P-MATCHING FOR READ-ONCE FUNCTIONS

[1]*Anderson Santos da Silva,* [2]*Vinicius Callegaro,* [1,2]*Renato P. Ribas,* [1,2]*André I. Reis*
[1]*Institute of Informatics /* [2]*PPGC - UFRGS, Porto Alegre, Brazil*
{assilva, vcallegaro, rpribas, andreis}@inf.ufrgs.br

ABSTRACT

This paper presents a method to compute P-matching equivalence of read-once Boolean functions. In logic synthesis, the technology mapping process can be a very time consuming task when binding cells from a target library. Our approach transforms a Boolean function in a logical tree and constructs a code to sort this generated tree in an unambiguous order that match with other equivalent sorted tree. Since the method has a polynomial time complexity, it enables the algorithm to scale to hundreds of input variables in quite fast computation. The algorithm efficiency has been evaluated in comparison to related work.

1. INTRODUCTION

The standard cell flow plays a major role on IC digital design. This flow is divided into several steps. One of them is the technology mapping process, which receives a netlist, (e.g. a description about the connectivity of an electronic design representing a Boolean function), and matches covers candidates against cells from a library [1]. Since there are several cuts on netlist, many of them should be considered. Clearly, such task needs to be as fast as possible.

The problem of determining when two Boolean functions are equivalent under permutation of variables is named P-matching [2]. Several methods have been proposed to solve it [3-4] [5], but they are not scalable for functions with more than 8 variables. On the other hand, the read-once (RO) class of functions [6], representing the majority of logic gates in a library [7], have singular properties that can be exploited to accelerate the matching process. A new algorithm of P-matching with this class of functions is proposed in this work and the library presented in [7] is taken into account to compare our approach against the current state-of-the-art matching algorithms. This comparison demonstrated that our approach is very fast, signing the entire library in few milliseconds. Additionally, it is also scalable, matching functions with up to 16 variables.

The remainder of the paper is organized as follows. Section 2 presents basic concepts for a better understanding of the method. In Section 3, the process of coding and ordering a tree from read-once expression is presented. Section 4 calculates the complexity of the proposed algorithm, and Section 5 presents results in comparison to other approaches. The conclusions are outlined in section 6.

2. PRELIMINARIES

Boolean expressions are very useful form to represent logic functions. In this way, it is sometimes interesting to derive other expressions from the original one. This process is known as factorization [8-9]. It generates an expression that respects a factorization criterion, for like as, reduced literal count.

In this context, we can detach three concepts among Boolean functions and its expressions:

P-equivalent expressions - Two logical expressions are P-equivalent if a permutation (P) operation over its literals can transform one expression in the other one, as depicted in Fig. 1. This type of equivalent search is named P-matching.

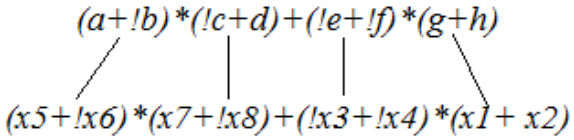

Figure 1- Example of P-equivalent RO functions.

Expression tree - Given a Boolean expression, a rooted tree can be generated directly from it. This rooted tree is an acyclic connected graph *G (V, E)*, where *V* represents a set of nodes in graph and *E* represents a set of edges in graph demonstrating the hierarchical relation between root node and its children. Each node *n* in *V* belongs to the set {+, *, !, α}, where α represents a set of variables. Notice that the literals are in the leaves of the tree, as illustrated in Fig. 2.

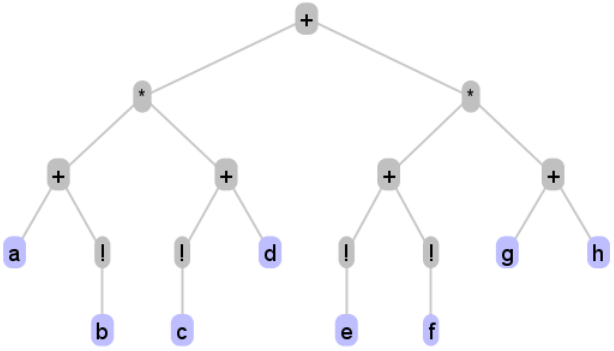

Figure 2- ***Expression tree example representing f= (a+!b)*(!c+d)+(!e+!f)*(g+h) with α ={a,b,c,d,e,f,g,h}.***

Read-Once functions - Read-once (RO) Boolean functions are well known for a long time [8], but their special properties still play important role in modern circuit synthesis flow [6] [10].

A Boolean function is considered RO if it can be represented by a factored form where each variable appears only once [11].

3. PROPOSED P-MATCHING

The task of finding P-matching is very difficult in naïve approaches. Given two RO expressions, a naïve algorithm tries all the possibilities in the variable arrangement generating all permutations in the original expression. Denoting *n* as the number of variables in both expressions, this approach runs in *O (n!)* in the worst case time complexity. In the case of RO expressions, a canonical structure permits a fast way to decide when two RO expressions are equivalent. In the next sub-sections, we present a method to calculate RO P-matching that takes advantage of the canonical structure of RO expressions.

3.1 Tree Codification

Given a RO expression tree, it is possible codify all nodes present in the tree based on codification presented in [6]. In this work, the authors use a codification to verify when two logic tree implements the same logic function. Here, the codification is incremented to represent the relations needed to generate a RO P-matching.

The method starts by replacing all variables in the expression by *'t'* and saving them into a list.

Every leaf node receives a code {1}, as illustrated in Fig. 3.

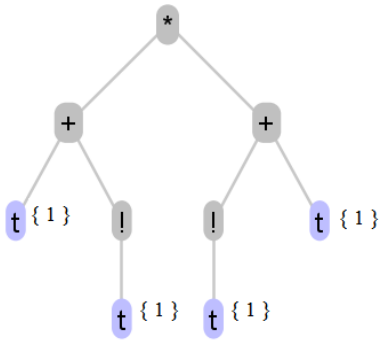

Figure 3- Leaf nodes codification.

Notice that a child can be rooted by NOT, OR or AND nodes. A NOT node has as child only literal nodes. Therefore, these three cases must be treated to generate the code for root nodes.

Definition 1: Given two vectors, $v = \{v_1, v_2, \ldots, v_n\}$ with *n* components and $u = \{u_1, u_2, \ldots, u_m\}$ with *m* components**,** the step of vector 'multiplication' (π) is define as follows:

$\pi(v, u) = \{ v_1 + u_1, \ldots, v_1 + u_m, v_2 + u_1, \ldots, v_2 + u_m, \ldots v_n + u_1, \ldots, v_n + u_m \}$ (1)

Definition 2: Given two vectors, $v = \{v_1, v_2, \ldots, v_n\}$ with *n* components and $u = \{u_1, u_2, \ldots, u_m\}$ with *m* components**,** the vector 'concatenation' (μ) is define as follows:

$\mu(v, u) = \{ v_1, v_2, \ldots, v_n, u_1, u_2, \ldots, u_m\}$ (2)

Definition 3: Given an *AND* root node with $\{k_1, k_2, \ldots, k_m\}$ children, the code associated to this node is π $(c_1, c_2, \ldots, c_m)$, where c_i *is* the code for k_i child, being $0 \leq i \leq m$, viewed as a vector.

Ex: $\pi(\{1, 1, 1\}, \{0, 1\})$
$= \{1+0, 1+1, 1+0, 1+1, 1+0, 1+1\}$
$= \{1, 2, 1, 2, 1, 2\}$ (3)

Definition 4: Given an *OR* root node with $\{k_1, k_2, \ldots, k_m\}$ children, the code associated to this node is $\mu(c_1, c_2, \ldots, c_m)$, where c_i *is* the code for k_i child, being $0 \leq i \leq m$, viewed as a vector.

Ex: $\mu(\{1, 1, 1\}, \{0, 1\})$
$= \{1, 1, 1, 0, 1\}$ (4)

Definition 5: Given a NOT root node, the code associated to this node is a separated code constructed from leaves. Initially, the code has the value {0}. If a NOT root node is found, the value of this code changes to {1}. After this initial step, the codification follows identical to the cases defined above. An example of NOT codification is presented in Fig. 4.

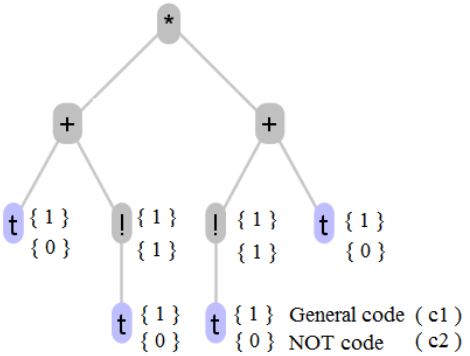

Figure 4- Codification of NOT node.

Using the definitions above, a sequence of these operations generates the codification for a given tree. Fig. 5 shows an example of complete tree codification.

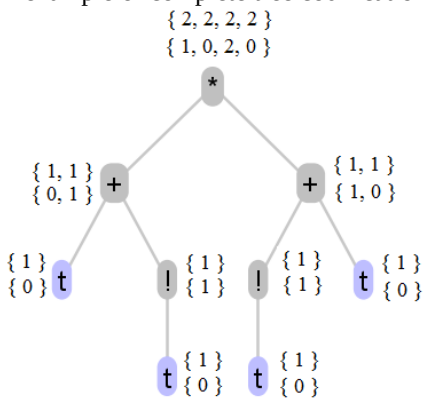

Figure 5- Codified tree.

3.2 Tree Normalization

The process of normalizing the expression tree uses the codes generated as described in sub-section 3.1. Notice that the idea is to reorder the nodes such that the nodes that are more dense stay in the left side of the tree. For this, a decreasing relation is defined for each code viewed as vector.

Definition 6: Given two vectors, $v = \{v_1, v_2, \ldots, v_n\}$ with n components and $u = \{u_1, u_2, \ldots, u_m\}$ with m components:

- $v = u$, if and only if, $n=m$ and v_i is equals to u_i for $(0 < i < n)$;
- $v > u$, if and only if, $n > m$ or $n=m$, and some v_i is larger than u_i for $(0 < i < n$;
- $v < u$, if and only if, $n < m$ or $n=m$, and some v_i is smaller than u_i for $(0 < i < n)$.

Each node in the tree has two codes associated, as presented in Fig. 4.

The node code is put in a pair (c_1, c_2). Therefore, in the node with n children, all n pairs are sorted in first component and, if exists tie, the second component is considered.

If we sort the tree such that larger vectors are present in leftmost side and smaller vectors are present in rightmost side, a normalized tree is generated as depicted Fig.6.

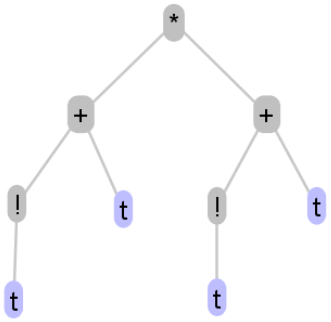

Figure 6- Normalized tree.

3.3 Normalized Logical Expression

Given a normalized expression tree, the inorder traversal sequence returns normalized expression with 't' instead of the name of original variable. Therefore, it can be used as the signature to match this expression with to others generated with the same algorithm. If two trees are equivalent, then their signature resulting of tree traversal are the same. We use these signatures for P-matching. In case of matching, in order to recuperate the original information for each variable replaced by 't', it is enough to access the list of variable set saved in the first step of this algorithm and performs the new replacement in 't' values. Such two equivalent expressions have the same structure, and the matching mapping in variables is the variables placed in the same position in both signatures.

4. ALGORITHM COMPLEXITY AND SCALABILITY ANALYSIS

The time complexity of our approach is calculated as follow. Let l be the number of literals in tree, and let m be the number of operators in tree. The size s of tree is $l + m$ nodes. Every RO expression is normal, i.e., every code does not exceed k, where k is the number of cubes in the original expression in sum-of-products (SOP) form [6].

The worst cases in codifications is the π operation running in $O(k^2)$ and the μ operation running in $O(k)$.Therefore, the time complexity for codification is $O(k^2 + k) = O(k^2)$. In the worst case, we can have s codes with k length. Hence, this runs in $O(k^2 \; x \; s)$ time complexity. The ordering process is done together with the codification, since our algorithm begins the code construction in leaves, any root R_i is achieved when its children have been visited. Hence, its children can be ordered in $O(l \; log \; (l))$ time complexity. The inorder traversal to construct signature runs in $O(s)$, and being a separated step in the algorithm. Consequently, the proposed algorithm runs in $O(k^2 \; x \; s)$.

5. EXPERIMENTAL RESULTS

We compared our algorithm for P-matching of RO expressions with related work [3] [4], running over the genlib set of functions [7]. This library has 3503 RO functions.

The results presented in Table 1 show the best and the worst cases running over the genlib functions with up 7 variables.

Table 1- Time for genlib up to 7 variables.[7]

	Proposed RO-matching	***Heinsberger[1]***	***Sasao[2]***
Higher time	10 ms	312 ms	1123 ms
Lower time	< 1ms	< 1 ms	< 1 ms

The proposed RO-matching run over all genlib functions and generates the signature for each function in a total time of 1220 ms. The largest time observed was 20 ms for a function with 16 variables. Other evaluated approaches are not scalable for functions with more than 8 variables [3][4].

Table 2- Comparison of larger running time with more than 7 variables

Variables	***Proposed RO-matching***	***Heinsberger[1]***
7	10 ms	40 ms
8	16 ms	141ms
9	16 ms	312 ms
10	16 ms	1357 ms
11	20 ms	10371 ms

The table 2 presents the time in milliseconds for functions from genlib with 7, 8, 9, 10 and 11 variables. Notice that Sasao's approach [4] does not scale to functions with more than 8 variables, that is the reason for

its column is be presented empty in Table 2. Our implementation of Heinsberger's approach [3] exceeds more than 100 ms when number of variables exceeds 7 variables. The results show that RO-matching is scalable for more than 7 variables in comparison to existing approaches.

6. CONCLUSION

This paper proposed a polynomial algorithm to decide if two RO expressions are equivalent in permutation, providing also a canonical signature. The proposed method receives two RO expressions and normalizes them based on their tree representations. An algorithm with complexity of order $O(k^2 \times s)$ is provided.

ACKNOWLEDGEMENTS

Research partially funded by Nangate Inc. under a Nangate/UFRGS research agreement, by CAPES and CNPq Brazilian funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community's Seventh Framework Programme under grant 248538 – Synaptic.

REFERENCES

[1] A . Mishchenko; S. Chatterjee; R. Brayton; W. Wang and T. Kam. "Technology Mapping with Boolean Matching, Supergates and Choices," *ERL Technical Report, EECS Dept., UC Berkeley*, Mar 2005.

[2] T. Sasao and J. T Butler, "Progress in Applications of Boolean Functions," *Synthesis Lectures on Digital Circuits and Systems*, vol. 4, no. 1, 2009, pp. 1-153.

[3] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," *In Proc. Design Automation Conference (DAC)*, Jun. 1998, pp. 206–211.

[4] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," *In Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2004, pp. 591-596.

[5] H. Katebi and Igor L. Markov,"Large-scale Boolean matching," *In Proc. Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 771-776.

[6] M. C. Golumbic, A. Mintz and U. Rotics, "Factoring and recognition of read-once functions using cographs and normality," *In Proc. Design Automation Conference (DAC)*, 2001, pp.109-14.

[7] E. Sentovich; K. Singh, L. Lavagno; C. Moon; R. Murgai; A. Saldanha, H. Savoj; P. Stephan; R. Brayton; and A. Sangiovanni-Vincentelli. "SIS:A system for sequential circuit synthesis," *Tech. Rep.UCB/ERL M92/41. UC Berkeley, Berkeley*, 1992.

[8] A. Mintz and M. C. Golumbic, "Factoring Boolean functions using graph partitioning," *Discrete Applied Mathematics*, vol. 149, no. 1–3, 2005, pp.131-53.

[9] R. K. Brayton. "Factoring logic functions," *IBM J. Res. Dev.* vol 31, nº 2, Mar. 1987, pp 187-198.

[10] M. C. Golumbic, A. Mintz and U. Rotics, "An improvement on the complexity of factoring read-once Boolean functions," *Discrete Applied Mathematics*, vol. 156, no. 10, May 2008, pp.1633-36.

[11] J. P. Hayes, "The fanout structure of switching functions," *J. ACM*, vol. 22, no. 4, Oct. 1975, pp.551-71.