A New Structural Pattern Matching Algorithm for Technology Mapping *

Min Zhao, Sachin S. Sapatnekar*
Advanced Tools, Motorola Inc., Austin, TX
*Dept. of ECE, University of Minnesota, Minneapolis, MN

ABSTRACT

In this paper, a new structural matching algorithm for technology mapping is proposed. The algorithm is based on a key observation that the matches for a node in a subject Boolean network are related to the matches for its children. The structural relationships between the library cells are modeled using a lookup table. The proposed method is fast, has low memory usage, and is easy to implement. Experimental results show speedups of 20× over Matsunaga's fast mapping approach, and orders of magnitude over SIS, with the same or slightly better results, and much lower memory utilization.

1. INTRODUCTION

Technology mapping is an important step of synthesis. One of the crucial tasks for technology mapping is the process of matching, which tries to determine which cells in the library may be used to implement a set of nodes in the *subject Boolean network*. There are two major approaches to solving the matching problem: structural matching and Boolean matching. In this paper, we propose a fast and simple structural matching algorithm for technology mapping.

Boolean matching algorithms are summarized in [1]. Structural matching algorithms were originally addressed in [2, 3, 4] and are summarized in [5, 6]. A straightforward method is to match each pattern at each node of the subject Boolean network. A more sophisticated approach is to formulate this matching problem into a string matching problem and to apply existing string matching algorithms, such as the Aho-Corasick algorithm [7], to solve the problem. In general, the subject Boolean networks and library cells are decomposed into the same set of the simple functions, called *base functions*. One of the main drawbacks of structural mapping is that the decomposition of a library cell into base functions is not unique. There could be numerous decomposed patterns for each cell, particularly for gates with a large number of inputs, and therefore a significant amount of the total computation during technology mapping is spent on pattern matching. With the recent trend of using larger complex gates in a library, the number of patterns increases dramatically and thus the matching procedure increasingly becomes computationally expensive. Recently, [8] proposed a pattern matching method that does not require patterns to be decomposed. The algorithm uses an implicit pattern matching procedure and speeds up the matching using isomorphism and containment relations between cells.

In this paper, a novel structural matching algorithm is proposed. The method is simple to implement and efficient in terms of speed and memory. The main contributions of this work are as follows.

- The algorithm relates the matching of a node to the matching of its children. Given a node in a subject network, the valid matches of the current node can be obtained from the valid matches of its children and the node type of the current node. This property can be used to reduce a large number of redundant computations that exist in the traditional structural matching methods. Our algorithm improves over the traditional method by making use of this property. Recently we have learnt from a reviewer that the similar idea was discovered independently in [9] for the general problem of tree pattern matching in computer science. However, we also handle technology mapping specific issues such as the associativity and commutativity of the logic operators. To the best of our knowledge, the idea of matching a node by using the matches its children has not found prior use in technology mapping according to the published literature.
- In our method, each canonical structure or substructure of a library cell that could be represented by a tree is abstracted into a unique index, called the *pattern index*. The structural relationships between these structures or substructures are modeled into a lookup table, called the *pattern table*. In the past, the relations between patterns have either been ignored, or formed into automata [3, 4, 5] or Contain-Relation-Graph [8]. On the other hand, our lookup table is based on a canonical representation of patterns. Therefore, the problem associated with the traditional approaches, where the large number of possible decomposed patterns results in a significant computation cost, is overcome by our approach.

Currently, our method is restricted to the library cells that

*This work was supported in part by the Semiconductor Research Corporation under contract 99-TJ-692

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

can be represented in the form of a tree structure and the cells that can only be represented by a DAG, such as EXOR and EXNOR, are not considered. The method for handling the general non-tree structures in DAG mapping using our method will be discussed in Section 6. In our implementation, the subject Boolean network and the library cells are decomposed into the base functions of AND/OR/NOT. First, we generate the pattern table from a given library; During the technology mapping procedure, we use this pattern table and perform the matching procedure.

2. PATTERN TABLE GENERATION

The task of pattern table generation is to explore the structural relationships between cells. Each canonical tree structure or substructure of a cell is abstracted into a unique pattern index and the relationships are modeled into the pattern table. The pattern table consists of four parts: AND table, OR table, *isGate* array and *isInvGate* array.

2.1 Pattern index assignment

Given an arbitrary cell that can be represented by a tree structure, a unique multiple-input AND/OR/NOT tree can be built. If there are inverting functions in the cell, the inverters will first be pushed to the root node and the remaining inverters that are not at the root node are then pushed to the leaf nodes. By specifying the additional rule that the type of a node (AND or OR) cannot be the same as the type of its parents, a one-to-one correspondence between a cell and its structural tree is formed. In our method, we assign a unique pattern index to each such a canonical structure or substructure of a tree. Each pattern index either represents a cell or a substructure of a tree corresponding to a cell. Our AND/OR representation is canonical only for a tree structure, which implies that all inputs to the gate must be logically independent. In case that the inputs are logically dependent, such as leaf-DAG[5], the structure is not, in general, canonical.

2.2 AND/OR table

The AND/OR tables constitute the main parts of the pattern table. They reflect the fact that each arbitrary tree structure can be built by applying the AND(OR) operation to two other tree structures. In the AND(OR) table, both rows and columns are composed of pattern indices. Each entry (i, j) in an AND(OR) table is the pattern index of the tree structure that can be built by applying the AND(OR) operation to the two tree structures represented by the pattern indices of row i and column j. If such a tree does not exist in any tree structure or substructure of the cells of the library, the entry is assigned an *invalid value*. Each table is an $M \times M$ two dimensional table, where M is number of the pattern indices. In the ensuing discussion, [x,y,z] will be used to represent an entry of the pattern table, where x is a binary variable that shows whether the the entry belongs to the AND table or the OR table, and y and z are pattern indices, which define the position of the entry. Physically, entry [x,y,z] corresponds to a tree whose root node's type is x and whose two children are y and z. The pattern table is symmetric due to the symmetric property of the AND and OR operators.

An example of an AND/OR table is shown in Figure 1, where pattern index n1(n2) represents the left(right) tree structure. The composite tree structure is assigned to pattern index n3. Since the AND of tree n1 and tree n2 creates the tree n3, the entry [AND,n1,n2] is set to n3. Entry [AND,n2,n2] is an invalid value (denoted as "-" in this paper), since the structure whose root node is AND and two children are n2 does not exist as a cell or as a substructure of any cell in the library.

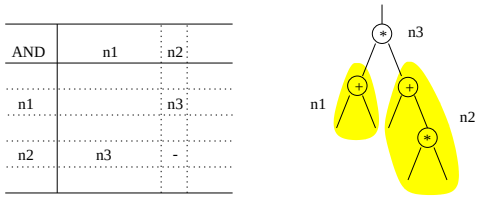


Figure 1: An example of entries of a pattern table

2.3 Decomposition

Although the AND(OR) table is a two-dimensional table, the canonical tree structures consist of multiple-input nodes. Therefore, the root node of the tree we are considering must be decomposed into two-input nodes to be fit into this AND(OR) table. Several decomposition schemes are possible for a given multiple-input node. In our pattern table generation algorithm, for each canonical tree labeled by a pattern index, all decomposition schemes of the root node of the tree are enumerated and the corresponding elements are entered into the AND/OR table.

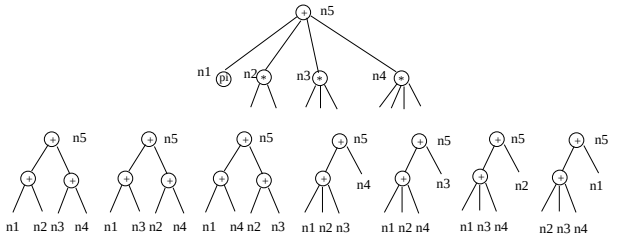


Figure 2: An example of decomposition

This idea can be illustrated by an example in Figure 2, where all of the decomposed configurations of tree n5 are enumerated. For each decomposed tree, the pattern indices representing the left and right children are identified, and then the corresponding entries in the OR table are filled with the pattern index of the current tree, n5. In our example, there are 7 different decomposition schemes for the root node of n5, and therefore there are 7 entries filled with n5 in OR table. Note that in the case where two or more decomposed structures of a given pattern index are isomorphic, the number of entries would be smaller.

2.4 The other parts of the pattern table

The *isGate* and *isInvGate* arrays are used to reflect the corresponding relations between a tree structure with a cell. In general, a library may not be a complete set. As a result, some of the tree structures identified above may merely be substructures used to form the other trees but may not be cells in the library. The *isGate* array is an M-dimensional vector that is used to indicate whether the pattern index representing a tree structure is a cell of a library or not.

As stated earlier, the inverters in the tree structure are pushed to the root or the leaf nodes of the tree. The M-

dimensional *isInvGate* vector is used to indicate whether there is an inverter at the root of a tree or not.

2.5 The algorithm for pattern table generation

The pattern table generation algorithm is shown in Figure 3. In this algorithm, all the cells in a library are considered one by one, and the corresponding entries in the pattern table are filled. Each tree structure appearing in the library is assigned a pattern index. The *Look_up* procedure, described in Figure 4, returns a pattern index for a given input tree structure, and fills in the entries of the AND/OR table. The pattern index of a leaf node is denoted as $leaf\_index$. A tree with an inverter before a leaf node is assigned the pattern index inv_leaf_index. These are the only two reserved patterns, and all other patterns are detected by the algorithm. For each two-input decomposition of the tree, the procedure is invoked recursively to fill in the pattern table. If tree T has already been visited before, the branch at line 6 of Figure 4 is taken and the field $Pattern\_index(T)$ is returned. If a tree isomorphic to tree T has been visited somewhere else, only one decomposition is performed to find out the pattern index and the branch at line 12-13 of Figure 4 is taken. Otherwise, a new pattern index will be assigned to the tree T and the enumerations of decomposition schemes between Line 8-14 of Figure 4 will be performed.

Pattern_Table_Generation Algorithm	
Input:	A library
Output:	AND table; OR table; isGate array; isInvGate array
1.	Initialize each entry of the pattern table with invalid value
2.	For each cell of library
3.	Form a multiple-input AND/OR/NOT tree T
4.	pattern_index = $Look\_up$(T)
5.	isGate[pattern_index] = 1
6.	If there is inverter at the root node of T
7.	isInvGate[pattern_index]=1

Figure 3: The pattern table generation algorithm

Procedure *Look_up(tree T)*	
Input:	An AND/OR/NOT tree
Output:	Pattern index of the tree; Entries in the pattern table
Global variable:	newIndex(store number of the patterns)
1.	If T is a leaf
2.	return leaf_index
3.	If T is an inverter before a leaf
4.	return inv_leaf_index
5.	If Pattern_index(T) is a valid value
6.	return Pattern_index(T)
7.	NodeType = type of root node of tree T
8.	For each two-input decomposition of root node of tree T, T′
9.	IndexL= $Look\_up$(leftchild_of_T′)
10.	IndexR= $Look\_up$(rightchild_of_T′)
11.	If [NodeType,IndexL,IndexR] is a valid value
12.	Pattern_index(T)=[NodeType,IndexL,IndexR]
13.	Return Pattern_index(T)
14.	Pattern_index(T)=[NodeType,IndexL,IndexR]=newIndex
15.	Increment newIndex
16.	Return Pattern_index(T)

Figure 4: The pattern index lookup procedure

2.6 An example of pattern table generation

The pattern table generation procedure is illustrated in Figure 5. In Figure 5(a), an example library composed of four cells, is shown and the resulting pattern table is shown in Figure 5(b). $leaf\_index$ is denoted by "0". Here, each pattern except for pattern 4 corresponds to a cell of the library. Therefore, all elements of $isGate$ are set to "1" except at index 4. All of the gates here are inverting complex gates and thus the corresponding elements of array $isInvGate$ are set to "1". For instance, Pattern 4 can correspond to the configurations of the AND of trees 1 and 2, as well as the AND of trees 3 and 0. Therefore [AND,1,2], [AND,2,1], [AND,3,0] and [AND,0,3] are filled with index 4. In reality, since the pattern table is symmetric, it is sufficient to store only the upper or lower triangle of the matrix.

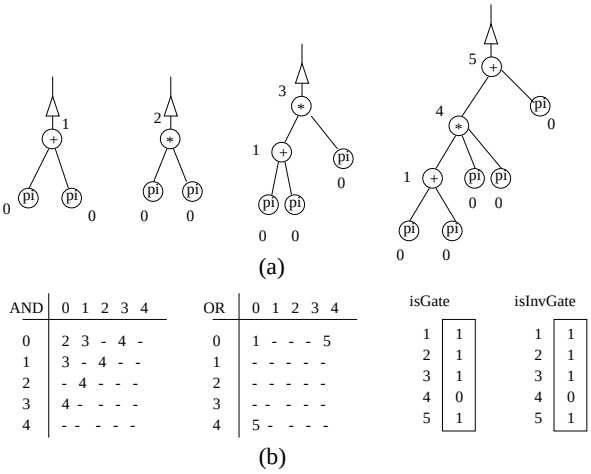


Figure 5: An example of pattern table generation

3. MATCHING

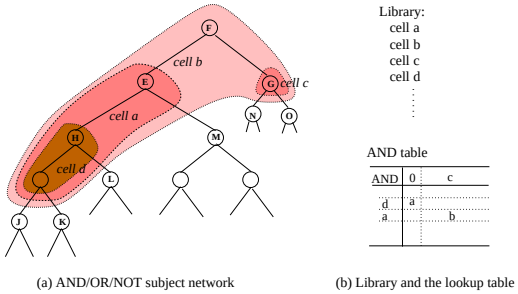
In Section 2, we have addressed a preprocessing procedure where the structural relations between cells are derived and stored in the pattern table. In this section, we explore the relationship between matching of a node of the subject Boolean network and matching of its children. A matching method utilizing these two relationships is proposed.

3.1 Motivations

In the traditional matching method, each node is matched independently, which may cause redundant operations. In our approach, the matching of a node in the subject Boolean network relates to the matching of its children. To illustrate this, we consider the AND/OR/NOT subject network shown in Figure 6(a), and it is to be mapped to the library that is partially described in Figure 6(b), consisting of cells a, b, c, d, etc. In our notation, a, b, c, d also refers to the pattern indices of the cells. The pattern table shows the structural relations between these cells. Since entry [AND,a,c] is filled with pattern index b, we see that the AND operator applied to cells a and c constitutes the cell b. Suppose cell a is a match for node E and cell b is a match for node F. In the traditional method, nodes E and F are matched independently: to match cell b against node F, the matching procedure must traverse from the root node all the way to the leaf nodes of cell b and the corresponding part of the subject network. However, we can see that a part of this procedure, where node E and its children are visited, has already been performed while matching node E to the tree structure a, and is redundant. Similarly, the operation that matches node H against cell d is repeated during the matching procedure for all of nodes H, E and F.

Our approach shares these operations by considering the matching relationships between a node and its children. At the time we match subtrees rooted at node F, we know that

cell a is a valid match of node E and cell c is a valid match of the other child, G, and type of node F is AND. Therefore, we can check whether cell b is a valid match of node F by merely looking up the entry [AND,a,c] in the pattern table, an operation whose computation cost is constant.



(a) AND/OR/NOT subject network (b) Library and the lookup table

Figure 6: Matching of a node and its children

3.2 Matching algorithm outline

The *match set* of a node in a subject network is composed of all of the pattern indices whose corresponding tree structures are isomorphic to the segment of the subject Boolean network rooted at this node. Given the node type of the current node of the subject Boolean network, NT, the match set of the left child L and the match set of the right child R, each element of the cross-product $L \times R$, (l, r), is checked against the corresponding entry of the pattern table, $[NT, l, r]$. If $[NT, l, r]$ returns a valid pattern, then that pattern corresponds to one match of the current node. In addition, each node in a subject network may be a leaf of a cell. Therefore, the pattern $leaf\_index$ is included in the match set of each node. The matching procedure for a node in a subject Boolean network is described in Figure 7.

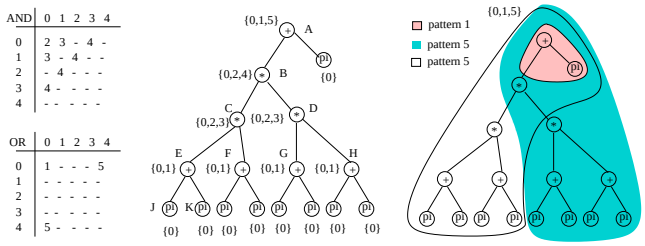
Procedure Matching(current_node N)
Input: Match sets of N's children; The pattern table
Output: Match set of N
1. NT = node type of N
2. For each match of the left child, l
3. For each match of the right child, r
4. k = $[NT, l, r]$
5. If (k is valid value)
6. Insert k into the match set of N
7. If isGate[k]==1
8. $cell\_handling(k)$
9. Insert leaf_index, inv_leaf_index into the match set of N

Figure 7: The matching procedure at each node

The match set of each node is used to form the match set of its parents. The match set of each node consists of two types of patterns: patterns representing a cell, and patterns representing a substructure of a cell. If a pattern corresponds to a cell, then the condition on line 7 of Figure 7 is satisfied, and the procedure that handles a matched cell will be executed. The *cell_handling* procedure will typically involve cost calculation, pruning, etc. If a pattern is a substructure of a cell, then it will be retained in the match set since it may be useful for the matching step for its parent node.

3.3 An example of the matching procedure

The matching procedure is illustrated in Figure 8. This example uses the same library as the example in Section 2.6, and the pattern table is shown in Figure 8(a). The subject AND/OR/NOT network is shown in Figure 8(b). The elements within curly braces next to each node constitute the match set of the node. All primary input are initialized with the match set $\{leaf\_index\}$, where $leaf\_index$ is denoted by "0." The network is traversed from the primary input upwards, and the match set of each node is obtained from the matching algorithm of Figure 7. For example, At node B, by looking for every element of set $\{0, 2, 3\} \times \{0, 2, 3\}$ in the AND table, a match set of $\{0, 2, 4\}$ is obtained. This is because the lookups at $[AND, 0, 0]$ and $[AND, 0, 3]$ return patterns 2 and 4, respectively, and the other combinations return invalid values and are therefore ignored. In addition, node B can be a leaf of a cell and thus pattern 0 also belongs to its match set. By checking the *isGate* array shown in Figure 5, we can tell that not every pattern in the match set $\{0, 2, 4\}$ of node B is a cell: pattern 4 is merely a substructure of pattern 5, and only pattern 2 is a real cell in the library. The objective of keeping pattern 4 in node B is to check the possibility of matching pattern 5 to the parent of node B.



(a) Lookup tables (b) matching of the subject network (c) Matches of Node A

Figure 8: An example of the matching procedure

$\{0, 1, 5\}$ is the match set of Node A, of which pattern 1 and 5 are valid matched cells. The matches for node A are illustrated in Figure 8(c). Here, two matches correspond to the same pattern 5. This is because both $[AND, 0, 3]$ and $[AND, 3, 0]$ return the same pattern 4 during the matching procedure at node B.

3.4 Data structure and cost calculation

As seen in Figure 8(c), each pattern index at a node of the subject Boolean network may correspond to more than one match. Therefore, it is necessary to distinguish between the different matches whose pattern indices are the same. Our procedure differentiates between them by remembering the pointers to the matches of the left child and the right child. A unique match is identified by the triple: pattern index, pointer to the match of the left child and pointer to the match of the right child. The data structure of each match set is illustrated in Figure 9.

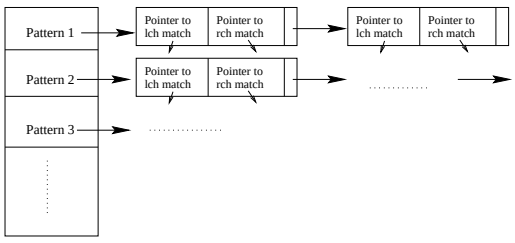


Figure 9: Data structure for each node's match set

This matching method could also possibly help to reduce

the computation time of cost calculation of the matches, in the case where the cost of a match at one node can be formulated into sum of the cost from children and the cost from the matched cell. This can be illustrated with the example from Figure 6, using area as the cost function. In Figure 6, the area cost of match a at node E is $cost(J) + cost(K) + cost(L) + cost(M) + cost(Cell\_a)$, where the $cost(J) + cost(K) + cost(L) + cost(M)$ is the cost from the children, and is denoted C_{child}, and the last term is the cost from the current matched cell. Instead, the area cost of match a at node E can be expressed as $C_{left\_child} + C_{right\_child} + cost(Cell\_a)$, thus saving the duplicate calculation of $cost(J)+cost(K)+cost(L)$. Similarly, the cost calculation of $cost(J) + cost(K)$ can be shared by match d at node H, match a at node E and match b at node F. In the case that the cost is pin dependent or load dependent, where the cost of a match at one node cannot be separated into cost from children and cost of the current cell, it is difficult to simplify cost calculation as above.

4. COMPLEXITY ANALYSIS

4.1 Pattern table generation

The computation cost of pattern table generation algorithm is to enumerate all of the decomposition schemes at the root node of a canonical pattern and fill in the corresponding entries, corresponding to Lines 8–14 of Figure 4. For each decomposition scheme, one traversal of the decomposed two-input tree is performed to lookup the corresponding pattern index. If k is the maximal number of decomposed schemes for a multiple-input node, and n is the maximum number of nodes in the two-input decomposition tree of a canonical pattern, then the time complexity of the pattern table generation procedure is $O(knM)$. Typically, the maximum number of series (parallel) transistors of the cells in a library is 4 or less, and thus the combinatorial number of decompositions, k, is within 7 and is therefore bounded by a constant.

In the pattern table generation procedure, the main overhead of memory space is the pattern table. From the statistics from several libraries, the average number of valid entries of each column(row) in a pattern table is between 1 and 2. The AND/OR table can be stored in sparse format and the memory requirement is found to be linear in the dimension of the table. Hence, the space complexity of the table lookup generation method is $O(M)$, where M is that number of canonical patterns in the library.

4.2 Matching

The main computation cost of the matching procedure at each node corresponds to the two loops shown between Lines 2-8 of Figure 7. Therefore, the time complexity of the matching procedure is $O(M^2N)$, but in practice, the number of executions at a node is generally much lower than M^2. From the statistics of 20 benchmark circuits on library $44-6$, we find the approximately 7.4 lookups are performed, on an average, at every node.

During the matching procedure, the main data structure at each node is shown as Figure 9. Additionally, there is some memory requirement for storing the pattern table. Therefore, the space complexity is $O(mN + M)$, where m is the maximum number of unique matches in each node and N is the number of nodes in the decomposed subject Boolean network. Here, m is somewhat different from M since each pattern index may represent several matches. In general, the number of matches in each node is much lower than M. The statistics of 20 benchmark circuits on library $44-6$ shows an average of 9.6 matches at every node, so that in practice, it can be bounded by a constant.

5. EXPERIMENTAL RESULTS

Our technology mapping procedure has been implemented using C++ and STL. All of the input experiment circuits are first optimized using with *script.rugged*, and then decomposed into a 2-input AND/OR/NOT network. At each node, both positive and negative polarities of each node are matched against the available patterns.

Table 1: The status of the pattern table

Lib	#Cells	#Decomp. patterns	# Canon. patterns	# Valid entries	
				OR tbl	AND tbl
43-5	395	835	395	324	756
44-3	624	4512	624	1152	1152
44-6	3502	12630	3502	4638	4638

Table 1 shows the number of canonical patterns and valid elements in the pattern table for several libraries that are available with SIS. The three libraries, "43-5," "44-3," and "44-6" that are used are listed in Column 1, and the number of cells in each library is shown in Column 2. Column 3 reports the number of the decomposed two-input NAND/NOT node patterns in each library [4]. The next three columns show the results associated with our pattern table. Column 4 reports the number of canonical patterns in the library. This number is also the dimension of our pattern tables. Columns 5 and 6 denote the number of entries with valid values in the OR and AND pattern tables, respectively. From table 1, we can see that in all of the three example libraries, each canonical pattern is also a cell since the number of cells is identical to the number of pattern indices. The number of decomposed patterns is about $2\times$ to $7\times$ the number of canonical patterns.

Table 2: Benchmark circuits

Circuits	#Nodes	Circuits	#Nodes	Circuits	#Nodes
C1355	390	C880	336	apex7	179
C1908	383	C5315	1350	b9	93
C2670	584	C6288	2376	des	2889
C3540	956	C7552	1778	f51m	74
C432	155	9symml	168	rot	506
C499	390	apex6	594	z4ml	27
Total	13228				

We compare our pattern matching results with [8], which, to the best of our knowledge, presents the fastest reported structural matching results. We use the same kind of workstation, Ultra Sparc(300MHz), and our experiments are executed on the same set of LGSynth91 benchmark circuits. The names of the benchmark circuits and the number of 2-input AND/OR nodes in their subject network are reported in Table 2.

Table 3: Comparison with [8] for matching

Lib	# Matches		CPU PG(TG)		CPU matching	
	[8]	ours	[8]	ours	[8]	ours
43-5	75629	94278	1.47	0.05	9.72	2.68
44-3	84390	101956	1.51	0.10	9.98	2.76
44-6	102898	126858	46.59	0.56	23.20	2.97

Table 3 shows the comparison of our results with [8] in terms of both speed and quality. The second and third columns show the number of matches detected with algorithm [8] and with our method, respectively. The fourth

column shows the CPU time spent for pattern generation (PG) by the algorithm in [8], and the CPU time required by our pattern table generation (TG) algorithm is shown in the fifth column. Finally, the last two columns show a comparison between the CPU time for matching with the algorithm in [8] and with our method. These CPU times for matching correspond to the summation over the times required for matching over all benchmark circuits from Table 2, and do not include the CPU times for cost calculation and the backward traversal process for selecting the best cell assignment.

From table 3, we can see that our method is much faster. The larger the library, the greater the gains in speed shown by our results. Our matching procedure is about 8× faster than [8] for the library "44-6." Our pattern table generation procedure is about 80× faster than the pattern generation procedure of [8]. By summing the CPU time of pattern generation with matching procedure, our mapper is nearly 20× faster for library "44-6". Our pattern table generation procedure need not run on-the-fly during technology mapping and it can be executed only once when a new library is generated and the corresponding pattern table is stored. In the CPU time listed in Column 5 and 7 of Table 3, the CPU time for writing out and reading in the pattern table has already been included, respectively. Moreover, our memory requirement for matching is small. In our method, only a couple of tables are required to be loaded into the memory. These are stored in sparse format and occupy a space that is linear in the number of pattern indices.

The number of matches detected in our mapper is about $20\% - 30\%$ more than [8]. We suspect that this is because the AND/OR/NOT base function is used in our method, and the number of nodes in our AND/OR/NOT network is about $20\% - 30\%$ more than the nodes of the NAND/NOT decomposed network [8].

Table 4: Comparison with SIS for TM

Lib	Area cost		CPU time for TM			Mem.(M)	
	SIS	ours	SIS(R)	SIS(M)	ours	SIS	ours
43-5	29971	29522	18.04	219.09	3.02	27	0.36
44-3	30200	29372	108.88	1038.20	3.13	42	0.37
44-6	29773	28888	759.80	2968.60	3.53	114	0.41

In table 4, we compare our technology mapping (TM) results with SIS in terms of speed, memory and quality. As stated in Section 3.4, another potential advantage of our method is to speed up the cost calculation procedure. The computation cost of our mapper consists mainly of the computation cost associated with matching, cost calculation and cell assignment. The simple tree-by-tree mapping method was used by our mapper, and both mappers use the objective of area minimization. The same workstation, benchmark circuits and libraries as Table 3 were applied.

In table 4, Columns 2 and 3 show the total area cost of the mapped circuits with SIS and with our method, respectively. The next three columns show the CPU time required for technology mapping. Column 4 shows the CPU time required for reading the library (R) in SIS, Column 5 reports the CPU time of mapping (M) with SIS, and Column 6 shows the CPU time for our mapper. Columns 7 and 8 report the peak memory requirements for the technology mapping procedure in SIS and in our mapper. For the SIS procedure, 90% of the memory usage was loaded during the library reading procedure.

The results shows that our mapper is about 70× to 1000× faster than SIS and that the memory usage of SIS is about 70× to 280× more than that of our mapper. In addition, the quality of the results is slightly better in our approach. We are not exactly sure the reasons for better quality, but one possible reason is that the methods for handling inverters are different. The comparison of our algorithm with SIS is just an approximate comparison to show the efficiency of our matching method. Because SIS is a general purpose program that does a lot more than just technology mapping, the overhead for both memory usage and computation time may possibly be much higher than our technology mapping program.

6. CONCLUSION

In this paper, a novel structural matching algorithm using table-lookup is proposed. The method explores the relations between the matches of a node and the matches of its child nodes, as well as the structural relations between cells. The method has the advantages of fast speed and low memory usage, and is easy to implement. Comparisons with SIS and the fast matching algorithm in [8] demonstrate the excellent speed up, low memory requirement and quality improvements of our approach over existing methods.

Currently, our structural matching method is limited to the library cells that can be represented by a tree. For a library composed of both cells of tree structure and cells with a DAG structure (EXOR,EXNOR cells), we suggest a hybrid matching method that matches tree-structured cells with our pattern table lookup method, and matches DAG-structured cells with a simple graph isomorphism method. In future work, we will consider extending our pattern table lookup method to DAG cells with graph-based matching.

We expect that the approach should support an arbitrary set of base functions. Currently, the method is implemented using AND/OR/NOT base function for our convenience. There are some arguments and experimental results favoring the choice of using one base function only (e.g., NAND2 or NOR2 plus inverters) [5, 10], and we will consider this extension in future work.

7. REFERENCES

[1] L. Benini and G. D. Micheli, "A survey of boolean matching techniques for library binding," *ACM TODAES*, vol. 2, pp. 193–226, July 1997.

[2] D. Gregory, K. Bartlett, A. D. Geus, and G. Hachtel, "SOCRATES: A system for automatically synthesizing and optimizing combinational logic," in *Proc. DAC*, pp. 79–85, 1986.

[3] K. Keutzer, "DAGON: technology mapping and local optimization," in *Proc. DAC*, pp. 341–347, 1987.

[4] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in MIS," in *Proc. ICCAD*, pp. 116–119, 1987.

[5] G. D. Micheli, *Synthesis and optimization of digital circuits.* McGraw-Hill, Inc., New York, NY, 1994.

[6] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis.* New York: McGraw-Hill, 1994.

[7] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, pp. 333–340, June 1975.

[8] Y. Matsunaga, "On accelerating pattern matching for technology mapping," in *Proc. ICCAD*, pp. 118–123, 1998.

[9] C. Hoffmann and M. ODonnell, "Pattern Matching in Trees," in *Journal of the ACM*, pp. 68–95, 1982.

[10] R. Rudell, *Logic synthesis for VLSI Design.* Memorandum UCB/ERL M89/49, Ph.D. Dissertation, UC Berkeley, 1989.