Boolean Matching using Binary Decision Diagrams with Applications to Logic Synthesis and Verification

Yung-Te Lai  
Dept. of EE-Systems  
Univ. of So. California  
LA, CA 90089

Sarma Sastry  
Dept. of ECE  
Univ. of Arizona  
Tuscon, AZ 85721

Massoud Pedram  
Dept. of EE-Systems  
Univ. of So. California  
LA, CA 90089

Abstract

In this paper we present a new algorithm for boolean matching based on Binary Decision Diagrams using a level-first search strategy. This algorithm is generally not restricted to circuits with just a few inputs and can be used for both technology mapping and logic verification. Unlike depth-first and breadth-first strategies, a level-first strategy permits significant pruning of the search space. In addition, we describe a set of filters which further improve the the performance of the matching algorithm. We present a method of analyzing the effectiveness of a filter and rank the various filters based on their effect/cost ratio. We present experimental results on a number of benchmark circuits, comparing the basic matching algorithm with and without the use of various filters. Finally, we show how the matching algorithm and the filters can be extended to boolean functions with don't cares.

1 Introduction

Boolean matching is a method to show whether or not two boolean functions are equivalent under a permutation of their variables. In [8] Mailhòs and De Micheli describe a procedure for carrying out technology mapping [5] using boolean matching. In logic verification, a specification describing the intended behavior and an implementation describing the behavior of the basic components and their interconnections are to be shown as equivalent. Existing logic verification systems and approaches [4, 6, 7, 9] assume that both the specification and implementation use the same set of variables. Such a constraint can be relaxed if we incorporate boolean matching into the verification process.

There are three reasons why the boolean matching algorithm of [8] may not suitable for hardware verification. First, they use properties of unateness and symmetry of variables as filters to speed up their matching algorithm. The property of symmetry may be useful for circuits with relatively few inputs but it is unlikely that same will be true for circuits with a large number of inputs. Second, they use a matching compatibility graph to cover the cases of incompletely specified functions, that is, functions with don't cares. Due to the nature of such a graph, it seems practical for circuits with \( \leq 4 \) inputs. The third and perhaps the most important reason is that their matching algorithm is depth-first. A depth-first procedure performs poorly in pruning the search space even with the use of various filters. That is, if \( u \) is a node in the search tree, a depth-first procedure will generally not be able to determine that two functions are not equivalent until the siblings of \( u \) are processed.

In this paper, we present a new algorithm for boolean matching which can be applied to both logic verification and technology mapping. Our algorithm uses Binary Decision Diagrams (BDD) [2] as the basic data structure for representing boolean functions and is based on checking whether two BDDs are isomorphic. Our method for checking for isomorphism is a significant departure from the standard approach in that it is performed in a level-first manner and is combined with a partial permutation on the variables. This approach has a number advantages over breadth-first and depth-first methods. First, when the isomorphism check fails at level \( i \) on the permutation \( (y_0, \ldots, y_{i-1}, y_i, \ldots, y_{n-1}) \), then the check of another permutation \( (y_0, \ldots, y_{i-1}, y_{\pi(i)}, \ldots, y_{\pi(n-1)}) \) can be started at level \( i \) instead of level 0. Second, the \((n-1)!\) permutations on \( y_{n+1}, \ldots, y_{n-i} \) of \( (y_0, \ldots, y_i, y_{\pi(i+1)}, \ldots, y_{\pi(n-1)}) \) can be discarded. Finally, the use of BDDs for boolean matching makes the extension to incompletely specified functions straightforward.

In addition to a new procedure for boolean matching, we describe a set of filters which are easy to compute using BDDs and are more effective than filters based solely on the properties of unateness and symmetry. We analyze the complexity or cost of applying each filter and present a method for analytically comparing the effectiveness of a filter. Finally, we provide results of experiments that were carried out on a number of benchmark circuits applying boolean matching with and without the use of various filters.

2 Boolean Matching Algorithm

The problem of boolean matching is defined as follows:

Given two vectors of boolean functions
\[
\langle f_1(x_0, \ldots, x_{n-1}), \ldots, f_m(x_0, \ldots, x_{n-1}) \rangle \quad \text{and} \quad \langle g_1(y_0, \ldots, y_{n-1}), \ldots, g_n(y_0, \ldots, y_{n-1}) \rangle,
\]

does there exist two bijections \( \pi_0 : \{f_1, \ldots, f_m\} \rightarrow \{g_1, \ldots, g_n\} \) -

-
This is clearly not practical. Our matching algorithm is to enumerate the set of all bijections, of which there are $2^{m!2^n!}$ in number, and perform tautology checking. When such bijections do exist, consisting of two types of nodes. A node is lent where

Based on the isomorphism checking on BDDs which can be carried out in the time proportional to the sizes of BDDs, the number of nodes in a BDD will be denoted by $m$. Respectively, then

Two BDDs rooted by $bdd_1$ and $bdd_2$ are isomorphic iff there exists a bijection $h$ between the descendant nodes of $bdd_1$ and $bdd_2$ such that if $h(u) = v$, where $u$ and $v$ are the descendant nodes of $bdd_1$ and $bdd_2$, respectively, then

1. $index(u) = index(v)$,
2. $value(u) = value(v)$, if $u$ and $v$ are terminal nodes,
3. $h(child_1(u)) = child_1(v)$ and $h(child_2(u)) = child_2(v)$.

We first describe the basic algorithm for checking input-permutation equivalence based on two BDDs and then discuss how it can be extended to check output-permutation, input- and output-negation equivalences. The various filters that speed up the process will be presented in the following section.

Given two functions $f$ and $g$, we first construct their BDDs, $bdd_f$ and $bdd_g$. Then $bdd_g$ is incrementally transformed to the same form as $bdd_f$. The transformation is carried out level by level by permuting a subset of variables of $g$ while performing a check for isomorphism.

To check whether $bdd_f$ and $bdd_g$ are isomorphic, the nodes are searched in level-first manner, that is, pairs of nodes to be mapped are added to a set $M$ and the elements of this set are processed in the order of the index of the node in $bdd_f$. As an example, consider the BDDs shown in Figure 1. Using breadth-first search, the order of the elements that are processed is $(w_0, v_0), (w_1, v_1), (w_2, v_2), ...$ etc. In level-first search, the order would be $(w_0, v_0), (w_2, v_2), (w_1, v_1), ...$ etc.

At each level $i$, we perform the following operations. For every $(u, v)$ in $M$ such that $index(u) = i$, we first check if $index(v) = i$. If this condition is satisfied, we then check if $(child_1(u), child_1(v))$ and $(child_2(u), child_2(v))$ will cause conflicts. A conflict exists between two nodes if both of them are terminal nodes but have different values, or if one of them is a terminal node while the other one is not.

If there is no conflict, then we proceed to the next level. If $index(v) \neq i$ or there is a conflict between $child_1(u)$ and $child_1(v)$ or between $child_2(u)$ and $child_2(v)$, then we perform the replace-root operation as shown in Figure 2. In Figure 2(a), the right child of $u$ and $v$ cause a conflict. Thus, we make $y_i$ to be the new root of the subtree previously rooted at $y_i$ (Figure 2(b)).

After performing replace-root, we carry out another check for isomorphism starting at level $i$ for the new permutation $(...y_{i-1}, y_i, y_{i+1}, ..., y_{i-1}, y_{i+1}, ...)$.

Note that all the $(n - i - 1)$! permutations of $y_0, ..., y_i, ...$ (with $y_0, ..., y_i$, fixed in the first $i + 1$ positions) have been discarded. If $y_i$ still fails, then we try $y_i$ which has not yet been tried as a new root. This replace-root operation can be repeated until no $y_i$ can be the new root, at which point we backtrack to level $i - 1$ and perform a replace-root operation on $y_{i-1}$. Proceeding in this manner, an isomorphism is found if we advance to level $n$, or we conclude that no isomorphism exists if we backtrack to level $-1$.

In the procedure replace-root, cofactor creates two BDDs, $bdd_l$ and $bdd_r$, such that the functions $h, l$ and $r$ denoted by $bdd_l$, $bdd_l$ and $bdd_r$ satisfy $h = z + r$. The procedure newbdd creates a new BDD rooted by a node $v$ with name$(v) = z$, child$(v) = bdd_l$, and child$(v) = bdd_r$. For every variable $s' \in index(z) > index(z') > index(bdd_h)$, procedure update_index changes its index to $index(z') + 1$ and then changes the index of $z$ to index(bdd_h).

In the procedure replace_root, cofactor creates two BDDs, $bdd_l$ and $bdd_r$, such that the functions $h, l$ and $r$ denoted by $bdd_l$, $bdd_l$ and $bdd_r$ satisfy $h = z + r$. The procedure newbdd creates a new BDD rooted by a node $v$ with name$(v) = z$, child$(v) = bdd_l$, and child$(v) = bdd_r$. For every variable $z'$ with index$(z) > index(z') > index(bdd_h)$, procedure update_index changes its index to $index(z') + 1$ and then changes the index of $z$ to index(bdd_h).

replace_root(bdd_h, z) {
  if (index(bdd_h) > index(z)) return(bdd_h);
  else {
    (bdd_l, bdd_r) = cofactor(bdd_h, z);
    bdd_new = newbdd(z, bdd_l, bdd_r);
    update_index(bdd_new, index(z));
    return(bdd_new);
  }
}

Example 2.1 As an example of how our boolean matching algorithm works, consider the BDDs shown in Figure 1. Initially, $M$ consists of $\{(u_0, v_0)\}$. At level 0, we first
check if $index(v_0) = 0$, and then add $\{u_1, v_1, (u_2, v_2)\}$ to the set $M$. At level 1, since $index(u_2) = 1$, we check if the index of $v_2$ is 1, as it is in this case. We then find that $child(u_2)$ and $child(v_2)$ cause a conflict. Thus we perform replace_root operation on variable $y_2$. After replace_root, we have the new $bdd_2$ as shown in Figure 3 and the steps outlined above are carried out.

For output-permutation equivalence, two vectors of BDDs are used for isomorphism checking. To handle input-negation equivalence, the conflict checking of $(child(u),child(v))$ is replaced by the checking of $(child(u),child(v))$ and $(child(u),child(v))$. In the basic algorithm, the terminal node with value $1(0)$ is isomorphic with the terminal node with value $1(0)$, while in the case of output-negation equivalence checking, the terminal node with value $1(0)$ is isomorphic with the terminal node with value $0(1)$.

3 Matching Filters

An output variable filter is a function $F^\text{out} : \{f_1, \ldots, f_n\} \rightarrow r$, $r \in R(F^\text{out})$, where $R(F^\text{out})$ is some finite set, and $f_i$ is a boolean function on $n$ variables.

Similarly, an input variable filter is a function $F^\text{in} : \{f(x_0, \ldots, x_{n-1}), x_r \rightarrow r, x_r \in \{x_0, x_{n-1}\}, r \in R(F^\text{in})\}$, where $R(F^\text{in})$ is some finite set.

To speed up the task of boolean matching, we examine a set of output and input variable filters which are derived from the necessary conditions for nnpp-equivalence, that is, they have the properties that if $F^\text{out}(f_i) \neq F^\text{out}(g'_j)$ then $(\ldots, f_{i+1}, \ldots)$ and $(\ldots, g'_{j+1}, \ldots)$ cannot result in nnpp-equivalent under $g'_j = \pi_0(f_i)$, and if $F^\text{in}(f(x_i)) \neq F^\text{in}(g(x_r))$ then the set of variables $(\ldots, x_{i+1}, \ldots)$ under $g'_j = \pi_0(f_i)$.

An example of a simple filter is the size of the BDD prior to the execution of boolean matching, or the on-set.

All filters presented in this paper are computed using the BDD representation. A filter can be applied to the entire BDD prior to the execution of boolean matching, or to the sub-BDD rooted at nodes that are being compared. It can also be applied to the outputs of a multiple output circuit. The latter is particularly useful for circuits that implement arithmetic functions where almost every output function results in a unique value of $F^\text{out}$. An input variable filter based on the dependence set is $F^\text{in}(f, x_r) = true(false)$ if $x_r$ is/is not in $Dep(f)$.

3.2 Cardinality of On-set

The on-set of a function $f(x_0, \ldots, x_{n-1})$ is the set of input assignments where $f = 1$. $F^\text{on}$ is the cardinality of the on-set of $f$ and can be computed in $O(m)$ time and $R(F^\text{on}) = \{0, 1, \ldots, 2^m\}$.

$$F^\text{on}(f) = \begin{cases} \text{if } (index(f) == n) \text{ return } value(f) \text{; } \\
\text{else } \\
\{ \\
\text{left} = Dep(child(l(f))) \\
\text{right} = Dep(child(r(f))) \\
\text{return } (leftlicts right u index(f)) \text{; } \\
\} \}$$

$F^\text{on}$ as defined above, can be used to reduce the number of possible mappings between the outputs of two multiple output circuits. It can also be used to reduce the number of mappings among input variables. For this we define a function $F^\text{on}(f, x_r) = F^\text{on}(f(x_0, \ldots, x_{n-1}, 1, x_r, \ldots))$. If $F^\text{on}(f, x_r) = \ldots = F^\text{on}(f, x_{r-1}) = F^\text{on}(f, y_r) = \ldots = F^\text{on}(f, y_{r-1})$, then the sets of variables $\{x_1, \ldots, x_r\}$ must be mapped to the set $\{y_1, \ldots, y_r\}$. This property can be used when attempting to match $x_r$ to $y_r$ during boolean matching. Furthermore, for two functions $f(x_0, \ldots, x_{n-1})$ and $g_0(y_0, \ldots, y_{n-1})$ to be $p$-equivalent, the multi-sets $\{F^\text{on}(f, x_0), \ldots, F^\text{on}(f, x_{n-1})\}$ and $\{F^\text{on}(g_0, y_0), \ldots, F^\text{on}(g_0, y_{n-1})\}$ must be the same. Thus, a new filter $F^\text{on}_p(f, x_r)$ can be defined as $\{F^\text{on}_p(f, x_0), \ldots, F^\text{on}_p(f, x_{n-1})\}$.

3.3 Sizes of Distance $k$

Let $D_k(f)$ denote the set of pairs of one-points of $f(x_0, \ldots, x_{n-1})$ whose Hamming distance is $k$. Then $F^\text{on}_k = D_k(f)$ and $R(F^\text{on}_k) = \{0, 1, \ldots, 2^m\}$. $F^\text{on}_k$ represents a set of filters. $F^\text{on}_k$ can be applied to the outputs of a multi-output circuit prior to executing boolean matching, thereby reducing the number of possible mappings between the outputs of two circuits. $F^\text{on}_k$ can also
be used while performing boolean matching. We present algorithms for determining $F_{\text{out}}$ and $F_{\text{out}}^\text{swap}$. In general, the computation of $F_{\text{out}}^\text{swap}$ can be carried out in the same way as of $F_{\text{out}}$.

$$F_{\text{out}}^\text{swap}(bdd)$$

- if (index(bdd) == n) return(0);
- else {
  $$dn = F_{\text{out}}(\text{bdd.and(child(bdd)), swap(child(bdd)))}$$
  if (index(child(bdd)) > index(child(bdd)))
    return(dn * $2^n$ - index(child(bdd)));
  else return(dn + $2^n$ - index(child(bdd)));
}

swap(bdd)

- if (index(bdd) == n) return(bdd);
- else return(new_bdd (root(bdd), swap(child(bdd)), swap(child(bdd))));

In $F_{\text{out}}$, both $F_{\text{out}}$ and swap take $O(m)$ time, but the operation of bdd.and(u, v) takes $O(m^2)$ time. Thus, the complexity of $F_{\text{out}}^\text{swap}$ is $O(m^2)$.

$$F_{\text{out}}^\text{swap}(bddd)$$

- n1 = 0;
  for (i = 0; i < n; i++) {
    cofactor(bdd, x0, bdd, bdd);  
    left = $F_{\text{out}}$ (update.index(bdd, x0));
    right = $F_{\text{out}}$ (update.index(bdd, x0));
    n1 = n1 + left + right;
  }

Both cofactor and update.index take $O(n)$ time while $F_{\text{out}}$ takes $O(m^2)$ time, thus the complexity of $F_{\text{out}}^\text{swap}$ is $O(nm^2)$. In general, the complexity of computing distance $n - k$ is $O(C_n^k)$ time. Similar to $F_{\text{out}}^\text{swap}(f, x)$, we define $F_{\text{out}}^\text{swap}(f, x) = F_{\text{out}}^\text{swap}(f, x_0, x_1, x_2, \ldots, x_n)$.

### 3.4 Unateness of Input Variables

A variable $x_i$ is monotone increasing (monotone decreasing) in a function $f$ if for all input assignments of $x_j \neq x_i$, $f(\ldots, x_{i-1}, 0, x_{i+1}, \ldots) \leq (\geq) f(\ldots, x_{i-1}, 1, x_{i+1}, \ldots)$. A variable $x_i$ is binate in $f$ if $x_i$ is neither monotone increasing nor monotone decreasing.

The unateness of an input variable can be used as a filter $F_{\text{unate}}(f, x_i) \in \{\text{inc}, \text{dec}, \text{binate}\}$. This property can also be used as an output variable filter in the following way. Let $S_{\text{inc}}(f) = \{x_i | F_{\text{unate}}^\text{inc}(f, x_i) = \text{inc} \}$, $S_{\text{dec}}(f)$ and $S_{\text{binate}}(f)$ are defined similarly. Then the cardinalities of these sets define the filter $F_{\text{unate}}^\text{inc}(f) = |S_{\text{inc}}|$, $|S_{\text{dec}}|$, and $|S_{\text{binate}}|$ with $R(F_{\text{unate}}^\text{inc}) = \{(n, 0, 0), (n-1, 1, 0), (n-2, 1, 1), \ldots, (0, n, n)\}$ and $R(F_{\text{unate}}^\text{dec}) = |C_n^2|$. The function $F_{\text{unate}}^\text{inc}(f, x_i)$ can be computed in $O(m^2)$ as shown below:

$$F_{\text{unate}}^\text{inc}(f, x_i)$$

  - cofactor(bdd, x0, bdd, bdd);
  - if (bdd.op(bdd, bdd) \geq) return(inc);
  - else (bdd.op(bdd, bdd) \leq) return(dec);
  - else return(binate);

3.5 Symmetry Classes of Input Variables

Two variables $x_i$ and $x_j$ are symmetric if they can be interchanged without changing the function value, that is, $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. We can determine if $x_i$ and $x_j$ are symmetric by testing if $f(\ldots, x_{i-1}, 0, x_{i+1}, 1, x_{j+1}, \ldots) = f(\ldots, x_{i-1}, 1, x_{i+1}, 0, x_{j+1}, \ldots)$. The following algorithm returns true if $x_i$ and $x_j$ are symmetric under the function denoted by bdd.

symmetry(bdd, x_, x_j)

  - cofactor(bdd, x_, bdd, bdd);
  - cofactor(bdd, x_j, bdd, bdd);
  - cofactor(bdd, x_, bdd, bdd);
  - cofactor(bdd, x_j, bdd, bdd);
  - return(bdd == bdd);

Let $S_{\text{sym}}(f)$ be the maximal symmetry classes of $f$, that is,

$$S_{\text{sym}}(f) = \{X_1, \ldots, X_k\}, \bigwedge X_i = \{x_{i_0}, \ldots, x_{i_{n-1}}\}, X_i \cap X_j = \emptyset, x_i, x_j \text{ are symmetric}$$

metric if and only if $x_i \in X_l$ and $x_j \in X_l$, for some $l$.

The input and output variable filters based on $S_{\text{sym}}(f)$ are

$$\begin{align*}
F_{\text{out}}(f) &= \{|X_1|, \ldots, |X_k| \} \\
F_{\text{in}}(f, x_i) &= |X_l|, \text{where } x_i \in X_l.
\end{align*}$$

The ranges of these filters are $R(F_{\text{out}}) = \{(1, 1), (1, 2), \ldots, (n)\}$ and $R(F_{\text{in}}) = \{1, 2, \ldots, n\}$. The cardinality of $R(F_{\text{sym}})$ is the number of decompositions of the integer $n$.

3.6 Use of Filters

In this section we explain, by the way of a small example, how the various filters can be used. As stated earlier, a filter can be applied on the outputs, inputs and during the matching process. Furthermore, the result of applying one filter can itself be used as another filter.

Consider the following boolean functions each of which has three inputs $a$, $b$, and $c$, where

$$f(a, b, c) = ab + bc + abc$$
$$g(a, b, c) = ab + abc$$
$$g_1(a, b, c) = bc$$
$$g_2(a, b, c) = ab + bc + ac$$
$$g_3(a, b, c) = abc + ab + ac$$
$$g_4(a, b, c) = ab + ac + bc$$
We want to find out which of \( g_n \) are \( p \)-equivalent to \( f \).

Applying \( F_{\text{out}} \) to the above functions results in

\[
F_{\text{out}}^p : (f, g_0, g_1, g_2, g_3, g_4) = (3, 3, 2, 3, 3).
\]

which prunes \( g_1 \). Then we apply \( F_{\text{on}}^p \) and get

\[
F_{\text{on}}^p : (f, g_0, g_2, g_3, g_4) = (4, 3, 4, 4, 4).
\]

This filter prunes function \( g_0 \). Next, we apply \( F_{\text{on}}^m \) to the rest of functions:

\[
F_{\text{on}}^m (f, a, b, c) = (2, 1, 2)
\]

\[
F_{\text{on}}^m (g_2) = (2, 1, 3)
\]

\[
F_{\text{on}}^m (g_3) = (1, 2, 2)
\]

\[
F_{\text{on}}^m (g_4) = (3, 1, 3).
\]

By comparing the multi-sets of \( F_{\text{on}}^p \) we know that both \( g_2 \) and \( g_4 \) cannot be \( p \)-equivalent to \( f \). Furthermore, variable \( b \) of \( f \) can only be mapped by variable \( a \) of \( g_2 \) because they have the same \( F_{\text{on}}^m \) value. This reduces the number of possible mappings among the inputs between \( g_2 \) and \( g_4 \) to 2 instead of 3.

Alternatively, instead of using \( F_{\text{on}}^m \) as an output filter we can use \( F_{\text{on}}^m \) to prune \( g_2 \) and \( g_4 \) because

\[
F_{\text{on}}^m (f, g_0, g_2, g_3, g_4) = (2, 3, 2, 3)
\]

\[
F_{\text{on}}^m (f, g_0, g_2, g_3, g_4) = (3, 2, 3, 3)
\]

\[
F_{\text{on}}^m (f, g_0, g_2, g_3, g_4) = (1, 1, 1, 1, 0).
\]

### 3.7 Comparison of Filters

As mentioned earlier, there is a tradeoff between the cost of applying a filter and how effective it is in pruning the search space. To measure a filter's effectiveness we need to examine the distribution of values of the filter over the sample space of all boolean function of \( n \) variables. A filter partitions the set of all \( 2^{2^n} \) functions of \( n \) variables into equivalence classes, where two functions belong to the same equivalence class if they have the same filter value.

Let \( \eta(F, n) \) be the number of equivalence classes formed by \( F \) and \( N(F, n, k) \) denote the number of boolean functions \( f \) of \( n \) variables such that \( F(f) = k \), \( 0 \leq k \leq \eta(F, n) \). Let \( P(F, n) \) be the probability that two arbitrarily chosen functions \( f \) and \( g \) of \( n \) variables result in \( F(f) = F(g) \). An equivalent measure of a filter's effectiveness. The condition \( P(F_1, n) < P(F_2, n) \) implies that two arbitrary functions will be less likely to have the same filter value under \( F_1 \) than under \( F_2 \). Thus, \( F_1 \) is more likely than \( F_2 \) to declare that the two non \( p \)-equivalent functions are indeed not \( p \)-equivalent. Once we know the number of equivalence classes \( \eta(F, n) \) and the cardinality of each class, then \( P(F, n) \) is easily computed. This is given in the following lemma.

**Lemma 3.1**

\[
P(F, n) = \frac{\sum k \eta(F, n) N^2(F, n, k) + 2^n}{(2^n + 1)2^n}.
\]

**Proof:** The proof rests on the following fact. Given a set of \( m \) elements, the number of ways of selecting \( j \) of them without regard to order and with repetitions allowed is \( C_{m+j-1} \) (binomial coefficient). Therefore the total number of ways of selecting two functions from the set of all \( 2^{2^n} \) functions of \( n \) variables without regard to order and with repetitions allowed is \( T_n = (2^{2^n} + 1)2^{2n/2} \). Using the same argument, the number of ways of selecting two functions from one of the \( \eta(F, n) \) equivalence classes which has \( N(F, n, k) \) elements is

\[
A_{n,k} = \frac{(N(F, n, k) + 1) \eta(F, n) \eta(F, n, k)}{2^n}.
\]

Therefore

\[
P(F, n) = \frac{\sum_{k=0}^{\eta(F, n)} A_{n,k}}{T_n}.
\]

The result follows since \( \sum_{k=0}^{\eta(F, n)} N(F, n, k) = 2^n \).

As an example, consider \( F_{\text{on}}^p \). Then \( \eta(F_{\text{on}}^p, n) = 2^n \), \( N(F_{\text{on}}^p, n, k) = C_k \) and \( P(F_{\text{on}}^p, n) \) is given by

\[
P(F_{\text{on}}^p, n) = \frac{(2^n + 1)^2 + 2^n}{(2^{2n} + 1)2^{2n/2}} \times \frac{1}{\sqrt{2^n}}.
\]

For \( F_{\text{dep}}, \eta(F_{\text{dep}}^p, n) = n \) and \( N(F_{\text{dep}}^p, n, k) \) is given by

\[
N(F_{\text{dep}}^p, n, k) = \sum_{i=0}^{k} (-1)^i (k \choose i) 2^{k-i}.
\]

A closed form for \( P(F_{\text{dep}}^p, n) \) is not yet available, and it must be computed using Equations 1 and 3. However, it can be shown that \( P(F_{\text{on}}^m, n) < P(F_{\text{dep}}^p, n) \), which implies that \( F_{\text{on}}^m \) would be more effective than \( F_{\text{dep}}^p \).

Similarly, we can determine the forms of \( P(F, n) \) for each of the filters described in this paper. In this way it is possible to order the filters based on increasing values of \( P(F, n) \). A complete analysis of the filters is beyond the scope of this paper. However, it is possible to get another equivalent but highly simplified measure of a filter's effectiveness. This is simply the cardinality of \( R(F) \). For the set of filters discussed in this paper, the order obtained using the simplified measure is the same as the one based on \( P(F, n) \). Table 7 shows the cost of applying each filter and the cardinality of its range.

Based on the simplistic measure of a filter's effectiveness, we note that

1. \( F_{\text{on}} \) has the best effect/cost ratio, a property that is supported by our experimental results.
2. Although the effect/cost ratio of \( F_{\text{dep}} \) is not as good as \( F_{\text{on}} \) and \( F_{\text{on}}^m \), it is very inexpensive to compute. Furthermore, the computation of \( F_{\text{on}} \) and \( F_{\text{on}}^m \) requires determining the dependence set.
3. The effect/cost ratio of \( F_{\text{on}}^m \) is moderate for extreme values of \( k \) with respect to \( n \).
Note that in the above analysis, each filter is analyzed independently of other filters. A much more accurate analysis is possible if the effect of one filter is taken into account in evaluating another. This work is currently in progress.

4 Experimental Results

The boolean matching algorithm and the various filters have been tested on the MCNC and ISCAS benchmark circuits. For each circuit we constructed two BDDs. The second BDD was generated by randomly permuting the ordering for the initial BDDs. By using any boolean matching algorithm with and without the use of filters. The following legend describes the data in each column.

1. A : boolean matching with no filters used.
2. B : \text{F}_{on} computed and compared during boolean matching.
3. C : \{F_{out}^o, F_{out}^l, F_{out}^m\} used in the given order as long as more than two outputs have the same filter value and computed and compared before boolean matching.
4. D : \text{F}_{on} filters are used as in C (before matching) and then \text{F}_{on} is used as in B (during matching).

The results displayed in Table 2 are summarized below.

- Comparing columns A and B, we find that in those cases where the boolean matching was completed in a reasonable amount of time, the application of \text{F}_{on} resulted in speed up ranging from 10 to 28 times.

- Comparing columns A and C, we find a speed up ranging from 17 to 430 times. We also note that for circuits \text{mises2}, \text{mises3}, \text{duke2} and \text{c432} the matching algorithm with any filters had to be aborted. However, using the filters \{F_{out}^o, F_{out}^l, F_{out}^m\} and \text{F}_{on} resulted in a successful completion within a very short period of time.

- When \text{F}_{on} or \text{F}_{in} were applied in conjunction with all the filters in \{\text{F}_{out}^o, \text{F}_{out}^l, \text{F}_{out}^m\}, then the matching algorithm was successful in all cases.

- For circuits with a small number of inputs the filters in \{\text{F}_{out}^o, \text{F}_{out}^l, \text{F}_{out}^m\} seem to be more effective than \text{F}_{on}. However, \text{F}_{on} is very effective for larger circuits.

The above results are generated using the input sequences as the ordering for the initial BDDs. By using any of the other ordering strategies that have been reported [1, 3] we expect further improvement in performance.

5 Don't Care Sets

In this section, we discuss the effect of boolean functions with don't cares on boolean matching. A boolean function \( f \) with don't cares is denoted by \( (f_{on}, f_{dc}) \) where the onset of \( f_{on} \) is the onset of \( f \) and the onset of \( f_{dc} \) is the don't care set of \( f \). Before we extend our matching algorithm, we have the following definitions and lemmas.

Definition 5.1 Two boolean functions \( f \) and \( g \) are unifiable, denoted by \( f \cong g \), if \( f_{on} \subseteq g_{on} + g_{dc} \) and \( g_{on} \subseteq f_{on} + f_{dc} \). That is, there exist some don't care assignments such that \( f \) and \( g \) become equal.

Definition 5.2 Given two boolean functions \( f \) and \( g \) which are unifiable, the maximum unifier of \( f \) and \( g \), denoted by \( f \equiv g \), is the function derived by the minimum don't care assignment on \( f \) and \( g \) such that they become equal.

Lemma 5.1 If \( f \cong g \) then \( f \equiv g \equiv (h_{on}, h_{dc}) \) where \( h_{on} = f_{on} + g_{on} \) and \( h_{dc} = f_{dc} + g_{dc} \).

Lemma 5.2 If \( f \equiv g \) then \( f \equiv g \equiv f \equiv g \equiv f \equiv g \equiv f \).

Note that in the above analysis, each filter is analyzed independently of other filters. A much more accurate analysis is possible if the effect of one filter is taken into account in evaluating another. This work is currently in progress.

6 Summary and Concluding Remarks

In summary, the advantages of our approach are twofold. First, in a breadth-first or depth-first manner, every isomorphism check is started from the root nodes. In our method, a new isomorphism check can be started as part of another isomorphism check. Second, and more importantly, if the isomorphism check can not be satisfied at level \( i \) while trying to match \( x_i \) with \( y_{on(i)} \), then the \((n - i)!\) permutations of \( y_{on(i+1)}, \ldots, y_{on(n-1)} \) on \( \{y_{on(i)}, \ldots, y_{on(n-1)}\} \) \( y_{on(i)}, y_{on(i+1)}, \ldots, y_{on(n-1)} \) will not be necessary and thus can be avoided.
7 Acknowledgement
This research was supported in part by NSF RIA grants MIP-9111206 and MIP-9211668.

References

Figure 1: (a) $x_0x_1 + x_2x_3$ (b) $y_0y_2 + y_1y_3$

Figure 2: The operation of replace_root

Figure 3: The new $b_{dd_f}$ after replace_root

Table 1: Comparison of Filters

| Filter F | Cost | $|R(F)| - 1 = n(F,m) |
|----------|------|------------------|
| $F_{dep}$ | $O(m)$ | $n$ |
| $F_{in}$ | $O(m)$ | $2^n$ |
| $F_{aux}$ | $O(C_m^2m^2)$ | $C_m^2n^{2m-1}$ |
| $F_{mate}$ | $O(m^2)$ | $C_m^{2n}$ |
| $F_{sym}$ | $O(n^2m)$ | $O(n^{-1}e^{n/17})$ |

Table 2: Experimental Results

<table>
<thead>
<tr>
<th>in.out</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa02</td>
<td>10</td>
<td>4</td>
<td>12.39</td>
<td>0.85</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>1.85</td>
<td>0.10</td>
</tr>
<tr>
<td>rd84</td>
<td>8</td>
<td>4</td>
<td>6.27</td>
<td>0.22</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>7</td>
<td>13.13</td>
<td>12.52</td>
</tr>
<tr>
<td>misex2</td>
<td>25</td>
<td>18</td>
<td>---</td>
<td>3.68</td>
</tr>
<tr>
<td>misex3</td>
<td>14</td>
<td>14</td>
<td>---</td>
<td>13.25</td>
</tr>
<tr>
<td>apex1</td>
<td>45</td>
<td>45</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>apex2</td>
<td>39</td>
<td>3</td>
<td>221.42</td>
<td>---</td>
</tr>
<tr>
<td>duke2</td>
<td>22</td>
<td>29</td>
<td>---</td>
<td>3.50</td>
</tr>
<tr>
<td>c432</td>
<td>36</td>
<td>7</td>
<td>1279.72</td>
<td>29.55</td>
</tr>
</tbody>
</table>