On Accelerating Pattern Matching for Technology Mapping

Yusuke Matsunaga

yusuke@flab.fujitsu.co.jp

Fujitsu Laboratories Limited

Abstract

Pattern matching algorithm is simple and fast comparing to other matching algorithms such as Boolean matching. One major drawback of the pattern matching is that there is a case where a cell needs a lot of patterns representing its logic function. That is because patterns are decomposed into 2-AND/NOT patterns to match against decomposed subject graphs. Furthermore, the conventional technology mapper does not pay much attention to relations among patterns. Each pattern is tried to match independently. In this paper, a novel pattern matching algorithm that does not require patterns to be decomposed and couple of speeding up techniques utilizing inter-relations among cells are described. These methods are very effective for large cell libraries with complex cells. Experimental results show that our methods gain matching time up to 40 times faster.

1 Introduction

Pattern matching algorithm is widely used for practical technology mapping programs[1, 2]. It is simple and fast comparing to other matching algorithms such as Boolean matching. One major drawback of the pattern matching is that there is a case where a cell needs a lot of patterns representing its logic function. That is because patterns are decomposed into 2-AND/NOT patterns to match against decomposed subject graphs. Indeed, the number of the patterns grows exponentially according to the number of inputs in the worst case. Recently, cell libraries that contain cells with many inputs and complex logic functions are provided. The conventional pattern matching algorithm is not enough to handle them. Furthermore, the conventional technology mapper does not pay much attention to relations among patterns. Each pattern is tried to match independently. This leads linear growth in CPU time according to the number of cells in a library. These points were not deeply considered when the algorithm was developed because cell libraries were not so large. However, current technology provides libraries with large sets of complex cells. These points are not to be ignored.

In this paper, we propose a novel pattern matching algorithm that does not require patterns to be decomposed and couple of speeding up techniques utilizing inter-relations among cells, which is effective for large set of cell libraries. In section 2, we describe our pattern matching algorithm, and in section 3 and in section 4, we describe speeding up techniques using isomorphic relations and containment relations, respectively. We show the experimental results in section 5.

2 Implicit pattern matching algorithm

Our algorithm is also based on the conventional pattern matching (graph matching), however, it does not require to decompose patterns into 2-AND/NOT patterns. Our algorithm tries to match N-AND/NOT patterns against 2-AND/NOT subject graphs directly. This reduces both CPU time and memory requirement greatly especially for complex cells with many inputs. The key idea of this algorithm is that if we know the rules of the target pattern structure we can find matches for the pattern without knowing the actual structure of the pattern. For example, the rules for k-input AND pattern is as follows; “If a tree subgraph has k leaves and no single inverter appears on any path from the root to a leaf, the subgraph matches with k-input AND.” When k becomes larger, the number of 2-AND/NOT patterns that represents k-input AND blows up, however, the rule is the same. So, it is better to use rules rather than patterns themselves. Figure 1 shows a simple algorithm that finds matches for N-input AND.

```
FindAndMatch(Stack of Nodes F, int n) {
    if (n == 0) {
        A match is found. */
        return;
    }
    s ← Pop(F);
    Bind s to the n-th input;
    FindAndMatch(F, n - 1);
    Unbind s;
    while (s is not an AND) {
        Skipping inverter-chains */
        if (s is not an INVERTER) {
            goto end;
        }
        t ← the fanin of s
        if (t is not an INVERTER) {
            goto end;
        }
        s ← the fanin of t
    }
    u, v ← the fanins of s;
    Push(F, u);
    Push(F, v);
    FindAndMatch(F, n);
    Pop(F); /* return value is discarded */
    Pop(F);
    end:
    Push(F, s);
}
```

Figure 1: An implicit matching algorithm for n-input AND

The first argument F is a stack holding ‘frontier’ nodes. The next argument n is the number of fanins remaining unbound. So, in the beginning, we push the fanin nodes of the root node r in a
stack $F$, and then call $\text{FindAndMatch}(F, N)$. At each recursion step, we first try to bind the top node $s$ of the stack $F$ to the $n$-th input and call $\text{FindAndMatch}$ recursively. If node $s$ is an AND gate, $s$ can be an internal node as well as an input node. In such case, the fanins of $s$ are pushed into the stack $F$ and become new frontiers instead of $s$.

As for the case of complex gates such as AND-OR-INVERTER, we need more complicated matching procedure (figure 2). Simply speaking, with invoking $\text{FindAndMatch}$ for each node in the pattern graph, we can enumerate all the matches. For this procedure, each node in the pattern graph has a stack, which we call the frontier stack. This stack is used for holding frontier nodes and is similar to the stack used in $\text{FindAndMatch}$. We denote the frontier stack of node $p$ as $F_p$. Each node also keeps the number of remaining unbound fanins, which is denoted as $n_{p}$. We need another stack called the node stack, which holds pattern graph nodes that we are going to process.

First, we invoke $\text{FindMatch}$ with the root node of the pattern graph as the first argument. Line 1 to 12 treat the case when all the fanins of $p$ is bound. If there is no node remaining in the pattern graph $N$, a match is found. Otherwise, we move to another node at the top of the stack. At line 13, we get node $s$ at the top of the frontier stack of $p$. There are two choices. One is to bind $s$ with a fanin of $p$. The other is to extend the frontier across $s$. In line 14 to 32, we treat the former case. If a fanin $p$ is not compatible with $s$ (i.e. node types are different), they can not be bound. Otherwise, we try to bind them. If that node is an input, $\text{FindMatch}$ is called simply (line 19). If not, we move to the fanin node $q$ and push $p$, which means $p$ must be processed again. Then the fanins of $s$ are pushed into the frontier stack of $q$, and $\text{FindMatch}$ is called (line 21–28). Line 33 to 49 are similar to $\text{FindAndMatch}$, line 33 to 43 handle inverter chain, and line 44 to 49 extend the frontier to the fanins of $s$.

$\text{FindMatch}$ exactly enumerates all the matches that the conventional graph matching algorithm enumerates. As for the complexity of the algorithm, we have not analyzed it theoretically. However, experimental results show that their average performance in the practical field is superior to the conventional algorithm. Memory overhead is negligible. The maximum size of the frontier stack of each pattern graph node is equal to the number of its fanins, and the maximum size of the node stack is equal to the number of nodes in the pattern graph.

3 Speeding up using isomorphism

The second feature of our method is about isomorphic patterns. For example, the patterns of 2-NAND cell and 2-NOR cell are identical without regarding inverters at inputs and outputs (in this paper, we call they are isomorphic.). Suppose we use the inverter chain heuristic, i.e. at every wire, a pair of inverters is inserted. If there is a match with 2-NAND cell, there is also a match with 2-NOR cell. In this case, we do not need to call pattern matcher twice. We can easily get the matches with 2-NOR cell from the match with 2-NAND cell with inverting inputs and outputs. This isomorphic relation forms equivalent classes. So, we only need to try to match with the representative pattern of each equivalent class. Matches of the other patterns are derived from the match of the representative pattern with modifying inputs and output polarities (i.e. including or excluding inverters). This is the intuitive idea of speeding up using isomorphism. However, the real things are not so simple because of the symmetry consideration. Generally, pattern matching algorithm takes care of the symmetry inputs, so that redundant equivalent matches are not generated\(^2\). The problem is that the symmetry

\(^2\) Symmetry inputs may have different delay information. So we should take care of pin assignments among the symmetry inputs in the case of delay optimization. However, this does not always enumerate all the permutation of the symmetry inputs\(^4\).
inputs are not always the same in the same isomorphic group. For example, the pattern (a) and the pattern (b) in figure 3 are isomorphic. The difference between (a) and (b) is the polarity of input C.

Figure 3: Isomorphic patterns

Figure 4: A subject graph

Figure 5: The mapping graph from (a) to (b)

Suppose to match them against the subject graph in figure 4. The pattern (a) has a match like \(A \rightarrow 1, B \rightarrow 2, C \rightarrow 3\). Because inputs \(A, B\) and \(C\) are symmetric, the other matches such as \(A \rightarrow 2, B \rightarrow 3, C \rightarrow 1\) are not considered. From this match, the match for (b) are derived with inverting \(C\). So we get the match for (b), \(A \rightarrow 1, B \rightarrow 2, C \rightarrow 6\). Of course, this match is a valid match for (b). However, there are two other distinctive matches for (b), \(A \rightarrow 2, B \rightarrow 3, C \rightarrow 4\) and \(A \rightarrow 1, B \rightarrow 3, C \rightarrow 5\). This is not derived simply. That is because inputs \(A, B\) and \(C\) are not symmetric for pattern (b), while they are symmetric for pattern (a). One naive solution is not to consider symmetry at all, but is not practical because all the permutations of symmetric inputs is considered.

Selection of the representative patterns is crucial for efficiency. Suppose the example in figure 3 again. If pattern (b) is the representative pattern, we have three distinctive matches against the subject graph in figure 4, \(A \rightarrow 1, B \rightarrow 2, C \rightarrow 6, A \rightarrow 2, B \rightarrow 3, C \rightarrow 4\).
3, C → 4, and A → 1, B → 3, C → 5. Then, we get matches for pattern (a) from these matches, A → 1, B → 2, C → 3, A → 2, B → 3, C → 1, and A → 1, B → 3, C → 2. These three matches are equivalent and thus redundant. Redundant matches do not improve mapping results but spend computation time. Furthermore, finding matches for (b) takes much time than finding matches for (a), because the search space is larger due to the asymmetry. So, pattern (a) is preferable for the representative pattern. From this observation, we choose a pattern that has the least number of symmetric input groups for the representative pattern.

4 Speeding up using containment relations

4.1 Pruning using containment relations

The last feature of our method is utilizing containment relations among patterns. In this paper, we call a subgraph that contains the output node as a root subgraph. If a pattern P is a root subgraph of another pattern Q, we call the pattern P is contained in the pattern Q. If a pattern does not have any match on some node of a subject graph, other patterns that contain the pattern also do not have any match on the node. Using this observation, we can prune hopeless matching effort. We build a directed acyclic graph for this pruning just after generating patterns for cell library. This graph, called the Containment Relation Graph (CRG), represents containment relations among patterns. Each node of the graph corresponds to a pattern, and a directed edge from node P to node Q represents that P is contained in Q. Figure 7 shows an example of CRG with five patterns.

![Figure 7: The containment relation graph](image)

The containment relation is transitive, i.e. A → B and B → C lead A → C. So, we do not need to keep such composed (indirect) relations. In the example of figure 7, the dotted edge from NAND4 to NAND2 is redundant, and thus removed. If there is a direct path from pattern P to pattern Q, P contains Q. So, before trying to match with P, we check whether all the predecessors of P have matches. If there is a predecessor pattern having no matches, it is obvious that P has no matches. Figure 8 shows a code fragment using this pruning.

```
Q ← patterns with no predecessors;
while (Q is not empty) {
    p ← Get(Q);
    f ← Marked;
    foreach predecessor q of p {
        if (q is not marked) {
            f ← NotMarked;
            break
        }
    }
    if (f = Marked) {
        generate_all_matches(p);
        if (p has matches) {
            Set mark to p;
            foreach successor r of p {
                Put(successor r of p);
            }
        }
    }
}
```

Figure 8: Pruning using containment relations

4.2 Containment relations for non-decomposed patterns

To check pattern p is a root subgraph of pattern q, we use pattern matching algorithm, i.e. if there is a match with p at the output node of q, then p is a root subgraph of q. This is the basic definition of the containment relations. In the case of non-decomposed patterns, however, it is not enough. For example, NAND3 contains NAND2, but 2-input AND node does not match 3-input AND node in usual way. So, we modify the rule of matching for containment check. Roughly speaking, n-input node matches m-input node if n ≤ m without exceptions. An exception arises in the case of complex gates. Intuitively, AOI21(AB + C) seems to be contained in AOI211(f = AB + C + D)(figure 9).

```
A
B
C
(a) AOI21
```
```
A
B
C
D
F
(b) AOI21
```

Figure 9: Two and-or-inverter cells

However, it is not. Figure 10 shows two decomposed patterns of AOI211. The pattern of AOI21 is a root subgraph of pattern (b), but is not a root subgraph of pattern (a). This means that if a subject graph contains pattern (a) of figure 10, node 3 in the graph has a match with AOI211 but does not have a match with AOI21. So, in this sense, AOI211 does not contain AOI21.
The key point is that 2-input AND node at the output of AOI21 has a fanin to internal node (inverters are ignored). This fanin prevent to match 3-input AND as in the case of pattern (a) of figure 10. The exact rule for matching for containment check is as follows. Let type($v$) be the type of node $v$ and ni($v$) be the number of node $v$.

1. type($v$) = primary input, or
2. type($v$) = type($u$) ∧ ni($v$) = ni($u$), or
3. type($v$) = type($u$) ∧ ni($v$) ≤ ni($u$) ∧ all fanins of $v$ is primary inputs

5 Experimental results

To demonstrate the efficiency of our method, we had some experiments. Table 1 shows statistics of cell libraries we used in the experiments. Lib2, 43-5, 44-3, and 44-6 are taken from sis distribution. The library named “indust.” is a real cell library, which has many NPN-equivalent cells. The second row in table 1 (# of cells) denotes the number of cells in each library, and the third row (# of rep, pat.) denotes the number of representative patterns. The fourth row denotes the number of decomposed 2-AND patterns. Each cell library has distinct characteristic. In the case of libraries from sis except 43-5, the number of representative patterns are just half of the number of cells, because they have dual cells. The “indust.” library has many NPN-equivalent cells, so the number of representative patterns is about one fifth of the number of cells. And it contains cells with many inputs, so the number of 2-AND patterns is much larger compared with the other libraries.

The experiments were done as follows. First a set of MCNC benchmark circuits were optimized using sis with script.rugged, then each circuit was decomposed into 2-AND/NOT network. We applied various matching algorithms on such networks. Table 2 shows the number of 2-AND nodes of those circuits. We used Ultra Sparc (300MHz) for the experiments. Table 3 shows the results of the experiments. The first column denotes the total number of matches against the benchmark circuits. The next two columns show the CPU time spent for pattern generation. The column named “2-AND” corresponds to decomposed 2-AND/NOT patterns, and the column named “no-decomp.” corresponds to non-decomposed patterns. This includes checking isomorphic patterns and building CRG. The rests of the columns denote CPU time spent for matching. Again, the column named “2-AND” corresponds to the conventional matching algorithm with 2-AND/NOT patterns. The column named “no-decomp.” corresponds to the implicit matching algorithm proposed in section 2, and the last column named “+isom.+cont.” corresponds to the implicit matching with speeding up using isomorphic/containment relations proposed in section 3 and section 4.

The results are obvious. For “44-3” and “indust.” libraries, remarkable gain is achieved with the implicit matching. Also, speeding up technique using isomorphic/containment relations is effective for all libraries. It is worth noting that these gains are not constant, i.e. library become larger, more gain is achieved. As a result, matching time does not increase linearly according to the size of library. Preprocessing time is also a crucial problem for technology mapping. Since we do not need to decompose patterns, preprocessing time of our method is not so much and memory requirement is less than the conventional algorithm. As for “44-6”, the preprocessing time is relatively long. But it requires about 60 seconds to read that library, so the preprocessing time is thought to be reasonable.

One may claim that since Boolean matching[5, 6, 7, 8] does not need pattern graphs, it is faster for cell with many inputs. In general, however, it is not true. Boolean matching requires enumeration of cluster functions. Cluster is a rooted sub-graph in the subject graph, and is a potential candidate of a match. We can not determine which cluster has a match without Boolean matching, so we must enumerate all the clusters.Enumerating clusters and making BDDs[9] for those clusters take much time as well as Boolean matching. In [8], experimental results shows that about 90% of the enumerated clusters have no maches. Furthermore, the number of clusters increases rapidly according to the number of inputs. So it is not suitable for cells with many inputs.

6 Conclusion

We developed a novel method to speed up pattern matching and achieved significant improvement with benchmark and real cell libraries, which is up to 40 times faster comparing with the conventional method. Our contribution is that,

1. An implicit pattern matching algorithm.
Table 1: Cell libraries

<table>
<thead>
<tr>
<th>name</th>
<th>lib2</th>
<th>43-5</th>
<th>44-3</th>
<th>44-6</th>
<th>indust.</th>
</tr>
</thead>
<tbody>
<tr>
<td># of cells</td>
<td>24</td>
<td>.395</td>
<td>624</td>
<td>3502</td>
<td>108</td>
</tr>
<tr>
<td># of rep. pat.</td>
<td>13</td>
<td>352</td>
<td>312</td>
<td>1751</td>
<td>27</td>
</tr>
<tr>
<td># of 2-AND pat.</td>
<td>32</td>
<td>835</td>
<td>4512</td>
<td>12630</td>
<td>5735</td>
</tr>
</tbody>
</table>

Table 3: Matching results

<table>
<thead>
<tr>
<th>library name</th>
<th># of matches</th>
<th>CPU time for pattern generation</th>
<th>CPU time for matching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2-AND pat</td>
<td>no-decomp.</td>
</tr>
<tr>
<td>lib2</td>
<td>52020</td>
<td>0.03</td>
<td>0.00</td>
</tr>
<tr>
<td>43-5</td>
<td>75629</td>
<td>9.41</td>
<td>1.47</td>
</tr>
<tr>
<td>44-3</td>
<td>84390</td>
<td>100.92</td>
<td>1.51</td>
</tr>
<tr>
<td>44-6</td>
<td>102898</td>
<td>764.41</td>
<td>46.59</td>
</tr>
<tr>
<td>indust.</td>
<td>446756</td>
<td>47.13</td>
<td>0.08</td>
</tr>
</tbody>
</table>

2. Speeding up using isomorphic relation.
3. Speeding up using containment relation.

Each technique is mutual complementary and totally they are very effective. For some libraries, implicit matching algorithm gains much, and for some other libraries, isomorphic and containment relation techniques gain much.

References