A GENERALIZED AND UNIFIED SPFD-BASED REWIRING TECHNIQUE

Pongstorn Maidee and Kia Bazargan

Department of Electrical and Computer Engineering, University of Minnesota
200 Union Street SE. Minneapolis, MN, 55455-0167, United States
email: pongstor,kia@ece.umn.edu

ABSTRACT

Traditionally, logic synthesis constrains the solution space of later design steps, such as physical design, because they are applied in sequence. Rewiring is a technique to restructure a circuit while maintaining its functionality. Since design properties and objectives can be considered during post-synthesis rewiring, it can help relieve constraints put forth by decisions made at earlier design steps. The extent of rewiring of a rewiring algorithm has a great impact on the success of the design flow. This paper presents a powerful rewiring technique that in addition to unifying all previously proposed Set-of-Pair-of-Functions-to-be-Distinguished based rewiring techniques, it can perform rewiring with more than one wire which increases our ability to circumvent poorly-decided design constraints. With this ability, the rewiring ability of using different numbers of wires is reported for the first time in this paper. It can be used for runtime/quality trade-off in any given rewiring application.

1. INTRODUCTION

Optimization is at the core of an integrated circuit design flow. Due to intractability of the majority of VLSI problems, the flow is organized in a sequential manner. Although interdependent, physical design is performed after logic synthesis. Thus, traditional physical design techniques are constrained by the circuit structure obtained from logic synthesis, resulting in sub-optimality. Since both logic synthesis and physical design are quite time consuming, reiterating over these two steps to improve solutions has found limited use. Rewiring has been proposed to restructure a circuit while its functionality is maintained, *ie.*, removing some wires in the circuit and adding other wires at different locations in the circuit with the condition that the global functions of primary outputs and flip-flop inputs are unchanged. As a result, rewiring techniques can be used to adapt the circuit structure to suit optimizations during physical design.

Rewiring techniques can be classified into automatic-test-pattern-generation (ATPG) and Set-of-Pair-of-Functions-to-be-Distinguished (SPFD) based techniques. ATPG-based rewiring relies heavily on the present node functions. Thus, its solution space is severely limited as a function provides minimal flexibility. SPFD was proposed to express a node's function flexibility [1]. Its application to rewiring has been shown to provide better results both in theory [2] and experiment [3]. SPFD-based rewiring can be used in many applications. For example, it has been used to reduce the number of Look-Up Tables used in implementing a circuit [4] and also shown to help reduce FPGA power consumption by 12% [5].

Rewiring ability is defined as a ratio of the number of wires that can be rewired to the number of total wires in a circuit. Thus, the rewiring algorithm with higher rewiring ability has more optimization capability. As a result, rewiring ability improvement is of great interest.

SPFD-based rewiring is composed of 2 steps: choosing the sources and targets for additional wires and checking if the circuit functionality is still maintained. The previously proposed checking mechanisms either restrict target nodes or limit the number of wires to be added or both. As a result, the rewiring ability is limited.

The contributions of this paper can be summarized as :

1. Provide necessary and sufficient conditions for generalized checking mechanism. The scheme unifies previous approaches into a single framework and is applicable for rewiring with more than one wire.

2. Show an efficient implementation of the scheme.

3. Report for the first time the rewiring ability of using different numbers of additional wires, which can be used for runtime/quality trade-offs in any rewiring application.

The paper is organized as follows. Section 2 summarizes notations used in this paper. Section 3 explains SPFD. The previous SPFD-based rewiring approaches are summarized in Section 4. Section 5 introduces our generalized checking scheme. An efficient implementation of the scheme is detailed in Section 6. Section 7 shows experimental results.

2. BASIC TERMINOLOGIES

A combinational circuit consists of nodes and directed edges between nodes. We use $wire(n_a, n_b)$ to call an edge from

node n_a to node n_b. The source and sink nodes of a wire, w, are denoted by $sr(w)$ and $sk(w)$, respectively. Thus, $sr(wire(n_a, n_b)) = n_a$. If there exists $wire(n_a, n_b)$, we say that n_a is a fanin node of n_b and n_b is a fanout node of n_a. Similarly, we call $wire(n_a, n_b)$ a fanin wire of n_b and fanout wire of n_a. $FI(n)$ denotes both fanin nodes or wires of Node n. $FO(n)$ is similarly defined as fanout nodes or wires. We use $TFI(n)$ and $TFO(n)$ to represent sets of transitive fanin and fanout nodes of node n, respectively. For example, if there exists $wire(n_a, n_b)$ and $wire(n_b, n_c)$, $n_a, n_b \in TFI(n_c)$ and $n_b, n_c \in TFO(n_a)$. Nodes with no fanout and no fanin nodes are called primary output and input nodes, PO, PI, respectively. As a circuit is a directed acyclic graph, nodes can be organized into levels using topological sort. The level of node n is denoted by $L(n)$.

Each node, n, implements a one output binary function, $f(n)$. Iteratively composing $f(n)$ using functions of nodes in $TFI(n)$, we obtain a global function of node n, $g(n)$. The global function of n before rewiring is denoted as $g_{org}(n)$.

3. SPFD

A binary function of n variables defines its ON and OFF sets of n-tuple binary numbers. Since each variable can take either 0 or 1, the total number of elements in both sets is 2^n. For any given binary function, we can draw a graph where each node represents one n-tuple number. Nodes in the graph can be organized into two groups, one for each set. To represent the fact that nodes from different sides will be evaluated into different values, an edge is added between any pair of nodes from different groups. The resulting graph is a bipartite graph as nodes in different groups are separated by an edge and there is no edge between nodes in the same group. From this construction, we can see that there is a one-to-one mapping between a set of n-input binary functions and a set of bipartite graphs with 2^n nodes. If there are some **don't care** tuples, the corresponding function is called incompletely specified function (ISF). These **don't care** nodes can be placed on either groups. However, different placement of them will represent different functions, but each of them implements the given ISF.

In 1996, Yamashita, et al., cleverly expanded this idea [1] by generalizing an edge to the following definitions.

Definition 3.1. *[1] For any boolean functions, f and g, let* $FX = x \mid f(x) = 1, GX = x \mid g(x) = 1$, *where x is the primary input vectors. If* $GX \subseteq FX$, *f* ***includes*** *g, written as* $g \leq f$ *or* $g \rightarrow f$, *which is equivalent to* $g \cdot \overline{f} = 0$.

Definition 3.2. *[6] A function f is said to* ***distinguish*** *a pair of functions g and h if either* $g \leq f \leq \overline{h}$ *or* $h \leq f \leq \overline{g}$ *is satisfied. Note that* $g \leq \overline{f} \Leftrightarrow f \leq \overline{g}$ *and if* $g \cdot h \neq 0$, *there is no function that satisfies the pair.*

For a given set of functions, Definition 3.2 can be extended to the following one.

Definition 3.3. *[6] A function f* ***satisfies*** *a* ***set of pairs to be distinguished*** $SPFD = \{(g_1, h_1), ..., (g_n, h_n)\}$ *, iff distinguishes every pair of the set, i.e.* $[(g_1 \leq f \leq \overline{h}_1) + (h_1 \leq f \leq \overline{g}_1)] \wedge ... \wedge [(g_n \leq f \leq \overline{h}_n) + (h_n \leq f \leq \overline{g}_n)]$.

Equivalently, SPFD can also be used to represent a set of functions that satisfy the SPFD. Therefore, $f \in SPFD$ can also be used to indicate that f satisfies SPFD

Essentially, SPFD is a collection of ISFs and it can be conceptualized in the form of a graph as well. However, a graph of SPFD may contain many connected components, each for one ISF. SPFD has been shown to better express flexibility of functions than ISF [6]. In a simple term, ISFs in a SPFD can be assigned to ON or OFF sets separately, instead of collectively when combining them into one ISF.

For a given circuit and its output functions, the SPFD at the output pins can be constructed from their ON and OFF sets. This SPFD can be represented as a bipartite graph. Each edge of the graph will be distributed to one of the node's inputs which can distinguish the edge [6]. Nodes will be processed from primary outputs to primary inputs. The summary of SPFD computation at a node is shown in Algorithm 1.

Algorithm 1 SPFD computation at a gate.

Require: $SPFD = (f_1^{on}, f_1^{off}), ..., (f_n^{on}, f_n^{off})$, inputs to the gate are $y_1, y_2, ..., y_k$

1: **for** each $(f_q^{on}, f_q^{off}) \in SPFD(f)$ **do**
2: Construct all possible minterms on the inputs of the gate,i.e., $b_0 = 000 = \overline{y}_1(x)\overline{y}_2(x)\overline{y}_3(x), ..., b_{2^n-1} = 111$.
3: Compute restricted minterms $a_i = b_i(f_q^{on} + f_q^{off})$. (f_q^{on}, f_q^{off}) is the care set. Thus, a_i described all minterms needed to be distinguished.
4: Distribute all care minterms into two sets:
1. $F_1 = a_i \mid a_i \subseteq f_q^{on}, a_i \neq 0$
2. $F_0 = a_i \mid a_i \subseteq f_q^{off}, a_i \neq 0$
5: Build complete bipartite graph $F = F_1 \times F_0$.
6: **for** each $(a_i, a_j) \in F$ **do**
7: Add (a_i, a_j) to at least one input k such that $a_i \leq f_k \leq \overline{a}_j$ or $a_j \leq f_k \leq \overline{a}_i$.
8: **end for**
9: **end for**

Algorithm 1 can be summarized as follows. For each pair of SPFD, **care** minterms are constructed (Line 2-3). Line 4-5 categorizes the minterms into two sets of minterms to be distinguished. Each pair of minterms will be assigned to a fanin of the gate that can distinguish the pair at Line 7. Applying Algorithm 1 to all nodes in the circuit from PO backward, SPFD of each node and wire can be computed.

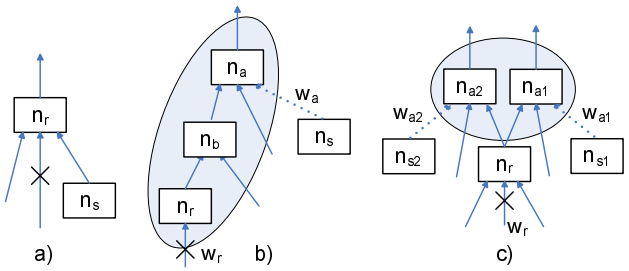

Fig. 1. SPFD-based rewiring techniques. a) and b) are local and global rewiring to replace one wire with none or one wire. c) replacement one wire with multiple wires, m-for-1 rewiring. Both global and m-for-1 rewiring can be viewed as local rewiring if nodes involved are merged as super nodes shown by shaded circles.

4. PREVIOUS WORK ON SPFD-BASED REWIRING

In this section, previous techniques on SPFD-based rewiring are outlined. We assume that $SPFD$ of each wire is already computed. The wire to be removed is denoted as w_r.

4.1. No additional wire for one wire removal (0-for-1) and One additional wire for one wire removal (1-for-1)

If $SPFD(w_r)$ can be redistributed to other existing wires, w_r can be removed without adding another wire, so called 0-for-1 rewiring. If another wire has to be added, the rewiring is called 1-for-1 rewiring. Rewiring algorithms process both rewiring techniques in similar ways. Hence, they are categorized based on the location of the target of additional wires.

4.1.1. Local rewiring

Let $n_r = sk(w_r)$, as shown in Figure 1a. If there is a wire w_a that satisfies $SPFD(w_r)$, *ie.*, $SPFD(w_r) \subseteq SPFD(w_a)$, w_a can be used to replace w_r. $f(n_r)$ needs to be updated after replacement. If $SPFD(w_r)$ is empty, w_r can be removed without adding any wire. Since the destination of both w_a and w_r are the same, this operation is referred to as local rewiring.

4.1.2. Global rewiring

A dominator node of w_r is defined as nodes through which all paths from w_r to any PO pass. The set of dominator nodes of w_r is denoted as $Dominator(w_r)$. For example, $Dominator(w_r) = \{n_a, n_b\}$ in Figure 1b. Let $sk(w_r) = n_r$. In global rewiring [3], the set of target nodes for additional wires is expanded from only n_r to $Dominator(w_r)$. If a node $n_a \in Dominator(w_r)$, the effect of removing w_r must pass through n_a. The global rewiring proceeds by removing w_r and propagating the change through fanout nodes of n_r until n_a is reached. A candidate w_a will be added as a fanin of n_a and checked if the resulting $SPFD(n_a)$ covers its original SPFD before removing w_r [3].

4.2. Many additional wires for 1 wire removal (m-for-1)

Even though, w_{a1} and w_{a2} in Figure 1c may not be used to individually replace w_r, they may collectively substitute w_r. However, only conditions that are likely to lead to this type of rewiring were suggested in [7]. For example, let $SPFD(w_r) \subseteq SPFD(n_{a1}) \cup SPFD(n_{a2})$. If $f(n_{s1})$ distinguishes $SPFD(w_r) \cap SPFD(w_{a1})$ and $f(n_{s2})$ distinguishes $SPFD(w_r) \cap SPFD(w_{a2})$, adding both $wire(n_{s1}, n_{a1})$ and $wire(n_{s2}, n_{a2})$ directs $SPFD(w_r)$ away from w_r. Hence, $SPFD(w_r) = \emptyset$ and w_r can be removed.

5. A UNIFIED FRAMEWORK

A generalized rewiring scheme, which includes all previous approaches as special cases, is proposed in this section. For a given n-variable function, *f*, we can also define the corresponding SPFD, representing pairs of n-tuples that the function can distinguish. Therefore, in this work we make the distinction between SPFD obtained from Algorithm 1 and SPFD that *f* can distinguish.

Definition 5.1. [1] *An SPFD derived from a node's global function is called arrival SPFD or $SPFD^A$. $SPFD^A$ at a node n can be computed from the functions of nodes in TFI(n). Therefore, $SPFD^A$ can also be perceived as forward propagation. If $n = sr(e)$,$SPFD^A(e) = SPFD^A(n)$. An SPFD at a node obtained by backward distribution (Algorithm 1) is call a required SPFD or $SPFD^R$.*

Properties of $SPFD^A$ and $SPFD^R$ can be shown in the following lemma.

Lemma 5.2.

$$SPFD^A(n_i) \subseteq \cup_{n_k \in FI(n_i)} SPFD^A(n_k) \quad (1)$$

$$SPFD^R(n_i) = \cup_{n_k \in FO(n_i)} SPFD^R(n_k) \quad (2)$$

Lemma 5.3. *A circuit, $\mathcal{C}$, works if and only if $\cup_{FI(n)} SPFD^R(n) = \cup_{FO(n)} SPFD^R(n), \forall n \in \mathcal{C}$ and either one of the following conditions hold.*

1. $SPFD^R(e) \subseteq SPFD^A(e), \forall e \in \mathcal{C}$.

2. $SPFD^R(n) \subseteq SPFD^A(n), \forall n \in \mathcal{C}$,

where $SPFD^R(n) = \cup_{e \in FO(n)} SPFD^R(e)$.

In SPFD computation outlined in Algorithm 1, a pair p will be distributed to the input *e* if $f(sr(e))$ distinguishes the pair, equivalently $p \in SPFD^A(sr(e))$. Therefore, the resulting SPFDs satisfy the above conditions.

[1] [8] also use arrival and required SPFDs but with different meanings.

A cut set can be derived from nodes' levels as follows. Each level has only one cut set. There are two types of nodes in the cut set at level i: 1) nodes with level i and 2) overpass nodes defined as nodes at levels less than i which have fanouts at levels greater then i but no fanout node at level i. Organizing a circuit into levels, the following corollary is obtained as a necessary and sufficient condition for correct circuit functionality.

Corollary 5.4. *A circuit works if*

1. $SPFD^A(n) \subset SPFD^R(n), \quad \forall n \in PO.$

2. Let N_i *be the set of nodes in the cut set of level i. For all levels i,* $\cup_{u \in PO} SPFD^R(u) \subseteq \cup_{v \in N_i} SPFD^A(v)$.

At a first glance, Corollary 5.4 is just a necessary condition because pairs may be lost according to (1). Therefore, some pairs in $SPFD^A$ existing at level i may not reach the output nodes which require them. However, if Corollary 5.4 holds, the first condition ensures that $SPFD^A$ exists at the appropriate POs, while the second condition ensures that the required $SPFD^R$ exists at all levels. However, Line 7 of Algorithm 1 dictates the propagation of $SPFD^R$ through a path from a primary input to a primary output. Thus, if the flow of $SPFD^R$ is maintained, Corollary 5.4 will lead to Lemma 5.3.

For a given circuit to be rewired, $SPFD^R$ at each node and wire can be computed by using Algorithm 1. Let W_r be a set of wires to be removed. The proposed replacement wires, W_a, has to be chosen such that $\cup_{e' \in W_r} SPFD^R(e') \subseteq \cup_{e \in W_a} SPFD^A(sr(e))$. After applying the wire removal and addition as well as updating $SPFD^A$ and $SPFD^R$ of all nodes in the circuit, Corollary 5.4 can be used to check if the rewiring still yields the correct circuit functionality. Note that the checking procedure does not pose any restrictions on the numbers of wires in W_r and W_a.

6. AN EFFICIENT IMPLEMENTATION

Since our scheme does not pose any restrictions on where and how new wires should be added, the checking process cannot be limited to a specific part of the circuit. As a result, a naïve implementation would result in unacceptable runtime. Three key techniques for efficient implementation are outlined in this section.

6.1. Processing when necessary

Let i be the minimum of $L(sr(e))$ for all edge $e \in W_r$. Since distribution of $SPFD^R$ at levels less than i are untouched, the checking can start at level i. Initially, all source nodes of wires in W_a and all sink nodes of wires in W_r will be processed. A node in any given level will be processed if at least one of its fanin nodes is processed and their global functions have been changed.

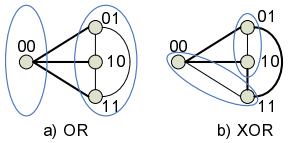

Fig. 2. $SPFD^A$ and bipartition. All minterms of a 2-input gate are organized into a graph. Different gate functions bipartition the graph in different ways as seen in a) and b). Thus, they produce different $SPFD^A$, shown by thick edges.

6.2. Implementable node functions

During SPFD computations in Algorithm 1, $SPFD^R$ is computed backwards. However, during the checking process, we are trying to propagate $\cup_{u \in FI(n)} SPFD^A(u)$ forward across node n. Different node functions will produce different $SPFD^A(n)$ sets. Consider the graph in Figure 2, which represent local $SPFD^A$ at a node with 2 inputs. The bipartition in Figure 2a, representing OR, propagates only (00,01), (00,10), (00,11) but not (01,10), (10,11), (01,11). However, XOR, bipartition in Figure 2b, propagates (00,01), (00,10), (11,01), (11,10).

At a given node n, we have to partition in such a way that $SPFD^R(n) \subseteq SPFD^A(n)$ and at the same time, we need $SPFD^A(n) \subseteq SPFD^R(w_r)$ to redirect $SPFD^R(w_r)$ away from w_r. This problem is formulated as a maximum cut graph bipartitioning problem in which a graph of $SPFD^R(n)$ is superimposed onto a graph of $\cup_{u \in FI(n)} SPFD^A(u)$. For those edges in $SPFD^R(n)$, their weights are set to be higher than the others to ensure they will be cut and then distinguished by the new node function. As there are 2^k nodes for a graph representing a k-input LUT, there are 2^{2^k} ways to bipartition the graph. Therefore, exhaustive search is inefficient even for $k = 4$. In a real situation, a graph representing $SPFD^R(n)$ may consists of many components and each component is a bipartite graph. The new node function must maintain bipartiteness of these components. Therefore, each component can be considered as one node and the maximum number of nodes can be reduced as well as the number of ways to bipartition them.

6.3. Early correctness declaration

When $SPFD^R(n_i) \subseteq SPFD^A(n_i)$ for all nodes n_i of a cut set, it is tempting to declare that the circuit functionality is correct by invoking Corollary 5.4 and assuming nodes in the cut set as artificial PO. However, it had been mentioned that if the function of some nodes change, $SPFD^R$ of their transitive fanouts may become non-bipartite which is not implementable by any one output binary function [9].

A simple example to demonstrate the phenomenon can be constructed as shown in Figure 3. A pair with 3-tuples and 2-tuple represent a global and local SPFD, respectively.

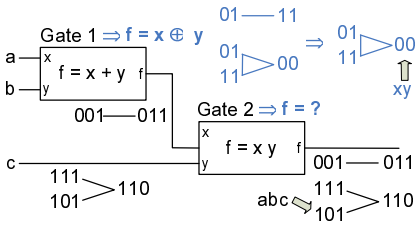

Fig. 3. Non-bipartiteness as a result of function changes.

SPFD at the output of Gate 2 contains two ISFs and they are distributed to fanin wires of Gate 2 as shown. If Gate 1 changes from OR to XOR, it is still able to distinguish (001,011) assigned to the gate. However, the new encoding at the input of Gate 2 changesAs can be seen, the required assignment is trying to distinguish between 11 and 01 in one ISF, while forcing 11 and 01 to be on the same side in the other ISF. Graphically, this encoding makes local SPFD at Gate 2 non-bipartite which is not implementable by any single output binary function.

However, if the new global function at a node n remains the same or becomes the inverse of the original global function before wire replacement, the $SPFD^A(n)$ will remain the same. Thus, entries in the column corresponding to Node n of a local function truth table at a node $n_k \in FO(n)$ will remain the same or flip which do not destroy the bipartiteness of local function at n_k, which is not the case for the example in Figure 3. Furthermore, $g(n_k)$ and $SPFD^A(n_k)$ will remain the same. As a result, if at a cut set i, $g(n_i)$ equals to $g_{org}(n_i)$ or $\overline{g}_{org}(n_i)$, for all nodes in the cut set, the circuit can be declared correct invoking Corollary 5.4 by using nodes in the cut set as artificial POs.

7. EXPERIMENTAL RESULTS

Rewiring can be used in various, diverse applications. In this work, we do not attempt to investigate rewiring ability for any application in particular. Therefore, our experiments are directed toward revealing the capability of rewiring independent of any specific application by considering all possible replacement wires.

First, the process to select candidate wires will be discussed. After that, rewiring ability will be compared against that of ATPG approach. Finally, rewiring ability using more than one wires will be reported.

7.1. Choosing replacement wires

We assume that a node is a 4-input LUT and it can accept at most one wire, even though it has more than one empty pins. This assumption should not severely limit rewiring ability as virtually no node has 3 free inputs and only some nodes have more than 1 free pin. For a given wire w_r to be removed, $SPFD^R(w_r)$ will be removed from $TFO(sk(w_r))$. These nodes will be used as possible targets for new wires.

The previous approaches search for wires with suitable $SPFD^A$. However, node functions, thus their $SPFD^A$, are derived without provisions for rewiring. Therefore, the number of candidates is limited. Consider an example in Figure 2. Let $SPFD^R(w_r) = (11, 00)$. Assuming Node n implements the XOR function, whose $SPFD^A(n)$ is shown in Figure 2b, with $SPFD^R(n) = (00, 01), (00, 10)$. According to previous approaches, Node n is not a candidate for rewiring. However, if the node function is changed to OR, $SPFD^A(n)$ can satisfy all $(00, 01)$, $(00, 10)$ and $(11, 00)$.

Thus, a node n_s is a possible source for a new wire, $wire(n_s, n_a)$, where $n_a \in TFO(sk(w_r))$ if

1. $\cup_{u \in FI(n_s)} SPFD^A(u) \supseteq SPFD^R(w_r) \cap SPFD^R(n_a)$.
2. The circuit depth does not increase.

7.2. Rewiring ability using none or one wire

In this experiment, the rewiring ability of using our scheme is compared against that of other techniques. Table 1 shows rewiring ability of different approaches. The circuits used in the experiment and their numbers of nodes and wires are shown in Columns 1,2 and 3, respectively. During SPFD computation, BDD sizes can grow exponentially (also observed in [10]). Thus, although our checking process is quite efficient, only medium size circuits can be handled in the current implementation. However, the technique proposed in [10] can speedup SPFD computation by at least 23 times which will make our techniques applicable to large circuits.

ATPG-based rewiring ability (quoted from [3]) are shown in Column 4. The best known rewiring ability of SPFD-based rewiring was reported in [3]. The order of SPFD distribution at Step 7 of Algorithm 1 affects rewiring ability as it dictates concentration of $SPFD^R$ on a wire. This observation has been used to improve rewiring ability [11] as well as power reduction for FPGAs [5]. Hence, Table 1 is not meant to be a direct comparison between our scheme and that of [3] as SPFD distribution of [3] is not known. Furthermore, in [3], the rewiring was declare feasible as soon as $SPFD^R$ of a target node covered the one before wire removal. Thus, some nodes may not be implementable. (see Section 6.3 for an example.) Thus, we quote the results as a pseudo upper bound on rewiring ability in Column 6.

Our implementation uses CUDD (a BDD package) to represent functions and SPFDs. As replacement wires are observed to be close the removed wire [3], a circuit is partitioned into clusters of 60 nodes by using hMetis, similar to [3]. As partitioning might limit rewiring ability especially for wires next to partition lines, we ran our scheme in multiple passes. The wires with successful rewiring are not considered again in subsequent passes. To reduce the effect of partitioning, a wire crossing a partition line are discouraged

Table 1. Rewiring ability.

circuits			rewiring ability		
name	#nodes	#wires	ATPG from [3]	attainable rewiring	upper bound
term1	106	244	60	93	99
ex2	220	433	42	124	136
x1	222	557	104	183	222
x3	403	958	65	292	318
apex6	421	1025	92	245	346
dalu	448	1338	157	465	704
alu2	162	510	106	184	253
C1908	160	423	18	80	96
C432	190	538	81	115	176

Table 2. Rewiring ability with more than one wire.

circuit	#repleacement wire						average checking
	0	1	2	3	4	5	time / candidate (sec)
term1	25	68	0	0	2	0	0.067
ex2	5	119	1	0	1	0	0.026
x1	15	168	8	0	0	0	0.050
x3	14	278	55	2	0	0	0.034
apex6	69	176	16	19	0	0	0.047
dalu	49	416	0	1	0	0	0.112
alu2	7	177	1	0	0	0	0.532
C1908	1	79	16	0	0	0	0.153
C432	6	109	3	1	4	0	0.430

to be cut in the next pass. The rewiring ability of our scheme using three passes is shown in Column 5 of Table 1.

Our attainable rewiring ability tracked the upper bounds well. However, the difference is widen for circuits with complex structures, measured by the ratio of the number of wires to that of nodes. For example, the gaps are wide for $C432$ and $dalu$ which have the ratios about 2.8, but small for $term1$ and $x3$ which have the ratios about 2.3. In complex circuits, many nodes may have multi-component local $SPFD^R$ which becomes un-implementable.

7.3. Rewiring ability using more than one wire

Rewiring using more than one wire was claimed to improve rewiring ability when 5-input LUTs were used [7]. But, the method implicitly used m-for-1 rewiring in delay optimization without reporting rewiring ability due to a lack of explicit techniques [7]. Rewiring with many wires is usually time consuming. Thus, rewiring ability of using several wires must be studied for runtime-quality tradeoffs. In this section, the same checking scheme used in the previous experiment was used to reveal rewiring ability.

Rewiring ability of our scheme for using 0 to 5 replacement wires is shown in Table 2. Although logic synthesis is supposed to remove all redundant wires, $SPFD^R$ redirection was able to make some wires redundant without adding any wire (see Column 2). Rewiring ability of one additional wire increased by multiple folds. However, as the number of additional wires increases further, rewiring ability drastically decrease and becomes virtually zero if three or more wires are used because for most of the time there are not enough candidate replacement wires.

The orders of replacement wire generation vary for different applications and implementations. Thus, the runtime for checking each proposed rewiring averaging across using 0 to 5 replacement wires was reported in Column 8 instead of the total runtime. The upper bound on runtime of any application can be estimated as the number of candidates times this average checking time. The total runtime is consisted of checking each candidate and SPFD computation, performed once for a cluster. The SPFD computation time varies with the cluster size as well as complexity of functions in the cluster from seconds for $term1$ to 2 hours for some clusters of $dalu$. Note that this SPFD computation is common among all SPFD-based rewiring method and its runtime can be reduced by at least 23 times using the simulation-and-SAT (S&S) technique [10].

8. REFERENCES

[1] S. Yamashita, et al., "A new method to express functional permissibilities for LUT based FPGAs and its applications," *Proc. Int. Conf. Computer-Aided Design*, 1996, pp. 254–261.

[2] S. Sinha, "SPFDs: A new approach to flexibility in logic synthesis," *Ph.D Thesis, University of California, Berkeley*, 2002.

[3] J. Cong, et al., "SPFD-based global rewiring," *Proc. the Int. Symp. on Field-programmable gate arrays*, 2002, pp. 77–84.

[4] T. Kouda, et al., "Reduction of the number of FPGA blocks by maximizing flexibility of internal functions," *IEICE Trans Fundamentals*, vol. E81-A, no. 12, pp. 2554–2562, 1998.

[5] B. Kumthekar, et al., "Power and delay reduction via simultaneous logic and placement optimization in FPGAs," *Proc. Design And Test in Europe Conf.*, 1998, pp. 202–207.

[6] R. K. Brayton, "Understanding SPFDs: A new method for specifying flexibility," *Proc. Int. Works. on Logic Synthesis*, 1997.

[7] K. Tanaka, et al., "SPFD-based flexible transformation of LUT-based FPGA circuits," *IEICE Trans Fundamentals*, vol. E88-A, no. 4, pp. 1038–1046, 2005.

[8] S. Sinha, et al., "Topologically constrained logic synthesis," *Proc. Int. Conf. Computer-aided design*, 2002, pp. 679–686.

[9] S. Sinha and R. K. Brayton, "Implementation and use of SPFDs in optimizing boolean networks," *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 103–110.

[10] A. Mishchenko, et al., "Using simulation and satisfiability to computeflexibilities in boolean networks," vol. 25, no. 5, pp. 743–755, May 2006.

[11] J. Cong, et al., "A new enhanced SPFD rewiring algorithm," *Proc. Int. Conf. Computer-aided design*, 2002, pp. 672–678.