Exploiting Symmetry in SAT-Based Boolean Matching for Heterogeneous FPGA Technology Mapping

Yu Hu[1], Victor Shih[2], Rupak Majumdar[2] and Lei He[1]
1. Electrical Engineering Department
2. Computer Science Department
University of California, Los Angeles

ABSTRACT

The Boolean matching problem is a key procedure in field-programmable gate array (FPGA) technology mapping. SAT-based Boolean matching provides a flexible solution for exploring various FPGA architectures. However, the computational complexity of state-of-the-art SAT-based Boolean matching prohibits its application practically. In this paper we revisit the SAT-based Boolean matching (SAT-BM) problem for heterogeneous FPGAs and propose a very efficient algorithm by exploring function and architectural symmetries. While recent work obtained up to 13x speedup, our algorithm achieves up to 200x speedup by considering the symmetries, when compared to the original SAT-BM algorithm.

1. INTRODUCTION

FPGAs are programmable logic chips that can be configured to implement various digital circuits. FPGAs are quickly replacing custom ASICs in many areas due to their flexibility and fast turn-around times for product development. However, these benefits come at the heavy cost of area, speed, and power.

The programmable logic block (PLB) is the basic element of an FPGA design. Various programmable devices can be placed within a PLB; a lookup table (LUT) is one such programmable device. LUT-based FPGAs use PLBs populated with LUT components to implement various logic functions. A K-LUT device consists of K inputs, one output, and 2^K configuration bits that serve as truth table entries. With its 2^K configuration bits programmed accordingly, the K-LUT can implement any K-input function. For example, Figure 1 shows a simple LUT-2. By setting the configuration bits to 0, 0, 0, and 1 for L_0, L_1, L_2, and L_3, respectively, we can implement a 2-input AND gate with this LUT-2.

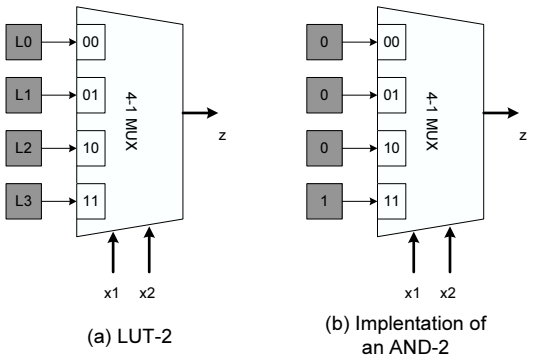

Figure 1: A 2-input lookup table (a) and its implementation for a 2-input AND gate (b)

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 circuit function can be given in terms of a synthesized multi-level netlist, input/output functional relationship, or other representation. Depending on the technology mapping approach, the resulting network will exhibit direct area, delay, and power costs. Most of the existing work for FPGA technology mapping [1, 2, 3] assumes that the only logic elements within a PLB are K-LUTs, also referred to as *homogeneous* FPGAs. Since a K-LUT can implement any K-input function, the technology mapping for homogeneous FPGAs is to find optimal K-bounded covers [3] in the subject graph, and the logic functionality of each K-bounded cover does not need to be considered.

Alternatively, modern FPGAs such as Xilinx Virtex IV [4] and Altera Stratix II [5] employ *heterogeneous* PLBs, which contain various logic devices such as logic gates and LUTs with different inputs. Figure 2 shows the PLB architecture for Altera Stratix II ALM [5]. This heterogeneity allows more flexibility in FPGA designs, which can result in reduced on-chip power dissipation, reduced area overhead, and improved performance. On the other hand, the extra flexibility of heterogeneous FPGAs increases the search space of technology mapping. As an example, 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 mapping.

Boolean matching [6, 7] is the most important sub-problem in technology mapping for heterogeneous FPGAs. 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. Most of the existing work for Boolean matching is based on function decomposition [6] or on canonicity and Boolean signatures [7]. These approaches are limited by the input size of the functions they can handle. Recently, a SAT-based approach [8] has been proposed to solve Boolean matching and was improved by [9] with a 3x speedup, and was further improved by [10] with up to 13x speedup. While SAT-based Boolean matching offers great flexibility in handling various FPGA architectures, it still suffers from long runtimes due to high computational complexity. For example, the Boolean matching procedure is called over 50,000 times for the MCNC circuit $i10$ with less than 3000 gates, with a typical runtime for completing one SAT-based Boolean matching [8] 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.

Inspired by a recent improvement on Boolean matching for ASIC

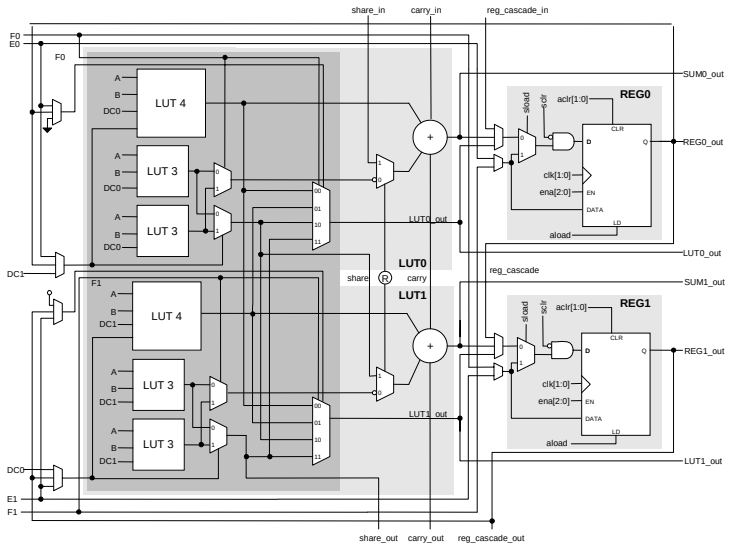

Figure 2: PLB Architecture for Stratix II (from Altera Inc.)

[11] and an enhancement on SAT reasoning [12] by exploring symmetries, we revist the SAT-based Boolean matching problem for heterogeneous FGPAs. Our proposed algorithm targets orders of magnitude speedup over the existing algorithms [8, 9, 10]. The major contribution of this paper is to significantly improve the efficiency of the SAT-based Boolean matching by exploring the symmetries exhibited in both the Boolean function and the target PLB architecture. Considering function and architecture symmetries explicitly during CNF encoding, the SAT problem size and the SAT reasoning runtime are dramatically reduced. The experimental results show that the proposed algorithm obtains up to 200x speedup by considering symmetries compared to the original algorithm [8], while the recent papers [9, 10] obtained up to 13x speedup.

The rest of this paper is organized as follows: Section 2 formalizes the concepts involved in Boolean matching and reviews the SAT-based encoding [8]. Section 3 presents our heuristics for improving the efficiency of the SAT-based Boolean matching approach using symmetries. Section 4 details our experimental results, and section 5 concludes the paper.

2. BACKGROUNDS 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 $\{p_1, \cdots, p_m\}$. 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.

2.1 From PLBs to CNF

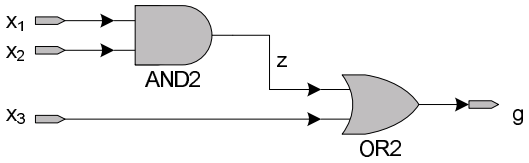

Figure 3: Example encoding for non-programmable devices

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

$$(x_1 \cdot x_2 \leftrightarrow z) \tag{1}$$

which in CNF becomes

$$(x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z) \tag{2}$$

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 3 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:

$$f_{AND2} = (x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z) \tag{3}$$

$$f_{OR2} = (\neg x_3 + g) \cdot (\neg z + g) \cdot (x_3 + z + \neg g) \tag{4}$$

$$f_{all} = f_{AND2} \cdot f_{OR2} \tag{5}$$

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, $L_1, \cdots, L_{2^K}$, 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 shown in Figure 1(a) can be encoded as

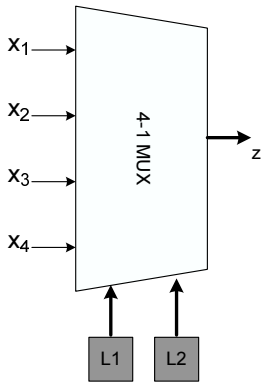

Figure 4: A 4-input programmable MUX

follows:

$$(x_1 + x_2 + \neg L_1 + z_1) \cdot (x_1 + x_2 + L_1 + \neg z_1) \cdot$$
$$(x_1 + \neg x_2 + \neg L_2 + z_1) \cdot (x_1 + \neg x_2 + L_2 + \neg z_1) \cdot$$
$$(\neg x_1 + x_2 + \neg L_3 + z_1) \cdot (\neg x_1 + x_2 + L_3 + \neg z_1) \cdot$$
$$(\neg x_1 + \neg x_2 + \neg L_4 + z_1) \cdot (\neg x_1 + \neg x_2 + L_4 + \neg z_1)$$

For a K-input programmable MUX, we have $\lceil \log K \rceil$ configuration bits, so we introduce $\lceil \log K \rceil$ additional variables. Figure 4 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 shown as follows:

$$z \;=\; \mathsf{ite}(L_2, \mathsf{ite}(L_1, x_4, x_3), \mathsf{ite}(L_1, x_2, x_1))$$

where $\mathsf{ite}(x, y, z)$ is the Boolean function $x \wedge y \vee \neg x \wedge z$. That is, when x is true, y is selected; when x is false, z is selected. When encoded in CNF, this becomes:

$$(L_1 + L_2 + \neg x_1 + z) \cdot (L_1 + L_2 + x_1 + \neg z) \cdot$$
$$(L_1 + \neg L_2 + \neg x_2 + z) \cdot (L_1 + \neg L_2 + x_2 + \neg z) \cdot$$
$$(\neg L_1 + L_2 + \neg x_3 + z) \cdot (\neg L_1 + L_2 + x_3 + \neg z) \cdot$$
$$(\neg L_1 + \neg L_2 + \neg x_4 + z) \cdot (\neg L_1 + \neg L_2 + x_4 + \neg z) \tag{6}$$

2.2 From Boolean Matching to SAT

Let $G(x_1, \cdots, x_n, L_1, \cdots, L_m, z_1, \cdots, z_l, o)$ 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, and variables $z_1, \ldots, z_l$ represent the intermediate circuit signals, and o 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 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 $L_1, \ldots, L_m$ such that for all input variables $x_1, \ldots, x_n$ there are valuations of the intermediate signals such that the output o of the PLB is equivalent to the output of the function f. Formally, the Boolean matching problem is formulated as the following quantified Boolean satisfiability (QSAT) problem:

$$\exists L_1, \cdots, L_m \forall x_1, \cdots, x_n \exists z_1, \cdots, z_l, o, f.$$
$$G(x_1, \cdots, x_n, L_1, \cdots, L_m, z_1, \cdots, z_l, o)$$
$$F(x_1, \cdots, x_n, f) \wedge (o \equiv f) \tag{7}$$

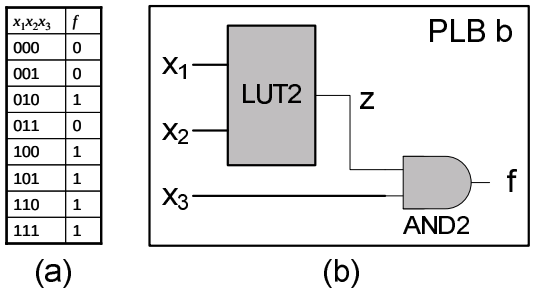

Figure 5: (a) Truth table for a function f, (b) An example PLB

As in [8], the universal quantifiers in (7) can be removed by enumerating the truth table of the function $F(x_1, \cdots, x_n)$. Therefore, (7) can be solved with SAT, where a satisfying assignment implies that the function can be realized by the configuration.

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, N is the number of inputs in the largest LUT 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.

2.3 Example

Consider the example PLB shown in Figure 5(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 5(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 instance with the following steps:

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

$$\begin{aligned} G_{LUT} &= (x_1 + x_2 + \neg L_1 + z)(x_1 + x_2 + L_1 + \neg z) \\ &\quad (x_1 + \neg x_2 + \neg L_2 + z)(x_1 + \neg x_2 + L_2 + \neg z) \\ &\quad (\neg x_1 + x_2 + \neg L_3 + z)(\neg x_1 + x_2 + L_3 + \neg z) \\ &\quad (\neg x_1 + \neg x_2 + \neg L_4 + z)(\neg x_1 + \neg x_2 + L_4 + \neg z) \\ G_{AND} &= (z + \neg o) \cdot (x_3 + \neg o) \cdot (\neg z + \neg x_3 + o) \end{aligned} \tag{8}$$

2. The characteristic function of the PLB is then obtained as:

$$G = G_{LUT} \cdot G_{AND} \tag{9}$$

Notice that the output variable is called o.

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:

$$\begin{aligned} G_f &= (\neg x_1 + \neg x_2 + \neg x_3 + f) \cdot (\neg x_1 + \neg x_2 + x_3 + f) \cdot \\ &\quad (\neg x_1 + x_2 + \neg x_3 + f) \cdot (\neg x_1 + x_2 + x_3 + f) \cdot \\ &\quad (x_1 + \neg x_2 + \neg x_3 + \neg f) \cdot (x_1 + \neg x_2 + x_3 + f) \cdot \\ &\quad (x_1 + x_2 + \neg x_3 + \neg f) \cdot (x_1 + x_2 + x_3 + \neg f) \end{aligned} \tag{10}$$

The QSAT formulation can then be expressed as follows:

$$\exists L_1 \exists L_2 \exists L_3 \exists L_4 \forall x_1 \forall x_2 \forall x_3 \exists z, o, f.$$
$$(G \cdot G_f \cdot (o \vee \neg f) \cdot (\neg o \vee f)) \tag{11}$$

$x_1x_2x_3$	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)

Figure 6: (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 (9) to remove the universal quantifiers on the input variables in X. This formulates G_{SAT}:

$$\begin{aligned} G_{SAT} \quad = \quad & G[X/000, o/0, z/z_1] \cdot G[X/001, o/0, z/z_2] \cdot \\ & G[X/010, o/1, z/z_3] \cdot G[X/011, o/0, z/z_4] \cdot \\ & G[X/100, o/1, z/z_5] \cdot G[X/101, o/1, z/z_6] \cdot \\ & G[X/110, o/1, z/z_7] \cdot G[X/111, o/1, z/z_8] \end{aligned} \tag{12}$$

A satisfiable assignment to G_{SAT} implies f can be implemented by the PLB. Based on the SAT solver, this SAT problem is unsatisfiable, meaning that the Boolean function shown in Figure 5(a) cannot be implemented by the PLB shown in Figure 5(b).

2.4 Input Permutations

An important issue in Boolean matching is *input permutation*, which allows different mappings from pins in a PLB to variables of a Boolean function. Figure 6 shows two Boolean functions which are equivalent under input permutation, i.e., function f_a can be transferred to f_b by the input permutation $\tau = (3, 2, 1)$. However, f_a cannot be implemented by the PLB shown in Figure 5(b), while f_b can.

In practice, input permutation is allowed in FPGA designs and must be considered during Boolean matching to maximize the number of implementable instances. However, the number of permutations for a K-input Boolean function is $K!$ which grows extremely quickly. In order to consider input permutations in the SAT formulation, [8] proposed to add programmable MUXs before each primary input of the target PLB (see Figure 7). 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 7) and the intermediate pins (e.g., z_1, z_2, z_3), and $O(n^2)$ additional clauses are needed, as well. Thus, considering input permutations adds $O(n^3)$ clauses and $n \cdot (\lceil \log n \rceil + 1)$ variables to the original formulation. In practice, the size of a LUT is usually less than six, so adding these MUXs can double the size of the SAT/QSAT problem if n is a large (i.e., greater than 6).

3. SPEEDUP BY SYMMETRIES

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

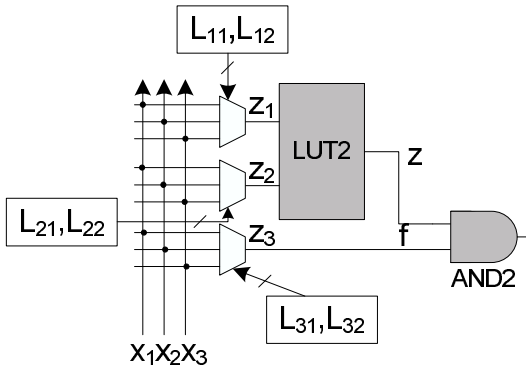

Figure 7: Considering input permutations with additional MUXs

3.1 Symmetries in Boolean Functions

Variable x_i and x_j of Boolean function $f(x_1, \cdots, x_n)$ are *symmetric* if the truth table of f remains the same when x_i and x_j are swapped, i.e., if $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. We can consider only *distinct permutations* by observing the variable symmetries exhibited in a Boolean function, making the programmable MUXs added in subsection 2.4 unnecessary.

Given an n-input Boolean function $f(x_1, \cdots, x_n)$, 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 four symmetries (0, 1, 6, 8), (3, 4, 5), (2), and (7) as shown in Figure 8. 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 $\tau_1 = \tau_2$ and only one of them needs to be tested in 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.

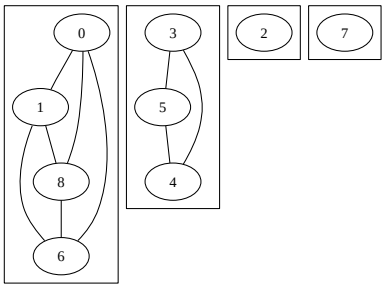

Figure 8: Symmetries in a 9-input Boolean function

Note that the time required to identify symmetries of an n-input function using the above algorithm is $O(n^2 \cdot 2^n)$. However, this computational cost is negligible in practice compared to the Boolean matching time, 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 [13].

3.2 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 $x_1, \cdots, x_k$ and inputs $y_1, \cdots, y_k$ for two k-input LUTs L_x and L_y, respectively, are symmetric under permutation $\pi(y_{i_1}, \cdots, y_{i_k}, x_{j_1}, \cdots, x_{j_k})$ if the outputs x and y of these two LUTs are symmetric.*

For example, in the PLB shown in Figure 9, 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 called 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.

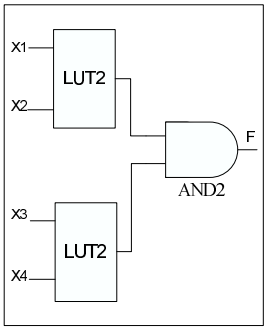

Figure 9: A second order symmetric PLB

To detect symmetries exhibited in the PLB architecture, we extend the structural analysis algorithm presented in [14] to consider programmable logic devices in the circuit. As shown in Figure 10(A), given a target PLB architecture, we first rewrite all the non-programmable logic into an AND-inverter graph (AIG), leaving the programmable logic (i.e., LUTs) unchanged. For AIGs, an implication supergate rooted at node n is a multi-input AND gate created by expanding the AND gate rooted at node n until either a primary input, or a complemented edge, or a programmable device is reached. Figure 10(B) shows the result of generating implication supergates for the AIG in 10(A).

Architectural symmetries can be detected by propagating the symmetry information of the implication supergates in topological order from the primary inputs to the primary outputs. We can examine higher order symmetries in the target PLB architecture by applying our treatment of second order symmetries to higher order logic levels. In the example shown in Figure 10, since the fanins of the 3-input AND gate f are symmetric, the inputs of the two LUTs are symmetric under permutation $\pi(y, d, a, x), \pi(y, d, x, a)$. We conclude that a and d are symmetric since $x \equiv y = b \cdot c$.

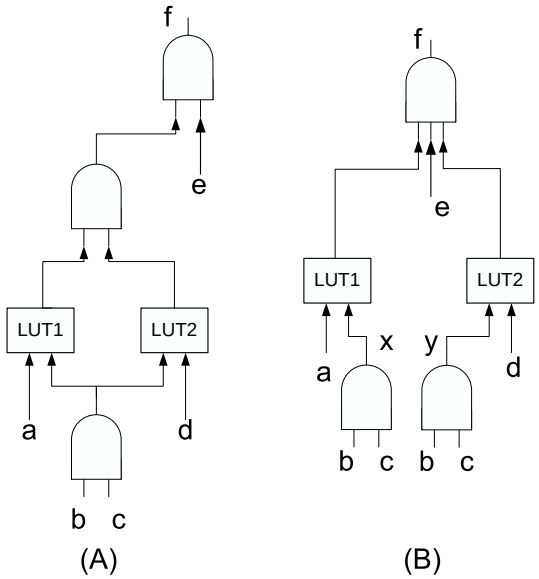

Figure 10: Illustration of architectural symmetry detection

Architectural symmetry detection can be done in the pre-process before re-synthesis. Since PLB sizes are typically small, runtime cost is not an issue.

3.3 Overall Algorithm

Figure 11 shows the flow of our overall algorithm. We first pre-process the architecture of the target PLB by extracting its architectural symmetry information (using the algorithm in Section 3.2) and generate a template of the characteristic function for the PLB. For each Boolean function to be tested, we first detect the function symmetries (using the algorithm in Section 3.1) 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. Given each permutation p, a SAT problem is generated by replicating the characteristic function based on the implicant table of p (the SAT encoding by implicant table was proposed in [10] and will be summarized in Appendix). If any permutation gives rise to a satisfiable solution, then the Boolean function can be implemented by the target PLB. If instead none of the SAT instances are satisfiable, then the function cannot be implemented by the target PLB.

3.4 Implementation Issues

Our algorithm can be extended to handle the output don't cares of a Boolean function. Since the point of including sub-expression $G(X/X_{value}, o/o_{value}, z/z_{value})$ in the replication form (12) is to ensure that the configuration of PLB which is programmed in correspondence with the satisfying assignment of (12) will produce o_{value} when X_{value} is on its inputs, the replication corresponding to the don't cares term DC can be removed from (12) 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 $G(X/101, o/DC, z/z_6)$ can

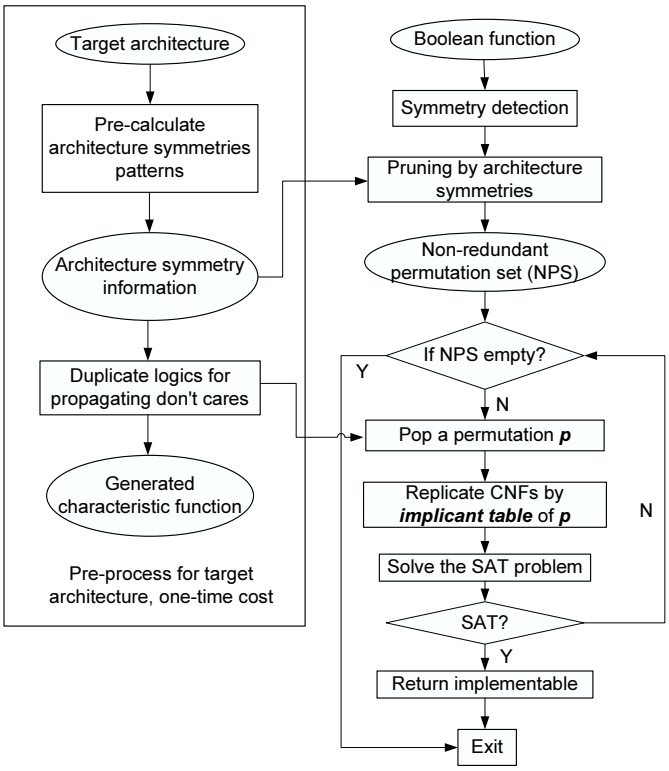

Figure 11: The flow of the overall algorithm

be removed from the replicated SAT encodes. And the replicated characteristic function (12) can be re-expressed as the following:

$$\begin{aligned} G_{SAT} \quad = \quad & G[X/000, o/0, z/z_1] \cdot G[X/001, o/0, z/z_2] \cdot \\ & G[X/010, o/1, z/z_3] \cdot G[X/011, o/0, z/z_4] \cdot \\ & G[X/100, o/1, z/z_5] \cdot \\ & G[X/110, o/1, z/z_7] \cdot G[X/111, o/1, z/z_8] \end{aligned} \tag{13}$$

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 [15]. All of these SAT instances share the same characteristic function (7), while the difference between two SAT instances for two different permutations is the truth table (or implicant table). Therefore 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*. In this sense the sequential SAT reasoning procedure can be performed incrementally as follows. We divide the clauses of a SAT instance into core clauses and soft clauses. The core clauses are processed first, and the resulting state Σ of the SAT solver is recorded. For each SAT instance, we incrementally insert new clauses from the soft clauses set. After solving the aggregated problem, we remove the soft clauses, reestablish state Σ and proceed with the next SAT instance.

4. EXPERIMENTS

We implement our algorithms in C++ and Perl. The SAT problems are solved by miniSAT2.0 [15]. The implicant table-based SAT encoding [10] (see Appendix) has been implemented and integrated into our algorithm as shown in Figure 11. To show the effectiveness of our improvement to the SAT-based Boolean matching algorithm (shown in Figure 11), we extract over 10k fanout-free cones (FFCs) with 5-9 inputs from MCNC benchmarks based on the method presented in [3] as the Boolean functions. The target PLB architecture is similar to the PLB in Figure 9 except that the two input LUTs have four inputs and the output logic is a 2-input LUT instead of a 2-input AND gate. All experimental data are collected in a Linux Server with a 1.9GHz CPU and 2GB memory.

We first randomly select 30 9-input Boolean functions from the 9-input cut set, and calculate the number of unique permutations considering symmetries as shown in Figure 12. 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.

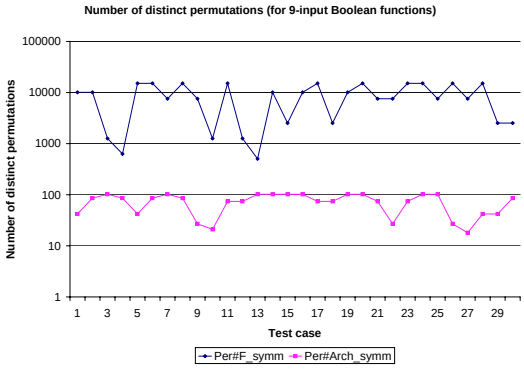

Figure 12: Number of permutations using Boolean function symmetries (F_symm) and using a combination of Boolean function symmetries and architecture symmetries (Arch_symm)

Table 1 compares the original algorithm SAT-BM presented in [8] and our improved algorithm (SAT-IP). 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 algorithm remains virtually the same regardless of the number of inputs in the Boolean function, and the number of unique permutations grows slowly. Compared to SAT-BM, SAT-IP achieves up to 400x overall speedup, where 200x speedup of SAT-IP is due to the consideration of symmetries and 2x more speedup is due to the integration of implicant table-based SAT-encoding (shown in Figure 13). More significant speedup is expected if Boolean functions with wider inputs are considered. Note that two recent improvements on the SAT-based Boolean matching problem, [9] and [10] obtained up to 13x speedup compared to [8]. The substantial speedup obtained by SAT-IP makes it possible to integrate the SAT-based Boolean matching algorithm within technology mapping and logic optimization during heterogeneous FGPA synthesis.

To break down the effectiveness of each components in SAT-IP, Figure 13 compares the runtimes of the SAT-BM algorithm (*SAT-BM*), the SAT-IP algorithm with truth table replication but considering only Boolean function symmetries (*F_symm_SAT-IP*), the SAT-IP algorithm with truth table replication and considering both Boolean function symmetries and architectural symmetries (*Arch_symm_SAT-IP*), and the final SAT-IP algorithm with im-

Testcases	func size	5.00	6.00	7.00	8.00	9.00
	problem#	1398	1981	2263	2172	2134
	variable#	867	1571	2979	5795	11427
SAT-BM	clause#	6945	13889	27777	55553	111105
	runtime (s)	0.47	2.22	2.39	27.53	173.59
	variable#/inst	367	442	358	405	454
	clause#/inst	1509	1850	1466	1683	1906
SAT-IP	unique perm#	4.30	17.73	30.90	26.6	161.47
	runtime (s)	0.01	0.05	0.07	0.07	0.43
	speedup	45x	48x	32x	409x	403x

Table 1: Comparison between SAT-BM and our improved algorithm (SAT-IP)

plicant table replication and considering all symmetries (*SAT-IP*). This demonstrates that Boolean function symmetry itself cannot bring significant speedup, but combined with architectural symmetry optimizations, two orders of magnitude speedup can be attained compared to the original SAT formulation. With the integration of implicant table-based replication, an additional 2x speedup can be achieved. This is different from the 3-13x speedup reported in [10] and is due to the overlap in techniques for runtime reduction.

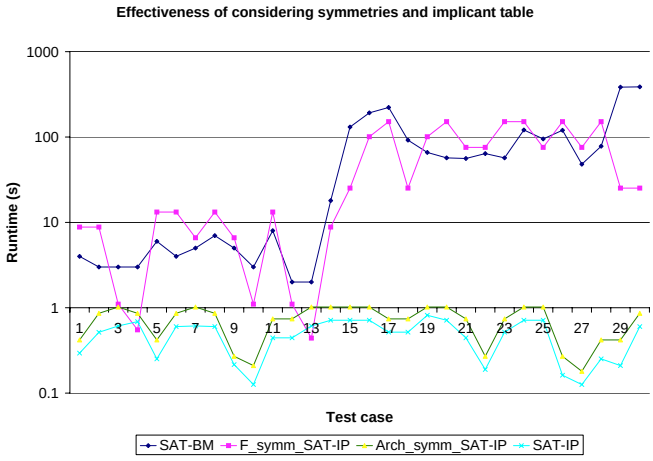

Figure 13: Runtime (seconds) of matching 30 9-input functions to the target PLB under symmetries and implicant table

5. CONCLUSION

Targeting orders of magnitude speedup over the existing algorithms for SAT-based Boolean matching [8, 9, 10], 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. Considering function and architecture symmetries explicitly during CNF encoding, the SAT problem size and the SAT reasoning runtime are dramatically reduced. The experimental results show that the proposed algorithm obtains up to 200x speedup by considering symmetries compared to the original algorithm [8], while recent work [9, 10] obtained up to 13x speedup. This improvement of SAT-based Boolean matching makes it practical for application during synthesis and optimization for heterogeneous FPGAs. Our future plans entail integrating our algorithm into technology mapping for FPGA architecture exploration.

6. REFERENCES

[1] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs," in *IEEE Trans. Computer-aided Design 13 (1), January 1994, 1–13.*, 1994.

[2] "Abc: A system for sequential synthesis and verification," in *http://www.eecs.berkeley.edu/ alumni/abc/*.

[3] 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.

[4] "Xilinx product datasheets," in *http://www.xilinx.com/literature*.

[5] D. Lewis and et al, "The stratix ii routing and logic architecture," in *Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, Feb 2005.

[6] J. Cong and Y.-Y. Hwang, "Boolean matching for lut-based logic blocks with applications to architecture evaluation and technology mapping," 2001.

[7] 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.

[8] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: a study of optimality," in *Proc. Design Automation Conf.*, 2005.

[9] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient sat based boolean matching for fpga technology mapping," in *Proc. Design Automation Conf.*

[10] 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.

[11] Z. Wei, D. Chai, A. Kuehlmann, and A. R. Newton, "Fast boolean matching with dont cares," in *Proc. Intl. Symp. on Quality Electronic Design*, 2006.

[12] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "Solving difficult instances of boolean satisfiability in the presence of symmetry," vol. 22, no. 9, pp. 1117–1137, 2003.

[13] 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.

[14] 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.

[15] N. Een and N. Sorensso, *http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html*.

[16] E. M. Sentovich et. al., "SIS: A system for sequential circuit systhesis," in *Department of Electrical Engineering and Computer Science, Berkeley, CA 94720*, 1992.

Appendix: Implicant Based SAT encoding

Instead of representing a Boolean function by its truth table, we can use its implicant table, which can be calculated efficiently in SIS [16]. Considering the truth table shown in Figure 15, where "-" represents a *don't care* input, it is clear that the number of rows in the implicant table is less than those in the truth table. In fact for a typical real circuit, the size of the implicant table for its Boolean function is usually linear to the number of inputs of the function.

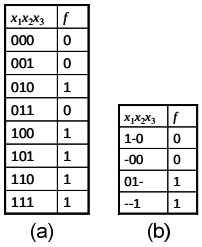

Figure 15: (a) Truth table, (b) Implicant table

[10] applied the above idea of considering the don't care terms in the implicant table for a LUT explicitly. The idea can be extended

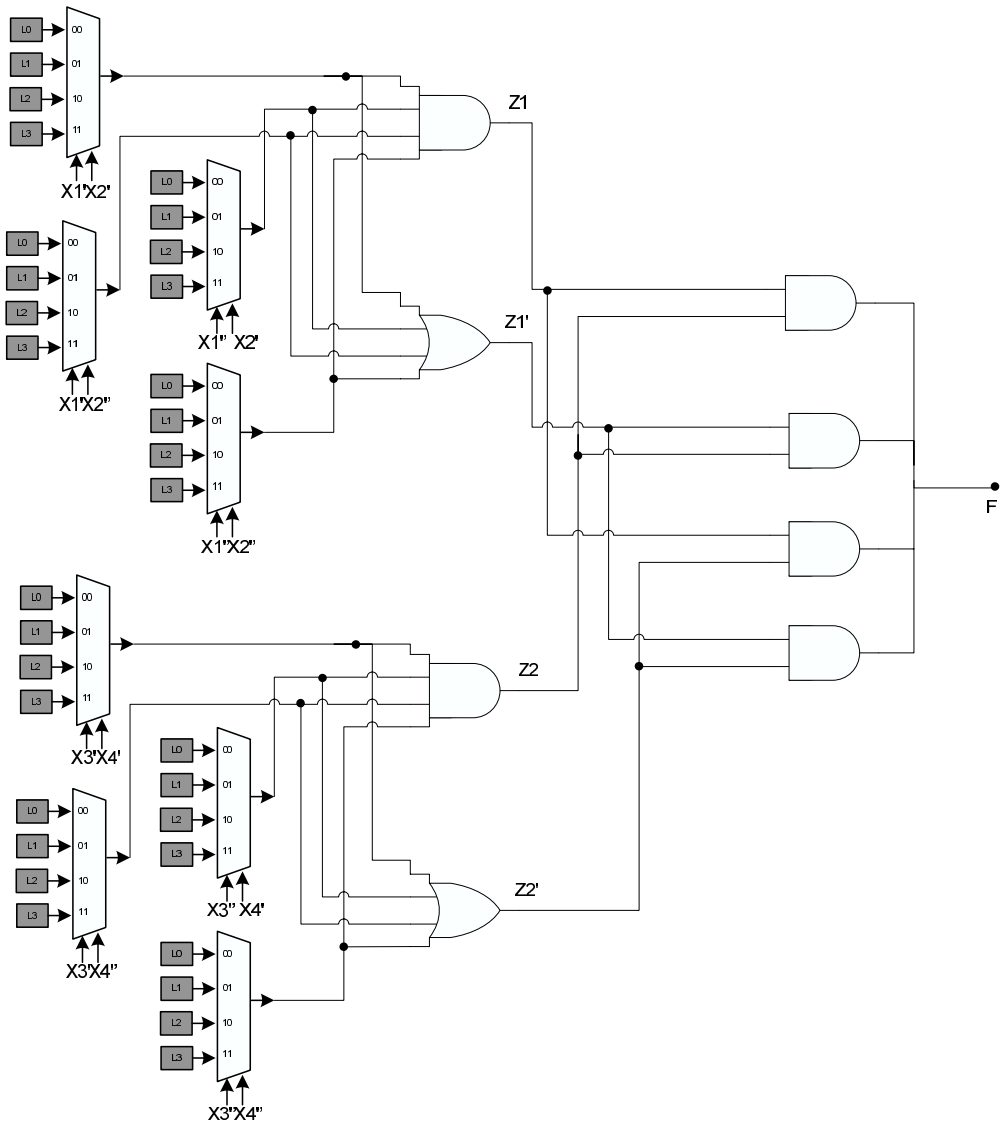

Figure 14: Duplication for the PLB shown in Figure 9

to handle general logic elements (e.g., an AND gate), not just LUTs. A logic element with k inputs must be duplicated 2^k times. The input variables are set by (14) and (15) below,

$$x_i' = \begin{cases} 0, & \text{if } x_i = \text{“-”} \\ x_i, & \text{otherwise} \end{cases} \tag{14}$$

$$x_i'' = \begin{cases} 1, & \text{if } x_i = \text{“-”} \\ x_i, & \text{otherwise} \end{cases} \tag{15}$$

During the replication of the characteristic function (7), for any input x_i of a logic element, inputs x_i' and x_i'' are generated to facilitate propagating don't care terms. If x_i is set to 0 (or 1), then x_i' and x_i'' are likewise both set to 0 (or 1). But if x_i is a don't care ("-"), then x_i' is set to 0 while x_i'' is set to 1.

The duplication of the logic element enables the propagation of don't care terms in the implicant table. Figure 14 shows an example of this duplication for the PLB from Figure 9. The first LUT (with inputs x_1 and x_2) in Figure 9 is duplicated as four LUT-2s whose outputs are hooked to a 4-input AND gate (with output z_1) and a 4-input OR gate (with output z_1'), respectively. Similarly the second LUT (with inputs x_3 and x_4) in Figure 9 is duplicated as another four LUT-2s whose outputs are hooked to a 4-input AND gate (with output z_2) and a 4-input OR gate (with output z_2'), respectively. The 2-input AND gate in Figure 9 is duplicated to four 2-input AND gates. Since the AND gate is the primary output of PLB_B, we hook up all outputs of its duplicated counterpart in Figure 14, which enforces that all duplicated outputs produce the same logic level.

Intuitively, the above logic duplication procedure considers input don't care terms *explicitly* by enumerating all possibilities of the don't care input combinations. The following theorem has been proven in [10].

THEOREM 1. *The decision of mapping a Boolean function (represented by its implicant table) to this duplicated PLB is equivalent to mapping the function to the original PLB.*