Efficient SAT-Based Boolean Matching for Heterogeneous FPGA Technology Mapping

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

ABSTRACT

The Boolean matching problem is the 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 integration into real technology mapping applications. In this project we present a thorough study of the SAT-based Boolean matching problem for heterogeneous FPGAs, conducting numerous comparisons among different approaches and heuristics. We propose several novel ideas to improve the efficiency of SAT-based Boolean matching, such as an implicant table-based representation and symmetries exploration. Overall, the experimental results show that our speedup techniques reduce runtime by approximately two orders of magnitude compared to the original SAT-based formulation.

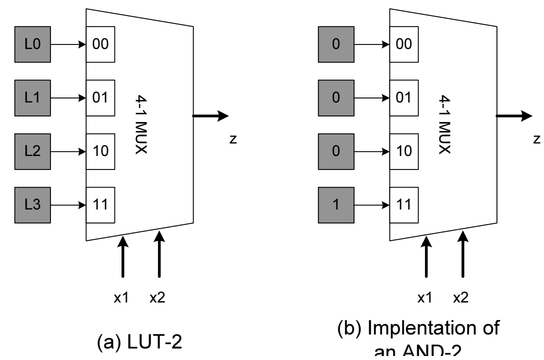


Figure 1: A 2-input lookup table (a) and its implementation for a 2-input AND gate (b)

1. INTRODUCTION

FPGAs are programmable logic chips that can be configured to implement various digital circuits. FPGAs are fast 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 the 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, 1" for L_0, L_1, L_2, and L_3, respectively, we can implement a 2-input AND gate with this LUT-2.

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* FPGA. 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.

Modern FPGAs such as Xilinx Virtex IV [4] and Altera Stratix II [5], have employed *heterogeneous* PLBs however, 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 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] for a three-times speedup. SAT-based Boolean matching offers great flexibility in handling various FPGA architec-

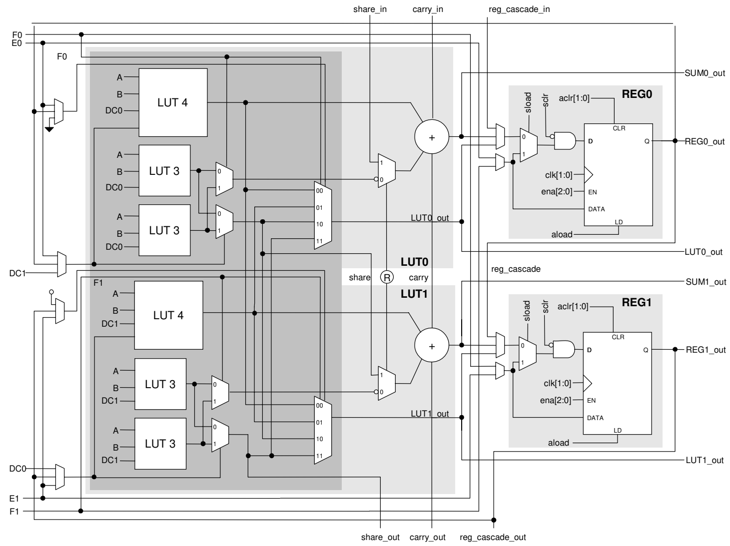


Figure 2: PLB Architecture for Stratix II (from Altera Inc.)

tures, but 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 typical runtime for completing one SAT-based Boolean matching [8] at more than 20 seconds. It appears the runtime overhead for heterogeneous FPGA technology mapping is prohibitively high due to the inefficiency of Boolean matching. In this project, *we revisit SAT-based Boolean matching and propose several ideas to improve its efficiency. Overall, our experimental results show that our speedup techniques reduce runtime by about two orders of magnitude compared to the original SAT-based formulation.*

The rest of this paper is organized as follows: Section 2 formalizes the concepts involved in Boolean matching, section **??** reviews SAT-based encoding [8], section 3 presents our studies for improving the efficiency of the SAT-based Boolean matching approach, and section 5 details our conclusions.

2. THE BOOLEAN MATCHING PROBLEM

A *configuration* $H(P)$ consists of a network of interconnected programmable logic blocks 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 configuration $H(P)$. We consider two kinds of programmable logic blocks 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 control bits. A K-MUX consists of K inputs, one MUX output, and $\lceil \log K \rceil$ control bits. With these denotations, the problem of Boolean matching is defined as follows: The *Boolean Matching* problem takes as input a configuration $H(P)$, and a boolean function $f(X)$ over the variables X such that $|X| \leq |P|$, and asks if the configuration $H(P)$ 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. In Figure 3 (i) and (ii) for example, it is not obvious what 7-input functions and 11-input functions, respectively, can be implemented in each configuration.

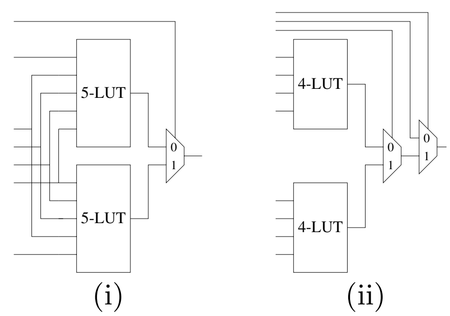


Figure 3: Simplifed versions of industrial FPGA PLBs

2.1 From Circuits to CNF

For non-programmable devices (e.g., combinational gates) in a configuration, we can describe the logic of the device as a Boolean formula in cojunctive 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 form 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 an intermediate variable for the output of each device and encode the relationship between the inputs and outputs of each device as CNF formulas in terms of the intermediate variables. Figure 4 shows an example encoding of a non-programmable device network, where an AND-2 gate and an OR-2 gate compose a 3-input logic function

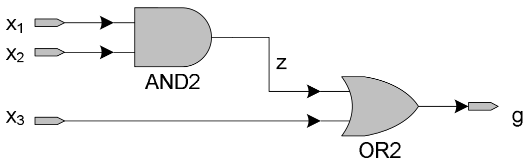


Figure 4: Example encoding for non-programmable devices

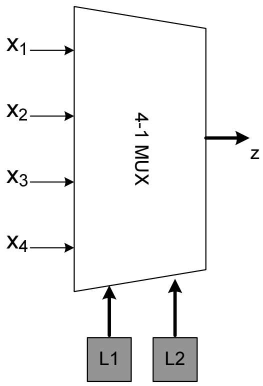


Figure 5: A 2-input programmable MUX

$g(x_1, x_2, x_3)$. The corresponding CNF, f_{all}, shown as follows:

$$f_{AND2} = (x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (\neg x_1 + \neg x_2 + z) \quad (3)$$
$$f_{OR2} = (\neg x_3 + g) \cdot (\neg z + g) \cdot (x_3 + z + \neg g) \quad (4)$$
$$f_{all} = f_{AND2} \cdot f_{OR2} \quad (5)$$

A similar encoding can be performed for LUTs and MUXes. For a K-input LUT, we introduce 2^K additional variables, $L_1, \cdots, L_{2^K}$, to represent every possible setting of the control bits. For example, the 2-input LUT with inputs x_1, x_2 and output $z1$ shown in Figure 1 (a) can be encoded as 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 $\lceil \log K \rceil$ control bits, so we introduce $\lceil \log K \rceil$ additional variables. Figure 5 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 control bits. The derivation of the CNF encoding for this 4-input programmable MUX is shown as follows:

$$z = L_2 \; ?[L_1 \; ? \; x_4 : x_3] : [L_1 \; ? \; x_2 : x_1]$$

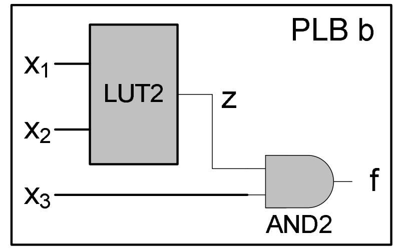
which encoded as a CNF formula becomes

$$(L_1 + L_2 + \neg x_1 + z) \cdot (L_1 + L_2 + x_1 + \neg z) \cdot$$
$$(L_1 + \neg L_2 + \neg x_2 + z) \cdot (L_1 + \neg L_2 + x_2 + \neg z) \cdot$$
$$(\neg L_1 + L_2 + \neg x_3 + z) \cdot (\neg L_1 + L_2 + x_3 + \neg z) \cdot$$
$$(\neg L_1 + \neg L_2 + \neg x_4 + z) \cdot (\neg L_1 + \neg L_2 + x_4 + \neg z)$$

(6)

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

(a)



(b)

Figure 6: (a) Truth table for a function f, (b) An example PLB

2.2 From Boolean Matching to SAT

Let

$$G = G(x_1, \cdots, x_n, L_1, \cdots, L_m, z_1, \cdots, z_o)$$

be a Boolean function in CNF representing a configuration, where variables $x_1, \ldots, x_n$ represent the input signals, variables $L_1, \ldots, L_m$ represent control bits, and variables $z_1, \ldots, z_o$ represent the intermediate circuit signals and output function of the configuration. Let $F(x_1, \ldots, x_n)$ be a Boolean function over the variables $x_1, \ldots, x_n$ (represented in CNF, for example, by computing a CNF from a truth table representation). Then, the Boolean matching problem asks if there is some setting of the control signals $L_1, \ldots, L_m$ such that for all input variables $x_1, \ldots, x_n$, there are valuations to the intermediate signals such that the output z_o of the configuration is equivalent to the output of the function F. Formally, the Boolean matching problem is formulated as the following quantified Boolean satisfiability (QSAT) problem:

$$\exists L_1, \cdots, L_m \forall x_1, \cdots, x_n \exists z_1, \cdots, z_o. \quad (7)$$
$$G(x_1, \ldots, x_n, L_1, \ldots, L_m, z_1, \ldots, z_o) \quad (8)$$
$$\wedge (z_o \equiv F(x_1, \ldots, x_n)) \quad (9)$$

As in [8, **?**], the universal quantifiers in (9) can be removed by enumerating the truth table of the function $F(x_1, \cdots, x_n)$. Therefore, (9) can be solved with SAT, where a satisfying assignment implies the function can be realized by the configuration.

With 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, 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. With QBF-based formulation, the number of clauses, existential variables, and universal variables are $O(k \cdot 2^N + 2^n)$, $O(k \cdot 2^N)$, and $O(n)$, respectively.

2.3 Example

Consider the example PLB shown in Figure 6 (b), which contains a 2-LUT and an AND-2 gate. We want to test if function f, whose truth table is shown in Figure 6 (a), can be implemented by PLB b.

We generate SAT formulation in the following steps:

1. Create CNF formulas for individual elements in the

PLB:

$$\begin{aligned} G_{LUT} &= (x_1 + x_2 + \neg L_1 + z) \cdot (x_1 + x_2 + L_1 + \neg z) \cdot \\ &\quad (x_1 + \neg x_2 + \neg L_2 + z) \cdot (x_1 + \neg x_2 + L_2 + \neg z) \cdot \\ &\quad (\neg x_1 + x_2 + \neg L_3 + z) \cdot (\neg x_1 + x_2 + L_3 + \neg z) \cdot \\ &\quad (\neg x_1 + \neg x_2 + \neg L_4 + z) \cdot (\neg x_1 + \neg x_2 + L_4 + \neg z) \\ G_{AND} &= (z + \neg f) \cdot (x_3 + \neg f) \cdot (\neg z + \neg x_3 + f) \qquad (10) \end{aligned}$$

2. Derive the programmable circuit's characteristic function from its component characteristic functions:

$$G = G_{LUT} \cdot G_{AND} \qquad (11)$$

3. Refactor the resulting Boolean function into CNF. We have two options, SAT-based formulation and QBF-based formulation.
(A) SAT-based formulation
Replicate (11) to remove the universal quantifiers on the input variables in X. This formulates G_{SAT} where a satisfiable assignment to G_{SAT} implies f can be realized in the programmable circuit.

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

(B) QBF-based formulation
Write the CNF for the truth table of the Boolean function f as follows:

$$\begin{aligned} G_f &= (\neg x_1 + \neg x_2 + \neg x_3 + f) \cdot (\neg x_1 + \neg x_2 + x_3 + f) \cdot \\ &\quad (\neg x_1 + x_2 + \neg x_3 + f) \cdot (\neg x_1 + x_2 + x_3 + f) \cdot \\ &\quad (x_1 + \neg x_2 + \neg x_3 + \neg f) \cdot (x_1 + \neg x_2 + x_3 + f) \cdot \\ &\quad (x_1 + x_2 + \neg x_3 + \neg f) \cdot (x_1 + x_2 + x_3 + \neg f) \end{aligned} \qquad (13)$$

The QBF formulation can then be expressed as follows:

$$\exists L_1 \exists L_2 \exists L_3 \exists L_4 \forall x_1 \forall x_2 \forall x_3 \exists z (G_{AND} \cdot G_{LUT} \cdot G_f) \qquad (14)$$

where a satisfiable assignment to the above QBF implies that f can be realized in the programmable circuit.

2.4 Considering Input Permutations

An important issue in Boolean matching is *input permutation*, which allows different maps from pins in a PLB to variables of a Boolean function. Figure 7 shows two Boolean functions which are equivalent under input permutation, i.e., function f_a can be transferred to f_b by input permutation $\tau = (3, 2, 1)$. Note that f_a cannot be implemented by the PLB shown in Figure 6, 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. [8] proposed to add programmable MUXs before each primary input of the target architecture (see Figure 8). All possible permutations are encoded by these MUXs. For each of these programmable MUXs, $\lceil \log K \rceil + 1$ more variables are needed to represent

$x_1x_2x_3$	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 7: (a) Truth table of $f_a = x_1 \cdot (x_2 + x_3)$, (b) Truth table of $f_b = x_3 \cdot (x_1 + x_2)$

the configuration bits (e.g., $L_{11}, L_{12}, L_{21}, L_{22}, L_{31}, L_{32}$) and the intermediate pins (e.g., z_1, z_2, z_3), and $O(K^2)$ additional clauses are needed as well. Therefore $O(K^3)$ clauses and $K \cdot (\lceil \log K \rceil + 1)$ variables are added to the formulation without considering input permutations. In practice, these MUXs actually double the size of the SAT/QBF problem if K is a large, i.e., greater than nine, since the size of a LUT is usually less than six.

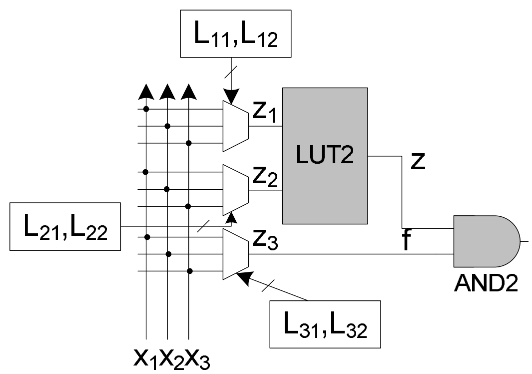


Figure 8: Considering input permutations with additional MUXs

3. EFFICIENCY IMPROVEMENT

3.1 Eliminating Permutation MUXs by Symmetries

3.1.1 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 after swapping x_i and x_j. For example, x_2 and x_3 of $f = x_1 \cdot (x_2 + x_3)$ are symmetric. In fact, we can explicitly consider all the *distinct permutations* by taking advantage of the variable symmetries exhibited in a Boolean function, instead of adding programmable MUXs as shown in subsection 2.4.

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 9. 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 symmetries is $9!/(4! \times 3! \times 1! \times 1!) = 2520$, which is 144 times less than the total number of permutations.

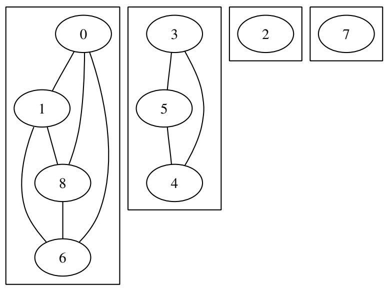


Figure 9: Symmetries in a 9-input Boolean function

Note that the time required to identify symmetries of an n-input function is $O(n^2 \cdot 2^n)$ without sophisticated data structures; however this computational cost is negligible in practice compared to the Boolean matching time as n is usually less than nine. In addition, symmetries can be detected even more efficiently based on [10]. On the other hand, the symmetries exhibited by a Boolean function help us to significantly reduce the number of permutations we need to test.

3.1.2 Symmetries in PLB Architecture

Most of the commercial PLB architectures exhibit symmetries with respect to their input pins. For example, x_1 and x_2 are symmetric, as well as x_3 and x_4 in PLB_B shown in Figure 12, which means that they can be swapped without affecting the decision of whether a Boolean function can be implemented by this PLB. We define the first and second order architecture symmetries as follows.

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

Definition 2. ***Second Order Architecture Symmetry****: The inputs $x_1, \cdot, x_k$ 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.*

To detect the symmetries exhibited in the PLB architecture, we extend the structure analysis algorithm presented in [11] 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-Invertor graph (AIG) and keep the programmable logics (e.g., LUTs) unchanged. For AIGs, an implication suppergate rooted at node n is a multi-input AND gate crated by expanding the AND gate rooted at node n until a PI or a complemented edge or a programmable device is reached. Figure 10 (B) shows the results of generated implicantion supergates for an AIG shown in (A).

The architecture symmetry can be detected by propagating symmetry information of the implication supergates in topological order from the PIs to the POs. We use the similar idea of symmetry propagation with consideration of higher order symmetries as defined in Definition 3. In the example shown in Figure 10, since the fanins of 3-input AND gate f are symmetric, the inputs of the two LUTs are symmetric under permutation $\pi(y, d, a, x), \pi(y, d, x, a)$. As $x \equiv y = b \cdot c$, a and b are symmetric.

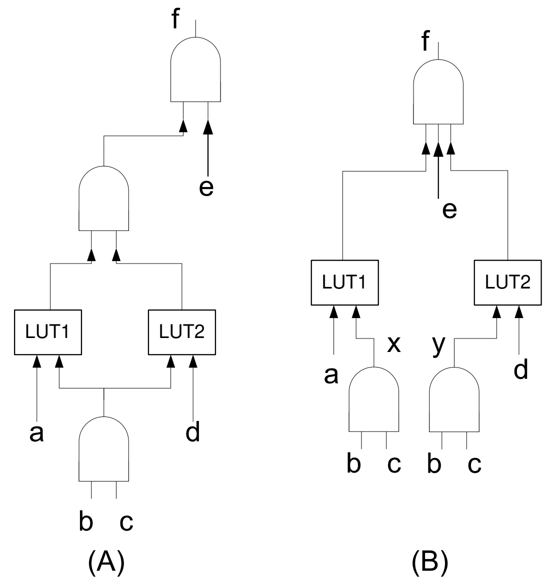


Figure 10: Illustration of architecture symmetry detection

The architecture symmetry detection can be done in the pre-process before resynthesis, and the runtime cost is not an issue. In fact, we can examine the higher order symmetries in the target PLB architecture by extending the definition of the second order symmetry.

3.2 Integration with 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 [12]. Consider the truth table shown in Figure 11. It is clear that the number of rows in the implicant table is less than the number in the truth table. In fact, the size of the implicant table for a Boolean function appearing in a real circuit is usually linear to the number of inputs of the function.

[13] presented a novel idea to consider the don't care terms in the implicant table for a LUT explicitly. In fact, the idea can be extended straightforwardly to handle general logic element (e.g., an AND gate) other than LUTs. A logic element with k inputs needs to be duplicated 2^k times. The inputs variables are set by (15) and (16) as follows. If x_1 is set to 0 or 1 then x_1' and x_1'' are both set to 0 or 1 correspondingly. But if x_1 is a "-", then x_1' is set to 0 while x_1'' is set to 1.

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

(a)

$x_1x_2x_3$	f
1-0	0
-00	0
01-	1
--1	1

(b)

Figure 11: (a) Truth table, (b) Implicant table

$$x'_i = \begin{cases} 0, & \text{if } x_i = \text{“-”} \\ x_i, & \text{otherwise} \end{cases} \tag{15}$$

$$x''_i = \begin{cases} 1, & \text{if } x_i = \text{“-”} \\ x_i, & \text{otherwise} \end{cases} \tag{16}$$

The duplication of the logic element enable the propagation of the don't terms in the implcant table. Figure 13 shows an example of such duplication for PLB_B shown in Figure 12.

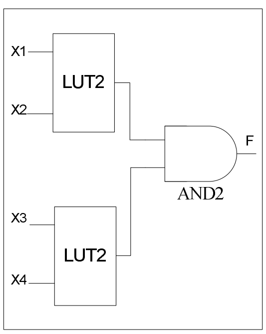


Figure 12: Architecture of PLB_B

3.3 Overall algorithm

Figure 14 shows the flow of the overall algorithm. We first pre-process the architecture of the target PLB by extracting its architecture symmetry information and generate the template of the characteristic function. For each Boolean function to be tested, we first detect the function symmetries and prune the redudant permutations based on architecture symmetries. After that, all the non-redundant permutations should be tested one by one with the replication of the characteristic function. Any of the permutation returns satstifiable indicates that the Boolean function can be implemented by the target PLB.

4. EXPERIMENTAL RESULTS

We implemented the SAT-based formulation without inserting additional programmable MUXs. Instead, we iteratively solved multiple smaller SAT problems by explicitly enumerating all distinct permutations, considering symmetries exhibited in the Boolean function and the target PLB architecture. To avoid the overhead of problem generation

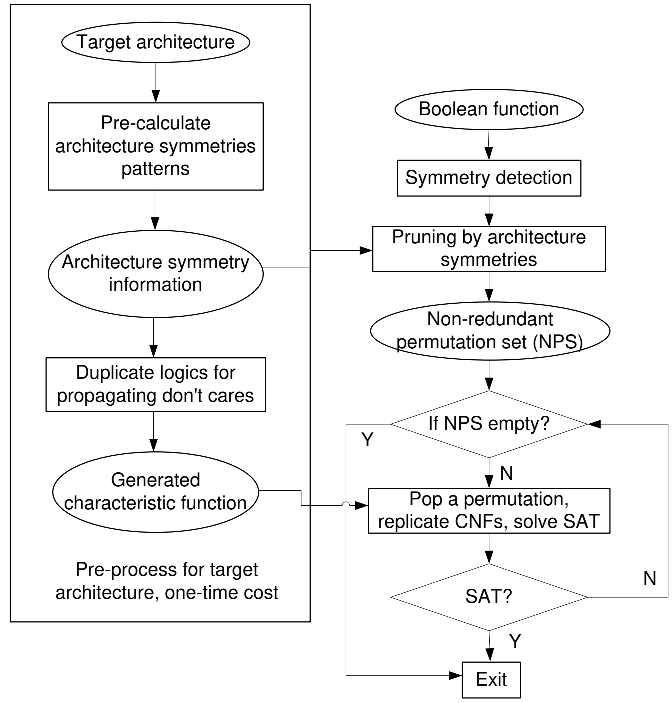


Figure 14: The flow of the overall algorithm

(replication) time in SAT formulation, we built a lookup table in advance and generated SAT clauses efficiently by performing a linear-time scan of this table.

Figure 15 shows the effectiveness of the permutation number reduction by considering symmetries. Compared to the total number of 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 architecture symmetries.

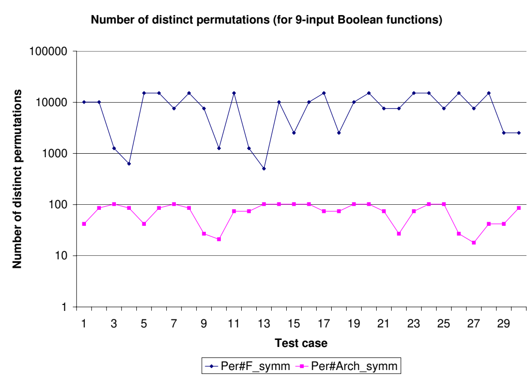


Figure 15: Number of permutations using Boolean function symmetries (F_symm) and using a combination of Boolean function symmetries and architecture symmetries (Arch_symm)

Figure 16 shows the effectiveness of consider symmetries by comparing the runtime of the original SAT formulation (Orig_SAT), the original SAT formulation with Boolean function symmetries (F_symm_SAT), and the original SAT formulation with both Boolean function symmetries and archi-

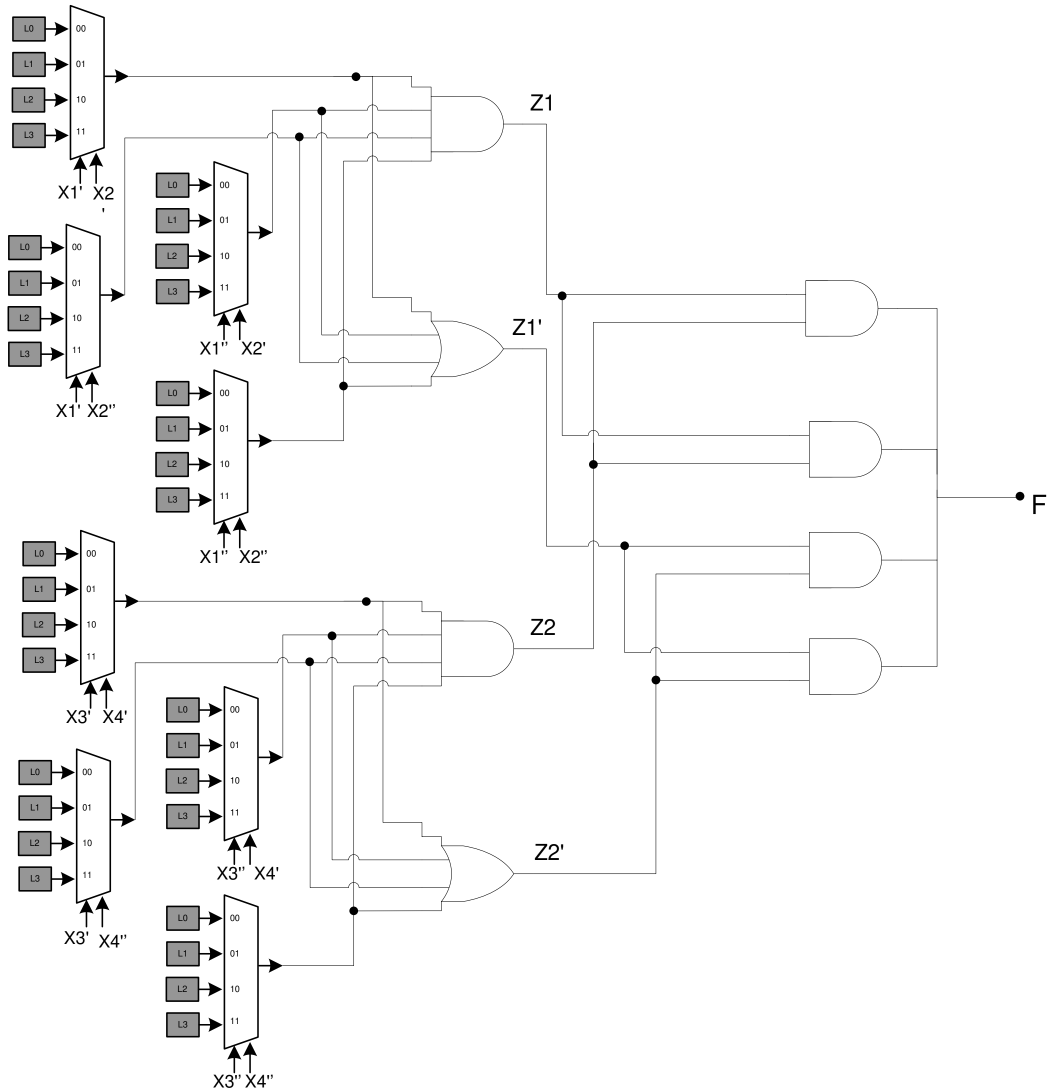


Figure 13: Duplicated PLB for PLB_B shown in Figure 12

tecture symmetries (Arch_symm_SAT). This demonstrates that Boolean function symmetry alone cannot bring significant speedup, but combined with architecture symmetry optimizations, two orders of magnitude speedup can be attained compared to the original SAT formulation. With the integration of implicant table based replication (column All), 2x more speedup can be achieved.

5. CONCLUSION

We have proposed several ideas based on symmetry reduction and implicant-based function representation to improve the efficiency of SAT-based Boolean matching for heterogeneous FPGAs. Overall, our preliminary experimental results show that these techniques can reduce runtime by approximately two orders of magnitude, making it feasible to apply SAT-based Boolean matching in industrial contexts.

6. REFERENCES

[1] J. C. et al, "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.

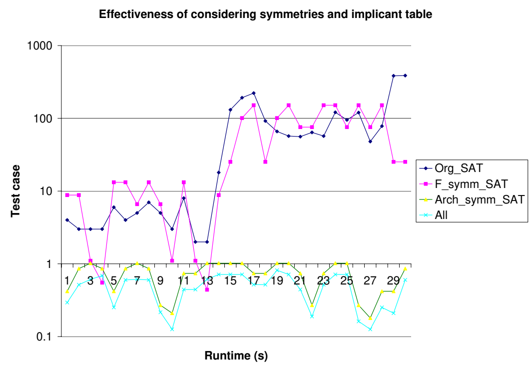


Figure 16: Runtime (seconds) of matching 30 9-input functions to PLB_D under symmetries and implicant table

[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 satbased boolean matching for fpga technology mapping," in *Proc. Design Automation Conf.*

[10] 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.

[11] 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.

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

[13] 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.