ← each connection in the initial circuit do
3:    Suppose the function realized at $C_{[i,j]}$ be unknown logic value,
      and recalculate logic functions toward the primary outputs by
      using this new logic function.
4:    Let $flag$ be $true$.
5:    for $N_m$ ← each LUT in the fanout cone of $C_{[i,j]}$ in the order
      from the primary inputs to the primary outputs do
6:       Find at most $l$ LUTs, $N_{k_1}, N_{k_2}, \cdots, N_{k_l}$, having the follow-
         ing conditions:
         • None of $N_{k_1}, N_{k_2}, \cdots, N_{k_l}$ is not a transitive fanout
           of $N_m$.
         • By adding redundant connections from
           $N_{k_1}, N_{k_2}, \cdots, N_{k_l}$ to $N_m$, $N_m$ can satisfy its
           SPFD calculated by the new derived logic functions.
7:       if there are no $l$ LUTs satisfying the above conditions then
8:          Let $flag$ be $false$, and break the loop between lines 5 to
            13 (i.e., go to line 14).
9:       else
10:         Add connections from $N_{k_1}, N_{k_2}, \cdots, N_{k_l}$ to $N_m$, and
            configure the functionality of $N_m$ so that it can satisfy
            its SPFD.
11:         Recalculate logic functions toward the primary outputs
            by using the new logic function of $N_m$.
12:      end if
13:   end for
14:   if $flag$ is $false$ then
15:      Conclude that $C_{[i,j]}$ cannot be made robust and remove all
         the added connections during the above loop between lines
         5 to 13.
16:   else
17:      Conclude that $C_{[i,j]}$ can be made robust.
18:   end if
19:   if $flag$ is $false$ then
20:      Suppose the function realized at $C_{[i,j]}$ be stuck-at-0, and
         do the same as the above (lines 3 to 18).
21:      Suppose the function realized at $C_{[i,j]}$ be stuck-at-1, and
         do the same as the above (lines 3 to 18).
22:   end if
23: end for
```

Fig. 6. A pseudo code for adding redundant connections.

because all the paths from $C_{[i,j]}$ to the primary outputs should go though at least one LUT. (Recall that we suppose that all the primary outputs are realized by LUTs in the initial circuit. However, we can easily generalize the proposed procedure to deal with a situation without the assumption.)

Between lines 3 to 18, we try to make the target connection robust. Unfortunately, we sometimes fail. Even in such a case, we further try to check whether the connection can be made robust to stuck-at-0 and stuck-at-1 at the lines 20 and 21, respectively. Since a robust connection (to unknown logic values) is also robust to stuck-at-0 and stuck-at-1, we perform lines 20 and 21 only when the target connection cannot be made robust between lines 3 to 18.

After adding redundant connections by the procedure, we further try to reduce the implementation cost of LUTs with many inputs by using MUXs as follows. In the above procedure, we add at most l redundant connections at one time. By doing so many times, the number of fanin connections may exceed k at some LUTs. Even so, by the nature of the above procedure, we need at most l additional inputs to such LUTs at the same time in order to make one connection robust. Thus, we add a MUX before the LUT to select necessary inputs between the added (redundant) inputs as explained in Section III..

We would like to stress that we can consider many other procedures to generate PPCs; the procedure in Figure 6 is just one of them.

V. Preliminary experiments

To evaluate how our method can increase the number of robust connections, we did the following preliminary experiments.

Step 1: We map some of MCNC benchmark circuits [13] into 4-input LUT networks. To do so, we use the synthesis and verification tool ABC [1].

Step 2: We generate initial PPCs such that only the LUTs at the primary outputs are selected. (The other LUTs are converted into the conventional gate sets.)

Step 3: We count the number of connections that are robust to stuck-at-0 (and stuck-at-1, respectively) in the above PPCs, and the total number of added connections.

We report the number of connections that can be robust to stuck-at-0 in the second column of Table I. The total number of added connections for making connections robust to stuck-at-0 is reported in the third column. Note that the added connections are also robust connections because they are redundant connections. Thus, we report the number of added connections in "Added." sub-column in "Rob." (meaning robust) column while "Orig."(meaning original) sub-column reports the number of original connections that can be made robust by our procedure. The fourth column denotes the numbers of connections that cannot be made robust to stuck-at-0 by our procedure. The columns 5 to 7 report the same numbers for stuck-at-1. From the table, we can observe that we indeed make some connections robust by just adding only some redundant connections.

VI. Conclusions

We propose to use a novel circuit model called *Partially-Programmable Circuits* (PPCs) to increase the yield of LSIs with low overhead compared to previous approaches. We expect that our proposed procedure increases the number of *robust* connections in PPCs, which apparently results in higher yield. Our preliminary experiments justify our expectation; we found that many connections indeed become redundant in benchmark circuits.

The presented procedure is just one example to utilize the idea of PPCs; our future work is obviously to consider and evaluate many other heuristics to utilize the notion of PPCs in order to increase the yield with low overhead.

TABLE I
Results: the number of connections that are robust to stuck-at-0/1.

Circuit	stuck-at-0			stuck-at-1		
	Rob.			Rob.		
Name	Orig.	Added.	Non-Rob.	Orig.	Added.	Non-Rob.
b9	92	51	25	91	46	26
lal	71	30	36	65	28	42
rot	421	161	218	411	172	228
vda	1251	153	221	1318	213	154
C1355	429	225	143	390	176	182
c8	70	23	33	62	25	41
x1	157	57	168	152	70	173
C880	185	32	144	212	57	117
cc	37	11	14	50	24	1
mux	73	3	8	74	6	7
C1908	626	37	66	599	36	93
example2	329	90	75	323	114	81
t481	1031	15	1085	1225	10	891
x3	504	174	141	489	191	156
C2670	710	59	176	704	66	182
alu2	586	13	25	582	20	29
f51m	113	10	17	116	14	14
pair	1174	280	456	1206	349	424
x4	257	123	211	412	223	56
C3540	1500	63	210	1462	52	248
alu4	1093	28	92	1070	25	115
comp	96	10	34	93	7	37
frg1	72	2	35	82	4	25
term1	195	36	59	194	31	60
apex6	591	86	106	589	98	108
too_large	472	15	256	453	9	275
apex7	171	52	36	166	39	41

References

[1] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 50905. http://www.eecs.berkeley.edu/ alanmi/abc/.

[2] 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 Implict Don't Cares. *IEEE Trans. on CAD*, 7(6):723–740, June 1988.

[3] G. Bourianoff. The future of nanocomputing. *IEEE Computer*, 36(8):44–53, 2003.

[4] V. K. R. Chiluvuri and I. Koren. Layout-synthesis techniques for yield enhancement. *IEEE Transactions on Semiconductor Manufacturing*, 8(2):178–187, May 1995.

[5] A. Doumar and H. Ito. Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: a survey. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 11(3):386–405, June 2003.

[6] Chen He and M.F. Jacome. Defect-aware high-level synthesis targeted at reconfigurable nanofabrics. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 26(5):817–833, May 2007.

[7] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Based on Permissible Functions. *IEEE Transactions on Computers*, 38(10):1404–1424, October 1989.

[8] A. Nardi and A. L. Sangiovanni-Vincentelli. Logic synthesis for manufacturability. *IEEE Design & Test of Computers*, 21(3):192–199, May-June 2004.

[9] H. Savoj and R. K. Brayton. The Use of Observability and External Don't Cares for Simplification of Multi-Level Networks. In *Proc. DAC*, pages 297–301, June 1990.

[10] H. Savoj, R. K. Brayton, and H. J. Touati. Extracting Local Don't Cares for Network Optimization. In *Proc. ICCAD*, pages 514–517, November 1991.

[11] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors 2007 Edition*. 2007.

[12] S. Yamashita, H. Sawada, and A. Nagoya. SPFD: A New Method to Express Functional Permissibilities. *IEEE Trans. on CAD*, 19(8):840–849, August 2000.

[13] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. *MCNC*, January 1991.