FPGA PLB Architecture Evaluation and Area Optimization Techniques using Boolean Satisfiability

Andrew C. Ling, *Member, IEEE,* Deshanand P. Singh, *Member, IEEE,*
and Stephen D. Brown, *Member, IEEE*

Abstract

This work presents a Field-Programmable Gate Array (FPGA) logic synthesis technique based upon Boolean Satisfiability (SAT). This work shows how to map any Boolean function into an arbitrary PLB architecture without any custom decomposition techniques. The authors illustrate several useful applications of this technique by showing how this technique can be used for architecture evaluation and area optimization. When evaluating FPGA architecture, the authors focus on the basic building block of the FPGA which they refer as a programmable logic block (PLB). In order to illustrate the flexibility of their evaluation framework, several unrelated PLB architectures are evaluated in an automated fashion. Furthermore, the authors show that using their technique is able to reduce FPGA resource usage by 27% on average in common subcircuits found in digital design.

Index Terms

Design Automation, Field-programmable gate array, Quantified Boolean Satisfiability, Boolean Satisfiability, Logic Synthesis, Resynthesis.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are integrated circuits characterized by two distinct features: programmable logic blocks (PLBs) and programmable interconnect structures. An FPGA consists of groups of PLBs known as *clusters* which are connected through programmable connection blocks and switch blocks to form a regular array of clusters as shown in Fig. 1. The cluster combined with its associated routing form a *tile*. Previous work has shown that grouping PLBs into clusters greatly improves the performance of FPGAs since the intra-cluster delay is an order of magnitude less than the inter-cluster delay [1].

Manuscript received November 20, 2005; revised November 30, 2005. This work was supported by the IEEE.

A. Ling is with the University of Toronto.
D. Singh is with Altera Corporation.
S. Brown is with Altera Corporation and the University of Toronto.

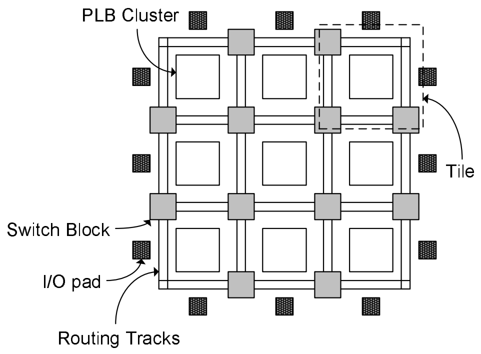


Fig. 1. An illustration of an FPGA consisting of a regular array of clustered PLBs.

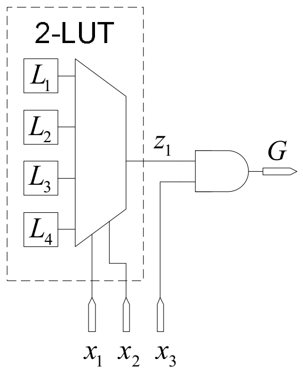


Fig. 2. Programmable Logic Block.

An example of a PLB is shown in Fig. 2. In this example, the logic block is composed of a 2-input lookup table (2-LUT) that feeds an `AND` gate. The 2-LUT is capable of implementing any arbitrary Boolean function of 2 variables. Assuming, K is the number of inputs to the LUT, the LUT is implemented with a set of 2^K static RAM (SRAM) bits that are programmed with the truth-table values for the function to be implemented (4 SRAM bits in Fig. 2). The 2 inputs (x_1, x_2) feed a multiplexer that selects the appropriate truth-table value from the SRAM bits. In cases where the PLB only consists of LUTs, we will refer to them as K-LUT architectures.

A. Motivation

In general, many modern PLBs are based on the K input lookup table (K-LUT). Although the K-LUT is very flexible, it is usually beneficial to add dedicated non-programmable logic to the PLB such as adders or XOR/AND-gates ([2], [3]). These features increase the number of functions that can be implemented by a PLB without the power, speed, and area costs associated with programmable logic. However, since this reduces the flexibility of the PLB, optimally mapping functions to these non-programmable components is difficult. This creates an area penalty which is hard to quantify objectively.

The PLB area usage significantly affects the cost of the final circuit implementation on the FPGA. Although the FPGA silicon area is dominated by the routing interconnect, the cost of implementing a circuit in an FPGA is directly proportional to the PLB capacity of the FPGA [4]. Since FPGAs are sold in a number of pre-fabricated sizes, decreasing the number of PLBs in the final circuit netlist may allow the circuit to be realized in a smaller FPGA, thereby reducing the cost of the design. Reducing the PLB usage also has a more localized effect since it allows subcircuits to be realized in a smaller number of clusters. This produces a much faster subcircuit since it reduces the number of inter-cluster connections in these subcircuits which are known to dominate FPGA delay [1].

Both the PLB architecture and the technology mapper which converts a gate-level netlist into a netlist of PLBs has a large impact on the final area of the circuit. Bad PLB designs and poor quality technology mappers can lead to very costly circuit implementations with poor performance in the FPGA. Thus, it is important to evaluate PLB architectures and develop high quality technology mappers during FPGA development.

In this paper, we present two tools that accomplish both of these goals using a new PLB function mapping approach based on Boolean Satisfiability (SAT). We will illustrate that the main benefit of our technique is its generality where it can be applied to any PLB architecture and requires no custom decomposition techniques. The first tool we present helps quantify the area usage of various PLB architectures. The second tool is a resynthesis technique which is guaranteed to optimally map functions to small subcircuits. During our resynthesis study, we focus on a class of functions known to be non-disjoint where we will show that synthesis and technology mappers have great difficulty solving optimally.

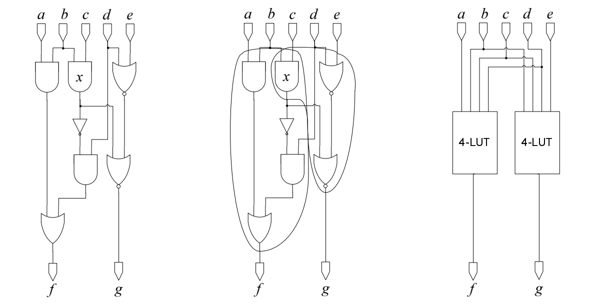
Before we introduce our tools, we present some background on the technology mapping problem in Section II. This is followed by a description of the SAT problem and an explanation on the transformation of PLB function mapping into the SAT problem in Section III. We follow this with a detailed description of our PLB evaluation method and resynthesis technique in Section IV. Finally, we present several results illustrating the generality of our technique in Section V.

II. Background

A. Technology Mapping

Technology mapping a circuit description into a netlist of PLBs occurs after logic synthesis. Logic synthesis optimizes a gate-level circuit description through a sequence of technology independent transformations [5] to improve area and delay. In this work, delay is considered proportional to the *depth* of a circuit where the depth of a node is defined as the longest path from the node to a *primary input*. A primary input is any node in a circuit with no fanin such as an input pin. The dual to this is a *primary output* which is any node in a circuit with no fanouts such as an output pin. Technology mapping takes the optimized gate-level netlist and converts it into a netlist of PLBs. Previous work showed that the depth-optimal technology mapping solution can be obtained in polynomial time using a dynamic programming procedure [6]. The disjoint relationship between logic synthesis and technology mapping often leads to technology mapped circuits that are far from optimal. In later sections, we will show methods to resolve this problem through SAT.

The process of technology mapping is often treated as a covering problem. For example, consider the process of mapping a circuit into LUTs as illustrated in Fig. 3. Fig. 3a illustrates the initial gate-level netlist, Fig. 3b illustrates a possible covering of the initial netlist using 4-LUTs, and Fig. 3c illustrates the LUT netlist produced by the covering. In the mapping given, the gate labeled x is covered by both LUTs and is said to be duplicated. In a duplication-free mapping, each gate in the initial circuit is covered by a single LUT in the mapped circuit [7]. However, surprisingly, the controlled use of duplication can lead to further area savings [8]. In contrast to the depth minimization problem, the area minimization problem was shown to be NP-hard for LUTs of size four and greater ([9], [10]). Thus, solving the area minimization problem requires heuristics.



(a) Initial Netlist (b) Possible Covering (c) LUT Mapping

Fig. 3. Technology mapping as a covering problem.

Another way to look at technology mapping is as a *cone* selection problem. The subcircuits circled in Fig. 3b are examples of cones. Technology mapping seeks to find the best set of cones that can be mapped to the current PLB architecture. "Best" is determined by the optimizing goal such as area, speed, or power. If the FPGA architecture consists solely of K-LUTs, mapping from cones to K-LUTs is a direct process since any cone with K-inputs or less can be implemented in a K-LUT. A cone with K-inputs or less is known to be K*-feasible*. Thus, to technology map circuits to K-LUTs, the circuit simply has to be decomposed into a set of K-feasible cones. However, if the FPGA architecture consists of generic K-input PLBs, mapping from cones to PLBs is much more difficult since PLBs cannot implement all possible K-feasible cones. For example, the PLB in Fig. 4 cannot implement a 3-input `OR` gate. Previous work solved this problem by using two main approaches:

- A specialized PLB is proposed and a customized mapping algorithm is implemented to map benchmark circuits to the proposed PLB [11].
- Functions are decomposed using specialized Boolean matching techniques such that it matched the structure of the PLB [12].

A problem with both of these approaches is that they require specific Boolean techniques to map functions to a given PLB architecture. We solve this problem in a general manner using SAT, allowing our technique to be applicable to any PLB architecture and any Boolean function.

Although more limited in functionality, PLBs offer speed, area, and power advantages over fully programmable K-LUTs. In general only a small subset of K-feasible cones will appear in most logic circuits; therefore, as long

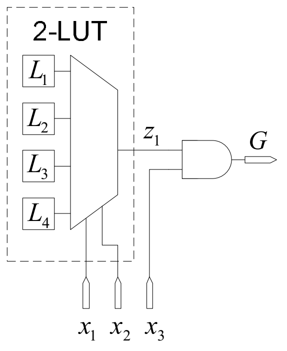


Fig. 4. Example PLB.

as a given PLB architecture captures most cones encountered in real circuits, it will be successful in implementing those circuits. In [12], the authors evaluate PLBs based upon the number of functions a given PLB can implement. We adopt a similar measure whereby we determine the flexibility of a PLB by extracting a set of K-feasible cones from benchmark circuits and determine how many of these cones can fit into the PLB where a high fit percentage is desired. Although, we adopt a similar comparison metric as in [12], no previous work has been done that has been general enough to apply to all the PLB architectures we present later. The PLB flexibility only gives a preliminary estimate on the efficiency of the PLB. To gauge how much area overhead the non-programmable components in the PLB will add to an FPGA, a full area estimate of the FPGA device is necessary. This can be calculated by deriving the number of PLBs required to implement a given circuit in conjunction with an FPGA tile area estimate containing a cluster of PLBs. We present two tools in this paper that do both these tasks.

III. Boolean Satisfiability Applied to FPGA Function Mapping

The following sections provides a brief overview of Boolean satisfiability and Quantified Boolean satisfiability. The informed reader may skip these sections and go on to Section III-B.

Boolean Satisfiability (SAT) has gained recent interest, particularly in CAD for digital circuits. The primary reason for this is that several problems that occur in CAD can be represented as a Boolean formula and thus can be solved using SAT. SAT was the first problem shown to be NP-Complete [13, ch.34] and is formally defined as the following:

Definition 3.1: The Boolean Satisfiability Problem: Given a Boolean Formula defined on variables $x_1, x_2, ..., x_n$, seek an assignment to these variables such that the Boolean formula evaluates to true. If this is possible, the Boolean formula is said to be *satisfiable* (SAT), otherwise, it is said to be *unsatisfiable* (UNSAT)

For ease of readability, we will use the term “SAT” to refer to both Boolean Satisfiability and *satisfiable* when the meaning is obvious from the context.

SAT solvers are tools that seek to solve the SAT problem. Generally, SAT solvers work on Boolean formulae in Conjunctive-Normal-Form (CNF, also known as a Product-of-Sums). A Boolean formula is in CNF if it consists only of a conjunction of clauses, where each clause contains a disjunction of literals and a literal is defined as any variable or its complement. In this form, SAT seeks an assignment of variables such that every clause in the Boolean formula has at least one literal evaluating to true. For example, Fig. 5 gives an illustration of a Boolean formula in CNF and a satisfying assignment.

$$F = (\overline{x_1} + \underbrace{\overline{x_2}}_{\text{literal}}) \cdot (x_2) \cdot \underbrace{(x_1 + \overline{x_2} + x_3)}_{\text{clause}}$$

$$x_1 = 0, x_2 = 1, x_3 = 1$$

Fig. 5. An example CNF with a satisfying assignment.

A. Quantified Boolean Satisfiability

$$F = \forall x_1 x_2 \exists x_3 (\overline{x_1} + \overline{x_2}) \cdot (x_2) \cdot (x_1 + \overline{x_2} + x_3)$$

Fig. 6. An example QBF.

SAT is actually a subset of the much more difficult problem called Quantified Boolean Satisfiability (QSAT). Definition 3.1 still holds for QSAT; however, QSAT is the more general problem of determining if a Quantified Boolean Formula (QBF) is satisfiable or not. A QBF is a Boolean formula where quantifiers are applied to its variables. For example, Fig. 6 show an example of a QBF in CNF. A Boolean formula is actually a special case of a QBF where all the variables on a Boolean formula have an implicit existential quantifier. Quantified Boolean Satisfiability (QSAT) is known to be P-Space Complete [14]. Although not formally proven, P-Space Complete problems are thought to be harder than NP-Complete problem where $PSPACE - C \neq NP - C$. An intuitive explanation of this can be shown through a simple example. Consider Equation 1, which shows a simple Boolean expression and a possible satisfying assignment. Now consider the same expression but with quantifiers added to its variables shown in Equation 2. The satisfiable assignment to the QBF shown in Equation 2 is much more elaborate than its unquantified counterpart. This simple example shows that QSAT must explore a much larger search space to find a satisfiable solution when compared against SAT.

B. Transforming FPGA Function Mapping to SAT

At its core, mapping digital circuits to FPGA fabric is the process of decomposing a circuit into a set of Boolean functions that map into a netlist of PLBs. In general, mapping Boolean functions into programmable logic is not trivial since general programmable structures with K inputs such as PLBs can only implement a small subset of K input functions. Interestingly, this problem can be represented as a QBF and solved using QSAT where a satisfying

$$\begin{gathered}(x_1 + \overline{x_2}) \cdot (x_3 + x_2 + \overline{x_1}) \cdot (\overline{x_3} + x_1) \\ x_1 = 1, x_2 = 1, x_3 = 0\end{gathered} \tag{1}$$

$$\begin{gathered}\exists x_3 \forall x_1 \exists x_2 (x_1 + \overline{x_2}) \cdot (x_3 + x_2 + \overline{x_1}) \cdot (\overline{x_3} + x_1) \\ x_1 = 1, x_2 = 1, x_3 = 0 \\ x_1 = 0, x_2 = 0, x_3 = 0\end{gathered} \tag{2}$$

Fig. 7. A QSAT complexity example.

assignment indicates that the mapping is possible. Furthermore, if satisfiable, QSAT will return the programmable configuration necessary to implement the Boolean function in the given programmable structure. We will show later how to transform QSAT into SAT and use SAT solvers to solve the simplified problem.

In order to formulate the Boolean function mapping problem as QSAT, it needs to be formalized as follows:

Problem 3.2: A Boolean function, F, with n inputs can be realized in a programmable circuit, G, with m inputs and l programmable bits, where $n \leq m$, if and only if there exists at least one configuration to the l programmable bits such that $G \equiv F$ for all inputs applied in the same manner to G and F.

The QBF representation of Problem 3.2 is shown in Fig. 8. To ensure that the inputs to F and G are applied in the same manner, we represent their inputs by the same variables, $x_1, x_2, ..., x_n$. $L_1, L_2, ..., L_l$ represent the programmable bits, and $z_1, z_2, ..., z_o$ represent any auxiliary variables found in the expression $G \equiv F$. The existence of auxiliary variables will be explained later, and is a side effect of the derivation method we use to construct $G \equiv F$.

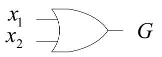
$$H = \exists L_1 L_2 ... L_l \forall x_1 x_2 ... x_n \exists z_1 z_2 ... z_o (G \equiv F)$$

Fig. 8. QBF representation of Problem 3.2.

In order to derive the expression $G \equiv F$, we adapt the circuit *characteristic function* to accomplish this. For a detailed description on the characteristic function, please refer to [15]. A characteristic function is a Boolean representation of a digital circuit, ψ, in CNF which can be modified to express $G \equiv F$. The characteristic function describes all consistent inputs, outputs, and intermediate wire vectors of the digital circuit. For example, consider the OR-gate shown in Fig. 9. Next to the OR-gate is its characteristic function truth-table. The onset of this truth-table represents all cubes that are consistent of an OR-gate such as $x_1 = 0, x_2 = 1, G = 1$. Using any standard minimization procedure, the OR-gate characteristic function can be derived as shown in Fig. 9.

Characteristic functions for large circuits can be derived from the conjunction of the characteristic functions of

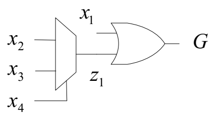
x_1	x_2	G	F_{OR}
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



$$F_{\text{OR}} = (x_1 + x_2 + \overline{G}) \cdot (\overline{x_1} + G) \cdot (\overline{x_2} + G)$$

Fig. 9. Deriving a characteristic function for an OR-gate.

their basic elements. For example, consider Fig. 10 which consists of an OR-gate fed by a 2:1 MUX. To construct the characteristic function of this circuit, we simply conjoin (logical AND) the characteristic functions of the MUX and OR-gate (the derivation of the MUX characteristic function can be accomplished using the technique shown in Fig. 9). The auxiliary variables, z_i, in Fig. 8 stem from the intermediate wires found in the configuration circuits. This is seen in Fig. 10 with variable z_1. These auxiliary variables provide a logical link between the basic characteristic functions to form a unified expression representing the entire configurable circuit.



$$\begin{aligned} F_{\texttt{MUX-OR}} =& (x_1 + z_1 + \overline{G}) \cdot (\overline{x_1} + G) \cdot (\overline{z_1} + G) \Leftarrow F_{OR} \\ & \cdot (x_4 + \overline{x_2} + z_1) \cdot (x_4 + x_2 + \overline{z_1}) \Leftarrow F_{MUX} \\ & \cdot (\overline{x_4} + x_3 + \overline{z_1}) \cdot (\overline{x_4} + \overline{x_3} + z_1) \end{aligned} \quad (3)$$

Fig. 10. Deriving characteristic functions from basic elements.

Using the previous procedure, the characteristic function, which we will refer as ψ, can be found for any configurable circuit. Using ψ, we can derive the expression for $G \equiv F$. In order to represent the equivalence operator, all instances of the variable G in ψ are replaced with the expression representing F. This substitution can be represented as $\psi[G/F]$ which will be used in later sections. Going back to original problem formulation, our goal is to find if their exists a configuration to our circuit G such that $G \equiv F$ for all inputs applied to G. Thus, as one final step, quantifiers are added to the expression $G \equiv F$ to form a final CNF expression representing Problem 3.2 as shown in Fig. 11 where $L_1 L_2 ... L_l$ represent the configuration bits and $x_1 x_2 ... x_n$ represent the inputs to G.

1) Removing Quantifiers on QBF to form SAT: Although function H in Fig. 11 can be solved using QSAT, it is often faster to remove the quantifiers found on a QBF and use SAT solvers. In [16], the author presents a method to remove all quantifiers on a QBF to convert the QSAT problem into SAT. We adopt a similar method in this work to remove the quantifiers in Fig. 11. To do this, first the unquantified expression $\psi[G/F]$ is replicated 2^n times

$$H = \exists L_1 L_2 ... L_l \forall x_1 x_2 ... x_n \exists z_1 z_2 ... z_o (\psi[G/F])$$

Fig. 11. Characteristic function based representation of Problem 3.2.

and conjoined together where n is the number of universally quantified variables $(x_1 x_2 ..., x_n)$. This is shown in the first line of Equation 4. Next, the universal variables in each replicated expression is replaced with one possible enumeration such that no two replicated expressions have identical enumerations, which is shown in Equation 5. The purpose of this is to explicitly cover all possible values of the universal variables. In addition to this, variables bound to the innermost existential quantifier $(z_1 z_2 ... z_o)$ are replaced with unique variable in each replicated expression. This preserves the meaning of their original existential quantifier. Finally, the remaining existential quantifier on the configuration variables $L_1 L_2 ... L_l$ does not have to be explicitly shown since variables without an explicit quantifier will implicitly have an existential quantifier applied to them. The resulting expression then can be passed to a SAT solver where a satisfying assignment implies that F can be realized in G.

$$\Psi = \psi_0 \cdot \psi_1 \cdot \psi_2 \cdot ... \cdot \psi_{2^n - 1} \tag{4}$$

$$\begin{aligned} =&\psi_0[x_1/0, x_2/0, ..., x_n/0, z_1/z_{o+1}, ..., z_o/z_{2o}]\cdot \\ &\psi_1[x_1/0, ..., x_n/1, z_1/z_{2o+1}, ..., z_o/z_{3o}]\cdot \\ &... \\ &\psi_{2^n-1}[x_1/1, ..., x_n/1, z_1/z_{(2^n-1)o+1}, ..., z_o/z_{2^n o}] \end{aligned} \tag{5}$$

Fig. 12. Removing the quantifiers on the QBF in Fig. 11.

$x_0 x_1 x_2$	$F(\mathbf{X})$
000	1
001	1
010	0
011	1
100	1
101	0
110	1
111	1

(a) Original Function.

$x_0 x_2 x_1$	$F(\mathbf{X})$
000	1
001	0
010	1
011	1
100	1
101	1
110	0
111	1

(b) New Function.

Fig. 13. Permutation example.

2) Permutable Inputs: In addition to having programmable bits to program the function being implemented, the inputs of programmable circuits are usually permutable. This greatly expands the number of functions a programmable circuit can implement. For example, consider the simple 3-input function shown in Fig. 13a. By changing the inputs x_1 and x_2 a new function can be realized as shown in Fig. 13b. In fact, a K-input function can be transformed to at most $K! - 1$ other functions by simply permuting its variables in every possible way. The way to model this flexibility in a digital circuit is to add multiplexers at the inputs as shown in Fig. 14. These virtual multiplexers are extremely versatile in that they can also add restrictions in routing. For example, assume configuration $V_5V_6 = 11$ would feed input x_3 into the XOR-gate (z_4). In order to prevent this, the clause $(\overline{V_5} + \overline{V_6})$ is conjoined to the PLB characteristic function.

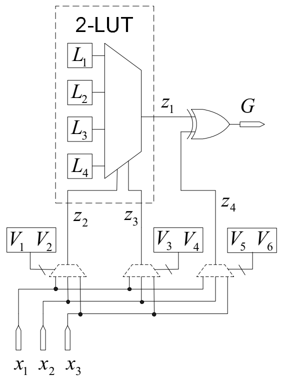


Fig. 14. Modeling permutable inputs of a programmable circuit.

3) Function Mapping using SAT Example: In order to give a better understanding of the previously described concepts, an example is provided. Taking the circuit shown in Fig. 14, we wish to determine if the Boolean function described in Fig. 13a can be realized in it.

The first step is to create the expression $\psi[G/F]$. This is found by finding the characteristic function of Fig. 14 as shown in Fig. 15. Notice that the characteristic function ψ is created from the conjunction of the basic characteristic function components that form the configurable circuit.

Following the construction of ψ, all instances of G need to be replaced by F to create the expression $G \equiv F$ as shown in Fig. 16.

The expression $\psi[G/F]$ is dependent on the variables representing all inputs, output, configuration bits, and intermediate wire variables. In this form, quantifiers can be added to these variables and the final QBF can be solved using a QBF solver where a satisfying assignment implies that function F can be realized in the configurable circuit. This QBF was shown in Fig. 11 where $x_1x_2...x_n$, $L_1L_2...L_l$, and $z_1z_2...z_o$, represent the inputs, configuration bits, and intermediate wire variables respectively.

As an extra step, there is the option to remove all quantifiers and solve the final expression using the SAT method shown previously in Fig. 12.

IV. FPGA Area Driven SAT Based Applications

The primary power of the SAT technique shown in the previous section is its generality. There are no restrictions on the type of circuit nor function that it can represent. We demonstrate several algorithms that use our SAT-

$$\begin{aligned} G_{LUT} =&(z_3 + z_2 + \overline{L_1} + z_1) \cdot (z_3 + z_2 + L_1 + \overline{z_1}) \\ &\cdot(z_3 + \overline{z_2} + \overline{L_2} + z_1) \cdot (z_3 + \overline{z_2} + L_2 + \overline{z_1}) \\ &\cdot(\overline{z_3} + z_2 + \overline{L_3} + z_1) \cdot (\overline{z_3} + z_2 + L_3 + \overline{z_1}) \\ &\cdot(\overline{z_3} + \overline{z_2} + \overline{L_4} + z_1) \cdot (\overline{z_3} + \overline{z_2} + L_4 + \overline{z_1}) \end{aligned} \tag{6}$$

$$\begin{aligned} G_{XOR} =&(z_1 + z_4 + \overline{G}) \cdot (z_1 + \overline{z_4} + G) \\ &\cdot(\overline{z_1} + z_4 + G) \cdot (\overline{z_1} + \overline{z_4} + \overline{G}) \end{aligned} \tag{7}$$

$$\begin{aligned} G_{VMUX1} =&(V_1 + V_2 + \overline{x_1} + z_2) \cdot (V_1 + V_2 + x_1 + \overline{z_2}) \\ &\cdot(V_1 + \overline{V_2} + \overline{x_2} + z_2) \cdot (V_1 + \overline{V_2} + x_2 + \overline{z_2}) \\ &\cdot(\overline{V_1} + \overline{x_3} + z_2) \cdot (\overline{V_1} + x_3 + \overline{z_2}) \end{aligned} \tag{8}$$

$$\begin{aligned} G_{VMUX2} =&(V_3 + V_4 + \overline{x_1} + z_3) \cdot (V_3 + V_4 + x_1 + \overline{z_3}) \\ &\cdot(V_3 + \overline{V_4} + \overline{x_2} + z_3) \cdot (V_3 + \overline{V_4} + x_2 + \overline{z_3}) \\ &\cdot(\overline{V_3} + \overline{x_3} + z_3) \cdot (\overline{V_3} + x_3 + \overline{z_3}) \end{aligned} \tag{9}$$

$$\begin{aligned} G_{VMUX3} =&(V_5 + V_6 + \overline{x_1} + z_4) \cdot (V_5 + V_6 + x_1 + \overline{z_4}) \\ &\cdot(V_5 + \overline{V_6} + \overline{x_2} + z_4) \cdot (V_5 + \overline{V_6} + x_2 + \overline{z_4}) \\ &\cdot(\overline{V_5} + \overline{x_3} + z_4) \cdot (\overline{V_5} + x_3 + \overline{z_4}) \end{aligned} \tag{10}$$

$$\psi = G_{LUT} \cdot G_{XOR} \cdot G_{VMUX1} \cdot G_{VMUX2} \cdot G_{VMUX3} \tag{11}$$

Fig. 15. Characteristic function of PLB seen in Fig. 14.

$$(G \equiv F) = \psi[G/F] \tag{12}$$

Fig. 16. Forming expressions $G \equiv F$.

based decision process. These applications can be categorized into PLB evaluation [17] and resynthesis. The PLB evaluation algorithm provides a quantitative area assessment to new PLB architectures, while the resynthesis algorithm helps reduce the final area of the circuit implementation in an FPGA.

A. Application to PLB Evaluation

In order to evaluate PLB area efficiency, we take two approaches. First, we develop a tool to characterize the flexibility of a PLB. The metric we use to represent PLB flexibility is a fit percentage which is the percentage of cones sampled from various circuits that can be realized in a single PLB. Using this metric, PLBs with a high fit percentage are thought to be more flexible than PLBs with a low fit percentage. This gives a rough indication on how often the non-programmable components are expected to be utilized.

Our second approach yields a more conclusive area estimate associated with a given PLB architecture. This approach uses the PLB resource usage required to implement various circuits and FPGA tile area to derive an overall area estimate. The flow of this process is illustrated in Fig. 17. As Fig. 17 shows, in order to derive the PLB usage, a generic PLB technology mapper is necessary. Since area is our primary comparison metric, one requirement for the technology mapper is to be competitive with state-of-the-art technology mappers, yet be general enough to

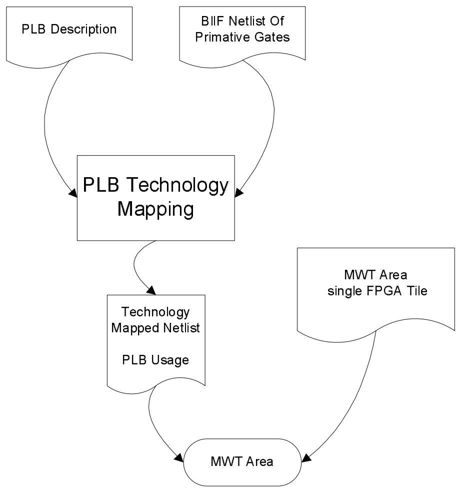

PLB Description
BlIF Netlist Of Primative Gates
PLB Technology Mapping
MWT Area single FPGA Tile
Technology Mapped Netlist
PLB Usage
MWT Area

Fig. 17. Area estimation flow for a given PLB architecture.

map to any PLB architecture. We achieve the competitive requirement by using the same heuristics as the IMap LUT mapper which outperforms all other technology mappers in terms of area; and we achieve the generality requirement by using our SAT-based function mapping technique to map functions to any PLB architecture. This is shown at the top of Fig. 17 where the PLB circuit description and netlist is passed to the PLB technology mapper. The technology mapper uses this description to generate the CNF expression representing the PLB as described previously in Section III-B. Once the PLB usage is derived, we can estimate how much area a technology mapped circuit will consume. Since FPGA area is known to be dominated by transistor area, we use minimum width transistor area [18] as our estimate of the overall area taken by the circuit. This is shown at the bottom of Fig. 17. Minimum width transistor area gives a process independent metric of the number of transistors required to implement a given circuit where larger transistors are counted as several minimum width transistors.

$$MWT_{device} = \text{CEIL}(\frac{USAGE_{PLB}}{PLBs\ per\ TILE}) \times MWT_{Tile}$$

Fig. 18. Minimum width transistor (MWT) counts for smallest device capable of fitting the given circuit.

The way we estimate total area from PLB usage and tile area is shown in Fig. 18. MWT_{Tile} is the minimum width transistor estimate of a single tile. $USAGE_{PLB}$ is the number of PLBs required to implement the given circuit. The term $\text{CEIL}(\frac{USAGE_{PLB}}{PLBs\ per\ TILE})$ returns the number of tiles required to implement a given circuit using a particular PLB architecture. Finally, MWT_{device} is the area estimate of the smallest FPGA required to implement the technology mapped circuit returned by our generic PLB technology mapper. Using this metric, a fair area comparison of various PLB architectures can be made.

1) PLB Fit Percentage: Fig. 19 shows a high-level overview of our PLB fit percentage algorithm. As stated previously, PLBs that can capture the functionality of most cones found in real circuits are desired since their non-programmable components will not be wasted. In order to help find such PLBs, our tool can be used to return a PLB cone fit percentage where a high fit percentage is preferred. This fit percentage is found by extracting a set of cones from a list of circuits, then applying our SAT decision step to remove cones that do not fit in the given architecture as shown in lines 1 and 2 of Fig. 19. By recording the number of cones generated and discarded, a fit percentage for various PLB architectures can be found.

```
1  $X \leftarrow$ GENERATECONES()
2  $Y \leftarrow$ REMOVENOFITCONES()
3  $FitPercent \leftarrow (X - Y)/X$
```

Fig. 19. An overview of the PLB evaluation algorithm.

A version of the algorithm described in [8] is used to generate and store all K-feasible cones in the graph. The K-feasible cones are generated as the graph is traversed in topological order from primary inputs to primary outputs. At every internal node v, new cones are generated by combining the cones at the input nodes.

2) Technology Mapping Using SAT: Our function mapping technique allows us to convert any K-LUT technology mapper into a K-input PLB technology mapper. As stated in Section II-A, technology mapping to LUTs can be considered as a covering problem. The same is true for K-input PLBs; however, because a K-input PLB is not fully programmable, not all K-input cones can fit into the PLB. Thus, when generating cones during the technology mapping phase, cones that do not fit into the given PLB should be discarded. This will leave a set of cones guaranteed to fit into the PLB architecture.

```
1  GENERATECONES()
2  REMOVENOFITCONES()
3  for  $i \leftarrow 1$ upto $MaxI$
4      TRAVERSEFWD()
5      TRAVERSEBWD()
6  end for
7  CONESTOPLBS()
```

Fig. 20. High-level overview of generic PLB technology mapper algorithm.

We base our work on IMap [19], an iterative K-LUT technology mapping algorithm. For a detailed description of IMap please refer to [19], which shows that IMap produces amongst the best area results of any known technology mapper. Here, we have a brief overview of the algorithm where the basic framework for our technology mapper is presented in Fig. 20. First, a call to GENERATECONES generates a set of K-feasible cones for each node in the graph, where K is the input size of the PLB. Next, a call to REMOVENOFITCONES discards all cones that cannot fit into the PLB architecture. This decision process uses SAT as described in the Section III-B. Once a set of valid cones is found, a series of forward and backward graph traversals is started to select the best cover of the graph. The cost of the cover is measured in terms of area and depth. The forward traversal, TRAVERSEFWD, selects a cone for each node, and the backward traversal, TRAVERSEBWD, selects a set of cones to cover the graph. Iteration is beneficial because every backward traversal influences the behavior of the forward traversal that follows it.

During the forward traversal, the algorithm updates the depth and the *area flow* for every node and edge encountered. Area flow is a heuristic for estimating the area of the mapping solution below a node or an edge where minimizing it leads to smaller mapping solutions as described in [19]. The definition of area flow is reshown here for convenience. As Fig. 21 shows, iteration is necessary since area flow is influenced by the covering found

$$AreaFlow(v) = 1 + \sum_{i \in fanin(C_v)} AreaFlow(i) \tag{13}$$

$$AreaFlow(i) = \frac{AreaFlow(u)}{\|fanout(C_u)\|} \tag{14}$$

Fig. 21. Area Flow definition for a node v and edge i. Note that i is an edge that flows from node u to v, C_v is a cone selected to cover v, and $\|fanout(C_u)\|$ is the number of fanouts leaving the cone covering u.

in the previous backward traversal (C_v). In the first iteration, where no previous backward traversal has occurred, C_v is estimated as the node v itself. Also, $\|fanout(C_u)\|$ must be estimated and is taken as the weighted average of the previous iterations where it is initially estimated as $\|fanout(u)\|$. A detailed description on this procedure can be found in [19].

At each internal node v, a cone rooted at v is selected to cover v and some of its predecessors in a mapping solution. The quality of the mapping solution is determined by the cone selection procedure. During area-oriented mapping, on the first mapping iteration, the cone with the lowest area flow is selected. If cones have equivalent area-flow, the cone with the lowest depth is selected. During depth-oriented mapping, the first forward traversal establishes the optimal mapping depth, $ODepth$, which can then be used in subsequent iterations to bound the depth of cones selected at every node. Using the optimal depth and the height of a node v, a bound can be defined on the depth of a cone C_v as follows

$$depth(C_v) \leq ODepth - height(v). \tag{15}$$

The height of a node or cone is defined as the longest path from that node or cone to a primary output of the circuit. Cones that meet the bound requirement are preferred and among a set of cones that meet the bound requirement, cones with lower area flows are selected. This selection strategy ensures that the mapping solutions will still achieve the optimal depth selected while minimizing area.

During the backward traversal, internal nodes of the graph are visited in the reverse topological where a cover of cones is produced. During this traversal, the $height(v)$ of all internal nodes are updated to the height of the cone covering it. This is for use in Equation 15 in the next forward traversal. If v is found in several cones, the largest height is used.

Finally, a call to CONESTOPLBS converts the cones selected by the final backward traversal into PLBs.

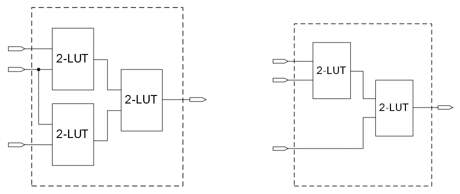
3) Generating k-Feasible Cones: A version of the algorithm described in [8] is used to generate and store all K-feasible cones in the graph. The K-feasible cones are generated as the graph is traversed in topological order from primary inputs to primary outputs. At every internal node v, new cones are generated by combining the cones at the input nodes. The original IMap algorithm combined the cones in every possible way. In our work, in order to prune the number of cones explored, the cone generation algorithm collapses cones if they have no more $(k+e)$ distinct inputs in total (i.e. $(l+m-1) \leq (k+e)$ where l and m are the number of distinct inputs to each cone being collapsed into one). As long as e was set to a sufficiently high number (2 in our experiments), this heuristic increased the speed of the cone generation process without significantly impacting the quality of the mapping solution.

B. Resynthesis

In this section, we address the problem of technology mapping where the technology mapper often fails to find an optimal solution for subcircuits. We consider state-of-the-art K-LUT technology mappers publicly available. As stated in Section II-A, technology mapping is a step that follows gate-level synthesis. Furthermore, gate-level synthesis and technology mapping are very disjoint steps. A problem arises when the cost metrics between gate-level synthesis and technology mapping do not coincide. This is explored in detail in Section V. To solve the problem between synthesis and technology mapping, we introduce a post technology mapping step that optimally resynthsizes small subcircuits.

1) Subcircuit Resynthesis: Resynthesizing several subcircuits in a sliding window fashion will reduce the overall LUT count of the entire circuit. Since a subcircuit of LUTs forms a cone, the subcircuit resynthesis problem is the function fitting problem as stated previously in Problem 3.2. In this case, the Boolean function is extracted from the subcircuit consisting of X K-input LUTs and then is checked if it can fit into a programmable structure containing less than X K-input LUTs. This check is done using our SAT-based technique.

To illustrate this process, consider Fig. 22. The original cone 22a consists of three 2-LUTs which implements a three input function. Since only three inputs enter the cone, it may be possible to resynthesize Fig. 22a into Fig. 22b to save one LUT. To determine if resynthesis from Fig. 22a to 22b is possible, Fig. 22b is converted into a CNF



(a) Original Cone (b) Resynthesized Cone

Fig. 22. Resynthesis of three-input cone of logic.

expression as described in Section III-B and the function extracted from Fig. 22a is tested using SAT to see if it fits into 22b. If the expression is satisfiable, resynthesis can proceed successfully.

Unfortunately, resynthesizing subcircuits with more than 6 distinct inputs cannot be used in real-time resynthesis engines due to speed limitations. However, this technique can be used to build a cache of optimal configurations of digital logic blocks. This is a similar technique used in [20] where the authors focus on multiplexer transformations. In [20], the authors traverse a technology mapped netlist and identify multiplexers. Once identified, they replace the multiplexer circuit with their cached optimal configuration. This is a linear time operation with respect to the circuit size, and thus will have negligible impact on the running time. Our tool can help extend this process and find the optimal configuration of several types of subcircuits that technology mapping fails to find.

V. Results

To demonstrate the previously described techniques, we provide several concrete examples here. We first show results for our PLB evaluation method and follow with results for our area optimization techniques.

When evaluating PLBs, we show that the main benefit of our technique is its generality. We prove this using three different approaches:

- We measure the flexibility of several PLB architectures.
- We explore a large set of hardwired PLB configurations in an automated fashion.
- We incorporating routing features into the PLB evaluation.

Each approach emphasizes how our technique can be applied to any PLB without any modification to our PLB technology mapper or evaluation framework.

After our PLB evaluation results, we focus on our area optimization technique. Here, we resynthesize several common subcircuits using our sliding window technique. This is followed by a discussion on why synthesis and technology mapping misses the optimal configuration provided by our resynthesis technique.

When running our experiments, we focus on the MCNC benchmark circuit set [21]. The SAT solver used to drive our function mapper was the Chaff solver developed by M. W. Moskewicz et al. [22]. All of our algorithms were built on top of the Berkeley MVSIS project [23].

A. PLB Evaluation

1) Generality of Technique: First, to illustrate the generality of our evaluation algorithm, several unrelated PLB architectures were evaluated. Fig. 23 shows the five different PLB architectures used for evaluation.

To derive the fit percentages, approximately 1000 K-input cones were extracted from each circuit sampled, where K was the input size of the PLB. Cones were extracted randomly to generate a large set of unrelated logic functions. Table I summarizes our results. Each column shows a fit percentage per circuit for each respective PLB, and *% Fit* shows the final fit percentages when considering all the circuits. Note that the cone fit percentage varies wildly for all PLBs depending on the circuit. This shows that PLB usefulness is dependent on the application of the circuit. Interestingly, PLB(b) failed for all circuits except the ALU circuit (C2670). A reason for this is because PLB(b) uses an XOR-gate which are very rare in most control circuits and are generally used for arithmetic logic.

PLB(e) was only able to fit 9-input cones for a few circuits. This was expected since PLB(e) is a simplified version of a commercial PLB primarily used to implement 5-input functions or a 4:1 MUX, and is rarely used as a general 9-input function generator [24]. In order to obtain a more accurate picture of this PLB's functionality, in

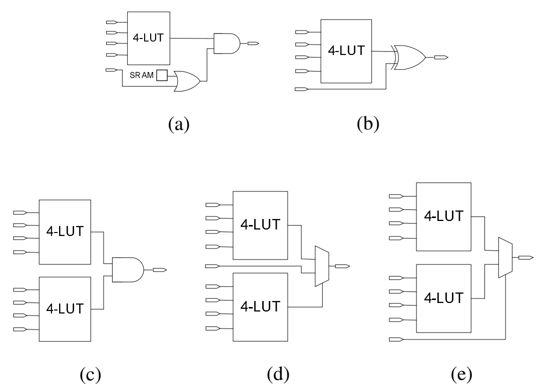


Fig. 23. Five PLBs used in evaluation experiments.

TABLE I
PLB EVALUATION RESULTS.

Circuit	*a*	*b*	*c*	*d*	*e*
`C2670`	27.9	1.59	41.8	0.00	0.00
`ex5p`	91.4	0.00	49.7	0.00	0.00
`clma`	61.5	0.00	40.5	1.29	1.29
`dalu`	78.2	0.00	38.5	0.00	0.00
`des`	12.2	0.00	72.6	0.00	0.00
`i9`	87.4	0.00	18.8	0.00	0.00
`x3`	21.5	0.00	38.9	20.2	20.1
`f51m`	21.7	0.00	18.0	0.00	0.00
`misex3`	70.2	0.00	45.4	11.8	12.9
`mm30a`	20.8	0.00	0.20	0.00	0.00
`mult16b`	2.91	0.00	0.00[1]	0.00	0.00
% Fit	46.0	0.151	36.4	3.34	3.44

addition to generating 9-input functions for PLB(e), 6, 7, and 8-input functions were evaluated. This is shown in Table II. As the numbers show, this PLB looks much more useful when considering a wider range of functions.

2) FPGA Architecture Exploration: Here, we demonstrate how to use our SAT-based technology mapper to give a full area comparison between various PLB architectures. The standard architectures that we used for area comparisons were 4 and 5-LUT based FPGAs. Our goal is to prove the generality of our technique by exploring the resource usage of a wide range of possible PLB structures using our technology mapping algorithm. This is followed by incorporating MWT areas and the routing architecture to get a full comparison. For all experiments in this section, we optimized our circuits using SIS [25] with *script.rugged*. These optimized circuits were then passed to our SAT-based technology mapper. Since we wanted to achieve the smallest area results, our technology mapper was tuned to optimize for area while ignoring depth for all circuits.

[1]no functions with 8-inputs or more could be found in this circuit

TABLE II
THE PERCENTAGE OF CONES THAT FIT INTO FIG. 23(E).

Circuit	*6-input*	*7-input*	*8-input*
`C2670`	1.00	0.00	0.00
`ex5p`	7.07	1.02	2.10
`clma`	8.62	5.13	2.24
`dalu`	6.77	1.45	0.00
`des`	0.00	0.00	0.00
`i9`	2.20	0.34	0.00
`x3`	28.3	25.2	17.7
`f51m`	38.7	11.5	0.10
`misex3`	16.7	15.7	13.8
`mm30a`	13.1	3.35	2.84
`mult16b`	0.00	0.00	0.00
Total % Fit	14.7	6.41	3.50

First, we focus on the highlighted steps shown in Fig. 24. As Fig. 24 illustrates, the PLB description can be

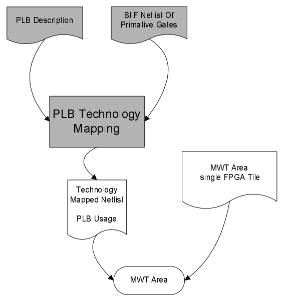

PLB Description
BlIF Netlist Of Primative Gates
PLB Technology Mapping
Technology Mapped Netlist
PLB Usage
MWT Area single FPGA Tile
MWT Area

Fig. 24. Steps in overall area estimation flow to derive the PLB usage.

modified to explore various PLB architectures. In order to explore several PLB architectures in an automated fashion, instead of manually creating several PLB structures, we used the PLB shown in Fig. 25 as our PLB. The PLB shown has four distinct inputs consisting of a 3-LUT in conjunction with a 3-input hardwired function. The benefit of this PLB architecture is that there are $2^8 = 256$ possible hardwired functions. Each possible hardwired function can be explored quickly by modeling the hardwired function as a preconfigured 3-LUT which will be common to all PLBs in the FPGA. To illustrate the exploration process, we technology mapped one benchmark circuit to all 256 possible PLB hardwired configurations. The results are illustrated in Table III which summarizes the hardwired SRAM preconfigurations that produced the lowest and highest overhead in terms of PLB usage. Row *4-LUT* shows the PLB usage if only 4-LUTs were used. Column *Ratio 4-LUT* shows the ratio of PLB usage when compared against the 4-LUT architecture. The configurations that produced the lowest PLB usage occurred for the preconfiguration values of 0x00 and 0xFF. This corresponds to the PLBs shown in Fig. 26. The original intent of this experiment was to show how we could evaluate a wide range of PLBs in an automated fashion. Since

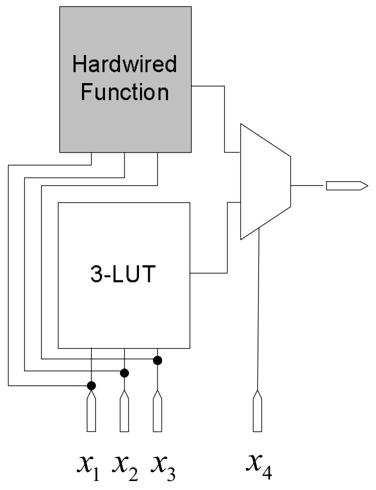


Fig. 25. 3-input hardwired support function based PLB.

TABLE III
SUMMARY OF SRAM PRECONFIGURATIONS THAT PRODUCED LOW AND HIGH PLB USAGAGE FOR CIRCUIT EX5P.

Bit Mask	*PLB Usage*	*Ratio 4-LUT*
0x00	818	1.11(low)
0xFF	906	1.23(low)
0xF7	1003	1.36(high)
0x70	1011	1.37(high)
4-LUT	737	1.00

.

this is just an illustrative study, only one benchmark circuit was used. A more conclusive study, however, would include a wide range of benchmark circuits. Having said that, note that the 0x00 and 0xFF configuration model an AND and OR gate cascade as seen in Fig. 27. Digital logic contains a large degree of AND and OR gates, thus we suspect that other benchmark circuits would yield similar results. Furthermore, the area results we obtain for the PLB models shown in Fig. 27 coincides with industrial findings where AND and OR-gate cascade structures are common in industrial FPGAs such as Altera's Apex20k [2].

Table III clearly shows that there is an associated PLB usage overhead when removing some programmability in the PLB. However, as long as the increase in PLB usage is amortized by the decrease in silicon area of the non-programmable components, the loss in flexibility may be beneficial. To explore this idea, we focused on the two PLB configurations that produced the lowest area overhead in Table III. Both of these PLB configurations can be realized as a single 4-input PLB using a 2:1 MUX and SRAM bit as shown in Fig. 28a which we refer as 4-MUX-PLB.

To estimate the area performance of the 4-MUX-PLB architecture, we technology mapped 182 MCNC benchmark circuits to it and recorded the PLB usage for each circuit. Furthermore, since we want to illustrate that our evaluation tool works for any PLB architecture and we know that the cascade structure of 4-MUX-PLB is similar to industrial

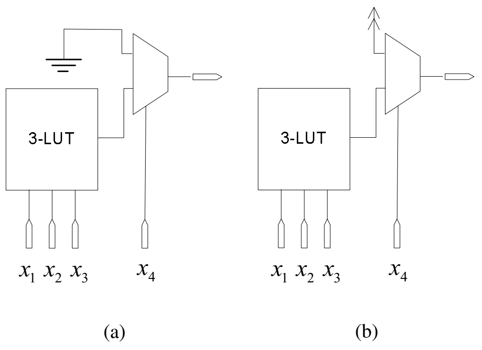


Fig. 26. PLB configurations that produced the smallest area results when compared against a simple 4-LUT architecture.

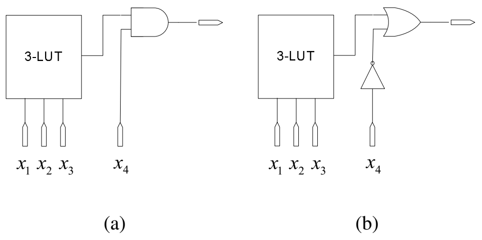


Fig. 27. Equivalent PLB representation using basic gates.

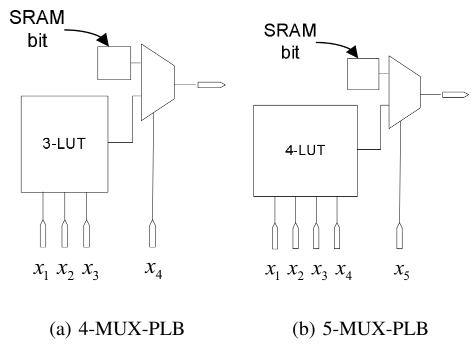


Fig. 28. Candidate PLB used in area experiments.

5-input PLBs, we also compared against the 5-LUT architecture using a 5-input PLB shown in Fig. 28b, referred as 5-MUX-PLB, for technology mapping. After the PLB usage is recorded, we use those numbers to calculate the MWT area and compare the final results.

A summary of the PLB usage is shown in Table IV where only the 20 largest circuits of the 182 circuits tested are shown in detail. A geometric mean of PLB usage ratios is shown where both a comparison against 4-LUTs and 5-LUTs is given. *Ratio* shows the ratio of the PLB architecture against the LUT architecture. *GeoMean* is the

TABLE IV
MCNC PLB USAGE SUMMARY AND COMPARISON AGAINST 4-LUT AND 5-LUT BASED FPGA ARCHITECTURES.

Circuit	*4-MUX-PLB*	*4-LUTs*	*Ratio*	*5-MUX-PLB*	*5-LUTs*	*Ratio*
apex3	655	598	1.10	559	518	1.08
ex5p	753	737	1.02	629	617	1.02
s298	926	794	1.17	718	649	1.11
tseng	993	772	1.29	771	708	1.09
alu4	1,031	901	1.14	812	735	1.10
apex4	1,032	935	1.10	860	792	1.09
misex3	1,074	949	1.13	851	776	1.10
apex2	1,123	1,023	1.10	911	857	1.06
spla	1,126	1,006	1.12	922	823	1.12
seq	1,140	1,003	1.14	904	838	1.08
ex1010	1,226	1,001	1.22	892	835	1.07
dsip	1,373	918	1.50	916	913	1.00
pdc	1,431	1,253	1.14	1,143	1,023	1.12
des	1,577	1,189	1.33	1,124	1,020	1.10
bigkey	1,938	1,032	1.88	1,032	690	1.50
elliptic	3,282	2,226	1.47	2,071	1,914	1.08
clma	3,841	3,254	1.18	3,050	2,674	1.14
frisc	4,001	2,380	1.68	2,252	2,074	1.09
s38584.1	4793	4018	1.19	3705	3082	1.20
s38417	5007	4084	1.23	3942	3489	1.13
GeoMean			1.201			1.106

geometric mean of the *Ratio* for all 182 circuits tested. Fig. 29 and 30 has a graphical view of the PLB usage overhead for all the circuits where the overhead is calculated as $Overhead = Ratio - 1$. The results show that the 4-MUX-PLB has a 20.1% usage overhead when compared against a 4-LUT architecture, and 5-MUX-PLB has a 10.5% overhead when compared against a 5-LUT architecture. Again, this PLB usage increase may be acceptable if it is amortized by the decrease in MWT area.

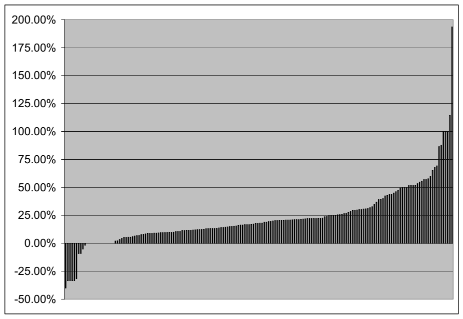


Fig. 29. 182 MCNC benchmark circuit PLB usage overhead when comparing the 4-MUX-PLB against the 4-LUT architecture. The Geometric mean of the overhead is 20.0%.

Finally, we finish our area estimation demonstration with the steps highlighted in Fig. 31. In these steps, we use the PLB usage counts to find a full area comparison. This requires the minimum width transistor area for an FPGA

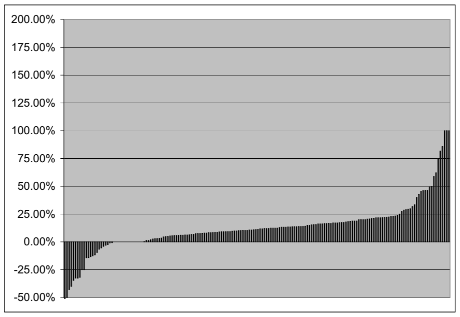


Fig. 30. 182 MCNC benchmark circuit PLB usage overhead when comparing the 5-MUX-PLB against the 5-LUT architecture. The Geometric mean of the overhead is 10.5%.

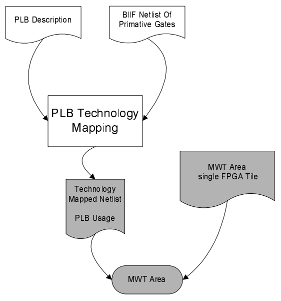

PLB Description
BLIF Netlist Of Primative Gates
PLB Technology Mapping
Technology Mapped Netlist
PLB Usage
MWT Area single FPGA Tile
MWT Area

Fig. 31. Steps in overall area estimation flow to derive final area estimation.

tile for each architecture. To model the tile, we used several standard cluster and routing characteristics which have proven to substantially reduce area while having minimal effect on the performance of the FPGA [1]. We kept these parameters constant for each input size to ensure that any changes in area were due to the change in PLB architecture, and not the routing architecture. For the 4-input PLBs, each cluster contained 10 PLBs ($N = 10$), 22 inputs feeding the cluster ($I = 22$), 1 clock input ($CLK = 1$), 50 routing tracks ($W = 50$), a feedback flexibility of 0.5 ($F_{cfb} = 0.5$), a connection block flexibility of 0.5 ($F_c = 0.5$), and a switch block flexibility of 3 ($F_s = 3$). The tile characteristics for the 5-input PLBs were the same except the number of inputs feeding the cluster was changed to 27 ($I = 27$). The feedback flexibility (F_{cfb}) describes the fraction of cluster outputs that feedback into the inputs of each PLB within the same cluster, the connection block flexibility (F_c) describes the fraction of PLB inputs that connect to the cluster inputs, and the switch block flexibility (F_s) describes the fraction of connections between inter-cluster routing tracks found in the switch blocks. These parameters are illustrated in Fig. 32 and are described in detail in [1]. We used VPR [18] to derive our final minimum width transistor estimate for each tile. To estimate the area of each PLB and cluster, we modified VPR to use the numbers in Table V while no changes to the routing estimation code was necessary since the routing architecture parameters (W, F_{cfb}, F_c, etc.) are already supported in VPR. Table V show the minimum width transistor area for an entire PLB tile for architecture 4-MUX-PLB, 5-MUX-PLB, and a 4 and 5-LUT architecture. MWT_{PLB} lists the minimum width transistor area for a single PLB, $MWT_{cluster}$ lists the minimum width transistor area for a cluster of PLBs, and MWT_{tile} lists

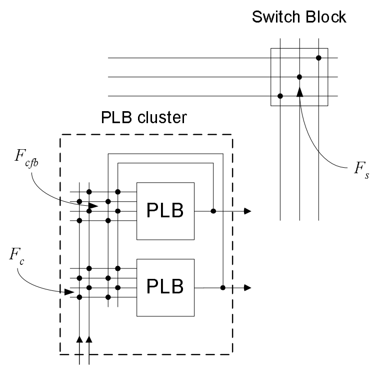


Fig. 32. An illustration of FPGA architecture parameters.

the minimum width transistor area for an entire tile. In order to model the components, we used the minimum width transistor areas listed in [18].

TABLE V
COMPARISON OF THE TILE AREAS FOR THE PLBS SHOWN IN FIG. 28 AGAINST A 4-LUT AND 5-LUT ARCHITECTURE.

PLB	MWT			*Ratio 4-LUT*			*Ratio 5-LUT*		
	PLB	$cluster$	$tile$	PLB	$cluster$	$tile$	PLB	$cluster$	$tile$
4-MUX-PLB	112	4,024	9,895	0.67	0.88	0.95	0.37	0.58	0.75
5-MUX-PLB	181	5,714	11,987	1.08	1.25	1.15	0.60	0.83	0.91
4-LUT	168	6,914	10,455	1.00	1.00	1.00	0.59	0.66	0.79
5-LUT	301	10,455	13,187	1.79	1.51	1.26	1.00	1.00	1.00

Using Table V and the PLB usage, an area estimate of each circuit was calculated using the equation shown in Fig. 18 which is reshown in Fig. 33 for convenience. The results and comparisons are shown in Table VI. A graphical view of the overhead for all circuits is shown in Fig. 34 and 35. The results clearly show that the 4-MUX-PLB has 12.4% area overhead when compared against the 4-LUT architecture. However, interestingly, the 5-MUX-PLB reduces area by 2.2% when compared against the 5-LUT architecture. This change in trend is due to two factors: the ratio of PLB usage overhead in 5-MUX-PLB to 5-LUTs is smaller than 4-MUX-PLB to 4-LUTs; the minimum transistor area reduction of 5-MUX-PLB to 5-LUTs is much larger than 4-MUX-PLB to 4-LUTs. The larger area reduction for 5-MUX-PLB to 5-LUTs is not surprising since the number of transistors required to implement a LUT is exponentially proportional to the LUT size. Although the results show minor deviations, the purpose of this set of experiments was to illustrate a full PLB evaluation using our method. No custom steps were required nor were any changes needed in our technology mapper or evaluation framework. The only manual steps included the derivation of the PLB architecture and calculating the MWT areas of each tile.

$$MWT_{device} = \text{CEIL}\left(\frac{USAGE_{PLB}}{PLBs\ per\ TILE}\right) \times MWT_{Tile}$$

Fig. 33. Minimum width transistor (MWT) counts for smallest device capable of fitting the given circuit.

TABLE VI

MINIMUM-WIDTH TRANSISTOR AREA OF MCNC CIRCUITS WHERE ONLY THE LARGEST 20 CIRCUITS HAVE BEEN SHOWN IN DETAIL.

Circuit	*4-MUX-PLB*	*4-LUT*	*Ratio*	*5-MUX-PLB*	*5-LUT*	*Ratio*
apex3	653,070	627,300	1.04	671,272	685,724	0.98
ex5p	752,020	773,670	0.97	755,181	817,594	0.92
s298	920,235	836,400	1.10	863,064	857,155	1.01
tseng	989,500	815,490	1.21	934,986	936,277	1.00
alu4	1,029,080	951,405	1.08	982,934	975,838	1.01
apex4	1,029,080	982,770	1.05	1,030,882	1,054,960	0.98
misex3	1,068,660	993,225	1.08	1,030,882	1,028,586	1.00
apex2	1,118,135	1,076,865	1.04	1,102,804	1,134,082	0.97
spla	1,118,135	1,055,955	1.06	1,114,791	1,094,521	1.02
seq	1,128,030	1,055,955	1.07	1,090,817	1,107,708	0.98
ex1010	1,217,085	1,055,955	1.15	1,078,830	1,107,708	0.97
dsip	1,365,510	961,860	1.42	1,102,804	1,213,204	0.91
pdc	1,424,880	1,317,330	1.08	1,378,505	1,358,261	1.01
des	1,563,410	1,244,145	1.26	1,354,531	1,345,074	1.01
bigkey	1,919,630	1,087,320	1.77	1,246,648	909,903	1.37
elliptic	3,255,455	2,331,465	1.40	2,493,296	2,531,904	0.98
clma	3,809,575	3,408,330	1.12	3,656,035	3,534,116	1.03
frisc	3,967,895	2,488,290	1.59	2,709,062	2,742,896	0.99
s38584.1	4,749,600	4,202,910	1.13	4,447,177	4,074,783	1.09
s38417	4,957,395	4,276,095	1.15	4,734,865	4,602,263	1.02
GeoMean			1.124			0.978

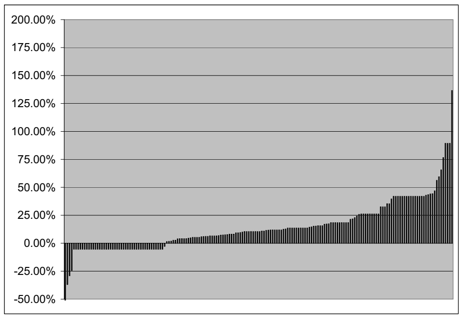


Fig. 34. Area overhead when comparing the 4-MUX-PLB against the 4-LUT architecture. The Geometric mean of the overhead is 12.4%.

3) Adding Routing Constraints with SAT: It is clear from Table V that the routing components of the PLB dominate the total area. Thus, it would be beneficial to explore both changes in routing architecture along with changes in PLB architecture. Fortunately, our SAT-based technology mapper can easily incorporate routing restrictions to explore various constraints on the PLB inputs. Our comparison methodology does not change, the only modification is to our circuit description such that the technology mapper restricts routing using our specifications. Also, our MWT area must represent any changes in the restrictive routing architecture. The way we illustrate this is through PLB(a) reshown here in Fig. 36. The previously shown results in Table I clearly show that PLB(a) should be robust enough to successfully implement a wide range of circuits. We could support PLB(a) using the same

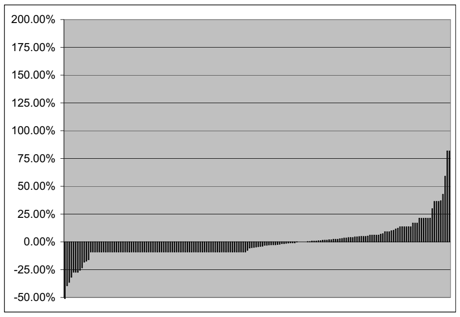


Fig. 35. Area overhead when comparing the 5-MUX-PLB against the 5-LUT architecture. The Geometric mean of the overhead is -2.2% (reduction in area).

routing architecture as 5-LUTs. However, the routing area would amortize any major reductions in PLB usage if compared against a 4-LUT architecture. A solution to this problem involves restricting the routing as illustrated in Fig. 36b. In this case, changes to the intra-cluster routing would only add an `OR`-gate, an `AND`-gate, and an extra SRAM bit. Note that our transistor implementation of the `AND`-`OR`-gate cascade is shown in Fig. 37, which requires only 8 minimum width transistors. As for the inter-cluster routing, only one extra input and output per cluster would need to be added. The goal of restricting routing is the dual of the previous experiments: we are adding logic to our PLB in an attempt to reduce the PLB usage which hopefully will amortize the area overhead of the additional logic. To model the routing restriction in the PLB technology mapper, we allowed, at most, one PLB

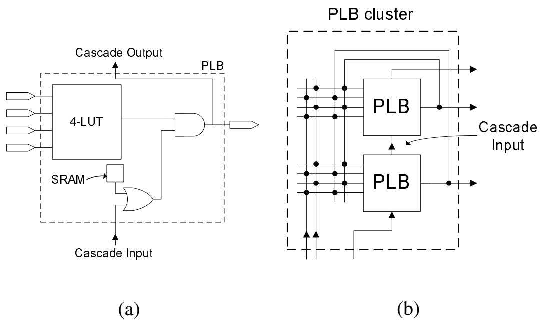


Fig. 36. `AND` based PLB (a) with restricted routing (b).

to feed a cascade input and disallowed other components, such as IO pins to feed a cascade input. This constraint is easily added to the characteristic function using the technique described in Section III-B.2. Thus, no changes to the technology mapper was necessary. However, we had to modify our intra-cluster MWT model to account for the more restrictive routing. In the unrestricted routing case, a multiplexer would feed the `AND` cascade input since several inputs could feed the cascade input. However, in our restrictive case, only the adjacent PLB output can feed the cascade input, hence a direct wire to connect a PLB output with an adjacent cascade input is sufficient.

To calculate the minimum width transistor area of the PLB tiles, we assumed the same cluster characteristics

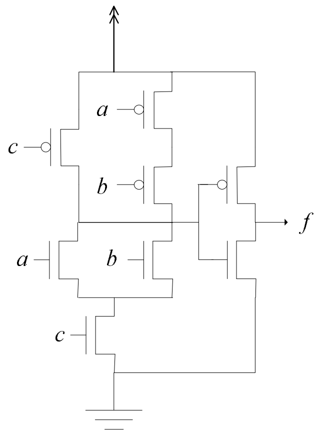


Fig. 37. Implementation of a cascade AND and OR-gate where $f = c \cdot (a + b)$

as the previous experiments (4-input PLBs: $N = 10$, $I = 22$, $W = 50$, $CLK = 1$, $F_{cfb} = 0.5$, $F_c = 0.5$, $F_s = 3$ 5-input PLBs: $N = 10$, $I = 27$, $W = 50$, $CLK = 1$, $F_{cfb} = 0.5$, $F_c = 0.5$, $F_s = 3$). In the case of the restricted routing architecture, most of the 4-input PLB characteristics were used, with a change to the number of inputs and outputs to 23 and 11 respectively. The inter-cluster routing was not changed, thus no changes to the VPR MWT routing estimates were necessary. The minimum width transistor area for a tile of PLB(a) is shown in Table VII where they are compared against the 4-LUT architecture. *No Restrict* refers to the non-restricted routing architecture, *OR-Restrict* refers to the restricted routing architecture (cascade input), and *4-LUT* is the 4-LUT architecture. As the results show, the cascade input adds some overhead compared to the 4-LUT architecture. However, the OR-Restrict architecture uses much less area, particularly when comparing the cluster and tile area. Finally, as with the previous 4-MUX-PLB and 5-MUX-PLB architectures we compare the PLB usage and MWT

TABLE VII
PLB-AND MINIMUM WIDTH AREA NUMBERS COMPARED AGAINST A 4-LUT ARCHITECTURE.

PLB	MWT			Ratio 4-LUT		
	PLB	$cluster$	$tile$	PLB	$cluster$	$tile$
No Restrict	189	5774	12047	1.12	1.26	1.15
OR-Restrict	182	4724	10930	1.08	1.03	1.05
4-LUT	168	4584	10455	1.00	1.00	1.00

area against the 4-LUT architecture. Here, we technology mapped our 182 MCNC benchmark circuits and calculated the geometric mean of the ratio of our PLB architecture to a simple 4-LUT architecture where we compare both PLB usage and MWT area. This is summarized in Table VIII where only the geometric mean is shown. The *PLB* column indicates which PLB architecture is being compared, the *4-LUT* columns report the geometric means of the comparisons done against the 4-LUT architecture, and the *No Restrict* columns report the geometric means of the comparisons done against the non-routing restricted architecture. As the results clearly show, although restricting the routing architecture increases the PLB usage by about 6%, the reduction in routing area amortizes this increase and reduces the minimum transistor width area by about 4%. These results were expected since the simple routing architecture of the cascade input uses much less area than a fully routable 5-input PLB. This proves that our SAT based technology mapper can easily and successfully add various routing constraints to the PLB architecture.

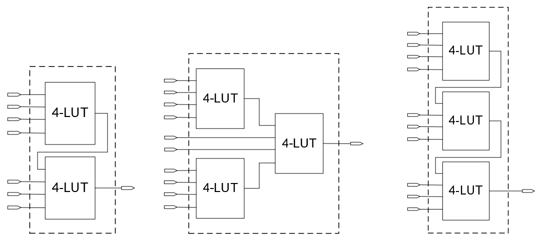
TABLE VIII
COMPARISONS AGAINST THE 4-LUT ARCHITECTURE.

	4-LUT		*No Restrict*	
PLB	$\frac{Usage_{PLB}}{Usage_{4-LUT}}$	$\frac{MWT_{PLB}}{MWT_{4-LUT}}$	$\frac{Usage_{Restrict}}{Usage_{NoRestrict}}$	$\frac{MWT_{Restrict}}{MWT_{NoRestrict}}$
No Restrict	0.91	1.06	1.00	1.00
OR-*Restrict*	0.97	1.02	1.06	0.96

B. Resynthesis Results

In this section, we illustrate the problems associated with state-of-the-art technology mappers and synthesis by technology mapping some very common subcircuits found in digital designs and resolve these problems using our SAT based technique. We wrote several designs in Verilog code, synthesized the code into basic gates using VIS [26], then optimized the gate-level netlists using SIS [25] (*script.rugged*) and Rasp [5] ($rasp\_syn$) and finally technology mapped the circuits to 4-LUTs using IMap.

Before applying our resynthesis to these circuits, we needed to select a set of optimal resynthesis structures. Because the number of 4-LUT resynthesis configurations is exponentially proportional to the number of inputs to the subcircuit, we restricted the size of our resynthesis structures to 10-inputs or less. This restricted the resynthesis structures to those shown in Fig. 38. Fig. 38a is applied for cones with a fanin size of seven or less and containing



(a) 7-input (b) 10-input #1 (c) 10-input #2

Fig. 38. Resynthesis Structures

more than two 4-LUTs; Fig. 38b and 38c are applied for cones with a fanin size of 10 or less and containing more than three 4-LUTs. Note that Fig. 38a and b are the optimal configuration for subcircuits containing 7 and 10 inputs respectively.

Once we chose our resynthesis structures, we resynthesized the 4-LUT netlists with our sliding window approach. The results are shown in Table IX. *Resynth* is the 4-LUT counts after our resynthesis algorithm is applied and *IMap* are the 4-LUT counts after IMap technology mapped the gate-level netlists. *Ratio* is the ratio $\frac{Resynth}{IMap}$.

TABLE IX
DIGITAL LOGIC BLOCK RESYNTHESIS RESULTS.

Building Block	*Resynth*	*IMap*	*Ratio*
`4:1 MUX`	2	3	0.67
`16:1 MUX`	13	17	0.76
`32-Bit Priority Encoder`	69	85	0.81
`4-Bit Barrel Shifter`	8	12	0.67
`16-Bit Barrel Shifter`	32	48	0.67
`6-Bit Set Reset Checker`	2	3	0.67
`2-Bit Sum Compare Const`	2	6	0.33
`2-Bit Sum Compare`	2	3	0.67
`6-Bit Priority Checker`	3	6	0.50
`8-Bit Bus Multiplexer`	16	24	0.67
Total	152	207	0.73

It is interesting to see the large reduction in area that we achieved for these common digital logic blocks. This trend proved to be common for highly non-disjoint functions where gate-level synthesis often create decompositions that technology mappers have difficulty working with. This is most easily understood through an example.

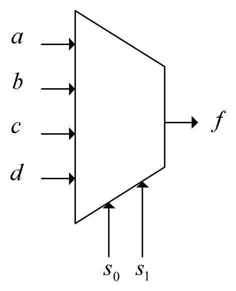


Fig. 39. Symbolic representation of a 4:1 MUX with 4 inputs and 2 selector inputs.

Consider the 4:1 MUX shown in Fig. 39. Decomposition into basic gates by gate-level synthesis using the SIS [25] synthesis package developed at UC Berkeley yields a solution shown in Fig. 40 (SIS actually returns an `AND-OR` netlist, which we converted to `NAND`-gates for simplicity). As this figure shows, this is simply a tree of three 2:1 MUXes implemented with `NAND`-gates. The cost metric driving SIS is the number of literals and gates produced in the final solution. Thus, for SIS, this is its best solution in which 11 gates are needed.

When applying technology mapping to the optimized circuit using IMap, the solution returned is shown in Fig. 41 which uses three 4-LUTs (one input per LUT is not used).

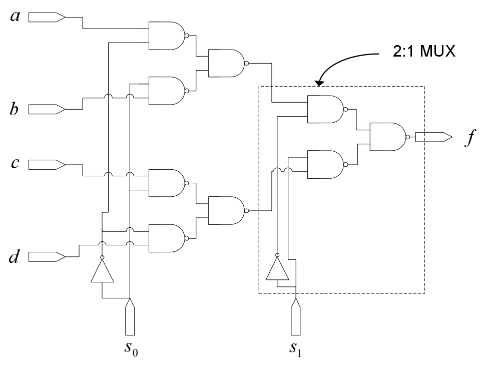


Fig. 40. Gate-level synthesis solution produced by SIS [25].

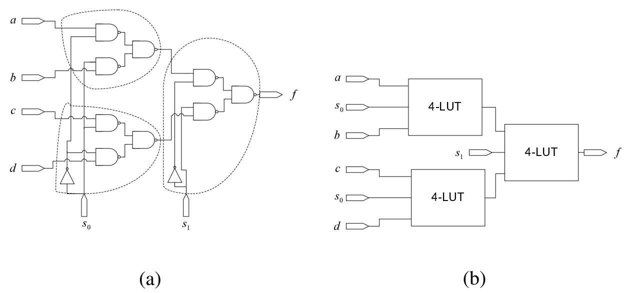


Fig. 41. 4-LUT solution produced by the IMap technology mapper. (a) illustrates the initial covering produced and (b) illustrates the final 4-LUT netlist.

The final solution shown in Fig. 41 was used in industry to implement a 4:1 MUX for several years until it was realized only two 4-LUTs [20] were required. Our results in Table IX confirm this and also produce a two 4-LUT solution for a 4:1 MUX. This solution can only be uncovered by current technology mappers if redundant logic is added to the gate-level netlist. The addition of redundant logic is very common in technology mappers [8], and surprisingly can reduce the final area of the circuit significantly. However, this is often limited to simple operations such as duplication of nodes. In order to allow IMap to uncover the optimal 4-LUT configuration of a 4:1 MUX, an entire extra multiplexer is added to the circuit and the circuit is restructured. This is shown in Fig. 42 where a 2:1 MUX has been added and s_0 has been rewired and replaced by the subcircuit feeding **x**.

This optimization is possible due to Observability Don't Cares (ODCs) as explained in Fig. 43. In Fig. 43a, $s_1 = 1$; hence, all outputs of the gates highlighted in black can be ignored due to ODCs. Furthermore, in this state gates highlighted in grey become functionally equivalent to a buffer. This allows the redundant MUX to select s_0 and pass it to the lower portion of the circuit. In Fig. 43b, $s_1 = 0$. Again, all gate outputs which can be ignored are highlighted in black and gates highlighted in grey are functionally equivalent to a buffer. Thus, both of these circuits are equivalent to the simplified SIS circuit shown in Fig. 40.

Adding the 2:1 MUX increase the cost of the circuit from 11 gates to 14 gates. Thus during gate-level synthesis,

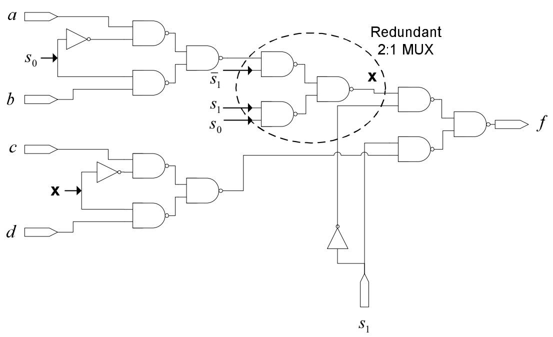
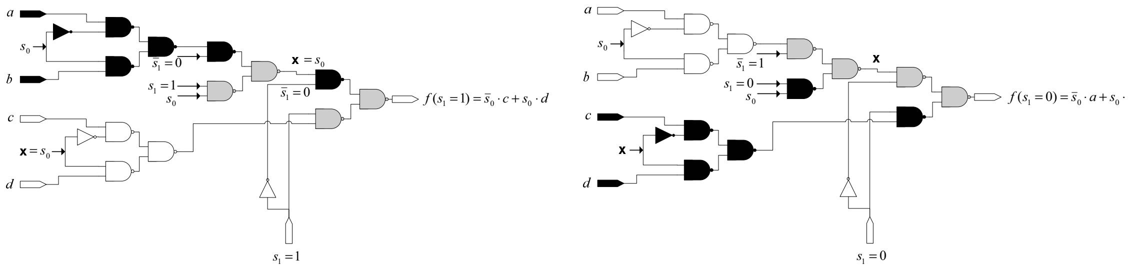


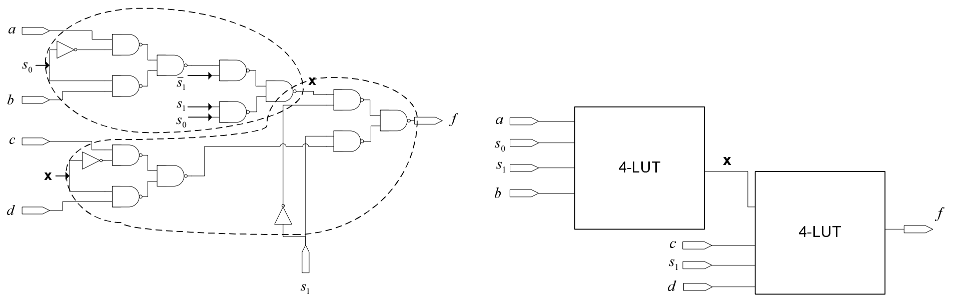
Fig. 42. Redundant logic added and circuit restructuring to help optimize final technology mapped solution.



(a) (b)

Fig. 43. Exploitation of ODCs to rewire circuit.

adding these gates would appear wasteful and would be optimized away. However, as Fig. 44 shows, adding the redundant 2:1 MUX allows IMap to find the optimal two 4-LUT configuration. Thus, if gate-level synthesis was aware of technology mapping steps that were to follow it, this could drive synthesis to perform more intelligent optimization steps to help technology mapping.



(a) (b)

Fig. 44. Optimal 4-LUT solution produced by the IMap technology mapper. (a) illustrates the initial covering produced and (b) illustrates the final 4-LUT netlist.

TABLE X
AVERAGE RUNNING TIMES FOR A 9-INPUT PLB.

Fit (sec)	No Fit (sec)
12.3	35.3

TABLE XI
COMPARISON OF USING THE QUANTOR QBF SOLVER V.S. REMOVING QUANTIFIERS ON THE QBF AND USING CHAFF.

PLB size	QSAT-Quantor (sec)	SAT-Chaff (sec)
6-input	$\ll 0.01$	$\ll 0.01$
7-input	7.12	334
8-input	1.86	60.2

C. Running Time

For PLBs of input size 6 or less, there was a negligible cost of using SAT during technology mapping where the evaluation time for each cone was under 1 millisecond. However, once the PLBs grew beyond 6-inputs, the SAT running time increased significantly, particularly when the cones did not fit into the given PLB. To illustrate, Table X shows the average running time of 50 instances of attempting to fit an 8-input function into the 9-input PLB shown in Fig. 23(e) when running on a Sunblade 150 with 1.5 GB of RAM. The Fit column shows the running times in instances where the functions could fit into the given PLB and the No Fit column shows the running times in the instances where the functions could not fit into the given PLB.

The dramatic difference in running time between small PLBs ($\leq$ 6-inputs) and large PLBs ($>$ 6-inputs) can be attributed to the exponential relationship in the SAT CNF expression size with respect to the number of inputs. This is due to the replication process to remove universal quantifiers described in Section III-B.1. The original motivation of removing quantifiers was to simplify our problem. However, promising results obtained from the Quantor QBF solver imply that this may not be necessary. The Quantor [16] QBF solver was very successful in solving some hard PLB fit instances as shown in Table XI. This does not contradict our claim that the our conversion to SAT simplifies the problem, since the technique that Quantor uses to solve QBFs is very similar to our replication process, which they call “resolve and expand”. However, unlike our technique Quantor uses several advanced optimizations to further simplify the final replicated CNF expression. The benefit of these optimizations are shown in Table XI. The QSAT-Quantor column shows the Quantor QBF solver running time in seconds when attempting the fit a 6-input function, 7-input function, and an 8-input function; the SAT-Chaff column shows the running time in seconds when attempting to solve the same problem, but first converting the problem to SAT and solving it with the Chaff SAT solver. As the results show, for functions of 6-inputs or less, converting the problem to SAT seems to be equivalent to solving the problem with Quantor. This, however, dramatically changes for the large function sizes of 7-inputs or more where Quantor has a substantial speedup over the SAT conversion method.

VI. Conclusion and Future Work

In this work, we have shown a practical application of Boolean Satisfiability to the problem of cost reduction of FPGA-based circuit realizations. We have described two different methods: A generic PLB evaluation method and a resynthesis technique.

Our PLB evaluation method is based on a generic technology mapper that is capable of mapping a circuit description to any arbitrary logic block. This allows for the study of architectural features that can be used to reduce the overall area of the FPGA. This includes both architectural changes in the PLB and routing features. The automation of our method derives from the generality of the technology mapping algorithm. The FPGA architect simply needs to give our tool a description of the PLB architecture. This includes input pin restrictions along with the PLB description which may be as simple as a K-LUT or as complex as a PLB containing MUXes and/or cascade gates. Also, the MWT area estimate of each PLB needs to be provided. Once these descriptions are passed to our tool, the technology mapper will provide a mapped solution of the circuit and give the PLB usage along with the MWT area. During our study, we successfully technology mapped circuits to several PLB architectures with no specialized decomposition techniques, some with several constraints including input-pin permutability. In particular, we have shown a PLB architecture that is competitive with a 5-LUT architecture and that the use of an extra dedicated OR-AND cascade can lead to significant area reductions in many cases.

Our resynthesis technique reduces the number of K-LUTs required to implement subcircuits for LUT-based FPGA architectures. We have shown area reductions of up to 67% in some cases using these techniques with an average reduction or 27%.

An obvious extension to our work would involve the combination of the two applications discussed in this paper. Our future research directions also involve the development of QBF solvers to bring down the significant runtime penalty from using SAT-based techniques. The drop in runtime would allow us to explore larger resynthesis structures. This could be used to build an extremely large cache of optimal subcircuit configurations which could help perform subcircuit restructuring after technology mapping. In addition, a drop in runtime would allow for a real-time technology mapping engine capable of technology mapping to extremely large PLBs (more than 8 inputs). Large PLBs are becoming more common in commercial FPGAs, where some already have 8-input PLBs [2]. The Quantor solver gives promising results for running time. Tuning Quantor to solve the function mapping problem described previously could yield a substantial speedup. We also hope to provide a number of benchmarks that will help to drive the development of an efficient QBF solver such as Quantor.

Overall, we have presented an elegant approach to the PLB mapping problem and have shown some promising applications of this technique.

References

[1] G. G. Lemieux and D. M. Lewis, “Using sparse crossbars within LUT clusters,” in *FPGA*, 2001, pp. 59–68. [Online]. Available: citeseer.ist.psu.edu/lemieux01using.html

[2] Altera, “Component selector guide ver 14.0,” 2004.

[3] Xilinx, "Virtex-ii complete data sheet ver 3.3," 2004.

[4] Arrow Electronics. [Online]. Available: www.arrow.com

[5] J. Cong, J. Peck, and Y. Ding, "RASP: A general logic synthesis system for SRAM-based FPGAs," in *FPGA*, 1996, pp. 137–143. [Online]. Available: citeseer.ist.psu.edu/cong96rasp.html

[6] J. Cong, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs," 1994. [Online]. Available: citeseer.ist.psu.edu/cong94flowmap.html

[7] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Design Automation Conference*, 1993, pp. 213–218. [Online]. Available: citeseer.ist.psu.edu/cong94areadepth.html

[8] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: enabling a general and efficient fpga mapping solution," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 1999, pp. 29–35.

[9] I. Levin and R. Y. Pinter, "Realizing Expression Graphs using Table-Lookup FPGAs," in *Proceedings of the European Design Automation Conference*, 1993, pp. 306–311.

[10] A. H. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1319–1332, Nov. 1994.

[11] A.Kaviani and S. Brown, "The hybrid field programmable architecture," *IEEE Design and Test*, pp. 74–83, April–June 1999.

[12] 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*, vol. 20, pp. 1077–1090, Sept. 2001.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., R. M. Osgood, Jr., Ed. Cambridge, Massachusetts: The MIT Press, 2001.

[14] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time(preliminary report)," in *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*. ACM Press, 1973, pp. 1–9.

[15] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 1, pp. 4–15, Jan. 1992.

[16] A. Biere, "Resolve and expand," in *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, May 2004, pp. 10–13.

[17] A. C. Ling, D. P. Singh, and S. D. Brown, "Fpga plb evaluation using quantified boolean satisfiability," in *Field Programmable Logic and Applications*, 2005.

[18] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[19] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in lut-based fpga technology mapping," in *International Workshop on Logic and Synthesis (IWLS'04)*, 2004.

[20] P. Metzgen and D. Nancekievill, "Multiplexer restructuring for fpga implementation cost reduction," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM Press, 2005, pp. 421–426.

[21] S. Yang, "Logic synthesis and optimization benchmarks user guide version," 1991. [Online]. Available: citeseer.ist.psu.edu/yang91logic.html

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001. [Online]. Available: citeseer.ist.psu.edu/moskewicz01chaff.html

[23] D. Chai, J. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton, "MVSIS 2.0 Programmer's Manual, UC Berkeley," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 2003.

[24] Xilinx, "Spartan-iie 1.8v fpga family: Function description," 2003.

[25] E. M. Sentovich, K. J. Singh, C. M. L. Lavagno, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 1992. [Online]. Available: citeseer.ist.psu.edu/sentovich92sis.html

[26] R. K. Brayton and G. D. H. et al., "VIS: a system for verification and synthesis," in *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, 1996, pp. 428–432. [Online]. Available: citeseer.ist.psu.edu/brayton96vis.html