

SPFD-Based Global Rewiring

Jason Cong, Yizhou Lin, Wangning Long*

Department of Computer Science, UCLA

Los Angeles, CA 90095

{cong, yizhou, longwn}@cs.ucla.edu

ABSTRACT

This paper presents the theory and algorithm for SPFD-based global rewiring (SPFD-GR). SPFD-GR allows us to globally replace a target wire with some alternative wire possibly far away from the target. It successfully overcomes the limitations of the existing SPFD-based local rewiring algorithm (SPFD-LR), which can only replace a wire with another wire that has the same destination node. In order to perform SPFD-based global rewiring, we developed the theory and algorithm for solving a fundamental problem in SPFD-based rewiring: Given the 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? Combined with a state-of-the-art partitioning algorithm, SPFD-GR scales well to large circuits with good synthesis quality. Our SPFD-based rewiring algorithm is ideal for LUT-based FPGAs, where the node's internal function can be changed freely without any area or delay penalty. Extensive experimental results show that for LUT-based FPGAs, the rewiring ability of SPFD-GR (in terms of the number of wires that have alternative wires) is 1.45, and 3 times that of SPFD-LR and an ATPG-based rewiring algorithm (with a preliminary experimental flow), respectively, while the run time is quite acceptable. When applied to the post-mapping area reduction for large LUT-based FPGAs under circuit depth restriction, SPFD-GR achieves 17.1% average area reduction, with no or little delay increase.

Keywords

logical re-synthesis, FPGA synthesis, SPFD, SPFD-based global rewiring.

1. INTRODUCTION

Rewiring is a technique that replaces a wire with another wire to achieve performance improvement or area reduction. Recently, it has received increased attention due to the need for the interaction between logic synthesis and layout design to solve the timing closure problem. The existing rewiring approaches include the automatic test pattern generation (ATPG) based redundancy addition and removal [2] [5] [6] [7] [12] [14] [15], symmetry detection [4], and the SPFD (*Set of Pairs of Functions to be Distinguished*) [19][17] based algorithms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA '02, February 24-26, 2002, Monterey, California, USA.

Copyright 2002 ACM 1-58113-452-5/02/0002...\$5.00.

ATPG-based redundancy addition and removal is the earliest and widely used approach. It uses ATPG techniques to add a redundant wire (*alternative wire*) making 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 possibly "far away" from the target wire in the circuit. However, when applied to LUT-based FPGAs, it is hard for the ATPG-based rewiring methods to take the advantage of the flexibility of k -input look-up tables (which can implement any k -input Boolean function).

The SPFD-based rewiring algorithm was first proposed in [19] and applied to the LUT-based FPGA synthesis. The authors of [17] successfully applied the SPFD method to the technology-independent logic synthesis. The SPFD-based method has also been applied to floorplanning and placement of multi-level PLAs [8] and low-power designs for FPGAs [13].

The SPFD-based method can easily change a node's internal function, which makes it especially attractive for LUT-based FPGA synthesis. However, existing SPFD-based rewiring algorithms only find alternative wires *locally*, requiring the destination node of the alternative wire to be the same as that of the target wire. In this paper, existing SPFD-based rewiring algorithms generally are referred as SPFD-LR (SPFD-based local rewiring).

We present an SPFD-based *global* rewiring algorithm, SPFD-GR, which is capable of finding a global alternative wire whose destination node may be different from that of the target wire. We also apply it to LUT-based FPGA synthesis. Our main contributions are as follows:

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 (Section 5)?
2. We developed an SPFD-based global rewiring algorithm (SPFD-GR) using the concept of the dominators and node modification technique stated above, allowing global rewiring with the flexibility of changing internal node functions in the network to maximize the opportunity of rewiring (Section 4).
3. When combined with a state-of-the-art multi-level/multi-way partitioning algorithm [10], SPFD-GR scales well to large designs (Section 6).
4. Extensive experimental results show that the rewiring ability of SPFD-GR, in terms of the number of the target wires having alternative wire(s), is 1.45 and 3 times that of SPFD-

* The current email address of Dr. Long is longwn@aplus-dt.com

LR and an ATPG-based rewiring algorithm, respectively, for LUT-based FPGA synthesis. When applied to the post-mapping area reduction under circuit depth restriction for large FPGA designs, *SPFD-GR* achieves an average of 17.1% in area reduction with little or no delay increase (Section 7).

2. TERMINOLOGY AND DEFINITIONS

This section reviews some terminology and definitions. The circuits referred in this paper are combinational circuits. A sequential circuit can be treated as a combinational circuit by assuming the outputs and inputs of sequential elements as the primary inputs and outputs of the circuit, respectively. We assume that the input circuit is a mapped *K*-LUT network, meaning that each **logic cell** (or **node**) in the circuit has a single **out-pin** p_0 and up to *K* **in-pins** ($p_1, p_2, \dots, p_n, n \leq K$). Each pin is associated with a global logic function $g_0, g_1, g_2, \dots, g_n$, respectively, in terms of the primary inputs of the circuit. Each node has an internal logic function $g_0 = f(g_1, \dots, g_n)$ that defines the logic relationship between the out-pin and the in-pins of the node, and can be any *K*-input function.

For wire $q_1 \rightarrow q_2$, as shown in Figure 1, G_2 is its **source node** and G_3 is its **destination node**. **Transitive fanout nodes of pin** q_i are the nodes on the paths from q_i to a primary output (**PO**). **Transitive fanout nodes of a node** are the transitive fanout nodes of the node's out-pin. A **dominator** of pin q_i is a transitive fanout of q_i through which all the paths from q_i to POs must pass. A **dominator of a wire** is the dominator of the wire's destination pin. For example, in Figure 1, G_5 is the only dominator of pin p_1 , while both G_3 and G_5 are the dominators of pin q_2 as well as wire $q_1 \rightarrow q_2$.

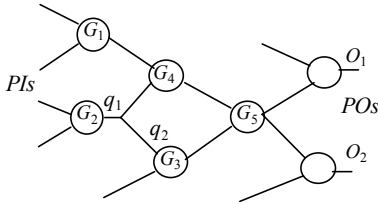


Figure 1. Illustration of dominator

Given a *function pair* (π_1, π_0) , $\pi_1 \neq 0$, $\pi_0 \neq 0$ and $\pi_1 \pi_0 \equiv 0$, function f is said to **distinguish** (π_1, π_0) if either one of the following two conditions is satisfied [17]:

$$\pi_1 \leq f \leq \bar{\pi}_0 \quad \text{or} \\ \pi_0 \leq f \leq \bar{\pi}_1$$

where $\pi_1 \leq f \leq \bar{\pi}_0$ can be understood as $f = 1$ when $\pi_1 = 1$, or $f = 0$ when $\pi_0 = 1$ [19]. The second condition can be interpreted in the same way.

An **SPFD** is a *Set of Pairs of Functions to be Distinguished*. It is usually represented as $P = \{(\pi_{11}, \pi_{10}), (\pi_{21}, \pi_{20}), \dots, (\pi_{m1}, \pi_{m0})\}$. Function f is said to **satisfy** an *SPFD* if f distinguishes all the function pairs in the *SPFD* set. An **atomic SPFD pair** is a function pair in which the two functions are the minterms which are expressed by the input functions of a node. An **atomic SPFD** is an *SPFD* that contains only one or several atomic *SPFD* pair(s). *SPFD* is described as a new way to express the “don’t-care” conditions and provide flexibility to implement a node [3].

3. REVIEW OF SPFD CALCULATION

First, we briefly review the method proposed in [19] for *SPFD* calculation. For an existing logic network, the calculation of the *SPFD*s usually consists of two steps:

- 1) Traverse the entire circuit from primary inputs (PIs) to primary outputs (POs) and calculate the logic functions at all pins.
- 2) Calculate the *SPFD*s backward from POs to PIs.

At each pin, the *SPFD* calculation methods is done according to the following 3 cases:

- a) At each PO, O_i , the *SPFD* has only one function pair, $P = \{(f_{i1}, f_{i0})\}$, where f_{i1} is the on-set function of O_i , and f_{i0} is the off-set function of O_i .
- b) At a node's out-pin, the *SPFD* is the union of its fanout pins' *SPFD*s.
- c) For the in-pins of a node, once its out-pin *SPFD* has been obtained, the in-pin *SPFD*s are obtained by decomposing its out-pin *SPFD* into atomic *SPFD* pairs and assigning the function pairs backwards to in-pins.

In [19], the *SPFD*-based rewiring algorithm is done in the following way: Given a target wire w_r with destination node G , as shown in Figure 2, if there is another alternative wire w_a whose function satisfies the *SPFD* $S = \{(\pi_{11}, \pi_{10}), (\pi_{21}, \pi_{20}), \dots\}$ assigned to w_r , w_a can be used to replace w_r . Note that the alternative wire found by this process must have the same destination node as the target wire. Therefore, we refer this operation as *local rewiring* (*SPFD-LR*).

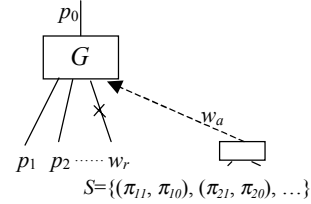


Figure 2. *SPFD*-based local rewiring

4. SPFD-BASED GLOBAL REWIRING ALGORITHM

In this section, we present our *SPFD*-based global rewiring algorithm, named *SPFD-GR*. The general global rewiring problem can be formulated as follows: Given target wire w_r with destination node G_D , can we add at most one wire to some node G_D in the network with possible modification of the internal function of G_D and other nodes in the circuit so that we can remove w_r while preserving functions at the network's primary-output? (See illustration in Figure 3.)

This problem can be divided into two cases: 1) When $G_D = G_1$, the problem can be solved using *SPFD-LR*; 2) When $G_D \neq G_1$ and the *SPFD* on w_r is not empty, we must determine how to select G_D and how to perform the logic transformation. This paper solves the second case (considerably more difficult), and our solution in fact also subsumes the first one.

We make use of the concept of dominators of a wire (e.g. G_D in Figure 3), which is widely used. The effect of removing the target wire must pass through its dominators to any PO. Therefore, after removing w_r , if we have a way to modify the internal

functions of some dominator G_D and possibly other nodes so that G_D 's out-pin SPFD is satisfied, then, we have a way to keep the logic functions of all POs unchanged. This idea can be formulated to the following algorithm.

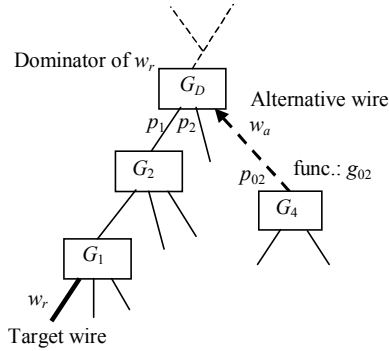


Figure 3. SPFD-based global rewiring

SPFD-based Global Rewiring Algorithm (SPFD-GR):

Given a target wire w_r and one of its dominators G_D ($G_D \neq G_1$) as shown in Figure 3,

- Step 1) Temporarily remove w_r from G_1 and re-calculate the output function of G_1 ;
- Step 2) Propagate G_1 's new output function through its transitive fanouts until reaching G_D ;
- Step 3) Try to modify G_D without wire addition so that f_D distinguishes the SPFD at G_D 's out-pin (using the theory and algorithm to be presented in Section 5.1). If successful, go to Step 6. Otherwise, go to the next step;
- Step 4) Try to modify G_D by adding a new wire w_a so that f_D distinguishes the SPFD at G_D 's out-pin (using the theory and algorithm to be presented in Section 5.2). If successful, go to Step 6; or try another candidate wire and repeat this step. If no candidate is left, go to the next step;
- Step 5) (Fail) Restore the functions of G_1 and its transitive fanouts until G_D . Return **fail**.
- Step 6) (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 destination node of the wire selected as the next target wire. Return **success**.

For a given target wire w_r , in order to find an alternative wire, the algorithm will go through all of its dominators one by one from the destination node of w_r to primary outputs until an alternative wire is found or exhaust all dominators.

The SPFD-GR algorithm, assumes that the target wire is given. The way to choose a target wire depends on the application.

* By removing a wire from a node, we mean to set the wire's logic value to either 0 or 1. At first we set the target wire as 1, and try to find an alternative wire under this situation. If we cannot find any alternative wires, we will set the target wire as 0 and repeat the process.

When our objective is to minimize the circuit area, we select a wire that enables deletion of a node or packing of the node with some other nodes. For example, if we can replace a wire which is the only fanout of some nodes, we can achieve area reduction.

To maximize area reduction, we developed an edge distribution heuristic. When we distribute an atomic SPFD pair of the output of a LUT to its inputs, we may have different input ordering. The authors of [19] use the natural ordering of a node's inputs. In this paper, we propose a fanout-oriented input ordering heuristic. We first choose the input edge that has the largest fanout number to assign an atomic SPFD. Therefore, the edge with fewer fanouts will have fewer atomic SPFDs. As a result, this edge is easier to be replaced by another one.

It is worth noting that in Figure 3, G_1 is also a dominator of w_r . Therefore, the SPFD-GR is also capable of doing local rewiring. The difference of both methods in local rewiring is that SPFD-GR "cares" whether the SPFD at the out-pin of a node can be satisfied, while SPFD-LR "cares" whether the SPFD at an input of a node can be satisfied by another node's out-pin function. For example, in Figure 3, we check whether the SPFD of G_D 's out-pin can be satisfied by a combination of p_1, p_2, \dots , and w_a . In Figure 2, we only verify whether the SPFD of input wire w_r can be satisfied by w_a . Therefore, SPFD-GR can find more local rewiring cases than SPFD-LR does. However, SPFD-LR is faster and covers enough local rewiring cases. Therefore we still use SPFD-LR to do local rewiring in our experiments.

5. THEORY OF SPFD-BASED GLOBAL REWIRING

As shown in the previous section, the key steps in the SPFD-GR algorithm are based on answering to the following 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 the answer to question 1 is "no", 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 follows:

Definition 1: Let $B = \{\beta_0, \beta_1, \dots, \beta_{N-1}\}$ ($N = 2^n$, n is the number of inputs of the node), where β_i ($0 \leq i \leq N-1$) is a *minimum product term* of the node's in-pin functions, called **MP-term**, i.e. β_i is in the form of

$$\begin{aligned} \beta_0 &= \bar{g}_1 \bar{g}_2 \cdots \bar{g}_n \\ \beta_1 &= \bar{g}_1 \bar{g}_2 \cdots g_n \end{aligned} \quad (1)$$

...

$$\beta_{N-1} = g_1 g_2 \cdots g_n$$

where g_i ($1 \leq i \leq n$) is the global function at the node's i -th input. Given a function pair (π_1, π_0) , $\alpha_i = (\pi_1 + \pi_0)\beta_i$ is called the **restricted MP-term**.

Given a function pair (π_1, π_0) . Let f be a logic function, $f_\alpha = (\pi_1 + \pi_0)f$ is called the **restricted function** of f . The restricted function and the original function have the following relationship:

Lemma 1: $\pi_1 \leq f \leq \bar{\pi}_0 \Leftrightarrow \pi_1 \leq f_\alpha \leq \bar{\pi}_0$ and $\pi_0 \leq f \leq \bar{\pi}_1 \Leftrightarrow \pi_0 \leq f_\alpha \leq \bar{\pi}_1$.

In other words, f distinguishes (π_1, π_0) if and only if f_α distinguishes (π_1, π_0) .

To choose a proper node function, we can try every combination of the *MP-terms*, i.e. $f = \sum_{i=0}^{N-1} k_i \beta_i$, where $k_i = 0$ or 1 ,

to check if f distinguishes the SPFD at the node's out-pin. However, the time complexity of this simple approach is too high $O(2^{2^n})$. For 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 $P = \{(\pi_1, \pi_0)\}$ and in-pin functions are g_1, g_2, \dots, g_n , where n is the input number of the node, if each non-empty restricted *MP-term* $\alpha_i = (\pi_1 + \pi_0)\beta_i$ satisfies *one* of the following conditions,

$$\begin{cases} \alpha_i \leq \pi_0 \text{ or} \\ \alpha_i \leq \pi_1 \end{cases} \quad (2)$$

Then there exists a function that distinguishes (π_1, π_0) . Moreover, f can be constructed as following,

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

where $N=2^n$; $k_i = 0$ if $\alpha_i \leq \pi_0$, and $k_i = 1$ if $\alpha_i \leq \pi_1$.

Proof: See [21].

Actually, Equation (3) gives us a way to construct the internal function of a node when the condition in Theorem 1 has been 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 [19] to construct the internal function. Examples can be found in [21].

In fact, we can show that the conditions in Theorem 1 are also necessary for a node to have an output function that distinguishes (π_1, π_0) . The proof is not included in this paper 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, $P = \{(\pi_{11}, \pi_{10}), (\pi_{21}, \pi_{20}), \dots, (\pi_{m1}, \pi_{m0})\}$, at a node's out-pin, and a function φ which distinguishes all the pairs in P , without loss of generality, suppose $\forall (\pi_{i1}, \pi_{i0})$ ($1 \leq i \leq m$), $\pi_{i1} \leq \varphi \leq \bar{\pi}_{i0}$. Let $\pi_1 = \sum_{i=1}^m \pi_{i1}$ and

$\pi_0 = \sum_{i=1}^m \pi_{i0}$. Then $f = f(g_1, g_2, \dots, g_n)$ distinguishes any pair $p \in P$

if f distinguishes (π_1, π_0) .

Proof: See [21].

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 φ is needed to classify the functions in each pair into φ 's on-set function and off-set function. Since the node's initial output function must distinguish the SPFD at its out-pin, we can simply use it as φ .

In our implementation, if the *SPFD* at a node's out-pin has several function pairs, we use Theorem 2 to combine them into a single pair.

5.2 Node Modification With Wire Addition

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 to the node.

Wire Addition Algorithm:

- Step 1) Calculate the *MP-term* set, $\mathbf{B} = \{\beta_0, \beta_1, \dots, \beta_{N-1}\}$ ($N = 2^n$), 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 $(\pi_1 + \pi_0)\beta_i \leq \pi_0$; \mathbf{B}_1 is the set whose members satisfy $(\pi_1 + \pi_0)\beta_i \leq \pi_1$; and \mathbf{B}_2 is the set whose members do not satisfy any conditions in (2). Let $\mathbf{B}_1 = \{\beta_{i_1}, \beta_{i_2}, \dots, \beta_{i_m}\}$ and $\mathbf{B}_2 = \{\beta_{s_1}, \beta_{s_2}, \dots, \beta_{s_k}\}$. If $\mathbf{B}_2 = \emptyset$, then we can successfully modify the node without adding a wire, and we can use the method in the previous subsection, return **success**. Otherwise, it is necessary to add a wire, and go to the next step.
- 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 $\beta_{s_j} \in \mathbf{B}_2$, let $\alpha_{s_j} = (\pi_1 + \pi_0)\beta_{s_j}$, calculate $r_0 = \bar{g}_{n+1}\alpha_{s_j}$ and $r_1 = g_{n+1}\alpha_{s_j}$. If both satisfy either $r_i \leq \pi_0$, or $r_i \leq \pi_1$ ($r_i \neq 0$), then we can add this wire. Go to the next step. Otherwise, find another candidate node and repeat this step, until all nodes have been tried, return **fail**.
- Step 3) Using the calculation results in Steps 1 and 2 to get the G 's output function that distinguishes (π_1, π_0) :

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

where $f_1 = \beta_{i_1} + \beta_{i_2} + \dots + \beta_{i_m}$, $\beta_{i_m} \in \mathbf{B}_1$, which is obtained in Step 1. For each k_i in the second term, $k_i = 1$ if $\alpha_{s_i} g_{n+1} \leq \pi_1$, otherwise $k_i = 0$. For each k_j in the third term, $k_j = 1$ if $\alpha_{s_j} \bar{g}_{n+1} \leq \pi_1$, otherwise $k_j = 0$.

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

Theorem 3: The wire addition algorithm is correct.

Proof: See [21].

As B_2 is a sub-set of B , which is usually very small, the checking procedure in the above algorithm does not consume too much time. In addition, B_1 and B_2 remain the same for any candidate node. Therefore, they can be re-used in the computation.

Example 1 (Node modification with wire addition): Given the in-pin functions g_1 and g_2 as shown in Figure 4, the SPFD at G 's out-pin p_0 is $SPFD_0 = \{(\pi_1, \pi_0)\}$, where $\pi_1 = x_1 + x_2$ and $\pi_0 = \bar{x}_1 \bar{x}_2$.

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

- 1) Condition checking (without wire addition):

$$\begin{aligned}\alpha_0 &= \bar{g}_1 \bar{g}_2 (\pi_1 + \pi_0) = \bar{x}_1 \bar{x}_2 + x_1 x_2 \\ \alpha_1 &= g_1 g_2 (\pi_1 + \pi_0) = x_1 \bar{x}_2 \leq \pi_1 \\ \alpha_2 &= g_1 \bar{g}_2 (\pi_1 + \pi_0) = \bar{x}_1 x_2 \leq \pi_1 \\ \alpha_3 &= g_1 g_2 (\pi_1 + \pi_0) = 0\end{aligned}$$

Thus, we can set $B_0 = \emptyset$, $B_1 = \{\alpha_1, \alpha_2\}$ and $B_2 = \{\alpha_0\}$. As $B_2 \neq \emptyset$, it is necessary to add a wire.

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

$$\begin{aligned}\bar{g}_1 \bar{g}_2 \bar{g}_3 &= \bar{x}_1 \bar{x}_2 \leq \pi_0 \\ \bar{g}_1 \bar{g}_2 g_3 &= x_1 x_2 \leq \pi_1\end{aligned}$$

- 3) The conditions are satisfied. Therefore we can add the wire from g_3 to G and obtain $g_0 = \bar{g}_1 g_2 + g_1 \bar{g}_2 + \bar{g}_1 \bar{g}_2 g_3$ that distinguishes (π_1, π_0) .

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

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

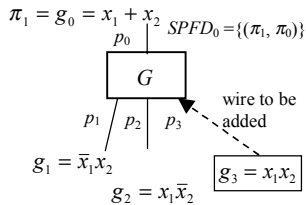


Figure 4. Illustration of node modification with wire addition

6. USE OF PARTITIONING FOR LARGE CIRCUITS

In the implementation of *SPFD-GR*, ROBDD is used to represent logic function and SPFDs. (We use the CUDD package[18].) 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* is usually dominated by the runtime of various BDD operations, including SPFD calculation and condition check.

For large circuits, it is usually impossible to directly build the BDDs for all node functions based on the global variables of the circuit. In our implementation, we successfully combined *SPFD-GR* with a state-of-the-art partitioning algorithm, which makes it possible to apply *SPFD-GR* to large circuits. In our synthesis flow, we first use a partitioning algorithm to partition a big circuit into

smaller blocks, and then do rewiring for each block separately. After all blocks have been processed, we combine the results to get the solution for the entire circuit.

The use of partitioning does not significantly impact the ability to finding alternative wires because most alternative wires are actually not very far from the target wire. An experiment in [20] shows that when applying an ATPG-based rewiring algorithm to some MCNC benchmark circuits, there are 96% alternative wires that are within distance two from the target wires (i.e., connected through at most two intermediated wires). Therefore, with a good partitioning algorithm, *SPFD-GR* can achieve a good rewiring quality while reducing the run time for large circuits. Our experimental results in the next section show that the area minimization with partition is only 2.3% worse than that without partitioning; with over 2.6X runtime reduction for some mid-size MCNC benchmark circuits. Therefore, we conclude that it is practical and efficient to combine our *SPFD-GR* with the circuit-partitioning algorithm when handling large designs.

We used a state-of-the-art, multi-level, multi-way partitioning program KPM/MESC [10] in our implementation. During our experiments, we noticed that the results are good when the logically related nodes are grouped into the same partition block.

In order to restrict the depth of the circuits, we calculate the depth of each node after partitioning. If we update the depth information for the entire circuit after each rewiring, it would result in long CPU time. So we only update the depth information inside a block after each rewiring change, and for the entire circuit after the rewiring for the block is done. We can ensure that the resulting depth after rewiring will not increase if there is no combinational feedback from an output of the block to one of its input (sometimes the feed back may pass through other blocks). However, our current partitioning algorithm does not guarantee this condition. Therefore, there may be a little possibility that the depth may increase (especially for some large circuits where the feedback is likely to happen). In our experiments in the next section, almost all of the circuits keep their depth or decrease depth.

7. EXPERIMENTAL RESULTS

We implemented the SPFD-based global rewiring algorithm (*SPFD-GR*) for the LUT-based FPGA synthesis and integrated it into the RASP system [11]. We performed three sets of experiments: 1) rewiring ability comparison with the existing SPFD-based local rewiring (*SPFD-LR*) and an ATPG-based rewiring algorithm (with a preliminary experimental flow), 2) area minimization results, 3) the impact of partitioning. These experiments are carried out on a Sun Ultra 60 workstation.

7.1 Comparison of Rewiring Ability

In our study, we use *rewiring ability* to compare two rewiring algorithms. Rewiring ability is defined as the *number of wires* having at least one alternative wire. The higher rewiring ability, the more rewiring choices an algorithm has.

Table 1 compares the rewiring ability of *SPFD-GR* with *SPFD-LR*, and an ATPG-based rewiring algorithm (with a preliminary experimental flow). We implemented the *SPFD-LR* algorithm according to [19]. For the ATPG-based rewiring, we use an ATPG rewiring engine that was used in [12][14], which was originally designed for the cell-based post-layout synthesis, with and without recursive learning, denoted as A0 (no recursive

Cir. Name	Total # of wires	Rewiring ability (# of target wires having alternative wires)				CPU time (s)			
		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
term1	244	55	60	62	87	1.6	5.9	0.8	1.4
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	8737	889	1079	2211	3202	421.1	2485.4	132.4	430.5
Ratio	1.00	10.2%	12.3%	25.3%	36.6%	0.98	5.77	0.31	1.00

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

learning) and A1 (learning level 1), respectively. To apply this ATPG-based rewiring engine, we decomposed the LUT-based network into a simple gate-based network that is fed into the rewiring engine. The engine then returns a set of alternative wire candidates. With this flow, the circuits fed into the rewiring engine are 2 to 4 times larger than the original LUT circuits. This may be one of the reasons that the rewiring engine runs slower in this

application than in [12][14]. Hence, Table 1 only gives us a preliminary comparison between SPFD-GR and the ATPG-based rewiring for LUT-based FPGAs.

All four programs traverse the entire circuit once, selecting every wire as a target wire, and trying to find alternative wires that satisfy the circuit depth restriction. The circuits used in Table 1 are 4-LUT networks obtained after running *script.rugged*, *Cutmap* [9], *Red_Removal* and *Greedy_Pack*. All these routines are available in SIS [16] and RASP [11]. Column 2 lists the number of wires in each circuit. Columns 3 ~ 6 show the *rewiring ability* of the four methods. Columns 7 ~ 10 show the run time of the four algorithms in seconds.

From Table 1, one can see that for LUT-based FPGA networks, ATPG rewiring algorithms, with or without, recursive learning can find alternative wires for 10.2% and 12.3% of the total wires, respectively. SPFD-LR and SPFD-GR can find alternative wires for 25.3% and 36.6% of the total wires, respectively. SPFD-GR is about 3 times slower than SPFD-LR, as fast as the ATPG-based algorithm without recursive learning used in this experiment, and 5.7X faster than ATPG-based algorithm with recursive learning. Reference [21] provides an example that shows SPFD-GR can find more target wires which have alternative wires than the other 3 algorithms.

7.2 Area Minimization Results

We applied SPFD-LR and SPFD-GR to a post-mapping area reduction for MCNC benchmarks and 5 large circuits (provided by one of our industry partners), whose size is up to 100K 2-input gates. Table 2 shows the results for 7 MCNC combinational circuits and 6 MCNC sequential circuits, and Table 3 shows the results for 5 large industrial circuits.

In Table 2, all initial circuits are synthesized and mapped by “*script.algebraic*,” *Cutmap*, and *Greedy_Pack*. We use a greedy

Circuit	Initial Circuits			SPFD-LR				SPFD-GR			
	Area	Depth	Delay(ns)	Area	Area Reduction	Delay(ns)	Delay Ratio	Area	Area Reduction	Delay(ns)	Delay Ratio
C1355	92	6	11.584	92	0.0%	11.76	1.02	76	17.4%	10.58	0.91
C1908	135	9	14.931	119	11.9%	13.476	0.90	108	20.0%	13.275	0.89
C2670	201	10	11.639	176	12.4%	12.569	1.08	177	11.9%	11.729	1.01
C3540	397	14	20.684	369	7.1%	20.121	0.97	355	10.6%	21.588	1.04
C5315	546	9	13.895	500	8.4%	14.115	1.02	478	12.5%	13.623	0.98
C6288	1021	33	49.155	903	11.6%	51.369	1.05	786	23.0%	51.105	1.04
C7552	525	12	17.4	444	15.4%	17.866	1.03	417	20.6%	17.013	0.98
s9234	385	8	13.651	336	12.7%	13.417	0.98	328	14.8%	12.35	0.90
s13207	878	10	18.424	775	11.7%	16.291	0.88	749	14.7%	16.789	0.91
s15850	1202	13	23.064	1076	10.5%	22.301	0.97	1014	15.6%	21.741	0.94
s35932	3209	3	6.834	3204	0.2%	7.272	1.06	3183	0.8%	6.709	0.98
s38417	3642	10	21.659	2883	20.8%	18.138	0.84	2851	21.7%	17.487	0.81
s38584	4443	9	16.002	3947	11.2%	16.466	1.03	3693	16.9%	15.263	0.95
Average					10.3%		0.99		15.4%		0.95

Table 2. Area minimization results for MCNC benchmarks

Circuit	Initial Circuits		SPFD-LR				SPFD-GR				
	Area	Depth	Area	Area Reduction	Depth	CPU(s)	Area	Area Reduction	Depth	CPU(s)	CPU ratio
Ind1	11051	13	10353	6.3%	14	2654.7	10037	9.2%	13	3384.6	1.27
Ind2	10553	81	9777	7.4%	81	4432.6	9312	11.8%	81	5924.7	1.34
Ind3	11432	56	8243	27.9%	54	4415.0	7961	30.4%	51	5411.4	1.23
Ind4	20200	141	17979	11.0%	141	6979.4	16719	17.2%	142	16119.3	2.31
Ind5	44922	27	39589	11.9%	27	16440.6	37336	16.9%	27	42161.8	2.56
Average				12.9%				17.1%			1.74

Table 3. Area minimization results for industry benchmarks

strategy to do the SPFD-GR based area minimization, which tries to remove as many wires as possible. At the end of each pass, *greedy_pack* [11] is called to further pack the circuit.

In Table 2, Columns 2, 3 and 4 are the areas (in terms of total number of 4-LUT nodes), the depths and the longest path delay of the initial circuits. Since depth does not increase when using *SPFD-LR* and *SPFD-GR* for rewiring, we do not report depth again. We pass the resulting circuits after SPFD to Quartus II for placement and routing. Then we get the longest delay from the Quartus result. Columns 5 and 9 are the area minimization results after *SPFD-LR* and *SPFD-GR* respectively. Columns 6 and 10 are the area reduction compared to the original circuits, and the results show that *SPFD-GR* achieves 15.4% area minimization while *SPFD-LR* get 10.3%. Columns 7, 8, 11 and 12 are the delay information of those circuits. The results show that the average delay is not increased after rewiring.

Table 3 lists the results for some large industrial circuits. Since these circuits cannot be fitted into any single APEX device for placement and routing (due to I/O limitation), we did not use Quartus to get the delay number. All the circuits go through a similar synthesis flow as in Table 2, except Ind5, which skips “*script.algebraic*” because it fails to pass the script after running for 20 hours on Sun Ultra 60.

We used the KPM/MESC partition program [10] 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 80 LUT nodes. For each circuit, the program runs for 6 iterations. Within each iteration, the program runs up to 3 times for each partition.

In Table 3, Columns 2 and 3 are the areas (in terms of total number of 4-LUT nodes), and the depths of the initial circuits. Columns 4, 5, 8 and 9 are the area minimization results after *SPFD-LR* and *SPFD-GR* respectively. It shows that *SPFD-GR* achieves 17.1% area minimization while *SPFD-LR* gets 12.9%. Columns 6 and 10 are the circuit depths. Columns 7, 11 and 12 show the CPU times and the ratio of these two algorithms respectively. The CPU time of *SPFD-GR* is within 2 times of that of *SPFD-LR*. The depth of Ind4 increased by 1 since we do not update depth information after each rewiring. It is explained in Section 6.

7.3 The Impact of Partitioning

A major concern is whether the use of the partitioning technique with *SPFD-GR* will considerably degrade the solution quality. In Table 4, 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 Table 1. In Table 4, “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 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, because the BDD calculation time is too great. We also performed a similar experiment as in Table 4, except that we used a 2-way partition for all circuits. In that experiment, the overall result shows that the partitioned one is

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 4. Impact of partitioning in area minimization

only 1.8% worse than the non-partitioned one, and the former is around 1.9 times faster than the later.

8. CONCLUSIONS

This paper presents an SPFD-based global rewiring algorithm (*SPFD-GR*), which is capable of finding alternative wires far away from the target wire. Theorems and algorithms are developed for node modification and global rewiring.

The experimental results show that the rewiring ability (in terms of the percentage of wires that have at least one alternative wire) of the *SPFD-GR* method is 1.45 and 3 times that of the existing SPFD method (*SPFD-LR*) and the ATPG-based method (with a preliminary experimental flow), respectively.

Using the KPM/MESC partitioning algorithm, the *SPFD-GR*-based algorithm scales well to the large circuits. The experimental results show that *SPFD-GR* gets 17.1% area reduction, 4.2% more than *SPFD-LR*.

In our future work, we plan to develop some heuristics to guide the strategy in selecting target wires and alternative wires. Thus, we can further speed up the *SPFD-GR* process and also apply this technique to get performance improvement. Since *SPFD-GR* can be a general tool in post-layout logic re-synthesis, we will also apply our engine to other applications such as routability improvement.

9. 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 Dr. Ric Huang of UC Santa Barbara for providing the ATPG rewiring engine; S. Yamashita, H. Sawada and A. Nagoya of NTT Corp., Japan, for their binary code of the original SPFD program [19] for experimental purpose; Dr. Sung Lim of UCLA for adapting his partitioning algorithm [10] for the needs of this work; and Prof. Robert Brayton of UC Berkeley for his stimulating discussions on the SPFD technique during various GSRC workshops.

10. REFERENCES

- [1] Altera, Quartus II Software Overview, <http://www.altera.com/products/software/quartus2/qts-index.html>.
- [2] 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.
- [3] R. K. Brayton. Understanding *SPFDs*: A New Method for Specifying Flexibility. In *International Workshop on Logic Synthesis*, 1997.
- [4] 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.
- [5] S.-C. Chang, M. Marek-Sadowska, and K-T Cheng. Perturb and Simplify: Multilevel Boolean Network Optimizer. *IEEE Trans CAD of ICAS*, Vol. 15, No. 12, Dec 1996, pp. 1494 - 1504.
- [6] S.-C. Chang, K.-T. Cheng, N.-S. Woo, and M. Marek-Sadowska. Postlayout rewiring using alternative wires. *IEEE Transaction on CAD of ICS*, Vol. 16, No.6, p.587-96, June 1997.
- [7] 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.
- [8] 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.
- [9] J. Cong, 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.
- [10] J. Cong, 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] W. Kunz and P. R. Menon. Multilevel Logic optimization by implication Analysis", In *International Conference on Computer Aided Design*, p. 6-13.
- [16] E. Sentovich, *et. al.* SIS: A System for Sequential Circuit Synthesis. Memorandum No. UCB/ERL M92/41, Dept. EECS, UC Berkeley, 1992.
- [17] 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, 1998.
- [18] Fabio Somenzi. CUDD: CU Decision Diagram Package Release 2.3.0. Technique Report, Dept. of ECE, Univ. of Colorado at Boulder, 1998.
- [19] S. Yamashita, 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.
- [20] Y. Wu, W. Long, and H. Fan. A Fast Graph-Based Alternative Wiring Scheme for Boolean Networks. In *IEEE 13th International Conference on VLSI Design (VLSI Design 2000)*, p.268-73, Jan. 2000.
- [21] K. C. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar. DAG-Map: Graph-Based FPGA Technology Mapping for Delay Optimization. In *IEEE Design & Test*, pp. 7-20, September, 1992.
- [22] J. Cong, Y. Lin, W. Long. *SPFD*-based Global Rewiring. In *UCLA CSD Tech. Report*. No.010043. Dec. 2001.