Exploiting Symmetries to Speed Up SAT-Based Boolean Matching for Logic Synthesis of FPGAs

Yu Hu, *Student Member, IEEE*, Victor Shih, Rupak Majumdar, and Lei He, *Member, IEEE*

***Abstract*—Boolean matching is one of the enabling techniques for technology mapping and logic resynthesis of field-programmable gate arrays (FPGAs). Boolean satisfiability (SAT)-based Boolean matching (SAT-BM) has been proposed, but computational complexity prohibits its practical deployment. In this paper, we leverage symmetries present in both Boolean functions and target FPGA architectures to prune the solution space, and we also propose some techniques to reduce the replication runtime for SAT instance generation using the incremental SAT reasoning engine. Experiment shows that our SAT-BM reduces runtime by 226× compared with the original SAT-BM algorithm, making SAT-BM more practical.**

***Index Terms*—Boolean satisfiability (SAT), field-programmable gate array (FPGA), logic synthesis.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) are programmable logic chips that can be configured to implement various digital circuits. The programmable logic block (PLB) is the basic element of an FPGA. Various programmable devices such as lookup tables (LUTs) or macro gates [1] can be placed within a PLB. Given a logic-level design, a critical step in the overall FPGA computer-aided design (CAD) flow is *technology mapping*, where a circuit is converted into a network of PLBs. The existing technology mapping algorithms can be divided into two categories: *structural* and *functional* [2]. Structural technology mappers [3]–[5] consider a given circuit graph and find a corresponding graph of PLBs which covers it. Functional approaches perform Boolean decomposition of the logic functions of circuit nodes into subfunctions of limited size which are realizable by individual PLBs. Since area-optimal technology mapping for LUT-based FPGAs is NP-hard [6], *logic resynthesis*—rewriting circuit structures while maintaining functionality—has been applied, accompanied by technology mapping, to reduce area [7]–[9].

In both technology mapping and logic resynthesis, *Boolean matching* [10], [11] serves as one of the enabling techniques. Given a target PLB architecture p and a Boolean function f, the Boolean matching problem either maps function f to PLB p by describing the appropriate configuration bits, or concludes that PLB p cannot implement function f. The key characteristics of a Boolean matching algorithm are scalability in terms of both runtime and memory, and flexibility with regard to reusability across different PLB architectures. Most of the existing work for Boolean matching is based on function decomposition [10] or on canonicity and Boolean signatures [11], [12]. However, the function decomposition technique lacks flexibility and needs to be customized for different PLB architectures, and canonicity-based approaches can only handle functions of limited input size. For example, Abdollahi and Pedram [12] assume that the technology library can be precomputed before the mapping phase. While this is feasible for application-specified integrated circuit designs, it is computationally intensive for technology mapping with complex programmable devices, which can implement millions of different logic functions. Recently, a SAT-based approach [8] has been proposed to solve Boolean matching, which was improved by Safarpour *et al.* [13] with a 3× speedup and was further improved by Cong and Minkovich [9] with up to 13× speedup.

While SAT-based Boolean matching (SAT-BM) offers great flexibility in handling various PLB architectures, runtimes are excessively long due to high computational complexity, even with the improvements by Cong and Minkovich [9] and Safarpour *et al.* [13]. Recent FGPAs have employed heterogeneous PLBs to reduce power dissipation and area; this additional flexibility comes at the cost of even greater complexity for Boolean matching. For instance, suppose that we map a design to an FPGA with K-input heterogeneous PLBs; the functionality of each K-bounded cover must be considered explicitly during technology mapping.[1] In practice, the Boolean matching procedure is called over 50 000 times for an MCNC circuit, $i10$, which has less than 3000 gates, with an average runtime of completing one SAT-BM [8] for a nine-input subcircuit at more than 20 s. It appears that the runtime for heterogeneous FPGA technology mapping is prohibitively high due to the inefficiencies of SAT-BM.

Manuscript received January 16, 2008; revised April 30, 2008. Current version published September 19, 2008. This work was supported in part by the NSF Grants CCR-0306682 and CCF-0702743 and in part by UC MICRO sponsored by Actel. This paper was recommended by Associate Editor W. Kunz.

Y. Hu and L. He are with the Electrical Engineering Department, University of California, Los Angeles, CA 90095 USA (e-mail: hu@ee.ucla.edu; lhe@ee.ucla.edu).

V. Shih and R. Majumdar are with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: vicshih@cs.ucla.edu; rupak@cs.ucla.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2003272

[1]For LUT-based homogeneous FPGA, technology mapping becomes a graph-covering problem as a K-LUT can implement any Boolean function with no more than K inputs; therefore, the functionality check and the Boolean matching are not needed.

This paper proposes an efficient SAT-BM that exploits the symmetries exhibited in both the Boolean function and the target PLB architecture. The experimental results show that the proposed algorithm obtains up to 226× speedup compared with the original algorithm [8], whereas recent papers [9], [13] obtained up to 13× speedup.

The rest of this paper is organized as follows. Section II introduces Boolean matching and SAT-based encoding [8]. Section III presents a technique to improve the efficiency of SAT-BM using symmetries. Section V details our experimental results, and Section VI concludes this paper. A four-page extended abstract of preliminary results of this paper was presented at the 2007 International Conference on CAD [14]. More details of this work can be found in the technical report [15].

II. Background and Preliminaries

A PLB $H(P)$ consists of a network of interconnected nonprogrammable and programmable logic devices with a set P of input pins $\{p_1, \ldots, p_m\}$. We sometimes omit the set of input pins and write H to refer to the PLB $H(P)$. We consider the mix of two kinds of programmable logic devices in this paper: the K-input LUT and the K-input multiplexer (MUX). A K-LUT consists of K inputs, one output, and 2^K configuration bits. A K-MUX consists of K inputs, one MUX output, and $\lceil \log K \rceil$ configuration bits.

The *Boolean matching* problem takes as input a PLB $H(P)$ and a Boolean function $f(X)$ over the variables X such that $|X| \leq |P|$, and it asks if the PLB $H(P)$ can implement the function $f(X)$. For the simple case where H is a K-LUT, any function $f(X)$ where $|X| \leq K$ can be implemented by the K-LUT. When H contains multiple LUTs, however, the question becomes nontrivial.

In the following, we first review the SAT encoding scheme presented in [8] and then point out the inherent problem of this approach.

A. From PLBs to CNF

For nonprogrammable devices (for example, combinational gates) in a PLB, we can describe the logic of each device as a Boolean formula in conjunctive normal form (CNF) relating its inputs and outputs. For example, a two-input AND gate with inputs x_1 and x_2 and output z can be expressed as

$$(x_1 \cdot x_2 \leftrightarrow z)$$

which in CNF becomes

$$(x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z).$$

For networks composed of multiple nonprogrammable devices, we add intermediate variables for the output of each device and encode the relationship between the inputs and outputs of each device as CNF formulas in terms of those intermediate variables. Fig. 1 shows an example of a nonprogrammable device network, where an AND-2 gate and an OR-2 gate compose

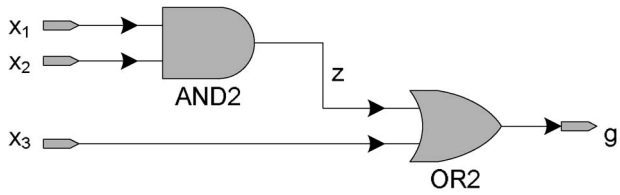


Fig. 1. Example encoding for nonprogrammable devices.

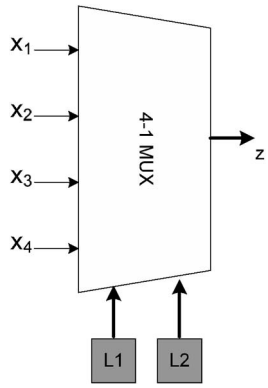


Fig. 2. Four-input programmable MUX.

a three-input logic function $g(x_1, x_2, x_3)$. The corresponding CNF f_{all} is constructed as follows:

$$f_{\text{AND2}} = (x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z)$$
$$f_{\text{OR2}} = (\neg x_3 + g) \cdot (\neg z + g) \cdot (x_3 + z + \neg g)$$
$$f_{\text{all}} = f_{\text{AND2}} \cdot f_{\text{OR2}}.$$

A similar encoding can be performed for the programmable devices (LUTs and MUXs) in a PLB. For a K-input LUT, we introduce 2^K additional variables, namely, $L_1, \ldots, L_{2^K}$, to represent every possible setting of the configuration bits. For example, the two-input LUT with inputs x_1 and x_2 and output z_1 can be encoded as follows:

$$\begin{aligned}(x_1 + x_2 + \neg L_1 + z_1) &\cdot (x_1 + x_2 + L_1 + \neg z_1) \\ &\cdot (x_1 + \neg x_2 + \neg L_2 + z_1) \cdot (x_1 + \neg x_2 + L_2 + \neg z_1) \\ &\cdot (\neg x_1 + x_2 + \neg L_3 + z_1) \cdot (\neg x_1 + x_2 + L_3 + \neg z_1) \\ &\cdot (\neg x_1 + \neg x_2 + \neg L_4 + z_1) \cdot (\neg x_1 + \neg x_2 + L_4 + \neg z_1).\end{aligned}$$

For a K-input programmable MUX, we have $\lceil \log K \rceil$ configuration bits; thus, we introduce $\lceil \log K \rceil$ additional variables. Fig. 2 shows a four-input programmable MUX with inputs x_1 and x_2 and output z, where L_1 and L_2 are the variables corresponding to the configuration bits. The derivation of the CNF encoding for this four-input MUX is essentially symmetric to that of the two-input LUT and is therefore omitted.

B. From Boolean Matching to SAT

Let $G(x_1, \ldots, x_n, L_1, \ldots, L_m, z_1, \ldots, z_l, f)$ be a Boolean function in CNF representing a PLB, where variables $x_1, \ldots, x_n$ represent the input signals, variables $L_1, \ldots, L_m$ represent configuration bits, variables $z_1, \ldots, z_l$ represent the

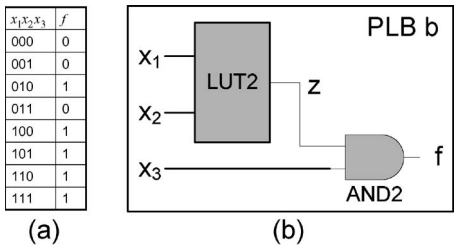


Fig. 3. (a) Truth table for a function f. (b) Example PLB.

intermediate circuit signals, and f represents the output function of the configuration. Let $F(x_1, \ldots, x_n, f)$ represent a Boolean function over the variables $x_1, \ldots, x_n$ with output signal f. We assume that F is represented in CNF, for example, by computing a CNF formula from a truth table representation of the function. The Boolean matching problem then asks if there exists a setting of the configuration signals $L_1, \ldots, L_m$ such that, for all input variables $x_1, \ldots, x_n$, there are valuations of the intermediate signals such that the output f of the PLB is equivalent to the output of the Boolean function f. Formally, the Boolean matching problem is formulated as the following quantified Boolean satisfiability (QSAT) problem:

$$\begin{aligned}&\exists L_1, \ldots, L_m \forall x_1, \ldots, x_n \exists z_1, \ldots, z_l, f \\ &\cdot G(x_1, \ldots, x_n, L_1, \ldots, L_m, z_1, \ldots, z_l, f) \\ &\qquad F(x_1, \ldots, x_n, f). \end{aligned} \tag{1}$$

As in [8], the universal quantifiers in (1) can be removed by enumerating the truth table of the function $F(x_1, \ldots, x_n)$. Therefore, (1) can be solved with SAT, where a satisfying assignment implies that the function can be realized by the configuration.

C. Example

Consider the example PLB shown in Fig. 3(b), which contains a LUT-2 and an AND-2 gate. We want to test if function f, whose truth table is shown in Fig. 3(a), can be implemented by this PLB. Let $X = \{x_1, x_2, x_3\}$ be the set of input pins. We generate a SAT problem using the following steps.

1) Create CNF formulas for individual elements in the PLB.

$$\begin{aligned}G_{\mathrm{LUT}} = &(x_1+x_2+\neg L_1+z)(x_1+x_2+L_1+\neg z) \\ &\cdot (x_1+\neg x_2+\neg L_2+z)(x_1+\neg x_2+L_2+\neg z) \\ &\cdot (\neg x_1+x_2+\neg L_3+z)(\neg x_1+x_2+L_3+\neg z) \\ &\cdot (\neg x_1+\neg x_2+\neg L_4+z)(\neg x_1+\neg x_2+L_4+\neg z) \\ G_{\mathrm{AND}} = &(z+\neg f) \cdot (x_3+\neg f) \cdot (\neg z+\neg x_3+f).\end{aligned}$$

2) The characteristic function of the PLB can then be expressed as

$$G = G_{\mathrm{LUT}} \cdot G_{\mathrm{AND}}. \tag{2}$$

3) Decide on either a QSAT-based formulation or a SAT-based formulation.

a) **QSAT-based formulation.** For the QSAT-based formulation, write the CNF for the truth table of the Boolean function f as follows:

$$\begin{aligned}G_f = &(\neg x_1+\neg x_2+\neg x_3+f) \cdot (\neg x_1+\neg x_2+x_3+f) \\ &\cdot (\neg x_1+x_2+\neg x_3+f) \cdot (\neg x_1+x_2+x_3+f) \\ &\cdot (x_1+\neg x_2+\neg x_3+\neg f) \cdot (x_1+\neg x_2+x_3+f) \\ &\cdot (x_1+x_2+\neg x_3+\neg f) \cdot (x_1+x_2+x_3+\neg f).\end{aligned}$$

The QSAT formulation can then be expressed as follows:

$$\exists L_1 \exists L_2 \exists L_3 \exists L_4 \forall x_1 \forall x_2 \forall x_3 \exists z, f.(G \cdot G_f).$$

A satisfiable assignment to the aforementioned QSAT instance implies that f can be implemented by the PLB.

b) **SAT-based formulation.** In the SAT-based formulation, we replicate (2) to remove the universal quantifiers on the input variables in X.[2] This defines G_{SAT} as

$$\begin{aligned}&G[X/000, f/0, z/z_1] \cdot G[X/001, f/0, z/z_2] \\ &\cdot G[X/010, f/1, z/z_3] \cdot G[X/011, f/0, z/z_4] \\ &\cdot G[X/100, f/1, z/z_5] \cdot G[X/101, f/1, z/z_6] \\ &\cdot G[X/110, f/1, z/z_7] \cdot G[X/111, f/1, z/z_8].\end{aligned} \tag{3}$$

Finding a satisfiable assignment of G_{SAT} implies that f can be implemented by the PLB. In this case, the SAT solver will find that the problem is unsatisfiable, which means that the Boolean function shown in Fig. 3(a) cannot be implemented by the PLB shown in Fig. 3(b).

D. Input Permutation

An important issue in Boolean matching is *input permutation*, which expands the solution space of a given circuit by considering different mappings from the variables of the Boolean function to the pins of the PLB. Fig. 4 shows two Boolean functions which are equivalent under input permutation—i.e., function f_a can be transformed into f_b by the permutation $\tau = (3, 2, 1)$. Note that f_a cannot be implemented by the PLB shown in Fig. 3(b), whereas f_b can.

In practice, input permutation is employed in FPGA designs and must be considered during Boolean matching to maximize the number of implementable functions. However, the number of permutations for a K-input Boolean function is $K!$, which grows extremely quickly as K increases. In order to consider input permutations in the SAT formulation, Ling *et al.* [8] proposed adding programmable MUXs before each primary input of the target PLB (see Fig. 5). All possible permutations

[2]Note that such a transformation is an application of the general expand operator used in Quantor [16] and also used by Ayari and Basin [17].

$x_1x_2x_3$	f
000	0
001	0
010	0
011	0
100	0
101	1
110	1
111	1

(a)

$x_1x_2x_3$	f
000	0
001	0
010	0
011	1
100	0
101	1
110	0
111	1

(b)

Fig. 4. (a) Truth table of $f_a = x_1 \cdot (x_2 + x_3)$. (b) Truth table of $f_b = x_3 \cdot (x_1 + x_2)$.

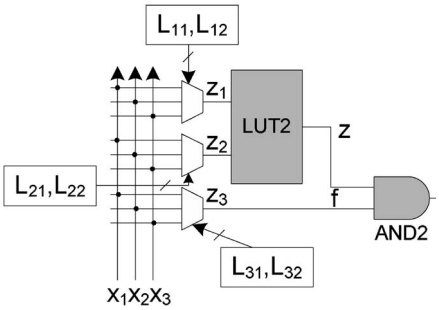


Fig. 5. Considering input permutation with additional MUXs.

are encoded by these MUXs. For each of these programmable MUXs, $\lceil \log n \rceil + 1$ additional variables are needed to represent the configuration bits (e.g., $L_{11}, L_{12}, L_{21}, L_{22}, L_{31}, L_{32}$ in Fig. 5) and intermediate pins (e.g., z_1, z_2, z_3), as well as $O(n^2)$ clauses. Thus, accounting for input permutation by adding n MUXs adds $n \cdot (\lceil \log n \rceil + 1)$ variables and $O(n^2)$ clauses to the original formulation. Depending on the circuit, this can have significant impact on the problem size. For instance, a typical SAT/QSAT problem effectively doubles when adding MUXs to a circuit composed of four LUTs. Since the runtime complexity is exponential to the size of a SAT instance, these programmable MUXs increase the runtime exponentially.

III. Considering Symmetries

We present an efficient algorithm which eliminates the need for permutation MUXs by explicitly considering symmetry in the SAT formulation.

A. Symmetry in Boolean Functions

Variables x_i and x_j of Boolean function $f(x_1, \ldots, x_n)$ are *symmetric* if the truth table of f remains the same when x_i and x_j are swapped, i.e., if $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. By observing the variable symmetries exhibited in a Boolean function, we can make the programmable MUXs added in Section II-D unnecessary by pruning all but the *distinct permutations*.

Given an n-input Boolean function $f(x_1, \ldots, x_n)$, we can first test the symmetry of every input pair (x_i, x_j) by comparing the truth tables before and after swapping variables x_i and x_j. Once all symmetric relationships between variable pairs are computed, we can find the connected components of the undirected graph where each variable is a node and each symmetry between variables is an edge. For example, consider a nine-input Boolean function having the four symmetries (0, 1, 6, 8), (3, 4, 5), (2), and (7). For any two permutations τ_1 and τ_2, if the only difference between them is within the same symmetry cluster [(0, 1, 6, 8) or (3, 4, 5), in this example], we have $f \circ \tau_1 = f \circ \tau_2$, and only one of these permutations needs to be tested in the Boolean matching. In fact, the number of distinct permutations under such a symmetry is $9!/(4! \times 3! \times 1! \times 1!) = 2520$, reducing the number of permutations to consider by a factor of 144.

Note that the time required to identify symmetries of an n-input function using the aforementioned algorithm is $O(n^2 \cdot 2^n)$. This computational cost is negligible in practice compared with the Boolean matching time, however, as usually $n < 9$. Taking advantage of the symmetries exhibited by a Boolean function allows us to significantly reduce the number of permutations to test. In addition, symmetries can be detected efficiently using sophisticated algorithms [18].

Once the functional symmetries have been computed, the reduced set of distinct input permutations can be generated. First, the variables within each cluster are sorted to form a canonical representation—(0, 1, 6, 8), in the preceding example. Each cluster is then assigned a label, which all variables in that cluster inherit. A list is generated from these variable labels, and the permutations of this list are computed. In the example, the following labels are applied: $(0, 1, 6, 8) = A$, $(3, 4, 5) = B$, $(2) = C$, and $(7) = D$. Thus, the list of labels is $(A, A, A, A, B, B, B, C, D)$. For each of the distinct permutations of this list, the corresponding order of inputs is created by enumerating the permutation, restoring a variable from each cluster of that label, in cluster order. In the example, the label permutation $(A, B, A, A, B, A, B, C, D)$ corresponds to the order (0, 3, 1, 6, 4, 8, 5, 2, 7).

B. Symmetry in PLB Architectures

Most commercial PLB architectures exhibit symmetry with respect to their input pins. Additional levels of symmetry can be found if more logical levels are considered. Formally, we define *first-* a and *second-order architectural symmetries* as follows.

Definition 1: First-Order Architectural Symmetry: Any two input pins x_i, x_j connected directly to the same k-input LUT are symmetric under the permutation (x_i, x_j).

Definition 2: Second-Order Architectural Symmetry: The inputs $x_1, \ldots, x_k$ and inputs $y_1, \ldots, y_k$ for two k-input LUTs L_x and L_y, respectively, are symmetric under permutation $\pi(y_{i_1}, \ldots, y_{i_k}, x_{j_1}, \ldots, x_{j_k})$ if the outputs x and y of these two LUTs are symmetric.

For example, in the PLB shown in Fig. 6, the inputs x_1 and x_2 are symmetric, as are the inputs x_3 and x_4, which means that ignoring the configurations where they are swapped avoids redundant calculations. The symmetries between x_1 and x_2 and between x_3 and x_4 are first-order architectural symmetries. Furthermore, since the outputs of both LUTs feed into a

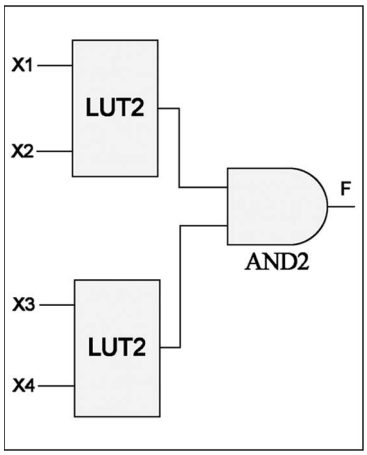


Fig. 6. Second-order symmetric PLB.

two-input AND gate whose inputs are symmetric, ignoring the configurations where two groups of pins (x_1, x_2) and (x_3, x_4) are swapped under the permutations $\pi = (x_3, x_4, x_1, x_2)$, $\pi = (x_3, x_4, x_2, x_1)$, and $\pi = (x_4, x_3, x_1, x_2)$ will not affect the Boolean matching decision. This is an example of a second-order architectural symmetry.

C. Detection of Architectural Symmetry

Architectural symmetry detection can be done as a processing step before resynthesis and can be performed either manually or automatically. For automatic symmetry detection, we propose the following algorithm.

Architectural symmetry information is computed for a particular PLB by generating a *constraint list*, as described in Algorithm 3. A constraint in this context is an enforced ordering between the function variable indices mapped to two specified pins. That is, a "less than" constraint between pins m and n implies that the index of the variable mapped to m must be less than the index of the variable mapped to n. A constraint list is simply a collection of constraints, which together can describe the symmetries exhibited by a particular PLB.

The constraint list is computed recursively for all gates of the PLB by comparing the topologies of each gate's fan-in cone. Topologies are compared by first recursively generating string representations of each cone, as described in Algorithm 1, and then comparing strings. All fan-in cones with the same topology are symmetric; therefore, we enforce an ordering between these cones. Doing so requires calculating a fan-in cone's first pin, as described in Algorithm 2.

Algorithm 1 topology (*gate*)
1: {Returns τ, a string representation of *gate*'s topology}
2: $\tau =$ ""
3: {*gate*.gateType is a string unique to each gate type: "AND," "LUT," etc.}
4: **if** *gate*.gateType = "PI" **then**
5: {Base case—a primary input}
6: $\tau =$ "PI"
7: **else**
8: {Recursive case—aggregate topologies of fanins, in sorted order}
9: $\lambda = []$
10: **for all** $f \in$ *gate*.fanins **do**
11: λ.add(topology(f))
12: $\tau =$*gate*.gateType + "(" + λ.sort().join(",") + ")"
13: **return** τ

Algorithm 2 firstPin(*gate*)
1: {Returns the first (lowest index) PI of *gate*'s cone}
2: **if** {*gate*.gateType = "PI"} **then**
3: {Base case—a primary input}
4: **return** *gate*
5: **else**
6: **return** firstPin(*gate*.fanins[0])

Algorithm 3 computeConstraintList(*architecture*)
1: {Returns a list of constraints which all unique permutations will satisfy}
2: *constraintList* = []
3: **for all** $g \in$ *architecture*.gates **do**
4: {ϕ is a queue of untested fanins}
5: $\phi = g$.fanins
6: *testFanin* = ϕ.pop()
7: **for all** $f \in \phi$
8: {Perform string comparison on topologies}
9: **if** topology(*testFanin*) = topology(f)
10: {Constrain permutations such that *testFanin*'s cone should always come before f's cone in pin ordering}
11: $\chi =$ ConstrainLessThan(firstPin(*testFanin*).pinIndex, firstPin(f).pinIndex)
12: *constraintList*.add(χ)
13: {New constraints should reference f}
14: *testFanin* = f
15: {f has been tested}
16: ϕ.remove(f)
17: **return** *constraintList*

For example, in Fig. 6, the inputs $X1$ and $X2$ both have the topology "PI," since they are primary inputs. Therefore, they are symmetric, and since they are primary inputs, their first pins are the inputs themselves. Thus, a constraint specifying that the variable mapped to $X1$ must have an index less than that of the variable mapped to $X2$, which we express as $[X1] < [X2]$. Similarly, the constraint $[X3] < [X4]$ is added after analyzing the second LUT.

The topology for the AND gate F in the same figure is represented by the string "AND(LUT(PI, PI), LUT(PI, PI))." Because the second LUT has the same topology as the first, the two LUTs are also symmetric. Thus, a constraint is added between the first pin of the first LUT $(X1)$ and the first pin of the second LUT $(X3)$, namely, $[X1] < [X3]$.

Once the constraint list is calculated, it is applied to the remaining input permutations, which has already been pruned according to functional symmetries. Each permutation is tested

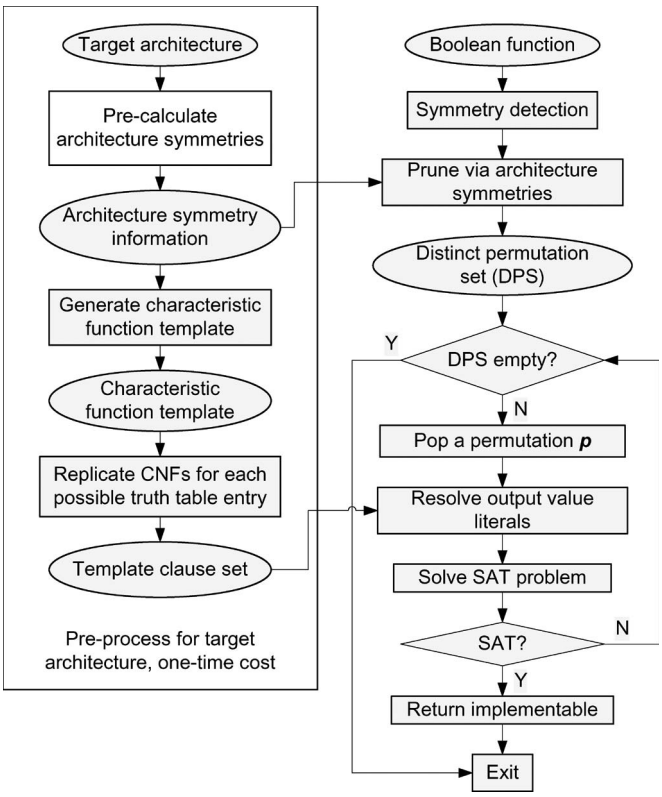


Fig. 7. Overall algorithm flow.

whether it satisfies the list of constraints, as described in Algorithm 4. If the permutation violates any one of the constraints, it is considered redundant and, thus, pruned.

Algorithm 4 pruneInputPermutations(*constraintList*, *inputPerms*)

1: **for all** $\pi \in$ *inputPerms* **do**
2: **for all** $\chi \in$ *constraintList* **do**
3: **if not** χ.isSatisfiedBy(π) **then**
4: *inputPerms*.remove(π)

Note that a more sophisticated algorithm than the one just described could be developed which would extend the structural analysis algorithm presented in [19] to consider programmable logic devices. Alternatively, detecting symmetry manually is a reasonable approach since the number of PLB structures used in an FPGA is very limited.

D. Algorithm Overview

Fig. 7 shows the overall flow of our approach. We first extract the architectural symmetry information from the target PLB (manually or using the algorithm in Section III-C) and generate a template of its characteristic function. We also generate the template clause-set by replicating the characteristic function for each possible truth table entry (to be explained in detail in Section IV). For each Boolean function to be tested, we compute its functional symmetries (using the algorithm in Section III-A or in [18]). Of the possible input permutations, we prune those considered redundant as informed by the architectural and functional symmetry calculations, and the distinct permutations are collected ("distinct permutation set"). We iterate over this set ("pop a permutation p"), generating a SAT problem for each permutation by replicating the PLB's characteristic function for each possible truth table entry (as in (3) of Section II-C). Note that during SAT problem generation, we do not actually replicate the CNFs for every truth table entry, for each permutation. The precalculated template enables us to update only the truth table output values for each minterm of the Boolean function ("Resolve output value literals," details to be explained in Section IV-B). After solving the generated SAT problem, if any permutation gives rise to a satisfiable solution, the Boolean function can be implemented by the target PLB. On the other hand, if instead none of the SAT instances are satisfiable, we conclude that the function cannot be implemented by the target PLB.

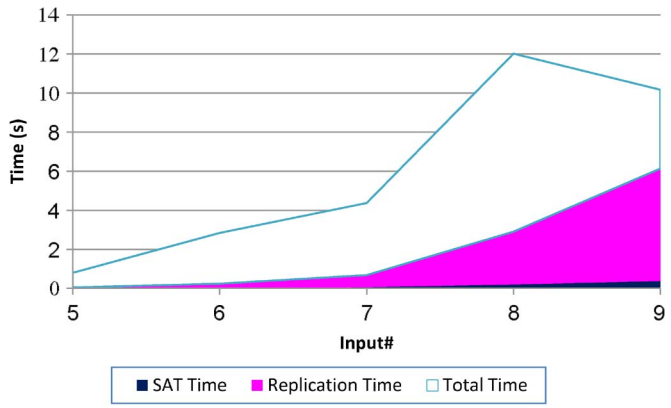


Fig. 8. Replication time as a proportion of total runtime.

IV. Implementation Issues

As a practical consideration, a large proportion of the runtime of SAT-BM techniques is spent on replicating clauses. In our initial implementation, we observed that the percentage of time consumed by the replication phase was as high as 57% of the total runtime for nine inputs, whereas actual SAT solution time peaked at only 4% of the total runtime for the same number of inputs. Fig. 8 shows the proportion of total runtime that replication consumes as compared with that of SAT solving time, with the trend increasing as the number of inputs grows.

A number of techniques are applied to address this disproportionate amount of time spent on problem construction. In the following, we describe two different attempts and their corresponding improvements to the overall runtime.

A. Iterative Clause Testing

Clause replication requires much more runtime than SAT solving; it is prudent, therefore, to avoid unnecessary replication if possible. SAT solving, on the other hand, is relatively inexpensive in this context; it is worth making several SAT solver calls if doing so saves even a few replications. This observation motivates the following approach, which takes advantage of the characteristic structure of the SAT problem by breaking down each problem into smaller subproblems.

Since an SAT problem is made up simply of several clauses which are ANDed together, it can easily be partitioned into any combination of subsets of clauses which, in turn, must be ANDed together. If any subset is found to be unsatisfiable, then clearly the entire SAT problem is unsatisfiable. If, on the other hand, the conjunction of all subsets is found satisfiable, then the entire problem is satisfiable. This leads to a straightforward solution to the costly replication issue—replicate subsets of clauses, testing each one for satisfiability.

If any subset is unsatisfiable, we return unsatisfiable and can avoid the cost of replicating the remaining clauses. If instead the subsets are all satisfiable, at some point, the algorithm should determine that it is worth testing the entire set of clauses. We call this technique *iterative clause testing*. What remains to be determined is how exactly to partition the clauses into subsets. Partitioning into too many subsets will incur the cost of unnecessary replications in the satisfiable case; partitioning into too few will not achieve significant savings in the unsatisfiable case.

Note also that two SAT clauses may be satisfiable, yet their conjunction may be unsatisfiable if a conflict exists between them. Thus, partitioning the problem into disjoint subsets is not an efficient approach, as conflicts will not be detected as early as possible. Iteratively testing a set of clauses which subsumes the previous subset is a better technique. This can also take advantage of any incremental capabilities of the SAT solver; if supported, the replication of the previous subset can be avoided as new clauses are simply added to the currently instantiated problem.

In our implementation, we start with a subset containing clauses representing one truth table entry. Whenever the subset is found satisfiable, we double its size by adding the appropriate number of untested clauses. We continue doubling the size of the subset until it is either found unsatisfiable, or it contains all clauses of the SAT problem and is found satisfiable. Our results show the iterative clause testing technique to be a significant improvement, performing faster than the implicant representation improvement presented by Cong and Minkovich [9].

We also applied the iterative clause testing technique to the implicant representation implementation; however, this enhancement only provided modest improvement. In fact, the iterative technique with the truth table representation outperformed the iterative technique with the implicant representation by a factor of two on average. We suspect that nature of the clauses generated by the truth table representation enables early unsatisfiability detection more often than with the implicant representation.

B. Template Clause-Set

Noting the incremental capabilities of our particular SAT solver led us to an improvement which surpassed all of our previous approaches regarding CNF replication time. Note that there is a large amount of information common to all input permutations to be tested. The key to this technique lies in extracting it effectively to reduce unnecessary replication.

Given a Boolean function $f(x_1, x_2, \ldots, x_n)$, the truth table of f has 2^n entries, with possible inputs 000...0 through 111...1. Note that any permutation of f will have not only the same number of inputs but also, in fact, the exact same set of input values 000...0 through 111...1, although in a different order. What remains distinct between the permutations is the order of the output values induced by the permuting of the input values. Fig. 9 shows an example where f' is equivalent to f, but with x_1 and x_3 swapped.

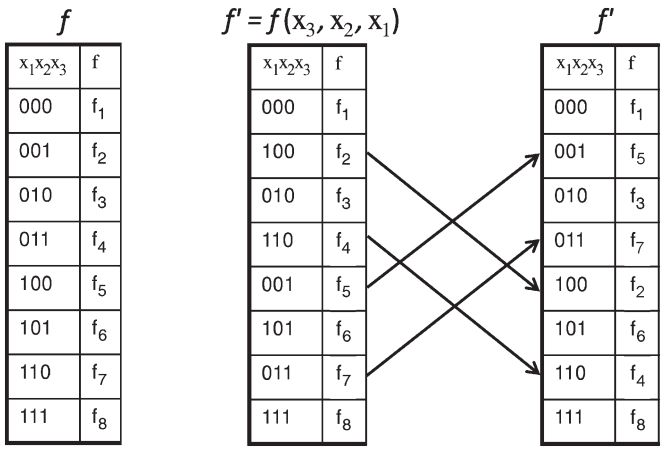


Fig. 9. Illustration of identical input values for function f and permutation f'.

We can take advantage of this common information during the replication phase in the following way. Rather than generating a new set of SAT clauses for each permutation, we instead create a CNF *template clause-set* once for all permutations of a particular SAT problem. Recall that in the replication phase, the set of clauses representing the characteristic function is replicated several times, once for each truth table entry. In each replication, the Boolean literals representing the circuit inputs and outputs are substituted with the input and output values of the corresponding truth table entry, respectively. Instead of substituting both input and output values, the improved strategy continues to substitute the input values but leaves the output values $f_1, f_2, \ldots, f_{2^n}$ unmodified, effectively leaving them as unbound free variables for the SAT solver. This is the template clause-set.

To test a permutation, we calculate the reordering of output values induced by the permutation of input variables. Applying this new order to the function's output values, we then determine how the unbound variables $f_1, \ldots, f_{2^n}$ of the template clause-set should be constrained. By taking advantage of our SAT solver's ability to accept assumptions as a parameter when solving, each unbound variable is bound as an assumption accordingly. Testing each new permutation only requires calculating the output value order and binding the output values as assumptions. Thus, the CNF replication process, which was performed for every truth table entry, for each permutation, is now performed only once per Boolean function tested. Note that although any information learned by the incremental SAT solver is reset between permutations, information is still recorded and used to speed up conflict-finding when testing the input vectors under one permutation.

We find that the template clause-set technique with incremental SAT reasoning is significantly better at reducing runtime than the iterative-clause-testing-based approach. The comparison between these two approaches is shown in Section V-A.

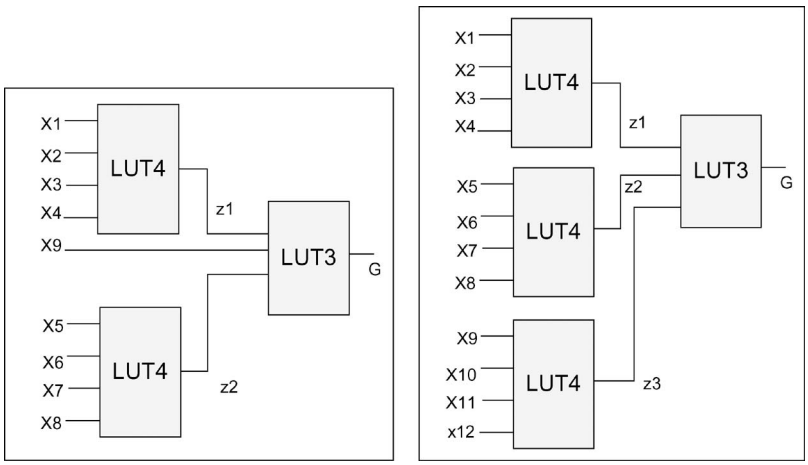


Fig. 10. Nine-input PLB and 12-input PLB.

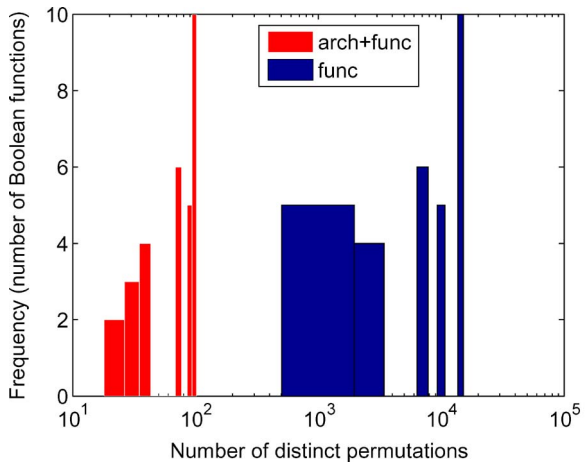


Fig. 11. Comparison of the number of distinct permutations when pruning function symmetries ("func") and when pruning function and architectural symmetries ("arch").

TABLE I
COMPARISON OF SAT SOLVING TIME BETWEEN SAT-BM AND OUR IMPROVED ALGORITHM (SAT-IP-TEMPLATE)

Test cases	func size	5	6	7	8	9
	problem#	1398	1981	2263	2172	2134
SAT-BM	variable#	867	1571	2979	5795	11,427
	clause#	6945	13,889	27,777	55,553	111,105
	SAT time (s)	0.0423	0.126	0.424	2.56	46.95
	Total runtime (s)	0.745	1.55	3.28	9.79	73.12
SAT-IP-template	variable#/inst	1576	1576	1576	1576	1576
	clause#/inst	6144	6144	6144	6144	6144
	unique perm#	34.9	64.1	94.97	100.0	80.7
	SAT time (s)	0.00576	0.00918	0.0156	0.0221	0.0344
	Total runtime (s)	0.134	0.180	0.256	0.292	0.323
	SAT speedup	7.3×	14×	27×	116×	1364×
	Total speedup	5.5×	8.6×	13×	34×	226×

V. EXPERIMENTS

We implement our algorithms in C++ and Perl, using miniSAT2.0 [20] as our SAT solver. The implicant table-based SAT encoding [9] has been implemented and integrated into our algorithm, as shown in Fig. 7. To show the effectiveness of our improvement to the SAT-BM algorithm (shown in Fig. 7), we extract over 10 000 fan-out-free cones with five to nine inputs from MCNC benchmarks based on the method presented in [4] as the Boolean functions. The target PLB architecture is the nine-input PLB shown in Fig. 10. This architecture exhibits both first-order symmetry (e.g., permutations of the input variables x_1, x_2, x_3, x_4 are redundant) and second-order symmetry (e.g., permutations which swap the input groups (x_1, x_2, x_3, x_4) and (x_5, x_6, x_7, x_8) are redundant). Both levels of symmetry are used in our SAT-BM. All experiments are run on a 1.9-GHz CPU Linux server with 2-GB memory.

We first randomly select 30 nine-input Boolean functions from the nine-input cut set and calculate the number of unique permutations considering symmetries, as shown in Fig. 11. Compared with the total number of unique permutations $(9! = 362,880)$, we reduced computation by over two orders of magnitude by considering Boolean function symmetries and by another two orders of magnitude by considering architectural symmetries.

Table I compares the original algorithm SAT-BM presented in [8] and our improved algorithm, SAT-IP-template, which is our symmetry-aware algorithm combined with the template clause-set improvement, as described in Section IV-B. The average SAT instance sizes and runtime of both algorithms are shown in the table. As the number of inputs in the Boolean function increases, the SAT instance size increases exponentially for SAT-BM. On the other hand, the size of the SAT problem representing a single permutation for our SAT-IP-template algorithm remains virtually the same, independent of the number of inputs in the Boolean function. In fact, the size of each SAT-IP-template SAT instance is decided by the number of distinct permutations, which is further dependent on the symmetries exhibited by the Boolean function and the PLB architecture. As shown in Table I row "unique perm#," the number of unique permutations grows slowly after pruning based on symmetries. Compared with SAT-BM, SAT-IP-template achieves 1364× and 226× speedup in terms of SAT reasoning runtime and total runtime, respectively, for nine-input logic functions. More significant speedup is expected if Boolean functions with

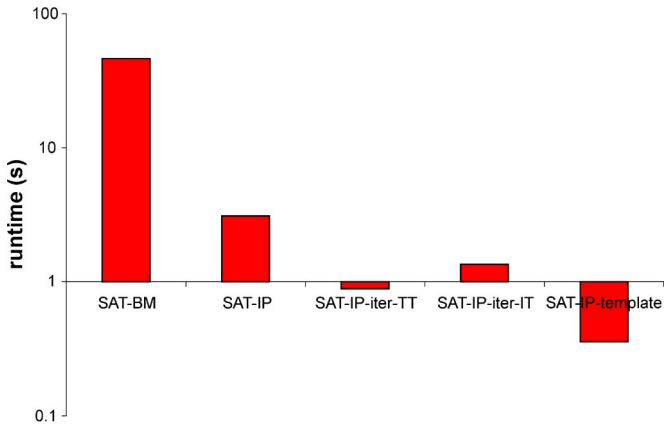


Fig. 12. Comparison of average runtime with different CNF replication speedup techniques.

wider inputs are considered.[3] Note that two recent improvements on the SAT-BM problem, namely, [9] and [13], obtained up to 13× speedup compared with [8]. The substantial speedup obtained by SAT-IP-template makes it possible to integrate the SAT-BM algorithm within technology mapping and logic optimization during heterogeneous FPGA synthesis.

A. Comparison of Speedup Techniques

Fig. 12 shows a comparison of the average runtime on 3000 nine-input Boolean functions with the following different speedup techniques: 1) Ling's approach (SAT-BM); 2) our initial algorithm without optimization for CNF replications (SAT-IP-base); 3) our algorithm with iterative clause testing and the truth-table-based representation (SAT-IP-iter-TT); 4) our algorithm with iterative clause testing and the implicant table-based representation [9] (SAT-IP-iter-IT); and 5) our algorithm with the template clause-set improvement (SAT-IP-template). The average (geometric mean) runtime for these techniques under all testing cases is shown in Fig. 12.

An interesting observation is that the improvements gained by our incorporation of the implicant representation as presented by Cong and Minkovich [9] are completely superseded by our template clause-set implementation. There are a number of possible reasons for this. First, for every programmable PLB element with k inputs in the original architecture, the implicant representation adds another $2^k - 1$ such elements. During the replication phase, this effectively allows for element-centric replication rather than circuit-centric replication. That is, rather than replicating CNF clauses which represent the entire circuit, as is the case when replicating the truth table representation, only clauses which represent each circuit element are replicated. While replication at the PLB element level may result in fewer duplicated SAT clauses overall, typical reduction of SAT problem size is on the order of one half. Thus, performance is improved, but not substantially. Second, the template clause-set implementation is very effective because the majority of CNF replication time is incurred only once, while the implicant representation must repeat the same CNF replication for each input permutation.

[3]We are limited to nine-input Boolean functions; for larger functions, SAT-BM generates SAT problems which exhaust available memory when run with miniSAT.

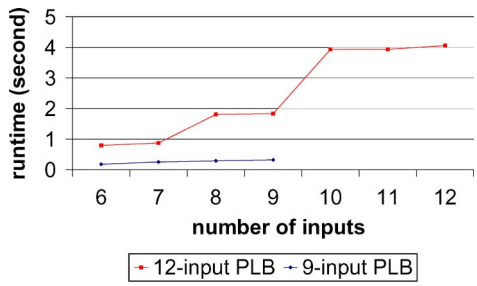


Fig. 13. Scalability study of SAT-IP-template.

We have also compared our best algorithm, i.e., SAT-IP with the template clause-set, with a symmetry detection and CNF optimization tool, namely, Shatter/Saucy [21], which takes advantage of the symmetries existing in CNFs of a SAT instance. Our results (described in our extended technical report [15]) show that Shatter is not able to take much advantage of the symmetry inherent in these SAT problems. This is to be expected, as the tool targets symmetries among literals and is not designed to be aware of the particular nature of our problems. These results validate the need for specialized symmetry detection at the Boolean matching level.

B. Scalability Study

To test the scalability of our SAT-IP-template algorithm, we have used SAT-IP-template to map Boolean functions (extracted from MCNC benchmarks) with 5 to 12 inputs against the 12-input PLB shown in Fig. 10. Fig. 13 compares the runtime for SAT-BM with the 12-input PLB and the nine-input PLB shown in Fig. 10. The average runtime for each input number is shown. Compared with SAT-BM with the nine-input PLB, SAT-BM with the 12-input PLB shows a 10× runtime increase. Note that the runtime growth is much slower than the growth of the problem complexity, which depends on the number of configuration bits (1.3× increase) and permutations ($> 1000\times$ increase). In addition, the curve for SAT-BM with 12-input PLB indicates that our algorithm scales well for Boolean matching with large PLBs and wide inputs. Note that the runtime for functions with 10, 11, and 12 inputs remains virtually the same. Again, the reason is that the overall runtime is dependent on the number of distinct permutations after pruning; the SAT solving time for each permutation is largely determined by the target PLB architecture and does not change significantly from one input number to the next.

VI. Conclusion

Leveraging the symmetries exhibited in both Boolean functions and target PLB architectures, we have obtained two orders of magnitude speedup compared with the original SAT-BM algorithm [8]. In contrast, recent work [9], [13] obtained up to 13× speedup. Our work makes SAT-BM more practical for synthesis and optimization of heterogeneous FPGAs. Since our

SAT-BM applies to any PLB architecture, it is valuable for FPGA architecture evaluation. Nevertheless, our SAT-BM is still slow compared with structural technology mapping and may not prove practical for FPGA end users. In the future, we plan to improve SAT-BM further and apply it to FPGA architecture exploration.

REFERENCES

[1] Y. Hu, S. Das, S. Trimberger, and L. He, "Design, synthesis and evaluation of heterogeneous FPGA with mixed LUTs and macro-gates," in *Proc. Int. Conf. Comput.-Aided Design*, 2007, pp. 188–193.

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *Proc. ACM Int. Symp. Field-Program. Gate Arrays*, 2006, pp. 41–49.

[3] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1–12, Jan. 1994.

[4] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. ACM Int. Symp. Field-Program. Gate Arrays*, 1999, pp. 29–35.

[5] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. Int. Conf. Comput.-Aided Design*, 2004, pp. 752–759.

[6] A. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 11, pp. 1319–1332, Nov. 1994.

[7] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting," in *Proc. Design Autom. Conf.*, 2005.

[8] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: A study of optimality," in *Proc. Design Autom. Conf.*, 2005, pp. 427–432.

[9] J. Cong and K. Minkovich, "Improved SAT-based Boolean matching using implicants for LUT-based FPGAs," in *Proc. ACM Int. Symp. Field-Program. Gate Arrays*, 2007, pp. 139–147.

[10] J. Cong and Y.-Y. Hwang, "Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1077–1090, Sep. 2001.

[11] L. Benini and D. Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electron. Syst.*, vol. 2, no. 3, pp. 193–226, Jul. 1997.

[12] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification," in *Proc. Design Autom. Conf.*, 2005.

[13] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient SAT-based Boolean matching for FPGA technology mapping," in *Proc. Design Autom. Conf.*, 2006, pp. 466–471.

[14] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping," in *Proc. Int. Conf. Comput.-Aided Design*, 2007, pp. 350–353.

[15] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetries to speed-up SAT-based Boolean matching for logic synthesis of FPGAs," Tech. Rep. UCLA Engr 07-265, 2007. [Online]. Available: http://www.ee.ucla.edu/~hu/pub/techReport07_sat_bl.pdf

[16] A. Biere, "Resolve and expand," in *Proc. SAT*, 2004, pp. 238–246.

[17] A. Ayari and D. Basin, "Qubos: Deciding quantified Boolean logic using propositional satisfiability solvers," in *Proc. FMCAD*, 2002, pp. 187–201.

[18] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch, "Generalized symmetries in Boolean functions: Fast computation and application to Boolean matching," in *Proc. Int. Workshop Logic Synth.*, 2004.

[19] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large Boolean functions using circuit representation, simulation, and satisfiability," in *Proc. Design Autom. Conf.*, 2006, pp. 510–515.

[20] N. Een and N. Sorensson, [Online]. Available: http://www.minisat.se/

[21] F. Aloul, I. Markov, and K. Sakallah, "Efficient symmetry-breaking for Boolean satisfiability," in *Proc. IJCAI*, 2003, pp. 271–282.

Yu Hu (S'04) received the B.S. and M.S. degrees in computer science from Tsinghua University, Beijing, China, in 2002 and 2005, respectively. He is currently working toward the Ph.D. degree in the Electrical Engineering Department, University of California, Los Angeles (UCLA).

He was with Xilinx Research Laboratory in the summer of 2006. He is currently a Graduate Student Researcher with the Electrical Engineering Department, UCLA. He is the author of over 30 technical papers and the holder of four patents in the field of computer-aided design (CAD) for very large scale integration designs. His current research interests include CAD for field-programmable gate array synthesis and application-specified integrated circuit physical synthesis.

Mr. Hu has been a full member of Sigma Xi since 2007. He was the recipient of the Outstanding Graduate Student Award in 2005 from Tsinghua University and of the Best Contribution Award of IEEE Programming Challenge at the International Workshop on Logic and Synthesis in 2008.

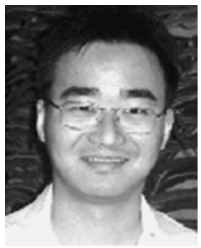
Victor Shih received the B.S. degree in computer science from the University of California, San Diego, in 1996 and the M.S. degree in computer science from the University of California, Los Angeles (UCLA), in 2008.

He is currently working on a startup venture as a Senior Software Developer with the Computer Science Department, UCLA.

Rupak Majumdar received the B.Tech. degree in computer science from the Indian Institute of Technology (IIT), Kanpur, India, in 1998 and the Ph.D. degree in computer science from the University of California (UC), Berkeley, in 2003.

Since 2003, he has been an Assistant Professor with the Computer Science Department, University of California, Los Angeles. His research interests include the verification and control of reactive, real-time, hybrid, and probabilistic systems; software verification and programming languages; game theoretic problems in verification; and logic and automata theory.

Dr. Majumdar was the recipient of the President's Gold Medal from IIT, the Leon O. Chua Award from UC Berkeley, and the National Science Foundation CAREER Award in 2006.



Lei He (S'94–M'99) received the Ph.D. degree in computer science from the University of California, Los Angeles (UCLA), in 1999.

He was a Faculty Member with the University of Wisconsin, Madison, between 1999 and 2001. He is currently an Associate Professor with the Electrical Engineering Department, UCLA. He also held visiting or consulting positions with Intel, Hewlett-Packard, Cadence, Synopsys, Rio Design Automation, and Apache Design Solutions. He has published over 160 technical papers. His research interests include very large scale integration circuits and systems and electronic design automation.

Dr. He has been a technical program committee member for a number of conferences, including the Design Automation Conference, the International Conference on Computer-Aided Design, the International Symposium on Low Power Electronics and Design, and the International Symposium on Field-Programmable Gate Array. He was the recipient of the National Science Foundation CAREER Award in 2000, the UCLA Chancellor's Faculty Career Development Award (highest class) in 2003, the IBM Faculty Award in 2003, the Northrop Grumman Excellence in Teaching Award in 2005, and the Best Paper Award at the 2006 International Symposium on Physical Design.