FPGA Programmable Logic Block Evaluation using Quantified Boolean Satisfiability

Andrew C. Ling, Deshanand P. Singh, and Stephen D. Brown,[*]

December 12, 2005

Abstract

This paper describes a novel Field Programmable Gate Array (FPGA) logic synthesis technique which determines if a logic function can be implemented in a given programmable circuit and describes how this problem can be formalized and solved using Quantified Boolean Satisfiability. This technique is general enough to be applied to any type of logic function and programmable circuit; thus, it has many applications to FPGAs. The application demonstrated in this paper is FPGA PLB evaluation where the results show that this tool allows radical new features of FPGA logic blocks to be evaluated in a rigorous scientific way.

[*]Manuscript received October 20, 2005; revised December 4, 2005.
[†]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.

1 Introduction

FPGAs are integrated circuits characterized by a regular array of clustered programmable logic blocks (PLBs) connected together by programmable interconnects as shown in Fig. 1. The tile shown in the figure can be thought as the basic building block of an FPGA and includes one cluster of PLBs along with the associated routing for that cluster. Clustering PLBs into regular groups has proven to improve the performance of FPGAs both in terms of area and speed [1].

An example of a PLB is shown in Fig. 2. The logic block is composed of a 4-input lookup table (4-LUT) that is capable of implementing any arbitrary Boolean function of 4 variables. The LUT is implemented with a set of $2^4 = 16$ static RAM (SRAM) bits that are programmed with the truth-table values for the function to be implemented. In general, many modern PLBs are based on the k-input lookup table (k-LUT) which contains 2^k SRAM bits. Although the k-input LUT is very flexible, it is usually beneficial to add dedicated non-programmable logic to the PLB such as adders and `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, because this reduces the flexibility of the PLB, optimal mapping of functions to these non-programmable components is difficult.

1.1 CAD for FPGAs

Fig. 3 depicts a simplified FPGA CAD flow. The input to the CAD flow is a circuit description, which is typically given with a schematic or a hardware description language (HDL). From this point, synthesis operations are performed, such as finite state machine extraction and implementing HDL operators (such as +). The circuit representation is converted into a netlist of basic logic gates. Technology independent optimizations then try to minimize the netlist area and/or optimize

the speed-performance. Technology mapping follows and attempts to convert the gate level netlist into a netlist of PLBs.

Clustering is a netlist partitioning step that identifies highly connected groups of PLBs. The placement step then assigns these groups to specific locations on the device. After placement, the routing step assigns wire segments and routing switches to implement all PLB-PLB connections in the netlist. Finally, the software creates a configuration bitstream used for programming the target device with the required configuration bits.

1.2 Motivation

The cost of implementing a circuit in an FPGA is directly proportional to the number of PLBs required to implement the functionality of the circuit. FPGAs are sold in a number of pre-fabricated sizes. Decreasing the number of PLBs may allow a circuit to be realized in a smaller FPGA. Typical pricing is roughly linear to the number of PLBs in the FPGA device [4].

The PLB architecture has a significant impact on the number of PLBs required to realize a particular circuit. Thus, clever PLB designs are necessary that capture the majority of the functions encountered in typical circuits. In this paper we will show how methods based on Quantified Boolean Satisfiability (QSAT) can be used to rapidly determine if a PLB architecture will achieve a high capture rate. We also show how to convert this problem into a simpler form and solve it using Boolean Satisfiability (SAT).

2 Background

2.1 Technology Mapping

The technology mapping step in the FPGA CAD flow converts a gate-level network consisting of primitive gates into the PLBs that are present in the target FPGA architecture. The goal of the technology mapping step is to reduce area, delay, or a combination thereof in the network of PLBs that is produced. In this work, delay is 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. Previous work showed that the depth-optimal mapping solution can be obtained in polynomial time using a dynamic programming procedure [5].

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. 4. Fig. 4a illustrates the initial gate level network, Fig. 4b illustrates a possible covering of the initial network using 4-LUTs, and Fig. 4c illustrates the LUT network 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 [6]. However, surprisingly, the controlled use of duplication can lead to further area savings [7]. In contrast to the depth minimization problem, the area minimization problem was shown to be NP-hard for LUTs of size four and greater [8]. Thus, heuristics are necessary to solve the area minimization problem.

Another way to look at technology mapping is as a cone selection problem. The subcircuits circled in Fig. 4b 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. 5 cannot implement a 3-input `OR` gate. Thus, to technology map to generic PLBs, one must discard all K-feasible cones which cannot map into the given PLB architecture. We will show how we successfully use QSAT to accomplish this in later sections.

Although more limited in functionality, PLBs offer speed, area, and power advantages over fully programmable K-LUTs. Furthermore, in general only a small subset of K-feasible cones will appear in most logic circuits. Thus, so long as a given PLB architecture captures most cones encountered in real circuits, it will be successful in implementing circuits. One way to determine a PLB's capture success is to extract a set of K-feasible cones from benchmark circuits and determine how many of these cones can fit into a given PLB where a high fit percentage is desired. Furthermore, resource usage is necessary to determine if the non-programmable components of a K-input PLB will add minimal overhead to the overall FPGA. We present two tools in this paper that do both these tasks.

2.2 Quantified Boolean Satisfiability

As stated in Sec. 1, the main contribution of this work is to examine the use of QSAT for use in PLB evaluation. QSAT is the problem of determining if a quantified Boolean formula (QBF), $F = Q_1 x_1 ... Q_n x_n f(x_1 ... x_n)$ where $Q_i \in \{\exists, \forall\}$, has an assignment to its variables, $x_1 ... x_n$, such that F evaluates to true. If so, F is said to be satisfiable, otherwise it is unsatisfiable. This is

analogous to the much simpler problem of Boolean Satisfiability (SAT) where SAT seeks a single assignment to a Boolean formula F such that F evaluates to true. Interestingly, SAT is a special case of QSAT where SAT deals with Boolean formulae without any universal quantifiers (variables in Boolean formulae without any quantifiers implicitly have a single existential quantifier bound to them). QSAT, however, may have universally quantified variables, and thus seeks all assignments to its universally quantified variables to satisfy a QBF. For example, consider the expressions in Equ. 1. The first expression shows a satisfiable Boolean formula with its associated satisfying assignment. In contrast, simply by adding quantifiers to it, the QBF shown in the second expression is unsatisfiable due to the universally quantified variable x_2.

$$\begin{aligned} & (x_1 + x_2) \cdot (\overline{x_1} + \overline{x_2}) \\ & \text{satisfiable} \rightarrow [x_1 = 0, x_2 = 1] \\ \exists x_1 \forall x_2 \quad & (x_1 + x_2) \cdot (\overline{x_1} + \overline{x_2}) \\ & \text{unsatisfiable} \rightarrow [x_1 = 0, x_2 = \{0, 1\}] \\ & \text{unsatisfiable} \rightarrow [x_1 = 1, x_2 = \{0, 1\}] \end{aligned} \tag{1}$$

For all practical purposes, QSAT only deals with QBFs in Conjunctive-Normal-Form (CNF, sometimes referred as a Product-of-Sums). A Boolean function is in CNF if it consists solely of a conjunction of clauses, where a clause is a disjunction of literals and a literal is any variable or its complement. Equ. 1 are examples of formulae in CNF. In CNF, the problem of QSAT can be rephrased to: Given a QBF, $F = Q_1 x_1 ... Q_n x_n f(x_1 ... x_n)$ where $Q_i \in \{\exists, \forall\}$, find an assignment to its variables, $x_1 ... x_n$, such that each clause in $f(x_1 ... x_n)$ has at least one literal that evaluates to true.

3 Quantified SAT Applied to PLB Evaluation

The goal of PLB evaluation is to determine how useful a new PLB architecture will be in implementing circuits. These measures are often in terms of area, speed, and power. When evaluating area, the PLB flexibility is a concern. Since a k-input PLB consist of non-programmable components, it loses flexibility in terms of what functions it can implement. Thus, the non-programmable components in the PLB will not always be used thereby having an area overhead. However, so long as the PLB is flexible enough to implement most functions, the area overhead will be minimized. A flexible k-input PLB can be characterized by how many k-input cones can fit into it where a high fit percentage is desired. The underlying question asked when determining this fit percentage is as follows: Given an n-variable Boolean function, $F_{function}(x_1, x_2, ..., x_n)$, does there exist a programmable configuration to a circuit, G, such that the output of the circuit will equal $F_{function}(x_1, x_2, ..., x_n)$ for all inputs? Previously, robust heuristics to answer this question fell into two categories: a specialized PLB is proposed and a customized mapping algorithm is implemented to map benchmark circuits using the proposed element [9]; specialized Boolean-matching techniques are developed to decompose a logic function in such a way so that it matches the structure of the proposed PLB [10]. Both of these techniques require a specific logic manipulation technique for each PLB which suffers a lack of generality. In our technique, however, a much more general approach is taken using a novel QSAT based approach.

3.1 Formalizing Function Fitting Problem

Assuming that a programmable circuit can be represented as a Boolean function $G_{circuit} = G(x_1..x_n, L_1..L_m, z_1..z_o)$ where $x_i, L_j, z_k, G_{circuit}$ represent the input signals, configuration bits, intermediate circuit signals, and output function of the circuit respectively, the problem of function mapping into programmable

logic can represented formally as a QBF as follows.

$$\exists L_1 ... L_m \forall x_1 ... x_n \exists z_1 ... z_o (G_{circuit} \equiv F_{function}) \tag{2}$$

A satisfying assignment to Equ. 2 implies that $F_{function}$ can be realized in the programmable circuit.

In order to derive Equ. 2, the proposition $(G_{circuit} \equiv F_{function})$ must be represented as a CNF Boolean formula. This can be done using a well known derivation technique that converts logic circuits into a characteristic function in CNF [11]. This characteristic function describes all valid inputs, output, configuration bits, and internal signal vectors for the configurable circuit. For example, consider the truth-table in Fig. 6. Its onset describes all input-output relations of an `AND` gate. To extract the characteristic function, F_{AND}, from the truth-table, any standard minimization technique can be used.

Deriving characteristic functions directly from the circuit input-output relation is only practical for primitive gates and logic blocks where the number of inputs and outputs is small. Fortunately, characteristic functions for larger circuits can be derived iteratively from the conjunction of its subcircuit characteristic functions. For example, consider the cascaded gates shown in Fig. 6. Notice the wire connecting the two gates is labeled with variable Z for CNF construction. The characteristic function of the cascaded circuit is simply the conjunction of the two `AND` gate characteristic functions with variable Z as the logical link between the two functions. The characteristic function, F_{CASCADE} evaluates to true if all wire signals are consistent. This includes the primary inputs and outputs as in F_{AND}, plus any intermediate wire signals (i.e. Z).

The previous conversion technique for the cascaded `AND` structure can be extended to much larger circuits such as PLBs. This creates a characteristic function, Ψ, dependent on variables $x_1, ..., x_n$,

$L_1, ..., L_m$, $z_1, ..., z_o$, and G which represent the inputs, programmable bits, intermediate wires, and output of the circuit respectively. Thus, the proposition ($G_{circuit} \equiv F_{function}$) can be formed by substituting all instances of the output variable G in Ψ by the expression representing $F_{function}$. This is shown in Equ. 3 where the notation $[G/F(x_1, ..., x_n)]$ indicates that all instances of G have been replaced by $F(x_1, ...x_n)$. Sections following this will use similar notation to represent the substitution operation.

$$\begin{aligned}[G_{circuit} \equiv F_{function}] &\equiv \Psi\,[G/F(x_1, ..., x_n)] \\ &\equiv \exists L_1 ... L_m \forall x_1 ... x_n \exists z_1 ... z_o \psi\,[G/F(x_1, ..., x_n)]\end{aligned} \tag{3}$$

3.2 Removing Quantified Variables

Although QSAT solvers have shown initial promising results, it is often still faster to solve a QBF by removing the universal quantifiers and converting it to a SAT problem [12]. Removing the universal quantifiers eliminates the need to find multiple SAT instances for all universally quantified variable assignments, thus saving time; however, in doing so, the size of the Boolean formula increases substantially. To remove the universal quantifiers in a QBF, F, its proposition, f, is replicated to explicitly enumerate all possible assignments of the universally quantified variables. These replicated formulae are then conjoined with the logical `AND` operator to form a Boolean function that can be solved with SAT. In our work, we chose to remove the universal quantifiers since it proved faster to use SAT for most of the PLBs we evaluated.

In order to give better understanding to the previously described ideas, an example is given. Assume that a 3-input function F needs to be implemented in the PLB shown previously in Fig. 5. In the following steps,F represents the function of the cone under consideration for mapping, $\boldsymbol{X_i}$ represents input vector $x_1x_2x_3 = i$, and $F_i = F(\boldsymbol{X_i})$.

Step 1: Create CNF for individual elements in programmable circuit.

$$\begin{aligned} G_{LUT} = (x_1 + x_2 + \overline{L_1} + z_1) \cdot (x_1 + x_2 + L_1 + \overline{z_1}) \cdot \\ (x_1 + \overline{x_2} + \overline{L_2} + z_1) \cdot (x_1 + \overline{x_2} + L_2 + \overline{z_1}) \cdot \\ (\overline{x_1} + x_2 + \overline{L_3} + z_1) \cdot (\overline{x_1} + x_2 + L_3 + \overline{z_1}) \cdot \\ (\overline{x_1} + \overline{x_2} + \overline{L_4} + z_1) \cdot (\overline{x_1} + \overline{x_2} + L_4 + \overline{z_1}) \end{aligned} \tag{4}$$

$$G_{MUX} = (L_5 + \overline{x_3} + z_2) \cdot (L_5 + x_3 + \overline{z_2}) \cdot (\overline{L_5} + z_2) \tag{5}$$

$$G_{AND} = (z_1 + \overline{G}) \cdot (z_2 + \overline{G}) \cdot (\overline{z_1} + \overline{z_2} + G) \tag{6}$$

Step 2: Formulate the programmable circuit CNF from equations 4, 5, and 6.

$$G_{circuit} = G_{LUT} \cdot G_{MUX} \cdot G_{AND} \tag{7}$$

Step 3: Replication of equation 7 to remove quantified variables. This formulates G_{Total} where a satisfiable assignment to G_{Total} implies F can be realized in the programmable circuit.

$$
\begin{aligned}
G_{Total} = \; & G_{circuit}[\boldsymbol{X}/\boldsymbol{X_0}, G/F_0, z_1/z_3, z_2/z_4]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_1}, G/F_1, z_1/z_5, z_2/z_6]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_2}, G/F_2, z_1/z_7, z_2/z_8]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_3}, G/F_3, z_1/z_9, z_2/z_{10}]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_4}, G/F_4, z_1/z_{11}, z_2/z_{12}]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_5}, G/F_5, z_1/z_{13}, z_2/z_{14}]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_6}, G/F_6, z_1/z_{15}, z_2/z_{16}]\cdot \\
& G_{circuit}[\boldsymbol{X}/\boldsymbol{X_7}, G/F_7, z_1/z_{17}, z_2/z_{18}]
\end{aligned}
\tag{8}
$$

Note that in equation 8, the configuration bits are represented by the same variables (L_{1-5}) in each $G_{circuit}(\boldsymbol{X_i}, f_i)$ instance, where as all other signals are unique variables in each instance. This ensures that only one configuration will exist for all entries of the truth table.

In the previous example, the pins on the programmable circuit in Fig. 5 are not permutable. Given the labeling convention in Fig. 5, the function $F = (x_1 + x_2) \cdot x_3$ can be implemented; however, the function $F = (x_1 + x_3) \cdot x_2$ cannot. There is no need for restricting the labeling of the input pins in this manner because most programmable circuits are able to route signals to any input pins. In order to model this flexibility, virtual multiplexers controlled by virtual configuration bits, V_p, are added at each input pin of the programmable circuit. Going back to the circuit shown in the last example, Fig. 7 illustrates the previous circuit with virtual multiplexers added at the input pins. Thus, if $F = (x_1 + x_3) \cdot x_2$ is to be mapped into this network then the virtual multiplexers would force x_1 and x_3 onto the first two pins of the circuit and x_2 to the third pin feeding the `AND` gate to

generate a satisfiable solution. In order to add the virtual multiplexers to the previous example, the virtual multiplexer characteristic functions need to be added in **Step 1**, then the process proceeds normally as previously shown.

3.3 Application to PLB Evaluation

In the previous section we showed how to derive a general function mapping technique using QSAT. We will show how we use this to evaluate PLB architectures through two tools as follows.

3.3.1 PLB Fit Percentage

The following 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 QSAT decision step to remove cones that do not fit in the given architecture as shown in lines 1 and 2 of the following algorithm. By recording the number of cones generated and discared, a fit percentage for various PLB architectures can be found.

1 $|X| \leftarrow$ GenerateCones()

2 $|Y| \leftarrow$ RemoveNoFitCones()

3 $FitPercent \leftarrow (|X| - |Y|)/|X|$

A version of the algorithm described in [7] 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.

This tool cannot be used to fully evaluate area usage. However, fit percentage provides insights into the components that may be beneficial in a PLB. To obtain a more comprehensive evaluation of area, one should evaluate the PLB usage needed to implement a set of benchmark circuits. Using this in conjunction with an area model for the PLB, a full picture of area usage can be found. Unfortunately, obtaining the PLB usage for circuits requires custom decomposition techniques for technology mapping to the PLB architecture. We overcome this obstacle by incorporating our function mapping technique into a K-LUT technology mapper to form a general K-input PLB mapper.

3.3.2 Technology Mapping Using QSAT

Our function mapping technique allows us to convert any K-LUT technology mapper into a K-input PLB technology mapper. As stated in Section 2.1, technology mapping to LUTs can be thought 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 which are guaranteed to fit into the PLB architecture.

```
1  GenerateCones()
2  RemoveNoFitCones()
3  for   i ← 1 upto MaxI
4        TraverseFwd()
5        TraverseBwd()
6  end for
7  ConesToPLBs()
```

We base our work on IMap [13], an iterative K-LUT technology mapping algorithm. For a detailed description of IMap please refer to [13], which shows that IMap produces amongst the best

area results of any known technology mapper. The basic framework for our technology mapper is presented in the previous algorithm. First, a call to GenerateCones generates a subset of most 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 QSAT as described in the Sec. 3.1. 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 [13]. 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 selection procedure and thus the set of cones selected.

During depth-oriented mapping, on the first mapping iteration, the cone with the lowest depth is selected. Depth is often correlated with delay of the circuit, thus minimizing the depth of the circuit often leads to a faster circuit after technology 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

$$depth(C_v) \leq ODepth - height(v). \tag{9}$$

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 9 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.

3.3.3 Generating k-Feasible Cones

A version of the algorithm described in [7] 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. In contrast to the original IMap algorithm which combined the cones in every possible way, in our work, the cone generation algorithm combines cones if they have no more $(k+e)$ inputs in total. As long as e was set to a sufficiently high number (2 in the experiments), this heuristic sped up the cone generation process without significantly impacting the quality of the

mapping solution.

4 Results

4.1 Validaty of Technique

To validate our evaluation process, we first evaluated a wide range of PLBs to show the generality of our technique. Secondly, we used our technology mapping algorithm to evaluate a very successful commercial FPGA architecture to see if our results are consistent with what is reported in industrial literature [14].

4.2 Evaluation of Various PLBs

To show the power of the PLB evaluation algorithm, several unrelated PLB architectures were evaluated. Fig. 8 shows the five different PLB architectures used for evaluation.

Circuit	a 5-input	b 5-input	c 8-input	d 9-input	e 9-input
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

Table 1: PLB fit results.

To evaluate the versatility of each PLB, a set of cones were extracted from a list of circuits taken from the MCNC benchmark suite [15] (approximately 1000 K-input cones per circuit, where K was the input size of the PLB). These cones were tested for PLB fitting using the Chaff [16] SAT solver. The circuits used were unrelated to generate a large set of dissimilar cones. Table 1 shows the PLB fit percentage of cones per circuit. The last row shows the total percentage of all cones that fit. 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 and `XOR` gates are very rare in most control circuits and are generally used for arithmetic logic.

Also, PLB (e) was only able to fit 9-input cones for a few circuits. This was expected since it 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 [17]. Thus, in addition to generating 9-input functions for the PLB (e), 6, 7, and 8-input functions were evaluated to get a broader picture of the PLB's flexibility. This is shown in Table 2. As the numbers show, this PLB looks much more useful when adding a wider range of functions.

As a second part of the evaluation, we want to give a full area estimate on the area overhead associated with a given PLB. For our experiments, we will focus on PLB(a). The 5-input function fit percentage for that PLB is quite high, but the area overhead incurred by the `AND`-gate is uncertain. The `AND`-gate configuration in PLB (a) is very common in commercial PLBs [14]. The claimed benefit of adding an `AND`-gate is it allows one to chain PLBs together to create an extremely fast network of PLBs [14] with insignificant area overhead. Most FPGA companies accomplish this by restricting the routing into the `AND`-gate to adjacent PLBs as shown in Fig. 9. This prevents conditions such

[1] no functions larger than 7-inputs could be found in this circuit
[2] no 8-input functions could be found for this circuit

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[2]
Total % Fit	14.7	6.41	3.50

Table 2: The percentage of cones that fit into Fig. 8(e).

as the AND-gate being fed by IO-pins.

To model the AND based PLB, we construct a PLB similar to what is shown in Fig. 10 (registers, SRAM bits, and other details common to a 4-LUT based architecture have been ommitted for simplicity). The architecture we are assuming is a clustered island style FPGA with 60 routing tracks per channel. Each cluster consists of 10 PLBs, 22 inputs feeding each cluster, 1 clock input, and 75% of cluster inputs routable to any given PLB input (de-populated [18]). The depopulation of inter and intra-cluster inputs have been shown to improve area substantially while having minimal effect on timing and routing in the FPGA [18]. Since FPGA area is known to be dominated by transistor area, we use minimum-width transistors as our area metric [19]. Using the characteristics derived previously, we obtain the minimum-width transistor usage for a cluster of PLB(a). We also used VPR [19] to derive the minimum-width transistor usage of an entire tile. Using both of these metrics, we compared against a 4-LUT based architecture as shown in Table 4.2. (counts have been derived from [19]). The row labeled ratio is the ratio of the minimum-width transistor count of PLB(a) against the 4-LUT. As Table 4.2 show, there is about 3% area overhead per cluster and 1%

area overhead per tile with adding a cascaded AND-gate as illustrated in Fig. 10.

		PLB(a)	4-LUT
cluster	count	6743	6523
	ratio	1.03	
tile	count	24597.5	24377.5
	ratio	1.01	

Table 3: Minimum-width transistor count comparisons.

Finally, to evaluate the effect of PLB(a) on depth and area, we technology mapped a large set of MCNC benchmark circuits using our SAT based technology mapper and compared the PLB resource usage against a 4-LUT based architecture. The results are shown in Table 4. Column 'Area' indicates the PLB(a) or 4-LUT usage needed to implement the associated circuit and column 'Depth' indicates the depth of the technology mapped circuit, where each edge is given a length of 1. The 'Total' row indicates the summation of the depths and area numbers, and the 'Ratio' is the ratio of the total numbers when compared against the PLB(a) column. The results clearly show that adding an AND-gate to the PLB reduces the depth and area of the circuit. Using the claim that the cascade based routing structure of PLB(a) is much faster than a general routing channel [14], combined with the reduced depth values shown in Table 4, circuits implemented with PLB(a) should perform faster than a simple 4-LUT architecture. Furthermore, because the area overhead of the transistors show less than a 1% increase and the average reduction in PLB count is larger than 1% (7.9%), we expect no area overhead associated with adding a cascade AND-gate. Thus, the claim that an AND-gate has insignificant area overhead is consistent with our findings using our general PLB technology mapper.

	PLB(a)		4-LUT	
Circuit	Area	Depth	Area	Depth
alu4	1003	7	1045	7
apex2	1173	8	1236	8
apex4	997	7	1131	6
bigkey	1145	4	1586	3
C6288	814	24	1023	26
clma	4689	15	5032	14
des	1128	6	1239	6
diffeq	904	13	1025	12
dsip	1367	4	1144	4
elliptic	2094	16	2239	16
ex1010	2180	8	2639	8
ex5p	983	6	991	7
frisc	2275	21	2397	21
i10	811	14	914	15
misex3	1117	6	1115	7
pdc	1829	10	2180	10
s38417	4228	10	4296	10
s38584.1	3963	10	4124	11
seq	1102	6	1120	7
spla	1271	8	1380	8
Total	35073	203	37856	206
Ratio	1.000	1.000	1.079	1.015

Table 4: Comparing PLB(a) area and depth numbers against a 4-LUT architecture.

4.3 Running Time

For PLBs of input size 6 or less, there was a negligable effect of using SAT during technology mapping where the evaluation time for each cone was under 1 milisecond. 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. For illustration, we fount the average running time of 50 instances of attempting to fit an 8-input function into the 9-input PLB shown in Fig. 8(e) when running on a Sunblade 150 with 1.5 GB of RAM. For instances when the 8-input function could fit PLB(e), the problem took an average running time of 12.3 seconds while the no fit cases took an average running time of 35.3 seconds.

The dramatic difference in running time between PLBs with 6 or less inputs and PLBs with more than 6 inputs can be attributed to the exponential relationship in the SAT CNF expression size with respect to the number of inputs due to the replication process to remove universal quantifiers as described in Sec. 3.2. Fortunately, there has been some promising numbers from new QSAT solvers, thus the replication process to remove universal quantifiers from our original QBF expression may not be necessary. In particular, the QBF solver known as Quantor [20] was very successful in solving some hard PLB fit instances as shown in Table 5. The QSAT-Quantor column shows the Quantor QSAT solver running time in seconds when attempting the fit various functions of k-inputs. Specifically, the 7-input shows results for when attempting to fit a 7-input function into a 7-input PLB and the 8-input shows results for when attempting to fit an 8-input function into an 8-input PLB; the SAT-Chaff column shows the running time in seconds when attempting to solve the same problem, but first removing quantifiers and solving the problem with the Chaff SAT solver. As Table 5 show, for small functions of less than 6-inputs, the difference between the Quantor and Chaff is negligable, however, Quantor clearly show benefits for the larger fit instances with 7-inputs

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

Table 5: Comparison of using the Quantor QBF solver v.s. removing quantifiers on the QBF and using Chaff.

or more. These gains were specific to the Quantor QBF solver, and was not seen on some other popular QBF solvers such as Quaffle [21]. Quantor uses a similar replication process to our approach in order to remove universal quantifiers. However, unlike our approach, Quantor applies advanced optimizations to the replicated CNF before solving the QBF. These optimizations are suspected to cause the performance gains seen by Quantor.

5 Conclusion and Future Work

This work represents only the first step in using QSAT in FPGA CAD tools. Our research will progress to speed up the function mapping solver. We hope to use the Quantor QSAT solver in our PLB evaluation process. Initial numbers suggest that for large PLBs with 7-inputs or more, a dramatic speedup is expected.

Dramatic speedups in the running time for the function mapping problem will definitely open new avenues for our function mapping technique such as resynthesis. Furthermore, by finding more applications for QSAT, we hope this will help drive research for more sophisticated QSAT solvers.

References

[1] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," in *FPGA*, 2000, pp. 3–12. [Online]. Available:

citeseer.ist.psu.edu/ahmed00effect.html

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

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

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

[5] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 1, pp. 1–13, Jan. 1994.

[6] ——, "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

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

[8] 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.

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

[10] J. Cong and Y.-Y. Hwang, "Boolean matching for lut-based logic blocks with applications to architecture evaluation and technology mapping," *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 9, pp. 1077–1090, 2001.

[11] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiablity," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 6–22, 1992. [Online]. Available: citeseer.ist.psu.edu/larrabee92test.html

[12] D. Tang, Y. Yu, D. P. Ranjan, and S. Malik, "Analysis of search based algorithms for satisfiability of quantified boolean formulas arising from circuit state space diameter problems," in *SAT '04: The Seventh International Conference on Theory and Applications of Satisfiability Testing*, May 2004, pp. 10–13.

[13] 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.

[14] Altera Corporation, *APEX 20K Data Sheet*, Mar. 2004.

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

[16] 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

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

[18] 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

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

[20] A. Biere, "Resolve and expand," in *7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.

[21] L. Zhang and S. Malik, "Conflict driven learning in a quantified boolean satisfiability solver," in *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design.* ACM Press, 2002, pp. 442–449.

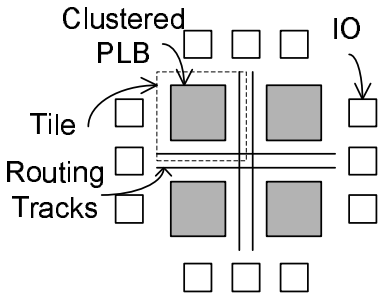


Figure 1: A generic island-style FPGA.

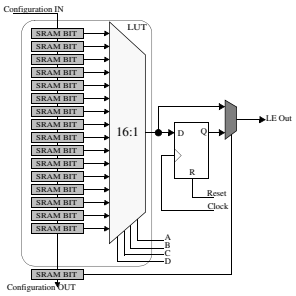


Figure 2: A simplified FPGA PLB.

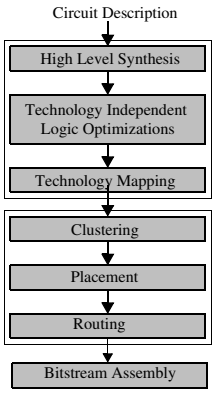
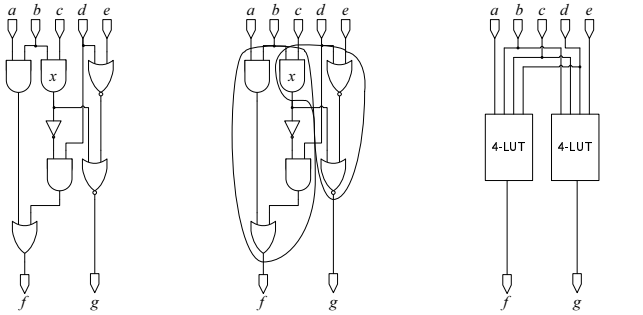


Figure 3: FPGA CAD Flow.



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

Figure 4: Technology mapping as a covering problem.

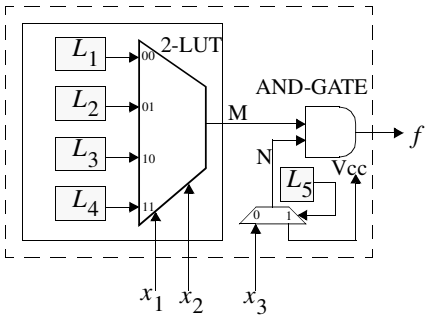
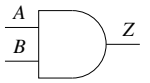
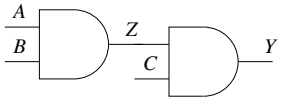


Figure 5: Example PLB.

A	B	Z	F_{AND}
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



$$F_{\text{AND}} = (A + \overline{Z}) \cdot (B + \overline{Z}) \cdot (\overline{A} + \overline{B} + Z)$$



$$F_{\text{CASCADE}} = (A + \overline{Z}) \cdot (B + \overline{Z}) \cdot (\overline{A} + \overline{B} + Z) \cdot (Z + \overline{Y}) \cdot (C + \overline{Y}) \cdot (\overline{Z} + \overline{C} + Y)$$

Figure 6: Deriving a circuit characteristic function.

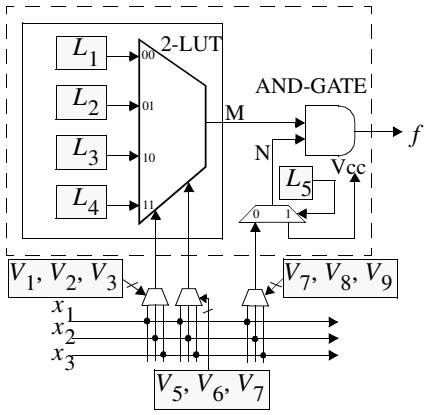


Figure 7: An example PLB with virtual multiplexers added.

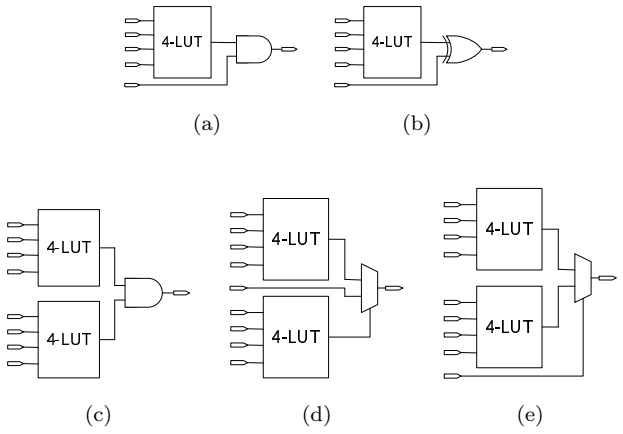


Figure 8: PLB architectures.

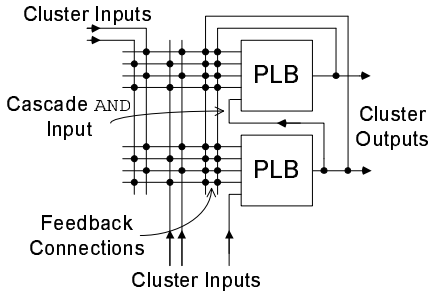


Figure 9: Illustration of routing restrictions for the cascade `AND` input.

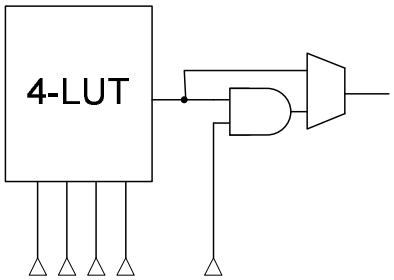


Figure 10: Simplified model of cascade `AND`-gate PLB.