A KERNEL-BASED APPROACH FOR FACTORING LOGIC FUNCTIONS

[1]*Vinicius Callegaro,* [2]*Leomar S. da Rosa Jr,* [1]*André I. Reis,* [1]*Renato P. Ribas*
{vcallegaro, andreis, rpribas}@inf.ufrgs.br, leomarjr@ufpel.edu.br

[1]UFRGS – Instituto de Informática – Nangate/UFRGS Research Lab, Porto Alegre, Brazil
[2]UFPel – Instituto de Física e Matemática – Departamento de Informática, Pelotas, Brazil

ABSTRACT

This paper presents a new algorithm for efficient logic functions factoring. The proposed approach is based on kernel association, leading to factored forms according to a given policy cost during the cover step. The experimental results address the literal count minimization, showing the feasibility of the proposed algorithm to manipulate Boolean functions up to 16 input variables.

1. INTRODUCTION

Factoring Boolean functions is one of the fundamental operations in logic synthesis. This process consists in deriving a parenthesized algebraic expression or factored form representing a given logic function, usually provided initially in a sum-of-products (SOP) form or product-of-sums (POS) form. In general, a logic function can present several factored forms. For example, the SOP *Eq=a*c+c*d+b*d* can be factored into the logically equivalents forms *Eq=c*(a+d)+b*d* and *Eq=a*c+d*(b+c)*. The problem of factoring Boolean functions into more compact logically equivalent forms is one of the basic operations in the early stages of logic synthesis. In some design styles (like standard CMOS) the implementation of a Boolean function corresponds directly to its factored form. In other words, each literal in the expression will be converted into a pair of switches to compose the transistor network that will represent the Boolean function. Thus, it is desired to achieve the most economical expression regarding the number of literals in order to obtain the most reduced transistor network. This step guarantees, for instance, that final integrated circuit will not present area overhead [1]. Other benefits of this optimization step may be the delay and power consumption minimization [2].

Generating an optimum factored form (a shortest length expression) is an NP-hard problem. According to Hachtel and Somenzi [3], the only known optimality result for factoring (until 1996) is the one presented by Lawler in 1964 [4]. Heuristic techniques for factoring achieved high commercial success. This includes the *quick_factor* and *good_factor* algorithms available in SIS tool [5]. Recently, a factoring method that produces exact results for read-once factored forms has been proposed [6] and improved [7]. However, the *IROF* algorithm [6,7] fails for functions that cannot be represented by read-once formulas. The *Xfactor* algorithm [8,9] is exact for read-once forms and produces good heuristic solutions for functions not included in this category. Another method for exact factoring based on Quantified Boolean Satisfiability (QBF) [10] was proposed by Yoshida [11]. The *Exact_Factor* [11] algorithm constructs a special data structure called eXchange Binary (XB) tree, which encodes all equations with a given number *n* of literals. The XB-tree contains three different classes of configurable nodes: internal (or operator), exchanger and leaf (or literal). All classes of nodes can be configured through configuration variables. The *Exact_Factor* algorithm derives a QBF formula representing the XB-tree and then compares it to the function to be factored by using a miter [12] structure. If the QBF formula for the miter is satisfiable, the assignment of the configuration variables is computed and a factored form with *n* literals is derived. The exactness of the algorithm derives from the fact that it searches for a read-once formula and then the number of literals is increased by one until a satisfiable QBF formula is obtained.

This paper presents a factoring algorithm based on kernel association. The experimental results address the literal count minimization, showing the feasibility of the proposed algorithm to manipulate Boolean functions up to 16 input variables. The main contribution of this work over the above mentioned heuristic solutions [5,6,7,8,9] is the ability of delivering shorter length expressions in terms of literals. When compared to Yoshida`s approach [11], this algorithm is able to deliver similar solutions without using QBF. The straightforward process only consists in generating and combining kernels to feed a cover table that will provide sub-expressions that could be used to compose factored forms.

The remaining of this paper is organized as follows. Section 2 presents the proposed algorithm for factoring. The results are presented in Section 3. Finally, Section 4 discusses the conclusions and future works.

2. PROPOSED ALGORITHM

The proposed algorithm to achieve factored forms is divided in a sequence of well defined execution steps. The next subsections will describe the algorithm and illustrate its functionality.

2.1. Converting the Input Expression to a SOP Form

The first step consists in converting any Boolean expression to a SOP form. This is required because the proposed algorithm uses the product terms from a SOP to find portions with identical literals that will be manipulated in the next step. The conversion is done through a BDD (Binary Decision Diagram) using well established algorithms [13]. The input Boolean expression is used to create a BDD structure and, afterward, all relevant paths on the BDD are extracted to compose product terms. The SOP is built using sum of these products.

2.2. Terms Grouping

From a SOP form it is possible to perform the identification of equal portions between products. Considering the example of Eq.1, the literal *e* is common for the products *e*f* and *e*g*h*. In this case a new term *e*(f+g*h)* can be built representing the same logical functionality of those two original products. The same occurs between products *b*c* and *a*c*, where a new term *c*(a+b)* can be generated. Notice that the literal *i* cannot be grouped to others since it is unique on the Eq.1.

$$Eq = b*c+a*c+e*f+e*g*h+i \quad \text{(Eq.1)}$$

This step of the algorithm is executed recursively because when generating new terms other groupings become possible. Considering the example of Eq.2, on the first pass the Eq.3 will be returned. Applying the algorithm recursively in the sub product *(c*e + c*f*g + c*f*h + d*e + d*f*g + d*f*h)* from Eq.3, the method will returns Eq.4. At this point, the optimized found terms will be used to compose equivalent expressions of the original one (Eq.2), as illustrated by Eq.5.

$$Eq = b*c*e+b*c*f*g+b*d*e+b*d*f*g+b*d*f*h \quad \text{(Eq.2)}$$
$$Eq = b*(c*e+c*f*g+c*f*h+d*e+d*f*g+d*f*h) \quad \text{(Eq.3)}$$
$$Eq = (f*(h + g) + e)*(d + c) \quad \text{(Eq.4)}$$
$$Eq = b*((f*(h + g) + e)*(d + c)) \quad \text{(Eq.5)}$$

All set of terms returned by this step will be used in the sequence. The number of returned terms will depend on the possibility of grouping different portions of the input expression.

2.3. Kernels Sharing

In order to perform a fine optimization, all terms returned by the previously step are analyzed and shared when it is possible. Taking into account a set of terms {*(a*(c+d))*, *(b*(c+d))*, *(c*(a+b))*, *(d*(a+b))*}, each term is divided in kernels as illustrated in Tab.1.

Tab. 1 – Kernels from a set of terms.

#	Terms	Kernel 1	Kernel 2
1	*a*(c+d)*	*a*	*(c+d)*
2	*b*(c+d)*	*b*	*(c+d)*
3	*c*(a+b)*	*c*	*(a+b)*
4	*d*(a+b)*	*d*	*(a+b)*

By evaluating the lists of kernels, the algorithm tries to find equivalent ones that are candidates to be shared with others. Considering kernel *a*, for instance, it is possible to observe that it cannot be shared with other kernel. However, kernel *(c+d)* can be shared with another identical kernel. Thus, the algorithm shares all candidates and builds new terms. In this example a new term *(a+b)*(c+d)* is obtained. The set of new terms generated during this step is {*a*(c+d)*, *b*(c+d)*, *c*(a+b)*, *d*(b+a)*, *(c+d)*(a+b)*}. Notice that if some repeated term is generated, then it is not added into the set.

2.4. Covering Step

The last step of the proposed algorithm receives all terms (sets) generated during the steps presented in subsections 2.1, 2.2 and 2.3. All terms are put together in order to compose a cover table. After that, a standard covering algorithm [14] is applied and the best solution is delivered as the factored expression. It is important to say that in this paper the number of literals was the cost to be minimized. Nevertheless, other costs may be considered to be minimized (like number of products or number of sums in the expression, for instance).

3. RESULTS

The algorithm described above was implemented in Java language. In order to validate the proposed method, the set of Boolean functions present in genlib.44-6 [5] was used. A total of 3321 logic functions were extracted from the library to feed the execution flow. In the sequence, the Boolean expressions were factored, one by one, using the proposed algorithm. The experiment was performed in a 1.86Ghz Core 2 Duo processor with 2Gb memory, CentOS 5.2 Linux operating system and Java virtual machine v.1.6.0.

Tab. 2 shows some factored expressions obtained with the proposed approach. It is possible to see that the method is able to deliver exact forms for read-once functions, even for input expressions with a reasonable number of literals.

Tab. 2 – Results of some factored expressions.

Input SOP	Factored Expression
!b*!d*!f*!g+!b*!d*!e+!b*!c+!a	!(a*(b+c*(d+e*(f+g))))
!b*!g*!h*!i+!b*!d*!f+!b*!d*!e+!b*!c*!f+!b*!c*!e+!a	!(a*(b+(c*d+e*f)*(g+h+i)))
!c*!d*!g*!i+!c*!d*!g*!h+!c*!d*!e*!f+!b*!g*!i+!b*!g*!h+!b*!e*!f+!a	!(a*(b*(c+d)+(e+f)*(g+h*i)))
!g*!h*!m*!n+!g*!h*!k*!l+!g*!h*!i*!j+!e*!f*!m*!n+!e*!f*!k*!l+!e*!f*!i*!j+!c*!d*!m*!n+!c*!d*!k*!l+!c*!d*!i*!j+!a*!b	!((a+b)*((c+d)*(e+f)*(g+h)+(i+j)*(k+l)*(m+n)))
!d*!n*!o*!p+!d*!k*!l*!m+!d*!h*!i*!j+!d*!e*!f*!g+!c*!n*!o*!p+!c*!k*!l*!m+!c*!h*!i*!j+!c*!e*!f*!g+!b*!n*!o*!p+!b*!k*!l*!m+!b*!h*!i*!j+!b*!e*!f*!g+!a*!n*!o*!p+!a*!k*!l*!m+!a*!h*!i*!j+!a*!e*!f*!g	!(a*b*c*d+(e+f+g)*(h+i+j)*(k+l+m)*(n+o+p))

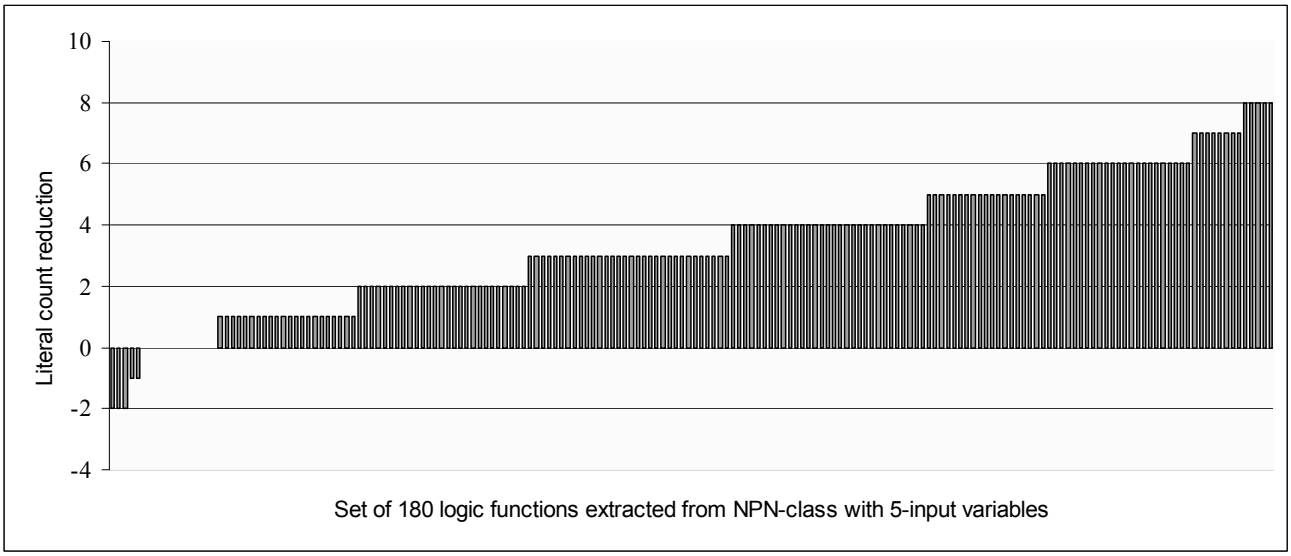

Fig. 1 – Literal count reduction over the SIS algorithms.

The average time for factoring Boolean expressions up to 10 different variables was less than a second. This average time grows up when more variables are added to the expression. The average time for factoring expressions with 16 different variables was around 181 seconds.

In order to evaluate results regarding non-read-once functions (where the minimum factored form cannot be expressed with just one literal per variable), a set of 180 logic functions extracted from the NPN-class with 5-input variables [15] was used. This set of functions was factored with the proposed method and with *quick_factor* and *good_factor* algorithms available in SIS tool [5]. These two well-established algorithms are used as reference to compare the quality of factoring methods.

Fig. 1 shows the literal count reduction when comparing to the *quick_factor* and *good_factor* algorithms. Only 5 factored expressions where obtained with more literal count when using the proposed algorithm. For 12 expressions the obtained literal count was the same. In general, the proposed algorithm demonstrated a significant improvement over the SIS algorithms, achieving reductions up to 8 literals. Tab. 3 describes the number of logic functions and the respective obtained reduction.

The average time for factoring this set of Boolean functions with the proposed method was 253ms.

Tab. 3 – Number of logic functions and reduced literals.

# of Reduced Literals	# of Functions
-2	3
-1	2
0	12
1	22
2	27
3	30
4	29
5	19
6	23
7	8
8	5
Total	**180**

4. CONCLUSIONS AND FUTURE WORKS

This paper presented a new algorithm for efficient Boolean factoring. Experimental results demonstrated that the algorithm is feasible to manipulate Boolean expressions up to 16 input variables in short CPU execution time. It is able to deliver minimum factored forms in terms of literal count for read-once functions. When comparing to the algorithms presented in SIS Tool, the proposed method is able to delivery expressions with less literal count.

As future works it is intended to expand the cover policies in order to allow the algorithm to provide factored forms concerning minimization of other costs (like products or sums length).

5. ACKNOWLEDGMENTS

This work has been developed in cooperation with Nangate Inc., including financial support.

6. REFERENCES

[1] Brayton, R. K. 1987. Factoring Logic Functions. IBM Journal Res. Develop., vol. 31, n. 2, pp. 187-198.

[2] Iman, S. and Pedram, M. Logic Extraction and Factorization for Low Power. DAC'95. ACM, New York, NY, pp. 248 – 253.

[3] Hachtel, G. D. and Somenzi, F. 2000. Logic Synthesis and Verification Algorithms. 1st. Kluwer Academic Publishers.

[4] Lawler, E. L. 1964. An Approach to Multilevel Boolean Minimization. J. ACM 11, 3 (Jul. 1964), pp. 283-295.

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

[6] Golumbic, M. C.; Mintz, A.; Rotics, U. 2001. Factoring and recognition of read-once functions using cographs and normality. DAC '01. ACM, New York, NY, pp. 109-114.

[7] Golumbic, M. C.; Mintz, A.; Rotics, U. 2008. An improvement on the complexity of factoring read-once Boolean functions. Discrete Appl. Math. Vol. 156, n. 10 (May. 2008), pp. 1633-1636.

[8] Golumbic, M. C. and Mintz, A. 1999. Factoring logic functions using graph partitioning. ICCAD '99. IEEE Press, Piscataway, NJ, pp. 195-199.

[9] Mintz, A. and Golumbic, M. C. 2005. Factoring boolean functions using graph partitioning. Discrete Appl. Math. Vol. 149, n. 1-3 (Aug. 2005), pp. 131-153.

[10] Benedetti. M. 2005. sKizzo: a suite to evaluate and certify QBFs. 20th CADE, LNCS vol. 3632, pp. 369–376.

[11] Yoshida, H.; Ikeda, M.; Asada, K., 2006. Exact Minimum Logic Factoring via Quantified Boolean Satisfiability. ICECS '06. pp. 1065-1068.

[12] Brand, D. 1993. Verification of large synthesized designs. ICCAD 93. IEEE, Los Alamitos, CA, pp. 534-537.

[13] Drechsler, R.; Becker, B. Binary Decision Diagrams: Theory and Implementation. Boston, USA: Kluwer Academic, 1998.

[14] Wagner, F.R.; Ribas, R.; Reis, A. Fundamentos de Circuitos Digitais. Porto Alegre: Sagra Luzzatto, 2006.

[15] Ledur, M.; Marranghello, F.; Da Rosa Junior, L. S.; Reis, A. I.; Ribas, R. P.. Set of Digital Cells According to Logic Equivalences. In: VII Student Forum on Microelectronics, 2007, Rio de Janeiro. VII Student Forum on Microelectronics CDROM. Porto Alegre: SBC, 2007.