

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

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

Abstract— Boolean matching is one of the enabling techniques for technology mapping and logic resynthesis of Field Programmable Gate Array (FPGA). SAT-based Boolean matching (SAT-BM) has been proposed, but the computational complexity prohibits its practical deployment. In this paper, we leverage the symmetries in Boolean functions and target FPGA architectures to prune the solution space. Besides the exploration of symmetries, we also propose a few techniques to reduce the replication runtime for SAT instance generation using the incremental SAT reasoning engine. In terms of the overall runtime, our SAT-BM obtains up to 226X speedup compared to the original SAT-BM algorithm, making SAT-BM more practical.

I. INTRODUCTION

Field-programmable gate arrays (FPGA) are programmable logic chips that can be configured to implement various digital circuits. In the past decade, FPGAs have achieved a higher growth rate than ASICs. The programmable logic block (PLB) is the basic element of an FPGA. Various programmable devices, e.g., lookup tables (LUTs) or macro gates [1], can be placed within a PLB. Given a logic-level design, a crucial step in the overall FPGA computer-aided design (CAD) flow is *technology mapping*. This step converts a circuit into a network of PLBs. The existing technology mapping algorithms can be divided into two categories, *structural* and *functional* [2]. The structural technology mappers [3], [4], [5] consider the circuit graph as given and find a covering of the graph with PLBs, e.g., K -input subgraphs corresponding to LUTs. Functional approaches perform Boolean decomposition of the logic functions of the nodes into sub-functions of limited support size realizable by individual PLBs. Since area-optimal technology mapping for LUT-based FPGAs is NP-Hard [6], *logic resynthesis*, a technique to rewrite circuit structures while maintaining the functionalities of transition and output functions, has been applied, accompanied by technology mapping, to reduce area [8], [14], [16].

In both technology mapping and logic resynthesis, *Boolean matching* [11], [12] serves as one of the enabling techniques. Given a target FPGA architecture, or more specifically, 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 criteria to judge a Boolean matching algorithm includes scalability, in

terms of both runtime and memory efficiency, and flexibility, i.e., one algorithm is re-usable for different PLB architectures. Most of the existing work for Boolean matching is based on function decomposition [11] or on canonicity and Boolean signatures [12], [13]. However, the function decomposition-based algorithms lack flexibility and need to be customized for different PLB architectures. In addition, the canonicity-based approaches are limited by the input size of the functions they can handle. For example, [13] assumes that the technology library can be pre-computed before the mapping phase, which is feasible for ASIC designs, while it is too computationally intensive for technology mapping with complex programmable devices, which can implement millions of different logic functions. Recently, a SAT-based approach [14] has been proposed to solve Boolean matching and was improved by [15] with a 3x speedup, and was further improved by [16] with up to 13x speedup.

While SAT-based Boolean matching offers great flexibility in handling various PLB architectures, it still suffers from excessive runtime due to high computational complexity even with the improvements in [15], [16], and the demands of the scalability of Boolean matching increase with the greater use of complex PLBs in industrial FPGAs. Figure 1 shows the PLB architecture for Altera Stratix II ALM [18]. This heterogeneity in the PLB allows more flexibility to reduce on-chip power dissipation and area overhead, and improve performance. On the other hand, the extra flexibility of heterogeneous PLBs increases the complexity of Boolean mapping. For instance, suppose 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 mapping¹. In practice, the Boolean matching procedure is called over 50K times for a typical MCNC circuit, *i10*, which has less than 3000 gates, with an average runtime for completing one SAT-based Boolean matching [14] for a 9-input sub-circuit at more than 20 seconds. It would appear that the runtime for heterogeneous FPGA technology mapping is prohibitively high due to the inefficiencies of Boolean matching.

Targeting orders of magnitude speedup over the existing algorithms [14], [15], [16], this paper proposes an efficient SAT-based Boolean matching by exploring the symmetries exhibited in both the Boolean function and the target PLB architecture. The experimental results show that the proposed algorithm obtains up to 226X speedup by considering sym-

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

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

¹For LUT-based homogeneous FPGA technology mapping, the Boolean matching degenerates to a graph cover problem as a K -LUT can implement any Boolean functions with no more than K inputs, and therefore the functionality check is not needed.

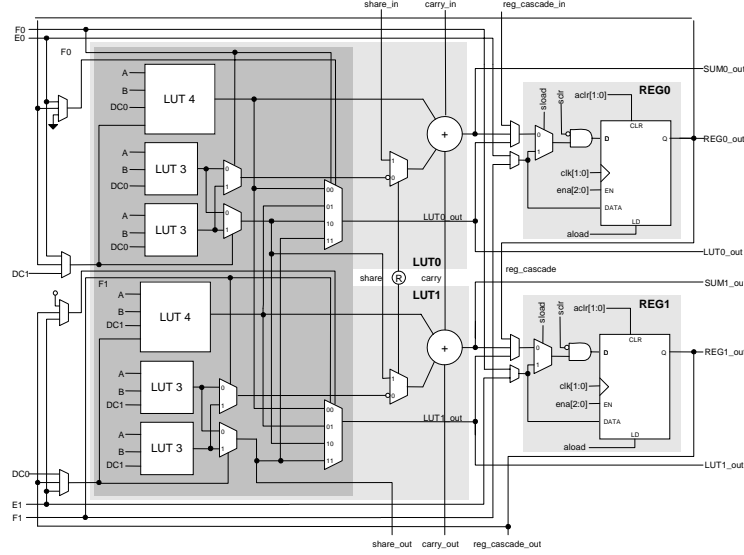


Fig. 1: PLB Architecture for Stratix II (from Altera Inc.)

metries compared to the original algorithm [14], while recent papers [15], [16] obtained up to 13x speedup.

The rest of this paper is organized as follows: section II introduces Boolean matching and SAT-based encoding [14]. Section III presents a technique to improve the efficiency of SAT-based Boolean matching using symmetries. Section V details our experimental results, and section VI concludes the paper. A four page extended abstract of preliminary results of this paper was presented at the 2007 International Conference on Computer-Aided Design [19].

II. BACKGROUND AND PRELIMINARIES

A *programmable logic block (PLB)* $H(P)$ consists of a network of interconnected non-programmable and programmable logic devices with a set P of input pins $\{p_1, \dots, p_m\}$. 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 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 non-trivial.

Below, we will first review the SAT encoding schemes presented in [14], and then point out the inherent problem of this approach.

A. From PLBs to CNF

For non-programmable devices (e.g., combinational gates) in a PLB, we can describe the logic of the device as a Boolean

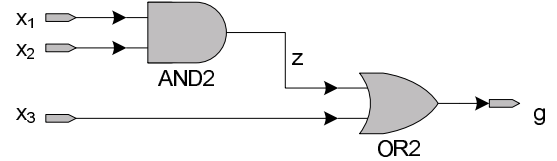


Fig. 2: Example encoding for non-programmable devices

formula in conjunctive normal form (CNF) relating the inputs and outputs. For example, a 2-input AND gate with inputs x_1, 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 comprised of multiple non-programmable 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. Figure 2 shows an example of a non-programmable device network, where an AND-2 gate and an OR-2 gate compose a 3-input logic function $g(x_1, x_2, x_3)$. The corresponding CNF, f_{all} , is constructed as follows:

$$\begin{aligned} f_{AND2} &= (x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z) \\ f_{OR2} &= (\neg x_3 + g) \cdot (\neg z + g) \cdot (x_3 + z + \neg g) \\ f_{all} &= f_{AND2} \cdot f_{OR2} \end{aligned}$$

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

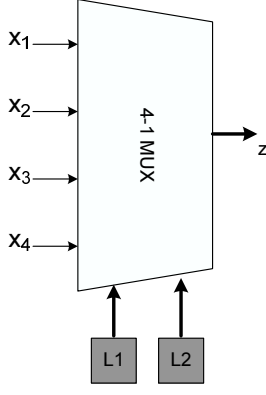


Fig. 3: A 4-input programmable MUX

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, so we introduce $\lceil \log K \rceil$ additional variables. Figure 3 shows a 4-input programmable MUX with inputs x_1, x_2 , and output z , where L_1, L_2 are the variables corresponding to the configuration bits. The derivation of the CNF encoding for this 4-input programmable MUX is omitted here since it is essentially the symmetric form of a LUT.

B. From Boolean Matching to SAT

Let $G(x_1, \dots, x_n, L_1, \dots, L_m, z_1, \dots, z_l, f)$ be a Boolean function in CNF representing a PLB, where variables x_1, \dots, x_n represent the input signals, variables L_1, \dots, L_m represent configuration bits, variables z_1, \dots, z_l represent the intermediate circuit signals, and f represents the output function of the configuration. Let $F(x_1, \dots, x_n, f)$ represent a Boolean function over the variables x_1, \dots, x_n with output signal f . We assume 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 is some setting of the configuration signals L_1, \dots, L_m such that for all input variables x_1, \dots, 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:

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

As in [14], the universal quantifiers in (1) can be removed by enumerating the truth table of the function $F(x_1, \dots, x_n)$. Therefore, (1) can be solved with SAT, where a satisfying assignment implies that the function can be realized by the configuration.

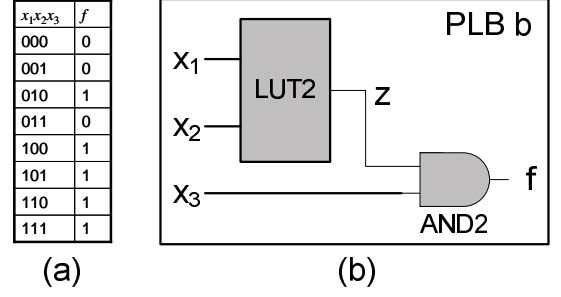


Fig. 4: (a) Truth table for a function f , (b) An example PLB

Assuming a PLB contains only N -input LUTs, and the LUTs are connected as a tree structure, the QSAT formulation uses $O(k \cdot 2^N + 2^n)$ clauses, $O(k \cdot 2^N)$ existential variables, and $O(n)$ universal variables, where k is the number of LUTs in the PLB, and n is the number of inputs in the Boolean function. When expanded to the SAT-based formulation, the number of clauses and variables are $O(k \cdot 2^N \cdot 2^n)$ and $O(k \cdot 2^N + 2^n)$, respectively.

C. Example

Consider the example PLB shown in Figure 4(b), which contains a LUT-2 and an AND-2 gate. We want to test if function f , whose truth table is shown in Figure 4(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.

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

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

$$G = G_{LUT} \cdot G_{AND} \quad (2)$$

- 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:

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

The QSAT formulation can then be expressed as follows:

$$\begin{aligned}
 & \exists L_1 \exists L_2 \exists L_3 \exists L_4 \forall x_1 \forall x_2 \forall x_3 \exists z, f. \\
 & \quad (G \cdot G_f)
 \end{aligned}$$

$x_1x_2x_3$	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. 5: (a) Truth table of $f_a = x_1 \cdot (x_2 + x_3)$, (b) Truth table of $f_b = x_3 \cdot (x_1 + x_2)$

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

(B) SAT-based formulation. For the SAT-based formulation, we replicate (2) to remove the universal quantifiers on the input variables in X^2 . This formulates G_{SAT} as:

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

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, meaning that the Boolean function shown in Figure 4(a) cannot be implemented by the PLB shown in Figure 4(b).

D. Input Permutation

An important issue in Boolean matching is *input permutation*, which expands the solution space for a given circuit by allowing different mappings from the variables of the Boolean function to the pins of the PLB. Figure 5 shows two Boolean functions which are equivalent under input permutation – that is, function f_a can be transformed into f_b by the permutation $\tau = (3, 2, 1)$. Notice that f_a cannot be implemented by the PLB shown in Figure 4(b), while 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, [14] proposed to add programmable MUXs before each primary input of the target PLB (see Figure 6). All possible permutations 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

Figure 6) 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, the SAT/QSAT problem effectively doubled when adding MUXs to a circuit comprised of 4-LUTs. Since the runtime complexity is exponential to the size of a SAT instance, these programmable MUXs increase the runtime exponentially.

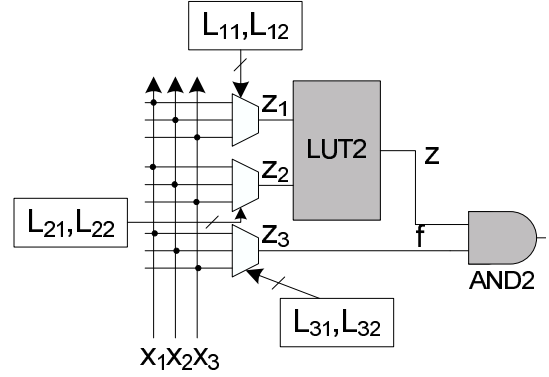


Fig. 6: Considering input permutation with additional MUXs

III. CONSIDERING SYMMETRIES

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

A. Symmetries in Boolean Functions

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

Given an n -input Boolean function $f(x_1, \dots, x_n)$, we can first test the symmetries of every input pair (x_i, x_j) by comparing the truth tables before and after swapping variables x_i and x_j . After building the symmetric relationships between every variable pair, we can find the connected components in this undirected graph, with each node representing a variable and each edge connecting two symmetric variables. For example, consider a 9-input Boolean function having the four symmetries (0, 1, 6, 8), (3, 4, 5), (2), and (7) as shown in Figure 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 above algorithm is $O(n^2 \cdot 2^n)$.

²Note that such a transformation is an application of the general expand operator used in Quantor [20] and introduced by [21], followed by Boolean Constraint Propagation (BCP).

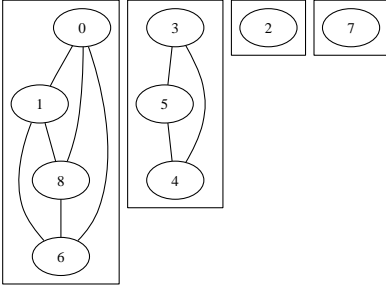


Fig. 7: Symmetries in a 9-input Boolean function

This computational cost is negligible in practice compared to the Boolean matching time, however, as n is usually less than nine. Taking advantage of the symmetries exhibited by a Boolean function allows us to significantly reduce the number of permutations to be tested. In addition, symmetries can be detected efficiently using sophisticated algorithms [22].

Given the detected functional symmetries, a canonical form of the input permutation can be defined by fixing the relative order of all inputs within one symmetric set, e.g., (0,1,6,8) in the above example. Specifically, we rename (0,1,6,8)=A, (3,4,5)=B, (2)=C, and (7)=D, and enumerate all distinct permutations of the renamed set (A,A,A,A,B,B,C,D). For each permutation, we substitute A,B,C and D with the corresponding order of the inputs, e.g., (A,B,A,A,B,A,B,C,D) represents permutation (0,3,1,6,4,8,5,2,7).

B. Symmetries in PLB Architecture

Most commercial PLB architectures exhibit symmetries with respect to their input pins. Symmetries can also be propagated, allowing us to discover more symmetries if more logical levels are considered. Formally, we define *first order* 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, \dots, x_k and inputs y_1, \dots, y_k for two k -input LUTs L_x and L_y , respectively, are symmetric under permutation $\pi(y_{i_1}, \dots, y_{i_k}, x_{j_1}, \dots, x_{j_k})$ if the outputs x and y of these two LUTs are symmetric.

For example, in the PLB shown in Figure 8, 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 will not affect the decision of whether a certain Boolean function can be implemented by this PLB. 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 2-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 permutation $\pi = (x_3, x_4, x_1, x_2)$, $\pi = (x_3, x_4, x_2, x_1)$, $\pi = (x_4, x_3, x_1, x_2)$ will not affect the Boolean matching decision. This is an example of a second order architectural symmetry.

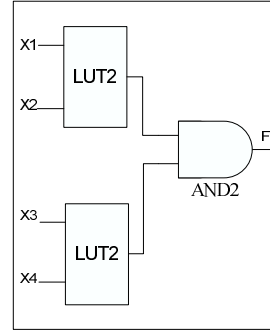


Fig. 8: A second order symmetric PLB

C. Detection for Architecture Symmetries

The architecture symmetries are detected as a pre-processing step before re-synthesis, and it can be performed either manually or automatically. To automatically detect the architecture symmetries, 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 completely 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 fanin cone. Topologies are compared by first recursively generating string representations of each cone, described in Algorithm 1, and then comparing strings. All fanin cones with the same topology are symmetric; therefore, we enforce an ordering between these cones. Doing so requires calculating a fanin cone’s first pin, described in Algorithm 2.

For example, in Figure 8 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 whether or not it satisfies the list of constraints, as described in Algorithm 4. If the permutation violates any one of the constraints, it is determined to be redundant and thus pruned.

Note that more sophisticated algorithm, which reveal more

Algorithm 1 `topology(gate)`

```

1: {Returns  $\tau$ , 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  $f \in \textit{gate.fanins}$  do
11:     $\lambda.add(\textit{topology}(f))$ 
12:   $\tau = \textit{gate.gateType} + "(" + \lambda.sort().join(",") + ")"$ 
13: return  $\tau$ 

```

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])

```

architecture symmetries than the above algorithms, can be developed for architecture symmetry detection, e.g., one can extend the structural analysis algorithm presented in [23] to consider programmable logic devices. Alternatively, it is possible to manually detect the architecture symmetries in the preprocessor, as the PLB structures used in an FPGA are very limited.

D. Overall Algorithm

Figure 9 shows the flow of our overall algorithm. We first pre-process the architecture of the target PLB by extracting its architectural symmetry information (manually or using the algorithm in Section III-C) and generate a template of the characteristic function for the PLB. We also need to generate the template for clause set by replicating CNFs for each possible truth table entry (will be detailed in Section IV). For each Boolean function to be tested, we first detect the function symmetries (using the algorithm in Section III-A or in [22]) and prune the redundant permutations based on both architectural and function symmetries. Then each distinct permutation is tested individually by replication of the characteristic function. All distinct permutations are stored in a queue, DPS. Each time, one permutation, p , is taken from this queue ("pop a permutation p "), and a SAT problem is generated by replicating the characteristic function based on the template clause set, e.g., (3) in Section II-C. In the process of SAT problem generation, we do not explicitly calculate the CNFs for each permutation every time, which requires excessive runtime. Instead, we use a pre-calculated template clause set and only update the truth table values for each min-term of a Boolean function in this template ("resolve output

Algorithm 3 `computeConstraintList(architecture)`

```

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

```

Algorithm 4 `pruneInputPermutations(constraintList, inputPerms)`

```

1: for all  $\pi \in \textit{inputPerms}$  do
2:   for all  $\chi \in \textit{constraintList}$  do
3:     if not  $\chi.isSatisfiedBy(\pi)$  then
4:       inputPerms.remove( $\pi$ )

```

value literals", details will be discussed in Section IV-B.2). 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.

IV. IMPLEMENTATION ISSUES

A. Implicant Representation

Our algorithm has been extended to handle the output don't cares of a Boolean function. Since the point of including sub-expression $G(X/X_{value}, f/f_{value}, z/z_{value})$ in the replication form (3) is to ensure that the configuration of PLB which is programmed in correspondence with the satisfying assignment of (3) will produce f_{value} when X_{value} is on its inputs, the replication corresponding to the don't cares term DC can be removed from (3) because we do not care about the output value when the input values are DC . For instance, if the truth value for inputs $(x_1, x_2, x_3) = (1, 0, 1)$ is a don't care, the sub-expression $G(X/101, f/DC, z/z_6)$ can be removed from the replicated SAT encoding. The replicated characteristic function (3) can then be re-expressed as the following:

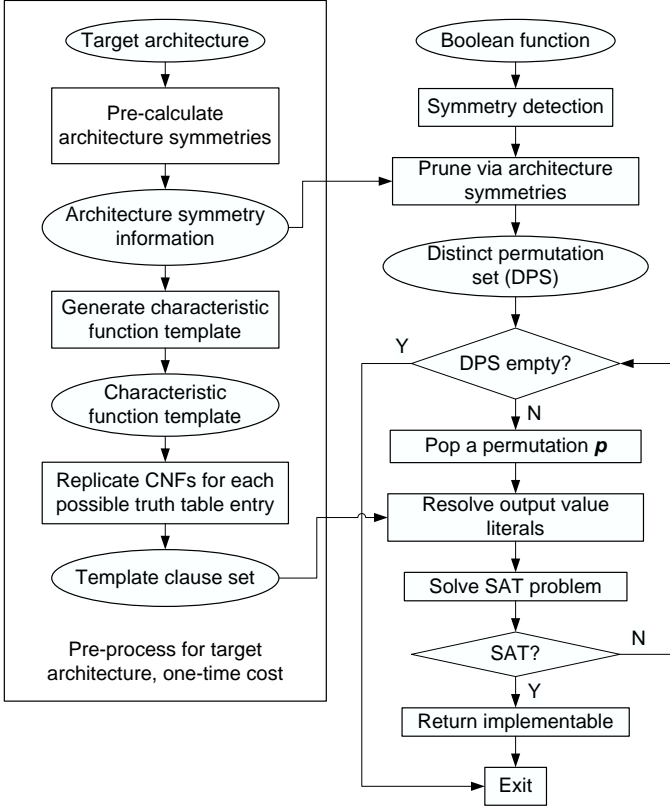


Fig. 9: Overall algorithm flow

$$\begin{aligned}
 G_{SAT} = & G[X/000, f/0, z/z_1] \cdot G[X/001, f/0, z/z_2] \cdot \\
 & G[X/010, f/1, z/z_3] \cdot G[X/011, f/0, z/z_4] \cdot \\
 & G[X/100, f/1, z/z_5] \cdot \\
 & G[X/110, f/1, z/z_7] \cdot G[X/111, f/1, z/z_8]
 \end{aligned} \tag{4}$$

Since we consider each unique permutation explicitly in our algorithm, we need to solve multiple SAT instances sequentially, one for each permutation. In fact this procedure is able to take advantage of the incremental SAT reasoning in miniSAT2.0 [24]. All of these SAT instances share the same characteristic function (1), while the difference between two SAT instances for two different permutations is the truth table (or implicant table). Therefore any two SAT instances will share a large portion of clauses, which we call *core clauses*. The distinct clauses related to the output values we call *soft clauses*. The sequential SAT reasoning procedure is performed incrementally by first introducing all the core clauses as assumptions in miniSAT. Then, for each unique permutation, we assert the soft clauses for that instance incrementally, and solve the resulting problem (with both the core and the soft clauses). If all clauses are added and the instance is satisfiable, we have found a matching. If the current formula (the core and the soft clauses) is unsatisfiable, we clear all the soft clauses (the assumptions still remain), and move to the next permutation. The structure of the formulas makes sophisticated clause removing unnecessary in the incremental construction.

B. CNF Clause Replication Time

As a practical consideration, a large proportion of the runtime of SAT-based Boolean matching techniques is spent replicating clauses. In our initial implementation, we observed that the percentage of time consumed by the replication phase was as high as 57% of total runtime for nine inputs, while actual SAT solution time peaked at only 4% of total runtime for the same number of inputs. Figure 10 shows the proportion of total runtime that replication consumes as compared to that of SAT solving time, with the trend increasing as the number of inputs grows.

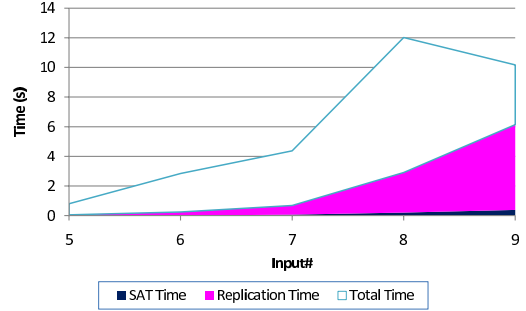


Fig. 10: Replication time as a proportion of total runtime

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

1) *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; it may be 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 a 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 are 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 [16].

We also applied the iterative clause testing technique to the implicant representation implementation; however, that 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 allows early unsatisfiability detection more often than with the implicant representation.

2) *Template Clause Set*: Noting the incremental capabilities of our particular SAT solver led to observation which precipitated an improvement which surpassed all previous techniques with regard to CNF replication time. We 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 $f(x_1, x_2, \dots, x_n)$. The truth table of f has 2^n entries, with possible inputs $000\dots 0$ through $111\dots 1$. Note that any permutation of f will have not only the same number of inputs, but in fact the exact same input values $000\dots 0$ through $111\dots 1$, though in a different order. What remains distinct between permutations is to which output values those same input values map. Figure 11 shows an example where f' is equivalent to f , but with x_1 and x_3 swapped.

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 are 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. Now 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, \dots, f_{2^n} unmodified, effectively leaving them as unbound free variables for the SAT solver. This is the

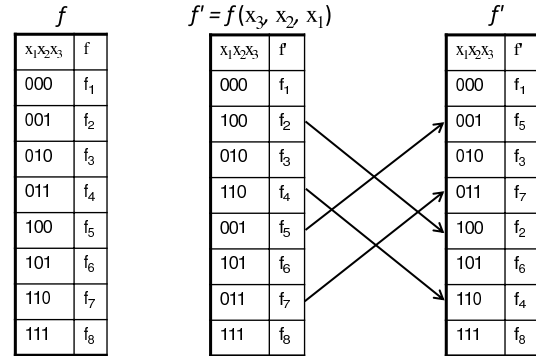


Fig. 11: Illustration of identical input values for function f and permutation f'

template clause set.

To test a permutation, we calculate the reordering of output values induced by the permutation of input variables. Applying this new ordering to the function's output values, we can then determine how the unbound variables f_1 through f_{2^n} of the template clause set should be constrained. 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 induced by the permutation and binding the output values to assumptions. Thus the CNF replication process, which was performed for every truth table entry, for each permutation, is now performed only once per permutation tested. Note that the incremental SAT learning information will be reset for each permutation but it still is recorded and used to speedup conflict finding when testing all input vectors under one permutation.

In practice, we find that the incremental SAT reasoning with template clause set is a more effective technique for runtime reduction, compared to the implicant clause testing based approach. The comparison between these two approaches will be shown in Section V-A.

V. EXPERIMENTS

We implement our algorithms in C++ and Perl, using miniSAT2.0 [24] as our SAT solver. The implicant table-based SAT encoding [16] has been implemented and integrated into our algorithm as shown in Figure 9. To show the effectiveness of our improvement to the SAT-based Boolean matching algorithm (shown in Figure 9), we extract over 10k fanout-free cones (FFCs) with 5-9 inputs from MCNC benchmarks based on the method presented in [4] as the Boolean functions. The target PLB architecture is shown in Figure 12. There exists both first order symmetries (e.g., swapping input variables x_1, x_2, x_3, x_4 of a LUT will not change the results) and the second order symmetries (e.g., swapping (x_1, x_2, x_3, x_4) as a group to (x_5, x_6, x_7, x_8) will not change the results). Both levels of symmetries will be used in our SAT-BM. All experiments are run on a 1.9GHz CPU Linux server with 2GB memory.

We first randomly select 30 9-input Boolean functions from the 9-input cut set, and calculate the number of unique

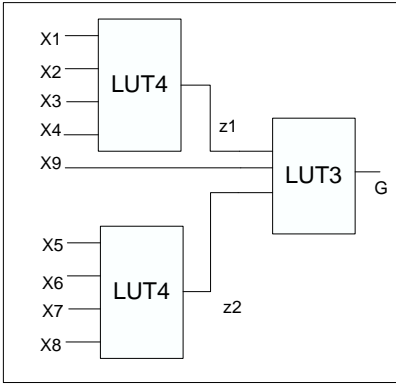


Fig. 12: A 9-input PLB

permutations considering symmetries as shown in Figure 13. Compared to the total number of unique permutations ($9! = 362,880$), we reduced computation by over two orders of magnitude by employing Boolean function symmetries, and by another two orders of magnitude by considering architectural symmetries.

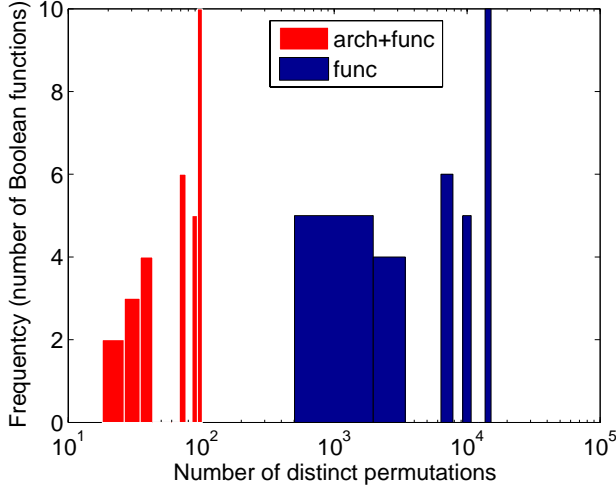


Fig. 13: Comparison of the number of distinct permutations when pruning function symmetries (“func”), and when pruning both function and architectural symmetries (“arch”)

Table I compares the original algorithm SAT-BM presented in [14] 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.2. 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 each sub-SAT instance for our SAT-IP-template algorithm remains virtually the same regardless 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 in the Boolean function and the PLB architecture. As shown in Table I row “unique perm#”, the number of unique per-

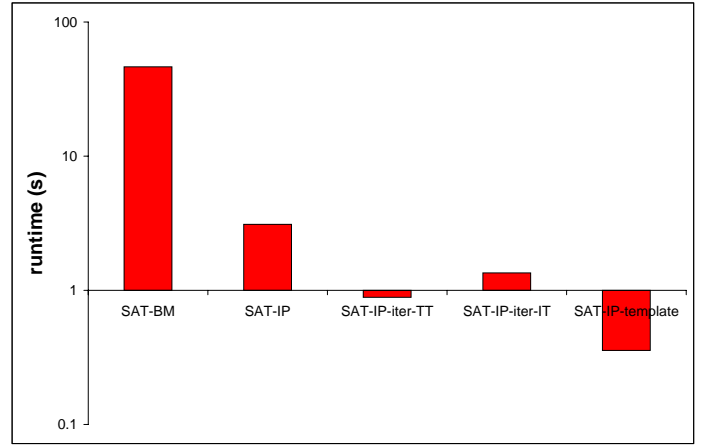


Fig. 14: Comparison of average runtime with different CNF replication speedup techniques

mutations grows slowly after pruning based on symmetries. Compared to SAT-BM, SAT-IP-template achieves 1364x and 226x speedup in terms of SAT reasoning runtime and total runtime, respectively, for 9-input logic functions. More significant speedup is expected if Boolean functions with wider inputs are considered³. Note that two recent improvements on the SAT-based Boolean matching problem, [15] and [16] obtained up to 13x speedup compared to [14]. The substantial speedup obtained by SAT-IP-template makes it possible to integrate the SAT-based Boolean matching algorithm within technology mapping and logic optimization during heterogeneous FPGA synthesis.

A. Comparison of speedup techniques

Tested on 3K 9-input Boolean functions, a comprehensive comparison of the average runtime with different speedup techniques is shown in Figure 14 between (a) Ling’s approach (SAT-BM), (b) our initial algorithm without optimization for CNF replications (SAT-IP-base), (c) our algorithm with iterative clause testing and the truth table based representation (SAT-IP-iter-TT), (d) our algorithm with iterative clause testing and the implicant table based representation [16] (SAT-IP-iter-IT), and (e) our algorithm with the template clause set improvement (SAT-IP-template). The average (geomean) runtime for these techniques under all testing cases is shown in Figure 14.

An interesting observation is that the improvements gained by our incorporation of the implicant representation as presented by [16] is 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

³We only compare the Boolean functions with up to 9 inputs because for larger Boolean functions the SAT-BM algorithm produces such a large SAT instance that miniSAT will lead to memory crash.

Test cases	func size	5	6	7	8	9
	problem#	1398	1981	2263	2172	2134
	variable#	867	1571	2979	5795	11,427
SAT-BM	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
	variable#/inst	1576	1576	1576	1576	1576
	clause#/inst	6144	6144	6144	6144	6144
SAT-IP-template	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	7.3x	14x	27x	116x	1364x
	Total speedup	5.5x	8.6x	13x	34x	226x

TABLE I: Comparison of SAT solving time between SAT-BM and our improved algorithm (SAT-IP-template)

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.

B. Comparison with Shatter/Saucy

In order to show the effectiveness of our symmetry-aware Boolean matching algorithm, we compare our best algorithm, i.e., SAT-IP with the template clause set, to a symmetry detection and CNF optimization tool, Shatter/Saucy [25], which takes advantage of the symmetries existing in CNFs of a SAT instance. The technique employed by this set of tools is to analyze a given SAT problem for symmetries with respect to literals. Then, the original SAT problem is augmented by adding symmetry-breaking clauses. This reduces the solution space but presumably makes finding a solution faster for the SAT solver. For instance, given the following Boolean formula in CNF, which has eight satisfiable assignments:

$$a + b + c$$

Shatter will generate the clauses

$$(-b + c)(-a + b)$$

which, when added to the original SAT problem, reduces the number of satisfiable solutions to one. Figure 15 shows the flow of our experiments with Shatter/Saucy tool chain.

The experimental results show that Shatter’s symmetry detection time was negligible. Figure 16 shows the speedup (times) with our proposed SAT-IP-template over the Shatter/Saucy-based flow (only SAT solving runtime is compared). Clearly 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 constructed problems. These results validate the need for specialized symmetry detection at the Boolean matching level.

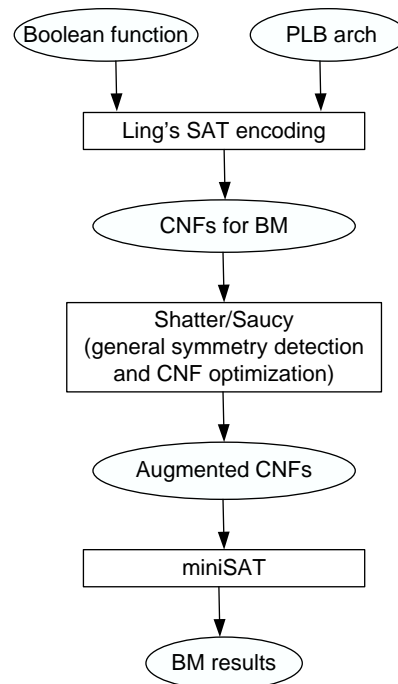


Fig. 15: The flow of the SAT-BM optimized by Shatter/Saucy

C. 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 5-12 inputs against the 12-input PLB shown in Figure 17. Figure 18 compares the runtime for SAT-BM with the 12-input PLB and the 9-input PLB shown in Figure 12. The average runtime for each input number is shown in the figure. Compared to the SAT-BM with the 9-input PLB, the SAT-BM with the 12-input PLB has a 10X 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 (1.3X increase) and permutations (>1000X 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

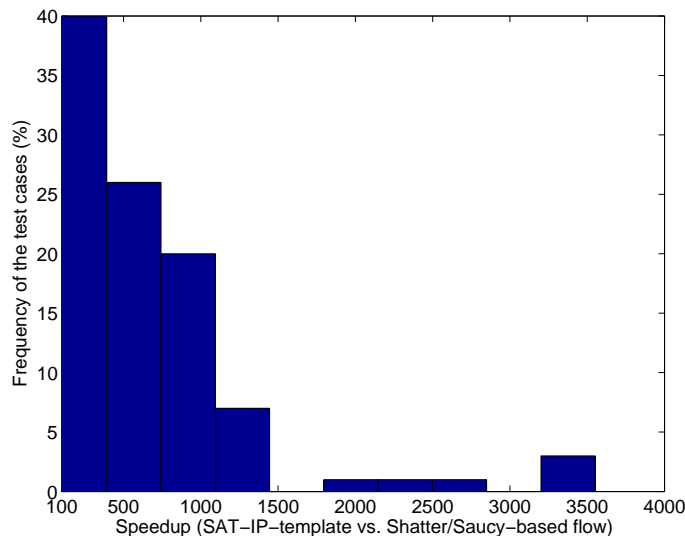


Fig. 16: Comparison of SAT solving time between Shatter/Saucy and our algorithm over several 9-input test cases

permutations after the pruning based on symmetries and the SAT time for each sub SAT instance is dominated by the target PLB architecture and does not change significantly from one input number to the other.

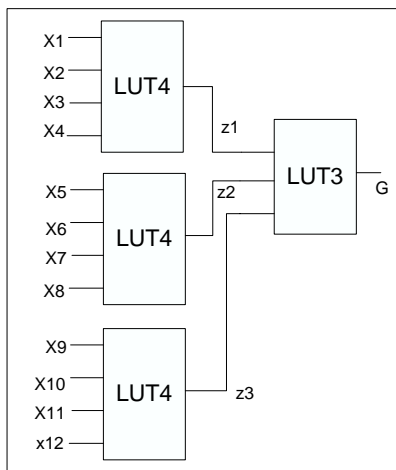


Fig. 17: A 12-input PLB

VI. CONCLUSION

Targeting orders of magnitude speedup over the existing algorithms for SAT-based Boolean matching [14], [15], [16], we have presented an algorithm to significantly improve the efficiency of SAT-based Boolean matching by exploring the symmetries exhibited in both the Boolean function and the target PLB architecture during CNF encoding. As shown in the experimental results, the SAT problem size and the SAT reasoning runtime are dramatically reduced. Our SAT-BM obtains up to two orders of magnitude speedup compared to the original algorithm [14]. In contrast, recent work [15], [16] obtained up to 13x speedup. Our work makes SAT-based Boolean matching more practical for synthesis and

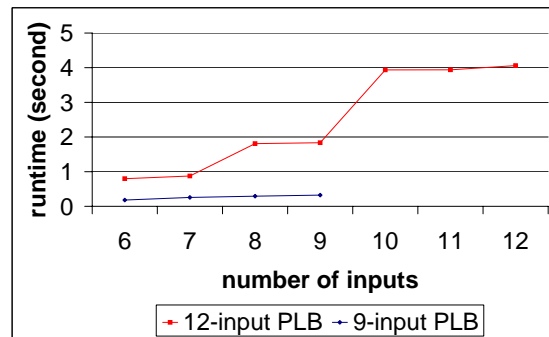


Fig. 18: Scalability study of SAT-IP-template

optimization of heterogeneous FPGAs. We envision that our SAT-BM is a favorable technique to be applied to FPGA architecture evaluation, resynthesis and technology mapping for high-end applications. Nevertheless, our SAT-BM is still slow compared to structural technology mapping, and it might not be practical for FPGA end users.

In the future, we will first apply Saucy [25] to truth tables rather than CNFs as shown in [26], and then compare with our proposed algorithm. Furthermore, we will integrate our SAT-BM into FPGA synthesis for further 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. Intl. Conf. Computer-Aided Design*, 2007.
- [2] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, 2006.
- [3] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1–12, January 1994.
- [4] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, 1999.
- [5] D. Chen and J. Cong, "Daomap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. Intl. Conf. Computer-Aided Design*, 2004.
- [6] A. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," vol. 13, no. 11, pp. 1319–1332, 1994.
- [7] G. D. Micheli, "Synchronous logic synthesis: algorithms for cycle-time minimization," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 63–73, 1991.
- [8] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting," in *Proc. Design Automation Conf.*, 2005.
- [9] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinatorial techniques," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, pp. 74–84, 1991.
- [10] R. Brayton and A. Mishchenko, "Sequential rewriting," in *International Workshop on Logic Synthesis*, 2007.
- [11] J. Cong and Y.-Y. Hwang, "Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2001.
- [12] L. Benini and G. D. Micheli, "A survey of Boolean matching techniques for library binding," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, no. 3, pp. 193–226, 1997.
- [13] A. Abdollahi and M. Pedram, "A new canonical form for fast boolean matching in logic synthesis and verification," in *Proc. Design Automation Conf.*, 2005.
- [14] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: a study of optimality," in *Proc. Design Automation Conf.*, 2005.

- [15] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient sat based boolean matching for FPGA technology mapping," in *Proc. Design Automation Conf.*
- [16] J. Cong and K. Minkovich, "Improved sat-based boolean matching using implicants for LUT-based FPGAs," in *Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, 2007.
- [17] "Xilinx product datasheets," in <http://www.xilinx.com/literature>.
- [18] D. Lewis and et al, "The stratix ii routing and logic architecture," in *Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, Feb 2005.
- [19] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetry in SATbased boolean matching for heterogeneous FPGA technology mapping," in *Proc. Intl. Conf. Computer-Aided Design*, 2007.
- [20] A. Biere, "Resolve and expand," in *SAT*, 2004.
- [21] A. Ayari and D. Basin, "Qubos: deciding quantified boolean logic using propositional satisfiability solvers," in *FMCAD*, 2002.
- [22] 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 *International Workshop on Logic Synthesis*, 2004.
- [23] 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 Automation Conf.*, 2006.
- [24] N. Een and N. Sorensson, <http://www.minisat.se/>.
- [25] F. Aloul, I. Markov, and K. Sakallah, "Efficient symmetry-breaking for boolean satisfiability," in *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 271–282, 2003.
- [26] K.-H. Chang, I. Markov, and V. Bertacco, "Postplacement rewiring by exhaustive search for functional symmetries," *ACM Trans. Design Automation of Electronics Systems*, 2007.