Accelerating Boolean Matching Using Bloom Filter

Chun ZHANG†, Member, Yu HU††, Lingli WANG‡‡, Lei HE†††(b), and Jiarong TONG†, Nonmembers

SUMMARY Boolean matching is a fundamental problem in FPGA synthesis, but existing Boolean matchers are not scalable to complex PLBs (programmable logic blocks) and large circuits. This paper proposes a filter-based Boolean matching method, F-BM, which accelerates Boolean matching using lookup tables implemented by Bloom filters storing pre-calculated matching results. To show the effectiveness of the proposed F-BM, a post-mapping re-synthesis minimizing area which employs Boolean matching as the kernel has been implemented. Tested on a broad selection of benchmarks, the re-synthesizer using F-BM is 80X faster with 0.5% more area, compared with the one using a SAT-based Boolean matcher.

Key words: FPGA, Boolean matching, Bloom filter, SAT, re-synthesis

1. Introduction

Boolean matching is a widely used technique in field-programmable gate array (FPGA) technology mapping [7], post-mapping re-synthesis [1] and architecture evaluation [2]. Ideally, an FPGA Boolean matcher should be scalable to large Boolean functions and complex PLB structures in terms of both runtime and memory, and be flexible to accommodate different PLB structures. Most existing Boolean matching algorithms are based on function decomposition [4], [8] or function canonical forms [5], [6].

- **Decomposition-based methods** try to split each Boolean function into smaller pieces, where each piece of function can be implemented by one component inside the PLB. However, such technique lacks flexibility as particular decomposition strategy needs to be customized for different PLB architectures. Depending on the function to be matched and the decomposition strategy employed, the run time efficiency varies greatly.

- **Canonical form-based methods** perform Boolean matching by computing and comparing canonical forms of different functions. A function’s canonical form is the unique representative of its belonging equivalence class, and only functions of the same equivalence class match with each other. In case of matching a Boolean function to the PLB, all implementable functions of that PLB should be considered as matching candidates. Due to the computation complexity of canonical forms, this technique can only handle functions with limited input size.

Recently, due to significant improvements of modern SAT solver [11], SAT-based Boolean matching (SAT-BM) [1] was proposed, which offers great flexibility in handling various PLB structures. However, even with numerous improvements [3], [7], [12], the expensive computational complexity of SAT-BM still limits its application to complex PLBs.

In this paper, we use Bloom filter [9] based lookup tables to accelerate Boolean matching, where partial sets of pre-calculated implementable and non-implementable functions of target PLBs are stored. These lookup tables help to quickly filter out non-implementable functions in multiple calls of a Boolean matcher, and the time-consuming SAT-BM is only called for remaining functions. Different from a lookup table-based Boolean matcher [15], our filter-based Boolean matching (F-BM) is capable of handling more complex PLBs.

To verify the effectiveness of the proposed F-BM, we integrate it into a post-mapping re-synthesis algorithm which minimizes area (i.e., LUT number) with the same logic depth constraint [1]. Tested on three different benchmark sets (MCNC, IWLS and Industrial designs) using 3GB memory, the re-synthesizer geared with F-BM is 80X faster with 0.5% more area than the one with the state-of-art SAT-BM [12]. With a 1500-second timeout, which is a common practice to avoid excessive runtime, F-BM based re-synthesizer reduces 2X more LUTs than re-synthesizer based on SAT-BM [12].

The remaining of the paper is organized as follows. Section 2 introduces preliminaries. Section 3 describes the proposed F-BM algorithm. Section 4 presents the post-mapping re-synthesis using the proposed F-BM and experimental results. Section 5 concludes the paper.

2. Preliminaries

2.1 Boolean Matching

An FPGA consists of an array of PLBs. As shown in Fig.1(b), a PLB $H(P)$ consists of a network of interconnected programmable and non-programmable logic devices with a set $P$ of input pins $\{x_1, \ldots, x_p\}$. We sometimes omit
the set of input pins and write \( H \) to refer to the PLB \( H(P) \). A \( K \)-input lookup table (\( K \)-LUT) consists of \( K \) inputs, one output, and \( 2^K \) configuration bits \([L_1, \ldots, L_{2^K}]\).

Boolean matching decides the equivalence of two Boolean functions under input negation/permutation and output negation (NPN). Specifically for FPGA where the PLB can implement only a partial set of all \( P \)-input functions, the Boolean matching problem takes as input a PLB \( H(P) \) and a Boolean function \( f(X) \) over variables \( X \) such that \(|X| \leq |P|\), and decides if the PLB \( H(P) \) can implement (i.e., realize the function) \( f(X) \). If it is implementable, correct configurations for the PLB are generated as well. For the simple case where \( H \) is a \( K \)-LUT, any function \( f(X) \) where \(|X| \leq K \) can be implemented by \( H \).

2.1.1 SAT-Based Method

Among various approaches, SAT-based Boolean matching (SAT-BM) offers the greatest flexibility across different PLB structures [7]. It translates Boolean matching into a SAT problem by formulating the target PLB structure into Conjunctive-Normal-Form (CNF) [16], which is then solved by SAT reasoning [11]. We take an example here to review the entire flow.

Equation (1) and (2) show the CNF encodings for the 2-LUT and AND gate in Fig. 1(b). Such encodings are consistent with the particular functionality of the target gate [16] due to the fact that they’re only satisfiable under correct input and output relationships. For example the first two terms in equation (1) ensure that the correct configuration \( L_1 \) is fetched as output for given input \( x_1 = x_2 = 0 \).

\[
G_{\text{LUT}} = (x_1 + x_2 + \overline{L_1} + z)(x_1 + x_2 + L_1 + \overline{z}) \\
(x_1 + \overline{x_2} + L_2 + z)(x_1 + x_2 + L_2 + \overline{z}) \\
(x_1 + x_2 + \overline{z})(x_1 + x_2 + L_3 + \overline{z}) \\
(x_1 + x_2 + L_3 + z)(x_1 + x_2 + L_4 + \overline{z}) \\
( x_1 + \overline{x_2} + \overline{L_1} + z)(x_1 + x_2 + L_1 + \overline{z}) \\
( x_1 + x_2 + L_2 + z)(x_1 + x_2 + L_2 + \overline{z}) \quad (1)
\]

\[
G_{\text{AND}} = (z + \overline{f})(x_3 + f)(\overline{z} + \overline{x_3} + f) \quad (2)
\]

Combining all components together, equation (3) formulates the CNF encoding for the target PLB.

\[
G = G_{\text{LUT}} \cdot G_{\text{AND}} \quad (3)
\]

To test whether the PLB is capable of implementing \( f(X) \) shown in Fig. 1(a), we need to check every entry of the truth-table for \( f(X) \), by simply extending the CNF encoding into equation (4), where \( X = \{x_1, x_2, x_3\} \) and \( X/000 \) means assigning 000 to \( X \).

\[
G_{SA T} = G(X/000, f/0, z/z_1) \cdot G(X/001, f/0, z/z_2) \cdot \\
G(X/010, f/1, z/z_3) \cdot G(X/011, f/0, z/z_4) \cdot \\
G(X/100, f/1, z/z_5) \cdot G(X/101, f/1, z/z_6) \cdot \\
G(X/110, f/1, z/z_7) \cdot G(X/111, f/1, z/z_8) \quad (4)
\]

\( G_{SA T} \) is then solved by general SAT solver such as [11]. If the target PLB is capable of implementing \( f(X) \), satisfiability (SAT) along with correct assignments for configuration bits (e.g., \( L_i \)) will be returned. On the other hand, unsatisfiability (UNSAT) will be reported if the function is not implementable. Since SAT solver is called every time as a sub-routine, the computational complexity of SAT-BM is still high even with numerous improvements recently [3], [7], [12].

2.2 Bloom Filter

Bloom filter [9], [18] is a space-efficient probabilistic data structure for element’s membership query against a set. It consists of one \( m \)-bit array \( M \) and \( k \) independent hash functions \( h_i(x), 1 \leq i \leq k \), each of which maps or hashes an element \( x \) to one of the \( m \) bit-array positions with a uniform random distribution. In practice, instead of using \( k \) different hash functions, one can pass \( k \) different initial values to a hash function or use \( k \) different bit-fields from the wide output of a hash function, to form the hash function set.

Initially, all bits in \( M \) are set to 0. To insert an element, we feed it to each of the \( k \) hash functions to get \( k \) array positions, and set the bits at all these positions to 1. To query an element (test whether it is in the set), we feed it to each of the \( k \) hash functions to get \( k \) array positions. If any of the bits at these positions is 0, the element is definitely not in the set. If all are 1, we can claim with a high probability that the element is in the set. Note that false positives (i.e., an element is falsely determined to be in the filter while it is actually not) are possible, if the bits at these positions are set to 1 during insertion of other elements.

The false positive rate or probability (FPR) of a bloom filter is bounded by [18]

\[
FPR = (1 - e^{-km/n})^k \quad (5)
\]

where \( n \) is the number of elements already inserted. Taking derivation with respect to \( k \), the optimal \( FPR \) can be achieved

\[
k = \frac{m}{n} \ln 2 \quad (6)
\]

\[
FPR = \left( \frac{1}{2} \right)^k \quad (7)
\]

It is easy to verify that for 1% error (false positive) with the above optimal value of \( k \), only 9.6 bits are required per
element—regardless of the size of the element.

There are three major advantages of Bloom filter over other data structures (e.g., binary search trees, tries and hash tables) for representing sets. Firstly, space efficiency is obtained that regardless of element’s actual size, only constant number of bits are needed per element. Secondly, it takes a constant time (i.e., $O(k)$) to insert or query an element. Thirdly, one can make the tradeoff between false positive rate and space cost depending on the application.

3. Filter-Based Boolean Matching

Bloom filter is used to build the lookup tables storing partial sets of pre-calculated implementable and non-implementable functions of target PLBs. Equipped with these tables, non-implementable functions are quickly filtered out before calling SAT-BM explicitly. In this section, we present details of the proposed F-BM method.

3.1 Building the Lookup Table

Modern SAT solver [11] stops once a satisfiable solution is found, or the whole solution space will be explored when the problem is unsatisfiable. As a result, the runtime for checking an implementable function (SAT) and a non-implementable function (UNSAT) differs significantly. Figure 2 compares the average runtime for checking implementability over 100,000 Boolean functions (with 7–9 inputs) extracted from MCNC benchmarks against a 9-input PLB ($PLB_1$ in Fig. 3) using state-of-art SAT-BM [12]. It shows that the SAT-BM for an implementable function is 5$X$ times faster than that for a non-implementable function.

Since Boolean matching is called as a sub-routine for multiple CAD (Computer-Aided Design) tasks, it is beneficial to prune those non-implementable functions, and only perform the time-consuming SAT-BM for remaining ones. Different from [7], where a coarse-grained SAT solving is used for the pruning, the proposed F-BM is more efficient which filters out non-implementable functions by simple table lookup.

There are different ways to build lookup tables storing implementable and non-implementable functions. The most straightforward way is to enumerate all implementable functions for a PLB. However, it is obviously not practical for large PLBs. Considering a PLB with $|P|$ inputs and $|C|$ configuration bits, the number of functions that it can implement could be $(|P|! \cdot 2^{|C|})$. For $PLB_1$ in Fig. 3 where $|C| = 32$, this number is up to $10^{15}$, which is too large to be enumerated.

Instead of brute-force enumeration, we propose to select a set of training circuits and extract those functions that frequently appear to build the lookup tables. For each of these functions, we use SAT-BM [12] to pre-compute its implementability and insert it into the tables. As will be shown in 3.3, Boolean functions in real circuits exhibit similarities across different benchmarks, and therefore we can apply information extracted from one set of circuits to the others. In our experiments, the training set consists of 10 largest circuits from MCNC benchmark set (i.e., apex2, des, ex1010, pdc, spla, clma, elliptic, frisc, s38417 and s38584). We extract Boolean functions with 5 to 9 inputs using ABC command “cut -K input_size -M 1000” [13]. There are about 2,700,000 distinct functions extracted in this procedure. For each of these functions, we compute and store up to 10,000 of its permutations. Overall, an upper bound of 3 billion functions is inserted into the lookup tables. The training took two weeks in a Linux server with Quad-Core Intel Xeon 2.33 GHz CPU and 32 GB DRAM. However, it is performed only once and these tables can be reused thereafter.

To implement such large lookup tables for both memory and runtime efficiency, we use the Bloom filter described in 2.2. Figure 4 compares the memory cost of the Bloom filter-based table and the hash table-based table to store training results. For Bloom filter, we set 1% false positive rate and thus 9.6 bits are required per function, while for hash table the memory cost grows exponentially with function input size (e.g., 512 bits are required to store a 9-input Boolean function). Note that extra memory to maintain the hash table data structure is ignored. Clearly, the hash table-based implementation quickly reaches the memory limit of a desktop PC (typically with less than 4 GB memory) when input size increases. On the contrary, a trade-off between false-positive rate and table size can further increase the capacity of Bloom filter-based lookup table.
3.2 Selection of Hash Functions

In order to achieve a scalable implementation with low false positive rate, the Bloom filter used in F-BM needs to be carefully customized, where the key design factor is the selection of adequate hash functions. Ideally, hash functions for a Bloom filter need to be perfect random, i.e., input keys should be hashed into each table position with exactly the same probability. Particularly in our application where a function is represented by truth table (0/1 bit string), different functions often show very similar characteristics. Therefore, a good hash function should also be able to magnify the small difference of input keys, e.g., one bit difference should lead to unrelated hash values.

We compare the following four commonly-used hash functions in our implementation, i.e., simple (used in a popular open source Bloom filter project [19]), hash2 and hashlittle2 (two general hash functions from Bob Jenkins [20]) and sha256 (a cryptographic hash function [21]). A bloom filter is implemented using each of these hash functions, respectively. For simple, hash2 and hashlittle2, we pass different initial values to generate the independent hash functions. For sha256, we extract different bit-fields from its 256-bit output as independent hash values.

We quantitatively evaluate the randomness of a hash function used in Bloom filter storing functions with truth table representation. For a Bloom filter of size \( m \) with \( k \) perfect random hash functions, after adding \( n \) elements, the expected number of bits that will be set to 1 for exactly \( i \) times is [9]

\[
E_i = m \cdot C^i_{kn} \cdot \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{kn-1}
\]

Table 1 compares the randomness of the four hash functions by inserting top-1000 most frequently occurred 9-input functions extracted from training set, with settings: \( k=4, m=10,000 \) and \( n=1,000 \). The “Randomness” column denotes the number of bits that are set to 1 for exactly \( i \) times. The data for perfect is calculated by (8), which is the ideal case. The closer the number is to perfect, the more randomness a hash function has. In addition, the “Time” column shows the average runtime to insert one function. Finally, hash2 is chosen to implement the Bloom filter of F-BM for best randomness and efficiency tradeoff.

3.3 Coverage of the Filter

We now evaluate the coverage of Bloom filter-based lookup table generated by the training set described in Sect. 3.1. A 9-input PLB (PLB1 in Fig. 3) is used for testing. Two Bloom filter-based lookup tables are maintained from the training step, one for implementable functions (BF-SAT) and the other for non-implementable functions (BF-UNSAT). The testing set consists of Boolean functions with 5 to 9 inputs extracted from the other 10 MCNC benchmarks (alu4, apex4, bigkey, diffeq, dsip, ex5p, misex3, s298, seq, tseng) using ABC command “cut -K input_size -M 1000.” For each input size, 100,000 functions are randomly selected from the testing set. The following four cases are analyzed:

- **S-hit**: Implementable functions found in BF-SAT;
- **S-miss**: Implementable ones not found in BF-SAT;
- **U-hit**: Non-implementable ones found in BF-UNSAT;
- **U-miss**: Non-implementable ones not found BF-UNSAT;

Column “F-BM” of Table 2 shows the result, which indicates that over 90% of implementable functions are found in BF-SAT (row “S-coverage”). For those functions that are not found in either filter, only 20% of them are implementable (i.e., percentage of S-miss over the sum of both S- and U-misses). In other words, any function that is not found in either filter has a high probability of being non-implementable, and one can drop it (i.e., consider it as non-implementable) without significantly degrading the quality.

To further explore the trade-off between runtime and quality, we propose a learn strategy, F-BM-L, which expands existing Bloom filter-based lookup tables by adding newly trained functions at runtime. For a testing function found in neither SAT-BF or UNSAT-BF, it is definitely not trained before. Therefore, we propose to train such functions using SAT-BM and add results to corresponding tables. In this manner, we’re able to capture special characteristics of testing circuits. Scalable Bloom filter [10], which is capable of dynamic growth when the number of inserted items exceeds pre-defined filter size, can be used to implement such a strategy. The “F-BM-L” column of Table 2 shows that the coverage for non-implementable 9-input functions is improved by over 17% using this strategy.

Table 3 summarizes the corresponding actions for SAT-
BM check based on the query results in both Bloom filter-based lookup tables. Action Check makes an explicit call of SAT-BM to decide a function’s implementability, while action Drop quickly determines a non-implementable function with high probability. For F-BM, only one check against SAT-BF is needed, and UNSAT-BF needs not necessarily be kept. However, for F-BM-L, we need to keep both lookup tables to decide untrained functions. Compared to F-BM, F-BM-L has better pruning quality (i.e., less implementable functions are erroneously pruned).

4. Re-Synthesis with F-BM

The post-mapping re-synthesis which minimizes area (i.e., LUT number) with the same logic depth constraint described in [1] is adopted as an application of the Boolean matching to show the effectiveness of the proposed F-BM. Algorithm 1 shows the pseudo-code of one iteration of the re-synthesis procedure. The algorithm works in a greedy mode, which takes a circuit or network mapped to 3-LUTs (mapped by ABC [13]) and scans the combinational portion of the circuit in a topological order (i.e., each circuit node is scanned strictly after all its fanin nodes having been scanned). During the scanning, new logic blocks (i.e., cuts) are generated by enumerating and combining the logic blocks at inputs of an LUT, which is called cut-enumeration [17]. In line 8, each logic block is checked for its implementability against PLB1 and PLB2 shown in Fig. 3 by calling the Boolean Matching procedure. When an implementable case is found by the Boolean matcher, the logic block is replaced by the corresponding PLB structure if such a replacement reduces the number of LUTs without increasing logic depths. The algorithm terminates after several iterations (e.g., set by user) of full scan of all LUTs, or until no LUT can be further reduced.

Two versions of the re-synthesizer are implemented, one uses the state-of-the-art SAT-BM [12] and the other uses F-BM (including both strategies described in Table 3). To explore the quality of the training set obtained from MCNC benchmarks, two other benchmark sets (IWLS 2005 [14] and Industrial designs) are also tested. Table 4 compares re-synthesis results of different approaches. Column “Reduced LUT #” compares number LUTs reduced during re-synthesis, and column “Total LUT #” denotes number of LUTs in the circuit after re-synthesis.

Compared with re-synthesizer geared with SAT-BM, the one with F-BM (where only implementable functions are stored in Bloom filter) is 80X faster with only 0.5% more area on average. In other words, F-BM-based re-synthesizer achieves magnitude of speedup with negligible area overhead. Concerning number of LUTs reduced, F-BM-based re-synthesizer reduces 11% less LUTs than SAT-BM-based one, due to the possible pruning of implementable functions. To achieve the same area reduction, the learn strategy F-BM-L (where both implementable and non-implementable functions are kept) can be adopted. Compared to SAT-BM-based re-synthesizer, F-BM-L-based one is still 5X faster. Note that 3 GB memory is used for both F-BM and F-BM-L strategies to make the comparison fair.

To further verify the effectiveness of F-BM, Fig. 5 shows the relationship of number of reduced LUTs vs. runtime for largest benchmark circuit leon3 (a micro-processor core with half million 3-LUTs). It is clear that F-BM-based re-synthesizer converges much faster than SAT-BM-based one. In other words, within the same time, F-BM-based re-synthesizer reduces more LUTs. As is shown, 2X more LUT reduction is obtained by F-BM-based re-synthesizer with a 1500s timeout.

The main reason why F-BM-L is a lot slower than F-BM is that much more calls to the SAT solver is required for F-BM-L. From Table 2, we observe the fact that the majority of untrained functions (i.e., misses) are UNSAT. In order to prune those functions, the F-BM-L needs explicit UNSAT checking while for F-BM simple table lookups are enough. Figure 5 gives us intuitions of the relationship of area reduction and runtime as well. Along the horizontal axis, the slope for the area reduction curve becomes less steep, indicating that more time is spent on UNSAT checking. In other words, it’s difficult to get the last a few area reductions. In fact, F-BM-L is designed for area-critical applications. For most applications, F-BM achieves the best area and runtime trade-off.
In this paper, we have presented F-BM, which accelerates Boolean matching using Bloom filter-based lookup tables to quickly prune non-implementable functions with affordable memory. Using post-mapping re-synthesis which minimizes area without increasing logic depth as an application, experiments on MCNC, IWLS 2005 and Industrial design benchmark sets show that re-synthesizer geared with F-BM using 3 GB memory space is 80X faster than the one with a state-of-art SAT-BM [12], with only 0.5% more area. To achieve the same area, F-BM with learn strategy is still more than 5X faster with the same memory cost.

In the future, we will target functions with more inputs and explore different structures of the Bloom filter. For example, we observe that top-10% most common 9-input Boolean functions in MCNC circuits cover 50% of 9-input cuts, and thus we can design a multi-level Bloom filter with different false positive rates at different levels, e.g., a lower false positive rate for 10% of most frequent functions and a higher false positive rate for the rest, to trade accuracy to memory. In addition, we plan to apply our F-BM to other CAD tasks, such as technology mapping, physical synthesis, etc. Finally, we’ll seek methods to store matched configurations as well as satisfiability information to further reduce the number of SAT calls.

5. Conclusions and Future Work

In this paper, we have presented F-BM, which accelerates Boolean matching using Bloom filter-based lookup tables to quickly prune non-implementable functions with affordable memory. Using post-mapping re-synthesis which minimizes area without increasing logic depth as an application, experiments on MCNC, IWLS 2005 and Industrial design benchmark sets show that re-synthesizer geared with F-BM using 3 GB memory space is 80X faster than the one with a state-of-art SAT-BM [12], with only 0.5% more area. To achieve the same area, F-BM with learn strategy is still more than 5X faster with the same memory cost.

In the future, we will target functions with more inputs and explore different structures of the Bloom filter. For example, we observe that top-10% most common 9-input Boolean functions in MCNC circuits cover 50% of 9-input cuts, and thus we can design a multi-level Bloom filter with different false positive rates at different levels, e.g., a lower false positive rate for 10% of most frequent functions and a higher false positive rate for the rest, to trade accuracy to memory. In addition, we plan to apply our F-BM to other CAD tasks, such as technology mapping, physical synthesis, etc. Finally, we’ll seek methods to store matched configurations as well as satisfiability information to further reduce the number of SAT calls.

References

Chun Zhang received his B.E. degree in Microelectronics department from Fudan University, Shanghai, China, in 2005. Since 2005, he has been working on his Ph.D. degree at State Key Lab of ASICs and Systems, School of Microelectronics, Fudan University, Shanghai, China. His research interests include computer-aided design for integrated circuits, design and architectures of Field-programmable gate arrays (FPGAs), FPGA error modeling and robust logic synthesis algorithms.

Yu Hu received his B.E. and M.E. degrees in computer science from Tsinghua University, Beijing, China, in 2002 and 2005, respectively and his Ph.D. degree in Electrical Engineering Department from University California, Los Angeles in 2009. Since 2010, he has been an Assistant Professor with the Department of Electrical and Computer Engineering at University of Alberta. His current research interests include CAD techniques for FPGAs, digital circuit design, etc., and has published two books and over 60 technical papers.