Improved SAT-Based Boolean Matching Using Implicants for LUT-Based FPGAs

Jason Cong and Kirill Minkovich
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA

{cong, cory_m}@cs.ucla.edu

ABSTRACT

Boolean matching (BM) is a widely used technique in FPGA resynthesis and architecture evaluation. In this paper we present several improvements to the recently proposed SAT-based Boolean matching formulation (SAT-BM-M) [10]. The principle improvement was achieved by deriving the SAT formulation using the implicant instead of minterm representation of the function to be matched which enables our BM formulation creates a SAT problem of size $O(m \cdot 2^k)$ as opposed to $O(2^n)$ in the original formulation, where n is the number of inputs to the function, k is the size of the LUT, and m is the number of implicants which is much smaller than 2^n and experimentally found to be around $3 \cdot n$. Using the new BM formulation, and considering 10-input functions, we can show a 14x run time improvement and can solve 4.6x more problem than the SAT-based BM formulation in [10]. Moreover, a relative improvement of 4.1x in run time and 2.6x in the number of problems solved is observed when compared to the recently proposed BM formulation in [16]. Moreover, using this improved Boolean matching formulation, we implemented (as a proof of concept) a FPGA resynthesis tool, called RIMatch, which was able to reduce the number of LUTs produced by ZMap by 10% on the MCNC benchmarks.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) have been gaining momentum as an alternative to application-specific integrated circuits (ASICs). FPGAs consist of programmable logic, I/O, and routing elements which can be programmed and reprogrammed in the field to customize an FPGA, enabling it to implement a given application in a matter of seconds or milliseconds. The most common type of programmable logic element used in an FPGA is called a *K*-LUT, which is a *K*-input 1-output lookup table (LUT), capable of implementing any *K*-input 1-output Boolean function. *K*-LUTs are commonly implemented as a 2^K truth table on the LUT's inputs. The number of LUTs required to implement a design defines the cost of the FPGA implementation, thus making the technology mapping step of the FPGA CAD process a crucial step in cost reduction. In general, the goal of technology mapping is to reduce delay, area or a combination of the two in the resulting design [2][5][14][18]. The goal of area-minimal technology mapping is to minimize the number of LUTs needed to cover all the logical elements in the circuit.

One possible approach to the FPGA technology mapping problem is to use Boolean matching (e.g. [1], [4]). The Boolean matching problem consists of checking to see whether a Boolean function *f* can be implemented using a specific logic block (in our case this logic block consists of 4-LUTs). Boolean matching is not only used in technology mapping but also very useful for evaluating different programmable logic block (PLB) architectures (e.g. with several hard-wired LUTs) to see what percentage of Boolean functions can be implemented by a given PLB architecture [11].

Current Boolean matching techniques are effective when targeting functions with a few inputs (typically seven or less), they have a huge blow up in time or space when dealing with functions that have ten or more inputs. The reason for this is that Boolean matching is primarily done by either functional decomposition [4][9][12] or by using Boolean signatures [1][6][9], which are both limited in the size of the functions they can handle. Most functional decomposition is carried out using BDDs, which may require exhaustive search of all or many possible BDD variable orders. Recently, a new approach for Boolean matching using SAT was presented in [10]. This approach traded space complexity for time complexity and was shown to work on 10-input functions, but it had a very slow run time.

To understand how Boolean Satisfiability (SAT) can be used to solve the Boolean matching problem, let us first consider a concrete example of Boolean matching where we want to see if function *f* (with three inputs) can be implemented using two 2-LUTs (Figure 1). By this transformation, we can show that function *f* can be implemented using two 2-LUTs if and only if we can find a true assignment to the SAT formulation specified by (1.1) to (1.3) with variables C_1-C_{14} (Figure 1) (to explained further in Section 3). SAT takes as input a Boolean expression in Conjunctive-Normal-Form (conjunction of clauses where each clause is a disjunction of literals) and seeks an assignment to the literals that sets at least one literal in each clause to *true*. The drawback of formulating the problem like this is that SAT is an NP-hard problem; however, in recent years there have been significant advances in SAT solvers (e.g. [7][15][19]) which allow SAT instances with thousands of variables to be solved in a matter of seconds.

Nevertheless, the Boolean matching method proposed in [10] (named SAT-BM-M in this paper) can be very slow for large functions (say over ten inputs) as it needs to replicate the clauses exponential number of times in terms of the number of input variables. For example, when SAT-BM-M was used to test the optimality of ZMap [5] by repeated resynthesis of 10-input cones, it ran for over two days on some of the MCNC benchmarks. This reduction in run time also allows this technique to be used on FPGA architecture evaluation [4].

In this paper, we present several algorithmic enhancement to the SAT-based Boolean Matching (BM) formulation presented in [10] and achieved drastic improvement of the runtime performance. The primary contributions of this work include:

1. We developed a novel approach to derive the SAT formulation for the BM problem using implicant representation that is not exponential in terms of the size of the function (Section 4.1). This new approach creates a SAT instance that is of size $O(m \cdot 2^k)$, compared to the original instance of size $O(2^n)$, where n is the number of inputs to the function, k is the size of the LUTs, m is the number of minterms used to describe the function (approximately 30 for a 10 input function).
2. We developed two additional enhancements – one is the MUX choice reduction in the SAT-based BM formulation (Section 4.2) and other is the reduction of the number of test structures (Section 4.3) to be considered for BM. As a result, our SAT formulation that enable us to use up to 6.4 times fewer variables, achieve 12X speedup, and solve 77% more problems than the original approach [10]. These combined improvements allowed us to apply Boolean matching on functions of up to 13 variables.
3. We developed (to confirm the extraordinary performance gain) a resynthesis algorithm called RIMatch (Resynthesis using Implicant Matching), based on our enhanced BM formulation, which is able to reduce the mapping solution produced by ZMap on the MCNC benchmarks by 10% with an average run time of a little over an hour.

2. DEFINITIONS

In this paper, we study the following Boolean matching problem: given a n-input Boolean function $f(i_0, i_1, \ldots, in)$, can it be implemented in l or fewer K-LUTs. Before any in-depth discussion of Boolean matching, we first define the different ways of representing a Boolean function. The two most popular representations of a Boolean are based on the use of minterms or implicants.

A *minterm* is an assignment from {0,1} to the inputs of a Boolean function to get an output value of 1 or 0, respectively. An example of the minterms of an AND gate can be seen in Figure 3. A function can be defined using a truth table which is simply the set of all its 2^n minterms.

A product term P (a conjunction of variables) is an *implicant* of the Boolean function F if P implies F. For example, when we consider Figure 4, the first implicant (the first row) tells us that if the first input is a 0, then the function will output a 0 independent of the other two inputs. In this paper we shall use the term implicant to refer to prime implicant. A *prime implicant* of F is defined to be an implicant that is minimal in terms of the number of literals - that is, if the removal of any literal from P results in a non-implicant for F.

Even though a function can be defined using a set of minterms (a truth table) or a set of implicants, it should be fairly clear that using implicants is a much more compact representation. We will experimentally show that for 10-input functions the number of implicants is approximately 30, while their corresponding truth tables have 1024 minterms.

3. REVIEW OF SAT-BM-M (SAT-BASED BM USING MINTERMS)

In order to check whether a function f can be implemented in n K-LUT, the method in [10] construct an SAT formulation in the conjunctive normal form (CNF) and then asking a SAT solver whether this is possible. To do this, a CNF representation of a given LUT structure must first be made with universal qualification, then duplicated for each minterm in the function's truth table before it is passed on to the SAT solver. Therefore, the CNF formula had to be duplicated 2^n (the number of minterms in a n-input function) times. This caused the original Boolean matching algorithm to generate a CNF that was exponential to the size of the input.

To illustrate how to create the CNF used in the original Boolean matching algorithm (SAT-BM-M) [10] we shall consider constructing the CNF G_{BM} which is needed for testing whether or not a 3-input Boolean function can be implemented by the two 2-LUT structure seen in Figure 1.

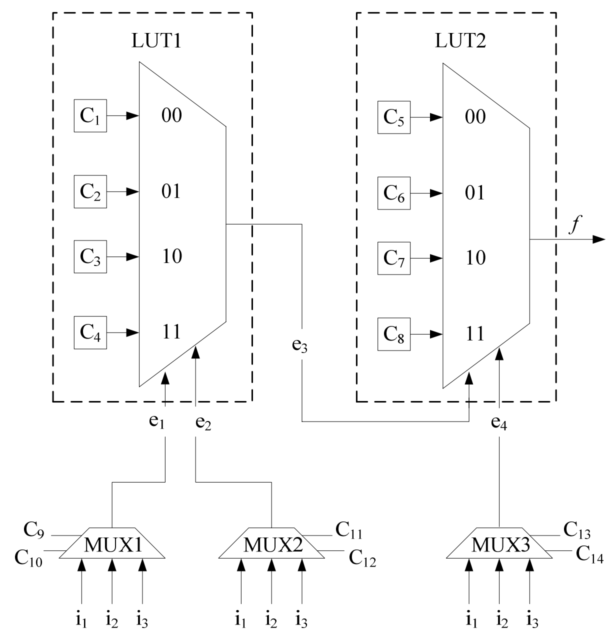

Figure 1. Test configuration using two 2-LUTs.

To create the CNF G_{BM} we first need to create a CNF equation for each of the individual elements in Figure 1; then we combine them to get a CNF for the whole structure.

The CNF, $G_{LUT}(\vec{x}, \vec{C}, f)$, for LUT2 has inputs $e_1 e_2 = \vec{x}$, configuration bits $C_1C_2C_3C_4 = \vec{C}$, and output $e_3 = f$.

$$\begin{aligned} G_{LUT} = {} & (e_1 + e_2 + \overline{C_1} + e_3)\cdot(e_1 + e_2 + C_1 + \overline{e_3})\cdot \\ & (e_1 + \overline{e_2} + \overline{C_2} + e_3)\cdot(e_1 + \overline{e_2} + C_2 + \overline{e_3})\cdot \\ & (\overline{e_1} + e_2 + \overline{C_3} + e_3)\cdot(\overline{e_1} + e_2 + C_3 + \overline{e_3})\cdot \\ & (\overline{e_1} + \overline{e_2} + \overline{C_4} + e_3)\cdot(\overline{e_1} + \overline{e_2} + C_4 + \overline{e_3}) \end{aligned} \quad (1.1)$$

The CNF, $G_{MUX}(\vec{x}, \vec{C}, f)$, for MUX1 has inputs $i_1 i_2 i_3 = \vec{x}$, configuration bits $C_9 C_{10} = \vec{C}$, and output $e_1 = f$.

$$
\begin{aligned}
G_{MUX} = & (C_9 + C_{10} + \overline{i_1} + e_1)\cdot(C_9 + C_{10} + i_1 + \overline{e_1})\cdot \\
& (C_9 + \overline{C_{10}} + \overline{i_2} + e_1)\cdot(C_9 + \overline{C_{10}} + i_2 + \overline{e_1})\cdot \\
& (\overline{C_9} + C_{10} + \overline{i_3} + e_1)\cdot(\overline{C_9} + C_{10} + i_3 + \overline{e_1})\cdot \\
& (\overline{C_9} + \overline{C_{10}})
\end{aligned}
\quad (1.2)
$$

Now combining equations (1.1) and (1.2) according to Figure 1, we get the following CNF with inputs $i_1i_2i_3$, and output *f*.

$$
\begin{aligned}
G_{BM}(i_1i_2i_3, f) = & G_{MUX}(i_1i_2i_3, C_9C_{10}, e_1)\cdot \\
& G_{MUX}(i_1i_2i_3, C_{11}C_{12}, e_2)\cdot \\
& G_{LUT}(e_1e_2, C_1C_2C_3C_4, e_3)\cdot \\
& G_{MUX}(i_1i_2i_3, C_{13}C_{14}, e_4)\cdot \\
& G_{LUT}(e_3e_4, C_5C_6C_7C_8, f)
\end{aligned}
\quad (1.3)
$$

According to the SAT-BM-M algorithm, in order to test whether function *f* could be implemented by the LUT configuration, G_{BM} would have to be duplicated for each of the possible 2^n inputs. This creates a CNF Φ (shown below), with only the configuration bits and the 2^n copies of the wires as variables, which can then be fed into a SAT solver.

$$
\Phi(f) = \prod_{k=0}^{2^{|\vec{i}|}-1} G_{BM}\left(\vec{i}_k, f\left(\vec{i}_k\right)\right)
$$

//creating new variables for the wires $e_1, \ldots, e_4$ (1.4)
// in each of the copies of G_{BM}

If the CNF problem is found to be not satisfiable, it means *f* cannot be implemented using the configuration of LUTs in Figure 1. If the problem is satisfiable, then the SAT solver provides a solution to the Boolean matching problem by returning satisfying assignments for all the configurations bits.

Observation: The original minterm formulation of the Boolean matching problem creates a SAT problem of size $O(2^n \cdot S)$, where *S* is the number of clauses needed to represent the structure.

Proof: Here, *S* is used for simplicity since we want to compare the two different formulations and both are somewhat dependent on *S*. Since there are 2^n minterms in the truth table representation of a Boolean function, we need to copy the CNF for the structure 2^n times as described in formula (1.4). The leads to an overall structure size on the order of $O(2^n \cdot S)$.

4. IMPROVEMENTS

In this section we will present an improved Boolean matching formulation using implicants. The actual speed-up of improvement shown by using this formulation, called SAT-BM-I, shall be described in the results section. Every improvement presented here will deal with reducing the complexity of the problem given to our SAT solver in terms of the number of variables and the number of clauses. In our final implementation we were able to get an answer from the SAT solver for most of our problems within four seconds.

4.1 Implicant vs. Minterm Formulation

The largest improvement over the original algorithm can be achieved only when we begin to consider the implicant representation of the functions versus their truth table representation. Here we are using don't cares (implicants) to simply a SAT problem which shouldn't be confused for [13] where they use SAT to calculate don't cares. The original method consumed exponential space and time in proportion to its input size, i.e., for a *n*-input function there are 2^n minterms, so the CNF for the test structure had to be copied 2^n times.

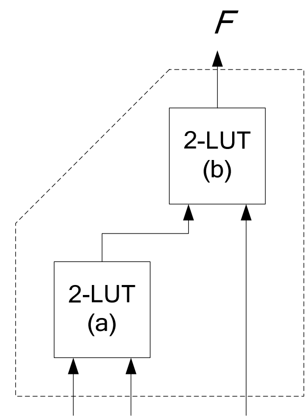

Figure 2. A 3-input function using two 2-LUTs.

To simplify the explanation, we will once again consider the problem of Boolean matching as it applies to a 3-input Boolean function and a 2-input LUT architecture. In this situation the question will be, can we find an assignment to configuration bits *C* such that the output *F* of the structure (Figure 2) has the same functionality as a function *f*; i.e., does $\exists C \text{ s.t. } \forall \vec{x}\ [f(\vec{x}) = F(\vec{x}, C)]$? Before we present the actual method, let us compare the number of implicants versus the number of minterms. For example, consider the Boolean 3-input AND function over three variables x_1, x_2, and x_3. The complete truth table representation of the function can be seen in Figure 3 which is exponential in the size of its inputs. Now, when we consider its equivalent representation using implicants (seen in Figure 4 and calculated internally in MVSIS [8]), we see that the number of lines/implicants that it takes to represent the function is cut in half. In general, the reduction is much more for functions with a larger number of inputs. From experimental data we were able to conclude that for n-input cones (where $7<n<11$), it takes roughly $3 \cdot n$ number of implicants to fully represent most functions, while the original method required 2^n.

AND gate			
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 3. Truth table of an AND gate.

AND gate			
x_1	x_2	x_3	f
0	*	*	0
*	0	*	0
*	*	0	0
1	1	1	1

Figure 4. AND gate representation using implicants.

The only thing left to consider is how to create a new CNF to handle the * in the implicant representation. Recall that the * represents the ability of that variable to be both 1 and 0. To take care of this we have to consider two cases: first, the internal LUT (one whose output feeds into another LUT, seen in Figure 5), and second, the output LUT (one whose output is the function output, seen in Figure 6). Consider Figure 7 for a graphical representation of how the two types of LUTs would connect together. When considering the internal LUT, the most difficult thing to model is the case where the output can become a * (its value can be either a 1 or a 0).

To take care of this case, we duplicate the LUT structure 2^k times where k is the number inputs to the LUT. This enables us to feed two versions of each variable into the LUTs (consider Figure 5 as an example). We follow the formula (1.5) for setting the variables. In Figure 5 if $x1$ is set to 0 or 1 then $x1'$ and $x1''$ are both set to 0 or 1. But if $x1$ is a * then $x1'$ is set to 0 while $x1''$ is set to 1.

$$x_i' = \begin{cases} 0 & \text{if } x_i = * \\ x_i & \text{else} \end{cases} \qquad x_i'' = \begin{cases} 1 & \text{if } x_i = * \\ x_i & \text{else} \end{cases} \tag{1.5}$$

Now the structure has the following property: if a certain input causes any of the LUTs to have different output values, f will be 0 and f' will be 1. But if all the LUTs have the same output value, f and f' will also have that same value.

While this modification enables the structure to take care of the * in the implicants, it also causes the structure to increase in size by 2^k. But if the number of implicants is greatly less than the number of minterms, this then reduces the complexity of the problem.

Observation: The implicant formulation of Boolean matching will create a SAT problem most of the time of size $O(m \cdot 2^k \cdot S)$, where S is the number of clauses needed to represent the structure, k is the size of the LUT, and m is the number of implicants in the function.

Proof: The new formulation needs to duplicate each LUT 2^k times; therefore, the total number of clauses needed to represent the structure is $O(2^k \cdot S)$. Since we only have to duplicate the structure m times, the total size of the SAT instance becomes $O(m \cdot 2^k \cdot S)$. ■

Theorem 1: The new CNF Φ is equivalent to the original CNF Φ^{orig}. That is

$$\Phi(f) = \prod_{k=0}^{|imp|} G_{BM}\left(imp_k, f\left(imp_k\right)\right) \equiv \Phi^{orig}(f) = \prod_{k=0}^{|min|} G_{BM}^{orig}\left(min_k, f\left(min_k\right)\right)$$

where imp is the set of implicants (product terms) of a sum-of-product representation of f, and min is the set of all minterms of f.

Proof: It is important to first remember that the CNF Φ is constructed using a set of G_{BM}, one for each implicant. Now if we were able to show that each G_{BM}, used to construct Φ, was equivalent to a set of G_{BM}^{orig} we could easily prove the equivalence by simple substitution since implicant representation can be expanded into minterm representation. Let us first assume we have an arbitrary function f. Now we arbitrarily pick a implicant imp from the set of implicants used to define f, so we want to show the resulting CNF $G_{BM}(imp, f(imp))$ implies the CNF $\prod_{k=0}^{|min|} G_{BM}^{orig}\left(min_k, f\left(min_k\right)\right)$ s.t. $imp = \vee_k min_k$. Since every implicant can be broken up into a set of minterms, we shall use this as an intermediate step.

$$G_{BM}\left(imp, f\left(imp\right)\right) = \prod_{k=0}^{|min|} G_{BM}\left(min_k, f\left(min_k\right)\right)$$

Now the question becomes does G_{BM} equal G_{BM}^{orig} when both are instantiated using a minterm. By examining the encoding (1.5) where all the LUTs that share common configuration will be equivalent we can see that G_{BM} does become G_{BM}^{orig} (consider Figure 7 as an example).

The other direction is trivial since a minterm can also be an implicant. Now by replacing every implicant generated G_{BM} by an equivalent set of minterm generated G_{BM}^{orig} we have transformed CNF Φ into CNF Φ^{orig} and showed their equivalence. ■

It was previously shown the original formulation, SAT-BM-M, created a SAT instance on the order of 2^n, while SAT-BM-I's were on the order of $m \cdot 2^k$. Experimentally, we have calculated that the average number of implicants needed to represent a Boolean function, m, is around $3 \cdot n$ or $O(n)$. Therefore, on average our problem size is on the order of $O(n \cdot 2^k \cdot S)$ which is significantly smaller than $O(2^n \cdot S)$ for large functions.

Let us examine the case where we are trying to test the feasible of implementing a 10-input cone using three 4-LUTs. In both formulations the variables in the SAT instance are the configuration bits and the wires, but the wires are determined by the configuration bits. Since the number of major decisions (i.e., the selection of the configuration bits) is the same in both formulations, the next thing that defines the complexity of the problem is the number of clauses. In the original formation the resulting CNF passed to SAT would be of size $2^{10} \cdot$ (size of structure) = $2^{10} \cdot S = 1024 \cdot S$, where S is the number of clauses needed to represent the structure's functionality. In our formula the size of the CNF would be around $(3 \cdot 10) \cdot 2^4 \cdot$ (size of structure) = $480 \cdot S$, and this number can go down to as little as $176 \cdot S$ (as in the case where you have an AND function on 10 variables, since that would only have 11 implicants). This shows that for larger cones, using implicants greatly simplifies the problem.

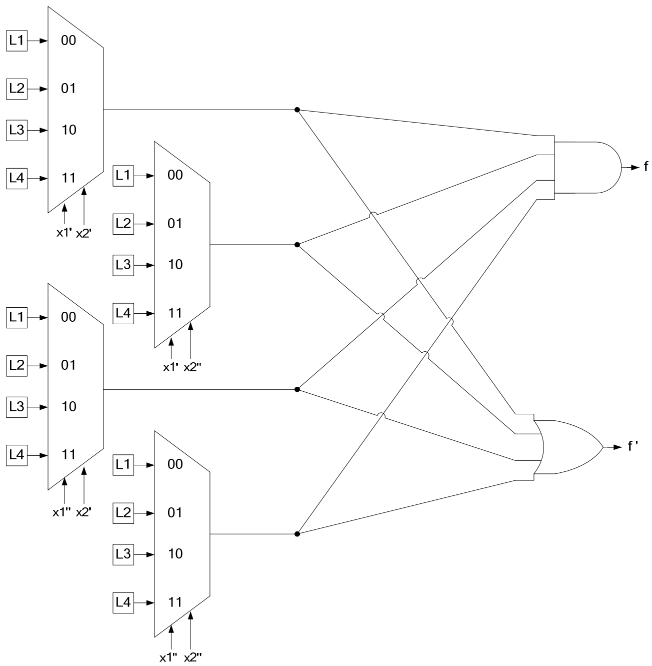

Figure 5. New internal LUT structure (Figure 2 LUT a).

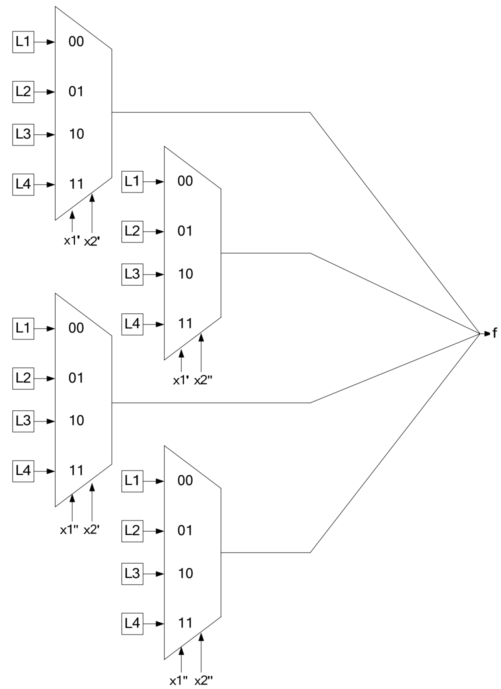

Figure 6. New output LUT structure (Figure 2 LUT b).

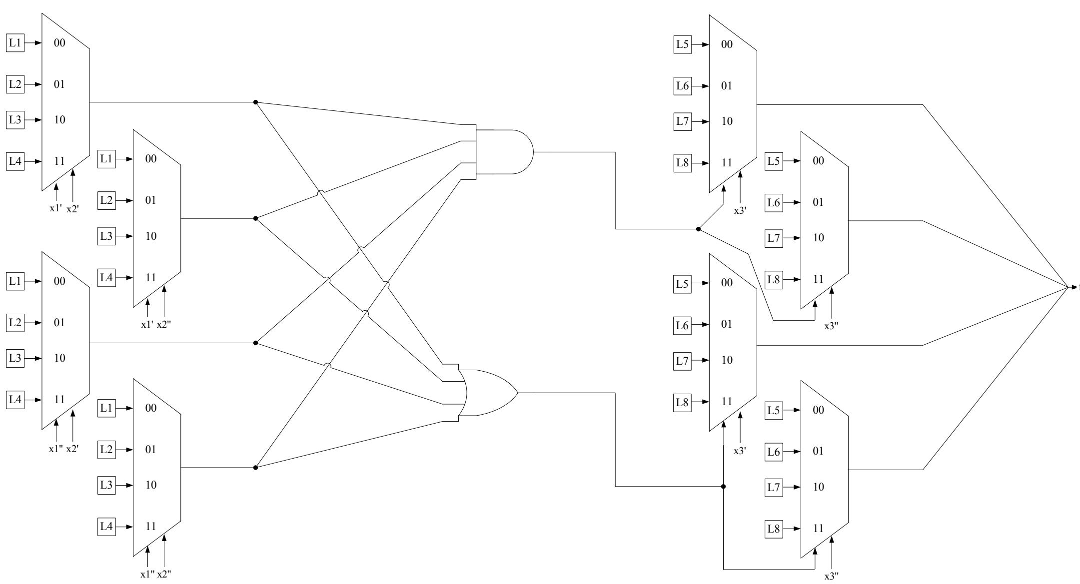

Figure 7. Detailed diagram a resynthesis structure for testing 3-input functions

4.2 MUX Choice Reduction

In the original formulation the LUT could accept any permutation of the inputs because every LUT had virtual multiplexers feeding its inputs. This allowed the resynthesis structure to increase the number of functions it could represent by a factor of $n!$, where n is the fanin size of the resynthesis cone. These “virtual multiplexers” added a great deal of unnecessary complexity to the problem. The number of distinct combinations to a single LUT is n choose k ($n!/(k!\cdot(n-k)!)$) not the $n!/(n-k)!$ combinations produced by the SAT-BM-M formulation. The way we reduce the number of combinations by $k!$ is by adding restrictions to the select signals of the MUXes corresponding to a single LUT. If we consider the structure proposed in Figure 1 and more specifically MUXes M1 and M2, we realize that by adding the restriction that M1 must select a signal with a lower index than M2, we can reduce the number of different possible inputs to the first LUT. This can be easily done by adding the following set of clauses to our CNF description (1.6). The reduction in exploration space is due to a reduction of symmetric configuration. This CNF can be easily generated by a multitude of tools available online for converting Boolean formulas into CNFs [17].

$(C_9 < C_{11}) + (C_9 = C_{11}) \cdot (C_{10} < C_{12})$

converting the + and =

$$= (C_9 \cdot \overline{C_{11}}) + ((C_9 \cdot C_{11}) + (\overline{C_9} \cdot \overline{C_{11}})) \cdot (C_{10} \cdot \overline{C_{12}}) \qquad (1.6)$$

after converting it to a CNF

$$= (\overline{C_9} + C_{11}) \cdot (\overline{C_9} + C_{12}) \cdot (\overline{C_9} + \overline{C_{10}}) \cdot (\overline{C_{10}} + C_{11}) \cdot (C_{11} + C_{12})$$

4.3 Reduction in Number of Test Structures used in Resynthesis

In the original formulation of test structures for resynthesis every connection pattern of LUTs required a different test structure. This might be acceptable for a testing a 10-input function, but as the number of inputs to the structure increases linearly, the number of structures grows exponentially. For example, when SAT-BM-M tested a 10-input function, we had to run two instances of SAT since there were two 10-input structures. These structures can be seen in Figure 8. When SAT-BM-M tries to test a 13-input function, it has to run SAT on four different CNFs.

There are two problems with this design. First, each 10-input cone has to be tested on two structures. Second, this doesn't extend to larger cones since a 13-input cone would have four separate resynthesis structures. The simplest and cleanest way to combine these two structures into one is by adding one MUX with one extra configuration bit. The resulting structure allows the SAT solver to pick which structure to use. An example of the new 10-input structure can be seen in Figure 9. This new method for creating resynthesis structures is easily extended to larger cones since it allows only one structure for any function size.

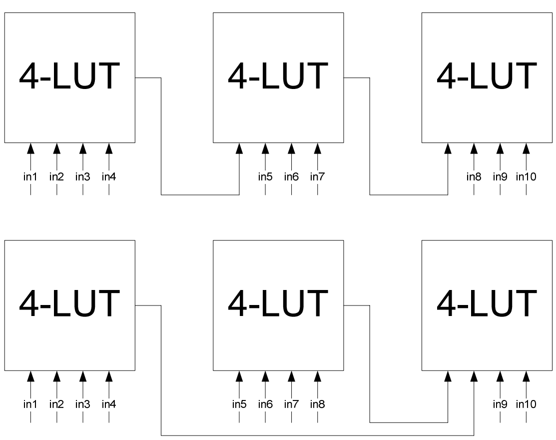

Figure 8. Original 10-input structures.

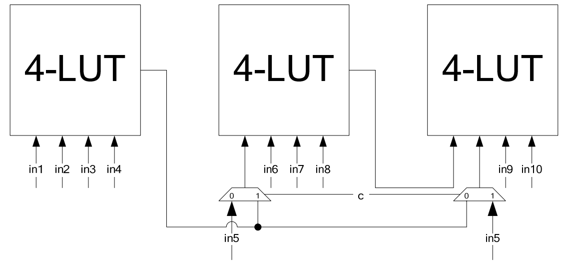

Figure 9. New 10-input structure.

5. RESULTS

The results will be presented in two sections. First we will compare our proposed method SAT-BM-I with SAT-BM-M [10] in terms of their SAT instance size: specifically, the number of variables, number of clauses and the run time. Second, we will use our Boolean matching formulation and apply it to resynthesis where we are able to reduce the area of ZMap (area-only minimizing mapper) by 10%.

5.1 Boolean Matching Improvement

In this section we present a quantitative comparison between the original Boolean matching algorithm (SAT-BM-M) [10] and our new implicant-based version (SAT-BM-I).

In our experiment, we selected first 220 eligible functions (i.e. those that either BM technique doesn't time out on) from the MCNC benchmarks and then ran them against both algorithms using ZChaff as the SAT solver. We only reported the results from eligible functions because functions that caused both methods to time out do not provide meaningful data for comparison. This method of selecting the functions was fair because neither the user nor the author has any influence on which functions were chosen. The data was gathered after running experiments on a 1.8GHz AMD Opteron processor. After evaluating functions with size varying from a 5- to 10-input, we saw that, on average, the new BM approach solved 77% more problem instances and ran 12 times faster. The most noticeable difference was seen when examining 10-input functions, where the new method solved 4.6 times as many problems and ran more than 14 times faster.

We can confirm the validity of our previous claim by examining the data in Table 1. First, when examining the details of SAT-BM-M we can see that it takes twice as many variables to test a function with one more input. This is because the problem size (as expressed by the number of clauses and variables) grows by a factor of two with each additional input variable. Second, we see that the size of the problem generated by SAT-BM-I only increases a very small amount between the functions of different size, as predicted by the theorem.

Table 1. SAT instance comparison

	function size	5	6	7	8	9	10	Total
	# functions examined	2	16	71	13	30	88	220
SAT-BM-M	# variables	309	608	1237	3393	6834	13411	4299
SAT-BM-M	# clauses	3111	7218	15838	53009	106878	211863	66319
SAT-BM-M	run time (s)	0.11	1.12	4.60	40.30	44.38	52.27	23.80
SAT-BM-M	% solved	100%	100%	94%	54%	43%	22%	56%
SAT-BM-I	# variables	307	505	497	906	847	949	668
SAT-BM-I	# clauses	7062	12206	11938	21989	20423	23147	16128
SAT-BM-I	run time (s)	0.02	0.29	0.55	5.28	1.92	3.70	1.96
SAT-BM-I	% solved	100%	100%	100%	100%	100%	100%	100%
Improvement	time reduction	5.50	3.86	8.41	7.63	23.17	14.12	12.15
Improvement	ratio solved	1.00	1.00	1.06	1.86	2.31	4.63	1.77

5.2 Resynthesis Algorithm using SAT-BM-I

We present the following algorithm not a resynthesis algorithm (even though the results suggest that it works fairly well) but as a usable proof of concept to our BM matching improvements. In our algorithm (written in the MVSIS [8] environment) we employed several techniques to speed up cut enumeration [14], and we also selected the cones based on a greedy approach from PO to PI. In this section we will present the results (Table 2) of running our resynthesis algorithm, RIMatch, on the mapped solutions produced by ZMap [5] on the MCNC benchmarks. Due to the speedup we were able to set a time limit of 4 seconds, twice the average run time (Table 1), for each SAT instance which enabled us to run our resynthesis algorithm two times on each one of the circuits. We used ZMap [5] because it is the best publicly available mapper that only minimizes area and it allowed us to give a direct comparison to the original resynthesis algorithm presented in [10] which also tested against ZMap. It should be noted that using this algorithm on ABC [18] and DAOmap [3] produced improvements of 6% and 8%, respectively, but in this paper we will not focus on these modern algorithms since their first goal is to optimize depth not area. Using the MCNC circuits that were initially mapped by ZMap, we were able to reduce the area by an average of 10%, which is 4% better than the results presented along with the original formulation [10]. RIMatch took about 20 hours to resynthesize every one of the MCNC benchmarks, but by running the resynthesis algorithm once over each circuit or by setting a smaller timeout, the total run time could be reduced to a few hours. Please note that this is a great improvement over the SAT-BM-M method [10] which would have taken almost two days to finish.

Table 2. Benchmark circuit resynthesis results.

Circuit	*Zmap*	*RIMatch*	*Ratio*
clma	5053	4602	0.91
b15_1	4272	3980	0.93
b15_1_opt	3854	3570	0.93
s38584.1	3822	3398	0.89
s38417	3593	2934	0.82
b14	3153	2622	0.83
frisc	2658	2415	0.91
pdc	1944	1808	0.93
misex3	1199	1121	0.93
seq	1191	1133	0.95
alu4	1140	1061	0.93
ex5p	1002	948	0.95
i10	794	741	0.93
Total	33675	30333	0.90

Since the code of the enhanced BM methods recently proposed in [16] is not available, we did not make a direct comparison with it. We did not attempt to re-implement the method in [16], as a lot of implementation details were left out on how the re-used conflict clauses were selected. Any attempt of simple-minded re-implementation would have not been fair to [16] since that is the main difference between that algorithm and the original one. Nevertheless, since the experimental results in [16] claimed a 340% run time improvement and 27% more success in mapping than the SAT-BM-M approach in [10], comparing to our results reported in this section, we conclude that our approach SAT-BM-I is 4.1x faster and solves 2.6x more problems when compared to [16]. It is interesting to note that our implicant-based method is orthogonal to the enhancement proposed in [16] by efficient the re-use of conflict clauses. So, we expect that our proposed techniques can be combined with that of [16] to get further improvement.

6. CONCLUSION

In this paper we presented several techniques that enable SAT-based BM to decide the optimal number of 4-LUTs needed to implement a function within a matter of seconds. Using our new implicant formulation (SAT-BM-I), when matching 10-input functions, we were able to get a 14 times better run time, and we were able to solve 4.6 times as many problems. Finally, we used SAT-BM-I to create an algorithm, RIMatch, to resynthesize a mapped circuit to reduce its overall area. Using the RIMatch algorithm, we were able to reduce the area of circuits produced by ZMap, on average, by 10%.

The binary for RIMatch, the Perl script to create the CNFs, and the CNFs for all the test structures can be downloaded at http://cadlab.cs.ucla.edu/~kirill.

7. ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under grants CCF-0306682 and CCF-0530261.

8. REFERENCES

[1] L. Benni and G. Micheli, “A Survey of Boolean Matching Techniques for Library Binding,” *ACM Trans. Design Automation of Electronic Systems*, Vol. 2, No. 3, pp. 193-226, July 1997.

[2] R. Brayton, S. Chatterjee, M. Ciesielski, and A. Mishchenko, “An Integrated Technology Mapping Environment,” *Proc. International Workshop on Logic and Synthesis*, pp. 383-390, 2005.

[3] D. Chen and J. Cong, “DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs,” *IEEE Transactions on Computer-Aided Design*, pp. 752-759, 2004.

[4] J. Cong and Y. Hwang, “Boolean Matching for LUT-Based Logic Blocks With Applications to Architecture Evaluation and Technology Mapping,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 20, No. 9, pp. 1077-1090, Sept. 2001.

[5] J. Cong, J. Peck, and Y. Ding, “RASP: A General Logic Synthesis System for SRAM-Based FPGAs,” *International Symposium on Field-Programmable Gate Arrays,* pp. 137-143, 1996.

[6] D. Debnath and T. Sasao, “Fast Boolean Matching Under Permutation Using Representative,” *Proc. Asia and South Pacific Design Automation Conference*, pp. 359-362, Jan 1999.

[7] N. Een and N. Sorensson, "Translating Pseudo-Boolean Constraints into SAT," JSAT 2006.

[8] M. Gao, J-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, T. Villa, and R. Brayton, “Optimization of Multi-Valued Multi-Level Networks,” *International Symposium on Multiple-Valued Logic*, May 2002.

[9] Y. Lai, S. Sastry, and M. Pedram, “Boolean matching using Binary Decision Diagrams with Applications to Logic Synthesis and Verification,” *Proc. International Conference on Computer Design*, pp. 452–458, Oct. 1992.

[10] A. Ling, D. Singh, and S. Brown, “FPGA Technology Mapping: A Study Of Optimality,” *Design Automation Conference*, pp. 427–432, 2005.

[11] A. Ling, D. Singh, and S. Brown. “FPGA PLB Evaluation using Quantified Boolean Satisfiability,” *International Conference on Field Programmable Logic and Applications*, 2005.

[12] F. Mailhot and G. De Micheli, “Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-12, No. 5, pp. 599–620, May 1993.

[13] A. Mishchenko and R. K. Brayton, “SAT-Based Complete Don't-Care Computation for Network Optimization,” *Design, Automation and Test in Europe Conference and Exposition*, pp. 412-417, 2005.

[14] A. Mishchenko, S. Chatterjee, and R. Brayton, “Improvements to Technology Mapping for LUT-Based FPGAs,” *International Symposium on Field-Programmable Gate Arrays*, Feb. 2006.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an Efficient SAT Solver,” *Design Automation Conference*, June 2001.

[16] S. Safarpour, A. Veneris, G. Baeckler and R. Yuan, “Efficient SAT-based Boolean Matching for FPGA Technology Mapping,” *Design Automation Conference*, July 2006.

[17] Artificial Intelligence: A Modern Approach Python Toolkit, http://aima.cs.berkeley.edu/python/logic.html

[18] Berkeley Logic Synthesis and Verification Group, “ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/~alanmi/abc/

[19] Boolean Satisfiability Research Group, “ZChaff,” http://www.princeton.edu/~chaff/zchaff.html