

# Post-Placement Rewiring and Rebuffering by Exhaustive Search for Functional Symmetries

Kai-hui Chang  
University of Michigan  
EECS Department  
Ann Arbor, MI 48109-2122  
changkh@umich.edu

Igor L. Markov  
University of Michigan  
EECS Department  
Ann Arbor, MI 48109-2122  
imarkov@umich.edu

Valeria Bertacco  
University of Michigan  
EECS Department  
Ann Arbor, MI 48109-2122  
valeria@umich.edu

## ABSTRACT

Separate optimizations of logic and layout have been thoroughly studied in the past and are well documented for common benchmarks. However, to be competitive, modern circuit optimizations must use physical and logic information simultaneously.

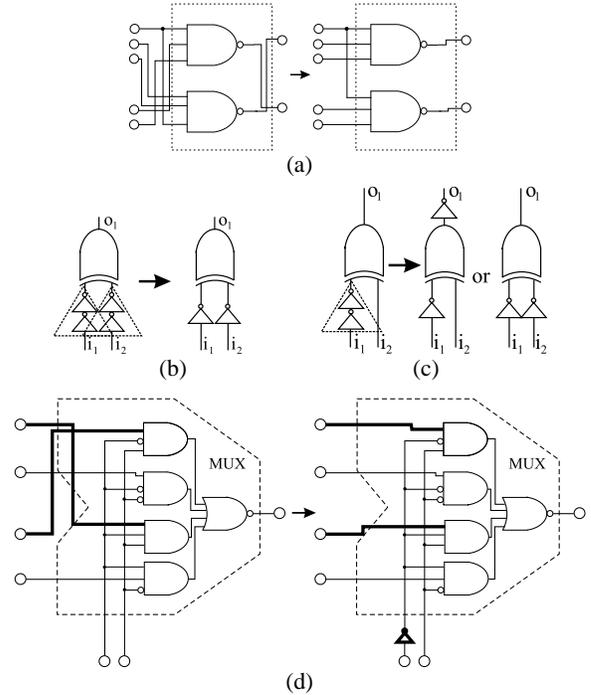
In this work, we propose new algorithms for rewiring and rebuffering — a post-placement optimization that reconnects pins of a given netlist without changing the logic function and gate locations. These techniques are compatible with separate layout and logic optimizations, and appear independent of them. In particular, when the new optimization is applied before or after detailed placement, it approximately doubles the improvement in wirelength.

Our contributions are based on exhaustive search for functional symmetries in sub-circuits consisting of several gates. Our graph-based symmetry finding is more comprehensive than previously known algorithms — it detects permutational and phase-shift symmetries on multiple input and output wires, as well as hybrid symmetries, creating more opportunities for rewiring and rebuffering.

## 1. INTRODUCTION

Power consumption and the speed of a circuit are two major metrics to evaluate the performance of a circuit — rewiring and rebuffering allow optimizing these design objectives. As illustrated in Figure 1, rewiring can reduce wirelength and improve congestion, while rebuffering allows one to replace some buffers with inverters to optimize timing and reduce power consumption. A key insight in our work is the importance of functional symmetries in rewiring and rebuffering. We propose new algorithms for these post-placement optimizations based on the knowledge of functional symmetries in small sub-circuits. A new symmetry detection technique is also proposed in this paper. It is more comprehensive than previous symmetry detection techniques in that it detects all possible input and output symmetries and uses a very compact representation of symmetries. Although in general, the new technique is potentially less scalable than existing techniques, its runtime in rewiring and rebuffering applications is reasonable.

For a given Boolean function, the simplest form of symmetry is the swap of two inputs that does not change the output under any circumstances. In some multi-output functions, outputs can be swapped simultaneously with inputs. Given that many common gates (NAND, NOR, XOR, etc.) are symmetric, pins of many standard cells can be swapped without changing circuit function. Performing such swaps after placement may shorten the wires connected to the swapped pins. This possibility has been known before, and sets of logically equivalent pins of a standard cell can be represented in the Cadence LEF (Library Exchange Format) files. However, LEF files do not carry other functional information and prevent detecting equivalent pins on different cells. Several publications discuss the detection of



**Figure 1: Rewiring and rebuffering examples: (a) multiple inputs and outputs are rewired simultaneously using pin-swap symmetry, (b) buffers on wires  $i_1$  and  $i_2$  are changed to inverters using phase-shift symmetry to reduce gate delay, (c) the buffer on wire  $i_1$  is replaced with an inverter using phase-shift symmetry and an extra inverter is inserted into either  $o_1$  or  $i_2$  to preserve logic correctness, (d) two inputs to a multiplexer are rewired by inverting one of the select bits. Bold lines represent changes made in the circuit.**

equivalent pins and uses in rewiring [11, 10, 13, 6], mostly focusing on two-variable input symmetries. The work in [8] extends symmetry detection to multi-variable input symmetries, but simultaneous permutations of inputs and outputs are not considered. Published symmetry detection algorithms typically manipulate circuit or BDD-based representations of Boolean functions, which in some cases limits detectable symmetries. Indeed, the topology of any AND-tree fails to carry all symmetries of the  $n$ -input AND function.

During rewiring, detecting fewer symmetries creates fewer possibilities for improving wirelength. While previous work considers input symmetries only, our comprehensive symmetry detection handles both input and output permutations, as well as their compositions. For example, the rewiring in the Figure 1(a) cannot happen

unless both input and output symmetries can be detected. “Phase-shift” symmetries considered in [8] can also be detected, and they can be used for rewiring or rebuffering to optimize timing. Examples of these applications are given in Figure 1(b), 1(c) and 1(d).

The post-placement rewiring and rebuffering approach proposed in our work is particularly convenient with min-cut placement algorithms, which in addition to cell locations produce a hierarchical collection of “placement bins” (buckets) that contain tightly connected cells. Other subcircuits suitable for rewiring and rebuffering include buckets of cells found by depth-first or breadth-first traversal starting from a cell. For rewiring, we exhaustively search for pin-swap symmetries and redundant pins within each bucket, rewire the netlist where possible, and proceed from smaller buckets to larger buckets. The experimental results show that this approach can reduce total wirelength by about 4.3%. Since this approach is orthogonal to other wirelength reduction techniques, it is an effective post-placement processing to reduce the wirelength of circuits. For rebuffering, we exhaustively search for phase-shift symmetries involving the buffered wires and replace the buffers with inverters if such a symmetry exists.

The remainder of the paper is organized as follows. Section 2 introduces the basic ideas and describes some relevant previous work on symmetry detection, circuit rewiring, and min-cut placement. In Section 3 the problem mapping and the algorithm of the symmetry detection technique are illustrated. Section 4 discusses the post placement rewiring algorithm used in this paper and proposed a new way to do buffer insertion using phase-shift symmetry. Experimental results are given in Section 5 and Section 6 concludes this paper.

## 2. BACKGROUND

The rewiring technique proposed in our paper is based on symmetry detection. Therefore in this section, some background on symmetry detection and related work is described. Previous work on post-placement rewiring and an a brief introduction on min-cut placement are also given.

### 2.1 Symmetries in Boolean Functions

One can distinguish semantic (functional) symmetries of Boolean functions from the symmetries of specific representations (syntactic symmetries). All syntactic symmetries are also semantic, but not vice versa. For example, no AND tree exhibits all permutational symmetries of the  $n$ -bit AND function for  $n \geq 3$ .

**DEFINITION 1.** *Functional (semantic) symmetries are transformations of inputs and outputs that do not change the functional relation between them.*

**DEFINITION 2.** *Syntactic symmetries are transformations that preserve a specific representation of the function (a circuit, a Boolean formula, a BDD, etc).*

**EXAMPLE 1.** *Consider the multi-output function  $z = x_1 \text{ XOR } y_1$  and  $w = x_2 \text{ XOR } y_2$ . The variable-permutation symmetries include: (1)  $x_1 \leftrightarrow y_1$ , (2)  $x_2 \leftrightarrow y_2$ , (3)  $x_1 \leftrightarrow x_2$ ,  $y_1 \leftrightarrow y_2$ , and  $z \leftrightarrow w$  (all swaps are performed simultaneously). In fact, all the symmetries of this function can be generated from combinations of the symmetries listed above. A set of symmetries with this property are called “symmetry generators”. For example, the symmetry “ $x_1 \leftrightarrow y_2$ ,  $y_1 \leftrightarrow x_2$ , and  $z \leftrightarrow w$ ” can be generated by applying the symmetries (1), (2) and (3).*

Much of previous work focuses on two-variable symmetries — the simplest and most common type, described below.

**DEFINITION 3.** *Consider two variables,  $x_i$  and  $x_j$ , of function  $F(\dots x_i, \dots, x_j, \dots)$ . They are symmetric if the function does not change when the variables are swapped:*

$$F(\dots x_i, \dots, x_j, \dots) = F(\dots x_j, \dots, x_i, \dots).$$

This type of symmetry is known as the *classical symmetry*, or the *non-skew non-equivalence symmetry*, denoted  $x_i!NEx_j$ . This symmetry is also denoted as  $F_{01} = F_{10}$  in [10].

There are other relationships among two-variable cofactors which lead to other symmetry types:  $F_{00} = F_{11}$  yields *non-skew equivalence symmetry*, and is denoted  $x_iEx_j$ . If one of the two cofactors are complemented, then two more symmetry relationships will be produced:  $F_{01} = \overline{F_{10}}$  is *skew non-equivalence symmetry*, and is also denoted  $x_i!NEx_j$ .  $F_{00} = \overline{F_{11}}$  is the *skew equivalence symmetry* and is denoted  $x_i!Ex_j$ .

Kravets et al.[8] generalized the symmetries discussed above by considering swaps of groups of ordered variables as higher-order symmetries. For example, if variables  $a, b, c$ , and  $d$  in the support of function  $f$  satisfy the condition:

$$F(\dots, a, \dots, b, \dots, c, \dots, d, \dots) = F(\dots, c, \dots, d, \dots, a, \dots, b, \dots)$$

then we say that  $f$  has a *second-order symmetry* between ordered variable groups  $(a, b)$  and  $(c, d)$ . Such higher-order symmetries are common in realistic designs. For example, in a 4-bit adder, all bits of the two input numbers can be swapped as groups (preserving the order of the bits), but no two input bits are symmetric by themselves. They also introduced phase-shift symmetry, which is defined below.

**DEFINITION 4.** *Traditionally, phase-shift symmetries are function-preserving transformations involving the inversion of one or more inputs, which nevertheless do not permute any of the inputs. This concept is generalized in this paper in that output symmetries involving inversion are also considered phase-shift symmetries. If input or output permutation is allowed, then it is a composite phase-shift symmetry, which consists of phase-shift and permutational symmetries. In this paper we will typically refer to composite phase-shift symmetries as just phase-shift symmetries, except for pure phase-shift symmetries which do not include permutations.*

**EXAMPLE 2.** *Consider again the multi-output function  $z = x_1 \text{ XOR } y_1$  and  $w = x_2 \text{ XOR } y_2$  given in Example 1. Besides the pin-swap symmetries discussed in Example 1, the following phase-shift symmetries also exist in the circuit: (1)  $x_2 \leftrightarrow y_2'$ , (2)  $x_1 \leftrightarrow y_1'$ , (3)  $x_2 \leftrightarrow x_2'$  and  $w \leftrightarrow w'$ , (4)  $x_1 \leftrightarrow x_1'$  and  $z \leftrightarrow z'$ . Among these symmetries, (1) and (2) are composite phase-shift symmetries because they involve both inversion and permutation of inputs, while (3) and (4) are pure phase-shift symmetries because only inversion of inputs and outputs are used.*

*Pure phase-shift symmetry generators can be used with permutational symmetry generators to generate more symmetries. For example, “ $y_2 \leftrightarrow y_2'$  and  $w \leftrightarrow w'$ ” can be generated with the combination of (3) in this example and (2) from Example 1.*

In this work, we restrict the transforms in functional symmetries to bipartite composite phase-shift symmetries of inputs and outputs, i.e., inputs cannot be swapped with outputs, but both types of variables can be permuted and/or phase-shifted simultaneously.

While handling a variety of symmetries is often desirable, the number of permutational symmetries on  $n$  variables can grow exponentially. Restricting the types of symmetries considered is one practical solution, and another is to develop compact representations. Aloul et al. [2] implicitly represented all symmetries of a graph by a small number of group generators. This representation has not been used to capture all functional symmetries of a Boolean function, but is employed in our work via the SAUCY tool [7].

Data structure used	Target	Symmetries detected	Main applications	Time complexity
BDD [10]	Boolean functions	All 1 <sup>st</sup> order input symmetries	Synthesis	$O(n^3)$
Circuit [6]	Gate-level circuits	1 <sup>st</sup> order input symmetries in supergates, opportunistically	Rewiring, technology mapping	$O(n)$
Graph (this work)	Both (with small number of inputs)	All input, output, phase-shift symmetries and all orders, exhaustively	Exhaustive small group rewiring	$\Omega(2^n)$

**Table 1: A comparison of different symmetry detection methods.** Currently known BDD and circuit-based methods can only detect a fraction of all symmetries in some cases while graph-based method (this work) can detect all symmetries exhaustively. Additionally, the symmetry-detection techniques in this work find all phase-shift symmetries as well as composite (hybrid) symmetries that simultaneously involve both permutations and phase-shifts. In contrast, existing literature on functional symmetries does not consider such composite symmetries.

## 2.2 Semantic & Syntactic Symmetry Detection

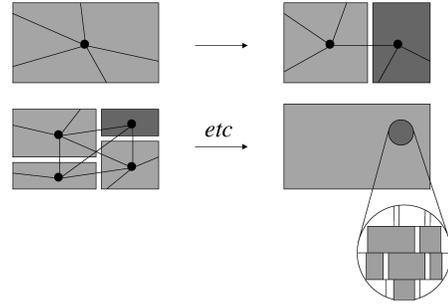
Symmetry detection in Boolean functions has been studied for a long time and has several applications, including technology mapping, technology-independent logic synthesis [9], BDD minimization [12], and circuit rewiring [6]. Methods for symmetry detection can be classified in three categories: BDD based, graph-based, and circuit-based. However, it is relatively difficult to find all symmetries of a Boolean function regardless of the representation used.

BDDs are particularly convenient for semantic symmetry detection because they support abstract functional operations, and research has been done on finding symmetries using BDD. One naive way to find two-variable symmetries is to compute the cofactors for each variable and for every pair of variables check if  $F_{01} = F_{10}$  or  $F_{00} = F_{11}$ . Recent research [10] indicates that symmetries can be found or disproved without computing all the cofactors independently and thus significantly speed up symmetry detection. However, work on BDD-based symmetry detection has been limited to swaps of variables and swaps of groups of variables. This is probably because the single-output nature of BDD made symmetry detection involving multiple outputs more difficult. Another problem is that the symmetries found by this method are often enumerated and this list is not very compact. For example, if there is a symmetry group involving five inputs, this method will enumerate all  $5!=120$  permutations.

Graph-based symmetry detection methods rely on efficient algorithms for the graph automorphism problem (i.e., finding all symmetries of a given graph). They construct a graph whose symmetries faithfully capture the symmetries of the original object, find its automorphisms (symmetries) and map them back to the original object. Aloul et al. [2] proposed a way to find symmetries for SAT clauses using this approach, which was very effective. The symmetry detection algorithm proposed in our work is inspired by their work.

Circuit-based symmetry detection methods assume an existing circuit for the function in question and usually convert it to a more regular form, where symmetry detection is more practical and efficient. For example, Wang et al. [13] transforms the circuit to NOR gates. Chang et al. [6] use a more elaborate approach by converting the circuit to XOR, AND, OR, INV and BUF first, and then partition the circuit so that each subcircuit is fanout free. Next, they form “supergates” from the gates and detect symmetries for those supergates.

A comparison of BDD-based symmetry detection [10], circuit-based symmetry detection [6] and the method proposed in this paper is summarized in Table 1.



**Figure 2: High-level overview of min-cut placement.** The rectangles represent bins that partition the layout region, and the lines represent wires. Movable objects assigned to a given bin are tentatively assigned to the geometric center of the bin.

## 2.3 Graph-based Algorithms

Darga et al. [7] have recently improved symmetry detection algorithms used by older software. Their symmetry detector Saucy and finds all symmetries of a given colored undirected graph. To this end, consider an undirected graph  $G$  with  $n$  vertices, and let  $V = \{0, \dots, n-1\}$ . Each vertex in  $G$  is labeled with a unique value in  $V$ . A permutation on  $V$  is a bijection  $\pi: V \rightarrow V$ . An automorphism of  $G$  is a permutation  $\pi$  of the labels assigned to vertices in  $G$  such that  $\pi(G) = G$ ; we say that  $\pi$  is a structure-preserving mapping or symmetry. The set of all such valid relabellings is called the automorphism group of  $G$ . A coloring is a restriction on the permutation of vertices – only vertices in the same color can map to each other. Given  $G$ , possibly with colored vertices, Saucy produces symmetry generators that form a compact description of all symmetries. Saucy is available at: <http://vlsicad.eecs.umich.edu/BK/SAUCY/>

## 2.4 Post-Placement Rewiring

Since rewiring using the symmetries detected will not affect the function of the circuit, it can be used to optimize circuit characteristics. Some rewiring examples are illustrated in Figure 1(a) and 1(d). Here the goal is to reduce wirelength, and swapping symmetric input and output pins accomplishes this.

Chang et al. [6] use the symmetry detection technique described above to optimize delay, power, and reliability. In general, the symmetry detection in their work is done opportunistically rather than exhaustively. Besides rewiring, their work uses symmetries for logic restructuring. Experimental results show that their approach can achieve these goals effectively using the symmetries detected. However, they cannot find the rewiring opportunity in Figure 1(a) and 1(d) because their symmetry detection technique lacks the ability to detect output and phase-shift symmetries.

## 2.5 Min-Cut Placement

Min-cut placement is a particularly convenient framework for rewiring because it identifies small sub-circuits that are good candidates for pin permutations. Min-cut placement uses a top-down approach to decompose a given placement instance into smaller instances by sub-dividing the placement region, assigning modules to subregions and cutting the netlist hypergraph. In this context a *placement bin* is used to represent (i) a placement region with allowed module locations, (ii) a collection of circuit modules to be placed in this region, (iii) all signal nets incident to the modules in the region, and (iv) fixed cells and pins outside the region that are adjacent to modules in the region. The top-down placement process can be viewed as a sequence of passes where each pass examines all bins and divides some of them into smaller bins. Eventually, bins become

so small that individual standard cells can be placed by exhaustive enumeration or branch-and-bound. The most commonly used technique is to divide the bins according to balanced min-cut partitioning algorithms. A high-level overview on min-cut placement is given in Figure 2. In the figure, each rectangle is a bin that partition the layout region and wires are represented by lines. Min-cut partition is used at each step until the bins are small enough. At that time, exhaustive enumeration of cells is used, as the right bottom circle shows.

In this work we use the top-down placer Capo [3, 1]. The bins created by Capo contain tightly connected cells and are used for symmetry detection and rewiring.

### 3. EXHAUSTIVE SEARCH FOR FUNCTIONAL SYMMETRIES

The symmetry detection method presented in our work can find all input, output, multi-variable and phase-shift symmetries including composite (hybrid) symmetries. It relies on symmetry detection of graphs, thus the original Boolean function must be converted to a graph first. After that, symmetry detection is used on the graph, and then the symmetries found are converted back to symmetries of the original Boolean function. This section describes the mapping from a Boolean function to a graph and explains how to use it to find symmetries of the Boolean function.

#### 3.1 Problem Mapping

To reduce functional symmetry-detection to the graph automorphism problem, we represent Boolean functions by graphs. Such a construction must admit a one-to-one mapping between the functional symmetries and graph symmetries. Clearly, the graph should have the following properties: (1) the inputs and outputs of the function must be represented in the graph and are permutable, (2) the graph must be unique for each Boolean function. The following constructs are used to achieve these goals.

1. Each input and its complement are represented by two vertices in the graph, and there is an edge between them to maintain Boolean consistency (i.e.  $x \leftrightarrow y$  and  $x' \leftrightarrow y'$  must happen simultaneously). These vertices are called *input vertices*.
2. Outputs are handled similarly to inputs, and the vertices are represented by *output vertices*.
3. Each minterm and maxterm of the Boolean function is represented by a *term vertex*. Since minterm and maxterm representations of a Boolean function are canonical, the graph built is also canonical.
4. We introduce an edge from every minterm vertex to the output and an edge from every maxterm vertex to the complement of the output. These edges are used to maintain the relationship between terms and outputs.
5. We introduce an edge between every term vertex and every input vertex or its complement, depending on that input is 1 or 0 in the term. It is used to maintain the relationship between terms and inputs.
6. Since inputs and outputs are bipartite permutable, all input vertices have the same color and all output vertices have another color.
7. All term vertices have the same color.

The idea behind this construction is that if an input vertex swaps with another input vertex, the term vertices connected to them will

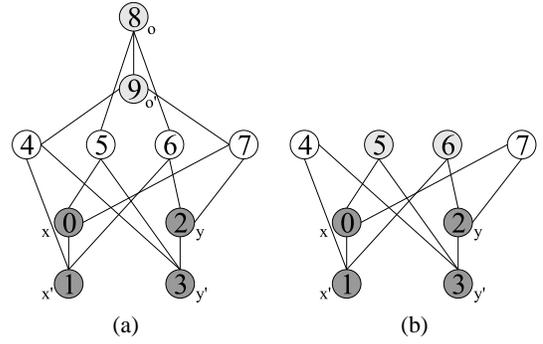


Figure 3: Mapped graph of a 2-input XOR gate: (a) original graph, (b) modified graph for faster symmetry detection.

also need to be swapped. However, since there are edges between term vertices and output vertices, the swapping is restricted to the following situations: (1) the swapping of term vertices does not affect the connections to output vertices, which means the outputs are not changed, (2) due to the connections between term vertices and output vertices, swapping term vertices may also require swapping output vertices, which captures output symmetries. The complete proof of correctness will be added in the full version of the paper.

Figure 3(a) gives an example of the mapped graph for a 2-input XOR gate. In the example,  $x$  and  $y$  are the first and the second inputs, and  $o$  is the output. The numbers in the circles are vertex indice assigned by the following rule: Suppose there are  $n$  inputs and  $m$  outputs. The  $i$ th input has number  $2i$ , while its complement is  $2i + 1$ . There are  $2^n$  terms, and the  $i$ th term is numbered  $2n + i$ . The  $i$ th output is numbered  $2n + 2^n + 2i$ , while its complement is numbered  $2n + 2^n + 2i + 1$ .

The symmetry detector Saucy [7] used in this work runs faster when there are more colors in the graph. Therefore if output symmetries do not need to be detected, a modified version of the graph can be used to detect input symmetries faster. It is constructed similarly to the previous graph, except that no output vertices are used. Assume that a “pattern” is a set of output vertices in the full graph that are connected to the given term vertex, the term vertices are colored according to their output patterns — each pattern has its own color. Figure 3(b) illustrates the modified graph for the two-input XOR function considered in our earlier examples.

All the minterms and maxterms of the Boolean function are used in the graph because we are focused on fully specified Boolean functions. However, it is also possible to extend our work to partially specified Boolean functions. Since we are not using such extensions in rewiring, they are not going to be discussed further.

#### 3.2 Symmetry Detection Algorithm

Our algorithm is given in Figure 4.  $N$  is the number of inputs,  $O$  is the number of outputs,  $V_i$  is a vertex, procedure *create*( $V_i$ ) creates a vertex, and procedure *connect*( $V_1, V_2$ ) adds an edge between  $V_1$  and  $V_2$ . *Connect\_ports*( $V_i$ ) connects a term vertex  $V_i$  to input/output vertices according to mapping rules 4 and 5. Each call to *Saucy*() returns symmetry generators in  $A$ , and *reverse\_map*( $A$ ) maps  $A$  back to symmetries in inputs/outputs.

Since the output of every input combination needs to be calculated, and there are  $2^n$  combinations, the time complexity of this algorithm is  $\Omega(2^n)$ .

#### 3.3 Discussion

Compared with other symmetry detection methods, the symmetry detection method proposed in our work has the following advantages: (1) it can detect all possible input and output symmetries

```

1 for i from 0 to  $2N+2^N+2O-1$ 
2   create  $V_i$ ;
3 for i from 0 to  $N-1$ 
4   connect( $V_{2i}, V_{2i+1}$ );
5 for i from 0 to  $O-1$ 
6   connect( $V_{2N+2^N+2i}, V_{2N+2^N+2i+1}$ );
7 for i from 0 to  $2^N-1$ 
8   connect_ports( $V_{2N+i}$ );
9  $A = \text{Saucy}()$ ;
10  $\text{symmetries} = \text{reverse\_map}(A)$ ;

```

Figure 4: Our symmetry detection algorithm.

```

1 repeat using different subcircuits
2   foreach subcircuit
3     bucket  $\leftarrow$  cells in subcircuit;
4     ckt=generate_circuit(bucket);
5     sym=symmetry_detect(ckt);
6     wl = wire_length();
7     rewire(sym);
8     nwl = wire_length();
9     if (wl < nwl)
10      unrewire(sym);

```

Figure 5: Our rewiring algorithm.

of a function, including multi-variable, higher-order and phase-shift symmetries, (2) symmetry generators are used to represent the symmetries and are very compact, and the relationship between input and output symmetries is very clear. These characteristics make the use of the symmetries easier than other methods which enumerate all symmetry pairs.

## 4. POST-PLACEMENT REWIRING AND REBUFFERING

This section describes two important techniques for post-placement optimization – rewiring and rebuffering. Rewiring uses symmetries detected from exhaustive search of extracted subcircuits to reduce wirelength or optimize timing. An innovative approach to buffer insertion using phase-shift symmetries is also proposed in this section.

### 4.1 Rewiring

After placement, symmetries can be used to rewire the cells to reduce the wirelength without changing the function of the circuit. It is achieved by exhaustive search and rewiring of functional symmetries found in subcircuits. Currently, subcircuits are extracted in the following order: (1) four passes from depth-first traversal of cells. Buckets containing 1, 2, 3 and 4 cells are used in each pass respectively, (2) two passes from breadth-first traversal of cells. Buckets containing 3 and 4 cells are used in each pass, (3) one pass using the cells in every half-bin produced by Capo, (4) one pass using the cells in every bin produced by Capo.

The algorithm for rewiring is given in Figure 5.

*Generate\_circuit* produces a circuit from cells in the bucket by the following rule: Suppose the circuit generated is called *ckt*. If the driver of a wire is not in *ckt*, it will become *ckt*'s input. If a wire in *ckt* drives a wire which connects to some ports outside *ckt*, it becomes *ckt*'s output.

*Symmetry\_detect* uses the symmetry detector discussed in the previous section and returns symmetries. If the symmetry group contains more than 6! permutations, 1000 random permutations are used instead of enumerating all of them. *Wire\_length* returns the wirelength of the circuit. The procedure *rewire* performs rewiring of the circuit according to the symmetry, and *unrewire* undoes the rewiring.

The reason why multiple passes with different sizes of buckets are

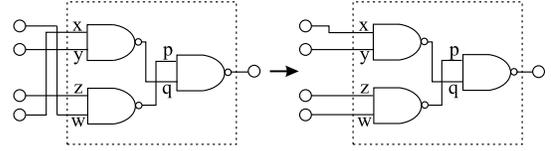


Figure 6: A rewiring opportunity for  $p$  and  $q$  that cannot be detected by only considering one bucket.

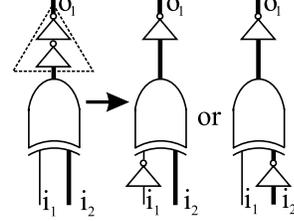


Figure 7: Rebuffering example: The bold line represents a critical path. The buffer on wire  $o_1$  is replaced with an inverter to reduce delay. An inverter is inserted into either  $i_1$  or  $i_2$  to preserve the correctness of logic.

used is that some symmetries in small buckets cannot be detected in larger buckets. For example, in Figure 6, if the bucket contains all the gates, only symmetries between  $x, y, z$  and  $w$  can be detected, and the rewiring opportunity for  $p$  and  $q$  will be lost. By using multiple passes for symmetry detection, more symmetries can be extracted from the circuit. It is noteworthy to know that if the buckets are chosen to be disjoint, it is very easy to parallelize symmetry detection and rewiring of the buckets.

The rewiring algorithm can be easily extended to utilize phase-shift symmetry: If the wirelength is shorter after the necessary inverters are inserted or removed, then the circuit is rewired.

Another application of rewiring is to reduce the delay on critical paths in order to shorten the clock cycle. The rewiring techniques proposed in our paper can be used for timing optimization if an incremental static timing analysis engine is available that allows us to quickly evaluate the effect of small changes without recomputing everything from scratch.

### 4.2 Rebuffering

Modern timing optimization heavily relies on buffer insertion, especially in long wires. While a single inverter is often sufficient to amplify the signal, logic correctness requires inserting a whole buffer, or two inverters, doubling gate delay, power consumption and area. Removing this overhead is very attractive, but one must find another way to compensate for the single inversion.

Phase-shift symmetries can be used to serve this purpose. Phase-shift symmetries are symmetries that involve inversion of pins. Therefore if buffer insertion is considered for a given wire, we can search for phase-shift symmetries involving inversion of that wire, and then use the symmetry found to replace some buffers with inverters, saving area as well as improving power consumption and delay. For example, in Figure 7, if buffer insertion for  $o_1$  is necessary, we can search for symmetries involving that wire. In the example, the fol-

```

1  $w \leftarrow$  wire for buffer insertion;
2 extract subcircuits with  $w$  as one of the outputs;
3 foreach ckt  $\in$  subcircuits
4   sym=symmetry_detect(ckt);
5   if (sym involves inversion of  $w$ )
6     insert inverter on  $w$ ;
7   rewire using sym;

```

Figure 8: Our proposed algorithm for inverter insertion.

lowing symmetries are found: If either  $i_1$  or  $i_2$  is complemented,  $o_1$  will be inverted. Therefore we can replace the buffer on  $o_1$  with an inverter. To preserve the correctness of logic, other pins are also rewired according to the symmetry, and there are multiple choices to rewire the circuit. The bold line in the figure represents the critical path. Since an inverter can be inserted into either  $i_1$  or  $i_2$ , and  $i_1$  is non-critical, it will be a better choice. However, if the wirelength of  $i_2$  is also long, it may be beneficial to insert the inverter into  $i_2$ . Such a flexibility from symmetry produces more optimization opportunities – inverters can be inserted into different nets in equivalent ways. The results of different rewiring choices are shown on the right of Figure 7. Another example which uses composite symmetry is given in Figure 1(d): One of the select pins is inverted and two data lines are swapped. Such composite symmetries are very useful in practice – inserting or removing an inverter on a select wire may allow permuting data wires, which may reduce wirelength and congestion.

The algorithm to replace buffer insertion with inverter is given in Figure 8. If the wire has only one fanout, then symmetry detection using that wire as an input can also be tried. In this situation, line 2 of the algorithm is changed to “with  $w$  as one of the inputs.”

### 4.3 Discussion

As seen from our empirical data in the next section, typical logic circuits offer a number of opportunities for rewiring and rebuffering. When these optimizations are applied together, timing-critical nets can be rewired and rebuffered to optimize timing, while the remaining nets can be rewired and rebuffered to optimize wirelength and congestion.

Rewiring and rebuffering are both effective methods to optimize timing. Another commonly used timing optimization technique is gate and wire sizing: The sizes of specific gates or wires are carefully chosen so that signal delay in wires can be balanced with gate delay, and the gates have enough capability to drive the wires. Rewiring and rebuffering may invalidate the optimality of gate and wire sizing solutions, but they can be employed before gate and wire sizing and achieve better optimization than using the sizing techniques alone.

## 5. EXPERIMENTAL RESULTS

Our implementation is written in C++ and integrates the search for symmetries in multi-output Boolean functions extracted from small groups of gates. It reads the *.blif* file of the circuit, the GSRC bookshelf *.nets*, *.nodes* files describing the netlist, the *.pl* placement file and *bin information* generated by Capo. After rewiring, it writes a text file indicating ports rewired. It is also able to generate rewired *.blif* and *.nets* files.

The testcases are selected from ITC99, ISCAS and MCNC benchmarks. To better reflect modern VLSI circuits, we chose the largest testcases from each benchmark suite, and added several small and medium ones for completeness. Our experiments use the min-cut placer Capo. Wirelength reduction is calculated after rewiring against the original wirelength after placement using half-perimeter wirelength. The platform used is Fedora 2 Linux on a Pentium-4 workstation running at 2.26GHz with 512M RAM. The numbers are averages of 5 independent runs.

We convert every testcase from BLIF to the Bookshelf placement format (*.nodes* and *.nets* files). For this, we enhanced the script previously available online in the Place-Utils entry of the GSRC Bookshelf [4, 5], and posted the new converter “blif2book.exe” at the same location. The command line option “+real\_size” is used in blif2book.exe to mimic realistic designs by the following rules: (1) all cell heights are 1, (2) latches have width 6, (3) inverters have width 1, buffers have width 2, (4) 2-input NAND/NOR gates have width 2, AND/OR/ have width 3, XOR/XNOR have size 5, (5) for a

Benchmark	Number of subcircuits	Symmetries				
		Input	Phase-shift input	Output	Phase-shift output	Input and output
alu2	1075	1054	138	253	141	220
alu4	18137	18130	134	1004	136	1003
b02	156	144	20	24	17	23
b10	1199	1088	163	207	135	176
b17	211250	202531	25164	34301	18477	26293
C5315	20980	19850	9213	5322	4606	4270
C7552	29745	27520	12440	7715	6696	6071
dalu	18182	16956	7058	3625	2881	3204
i10	16367	15802	4701	4065	3257	2867
s38417	148996	133442	79453	68769	62688	65288
s38584	129971	124644	58650	37631	31355	35655
Average	100%	94%	28%	23%	17%	20%

Table 2: Number of symmetries found in benchmark circuits.

cell with  $N$  inputs ( $N > 2$ ) and  $B$  lines in the truth table, the width is  $B + N$ , (6) pins are distributed evenly through each cell.

The first experiment shows the symmetries found from the subcircuits, and the results are summarized in Table 2. In the table, “number of subcircuits” is the number of subcircuits extracted from the benchmark for symmetry detection. “Input” is the number of subcircuits that contain input symmetries, “phase-shift input” is the number of subcircuits that contain phase-shift input symmetries. “Output” and “phase-shift output” are similar. “Input and output” are subcircuits that contain symmetries involving both inputs and outputs. Although experiments on timing optimization have not been conducted, the probability of finding phase-shift symmetries for inverter insertion can still be observed from this result: Phase-shift symmetry appears in 28% of subcircuit inputs and 17% of subcircuit outputs, which suggests that it is possible to find phase-shift symmetry for inverter insertion. From the results, it can also be observed that although output symmetries do not happen as often as input symmetries, its number is not negligible and rewiring techniques should take output symmetries into consideration.

Chang et al. [6] use symmetry to optimize timing and power, so their results are not directly comparable with our work. However, the number of symmetries detected from the circuits can be compared and are summarized in Table 5. The numbers in Chang’s work are from the experimental results in their paper. In the table, “largest inputs” is the largest number of inputs that appear in their supergates for symmetry detection. Their runtime includes the time to perform symmetry detection and rewiring. The “number of symmetries detected” in “this work” are limited to pin-swap input symmetries because Chang can only detect this type of symmetries, and it is calculated by subtracting the number of phase-shift input symmetries from the number of input symmetries. However, since symmetry generators are used in this work, the number of symmetries detected in this work can be potentially much larger than the numbers shown in the table. Note that the authors of [6] use a Sun Ultra10 workstation with 128M memory for their experiments, which is much slower than the machine used in our paper. Table 5 indicates that our algorithm finds more symmetries than the work in [6], mostly because the supergates used in [6] are very limited in size. We believe that our method can find all symmetries found by the methods from [6], as well as some additional symmetries. On average, we can detect twice as many symmetries than their method using additional time.

The second experiment compares the wirelength reduction gained from rewiring and detailed placement. It also compares the wirelength reduction of rewiring before and after detailed placement. The maximum number of inputs allowed for symmetry detection is 16 in this experiment. From the results, it is found that our method can effectively reduce wirelength by about 4.3%, which is comparable to the improvement due to detailed-placement. That the wirelength reduction is almost the same when rewiring is used before and after

Benchmark	This work		Chang's work [6]		
	Number of symmetries	Runtime (seconds)	Number of symmetries	Largest inputs	Runtime (seconds)
alu2	916	8.4	760	7	3.5
alu4	17996	77.8	1827	12	14.2
C5315	10637	9	2977	9	5.6
C7552	15080	9.8	2147	7	5.5
i10	11101	62.2	4472	11	11.3
s38417	53989	47	18579	21	81.6
Average	18286.5	35.7	5127	11.17	20.28

**Table 3: A comparison of our work and Chang et al [6] in terms of runtime and symmetries found.**

Benchmark	Wirelength	Wirelength reduction		Runtime (seconds)	
		Rewiring	Detailed placement	Rewiring	Capo
alu2	5532.83	7.11%	10.93%	8.4	2.2
alu4	40064.93	9.51%	4.69%	77.8	22.2
b02	142.9	8.29%	0%	0.5	2.6
b10	1556.7	5.2%	3.98%	2.2	3.8
b17	364912.8	2.93%	2.29%	648.79	416.6
C5315	35667.36	1.9%	2.31%	9	21.6
C7552	46758.46	2.02%	2.36%	9.8	31
dalu	23385.26	3.47%	4.32%	5.8	17.6
i10	54782.3	2.36%	2.95%	62.2	20
s38417	129967.6	1.96%	2.05%	47	208
s38584	174475.8	2.49%	2.25%	152	186.6
Average	79749.72	4.29%	3.47%	93.04	84.75

**Table 4: Performance and runtime comparisons between rewiring and detailed placement.**

detailed placement shows that wirelength reduction from rewiring is independent of detailed placement and can be used as an addition to detailed placement. Furthermore, the runtime of rewiring is close to the runtime of the placer, which shows that our method is efficient and effective for wirelength reduction. The results of the experiments are shown in Table 4 and Table 5.

The third experiment shows the relationship between the number of inputs allowed in symmetry detection, wirelength reduction, and runtime. Since our symmetry detection method is most efficient with small number of inputs, this relationship represents the trade-off between performance and runtime. The results indicate that the longer the rewiring program runs, the better the reduction will be. However, most improvement occurs with small number of inputs and can be achieved quickly. Given half the runtime of the placer, about 4% wirelength reduction can be achieved. The results of the experiment are shown in Table 6.

## 6. CONCLUSIONS

In this paper we proposed a new exhaustive search for functional symmetries and applied it to small subcircuits of common circuit benchmarks in the context of post-placement rewiring. We also developed a novel approach to rebuffering based on phase-shift and

Benchmark	Wirelength reduction		Runtime (seconds)	
	Before detailed placement	After detailed placement	Before detailed placement	After detailed placement
alu2	7.09%	7.11%	7.2	8.4
alu4	8.82%	9.51%	70.8	77.8
b02	8.29%	8.29%	0.5	0.5
b10	5.19%	5.2%	1.6	2.2
b17	3.06%	2.93%	585.79	648.79
C5315	1.92%	1.9%	9	9
C7552	2.16%	2.02%	9.6	9.8
dalu	3.52%	3.47%	5.6	5.8
i10	2.49%	2.36%	56.2	62.2
s38417	2.02%	1.96%	45.2	47
s38584	2.55%	2.49%	149.4	152
Average	4.29%	4.29%	85.54	93.04

**Table 5: The impact of rewiring before and after detailed placement.**

Number of inputs allowed	Runtime (seconds)	Wirelength reduction
2	3.25	1.52%
3	4.8	2.83%
4	7.27	3.3%
5	11.18	3.66%
6	15.2	3.8%
7	22.78	4.07%
8	34.92	4.15%
9	55.09	4.21%
10	67.98	4.25%
11	100.67	4.22%
12	101.7	4.32%
13	107.78	4.26%
14	117.9	4.21%
15	129.65	4.16%
16	158.16	4.3%

**Table 6: The impact of the number of inputs allowed in symmetry detection on performance and runtime.**

permutational symmetries. Such optimizations appear particularly useful for multiplexers, but are not restricted by gate type.

Compared with other symmetry detection techniques, ours finds more symmetries than other methods, including multi-variable permutational and phase-shift symmetries for both inputs and outputs. This is important in circuit rewiring because the more symmetries are detected, the more rewiring opportunities will be created.

Our experimental results on common circuit benchmarks indicate that in addition to many permutational symmetries, 28% of subcircuits possess phase-shift symmetry on their inputs and 17% on their outputs. These numbers suggest that rebuffering and rewiring using phase-shift symmetry are practical optimization methods. Experimental results also show that the wirelength reduction in our method is comparable to detailed placement but is orthogonal to it — our method can be used after detailed placement to double the wirelength reduction and improve pin access. Empirical trade-offs between runtime and wirelength reduction indicate that most wirelength reduction can be found relatively quickly, while achieving additional improvement takes much longer. This feature of our rewiring method can be convenient in practice: Since each rewiring step is independent of other steps, significant wirelength reduction can be found in a short time, and extra runtime can be used to further improve the result. An average of 4.3% wirelength reduction is observed from the benchmarks using similar amount of time as the placer, which shows that our rewiring method is very efficient and effective. However, the improvement due to rewiring is very different from improvements that may occur in global placement — rewiring can enhance pin access, improving the routability of good and poor global placements.

## 7. REFERENCES

- [1] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa and I. L. Markov, "Unification of Partitioning, Floorplanning and Placement", *ICCAD*, 2004, pp. 550-557.
- [2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry", *IEEE Transactions on CAD*, Sep. 2003, pp. 1117-1137.
- [3] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", *DAC*, 2000, pp. 693-698.
- [4] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Toward CAD-IP Reuse: The MARCO GSRV Bookshelf of Fundamental CAD Algorithms", *IEEE Design and Test*, May 2002, pp. 72-81.
- [5] <http://vlsicad.eecs.umich.edu/BK/PlaceUtils/>
- [6] C. W. Chang, M. F. Hsiao, B. Hu, K. Wang, M. Marek-Sadowska, C. H. Cheng, and S. J. Chen, "Fast

- Postplacement Optimization Using Functional Symmetries”, *IEEE Transactions on CAD*, Jan. 2004, pp. 102-118.
- [7] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, “Exploiting Structure in Symmetry Detection for CNF”, *DAC*, 2004, pp. 530-534.
  - [8] V. N. Kravets and K. A. Sakallah, “Generalized Symmetries in Boolean Functions”, *ICCAD*, 2000, pp. 526-523.
  - [9] V. N. Kravets, “Constructive Multi-Level Synthesis by Way of Functional Properties”, Ph. D. Thesis, University of Michigan, 2001.
  - [10] A. Mishchenko, “Fast Computation of Symmetries in Boolean Functions”, to appear in *IEEE Transactions on CAD*.
  - [11] D. Moller, J. Mohnke, and M. Weber, “Detection of Symmetry of Boolean Functions Represented by ROBDDs”, *ICCAD*, 1993, pp. 680-684.
  - [12] S. Panda, F. Somenzi, and B. F. Plessier, “Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams”, *ICCAD*, 1994, pp. 628-631.
  - [13] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli, “Structural Detection of Symmetries in Boolean Functions”, *ICCD*, 2003, pp. 498-503