Postplacement Rewiring by Exhaustive Search for Functional Symmetries

KAI-HUI CHANG, IGOR L. MARKOV, and VALERIA BERTACCO
University of Michigan

We propose two new algorithms for rewiring: a postplacement optimization that reconnects pins of a given netlist without changing the logic function and gate locations. In the first algorithm, we extract small subcircuits consisting of several gates from the design and reconnect pins according to the symmetries of the subcircuits. To enhance the power of symmetry detection, we also propose a graph-based symmetry detector that can identify permutational and phase-shift symmetries on multiple input and output wires, as well as hybrid symmetries, creating abundant opportunities for rewiring. Our second algorithm, called long-range rewiring, is based on reconnecting equivalent pins and can augment the first approach for further optimization. We apply our techniques for wirelength optimization and observe that they provide wirelength reduction comparable to that achieved by detailed placement.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: VLSI, placement, rewiring

ACM Reference Format:

Chang, K.-H., Markov, I. L., and Bertacco, V. 2007. Postplacement rewiring by exhaustive search for functional symmetries. ACM Trans. Des. Autom. Electron. Syst. 12, 3, Article 32 (August 2007), 21 pages. DOI = 10.1145/1255456.1255469 http://doi.acm.org/10.1145/1255456.1255469

1. INTRODUCTION

The semiconductor industry's tendency to always push the limit for minimum transistor sizing has led to new challenges in integrated circuit design. One key challenge is wire parasitics, whose aggragation is leading to increased delay and power consumption in wires to the point that their contribution dominates that of transistor switching. Since accurate wire topology is available only after the circuit is placed, postplacement optimizations have been studied extensively. Typically, these optimizations focus on improving delay, power,

Authors' addresses: K.-H. Chang, I. L. Markov, and V. Bertacco, Electrical Engineering and Computer Science Department, University of Michigan, 2260 Hayward Avenue, Ann Arbor, MI 48109-2121; email: {changkh,imarkov,valeria}@eecs.umich.edu.

reliability, and/or wirelength. However, we observe that existing techniques may fail to provide consistent improvements because, when optimizing one design aspect, they may worsen other physical aspects. For example, cell cloning may improve timing at the expense of significantly increasing wirelength [Hrkic et al. 2004], and the timing optimization may actually worsen the final delay because of critical path detours after routing. The widespread unpredictability of all these optimizations makes design closure an extremely difficult task, often leading to many iterations of postplacement optimization and delays in design completion.

In general, the stability (or lack thereof) of an optimization can be determined by checking whether it creates illegal changes to the layout. For instance, if an optimization changes placement by creating cell overlaps, then an additional legalization step needs to be performed to remove the overlaps, and the latter could cancel out or even worsen the optimizations of the former. As a specific example, rewiring techniques based on addition and removal of wires may create overlaps because cell types will be changed [Chang and Marek-Sadowska 2001; Chang et al. 1999, 1997; Jiang et al. 1997; Wu et al. 2000], and the evaluation of the changes must be delayed until legalization is performed. On the other hand, the impact of optimizations that do not affect cell locations can be evaluated immediately and reliably, therefore they have little risk of hampering the design closure. In addition, such optimizations allow *metal fix*, a postsilicon repair technique that only changes the metal layers while preserving the masks for manufacturing transistors. With mask cost approaching one million dollars per set, the ability to repair a circuit by metal fix reduces the cost of respin significantly.

Chang et al. [2006] pointed out that symmetry-based rewiring is the only postplacement optimization that does not affect cell placement. As a result, it can be integrated into any design flow safely, and allows metal fix. To this end, the symmetry-based rewiring system proposed by Chang et al. [2004] is effective in optimizing the delay, power and reliability of a digital circuit. Their system iteratively extracts subcircuits from the circuit and performs rewiring optimizations by exploiting symmetries. However, their symmetry-detection technique is only applicable to designs synthesized using certain basic gate types and cannot be applied to many practical circuits.

In this article we extend our preliminary work in Chang et al. [2005] and propose a novel rewiring technique that can be applied to designs mapped to any type of cell. This is achieved by detecting symmetries in Boolean functions, regardless of the logic structure used to represent them. In addition, we observe that detecting fewer symmetries creates fewer possibilities for optimization. While previous work on rewiring considers input symmetries only, our comprehensive symmetry detector allows us to handle both input and output permutations, as well as their composition. For example, the rewiring opportunity in Figure 1(a) cannot be discovered unless both input and output symmetries can be detected. In addition, "phase-shift" symmetries [Kravets and Sakallah 2000], that is, symmetries involving negation of inputs and/or outputs, can also be detected. An example of this application is given in Figure 1(c). In our rewiring approach, we extract subcircuits iteratively from

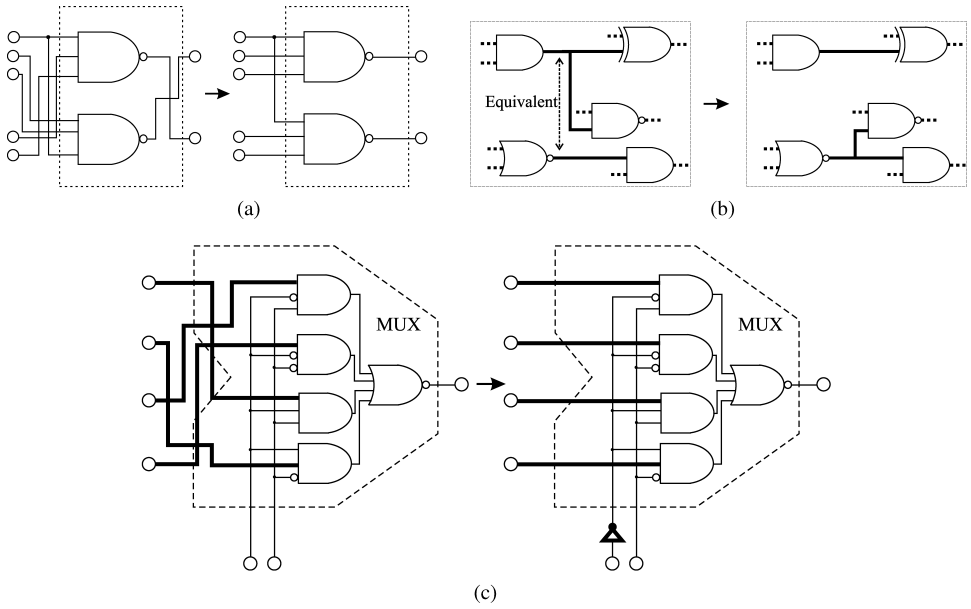

Fig. 1. Rewiring examples: (a) multiple inputs and outputs are rewired simultaneously using pin-permutation symmetry; (b) long-range rewiring using equivalent wires; and (c) inputs to a multiplexer are rewired by inverting one of the select bits. Bold lines represent changes made in the circuit.

the circuit, and use the symmetries detected to reconnect pins and optimize wirelength. We also propose a "long-range" rewiring technique which operates by rearranging the connection of equivalent pins, complementing the baseline method (an example is shown in Figure 1(b)). Long-range rewiring first identifies candidate equivalent wires by random simulation and uses formal methods to verify their equivalency. Similar to symmetry-based rewiring, it does not affect cell placement. Our experimental results show that the proposed approach can reduce total wirelength by approximately 5%. While this new symmetry-detection technique is potentially less scalable than previous ones, we found experimentally that the execution runtimes are reasonable.

The remainder of the article is organized as follows. Section 2 introduces basic principles of symmetry, and describes relevant previous work on symmetry detection and circuit rewiring. In Section 3 we describe our symmetry-detection algorithm. Section 4 discusses the postplacement rewiring algorithms. Finally, we provide experimental results in Section 5 and conclude in Section 6.

2. BACKGROUND

The rewiring technique proposed in this article is based on symmetry detection. Therefore, in this section, we present background ideas and related work about symmetry detection. Previous work on postplacement rewiring is also discussed.

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, in function "$o = (x + y) + z$", $x \leftrightarrow z$ is a semantic symmetry because the function will not be changed after the permutation of variables; however, it is not a syntactic symmetry because the structure of the function will be changed. On the other hand, $x \leftrightarrow y$ is both a semantic and syntactic symmetry. In this work we exploit functional symmetries, whose definition is provided next.

Definition 1. Consider a multioutput Boolean function $F : \mathcal{B}^n \rightarrow \mathcal{B}^m$, where

$$F(i_1...i_n) =< f_1(i_1...i_n),\ f_2(i_1...i_n)...\ f_m(i_1...i_n) > . \tag{1}$$

A *functional symmetry* is a one-to-one mapping $s : \mathcal{B}^{(n+m)} \rightarrow \mathcal{B}^{(n+m)}$ such that

$$< f_1(i_1...i_n),\ f_2(i_1...i_n)...\ f_m(i_1...i_n) >=< s(f_1)(s(i_1)...s(i_n)),\ s(f_2)(s(i_1)...s(i_n))...$$
$$s(f_m)(s(i_1)...s(i_n)) > . \tag{2}$$

In other words, a functional (semantic) symmetry is a transformation of inputs and outputs which does not change the functional relation between them.

Example 1. Consider the multioutput 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) $x_2 \leftrightarrow y_2$; and (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 before. A set of symmetries with this property is called a set of *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 of items (1), (2), and (3).

While most previous work on symmetry detection focuses on permutations of two variables, Pomeranz and Reddy [1994] and Kravets and Sakallah [2000] consider swaps of groups of ordered variables. These swaps are called *higher-order symmetries* in Kravets and Sakallah [2000]. 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 bits), but no two input bits in different bit positions are symmetric by themselves. Kravets 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. Our work generalizes this concept by including output symmetries involving inversion in the class of phase-shift symmetries. We also define *composite phase-shift symmetry* as a symmetry which consists of phase-shift and permutational symmetries.

In this article we commonly 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 multioutput 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 $w \leftrightarrow w'$; and (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 a complement of variables in another symmetry is the same symmetry. For example, $y_2 \leftrightarrow x_2'$ is the same as $x_2 \leftrightarrow y_2'$.

2.2 Semantic and Syntactic Symmetry Detection

Symmetry detection in Boolean functions has several applications, including technology mapping, technology-independent logic synthesis, BDD minimization [Panda et al. 1994], and circuit rewiring [Chang et al. 2004]. Methods for symmetry detection can be classified into four categories: BDD-based, graph-based, circuit-based and Boolean-matching-based. However, it is relatively difficult to find all the symmetries of a Boolean function, regardless of the representation used.

BDDs are particularly convenient for semantic symmetry detection because they support abstract functional operations. One naive way to find two-variable symmetries is to compute the cofactors for every pair of variables, say, whether v_1 and v_2, and check if $F_{\overline{v_1}v_2} = F_{v_1\overline{v_2}}$ or $F_{\overline{v_1}\,\overline{v_2}} = F_{v_1v_2}$. Recent research [Mishchenko 2003] indicates that symmetries can be found or disproved without computing all the cofactors thus significantly speeding-up symmetry detection. However, work on BDD-based symmetry detection has been limited to input permutations only, probably due to the single-output nature of BDDs.

In this article, symmetry-detection methods that rely on efficient algorithms for the graph automorphism problem (i.e., finding all symmetries of a given graph) are classified as graph-based. They construct a graph whose symmetries faithfully capture the symmetries of the original object, find its automorphisms (i.e., symmetries) and map them back to the original object. Aloul et al. [2003] proposed a way to find symmetries for SAT clauses using this approach. The symmetry-detection approach proposed in this article is inspired by their work.

Circuit-based symmetry-detection methods often convert a circuit representing the function in question to a more regular form where symmetry detection is more practical and efficient. For example, Wang et al. [2003] transform the circuit to NOR gates. Chang et al. [2004] use a more elaborate approach by converting the circuit to XOR, AND, OR, INV, and BUF first, and then partitioning the circuit so that each subcircuit is fanout free. Next, they form "supergates" from the gates and detect symmetries for these supergates. Wllace [2001] uses concepts from Boolean decomposition [Bertacco and Damiani 1997] and converts the circuit to "quasicanonical forms," and then the input

Table I. Comparison of Different Symmetry-Detection Methods

Data structure used	Target	Symmetries detected	Main applications	Time complexity
BDD [Mishchenko 2003]	Boolean functions	All 1st-order input symmetries	Synthesis	$O(n^3)$
Circuit—Supergate [Chang et al. 2004]	Gate-level circuits	1st-order input symmetries in supergates, opportunistically	Rewiring, technology mapping	$O(m)$
Circuit—Boolean decomposition [Wllace 2001]	Gate-level circuits	Input and output, permutational symmetries, higher-order	Rewiring, physical design	$\Omega(m)$
Circuit—simulation, ATPG [Pomeranz and Reddy 1994]	Gate-level circuits	Input, output, and phase-shift symmetries, higher-order	Error diagnosis, technology mapping	$\Omega(2^n)$
Boolean matching [Chai and Kuehlmann 2005]	Boolean functions	Input, output and phase-shift symmetries, higher-order	Technology mapping	$\Omega(2^n)$
Graph automorphism (this work)	Both (with small number of inputs)	All input, output, phase-shift symmetries, and all orders, exhaustively	Exhaustive small group rewiring	$\Omega(2^n)$

In the table, n is the number of inputs to the circuit and m is the number of gates. Currently-known BDD-based and most circuit-based methods can detect only a fraction of all symmetries in some cases, while the method based on graph automorphism (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.

symmetries are recognized from these forms. This technique is capable of finding higher-order symmetries. Another type of circuit-based symmetry detector relies on ATPG and simulation, such as the work by Pomeranz and Reddy [1994]. Although their technique was developed to find both first- and higher-order symmetries, they reported experimental results for first-order symmetries only. Therefore, its capability to detect higher-order symmetries is unclear.

Boolean matching is a problem related to symmetry detection. Its purpose is to compute a canonical representation for Boolean functions that are equivalent under negation and permutation of inputs and outputs. Symmetries are implicitly processed by Boolean matching in that all functions symmetric to each other will have the same canonical representation. However, enumerating symmetries from Boolean matching is not straightforward and requires extra work. This topic has been studied by Wu et al. [1994] and Chai and Kuehlmann [2005].

A comparison of BDD-based [Mishchenko 2003], circuit-based [Chang et al. 2004; Pomeranz and Reddy 1994; Wllace 2001], Boolean-matching-based symmetry detection [Chai and Kuehlmann 2005], and the method proposed in this article is summarized in Table I.

2.3 Graph Automorphism Algorithms

Our symmetry-detection method is based on efficient graph automorphism algorithms which have recently been improved by Darga et al. [2004]. 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 relabelings 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.

2.4 Postplacement Rewiring

Rewiring based on symmetries can be used to optimize circuit characteristics. Some rewiring examples are illustrated in Figures 1(a) and (c). Here the goal is to reduce wirelength, and swapping symmetric input and output pins accomplishes this.

Chang et al. [2004] use the symmetry-detection technique described earlier to optimize delay, power, and reliability. In general, the symmetry detection in their work is done opportunistically rather than exhaustively. Experimental results show that their approach can achieve these goals effectively using the symmetries detected. However, they cannot find the rewiring opportunity in Figures 1(a) and (c) 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. Three major techniques are used to determine the wires that can be reconnected. The first uses reasoning based on ATPG (automatic test pattern generation) such as REWIRE [Chang et al. 1999], RAMFIRE [Chang and Marek-Sadowska 2001], and the work by Jiang et al. [1997]. This type of technique tries to add a redundant wire that makes the target wire redundant so that it can be removed. The second class of techniques is graph based; one example is GBAW [Wu et al. 2000], which uses a predefined graph representation of subcircuits and relies on pattern matching to replace wires. The third technique uses SPFDs (sets of pairs of functions to be distinguished) [Cong and Long 2001] and is based on don't-cares. Although these techniques are potentially more powerful than symmetry-based rewiring because they allow more aggressive layout change, they are also less stable and do not support postsilicon metal fix.

3. EXHAUSTIVE SEARCH FOR FUNCTIONAL SYMMETRIES

The symmetry-detection method presented in our work can find all input, output, multivariable, and phase-shift symmetries, including composite (hybrid) symmetries. It relies on symmetry detection of graphs, thus the original Boolean function must first be converted to a graph. 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. Our main contribution is the mapping from a Boolean function to a graph, and shows how to use this to find symmetries of the Boolean function. These techniques are described in detail in this section.

3.1 Problem Mapping

To reduce functional symmetry-detection to the graph automorphism problem, we represent Boolean functions by graphs as described to follow:

(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*. Outputs are handled similarly, and those vertices are called *output vertices*.

(2) Each minterm and maxterm of the Boolean function is represented by a *term vertex*. We introduce an edge connecting every minterm vertex to the output, and an edge connecting every maxterm vertex to the complement of the output. We also introduce an edge between every term vertex and every input vertex or its complement, depending on whether this input is 1 or 0 in the term.

(3) Since inputs and outputs are bipartite permutable, all input vertices have one color, and all outputs vertices another. All term vertices use yet another 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 that the outputs are unchanged; and (2) permuting term vertices may also require permuting output vertices, thus capturing output symmetries. A proof of correctness appears in the Appendix.

Figure 2(a) shows the truth table of function $z = x \oplus y$, and Figure 2(b) illustrates our construction for the function. 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 $2n+2^n+2i$, while its complement is indexed by $2n + 2^n + 2i + 1$.

The symmetry detector Saucy [Darga et al. 2004] used in this work typically runs faster when the graph is smaller and contains more colors. Therefore, if output symmetries do not need to be detected, a reduced version of the graph can be used. 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 differently. Figure 2(c) illustrates the reduced graph for the 2-input XOR function.

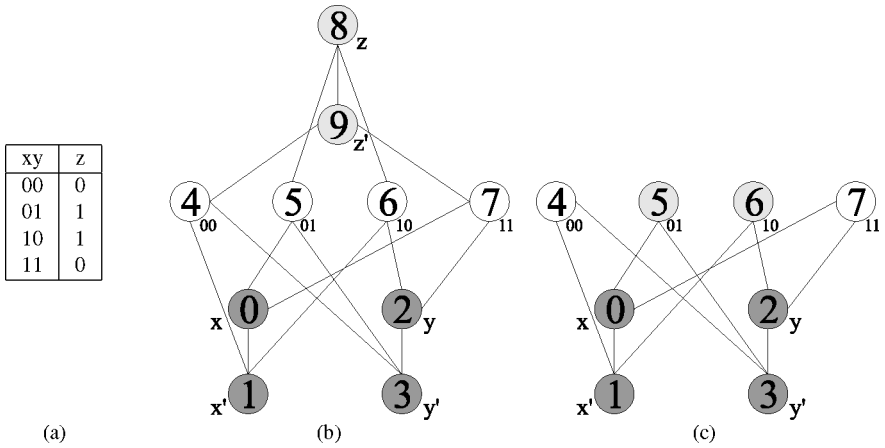

Fig. 2. Representing the 2-input XOR function by: (a) the truth table; (b) the full graph; and (c) the reduced graph for faster symmetry detection.

All the minterms and maxterms of the Boolean function are used in the graph because we focus on fully-specified Boolean functions. Since all the terms are used, and there are 2^n terms for an n-input function, the time and space complexity of our algorithm is $\Omega(2^n)$.

3.2 Discussion

Compared with other symmetry-detection methods, the one method proposed in our work has the following advantages: (1) It can detect all possible input and output symmetries of a function, including multivariable, higher-order, and phase-shift symmetries; and (2) our 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 symmetries easier than other methods which enumerate all symmetry pairs.

While evaluating our algorithm, we observed that Saucy is more efficient when there are few or no symmetries in the graph, whereas it takes more time when there are many. For example, the runtime of a randomly chosen 16-input function 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, respectively.

4. POSTPLACEMENT REWIRING

This section describes two techniques for postplacement optimization, permutative rewiring and long-range rewiring. Permutative rewiring uses symmetries of extracted subcircuits to reduce wirelength. Long-range rewiring finds rewiring opportunities from equivalent wires that are farther apart. It first identifies candidate opportunities by simulating the circuit on random inputs, then prunes such opportunities based on their potential for improving the interconnect, accepting only those which pass on-the-fly formal verification.

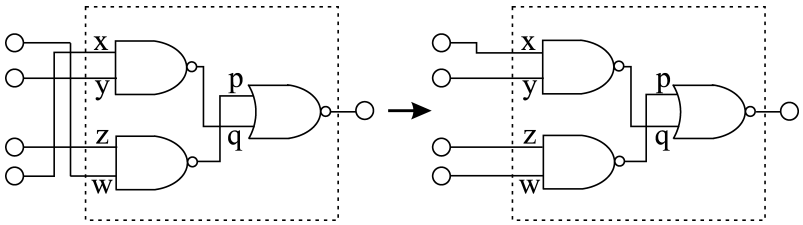

Fig. 3. Rewiring opportunities for p and q cannot be detected by only considering one bucket. To rewire p and q, a subcircuit with p and q as inputs must be extracted.

4.1 Permutative Rewiring

After placement, symmetries can be used to rewire the netlist to reduce wirelength. This is achieved by reconnecting pins according to symmetries found in subcircuits, and these subcircuits are extracted as follows.

(1) We represent the netlist by a hypergraph where cells are represented by nodes and nets are represented by hyperedges.
(2) For each node in the hypergraph, we perform breadth-first search (BFS) starting from the node, and use the first n nodes traversed as subcircuits.
(3) Similarly, we perform depth-first search (DFS) and extract subcircuits using the first m nodes.

In our implementation, we perform BFS extraction 4 times with n from 1 to 4, and DFS 2 times, with m from 3 to 4. This process is capable of extracting various subcircuits suitable for rewiring. In addition to logically connected cells, min-cut placers such as Capo [Caldwell et al. 2000; Adya et al. 2004] produce a hierarchical collection of "placement bins" (buckets) that contain physically adjacent cells, and these bins are also suitable for rewiring. Currently, we also use subcircuits composed of cells in every half- and full bin in our rewiring. After subcircuits are extracted, we perform symmetry detection on these subcircuits. Next, we reconnect the wires to the inputs and outputs of these subcircuits according to the detected symmetries in order to optimize wirelength.

The reason why multiple passes with different sizes of subcircuits are used is that some symmetries in small subcircuits cannot be detected in larger subcircuits. For example, in Figure 3, if the subcircuit contains all the gates, only those 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.

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. It can also be used to reduce the delay on critical paths.

4.2 Long-Range Rewiring

While permutative rewiring is effective in finding local rewiring opportunities, long-range rewiring is capable of finding equivalent wires that are farther apart. It can augment permutative rewiring to further optimize the circuit.

The basic idea is similar to that popular in equivalence checking: Simulation identifies potentially equivalent wires, and a CNF-SAT solver proves whether they are [Lu et al. 2004]. After equivalent wires are identified, we try reconnecting them and accept the reconnection that reduces wirelength. However, unlike in equivalence checking, our goal is layout optimization. Therefore, we can prune rewiring opportunities before formally verifying them. In particular, we perform verification only after wirelength reduction is guaranteed, since equivalence checking is relatively slow.

4.3 Implementation Insights

During implementation, we observed that for subcircuits with a small number of inputs and outputs, it is more efficient to detect symmetries by enumerating all possible permutations using bit operations on the truth table. This is because the required permutations can be implemented with just a few lines of C++ code, making this technique much faster than building the graph for Saucy. We call this algorithm *naive symmetry detection*. To further reduce its runtime, we limit the algorithm to detect first-order symmetries only. In our implementation, naive symmetry detection is used on subcircuits with number of inputs less than 11 and number of outputs less than 3. Experimental results show that the runtime can be reduced by more than half with almost no loss in quality, because lost rewiring opportunities can be recovered in larger subcircuits where Saucy-based symmetry detection is used.

4.4 Discussion

Our rewiring techniques described so far use permutational symmetries. Here we describe two applications of phase-shift symmetries.

(1) The ability to handle phase-shift symmetries may reduce the interconnect by enabling permutational symmetries, as the MUX example in Figure 1(c) shows.

(2) Phase-shift symmetries support the metal fix of logic errors involving only inversions of signals: By reconnecting certain wires, signals may be inversed.

Compared with other rewiring techniques, the advantages of our techniques include the following:

(1) Our rewiring techniques preserve placement, therefore the effects of any change are immediately measurable. As a result, our methods are "safe" and can be applied with every flow. In other words, their application can only improve the optimization objective and never worsen it. This characteristic is especially desirable in highly optimized circuits because changes in placement may create cell overlaps, and the legalization process to remove these overlaps may affect multiple gates, leading to a deterioration of the optimization goal.

(2) Our techniques support postsilicon metal fix, which allows reuse of transistor masks and can significantly reduce respin cost.

(3) The correctness of our optimizations can be verified easily using combinational equivalence checking.

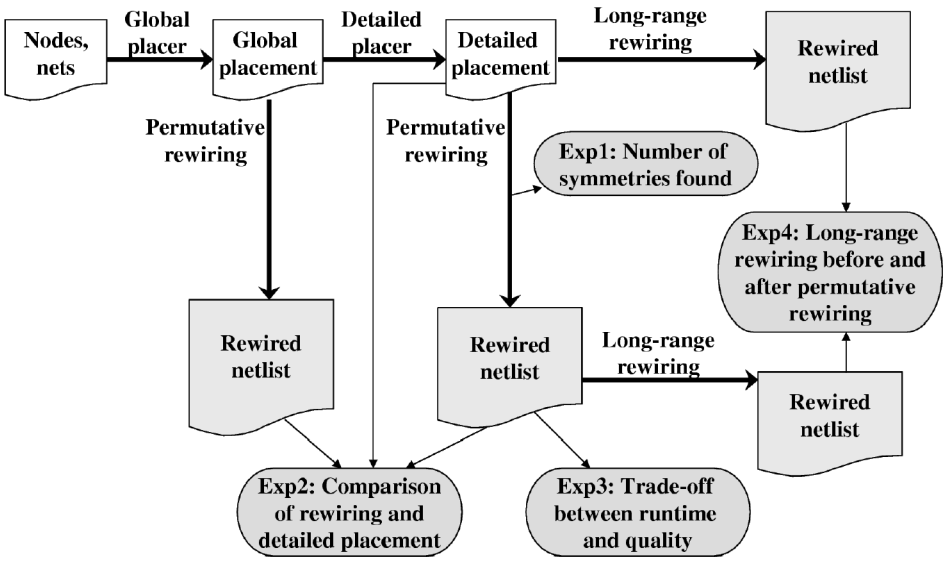

Fig. 4. Flow chart of our symmetry detection and rewiring experiments.

(4) Our techniques can optimize a broad variety of objectives, so long as the objectives can be evaluated incrementally.

The limitations of our rewiring techniques include:

(1) The performance varies with each benchmark, depending on the number of symmetries and equivalent wires which exist in a design. Therefore the improvement is not guaranteed.

(2) In permutative rewiring, the ratio of improvement tends to reduce when designs get larger. Since permutative rewiring is a local optimization, it cannot shorten global nets. However, we have not observed such a trend in long-range rewiring.

5. EXPERIMENTAL RESULTS

Our implementation is written in C++, and the test cases are selected from ITC99, ISCAS, and MCNC benchmarks. To better reflect modern VLSI circuits, we chose the largest test cases from each benchmark suite, and added several small and medium ones for completeness. Unselected test cases show the same trends, but are omitted in this article due to space limitation. Our experiments use the min-cut placer Capo. The platform used is Fedora 2 Linux on a Pentium-4 workstation running at 2.26 GHz with 512M RAM.

We convert every testcase from BLIF to the Bookshelf placement format (*.nodes* and *.nets* files) using the converter provided in Caldwell and Markov [2002] and GSRCBookshelf [2007]. We report two different types of experimental results in this section, including the number of symmetries detected and rewiring. A flow chart of our experiments on symmetry detection and rewiring is given in Figure 4.

Table II. Number of Symmetries Found in Benchmark Circuits

Benchmark	Number of subcircuits	Symmetries				
		Input	Phase-shift input	Output	Phase-shift output	Input and output
ALU2	876	855	120	249	126	211
ALU4	15933	15924	242	1245	243	1244
B02	143	130	18	22	15	21
B10	1117	1015	160	201	137	170
B17	198544	190814	23789	32388	17559	24474
C5315	20498	19331	9114	5196	4490	4145
C7552	28866	26626	12243	7540	6477	5895
DALU	16665	15506	6632	3272	2550	2852
I10	14670	14165	4298	3710	2929	2516
S38417	141241	126508	75642	64973	59319	61504
S38584	122110	117084	55966	35632	29661	33655
Average	100%	94%	28%	23%	18%	20%

5.1 Symmetries Detected

The first experiment evaluates the symmetries found in the benchmarks, and the results are summarized in Table II. In the table, the column labeled "number of subcircuits" is the number of subcircuits extracted from the benchmark for symmetry detection. "Input" is the number of subcircuits which contain input symmetries, "phase-shift input" is the number of subcircuits that contain phase-shift input symmetries. "Output" and "phase-shift output" are used in a similar way. "Input and output" are subcircuits that contain symmetries involving both inputs and outputs. The number of symmetries found in circuits can be used to predict the probability of finding rewiring opportunities: At least 66% of the subcircuits contain permutational input symmetries and are suitable for rewiring. It can also be observed that although output symmetries do not happen as often as input symmetries, their number is not negligible and rewiring techniques should take them into consideration.

5.2 Rewiring

In the rewiring experiments, wirelength reduction is calculated against the original wirelength after placement using half-perimeter wirelength. 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. These results are summarized in Tables III and IV, respectively. The maximum number of inputs allowed for symmetry detection is 16 in this experiment. From Table III, it is evident that our method can effectively reduce wirelength by about 3.7%, which is comparable to the improvement due to detailed placement.

Table IV shows that the wirelength reduction is a bit smaller when rewiring is used after detailed placement, suggesting that some rewiring opportunities interfere with the optimization from detailed placement. For example, detailed placement performs flipping of cells, which may interfere with permutative rewiring if the inputs of the cells are symmetric. However, the difference is

Table III. Performance and Runtime Comparisons Between Rewiring and Detailed Placement

		Wirelength reduction		Runtime (seconds)		
Benchmark	Wirelength	Rewiring	Detailed Placement	Rewiring	Detailed Placement	Global Placement
ALU2	5403.29	3.21%	8.98%	2.60	0.2	3.6
ALU4	35491.38	9.02%	3.54%	15.20	3.0	27.2
B02	142.90	8.29%	0%	2.80	0.4	0.0
B10	1548.28	5.04%	3.89%	7.20	0.0	1.0
B17	367223.20	2.92%	2.28%	350.60	32.6	206.2
C5315	30894.06	1.76%	1.52%	17.39	3.0	3.2
C7552	39226.30	1.71%	1.57%	23.80	4.0	2.8
DALU	20488.84	2.79%	3.46%	13.20	2.6	2.6
I10	50613.84	2.11%	2.05%	15.60	2.6	29.0
S38417	129313.20	2.01%	2.05%	180.80	22.2	17.2
S38584	174232.80	2.51%	2.27%	157.80	20.6	46.0
Average	77689.00	3.70%	2.87%	30.80	8.3.0	71.5.0

Table IV. Impact of Rewiring Before and After Detailed Placement

	Wirelength Reduction		Runtime (seconds)	
Benchmark	Before Detailed Placement	After Detailed Placement	Before Detailed Placement	After Detailed Placement
ALU2	3.49%	3.21%	3.4	3.6
ALU4	9.38%	9.02%	27.2	27.2
B02	8.29%	8.29%	0.2	0.2
B10	4.78%	5.04%	0.8	1.0
B17	3.0%	2.92%	199.6	206.2
C5315	1.71%	1.76%	3.6	3.2
C7552	1.82%	1.71%	2.6	2.8
DALU	2.9%	2.19%	2.8	2.6
I10	2.05%	2.11%	29.2	29.0
S38417	2.04%	2.01%	18.0	17.2
S38584	2.50%	2.51%	46.2	46.0
Average	3.82%	3.70%	30.3	30.8

very small, showing that the wirelength reduction from rewiring is mostly independent of detailed placement.

The third experiment evaluates the relationships between the number of inputs allowed in symmetry detection, wirelength reduction, and runtime. In order to show the true performance of Saucy-based symmetry detection, the use of naive symmetry detection is turned off in this experiment. Since our symmetry-detection method is most efficient with a small number of inputs, this relationship represents the tradeoff between performance and runtime. Empirical results, shown in Table V, indicate that the longer the rewiring program runs, the better the reduction will be. However, most improvement occurs with a small number of inputs and can be achieved quickly.

The fourth experiment shows the wirelength reduction gained from long-range rewiring and is summarized in Table VI. The SAT solver used is MiniSAT [Eén and Sörensson 2003]. For comparison, long-range rewiring is conducted

Table V. Impact of Number of Inputs Allowed in Symmetry Detection on Performance and Runtime

Number of Inputs Allowed	Runtime (seconds)	Wirelength Reduction
2	2.90	1.06%
4	4.30	2.58%
6	7.07	3.12%
8	14.98	3.50%
10	28.03	3.63%
12	41.34	3.72%
14	59.85	3.66%
16	82.30	3.68%

The numbers are averages of all the benchmarks.

Table VI. Wirelength Reduction of Long-Range Rewiring Before and After Permutative Rewiring

	Wirelength Reduction		Runtime (seconds)	
Benchmark	Before Permutative Rewiring	After Permutative Rewiring	Before Permutative Rewiring	After Permutative Rewiring
ALU2	1.63%	1.51%	0.66	0.33
ALU4	0.00%	0.05%	13.66	12.66
B02	0.00%	0.00%	0.33	0.33
B10	0.01%	0.00%	0.33	0.33
B17	0.83%	0.76%	445.33	421.00
C5315	1.33%	1.21%	13.66	12.00
C7552	1.69%	1.02%	28.66	17.66
DALU	2.10%	2.21%	23.33	21.66
I10	0.55%	0.48%	18.66	17.33
S38417	5.66%	5.84%	339.66	315.66
S38584	3.21%	3.03%	280.00	272.33
Average	1.55%	1.47%	105.78	99.18

before and after permutative rewiring. From the results, we can see that long-range rewiring tends to be more beneficial for larger circuits. This is because larger circuits contain more logic, therefore wires with equivalent functionality are more likely to exist. Furthermore, larger circuits contain more long wires, which makes wirelength reduction more significant for many rewiring opportunities. From the comparison of long-range rewiring applied before and after permutative rewiring, we can see that the wirelength reduction gained from long-range rewiring applied after permutative rewiring is less, but the runtime is also shorter. This is because some rewiring opportunities have already been discovered by permutative rewiring. However, since permutative rewiring runs much faster, it saves time for long-range rewiring. As a result, it is clear that long-range rewiring should be applied after permutative rewiring for better performance. When long-range rewiring is applied after permutative rewiring, an extra 1.47% of wirelength reduction can be gained, which makes the total wirelength reduction 5.17% using our rewiring techniques. Considering the fact

that no gate has been added or modified, our approach appears practical and very effective.

We also applied our rewiring techniques to the OpenCores suite [OpenCores 2007] in the IWLS'05 benchmarks [IWLSBenchmarks 2005], and performed routing to measure the wirelength reduction for routed wires. The results show that our prerouting optimizations effectively transform into postrouting wirelength reduction. Furthermore, we observe that via counts can also be reduced by our optimizations. These results show that our rewiring techniques are effective in reducing wirelength and the number of vias that can both reduce manufacturing defects and improve yield. Reducing via count is especially important in the deep submicron era because vias are a major cause of manufacturing faults. To study the effect of logic optimization on the performance of rewiring, we also conducted an experiment that performs rewiring on optimized and unoptimized netlists. In addition, we measured the maximum delay of several benchmarks to study the effect of rewiring on circuit timing. Empirical results for these experiments are available in the Appendix.

6. CONCLUSIONS

In this article we proposed a new symmetry-detection methodology and applied it to postplacement rewiring. We show experimentally that, compared with other symmetry-detection techniques, our method identifies more symmetries, including multivariable permutational and phase-shift symmetries for both inputs and outputs. This is important in circuit rewiring because more detected symmetries create more rewiring opportunities. In addition, we proposed a long-range equivalence detector for further rewiring opportunities.

Our experimental results on common circuit benchmarks show that the wirelength reduction is comparable and orthogonal to the reduction provided by detailed placement—the reduction achieved by our method performed before and after detailed placement is similar. This shows that our rewiring method is very effective, and should be performed after detailed placement for the best results. When applied alone, we observe an average of 3.7% wirelength reduction for the experimental benchmarks evalauted. When long-range rewiring is also applied, the total wirelength reduction increases to 5%.

In summary, the rewiring technique we presented has the following advantages: (1) It does not alter the placement of any standard cell, therefore no cell overlaps are created and the improvements from changes can be evaluated reliably; (2) it can be applied to a variety of existing design flows; (3) it can optimize a broad variety of objectives, such as delay and power, so long as they can be evaluated incrementally; and (4) it can easily adapt to other symmetry detectors, such as the one proposed by Chai and Kuehlmann [2006]. On the other hand, our technique has some limitations: (1) Its performance depends on the specific design being optimized and there is no guarantee of wirelength reduction; and (2) the improvement tends to decrease with larger designs, similar to what has been observed for detailed placement.

APPENDIX

Proof of Correctness of Symmetry Detection. We first prove the correctness of the reduced graph construction proposed in Section 3. Our proofs that follow are presented as a series of numbered steps.

(1) First, we need to prove that there is a one-to-one mapping between the function and its graph. This mapping can be defined following the graph construction in Section 3. The inverse mapping (from a graph to a function) is also given in Section 3.
(2) Second, we need to prove that there is a one-to-one mapping between symmetries of the function and automorphisms of the graph.
    (a) First, we want to show that a symmetry of the function is an automorphism of the graph. Symmetry of function means that permutation of the inputs will not change the output, and permutation in inputs corresponds to reevaluation of the outputs of that term. Since the inputs are symmetric, no output will be changed by the permutation, and the color of term vertices in the corresponding graph will remain the same. Therefore it is also an automorphism of the graph.
    (b) Next we want to show that an automorphism of the graph is a symmetry of the function. Since there is an edge between the input and its complement, mapping one input vertex, say x, to another vertex, say y, will cause x's complement to map to y's complement, so Boolean consistency is preserved. Since an input vertex connects to all the term vertices that contain it, swaps between two input vertices will cause all the term vertices that connect to them to be swapped according to the following rule: Suppose that the input vertex x swaps with input vertex y, then all term vertices that connect to both x and y will also be swapped because there is an edge between the term vertex and both x and y. Since a swap between term vertices is legal only if they have the same color, this means that all automorphisms detected in the graph will not map a term vertex to another color. And since the color of the term represents an output pattern in the Boolean function, this means that the outputs of the Boolean function will not be changed. Therefore, an automorphism of the graph maps to an input symmetry of the Boolean function.
(3) From Steps 1 and 2, there is a one-to-one mapping between the function and its graph, and a one-to-one mapping between symmetries of the function and automorphisms of the graph. Therefore, the symmetry-detection method for the reduced graph is correct.

Next, the correctness of the original graph is proved. The relationships between terms and inputs are described in the previous proof. Hence the proof here focuses on the relationship between terms and outputs. There are three possible situations: input symmetries that do not affect outputs, input symmetries that affect outputs, and output symmetries that are independent of the inputs.

Table VII. Characteristics of OpenCores Benchmarks

Benchmark	Gate Count	Net Count	Description
STEP	226	231	Stepper motor controller
SASC	549	566	Simple asynchronous serial controller
AC97	11855	11948	AC 97 Controller
USB	12808	12968	USB 1.1 Functional IP
AES	20795	21055	Advanced Encryption Standard IP core
PCI	16816	16990	PCI Bridge
Ethernet	46771	46891	Ethernet MAC core

(1) Input symmetries that do not affect the output: The way term vertices connect to output vertices represents an output pattern. If two term vertices have exactly the same outputs, then they will connect to the same output vertices; otherwise they will connect to at least one different output vertex. Mapping a term vertex to another term vertex which has different output pattern is invalid (except for the situation described in item 2, previously given) because at least one output vertex to which they connect is different, therefore the connections to output vertices behave the same way as coloring in the preceding proof.
(2) Input symmetries that affect the output: If all terms that connect to an output pattern can be mapped to all terms connecting to another output pattern, then the output vertices corresponding to the two patterns can also be swapped because the terms to which the outputs connect will not change after the mapping. In the meantime, input vertices that connect to the swapped minterms will also be swapped, which represents a symmetry involving both inputs and outputs.
(3) Output symmetries that are independent of the inputs: If two sets of output vertices connect to exactly the same term vertices, then the output vertices in the two sets can be swapped, which represents output symmetries. In this case, no term swapping is involved, so the inputs are unaffected.

OpenCores Benchmark Results. We also applied our rewiring techniques to real designs from OpenCores. Some characteristics of the benchmarks are given in Table VII, and the results are summarized in Table VIII. From the results, we can see that our techniques continue to be beneficial for these designs in terms of reduction both in wirelength and via counts. However, it can be observed that the improvement from permutative rewiring becomes much smaller compared with the benchmarks used in the article, while the improvement from long-range rewiring remains roughly the same. One reason is that these benchmarks are larger than the previous ones. As discussed in the article, permutative rewiring is a local optimization and its improvement tends to decrease when the design gets larger. Another reason is that the pins in the cells are close to each other and the gates are small. Since permutative rewiring is more effective for larger gates when pins are farther apart, the improvement becomes less significant. The results of reduction in via counts show that permutative rewiring is more effective than long-range rewiring in reducing via counts. The reason is that permutative rewiring can reduce local congestion and increase

Table VIII. Wirelength and Via Reduction After Detail Routing for OpenCores Benchmarks

	Original		Wirelength Reduction		Via Reduction	
Benchmark	Wirelength	Via Count	Permutative Rewiring	Perm+long Rewiring	Permutative Rewiring	Perm+long Rewiring
STEP	6582	1174	1.02%	5.07%	0.94%	2.56%
SASC	24471	3648	0.70%	1.54%	0.90%	0.66%
AC97	883291	84414	0.41%	2.05%	1.59%	1.70%
USB	1005226	87848	0.50%	1.87%	2.29%	3.49%
AES	1390080	131014	0.89%	3.69%	2.08%	3.39%
PCI	1483021	122701	0.26%	1.78%	0.25%	0.52%
Ethernet	7926180	430748	0.37%	4.01%	0.57%	1.64%
Average	1816979	123078.1	0.59%	2.86%	1.23%	1.99%

"Perm+long" the results of long-range rewiring applied after permutative rewiring.

Table IX. Characteristics of Benchmarks for the Logic Optimization Experiment

		Unoptimized			Optimized		
Benchmark	Description	Gate Count	Wirelength	Via Count	Gate Count	Wirelength	Via Count
MiniRISC	A mini-RISC CPU	4987	1045619	91338	4359	829725	83309
MD5	An MD5 encoder	13311	3073978	258399	9181	1789543	182868
DLX	An MIPS-Lite CPU	14788	4340807	320418	11028	3058868	248093
Alpha	An Alpha CPU	38831	15012263	855910	30212	9410743	672399

Optimized netlists were produced with great optimization efforts during logic synthesis, while unoptimized netlists were produced with little efforts.

Table X. Wirelength and Via Reduction After Detail Routing for Unoptimized and Optimized Circuits

	Unoptimized				Optimized			
	Wirelength Reduction		Via Count Reduction		Wirelength Reduction		Via Count Reduction	
Benchmark	Permutative Rewiring	Perm+long Rewiring	Permutative Rewiring	Perm+long Rewiring	Permutative Rewiring	Perm+long Rewiring	Permutative Rewiring	Perm+long Rewiring
MiniRISC	0.50%	5.04%	0.87%	1.85%	0.58%	1.08%	0.88%	1.94%
MD5	0.45%	6.29%	1.28%	3.05%	0.65%	2.93%	1.48%	2.80%
DLX	0.41%	5.67%	1.09%	5.51%	0.47%	1.78%	0.74%	1.67%
Alpha	0.24%	8.68%	1.02%	6.87%	0.36%	1.93%	1.34%	2.04%
Average	0.40%	6.42%	1.07%	4.32%	0.52%	1.93%	1.11%	1.94%

The column "Perm+long" shows the results of long-range rewiring applied after permutative rewiring.

pin access, which in turn reduce the use of vias; while long-range rewiring does not have such effects.

Effect of Logic Optimization on Rewiring. To study the effect of logic optimization on rewiring, we selected four register-transfer-level (RTL) benchmarks, whose characteristics are summarized in Table IX. For logic synthesis we used the Cadence RTL Compiler 4.1, configuring it with little optimization effort when generating "unoptimized" netlists, while the "optimized" netlists were generated with greater effort. We performed rewiring on both types of benchmark, and the results are summarized in Table X. From the results, we can observe that the wirelength reduction obtained using long-range rewiring is smaller for optimized netlists. The reason is that logic optimization involves merging equivalent nodes, which reduces the opportunities for long-range rewiring. However, the wirelength reduction achieved by

Table XI. Maximum Circuit Delay After Detail Routing

Benchmark	Changes in Maximum Delay		
	Before Rewiring (ns)	Permutative Rewiring	Perm+long Rewiring
MiniRISC	7.36	0.43%	3.40
MD5	20.98	−0.43%	35.05%
DLX	9.92	−9.71%	40.21%
Alpha	20.86	0.07%	7.36%
Average		−2.41%	21.50%

Column "Perm+long" shows the results of long-range rewiring applied after permutative rewiring.

permutative rewiring actually increases when the design is optimized, suggesting that permutative rewiring is orthogonal to logic optimizations. Since permutative rewiring requires both physical and logical information, our results indicate that it cannot be replaced by pure physical optimizations (i.e., detailed placement) or pure logic optimizations (i.e., those performed during synthesis).

Effect of Rewiring on Circuit Timing. To study the impact of rewiring on circuit timing, we instructed the detail router (Cadence NanoRoute 4.1) to report the maximum delay of the benchmarks in Table IX (optimized version) before and after rewiring, and the results are summarized in Table XI. The results show that indiscriminate long-range rewiring may increase circuit delay because it invalidates the buffer insertion performed during synthesis. Hence, it must be performed carefully to avoid delay increase in critical paths. However, circuit timing actually improves during permutative rewiring, despite the fact that timing is not considered in wirelength optimization—this is not surprising because permutative rewiring reduces the length of affected nets and does not introduce new wires. In the deep submicron era, power consumption and reliability are often as important as circuit timing. Our results show that permutative rewiring can safely improve both objectives without deteriorating circuit timing; shorter wires reduce dynamic power consumption, while reduction in via count makes signal delays more predictable (the resistance of Tungsten vias can vary by 20 to 30 times).

REFERENCES

ADYA, S. N., CHATURVEDI, S., ROY, J. A., PAPA, D. A., AND MARKOV, I. L. 2004. Unification of partitioning, placement and floorplanning. In *Proceedings of the ICCAD*. 550–557.

ALOUL, F. A., RAMANI, A., MARKOV, I. L., AND SAKALLAH, K. A. 2003. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 22,* 9, 1117–1137.

BERTACCO, V. AND DAMIANI, M. 1997. The disjunctive decomposition of logic functions. In *Proceedings of the ICCAD*. 78–82.

CALDWELL, A. E., KAHNG, A. B., AND MARKOV, I. L. 2000. Can recursive bisection alone produce routable placements? In *Proceedings of the DAC*. 477–482.

CALDWELL, A. E. AND MARKOV, I. L. 2002. Toward CAD-IP reuse: A web bookshelf of fundamental algorithms. *IEEE Des. Test Comput. 19,* 3, 72–81.

CHAI, D. AND KUEHLMANN, A. 2005. Building a better Boolean matcher and symmetry detector. In *Proceedings of the IWLS*. 391–398.

CHAI, D. AND KUEHLMANN, A. 2006. A compositional approach to symmetry detection in circuits. In *Proceedings of the IWLS*. 228–234.

CHANG, C.-W. J., HSIAO, M.-F., HU, B., WANG, K., MAREK-SADOWSKA, M., CHENG, C.-K., AND CHEN, S.-J. 2004. Fast postplacement optimization using functional symmetries. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 23,* 1, 102–118.

CHANG, C.-W. J. AND MAREK-SADOWSKA, M. 2001. Single-Pass redundancy addition and removal. In *Proceedings of the ICCAD*. 606–609.

CHANG, K.-H., MARKOV, I. L., AND BERTACCO, V. 2005. Post-Placement rewiring and rebuffering by exhaustive search for functional symmetries. In *Proceedings of the ICCAD*. 56–63.

CHANG, K.-H., MARKOV, I. L., AND BERTACCO, V. 2006. Keeping physical synthesis safe and sound. Tech. rep. CSE-TR-522-06, University of Michigan.

CHANG, S.-C., CHENG, K.-T., WOO, N. S., AND MAREK-SADOWSKA, M. 1997. Postlayout logic restructuring using alternative wires. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 16,* 6, 587–596.

CHANG, S.-C., VAN GINNEKEN, L. P. P. P., AND MAREK-SADOWSKA, M. 1999. Circuit optimization by rewiring. *IEEE Trans. Comput. 48,* 9, 962–970.

CONG, J. AND LONG, W. 2001. Theory and algorithm for SPFD-based global rewiring. In *Proceedings of the IWLS*. 150–155.

DARGA, P. T., LIFFITON, M. H., SAKALLAH, K. A., AND MARKOV, I. L. 2004. Exploiting structure in symmetry detection for CNF. In *Proceedings of the DAC*. 530–534.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the SAT*. 502–518.

GSRCBOOKSHELF. 2007. http://vlsicad.eecs.umich.edu/BK.

HRKIC, M., LILLIS, J., AND BERAUDO, G. 2004. An approach to placement-coupled logic replication. In *Proceedings of the DAC*. 711–716.

IWLSBENCHMARKS. 2005. http://iwls.org/iwls2005/benchmarks.html.

JIANG, Y.-M., KRSTIC, A., CHENG, K.-T., AND MAREK-SADOWSKA, M. 1997. Post-Layout logic restructuring for performance optimization. In *Proceedings of the DAC*. 662–665.

KRAVETS, V. N. AND SAKALLAH, K. A. 2000. Generalized symmetries in Boolean functions. In *Proceedings of the ICCAD*. 526–532.

LU, F., WANG, L.-C., CHENG, K.-T. T., MOONDANOS, J., AND HANNA, Z. 2004. A signal correlation guided circuit-SAT solver. *J. UCS 10,* 12, 1629–1654.

MISHCHENKO, A. 2003. Fast computation of symmetries in Boolean functions. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 22,* 11, 1588–1593.

OPENCORES. 2007. http://www.opencores.org/.

PANDA, S., SOMENZI, F., AND PLESSIER, B. 1994. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the ICCAD*. 628–631.

POMERANZ, I. AND REDDY, S. M. 1994. On determining symmetries in inputs of logic circuits. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 13,* 11, 1428–1434.

SAUCY. 2007. http://vlsicad.eecs.umich.edu/bk/saucy/.

WANG, G., KUEHLMANN, A., AND SANGIOVANNI-VINCENTELLI, A. L. 2003. Structural detection of symmetries in Boolean functions. In *Proceedings of the ICCD*. 498–503.

WLLACE, D. E. 2001. Recognizing input equivalence in digital logic. In *Proceedings of the IWLS*. 207–212.

WU, Q., CHEN, C. Y. R., AND ACKEN, J. M. 1994. Efficent Boolean matching algorithm for cell libraries. In *Proceedings of the ICCD*. IEEE Computer Society, Washington, DC, 36–39.

WU, Y.-L., LONG, W., AND FAN, H. 2000. A fast graph-based alternative wiring scheme for Boolean networks. In *Proceedings of the VLSI Design*. 268–273.

Received November 2005; revised November 2006; accepted January 2007