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 circuit performance — 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. Key insights in our work include (1) the importance of functional symmetries in rewiring and rebuffering, and (2) the emphasis on netlist changes that can be accommodated for a given placement without additional ECO requests. We propose new algorithms for post-placement optimization based on the knowledge of functional symmetries in small sub-circuits, as well as a new symmetry detection technique which, unlike previous work, detects all possible input and output symmetries, and uses a very compact representation of symmetries. While in general the new technique is potentially less scalable than existing ones, 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 is known, 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 equivalent pins and uses in rewiring [16, 15, 19, 7], mostly focusing on two-variable input symmetries. The work in [13] extends symmetry detection to multi-variable input symmetries, but simultaneous permutations of inputs and outputs are not considered, and they do not apply their techniques to rewiring. 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 com-

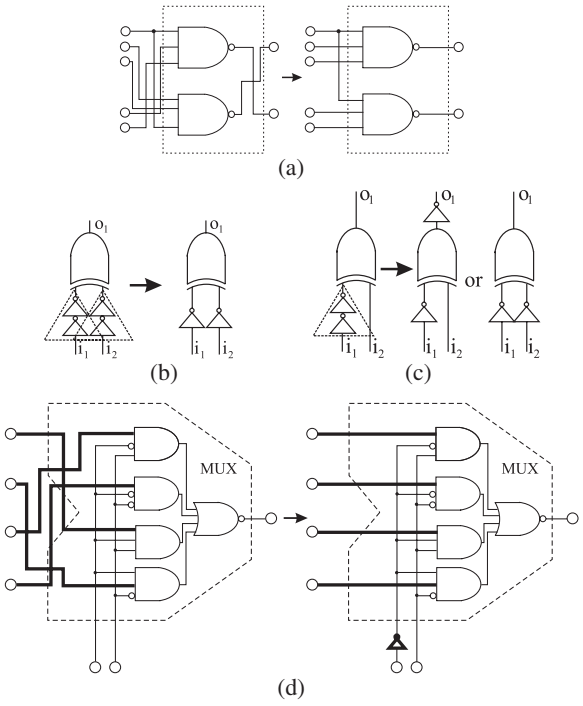

Figure 1: Rewiring and rebuffering examples: (a) multiple inputs and outputs are rewired simultaneously using pin-permutation symmetry, (b) buffers on wires i_1 and i_2 are changed to inverters using phase-shift symmetry to reduce gate delay, (c) the buffer on wire i_1 is replaced with an inverter using 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.

0-7803-9254-X/05/$20.00 ©2005 IEEE.

positions. For example, the rewiring opportunity in the Figure 1(a) cannot be discovered unless both input and output symmetries can be detected. "Phase-shift" symmetries considered in [13] can also be detected, and they can be used for rewiring or rebuffering to optimize timing. Examples of these applications are given in Figures 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. However, our approach does not impose any restrictions on algorithms used for placement and can be applied with analytical or simulated-annealing placers. To serve this purpose, we also use other subcircuits suitable for rewiring and rebuffering, including buckets of cells found by depth-first or breadth-first traversal starting from a cell. For permutative rewiring, we exhaustively search for pin-permutation 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 proposes 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 are described. Previous work on post-placement rewiring and 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 $z = x_1$ XOR y_1 and $w = x_2$ 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 $F(...x_i,...,x_j,...)$. They are symmetric if the function does not change when the variables are swapped:*

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

This type of of symmetry is known as the *classical symmetry*, or the *non-skew non-equivalence symmetry*, denoted $x_i NE x_j$. If we use F_0 to represent the negative cofactor of F with respect to x_i and F_1 to represent the positive cofactor, then this symmetry is also denoted as $F_{01} = F_{10}$, which means the cofactor of F with respect to $\overline{x_i}x_j$ is the same as the cofactor with respect to $x_i\overline{x_j}$.

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 $x_i E x_j$. If one of the two cofactors is complemented, then two more symmetry relationships will be produced: $F_{01} = \overline{F_{10}}$ is *skew non-equivalence symmetry*, and is also denoted $x_i !NE x_j$. $F_{00} = \overline{F_{11}}$ is the *skew equivalence symmetry* and is denoted $x_i !E x_j$.

Kravets et al.[13] 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:

$$F(..,a,..,b,..,c,..,d,..) = F(..,c,..,d,..,a,..,b,..)$$

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 as a function-preserving transformation involving the inversion of one or more inputs that 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.

DEFINITION 4. *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 $z = x_1$ XOR y_1 and $w = x_2$ XOR y_2 given in Example 1. Aside from 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 inversions of inputs and outputs are used. Note that due to Boolean consistency, a symmetry composed of complement of variables in another symmetry is the same symmetry. For example, $y_2 \leftrightarrow x_2'$ is the same as $x_2 \leftrightarrow y_2'$.*

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.

Data structure used	Target	Symmetries detected	Main applications	Time complexity
BDD [15]	Boolean functions	All 1^{st} order input symmetries	Synthesis	$O(n^3)$
Circuit - Supergate [7]	Gate-level circuits	1^{st} order input symmetries in supergates, opportunistically	Rewiring, technology mapping	$O(n)$
Circuit - Boolean decomposition [18]	Gate-level circuits	Input and output permutational symmetries, higher-order	Rewiring, physical design	$\Omega(n)$
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.

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 [11].

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 [14], BDD minimization [17], and circuit rewiring [7]. 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 BDDs. 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 [15] 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 BDDs 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 this paper 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. [19] transform the circuit to NOR gates. C. W. Chang et al. [7] 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. Wallace [18] uses concepts from Boolean decomposition [3] and converts the circuit to "quasi-canonical forms", and then input symmetries are recognized from these forms. This technique is capable of finding higher-order symmetries.

A comparison of BDD-based symmetry detection [15], circuit-based symmetry detection [7] and the method proposed in this paper is summarized in Table 1.

2.3 Graph-based Algorithms

Darga et al. [11] have recently improved symmetry detection algorithms used by older software. Their symmetry detector Saucy finds all symmetries of a given colored undirected graph. To this end, consider an undirected graph G with n vertices, and let $V = \{0,...,n-1\}$. 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 π of the labels assigned to vertices in G such that $\pi(G) = G$; we say that π 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 online 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.

C. W. Chang et al. [7] 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. Apart from 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.

Another type of rewiring is based on the addition and removal of wires, where gates need to be changed in order to reflect the change of wiring. For example, a 4-input AND gate may need to be changed to a 5-input AND gate if a wire is added to the fanin of that gate. Increasing the number of gate inputs also increases the size of the gate and may not be feasible for a given placement unless one can find empty sites around the gate in question. There are three major techniques used to determine the wires that can be reconnected. The first one uses reasoning based on ATPG (Automatic Test Pattern Generation) such as REWIRE [9] and RAMFIRE [8], which tries to add a redundant wire that can make the target wire redundant so that it can be removed. The second class of techniques is graph-based; one example is GBAW [20], which uses pre-defined graph representation of subcircuits and relies on pattern matching to replace wires. The

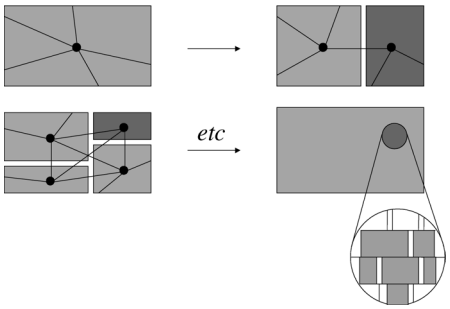

Figure 2: High-level overview of min-cut placement. The rectangles represent bins that partition the layout region, and the lines represent wires. Movable objects assigned to a given bin are tentatively assigned to the geometric center of the bin.

third technique uses SPFDs (Sets of Pairs of functions to be Distinguished) [21, 10] and is based on don't cares. Compared to these techniques, symmetry-based rewiring does not require any changes in gates and completely preserves the initial placement.

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 of min-cut placement is given in Figure 2. In the figure, placement bins are represented by rectangles and wires by lines connected to bin centers. Min-cut partitioning is used at each step until the bins are small enough for placement by branch-and-bound into legal sites.

In this work we use the top-down placer Capo [4, 1]. The bins created by Capo contain tightly-connected cells and are used for symmetry detection and rewiring. However, our rewiring technique is not limited to placements produced by min-cut placers and can also be applied to layouts produced by other types of algorithms.

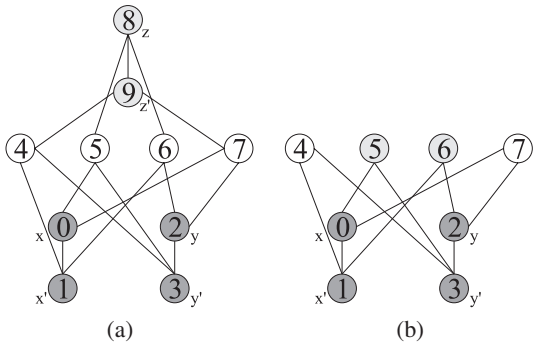

Figure 3: Representing the 2-input XOR function by (a) the full graph, (b) the reduced graph for faster symmetry-finding.

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, it solves the graph automorphism (symmetry detection) problem on this graph, and then the symmetries found are converted to symmetries of the original Boolean function. This section describes the mapping from Boolean functions to graphs 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 outputs vertices have another color.

7. All term vertices have the same color.

The idea behind this construction is that if an input vertex gets permuted with another input vertex, the term vertices connected to them will also need to be permuted. However, the edges between term vertices and output vertices restrict such permutations to the following cases: (1) the permutation of term vertices does not affect the connections to output vertices, which means the outputs are unchanged, (2) permuting term vertices may also require permuting output vertices, thus capturing output symmetries. A proof of correctness will appear in the full version of the paper.

Figure 3(a) illustrates our construction for $z = x \oplus y$. In general, vertex indices are assigned as follows. For n inputs and m outputs, the ith input is represented by vertex $2i$, while the complementary vertex has index $2i+1$. There are 2^n terms, and the ith term is indexed by $2n+i$. Similarly, the ith output is indexed by

```
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 = Saucy();
10 symmetries = reverse_map(A);
```

Figure 4: Our symmetry detection algorithm.

```
1  repeat using different subcircuits
2   foreach subcircuit
3     bucket ← cells in subcircuit;
4     ckt=generate_circuit(bucket);
5     foreach 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.

$2n+2^n+2i$, while its complement is indexed by $2n+2^n+2i+1$. For the XOR function, the minterms are $1(01_2)$ $2(10_2)$ and the maxterms are $0(00_2)$ $3(11_2)$. Therefore term vertices 5 and 6 are connected to z while vertices 4 and 7 are connected to z'.

The symmetry detector Saucy [11] used in this work typically runs faster when there are more colors in the graph and when the graph is smaller. Therefore if output symmetries do not need to be detected, a reduced version of the graph can be used to detect input symmetries faster. It is constructed similarly to the full graph, but without output vertices and potentially with more vertex colors. We define an *output pattern* as a set of output vertices in the full graph that are connected to a given term vertex. Further, term vertices with different output patterns shall be colored by different colors. Figure 3(b) illustrates the reduced graph for the two-input XOR function.

All the minterms and maxterms of the Boolean function are used in the graph because we focus on fully-specified Boolean functions. It may also be possible to extend our work to partially specified Boolean functions, but we do not pursue such extensions here.

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, the 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 our 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 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.

During implementation, we observed that our symmetry detector is more efficient when there are few or no symmetries in the graph and takes more time when there are many symmetries. For example, for a typical randomly chosen 16-input function the runtime is 0.11 sec because random functions typically have no symmetries. However, it takes 9.42 sec to detect all symmetries of the 16-input XOR function. Runtimes for 18 inputs are 0.59 sec and 92.39 sec resp.

4. POST-PLACEMENT REWIRING AND REBUFFERING

This section describes two techniques for post-placement optimization – rewiring and rebuffering. Rewiring uses functional symmetries of extracted subcircuits, detected by exhaustive search, to reduce wirelength or optimize timing. We also propose an innovative approach to buffer insertion using phase-shift symmetries.

4.1 Permutative Rewiring

After placement, symmetries can be used to rewire the netlist to reduce the wirelength without changing the function of the circuit. This is achieved by exhaustive search and rewiring of functional symmetries found in subcircuits. Since we represent the netlist by a hypergraph (where cells are represented by nodes and nets are represented by hyper-edges), breadth-first search (BFS) and depth-first search (DFS) starting from a given vertex will find connected cells that are suitable for rewiring. Currently, subcircuits are extracted in the following order: (1) four rounds of depth-first traversal of cells; subcircuits containing 1, 2, 3 and 4 cells are extracted in each round respectively, (2) two rounds of breadth-first traversal of cells; subcircuits containing 3 and 4 cells are extracted in each round, (3) one round using the cells in every half-bin produced by Capo, (4) one round using the cells in every bin produced by Capo (here we only consider the smallest bins that correspond to end-case placement).

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 generates symmetries from the returned symmetry generators in the following way: (1) put symmetries that share at least one variable into the same set, (2) enumerate all permutations of symmetries from each set except symmetry sets containing $\geq 7!$ permutations. In this case, 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* reverses the rewiring.

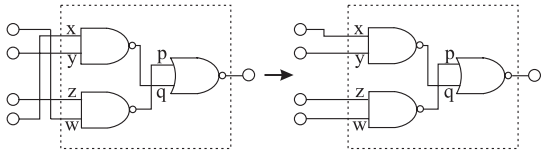

Figure 6: A rewiring opportunity for p and q that cannot be detected by only considering one bucket. To rewire p and q, a subcircuit with p and q as inputs must be extracted.

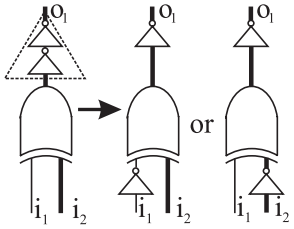

Figure 7: Rebuffering example: The bold line represents a critical path. The buffer on wire o_1 is replaced with an inverter to reduce delay. An inverter is inserted into either i_1 or i_2 to preserve the correctness of logic.

The reason why multiple passes with different sizes of buckets are 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. By choosing disjoint buckets, it is very easy to parallelize symmetry detection and rewiring.

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 a way to compensate for the single inversion.

Phase-shift symmetries can be used to serve this purpose because they involve negations of signals. Therefore if buffer insertion is considered for a given wire, we can search for phase-shift symmetries involving negation of that signal, 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 following 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 four 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.

```
1  w ←wire for buffer insertion;
2  extract subcircuits with w as one of the
     outputs;
3  foreach ckt ∈ 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.

Figure 8 shows our algorithm to replace buffers with inverters. 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. 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. However, they can be employed before gate- and wire-sizing to achieve better optimization than using the sizing techniques alone.

There are several other applications of phase-shift symmetries in addition to rebuffering. For example, the ability to handle phase-shift symmetries may reduce interconnect by enabling permutational symmetries and by facilitating additional long-range rewiring opportunities, as the MUX example in Figure 1(d) shows. The fact that many standard cells offer their output signals in both complemented and uncomplemented forms suggests another use of phase-shift symmetries which does not require netlist changes or changing the locations and sizes of cells. Such rewiring would be particularly applicable to latches and flip-flops with output pins Q and QN.

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 produces 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. Unselected testcases show the same trends but are omitted in this paper due to space limitation. 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. The correctness of rewiring has been successfully verified in all cases by SAT-based equivalence checking.

We convert every testcase from BLIF to the Bookshelf placement format (*.nodes* and *.nets* files). For this, we enhanced the script

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.

previously available online in the Place-Utils entry of the GSRC Bookshelf [5, 6], 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 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 are not reported due to space limitations, the probability of finding phase-shift symmetries for inverter insertion can be observed from reported data: 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.

C. W. Chang et al. [7] use symmetry to optimize timing and power, so their results are not directly comparable with our work. However, the numbers of symmetries detected from the circuits can be compared and are summarized in Table 3. 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 [7] use a Sun Ultra10 workstation with 128M memory for their experiments, which is much slower than the machine used in our paper. Table 3 indicates that our algorithm finds more symmetries than the work in [7], mostly because the supergates used in [7] are very limited in size. We believe that our method can find all symmetries found by the methods from [7], as well as some additional symmetries. On average, we can detect twice as many symmetries than their method using additional time.

Benchmark	This work		C. W. Chang's work [7]		
	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 C. W. Chang et al [7] in terms of symmetries found. The runtime is not directly comparable because the machine used by Chang is slower than ours.

Benchmark	Original Wirelength	Wirelength reduction		Runtime (seconds)		
		Re-wiring	Detailed placement	Re-wiring	Global placement	Detailed placement
alu2	5532.83	7.11%	10.93%	8.4	2.2	0.2
alu4	40064.93	9.51%	4.69%	77.8	19.6	3.4
b02	142.9	8.29%	0%	0.5	2.6	0.2
b10	1556.7	5.2%	3.98%	2.2	6	0.2
b17	364912.8	2.93%	2.29%	648.79	339.4	31.6
c5315	35667.36	1.9%	2.31%	9	16.6	3
c7552	46758.46	2.02%	2.36%	9.8	24.8	4
dalu	23385.26	3.47%	4.32%	5.8	13.2	2.8
i10	54782.3	2.36%	2.95%	62.2	15.8	2.8
s38417	129967.6	1.96%	2.05%	47	174.6	21.8
s38584	174475.8	2.49%	2.25%	152	161.4	20.2
Average	79749.72	4.29%	3.47%	93.04	70.56	8.2

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

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 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 total runtime of global placement plus detailed placement, which shows that our method is efficient and effective for wirelength reduction. Empirical results are shown in Tables 4 and 5.

The third experiment shows the relationship between the number

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/after detailed placement.

Number of inputs allowed	Runtime (seconds)	Wirelength reduction
2	3.25	1.52%
4	7.27	3.3%
6	15.2	3.8%
8	34.92	4.15%
10	67.98	4.25%
12	101.7	4.32%
14	117.9	4.21%
16	158.16	4.3%

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

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. Empirical results shown in Table 6 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, over 4% wirelength reduction can be achieved.

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 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. Furthermore, our rewiring techniques can be easily parallelized by distributing disjoint subcircuits to different processors. An average of 4.3% wirelength reduction is observed from the benchmarks, requiring approximately as much time as a global placer. This shows that our rewiring method is very 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 global placements regardless of their quality. Our ongoing work on non-permutative long-range rewiring shows that additional 1.5% savings in wirelength can be achieved with reasonable runtime, which makes our rewiring techniques more promising.

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 Trans. on CAD*, Sep. 2003, pp. 1117-1137.

[3] V. Bertacco and M. Damiani, "Boolean Function Representation Based on Disjoint-Support Decompositions", *ICCD*, Oct. 1996, pp. 27-32.

[4] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", *DAC*, 2000, pp. 693-698.

[5] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Toward CAD-IP Reuse: The MARCO GSRC Bookshelf of Fundamental CAD Algorithms", *IEEE Design and Test*, May 2002, pp. 72-81.

[6] `http://vlsicad.eecs.umich.edu/BK/PlaceUtils/`

[7] 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 Trans. on CAD*, Jan. 2004, pp. 102-118.

[8] C. W. Chang and M. Marek-Sadowska, "Single-Pass Redundancy Addition and Removal", *ICCAD*, Nov. 2001, pp. 606-609.

[9] S. C. Chang, L. P. P. P. van Ginneken, and M. Marek-Sadowska, "Circuit Optimization by Rewiring,"*IEEE Trans. on Computers*, Sep. 1999, pp. 962-969.

[10] J. Cong and W. Long, "Theory and Algorithm for SPFD-Based Global Rewiring", *IWLS*, June 2001, pp. 150-155.

[11] 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.

[12] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Apps of Satisfiability Testing, SAT*, 2003, pp. 502-518.

[13] V. N. Kravets and K. A. Sakallah, "Generalized Symmetries in Boolean Functions", *ICCAD*, 2000, pp. 526-523.

[14] V. N. Kravets, "Constructive Multi-Level Synthesis by Way of Functional Properties",Ph. D. Thesis, Univ. of Michigan,2001.

[15] A. Mishchenko, "Fast Computation of Symmetries in Boolean Functions", *IEEE Trans. on CAD*, Nov. 2003, pp. 1588-1593.

[16] D. Moller, J. Mohnke, and M. Weber, "Detection of Symmetry of Boolean Functions Represented by ROBDDs", *ICCAD*, 1993, pp. 680-684.

[17] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams", *ICCAD*, 1994, pp. 628-631.

[18] D. E. Wallace, "Recognizing input equivalence in digital logic", *IWLS*, 2001, pp. 207-212.

[19] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Structural Detection of Symmetries in Boolean Functions", *ICCD*, 2003, pp. 498-503

[20] Y. L. Wu, W. Long, and H. Fan, "A Fast Graph-Based Alternative Wiring Scheme for Boolean Networks", *International VLSI Design Conference*, 2000, pp. 268-273.

[21] S. Yamashita, H. Sawada, and A. Nagoya, "A New Method to Express Functional Permissibilities for LUT Based FPGAs and Its Applications", *ICCAD*, 1996, pp. 254-261.