FUNCTIONAL COMPOSITION APPLIED TO AIG CONSTRUCTIVE OPTIMIZATION

Thiago Figueiró, Renato Ribas and André Reis
{trfigueiro, rpribas, andreis}@inf.ufrgs.br
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500, Bloco IV, Porto Alegre, RS, Brazil

ABSTRACT

This work presents a logic synthesis algorithm that works on top of an And-Inverter-Graph (AIG) data structure and the final goal is to minimize the number of nodes in the AIG. This paper introduces a novel approach for AIG construction based on a new synthesis paradigm called functional composition. The idea is to perform the synthesis by associating simpler AIGs, in a bottom-up approach. Results indicate that this approach present a gain of 5% when compared to first optimize an equation and then build the AIG.

1. INTRODUCTION

Algorithmic logic synthesis is usually performed in two steps, one performed over Boolean equations (regardless any physical property) and another where the resulting logic is mapped into a physical cell library or other physical implementation. The first step is composed of several logic operations such as Decomposition Extraction, Factoring, Substitution and Elimination [1]. These operations may be either explicitly performed (SIS [2]) or implicitly performed by other methods such as And-Inverter Graph (AIG) rewriting (ABC [3]).
The main goal of technology independent synthesis is to compute a representation of a given combinational circuit with optimized costs measured independently of the target technology. These costs may be related to literals (e.g. number of literals and logic depth) or AIG nodes (e.g. number of nodes and graph depth measured in nodes). In SIS [2], the technology independent cost was based on literals. In more recent tools like ABC [3], the technology independent cost is based on AIG nodes. The use of AIG nodes is justified as it is expected to produce a better correlation with final area and delay once the circuit has been mapped to a target technology [4]. This advantage with respect to literals comes from (1) the fact that AIGs are multi-level representations allowing sharing of nodes; and (2) the AIG node is a simple structure, which keeps correlation with area as all nodes have homogeneous simple granularity. AIGs are graphs whose nodes are limited to two-input ANDs while inverters are indicated by a special attribute on the edges of the network. Sum of products [5] and factored forms [6] were used in SIS [2] to represent logic function of single output circuit nodes. However, the granularity of the logic functions could vary, leading to optimizations in the number of literals that would not translate in better circuit characteristics after mapping [4, 7]. These issues have been discussed in the IWLS benchmarking effort [7]. For this reason, most modern logic synthesis tools work on top of AIG representations [4, 8-9], including tools for FPGA synthesis that use the concepts of K-cuts [10-11], factor cuts [12] and KL-cuts [13] on AIGs. However, a recent work by Joswiak [14] has shown that for small circuits there is still room for area gains with respect to AIG based tools. In this sense there is a need for an aggressively Boolean algorithm for AIG rewriting. One example of such an algorithm devoted to factoring is presented in [15]. An additional advantage of that algorithm [15] is that it operates by functional composition of smaller known sub-functions, which allows the method to optimize cost functions that take more than just literals into account. Notice that the factoring algorithms presented in [6, 15] do not include sub-expression reuse, which makes multi level representations like TANT networks [16] competitive compared to two-level expressions [5]. The contribution of this paper is to adapt the algorithm in [15] to operate in multilevel expressions considering shared sub-expressions.
This paper presents a novel approach for AIG construction, based on this new synthesis paradigm called functional composition [15]. The approach proposed herein consists in constructing AIGs from association of simpler AIGs, in a bottom-up approach.
This paper is organized as follows. Section 2 presents basic concepts regarding technology independent synthesis algorithms and AIGs. Section 3 presents the proposed algorithm for AIG construction. Section 4 presents and discusses the results, compared with previous works and SIS/ABC available approach and the final section discusses the conclusions of this paper.

2. BASIC CONCEPTS

A Boolean function describes how to determine a Boolean value output based on some logical calculation performed over Boolean inputs. A function *f* is said to be a subfunction of *g* whenever exist another function *h* and an operation OP for which *g* = *f* op *h*. An equation is one representation of a function, which may also be described as a Binary Decision Diagram (BDD) or as a

Truth Table (TT), for instance. Every representation of a function may be classified as canonical or non-canonical. A representation is said to be canonical if every function will always be described exactly in the same way. Examples of canonical representations are BDDs and TTs (as long as the variables ordering are the same). Equations are non-canonical representations of a function; therefore, the same function may be described by different equations. For instance, equations (1), (2) and (3) represent exactly the same function. An equation is composed of literals. A literal is an instance of a variable (positive literal, for instance "A") or its complement (negative literal, for instance "!A".

F=A*B+C (1)

F=A*B+A*C (2)

F=(A+C)*(B+C) (3)

AND-INV graphs (AIGs) are another way to represent Boolean functions. An AIG is a graph composed exclusively of two inputs AND gates and inverters. It is frequent the representation of the inverters as a special flag on the graph's edge and therefore all nodes on the graph represent two input ANDs. Figure 1 present one AIG for the equation F=A*!B*C. AIGs are a not canonical, therefore, different AIGs may represent the same Boolean function. Figure 2 present three different AIGs for the same function (they implement the equations presented previously).

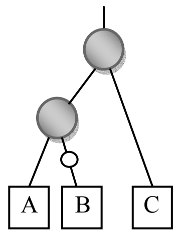

Figure 1. Sample of AIG for the equation F=A*!B*C

3. PROPOSED ALGORITHM

The proposed algorithm is performed in three steps: Building the Allowed Subfunctions, Creating Single Variable Functions and Combining and Evaluating Subfunctions. First, the set of allowed subfunctions is built to reduce the number of subfunctions to evaluate. Then, all possible subfunctions of one variable are created, considering them as the seeds to the associations that follow it. Finally, the initial (one variable) subfunctions are combined and evaluated until the target function is generated.

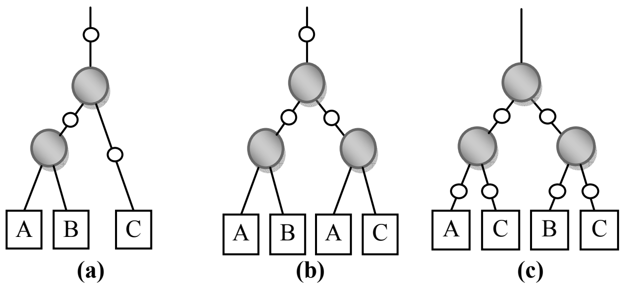

Figure 2. AIG representing the same function but described as (a) F= A*B+C, (b) F= A*B+A*C and (c) F=(A+C)*(B+C).

3.1. Building the Allowed Functions

In order to reduce the search space of subfunctions and, therefore, improve the performance and space required by the algorithm, a set of allowed subfunctions is determined. This set is composed by all positive and negative cofactors of the target function, and its "sub-cofactors", built recursively. Moreover, AND and OR operations are performed on all subfunctions generated, two by two. Also, the target function is also included as an allowed subfunction.

For example, consider the target function to be Q = A*B+C. Since it contains three literals, it will be represented by 8 bits. Therefore, the target function is (01010111). The allowed subfunctions will be the target function, plus the cofactors of this function in all variables "A", "B" and "C", its subcofactors and the results of AND and OR operations on the cofactors and subcofactors.

First, consider the cofactors in "A" of the target function, which are the negative (01010101) and the positive (01110111). The negative cofactor in "A" is already a variable, and no further evaluation will be performed (it is C). The positive cofactor in "A" will be evaluated as subcofactor in "B" and in "C". In "B" they are negative (01010101) and positive (11111111) while in "C" they are negative (00110011) and positive (11111111). Both positive subcofactors are constant "1" and no longer are evaluated. Both negative cofactors are variables and are also no further evaluated. Continuing with the target function, now the cofactors in "B" must be evaluated, which are the negative (01010101) and the positive (01011111). The negative cofactor in "B" is already a variable and, therefore, will no further be considered. The positive cofactor in "B" may be evaluated in subcofactors in both "A" and "C". In "A", they are negative (01010101), which is the variable "C" and positive (11111111). The cofactors in "C" are negative (00001111) which is the variable "A"and positive (11111111), which is the constant "1". The only missing evaluation is the "C" cofactors of the target function, which are negative (00000011) and positive (11111111). The positive is a constant and is no longer evaluated. The negative cofactor in "C" is then evaluated as subcofactors in "A" or "B". The subcofactors in "A" are

negative (00000000), which is a constant and positive (00110011), which is a variable. The negative cofactor in "C" is also evaluated in "B", generating the negative subcofactor (00000000) which is a constant and the positive one (00001111), which is the variable "A". The intermediate set of allowed subfunctions is: {01010111, 01010101, 00000011, 01110111, 00110011, 01011111, 00001111, 11111111, 00000000}.
This set is then combined two by two by the AND and OR operations. The final allowed subfunctions are: {00000000, 11111111, 00001111, 01011111, 00000011, 00111111, 00000111, 01111111, 00000101, 00010011, 01111111, 00000001, 01010111, 01010101, 00010001, 00000011, 01110111, 00110011, 01110111.

3.2. Creating Single Variable Functions

The creation of single variable subfunctions consists in generating the functions for each variable, as they appears in the truth table form. The order in this case is lexicographic.
Continuing with the example of the target function Q = A*B+C, the variables are "A" (00001111), "!A" (11110000), "B" (00110011), "!B" (11001100), "C" (01010101) and "!C" (10101010). These are the variables, but not all will be considered since only a subset of them are available in the allowed subfunction list. They are {00001111, 00110011, 01010101). This is consistent with the desired behavior once the target function is positive unate for all variables and, therefore, no negated variable is required for its description.

3.3. Combining and Evaluating Subfunctions

After generating the single variable subfunctions, the algorithm starts combining these subfunctions in order to generate larger subfunctions until finding the exact function targeted. The combination is performed by picking two by two all subfunctions and run the AND and OR operations over them. In this case, the elements "A" and "B" would generate the subfunctions "A*B" and "A+B". The generated subfunctions are only accepted if they are present in the allowed subfunctions list. If the subfunction is accepted, it is added in a different set, containing all functions generated in this step of combination and it is added in the already generated subfunctions, that will either avoid further generations or keep all possible implementations for the same subfunction, in order to be selected the best one according to its number of operations (nodes of the AIG), which will reflect a smaller area in the final circuit.
For instance, in the example of the target function (01010111), the single variable subfunctions that were also on the allowed subfunctions hash are the following {00001111, 00110011, 01010101}. These subfunctions are used to generate other subfunctions by applying the AND and OR operations. Hence, in step 2 we have 00001111 & 00110011 => 00000011 and 00001111 | 00110011 => 00111111. These two new subfunctions are also on the allowed list and, therefore, are accepted.
Step 1 is the one variable subfucntions. Step 2 is the combination by the operations AND and OR of the subfunctions in Step 1. Step 3 is the combination of the subfunctions in Step 1 with the ones in Step 2, avoiding duplicity and removing (striked-though) the subfunctions not present in the allowed list.

Table 1. Functions after every step of composition

Step 1	Step 2	Step 3
00001111	00000011	~~00011111~~
00110011	00111111	00000001
01010101	00010001	~~00000111~~
	01110111	01111111
	00000101	00110111
	01011111	**01010111**
		00010101

The algorithm stops as it finds a subfunction that is exactly the same as the target function. In this example, the solution is the combination of 00001111 AND 00110011 followed by an OR combination with 01010101. These steps, which were performed on the subfunctions, may now be performed on the AIG nodes. Figure 3 present the step by step on the AIG nodes.

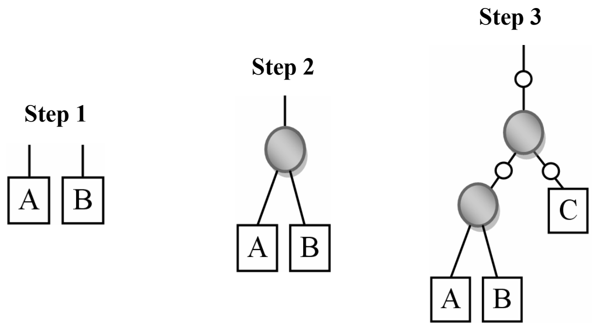

Figure. 3. The application of the composition identified on the subfunctions in the AIG nodes.

4. EXPERIMENTAL RESULTS

In order to evaluate the proposed algorithm, a comparative test was performed. A set composed of 3982 functions presenting up to 4 variables was processed by GF (from SIS) and by the proposed method. As the goal is to optimize the resulting And-Inverter Graph, the equations generated by ABC and Equation Composition algorithm were used as input for the FRAIG algorithm, available in

ABC. Table 2 presents the sum of nodes present in all AIGs, the nodes average for each AIG and their deviations. Figure 5 presents the distribution of gained nodes by the proposed method when compared to ABC followed by FRAIG.

Table 2. Comparison of And-Inverter Graphs generated by ABC + FRAIG and the proposed method

	ABC + FRAIG	Proposed Method
Number of Nodes	32813	31258
Average No. Nodes	8.24	7,84
Average Nodes Reduction	4,97%	-

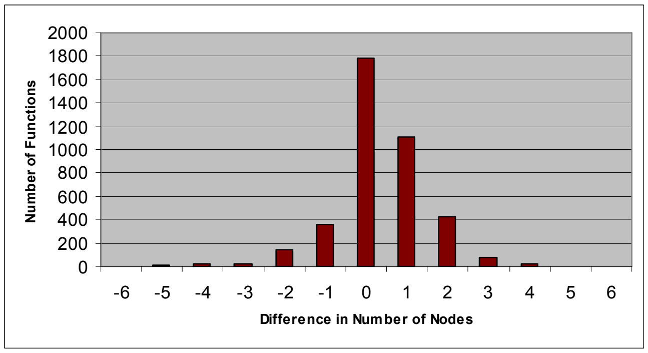

Fig. 5. Distribution of number of nodes gained by the proposed method when compared to ABC + FRAIG. Negative values represent advantage to ABC + FRAIG, while positive values represent advantage to the proposed method.

5. CONCLUSION

This paper has proposed a novel approach for local AIG rewriting algorithm. The proposed method is based on a new synthesis paradigm (functional composition) and it is able to produce smaller graphs (AIGs) when compared to other approaches, which consist in traditional two steps (equation factoring + graph construction). Moreover, the proposed algorithm has the ability to take secondary criteria for optimization into account, or a set of criteria (by using a cost function) and evaluating the already known costs of each partial solution during the composition procedure. Logic depth results have shown that the algorithm is very effective in taking secondary criteria into account.

6. ACKNOWLEDGMENT

Research partially funded by Nangate Inc under a Nangate/UFRGS research agreement, by CNPq Brazilian funding agency, and by the European Community's Seventh Framework Programme under grant 248538-Synaptic.

7. REFERENCES

[1] G.D. Hachtel and F. Somenzi, Logic Synthesis and Verification Algorithms, Kluver Academic Publishers (1996).

[2] 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, Berkely, 1992.

[3] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[4] Mishchenko, A., Chatterjee, S., and Brayton, R.. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. DAC '06, 532-535.

[5] Mishchenko, A. and Sasao, T. 2003. Large-scale SOP minimization using decomposition and functional properties. DAC '03, 149-154.

[6] M.C. Golumbic and A. Mintz, Factoring Logic Functions Using Graph Partitioning, ICCAD '99. IEE Press, Piscataway, NJ, 195-199.

[7] IWLS2003. No more counting of Literals. Presentation of discussion Group 3 at IWLS 2003. Available at: www.sigda.org/iwls/iwls2003/no_more_literals.ppt

[8] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton, "FRAIGs: A Unifying Representation for Logic Synthesis and Verification", ERL Technical Report, EECS Dept., UC Berkeley, March 2005.

[9] T. Figueiro, R. P. Ribas, A. I. Reis, AIG Rewriting Considering Multiple Objectives, 25th South Symposium on Microelectronics, Porto Alegre, Brazil, May 2010.

[10] Cong, J., Wu, C., and Ding, Y. 1999. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. FPGA '99, NY, 29-35.

[11] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," IEEE Trans. CAD, Vol.13, No. 1 (January 1994), pp. 1--12.

[12] Chatterjee, S., Mishchenko, A., and Brayton, R.. Factor cuts. ICCAD '06, 143-150.

[13] Osvaldo Martinello Jr, Felipe S. Marques, Renato P. Ribas, André I. Reis. KL-Cuts: A New Approach for Logic Synthesis Targeting Multiple Output Blocks. DATE 2010, pp. 777-782.

[14] Lech Jozwiak, Artur Chojnacki, Aleksander Slusarczyk, "High-Quality Circuit Synthesis for Modern Technologies," ISQED 2008, pp.168-173.

[15] A. Reis, A. Rasmussen, L. Rosa, R. Ribas., Fast Boolean Factoring with Multi-Objective Goals, International Workshop on Logic & Synthesis, IWLS 2009.

[16] Lee, H. -. 1978. An Algorithm for Minimal TANT Network Generation. IEEE Trans. Comput. 27, 12 (Dec. 1978), 1202-1206