

Theory and Algorithm for *SPFD*-Based Global Rewiring

Jason Cong and Wangning Long

Department of Computer Science, University of California, Los Angeles, CA 90095

Abstract

In this paper we present the theory and the algorithm for *SPFD*-based global rewiring (*SPFD-GR*), which allows us to replace a target wire globally in the circuit (by some wire possibly far away from the target). It successfully overcomes the limitation of the existing *SPFD*-based local rewiring (*SPFD-LR*) that can only replace a wire with another wire having the same sink node. We apply *SPFD-GR* to the post-mapping area reduction for *LUT*-based FPGAs under circuit depth restriction. Experimental results show that the rewiring ability of *SPFD-GR*, in terms of the number of target wires found to have alternative wires, is 1.45 and 3 times that of *SPFD-LR* and an ATPG algorithm (with a preliminary experimental flow), respectively, and the run time is quite acceptable. Using partitioning, the *SPFD-GR* algorithm scales well to large circuits with good synthesis quality.

1 Introduction

Rewiring is a technique that replaces a wire with another wire so as to achieve performance improvement or area reduction. Recently, it has received increasing attention due to the need of closer synthesis and layout interaction for timing closure. The rewiring problem has been widely investigated. The existing approaches are based on the automatic test pattern generation (ATPG) [1][4][5][10][12][13], the symmetry detection [3], the graph pattern recognition [17], and the *SPFD* (*Set of Pairs of Functions to be Distinguished*) [18][15], etc.

Among them, ATPG-based redundancy addition and removal is probably the earliest and most widely used approach. It uses ATPG techniques to add a redundant wire (*alternative wire*) to make the *target* wire redundant and removable. The advantage of the ATPG-based method is that it is capable of global rewiring, i.e., removing a target wire by adding an alternative wire “far away” from the target wire in the circuit. When applied to *LUT*-based FPGA circuits, however, the ATPG-based rewiring has limited flexibility of changing the internal logic function of a node, which may provide a greater opportunity for optimization.

Recently, the *SPFD*-based rewiring was proposed and developed in [18] and [15]. It has been used in technology-independent logic synthesis [15], *LUT*-based FPGA synthesis [18], floorplanning and placement of multi-level PLAs [6], and low power design for FPGAs [11]. The *SPFD*-based rewiring algorithm calculates a set of function pairs to be distinguished (*SPFD*) at each wire according to the initial circuit structure and the circuit’s primary output functions. During the rewiring stage, if the *SPFD* at a node’s in-pin, say p_1 , can be distinguished by the logic function at another node’s out-pin, say p_0 , then a new wire $p_0 \rightarrow p_1$ can replace the original wire ended at p_1 .

Unlike the ATPG-based method, however, the conventional *SPFD*-based rewiring (referred to *SPFD-LR* in this paper) can only find alternative wires *locally*, by which we mean that the sink node of the alternative wire is the same as that of the target wire. Conversely, the ATPG-based method is capable of finding a

global alternative wire that might be *far away* from the target wire.

In this paper we propose an *SPFD*-based global rewiring (*SPFD-GR*). Our purpose is to apply this method to *LUT*-based FPGA synthesis. Our main contributions include:

- 1) We developed the theory and algorithm for solving a fundamental problem in *SPFD* based rewiring: Given the set of in-pin functions of a node and the *SPFD* at the node’s out-pin, is there a way to modify the node’s internal function so that the *SPFD* at the node’s out-pin can be satisfied?
- 2) Using the concept of the dominators and node modification technique stated above, we developed *SPFD-GR* that allows global rewiring with the flexibility of changing internal functions of nodes to maximize the opportunity of rewiring.
- 3) Using a state-of-art multi-level / multi-way partitioning algorithm [8], *SPFD-GR* scales well to large designs.

2 Terminology and Definitions

In this section we will review some terminology and definitions. The circuits used in this paper are combinational circuits. We restrict the circuit in our research to a K -bounded network, meaning that each **logic cell** (or **node**) in the circuit, as shown in Figure 1, has an **out-pin** p_0 and up to K **in-pins** ($p_1, p_2, \dots, p_n, n \leq K$). Each node has an internal logic function $p_0 = f(p_1, \dots, p_n)$ that defines the logic relationship between the *out-pin* and the *in-pins* of the node. In this paper, the internal logic function of a node can be any *combination* of the node’s inputs. Every pin in the circuit has a global logic function in terms of the primary inputs of the circuit (denoted as g_i in Figure 1).

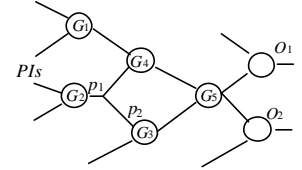
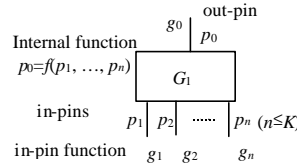


Figure 1 Logic cell

Figure 2 Illustration of dominator

For wire $p_1 \rightarrow p_2$, as shown in Figure 2, we call G_2 as its **source node** and G_3 as its **sink node**. A **transitive fanout of pin** p_i is a node on one of the paths from p_i to a primary output (**PO**). A **transitive fanout of a node** is a transitive fanout of the node’s out-pin. A **dominator** of pin p_i is a transitive fanout of p_i through which all the paths from p_i to POs pass. A **dominator of a wire** is the dominator of the wire’s sink pin. For example, in Figure 2, G_5 is the only dominator of pin p_1 while both G_3 and G_5 are the dominators of pin p_2 as well as wire $p_1 \rightarrow p_2$.

A function f is said to **distinguish** a function pair (p_1, p_0) , where $p_1 \neq 0$, $p_0 \neq 0$ and $p_1 p_0 \equiv 0$, if either one of the following two conditions is satisfied [15]:

$$p_1 \leq f \leq \bar{p}_0 \quad \text{or} \\ p_0 \leq f \leq \bar{p}_1$$

candidate wire and repeat this step. If no candidate is left, go to the next step;

- 6) **(Fail)** Restore the functions of G_1 and its transitive fanouts until G_D . Return **fail**.
- 7) **(Success)** Permanently remove w_r , and
 - Update the internal function of transitive fanouts of G_D as necessary.
 - Update the SPFDs from the changed nodes close to POs backwards to the sink node of the wire selected as the next target wire. Return **success**.

By removing a wire from a node in Step 1 of the above algorithm, we mean to set the wire's logic value as 0 or 1. As our approach deals with complex nodes, it can sometimes be difficult to determine how to assign 1 or 0 to the wire. We have a procedure to guide the assignment that minimizes the number of wires to be removed from this node. For example, given a node's internal function $f = a \cdot b + c$, where a is to be removed, we shall set $a = 1$ so that b and c are not removed at the same time. Otherwise, if we set $a = 0$, b will also be removed. Increasing the number of wires to be removed may decrease the possibility for Steps 3 and 4 to be successful.

Note that our algorithm assumes that the target wire is given. Choosing a target wire depends on the application used. To reduce the circuit delay, we select a wire on the critical path as the target wire. If our objective is to minimize the circuit area, we select a wire that enables deletion of a node or packing with some other nodes.

It is worth noting that in Figure 4, G_1 is also a dominator of w_r . Therefore, the above *SPFD-GR* algorithm actually covers the cases covered by *SPFD-GR*. In practice we use *SPFD-LR* to do rewiring when $G_1 = G_D$.

5 Theory of SPFD-Based Global Rewiring

As seen in the previous section, the key question for the *SPFD-GR* algorithm can be generalized as two questions: 1) For a node in the circuit, given its in-pin functions and out-pin *SPFD*, is there a way to modify the internal function of the node so that its out-pin function still distinguishes its out-pin *SPFD*? 2) If no to question 1, can we add a wire to the node and modify the node's internal function so as to make its out-pin function distinguish its out-pin *SPFD*? We call this problem the *node modification problem*.

In the following two subsections, we will present two efficient checking procedures to solve the node modification problem. We consider two cases: i) modifying a node without adding a wire; and ii) modifying a node by adding a wire.

5.1 Node Modification Without Wire Addition

Given a node's in-pin functions, the output function of the node can be expressed as the sum of minimum product terms of the in-pin functions, which is defined as following.

Definition 1 (minimum product term): Let $\mathbf{B} = \{\mathbf{b}_0, \mathbf{b}_1, \dots, \mathbf{b}_{N-1}\}$ ($N = 2^n$), where \mathbf{b}_i ($0 \leq i \leq N-1$) is a **minimum product term** of the node's in-pin functions, called **MP-term**, which is in the form of

$$\begin{aligned} \mathbf{b}_0 &= \bar{g}_1 \bar{g}_2 \cdots \bar{g}_n \\ \mathbf{b}_1 &= \bar{g}_1 \bar{g}_2 \cdots g_n \\ &\dots \\ \mathbf{b}_{N-1} &= g_1 g_2 \cdots g_n \end{aligned} \quad (1)$$

where g_i ($1 \leq i \leq n$) is the global function at node G_1 's i -th input. Given a function pair $(\mathbf{p}_1, \mathbf{p}_0)$, $\mathbf{a}_i = (\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_i$ is called the *restricted*

MP-term. $f_a = (\mathbf{p}_1 + \mathbf{p}_0)f$ is called the **restricted function** of f , where f is the original function. The restricted function and the original function have the following relationship:

Lemma 1: $\mathbf{p}_1 \leq f \leq \bar{\mathbf{p}}_0 \Leftrightarrow \mathbf{p}_1 \leq f_a \leq \bar{\mathbf{p}}_0$ and $\mathbf{p}_0 \leq f \leq \bar{\mathbf{p}}_1 \Leftrightarrow \mathbf{p}_0 \leq f_a \leq \bar{\mathbf{p}}_1$.

In other words, f distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$ if and only if f_a distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$.

To choose a proper node function f that satisfies the SPFD at the node's out-pin, we can try every combination of the *MP-terms*, i.e.

$$f = \sum_{i=0}^{N-1} k_i \mathbf{b}_i, \text{ where } k_i = 0 \text{ or } 1. \text{ However, the time complexity of}$$

this simple approach is too high $O(2^{2^n})$. For a single-pair SPFD, the following theorem provides a more efficient approach to perform the above checking with only $O(2^n)$ time complexity. This is much more efficient and quite affordable for a node with a small number of inputs, usually **4** or **5** for LUT-based FPGAs.

Theorem 1: Given a node whose out-pin SPFD is $P = \{(\mathbf{p}_1, \mathbf{p}_0)\}$ and in-pin functions are g_1, g_2, \dots, g_n , where n is the input number of the node, if every non-empty restricted *MP-term* $\mathbf{a}_i = (\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_i$ satisfies *one* of the following conditions,

$$\begin{cases} \mathbf{a}_i \leq \mathbf{p}_0 \text{ or} \\ \mathbf{a}_i \leq \mathbf{p}_1 \end{cases} \quad (2)$$

Then there exists a function that distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$. Particularly, f can be constructed as following,

$$f = f(g_1, g_2, \dots, g_n) = \sum_{i=0}^{N-1} k_i \mathbf{b}_i \quad (3)$$

where $N=2^n$; $k_i = 0$ if $\mathbf{a}_i \leq \mathbf{p}_0$, and $k_i = 1$ if $\mathbf{a}_i \leq \mathbf{p}_1$.

Proof: (i) Let $f_a = \sum_{i=0}^{N-1} k_i \mathbf{a}_i = \sum_{i=0}^{N-1} k_i (\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_i$, where $k_i = 0$ if $\mathbf{a}_i \leq \mathbf{p}_0$; $k_i = 1$ if $\mathbf{a}_i \leq \mathbf{p}_1$. It is clear that f_a only contains $\mathbf{a}_i \leq \mathbf{p}_1$. Hence, $f_a \leq \mathbf{p}_1$.

(ii) On the other hand, since $\sum_{i=0}^{N-1} \mathbf{a}_i = (\mathbf{p}_1 + \mathbf{p}_0)$, we know that

$\sum_{i=0}^{N-1} \mathbf{a}_i$ includes all the min-terms (of the primary inputs) of \mathbf{p}_1 . Since $\mathbf{p}_1 \mathbf{p}_0 \equiv 0$, we know that \mathbf{a}_i contains no min-terms of \mathbf{p}_1 when $\mathbf{a}_i \leq \mathbf{p}_0$. Thus, f_a includes all the min-terms of \mathbf{p}_1 . Hence, $f_a \geq \mathbf{p}_1$.

Combining (i) and (ii), we know $f_a = \mathbf{p}_1$, which means that f_a satisfies $\mathbf{p}_1 \leq f_a \leq \bar{\mathbf{p}}_0$ (because $\mathbf{p}_1 \mathbf{p}_0 \equiv 0$). Therefore, from Lemma 1, we know $\mathbf{p}_1 \leq f \leq \bar{\mathbf{p}}_0$. Q.E.D.

Actually, Equation (3) gives us a way to construct the internal function of a node when the condition in Theorem 1 is satisfied. For node G in Figure 1, we can get G 's internal function by substituting g_i in each *MP-term* of (3) with p_i and f with p_0 . (We can also use the method proposed in [18] to construct the internal function.)

Example 3: For node G in Figure 3, suppose the SPFD at G 's out-pin p_0 is $SPFD_0 = \{(f_1, f_0)\}$, where $f_1 = \bar{x}_1 x_2 + x_1 \bar{x}_2$ and $f_0 = \bar{x}_1 \bar{x}_2 + x_1 x_2$; the in-pin functions are $g_1 = \bar{x}_1 x_2$ at p_1 and $g_2 = \bar{x}_1 + x_2$ at p_2 .

Since $f_0 + f_1 = 1$, we have

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{b}_0 = \bar{g}_1 \bar{g}_2 = x_1 \bar{x}_2 \leq f_1 \\ \mathbf{a}_1 &= \mathbf{b}_1 = \bar{g}_1 g_2 = \bar{x}_1 \bar{x}_2 + x_1 x_2 \leq f_0 \\ \mathbf{a}_2 &= \mathbf{b}_2 = g_1 \bar{g}_2 = 0 \\ \mathbf{a}_3 &= \mathbf{b}_3 = g_1 g_2 = \bar{x}_1 x_2 \leq f_1 \end{aligned}$$

As all the restricted *MP-terms* satisfy the conditions in (2), we get G 's output function $f = \mathbf{b}_0 + \mathbf{b}_3$ that distinguishes (f_1, f_0) .

By substituting f with p_0 , g_1 with p_1 , and g_2 with p_2 , we get a feasible internal function $p_0 = \bar{p}_1 \bar{p}_2 + p_1 p_2$. (**End of example**)

Usually, the function constructed by Equation (3) needs to be simplified, *e.g.*, using SIS's *node simplification* [14].

In fact, we can show that the conditions in Theorem 1 are also necessary for a node to have an output function that distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$. The proof is not included here due to the page limitation.

Theorem 1 assumes that the SPFD at the output of a node has a single function pair. In practice, however, the SPFD at a node's out-pin may contain several function pairs. The following theorem demonstrates a way to combine the function pairs.

Theorem 2: Given the SPFD, $P = \{(\mathbf{p}_{11}, \mathbf{p}_{10}), (\mathbf{p}_{21}, \mathbf{p}_{20}), \dots, (\mathbf{p}_{m1}, \mathbf{p}_{m0})\}$, at a node's out-pin, and a function \mathbf{j} which distinguishes all the pairs in P , without loss of generality, suppose $\forall (\mathbf{p}_{i1}, \mathbf{p}_{i0}) (1 \leq i \leq m)$, $\mathbf{p}_{i1} \leq \mathbf{j} \leq \bar{\mathbf{p}}_{i0}$. Let $\mathbf{p}_1 = \sum_{i=1}^m \mathbf{p}_{i1}$ and $\mathbf{p}_0 = \sum_{i=1}^m \mathbf{p}_{i0}$,

Then $f = f(g_1, g_2, \dots, g_n)$ distinguishes any pair $p \in P$ if f distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$.

Proof: Without loss of generality, assume $\bar{\mathbf{p}}_1 \leq f \leq \bar{\mathbf{p}}_0$. Hence, for any function pair $(\mathbf{p}_{i1}, \mathbf{p}_{i0}) \in P$, there must be $\mathbf{p}_{i1} \leq \mathbf{p}_1 \leq f$.

On the other hand, $f \leq \bar{\mathbf{p}}_0 \leq \sum_{j=1}^m \bar{\mathbf{p}}_{j0} \leq \bar{\mathbf{p}}_{10} \cdot \bar{\mathbf{p}}_{20} \cdots \bar{\mathbf{p}}_{m0} \leq \bar{\mathbf{p}}_{i0}$.

Therefore $\mathbf{p}_{i1} \leq f \leq \bar{\mathbf{p}}_{i0}$.

Similarly, we can prove that $\mathbf{p}_0 \leq f \leq \bar{\mathbf{p}}_1 \Rightarrow \mathbf{p}_{i0} \leq f \leq \bar{\mathbf{p}}_{i1}$.

Therefore, f distinguishes any pair in P if f distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$. (Q.E.D.)

Notice that the condition in Theorem 2 is only a sufficient condition, but no longer a necessary condition (different from Theorem 1). In Theorem 2, function \mathbf{j} is needed to classify the functions in each pair into \mathbf{j} 's on-set and off-set. Since the node's initial output function must satisfy the SPFD at its out-pin, we can simply use it as \mathbf{j} .

5.2 Node Modification by Adding A Wire

Given a node, if no combination of its in-pin functions distinguishes its out-pin SPFD, we can try to add a wire to the node so that the combination of the in-pin functions, including the new wire's function, distinguishes the out-pin SPFD. The following algorithm gives us an efficient way to determine which wire can be added into the node.

Wire Addition Algorithm:

- 1) Calculate the *MP-term* set, $\mathbf{B} = \{\mathbf{b}_0, \mathbf{b}_1, \dots, \mathbf{b}_{N-1}\}$ ($N = 2^n$), which is defined in (1). The non-empty *MP-terms* can be classified into three sets, $\mathbf{B}_0, \mathbf{B}_1$ and \mathbf{B}_2 , where \mathbf{B}_0 is the set whose members satisfy $(\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_i \leq \mathbf{p}_0$; \mathbf{B}_1 is the set whose members satisfy $(\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_i \leq \mathbf{p}_1$; and \mathbf{B}_2 is the set whose members do not satisfy any conditions in (2). Let $\mathbf{B}_1 = \{\mathbf{b}_{i_1}, \mathbf{b}_{i_2}, \dots, \mathbf{b}_{i_m}\}$ and $\mathbf{B}_2 = \{\mathbf{b}_{s_1}, \mathbf{b}_{s_2}, \dots, \mathbf{b}_{s_k}\}$. If $\mathbf{B}_2 = \bar{\mathcal{A}}$, then we can successfully modify the node without adding a wire, return **success**. Otherwise, it is necessary to add a wire, go to next step.
- 2) Choose a candidate node in the network from which we want to link a wire to G 's new in-pin p_{n+1} . Suppose its function is g_{n+1} . For each $\mathbf{b}_{s_i} \in \mathbf{B}_2$, let $\mathbf{a}_{s_i} = (\mathbf{p}_1 + \mathbf{p}_0)\mathbf{b}_{s_i}$, calculate

$r_0 = \bar{g}_{n+1}\mathbf{a}_{s_i}$, and $r_1 = g_{n+1}\mathbf{a}_{s_i}$. If each satisfies either $r_i \leq \mathbf{p}_0$ or $r_i \leq \mathbf{p}_1$ ($r_i \neq 0$), we can add this wire. Go to the next step. Otherwise, find another candidate and repeat this step. If all nodes have been tried, return **fail**.

- 3) Using the calculation results in Steps 1 and 2 to get the G 's output function that distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$:

$$f = f(g_1, g_2, \dots, g_n, g_{n+1}) = f_1 + \sum_{i=0}^{N-1} k_i \mathbf{b}_{s_i} g_{n+1} + \sum_{j=0}^{N-1} k_j \mathbf{b}_{s_j} \bar{g}_{n+1} \quad (4)$$

where $f_1 = \mathbf{b}_{i_1} + \mathbf{b}_{i_2} + \dots + \mathbf{b}_{i_m}$, $\mathbf{b}_i \in \mathbf{B}_1$, which is obtained in Step 1. For each k_i in the second term, $k_i = 1$ if $\mathbf{a}_{s_i} g_{n+1} \leq \mathbf{p}_1$, and $k_i = 0$ else. For each k_j in the third term, $k_j = 1$ if $\mathbf{a}_{s_j} \bar{g}_{n+1} \leq \mathbf{p}_1$, and $k_j = 0$ else.

- 4) Use the similar method to the one used in the previous section to calculate G 's internal function according to (4). Return **success**.

As \mathbf{B}_2 is a sub-set of \mathbf{B} , which is usually very small, the checking procedure in the above algorithm does not consume too much time. In addition, \mathbf{B}_1 and \mathbf{B}_2 remain the same for any candidate node. Therefore, they can be re-used in the computation.

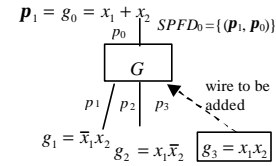


Figure 5. Illustration of Node modification with wire addition

Example 3 (Node modification with wire addition): Given the in-pin functions g_1 and g_2 as shown in Figure 5. The SPFD at G 's out-pin p_0 is $SPFD_0 = \{(\mathbf{p}_1, \mathbf{p}_0)\}$, where $\mathbf{p}_1 = x_1 + x_2$ and $\mathbf{p}_0 = \bar{x}_1 \bar{x}_2$.

We carry out the wire addition algorithm in the following steps:

- 1) Condition checking (without wire addition):

$$\begin{aligned} \mathbf{a}_0 &= \bar{g}_1 \bar{g}_2 (\mathbf{p}_1 + \mathbf{p}_0) = \bar{x}_1 \bar{x}_2 + x_1 x_2 \\ \mathbf{a}_1 &= \bar{g}_1 g_2 (\mathbf{p}_1 + \mathbf{p}_0) = x_1 \bar{x}_2 \leq \mathbf{p}_1 \\ \mathbf{a}_2 &= g_1 \bar{g}_2 (\mathbf{p}_1 + \mathbf{p}_0) = \bar{x}_1 x_2 \leq \mathbf{p}_1 \\ \mathbf{a}_3 &= g_1 g_2 (\mathbf{p}_1 + \mathbf{p}_0) = 0 \end{aligned}$$

Thus, we can set $\mathbf{B}_0 = \bar{\mathcal{A}}$, $\mathbf{B}_1 = \{\mathbf{a}_1, \mathbf{a}_2\}$ and $\mathbf{B}_2 = \{\mathbf{a}_0\}$. As $\mathbf{B}_2 \neq \bar{\mathcal{A}}$, it is necessary to add a wire.

- 2) Try to add a wire from $g_3 = x_1 x_2$, check

$$\begin{aligned} \bar{g}_1 \bar{g}_2 g_3 &= \bar{x}_1 \bar{x}_2 \leq \mathbf{p}_0 \\ \bar{g}_1 g_2 g_3 &= x_1 x_2 \leq \mathbf{p}_1 \end{aligned}$$

- 3) The conditions are satisfied. Therefore we can add the wire from g_3 to G . $g_0 = \bar{g}_1 g_2 + g_1 \bar{g}_2 + \bar{g}_1 \bar{g}_2 g_3$ is a function that distinguishes $(\mathbf{p}_1, \mathbf{p}_0)$.

- 4) Finally, we get G 's internal function:

$$p_0 = \bar{p}_1 p_2 + p_1 \bar{p}_2 + \bar{p}_1 \bar{p}_2 p_3$$

6 Implementation Notes

In the implementation of *SPFD-GR*, CUDD BDD [16] is used to represent logic function and SPFDs. The logic function at each pin is expressed as the primary inputs of the circuit, which is represented by a BDD. SPFDs are also represented by BDDs. The runtime of *SPFD-GR* algorithm is determined by the runtime of BDD constructions and variable BDD operations.

In order to handle large circuits, we use a state-of-the-art multi-level, multi-way partitioning program, named KPM/MESC [8], to

partition a circuit into smaller blocks and separately apply *SPFD-GR* to each block. After all the blocks have been processed, we combine the results to get the solution for the entire circuit.

7 Experimental Results

We applied the *SPFD-GR* in the *LUT*-based FPGA synthesis. We performed three experiments: 1) rewiring ability comparison; 2) area minimization for large circuits, and 3) impact of the partition size on the synthesis quality. As the circuit depth is important for *LUT*-based FPGAs, all of our experiments are under the circuit depth restriction, which means that the changes to the circuit will not increase the maximum circuit depth. All of our experiments are carried out on Sun Ultra 60 workstation.

Table 1 compares *SPFD-GR* with *SPFD-LR* and an ATPG-based rewiring algorithm with a preliminary experimental flow. A0 refers to the case that the ATPG rewiring engine does not use the recursive learning, while A1 refers to the case that the ATPG engine uses the recursive learning of level 1.

In the experiments for Table 1, each program traverses the entire circuit once, selecting every wire as a target wire and trying to find alternative wires that satisfy the maximum depth restriction. For the purpose of collecting statistical data, we did not make real changes to the circuit. The circuits used in Table 1 are 4-*LUT* networks obtained through *script.rugged*, *Cutmap* [7], *Red_Removal* and *Greedy_Pack*. All of these routines are available in SIS [14] and RASP [9]. Column 2 lists the number of wires in each circuit. Columns 3 ~ 6 show the *rewiring ability* of the four methods, which are ATPG-based algorithms without (A0) or with (A1) recursive learning, *SPFD-LR* and *SPFD-GR*, respectively. By *rewiring ability* of a rewiring algorithm, we mean the *percentage of wires* having at least one alternative wire. Columns 7 ~ 10 show the run time in seconds of the four algorithms.

We implemented *SPFD-LR* according to [18] and used it for comparison purpose. Moreover, we also used it in our program to cover the cases of local rewiring (which is covered by *SPFD-GR*). For ATPG-based rewiring, we use an ATPG rewiring engine that was used in [10][12], which is originally designed the rewiring for the cell-based design. To apply this ATPG rewiring engine, we decompose the *LUT*-based network into a simple gate based network that is then fed into the rewiring engine returning the alternative wire candidates. Notice that our experiment does not reflect the recent improvements of ATPG-based rewiring, e.g., [5]. The comparison with ATPG-based rewiring is preliminary.

Table 1 shows that the rewiring ability of the ATPG rewiring algorithms with and without recursive learning are 12% and 9.8% respectively. The rewiring ability of *SPFD-LR* and *SPFD-GR* are 25.3% and 36.7% respectively.

We applied the *SPFD*-based algorithms to the post-mapping area reduction. We use 5 circuits whose size is up to 100K 2-input gates. The results are listed in Table 2.

The circuits fed into the rewiring program are 4-*LUT* networks. Ind1 ~ Ind4 are synthesized and mapped by “*script.algebraic*” [14], *Cutmap* and *Greedy_Pack*. Ind5 only goes through *cutmap* and *greedy_pack* because “*script.algebraic*” fails to get a result within 20 hours. We use a greedy strategy to do the area minimization, which tries to remove as many wires as possible. At the end of each pass, *greedy_pack* [9] is called to further pack the circuit. We chose this strategy based on two considerations: 1) If a

Circuit Name	Num. of wires	Rewiring ability				CPU time (Secs)			
		A0	A1	SPFD-LR	SPFD-GR	A0	A1	SPFD-LR	SPFD-GR
C1908	423	16	18	70	96	4.2	27.6	37.6	54.9
C432	538	43	81	134	176	8.7	47.9	12.1	32.5
C5315	1772	141	160	402	510	17.0	92.0	42.7	247.1
alu2	510	88	106	159	230	25.2	130.4	1.5	3.2
alu4	939	154	194	267	399	103.2	763.3	4.8	12.9
apex6	1025	70	92	261	335	37.7	258.0	5.4	11.0
dalu	1338	142	157	457	693	176.8	977.1	17.4	46.6
example2	433	39	42	88	139	14.3	26.1	1.3	1.7
x1	557	93	104	155	215	9.1	25.2	3.3	5.8
x3	958	48	65	156	322	23.3	132.0	5.5	13.4
Total	8493	834	1019	2149	3115	419.5	2479	131.6	429.2
Ratio	1	9.8%	12%	25.3%	36.7%	3.2	18.8	1	3.3

Table 1. Comparison of rewiring ability for 4-*LUT* FPGA designs under circuit depth restriction

node has only one fanout wire and it can be removed, then the node can also be removed. 2) If the neighboring nodes have no more than 4 inputs, *greedy_pack* may further pack them.

Since these circuits are quite large, we used KPM/MESC partition program [8] to partition the design into a set of blocks. The average partitioning size, which is the *average number* of *LUT* nodes within each partition, is set to 60 *LUT* nodes and the skew of partition is set to 5%. Note that by this setting there may be some blocks sized larger than 80 *LUT* nodes. The program runs 4 iterations for a circuit. Within each iteration, the program runs up to 3 times for each partition. All the results in Table 2 pass the simulation verification of SIS with 8000 random patterns.

In Table 2, area represents the number of 4-*LUT* nodes. Column 2 lists the area of initial circuits. Columns 3 ~ 6 show the area and run time of *SPFD-LR* and *SPFD-GR*. The data show that *SPFD-LR* and *SPFD-GR* method achieve the area reductions of 11.4% and 14.4%, respectively. Notice that *SPFD-GR*'s run time is only 1.2 times that of *SPFD-LR*. Therefore, the speed of *SPFD-GR* is quite acceptable with a moderate partition size.

Circuit	Init. Area	SPFD-LR		SPFD-GR	
		Area	CPU(s)	Area	CPU(s)
Ind1	11051	10423	1723	10205	2084
Ind2	10553	9892	1815	9660	2512
Ind3	11432	8282	2777	8052	3264
Ind4	20200	18237	6149	17449	6812
Ind5	44922	40095	11987	38657	14265
Aver	19632	17386	4890	16805	5787
Ratio	1	88.6%	1	85.6%	1.2

Table 2 Area minimization for large circuit under circuits depth restriction with average partition size of 60.

A major concern is whether the use of the partitioning technique with *SPFD-GR* will considerably degrade the solution quality. In Table 3, we compare the results of *SPFD-GR* for area minimization with and without partitioning for a set of medium-size circuits. For the circuits whose 4-*LUT* numbers are less than 200, we use a 2-way partition. For the others, we use 4-way partition. The initial circuits are obtained by the same method as that for Table 1. In Table 3, “Non-p” refers to the non-partitioned result; while “part” refers to the partitioned result. Both programs run 3 iterations for each circuit.

The total area of the partitioned results is 2.3% worse than that without partitioning, while the former is 2.6 times faster than the

later. For some circuits, the partitioned results actually have a smaller area. This is due to the fact that we adopt a kind of greedy algorithm for area minimization. For alu4, the partitioned one is slower than the non-partitioned one. The reason is the BDD variable order for one partition is very bad resulting in a much longer BDD calculation time. We also performed a similar experiment as in Table 3, except that we use a 2-way partition for all circuits. We do not list the data here because of the page limitation. In that experiment, the overall result is that the partitioned one is only 1.8% worse than the non-partitioned one and the run time of the former is around 1.9 times faster than the later.

Circuit	Area			CPU(s)	
	Init.	Non-p	Part	Non-p	Part
C432	154	129	124	54	31
C1908	127	118	117	209	22
x1	163	141	139	12	9
alu2	152	134	134	10	10
alu4	284	242	250	42	96
apex6	286	240	260	25	15
example2	135	114	122	7	6
x3	268	240	244	25	17
dalu	373	275	297	69	30
C5315	511	483	487	406	100
Total	2453	2116	2174	859	336
Ratio	1	86.3%	88.6%	2.6	1

Table 3 Impact of partitioning in area minimization

8 Conclusion and Future Work

This paper presents an *SPFD*-based global rewiring (*SPFD-GR*), which is capable of finding alternative wires far away from the target wire.

The experimental results show that the rewiring ability of the *SPFD-GR* method is 1.45 and 3 times that of the conventional *SPFD* method and an ATPG-based method (with a preliminary experimental flow), respectively. Using partitioning, the *SPFD-GR*-based algorithm scales well to the large circuits.

Our future work will be: i) applying *SPFD-GR* methods to post-placement logic synthesis of *LUT*-based *FPGAs* to get the delay reduction; ii) finding some heuristics to help us to choose candidate alternative wires more efficiently, so as to improve the speed of *SPFD-GR*; and iii) using BDD variable ordering to speed up the BDD calculation.

Acknowledgement

This work is partially supported by the Gigascale Silicon Research Center (GSRC) and the California MICRO program with Actel, Altera, Lucent and Xilinx. We thank Prof. Tim Cheng and Ric Huang of UC Santa Barbara for providing the ATPG rewiring engine, S. Yamshita, H. Sawada and A. Nagoya of NTT Corp., Japan, for their binary code of the original *SPFD* program [18] for experimental purpose. We also thank Sung Lim of UCLA for adapting his partitioning algorithm [8] for the needs of this work. We also thank Prof. Robert Brayton of UC Berkeley for his stimulating discussions on the *SPFD* technique during various GSRC workshops.

References

[1] L. A. Entrena and K.-T. Cheng. Combinational and Sequential Logic Optimization by Redundancy Addition and

Removal. *IEEE Transaction on CAD of ICS*, Vol. 14, No. 7, pp. 909-916, July 1995.

[2] R. K. Brayton. Understanding *SPFDs*: A New Method for Specifying Flexibility. In *International Workshop on Logic Synthesis*, 1997.

[3] C.-W. Chang, C.-K. Cheng, P. Suaris, and M. Marek-Sadowska. Fast Post-placement Rewiring Using Easily Detectable Functional Symmetries. In *Design Automation Conference*, p. 286-289, 2000.

[4] S.-C. Chang, K.-T. Cheng, N.-S. Woo, and M. Marek-Sadowska. Postlayout Rewiring using Alternative Wires. *IEEE Trans. on CAD ICS*, Vol. 16, No.6, p.587-96, June 1997.

[5] S.-C. Chang, L. V. Ginneken, and M. Marek-Sadowska. Circuit Optimization by Rewiring. *IEEE Transaction on Computers*, Vol. 48, No. 9, pp. 962-970 September 1999.

[6] P. Chong, Y. Jiang, S. Khatri, F. Mo, S. Sinha, and R. Brayton. Don't Care Wires in Logical/Physical Design. In *International Workshop on Logic Synthesis*, pp. 1- 9, 2000.

[7] J. Cong and Y. Hwang. Simultaneous Depth and Area Minimization in *LUT*-Based *FPGA* Mapping. *Proc. ACM 3rd Int'l Symp. on FPGA*, Feb. 1995, pp. 68-74.

[8] J. Cong and S. K. Lim. Edge Separability based Circuit Clustering With Application to Circuit Partitioning. *IEEE/ACM Asia South Pacific Design Automation Conference*, p. 429-434, 2000.

[9] J. Cong, J. Peck, and Y. Ding. RASP: A General Logic Synthesis System for *SRAM*-based *FPGAs*. In *Proc. ACM/SIGDA Int'l Symp. on FPGAs*, p. 137-143, Feb. 1996.

[10] R. Huang, Y. Wang, and K.-T. Cheng. LIBRA-a library-independent framework for post-layout performance optimization. In *International Symposium on Physical Design*, p.135-140, 1998.

[11] J. - M. Hwang, F. - Y. Chiang, and T.- T. Hwang. A Re-engineering Approach to Low Power *FPGA* Design Using *SPFD*. In *Design Automation Conference*, p. 722-725, 1998.

[12] Y.-M. Jiang, A. Krstic, K.-T. Cheng, and M. Marek-Sadowska. Post-layout rewiring for performance optimization. In *Design Automation Conference*, p.662-665, 1997.

[13] W. Kunz and P. R. Menon. Multilevel Logic optimization by implication Analysis", In *International Conference on Computer Aided Design*, p. 6-13.

[14] E. Sentovich, *et. al.* SIS: A System for Sequential Circuit Synthesis. Memorandum No. UCB/ERL M92/41, Dept. EECS, UC Berkeley, 1992.

[15] S. Sinha and R. K. Brayton. Implementation and Use of *SPFDs* in Optimizing Boolean Networks. In *International Conference on Computer Aided Design*, p. 103 – 110, 1997.

[16] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3.0. Technique Report, Dept. of ECE, Univ. of Colorado at Boulder, 1998.

[17] Y. Wu, W. Long and H. Fan. A Fast Graph-Based Alternative Wiring Scheme For Boolean Networks. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E83-A, No.6, p.1131-1137, 2000.

[18] S. Yamshita, H. Sawada and A. Nagoya. A New Method to Express Functional Permissibilities for *LUT* based *FPGAs* and Its Applications. In *International Conference on Computer Aided Design*, p. 254 – 261, 1996.