Building a Better Boolean Matcher and Symmetry Detector

Donald Chai[1] Andreas Kuehlmann[1,2]
[1] University of California at Berkeley, CA, USA
[2] Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

Boolean matching is a powerful technique that has been used in technology mapping to overcome the limitations of structural pattern matching. The current basis for performing Boolean matching is the computation of a canonical form to represent functions that are equivalent under negation and permutation of inputs and outputs. In this paper, we first present a detailed analysis of previous techniques for Boolean matching. We then describe a novel combination of existing methods and new ideas that results in a matcher which is dramatically faster than previous work. We point out that the presented algorithm is equally relevant for detecting generalized functional symmetries, which has broad applications in logic optimization and verification.

1 Introduction

The technology mapping step in a synthesis flow converts a subject graph composed of logic primitives into an implementation composed of library cells. The mapping typically consists of two phases: (1) enumerating subgraphs and *matching* them against library cells, and (2) performing a *covering* of the graph with the matches found. The authors of [1] proposed the use of Boolean matching in the first mapping phase. This technique compares the subgraphs with the library cells by their associated functions. For a given subgraph, a Boolean matcher finds all library cells that are equivalent.

A generalized matching criterion considers, besides permuting function inputs and outputs, also their negations. Two functions are NPN-equivalent[1] if one can be obtained from the other by negation and/or permutation of the inputs and outputs. The dominant approach for Boolean matching is to compute for each library cell a canonical (or semi-canonical) form that is invariant with respect to input/output negation/permutation which is then stored in a hash table [2, 3, 4, 5, 6, 7]. To attempt a match for a subgraph, the matcher computes the canonical form for its associated function and looks it up in the hash table. Matchers based on canonical forms are particularly suited to large cell libraries, because the runtime is independent of the size of the library. This is especially important when cell libraries are enriched with user-defined elements.

A problem closely related to Boolean matching is the detection of functional symmetries: given a function f, find a negation and/or permutation of inputs and outputs which does not change f. In this paper, we use this generalized notion of symmetry and not the more narrow definition based on swapping pairs of variables.

Functional symmetries are important in synthesis, because they often represent functionally equivalent implementations with different costs. For example, the inputs of an AND gate are functionally identical, but have different timing properties. A technology mapper could chose a faster input for late arriving signals. Similarly, a buffer insertion program can use the fact that the function of an XOR gate is unchanged when one inverter is simultaneously placed at two inputs. Furthermore, the knowledge of functional symmetries can significantly aid verification methods [8, 9].

In this work, we discuss the design of a fast Boolean matcher based on canonical forms. We start by reviewing recent work on canonical form computation. We then propose a series of improvements to canonicizers that greatly speed up computation, and show the connection between symmetry detection and Boolean matching. Finally, we outline the overall design of a Boolean matcher that is orders of magnitude faster than those from previous works.

In the following we restrict the presentation to single-output functions. All concepts and algorithms can be easily extended to multi-output functions by examining their characteristic function.

[1] In this paper, "equivalent" without any preceding qualifiers means "NPN-equivalent".

2 Preliminaries and Analysis of Previous Work

Definition 1 *A* configuration *c of a function f is a negation and/or permutation of some of its inputs and output. The function resulting from applying configuration c to f is denoted by f^c.*

Clearly, for a function of n inputs, there are $2^{n+1}n!$ distinct configurations. We denote a configuration c by a string of literals assigning v's to x's to denote the permutation and phase of the inputs and a z indicating the phase of the output. For example, applying the identity configuration $c_I = v_{n-1} \ldots v_0 z$ to function $y = f(x_2, x_1, x_0)$ leaves the inputs and output unchanged. In contrast, applying configuration $\bar{v}_1 v_0 v_2 \bar{z}$ results in a new function $\overline{f(\bar{v}_1, v_0, v_2)}$.

Definition 2 *A configuration c is a* symmetry *for function f if $f^c = f$. Two configurations c_1 and c_2 are symmetric with respect to f if $f^{c_1} = f^{c_2}$.*

For example, the configuration $v_1 \bar{v}_0 \bar{z}$ is a symmetry for $y = x_1 \oplus x_0$ and it is symmetric to $\bar{v}_0 \bar{v}_1 z$. Note that the set of configurations $\{c \mid f^c = f\}$ forms a group, i.e., among other properties: $(f^{c_1} = f) \wedge (f^{c_2} = f) \Rightarrow (f^{c_1})^{c_2} = f$.

Definition 3 *Two functions f_1 and f_2 are* NPN-equivalent*,* (equivalent *for short) if $\exists c. f_1^c = f_2$. By this definition, a set of equivalent functions form an* equivalence class*.*

Given a n-input function f, a canonicizer M determines a configuration c which maps f to a unique representative $M(f) = f^c$ of the equivalence class. Therefore, for any two functions f_1 and f_2, $M(f_1) = M(f_2)$ iff f_1 and f_2 are equivalent.

To denote the unique representative, we use a 2^n-bit string listing the output values of f^c for each input assignment.[2] For example, suppose configuration $v_2 v_1 v_0 z$ is applied to f. The representation would be denoted by the values $f(0,0,0)f(0,0,1)f(0,1,0)\ldots f(1,1,1)$, for brevity written as $m_0 m_1 m_2 \ldots m_7$. A different configuration for the inputs would

[2] We restrict ourselves to this definition for a consistent presentation. For a function with m minterms, another representation would be a $m \times n$-bit table listing the minterms of f^c [10, 5, 3]. Similarly, a BDD with a fixed variable order could be used.

Configuration c	Minterm Permutation	Bit String for f^c
$v_2v_1v_0z$	$m_0m_1m_2m_3m_4m_5m_6m_7$	11101111
$v_1v_2v_0z$	$m_0m_1m_4m_5m_2m_3m_6m_7$	11111011
$v_2v_0v_1z$	$m_0m_2m_1m_3m_4m_6m_5m_7$	11101111
$v_1v_0v_2z$	$m_0m_2m_4m_6m_1m_3m_5m_7$	11111011
$v_0v_2v_1z$	$m_0m_4m_1m_5m_2m_6m_3m_7$	11111101
$v_0v_1v_2z$	$m_0m_4m_2m_6m_1m_5m_3m_7$	11111101
:	:	:
$\bar{v}_0v_2v_1z$	$m_4m_0m_5m_1m_6m_2m_7m_3$	**11111110**
$\bar{v}_0v_1v_2z$	$m_4m_0m_6m_2m_5m_1m_7m_3$	**11111110**
:	:	:

Figure 1: Configurations of function $f(x_2, x_1, x_0) = x_2 + \bar{x}_1 + \bar{x}_0$.

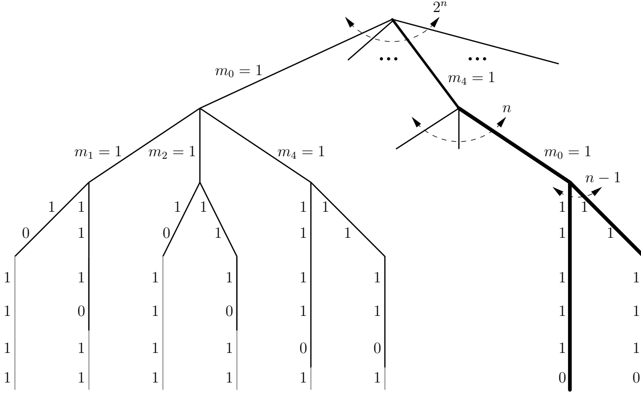

Figure 2: Search tree for the configurations in Figure 1.

permute the 2^n-bit string; e.g., the configuration $\bar{v}_0v_2v_1z$ produces $f(\bar{0}, 0, 0)f(\bar{1}, 0, 0)f(\bar{0}, 0, 1) \ldots = m_4m_0m_5 \ldots$. To find the representative for a particular f (i.e., an arrangement of the m_i), M chooses a configuration that is maximal for f under some ordering criterion $\preceq$.

Note that M must enumerate, explicitly or implicitly, all symmetric configurations. Suppose c is maximal. For M to be sound, any other maximal configuration must be symmetric to c. For M to be complete, it must prove $\forall c'.c' \preceq c$. In other words, because the set of maximal configurations is a coset of the function's symmetry group, the symmetry detection problem can be reduced to canonical form computation.

2.1 (NPN) Base Algorithm: Bit-string Comparison

To find a canonical representative for function f the authors of [4, 5, 6] compare the different configurations c by lexicographic comparison of the bit-string representations of their f^c and choose the one with the largest value. A naive method could compute the bit strings for all $2^{n+1}n!$ configurations and choose one of the maximum bit strings as representative (shown in bold in Figure 1). Note that the two maximal configurations are indeed symmetric.

Instead of exhaustively enumerating all configurations, recent works use a systematic tree-based search [4] as depicted in Figure 2, where paths of the tree (left to right) correspond to rows of Figure 1 (top to bottom). The top of the search tree has 2^n branches enumerating all possible input phase assignments. The subsequent branching levels enumerate places for the variables; i.e., the $(k+1)$th level assigns an x for v_k, uniquely determining the next 2^k labels for the m_i. Note that while the phases are determined at the top level, they are applied *after* permutation. Thus, the jth subtree at the root is the same as the first subtree, where the m_i are replaced with $m_{i \oplus j}$ (bitwise-XOR of i and j).

Searching for a maximum bit string can now be performed in breadth-first-search (BFS) order where branches that encounter a $m = 0$ can be pruned as long as there is at least one other active branch with a $m = 1$. The black part of the tree in Figure 2 shows the active part of the search for the given example, whereas the gray part was pruned. The bold branches lead to the maximum configurations.

The authors of [4] perform this tree-based search explicitly, whereas in [6] a faster implementation is suggested that precomputes the bit indices (m_i labels) of the entire search tree.[3,4] The authors of [5] use a different representation to compress the search by ignoring input combinations for which the function evaluates to 0, but at the cost of higher complexity at each step.

We would like to point out that these algorithms are correct, but solve a more difficult problem than necessary. The strict definition of canonicity based on the lexicographical order of the bit strings leads to a relatively complex search problem. For example, to cope with phase assignment, the tree-based search needs to enumerate an exponential number of phases before pruning can kick in. Later, we show that various simple criteria can dramatically reduce the number of candidates for the phase assignments and help prune the search of variable permutations.

2.2 (N) Satisfy Counts

In [4] and [6], input and output negations are incorporated into the overall search, requiring to consider all $2^{n+1}n!$ configurations of a single output function. However, for the vast majority of functions, the phase assignment search can be reduced dramatically when satisfy counts of f and its cofactors are considered [10, 11, 7].

Definition 4 *The* satisfy count *of a function f, denoted $|f|$, is the number of minterms for which f evaluates to* 1.

For the output phase assignment of a given function f, we consider $|f|$ and $|\bar{f}|$. If $|f| > |\bar{f}|$, then we apply the remaining search algorithm to $\bar{f}$, and vice versa. Only in the rare instance that $|f| = |\bar{f}|$, the search must consider both cases and choose the larger of their respective bit strings.

The input phase assignment problem is solved similarly using the cofactors of the function. With this scheme, if $|f|$ is odd, only one phase assignment of the inputs and output needs to be considered, rather than 2^{n+1}. Note that this selection criterion for a canonical configuration is different than the one used in Section 2.1. Instead of choosing a configuration which is maximal according to the lexicographical order of the bit strings, in this case a configuration is maximal only if its satisfy counts are minimal.

2.3 (P) Row/Column Sums

The authors of [3] use a different set of criteria to compare configurations of a function. They use a truth table to represent f, where all rows that correspond to a 0 output are purged.For the remainder of this paper, we refer to this representation as an *implicant table*. The proposed algorithm iteratively partitions the columns and rows of the table by using as discriminating factor the number of "1" entries in the column and row parts, respectively; this is the same as the procedure described in [12][5].

[3]As hinted earlier, we can precompute the first top-level subtree corresponding to permutations only, and compute the rest of the tree on-the-fly with XOR. This increases the algorithm's capacity and improves cache behavior.

[4]In addition, the tree can be folded in half vertically to save memory, because for any $i < 2^{n-1}$, $(2^n - 1) - i = (2^n - 1) \oplus i$ (personal communication, Zile Wei). This scheme implicitly finds symmetries when paths reconverge.

[5]The maximal bit-string of [4, 5, 6] w.r.t. permutation is equivalent to the maximal adjacency matrix of [12] for a graph consisting of nodes for inputs and implicants.

More recently, the authors of [7] use column sums of various cofactors for distinguishing variables, instead of row partitions, with impressive results. We note that using BDDs makes their algorithm scalable, but slow for small to mid-sized problems, and the use of depth-first search may require excessive searching to find the maximal configuration and prove its maximality.

3 Configuration Refinement

In this section, we show how the criteria used in previous work can be combined with new ideas to efficiently compute an NPN canonical form. For our purposes, we choose criteria which allow us to select phase assignments and permutations somewhat independently. Many of these concepts have also been developed to perform signature computation for distinguishing functions [13, 11]. In [6] it was suggested to apply signature computation as a filter to avoid canonical form computation when no match is possible. However, the proposed scheme does not utilize the signature information in the actual search procedure. As one of the new concepts presented here, we extend the use of signature computation to simplify the canonization step itself.

Using the terminology of [2], we first construct a set of "semi-canonical" configuration candidates and then use one of the previously described algorithms to choose a canonical form from this set. Key to the efficiency of our algorithm is the use of a sequence of criteria to maximally prune the candidates at each step before applying the bit string comparison to break ties. Furthermore, each criterion provides invariants that can be used in later stages. Our overall algorithm is outlined in Algorithm 1. Each step refers to the section that provides a more detailed description. Since the search algorithm of Section 2.1 uses phases *last* to compute the m_i, f "sees" phases *first*, and that is what we start refinement with. This corresponds to a reverse mapping of the x's to the v' which will be used to denote configurations for the rest of the paper.

Note that due to the refinement steps, the selected "maximal" configuration no longer has in the largest bit string over all configurations. However, canonicity is preserved because all functions from each equivalence class are treated equally.

Algorithm 1 Matching algorithm

1. Compute satisfy count for each cofactor
2. Constrain phases (§3.1)
3. Constrain permutations (§3.2)
4. Compress implicant table along unate variables (§3.3)
5. Repeat steps 1-3
6. Prune symmetric configurations (§4)
7. Run BFS canonicizer over remaining configurations to obtain canonical form (§2)

3.1 Negation

The first step of refinement attempts to reduce the number of phase assignments that need to be considered in the final BFS. The authors of [2] propose an algorithm to determine a phase canonical form. However, their solution depends on the permutation of variables and thus does not allow an independent pruning of phase assignments and permutations. Instead of examining variables individually (thus being permutation-dependent), we propose to use the collection (a.k.a. bag) of row sums obtained by summing across each row of the implicant table as a conceptual tool to distinguish between different phase assignments, and select a "maximal" one as canonical. Note that *bags* are independent of row and column order and thus allow the pruning of phase assignments without considering permutation. This is because functions with different bags cannot be made functionally equivalent by only permuting inputs.

x_3	x_2	x_1	x_0	Σ_r
0	1	0	1	2
0	0	0	0	0
0	0	1	0	1
0	0	1	1	2

(a)

Phase Assignment	$\sum(\Sigma_r^2)$
$x_3x_2x_1x_0y$	**9**
$x_3x_2\bar{x}_1x_0y$	11
$x_3x_2\bar{x}_1\bar{x}_0y$	**9**

(b)

Figure 3: Phase assignments for $f = \bar{x}_3(x_2\bar{x}_1x_0 + \bar{x}_2(\bar{x}_0 + x_1))$

Example The function $\bar{x}_3(x_2\bar{x}_1x_0 + \bar{x}_2(\bar{x}_0 + x_1))$ has row sum bag $\{0, 1, 2, 2\}$, as shown in column Σ_r of Figure 3a. Applying phase assignment $x_3\bar{x}_2x_1x_0y$ results in a new function with different bag $\{1, 1, 2, 3\}$. On the other hand, applying phase $x_3x_2x_1x_0\bar{y}$ gives a third function $x_3 + \bar{x}_2\bar{x}_1x_0 + x_2(x_1 + \bar{x}_0)$ with bag $\{1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4\}$.

In the following, we describe two fast criteria for comparing row sum bags without actually computing the bags.

3.1.1 Satisfy Counts

The first criterion chooses phase assignments which minimize the sum of the elements of each row sum bag (where $5 < 8 < 27$ in the previous example). This sum is the number of 1's in the implicant table. Finding the phase assignment with the minimum sum of row sums is equivalent to minimizing the satisfy counts of the individual columns as described in Section 2.2. By using this algorithm, an explicit enumeration of bags for candidate phase assignments is unnecessary.

Minimization (versus maximization) of the sum of row sums is adopted to make the final BFS terminate more quickly. Recall from Section 2 that the BFS canonicizer on line 7 of Algorithm 1 searches for a maximal bit string. By reducing the number of 1s, we increase their distinguishing power, and encounter them earlier in the search, thus being able to prune subtrees more quickly.

For our running example, we prune the candidate phase assignments to the four which result in a sum of 5: $x_3x_2x_1x_0y$, $x_3x_2\bar{x}_1x_0y$, $x_3x_2x_1\bar{x}_0y$, and $x_3x_2\bar{x}_1\bar{x}_0y$.

3.1.2 Sum of Squared Row Sums

Before proceeding to the next step, we apply the top step of the BFS of Figure 2 which prunes the phase assignments. Recall that we prune branches that encounter a 0 if there are others that encounter a 1. In our example, we discard $x_3x_2x_1\bar{x}_0y$ and keep the others, because 0001 is not an implicant of f, while $0000, 0010, 0011$ are.

In the case that several input phase assignment candidates are still indistinguishable after applying the previous steps, we compare the sum of squares of the corresponding row sum sets.

Example (cont'd) While $x_3x_2x_1x_0y$ results in the bag $\{0, 1, 2, 2\}$, configuration $x_3x_2\bar{x}_1x_0y$ produces the bag $\{0, 1, 1, 3\}$. The simple sum of their row sums are equal. However, when we compare $0^2 + 1^2 + 2^2 + 2^2 = 9$ to $0^2 + 1^2 + 1^2 + 3^2 = 11$ we can distinguish them. We favor phase assignment $x_3x_2x_1x_0y$ and discard $x_3x_2\bar{x}_1x_0y$. Again, the particular choice (9 versus 11) is a heuristic used to move 1s upwards in the search tree. The resulting sums for each phase assignment are shown in Figure 3b. After applying these criteria to our example, only two phase assignments remain for the rest of the algorithm. Note that these two can be made equivalent: $f = f^{x_3x_2\bar{x}_0\bar{x}_1y}$.

3.2 Permutation

The matchers described in [4, 5, 6] search over all permutations in order to find a canonical representation. In this section we describe how to reduce the set of permutations explored, by excluding certain mappings between the v's and the x's. We later show how this is useful for pruning symmetries.

For pruning permutations, we partition the input variables of a function according the criteria given in Sections 3.2.1 and 3.2.2. We then permute variables only within a partition. Note that partitions must be sorted to maintain canonicity.

3.2.1 Satisfy counts

In [11] and [3], satisfy counts of cofactors are used to partition the input variables of a function. These correspond to 1st order Walsh-Hadamard spectral coefficients [14, 15]. For two inputs x_i and x_j, if $|f_{x_i}| \neq |f_{x_j}|$ then x_i and x_j cannot be swapped without changing f. Similarly, for two functions f and f^c, if $|f_{x_i}| \neq |f^c_{v_j}|$ then f cannot be transformed into f^c by swapping x_i and v_j. This leads to a partitioning criteria for the inputs variables where the satisfy counts for the cofactors with respect to all variables of a partition are equal. Due to the phase refinement steps performed previously, input x_i may be negated later only if $|f_{x_i}| = |f_{\bar{x}_i}|$. Thus, the satisfy counts used are invariant.

Example Suppose we wish to find the canonical form of the function $f = \bar{x}_4(\bar{x}_3+\bar{x}_2+\bar{x}_1\bar{x}_0)$ using a search-based algorithm. Variables x_3 and x_2 are symmetric, as are x_1 and x_0; thus it is expected that four permutations result in the canonical form.

The satisfy counts are $|f_{x_2}| = |f_{x_3}| = 5$, $|f_{x_0}| = |f_{x_1}| = 6$, $|f_{x_4}| = 13$ which induce three partitions. Since variables may only be swapped within a partition, at most the four mappings between $\langle\{v_4, v_3\}, \{v_2, v_1\}, \{v_0\}\rangle$ and $\langle\{x_2, x_3\}, \{x_1, x_0\}, \{x_4\}\rangle$ will be explored.

3.2.2 Unateness

A second criterion for distinguishing variables is unateness, as suggested in [1]. A variable that is unate can never be swapped with a variable that is binate. This information can be applied to further refine the variable partitions defined previously.

3.3 Unateness Compression

Given a function that is negatively (positively) unate over all its inputs, the authors of [5] compress the implicant table by replacing it with a prime cover. For a function that contains a mixture of variable types, the authors used the uncompressed table.

In the following we propose an extension of this concept which compresses the implicant table with respect to all unate variables even if binate variables are present. Suppose f has k binate variables. We replace each of the 2^k cofactors of f w.r.t. the binate variables with a prime cover for the unate variables. Our implementation uses a bit string representation of the function for quick lookups when incrementally compressing a list of implicants. To maintain correctness while finding a maximal configuration, we must order the variables by their unateness which, as a side effect, further speeds up the search.

Compressing the implicant table is critical for algorithms that are polynomial in the table size. For a function that is binate in only one variable, this compression results in an exponential reduction of the runtime. In addition, compressing the table allows for additional opportunities to refine the set of configurations as demonstrated in the following example.

(a)

x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	0	1	0	1	1
1	1	1	1	1	1
1	0	0	1	1	0
$\langle b,2,?\rangle$	$\langle b,3,?\rangle$	$\langle b,4,?\rangle$	$\langle b,3,?\rangle$	$\langle p,5,?\rangle$	$\langle b,5,?\rangle$

(b)

x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	1	1
0	1	1	0	-	1
0	0	1	0	1	1
1	1	1	1	1	1
1	0	0	1	1	0
$\langle b,2,2\rangle$	$\langle b,3,2\rangle$	$\langle b,4,3\rangle$	$\langle b,3,3\rangle$	$\langle p,5,4\rangle$	$\langle b,5,4\rangle$

(c)

x_1	x_5	x_4	x_2	x_3	x_0
1	0	0	1	0	1
-	0	1	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1
1	1	0	1	0	0
$\langle p,5,4\rangle$	$\langle b,2,2\rangle$	$\langle b,3,2\rangle$	$\langle b,3,3\rangle$	$\langle b,4,3\rangle$	$\langle b,5,4\rangle$

Figure 4: Refining variable partitions of $f = x_5\bar{x}_4\bar{x}_3x_2x_1\bar{x}_0 + (\bar{x}_5\bar{x}_4\bar{x}_3 + x_5x_4x_3)x_2x_1x_0 + (x_4 + x_1)\bar{x}_5x_3\bar{x}_2x_0$ using an implicant table.

Example Consider the variable permutation for the function taken from [5] whose implicant table is shown in Figure 4a. The canonicizing algorithm of [5] would require a branch and bound evaluation of different configurations as shown in Figure 4 of that paper. We show that no branching is necessary; a maximal permutation of variables is found using our refinement technique alone. Below each column of the tables in Figure 4 is a triplet denoting the type of variable (binate/positive unate), and column sums. Variables x_1 and x_0 have identical column sum, but may be distinguished by their unateness. During a first partitioning step all variables but x_2 and x_4 are distinguishable. Next, one of the rows is removed by compression, and the column sums recomputed, where the dash "-" is counted as a zero. After that x_4 and x_2 are distinguishable (b). The variables are then partitioned by their associated triplets (c). Note that the original column sums are necessary to distinguish between x_2 and x_3 (and x_5 and x_4). Since the triplets are distinct, a maximal permutation $v_4v_3v_1v_2v_5v_0$ is determined.

4 Pruning Symmetries

Definition 5 *Configurations can be* composed*. Given configurations c_1 and c_2, we define their composition $c_1 \circ c_2$ in the usual manner, where $f^{c_1 \circ c_2} := (f^{c_1})^{c_2}$. For any c, $c_I \circ c = c$.*

As stated earlier, all maximal configurations are symmetric. If we know some of a function's symmetries *a priori*, we only need to search over the asymmetric configurations (also known as coset representatives) for the maximal configuration. Suppose $f^c = f$, $c \circ c_a = c_b$ and $c_a \neq c_b$. Therefore $f^{c_a} = f^{c_b}$, and c_a is maximal iff c_b is as well. Thus one of c_a or c_b can be safely ignored. The authors of [4] and [5] use a restricted set of symmetries to reduce the search space. In our work, we expand on the types of symmetries used for this purpose including ones between different input phase assignments and between different input permutations.

Since symmetries are not known *a priori*, they must be found by explicitly checking, with the assumption that finding them explicitly beforehand is faster than letting the Boolean matcher find them implicitly afterwards. Another option is to perform symmetry checks *during* the search with the help of the Boolean matcher. Finding symmetries early prunes more configurations, however the tradeoff is that there are also more symmetry candidates to consider. In this section, we explore the utility of performing symmetry checks at various stages of the breadth-first-search.

Checking whether a configuration c is symmetric in f is fast. Given f as a bit string and a list of minterms, we can apply c to the list. We now have a bit string representing f and a list of minterms representing f^c, and check for equality between the two. This procedure is faster than sorting as performed in [5]. If c is indeed symmetric, we remove from consideration all c_b where $c_a \circ c = c_b$.

4.1 Pre-search Symmetry Pruning

After applying the refinement techniques described in Section 3, more than one phase assignment or permutation may remain. In this case, we check for the simplest symmetries before proceeding to BFS: phase assignment and swapping of pairs of variables.

Example Consider $f = x_2 \oplus x_1 \oplus x_0$, which requires us to check 8 phase assignments even after applying the techniques of Section 3.1. We find that $c = x_2\bar{x}_1\bar{x}_0 y$ is a symmetry. Then we can discard, among others, phase assignment $\bar{x}_2 x_1 \bar{x}_0 y \circ c = \bar{x}_2\bar{x}_1 x_0 y$. We also discard c, since $c_I \circ c = c$. In this case, a single explicit symmetry check saves the BFS from exploring four sets of paths.

Note that in the case of XOR, three configurations combining permutation and phase assignment are sufficient to describe all symmetries. However, we look for simple symmetries (requiring more symmetry checks) to simplify the implementation.

Example Suppose $f = \bar{x}_3\bar{x}_2 + \bar{x}_1\bar{x}_0$; variables x_3, x_2, x_1, and x_0 lie in the same partition, meaning 24 permutations are candidates. Furthermore, swapping x_3 with x_2 leaves the function unchanged, as well as swapping x_1 with x_0. We can impose an arbitrary ordering between x_3 and x_2, and x_1 and x_0, resulting in six permutations. In addition, we can further impose a new condition to keep symmetric variables adjacent. In this case, we need to only consider the two permutations $x_3x_2x_1x_0$ and $x_1x_0x_3x_2$. This convention allows us to find other symmetries more easily.

4.2 In-search Symmetry Pruning

More complex symmetries may permute several variables simultaneously. For example, in function $f = \bar{x}_3\bar{x}_1 + \bar{x}_2\bar{x}_0 + \bar{x}_3\bar{x}_2$, x_3 and x_2 can be swapped while swapping x_1 and x_0 without changing the function. Again, we can use this information to reduce the search space for the BFS. However, there is a large number of options to permute several variables simultaneously, and it is unknown which swap attempts are more likely to result in symmetries.

The BFS algorithm itself provides a clue: the search algorithm, by discarding configurations that are asymmetric to the maximal configuration, provides a good set of candidates to check for symmetries. For two active paths in a branch and bound tree relating to two different permutations, we can swap the corresponding variables and check for symmetry. The authors of [5] propose to check for a limited type of symmetry at every step of the algorithm. In contrast, we propose to check for a more general set of symmetries at opportune moments, as shown in the next example.

Example Suppose $f = \bar{x}_3\bar{x}_1 + \bar{x}_2\bar{x}_0 + \bar{x}_3\bar{x}_2$. From satisfy counts, the variables are partitioned so that that only four mappings between $\langle\{v_3, v_2\}, \{v_1, v_0\}\rangle$ and $\langle\{x_3, x_2\}, \{x_1, x_0\}\rangle$ need to be considered. The branch and bound tree corresponding to these permutations is shown in Figure 5; other permutations are omitted. The minterms are converted to a unate cover, and columns for x_1 and x_0 are complemented to minimize 1s, resulting in (1001,0110,0000). The tree is labeled with the bit string representation for each permutation. In addition, we label the tree with ⓐ, ⓑ, and ⓒ to refer to three different stages of the search.

We can compare the partial permutations at any of these stages to check for symmetries. However, points ⓐ and ⓑ are ineffective. Point ⓐ will attempt to swap x_1 and x_0, which was already checked for. Point ⓑ, which is the end of a partition, checks for the rare rotational symmetry [16].

ⓒ is a good point in the search to check for hierarchical symmetry. The left path chose $\{x_3\}$ for $\{v_2\}$, and the right path chose $\{x_2\}$ for $\{v_2\}$. Together, they form a subpartition of the current

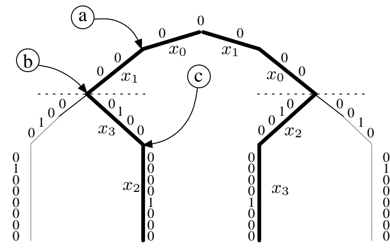

Figure 5: Detecting general variable symmetry.

partition $\{x_3, x_2\}$. If we reach this point in the search, it is likely that a symmetry exists.

Using this scheme, we avoided explicitly checking whether, for example, the two leftmost paths in the search tree correspond to symmetric configurations.

5 Experimental Results

We implemented the described enhancements using a Boolean matcher based on branch-and-bound. As a baseline, we use an implementation of the algorithm presented in [6] with two improvements. First, the bit index table is computed for permutations only, as mentioned in §2.1. Second, the output phase problem is solved as described in §2.2. These changes allow us to examine 9-input functions, but are minor, so we still refer to this matcher as "DS".

The functions provided to the matchers were obtained by decomposing the circuit `pj2.blif` into AND2 gates and inverters, and generating cuts according to the recursive definition in [17]. This circuit is of significant size, and thus provides a good source for a wide variety of patterns that occur in practical synthesis flows. We classify the functions based on input size, and run the two matching algorithms on approximately 20,000 instances of each class of functions on a workstation with 512MB of memory and a 3.0GHz Pentium 4 processor.

5.1 Runtime Dependence on Input Size

The runtimes for the matchers are shown in Table 1. When practical, the matching routines are averaged over 50 runs to amortize I/O cost. Column n denotes the number of inputs to each function.

The column "DS" refers to matcher "DS". The column "DS++" refers to the same matcher with the rest of the improvements described in this paper.

The column denoted by "Speedup" is a simple ratio of runtimes. To measure how the scalability of both algorithms compare, we report in the next column the speedup scaled by the number of input configurations. This value would be constant when comparing worst-case search with an oracle.

For all input sizes, matcher "DS++" greatly outperforms "DS". Symmetries and asymmetries are found quickly, so that the last column is roughly constant. The indicated runtime of "DS++" is exponential in n — this is to be expected, because the bit strings used as input and output are of length 2^n.

Note that the results reported here for "DS" greatly differ from those given in previous work — we will explain this during the discussion of the next set of experiments.

5.2 Runtime Dependence on Input Type

The next set of experiments illustrates the dependence of the runtime on the input types. We run the two matchers on three sets of 7-input functions: random functions, the functions from the previous experiment, and a 7-input XOR. The results are shown in Table 2. For random functions, "DS" performs fairly well. Random

	Average Runtime (μs)			
n	DS	DS++	Speedup (x)	Speedup$/n!2^n$
4	0.61	0.42	1.45	3.7e-3
5	8.57	1.34	6.40	1.7e-3
6	98.8	2.31	4.28e1	0.9e-3
7	3.12E3	3.69	8.46e2	1.3e-3
8	1.10E5	7.58	1.45e4	1.4e-3
9	6.26E6	15.0	4.17e5	2.2e-3

Table 1: Runtime of matching algorithms for different input sizes.

	Average Runtime (μs)		
Function	DS	DS++	Speedup$/7!2^7$
Random 7 input	88.2	5.2	2.6e-5
Typical 7 input	3.12E3	3.69	1.3e-3
XOR7	309E6	32.6	15

Table 2: Runtimes for extreme cases of 7-input functions.

functions (used in [6]) typically contain no variable symmetries[6], which assures that "DS" will not experience worst case behavior. On the other hand, the XOR contains many symmetries, and "DS" required five minutes to compute a canonical form.

5.3 Symmetry Finding

As discussed earlier, symmetry detection can be performed using Boolean matching or by finding graph automorphism. In order to confirm the effectiveness of our approach, we use our Boolean matcher and the graph automorphism package `saucy` [18] to detect symmetries. We stress that this is not intended as a thorough comparison of the merits of each approach — `saucy` is able to find automorphisms in arbitrary graphs, not just graphs representing functions. On the other hand, `saucy` was developed primarily to find symmetries in propositional formulas (i.e., functions).

In order to properly discount time spent in setup routines such as memory allocation, we perform these experiments on a Pentium III 800MHz computer with 256MB of RAM, running Linux 2.6 with the `perfctr` library. Our matcher and `saucy` are provided with the same set of functions used previously.

As the results in Table 3 show, our matcher is highly competitive, outperforming `saucy` on every benchmark. Our matcher precomputes search trees as in [6], meaning it has a somewhat unfair advantage, and is thus limited to functions with few inputs. However, the difference in runtime is rather large, and switching to a more scalable implementation of branch and bound [5] may not adversely affect performance too greatly. Overall, the ratio of runtime between our matcher and `saucy` is roughly constant, meaning that both are equally effective in finding symmetries.

6 Conclusions

In this paper we outlined several enhancements for Boolean matchers that are based on computing canonical representations. Rather than choosing a single criterion for canonicity, we define our canonical form in a way that applies a sequence of refinement steps with the goal to quickly prune the set of candidates to be further processed by a final "tie-breaker". In addition, we showed that different representations of a function can distinguish different properties of the function. We also outlined the connection between canonical form computation and symmetry detection, and showed that our matching algorithm is competitive with a state-of-the-art symmetry detector.

[6]For an arbitrary 7-input function, the probability that two particular variables are symmetric is $\frac{1}{2^{2^5}} = 2.3 \times 10^{-10}$.

	Average Runtime (cycles)		
n	Saucy	DS++	Speedup (x)
4	1.95e4	1.90e3	10.3
5	3.69e4	3.13e3	11.8
6	7.14e4	4.61e3	15.5
7	1.51e5	7.68e3	19.7
8	3.53e5	1.46e4	24.1
9	8.08e5	4.23e4	18.9

	Average Runtime (cycles)		
Function	Saucy	DS++	Speedup (x)
Random 7 input	1.09e5	2.91e4	3.73
Typical 7 input	1.51e5	7.68e3	19.7
XOR7	1.10e6	3.93e4	28.2

Table 3: Results for symmetry detection.

References

[1] F. Mailhot and G. D. Micheli, "Technology mapping using Boolean matching and don't care sets," in *European Design Automation Conference*, pp. 212–216, 1990.

[2] J. R. Burch and D. E. Long, "Efficient Boolean function matching," in *International Conference on CAD*, pp. 408–411, 1992.

[3] Q. Wu, C. Y. R. Chen, and J. M. Acken, "Efficient Boolean matching algorithm for cell libraries," in *International Conference on Computer Design*, pp. 36–39, 1994.

[4] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," in *Design Automation Conference*, 1998.

[5] J. Ciric and C. Sechen, "Efficient canonical form for Boolean matching of complex functions in large libraries," *IEEE Transactions on CAD*, vol. 22, pp. 535–544, May 2003.

[6] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," in *Asia and South Pacific Design Automation Conference*, pp. 591–596, 2004.

[7] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification," in *Design Automation Conference*, pp. 379–384, 2005.

[8] S.-W. Jeong, T.-S. Kim, and F. Somenzi, "An efficient method for optimal BDD ordering computation," in *Proc. International Conference on VLSI and CAD*, November 1993.

[9] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," in *Computer Aided Verification (CAV'93)*, pp. 463–478, Springer-Verlag, 1993.

[10] D. B. Netherwood, "Logic matrices and the truth function problem," *Journal of The ACM*, vol. 6, pp. 405–414, July 1959.

[11] J. Mohnke and S. Malik, "Permutation and phase independent Boolean comparison," *Integration*, vol. 16, pp. 102–129, 1993.

[12] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[13] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *International Conference on Computer Design*, pp. 452–458, 1992.

[14] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. Academic Press, 1985.

[15] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," in *Design Automation Conference*, pp. 54–60, 1993.

[16] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," in *Asia and South Pacific Design Automation Conference*, pp. 459–464, 1995.

[17] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *International Symposium on FPGAs*, pp. 29–35, 1999.

[18] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting structure in symmetry detection for CNF," in *Design Automation Conference*, pp. 530–534, 2004.