Applications of Functional Composition

[1]Mayler G. A. Martins, [1]Renato Perez Ribas, [1]André Inácio Reis
{mgamartins,rpribas,andreis}@inf.ufrgs.br,

[1]Universidade Federal do Rio Grande do Sul - PGMICRO

Abstract

This paper presents functional composition (FC), a new paradigm for combinational logic synthesis. FC is based on the following principles: (1) representation of logic functions as a bonded pair of functional/structural representations; (2) it starts from a set of initially known functions; (3) simpler functions are associated to create more complex functions; (4) a partial order that enables dynamic programming is respected; (5) a set of allowed functions is maintained to reduce execution time/memory consumption. We present functional composition algorithms variants for Boolean factoring, AIG rewriting, minimum decision chain computation and SOP generation.

1. Introduction

Functional decomposition (FD) is a method for combinational logic synthesis in which a Boolean function is decomposed into a set of smaller functions that implement it. FD has been introduced by the pioneering works of Ashenhurst [1] and Curtis [2]. The results of functional decomposition are highly Boolean by nature, meaning it is able to produce non-trivial logic rewritings that is very suitable to overcome the structural bias [3]. FD has been extensively used for FPGA mapping, as it is easy to control the number of inputs of each sub-function [4]. However, FD has two critical drawbacks in this context. Firstly, it is a top-down approach, which breaks the function to be decomposed into smaller ones. This way, the implementation cost of the functions is not necessarily known. Secondly, as it involves costly operations, it cannot be done in an exhaustive way, leaving parts of the solution space unexploited.

To overcome the drawbacks of functional decomposition applied to local function rewriting, we propose functional composition (FC). It is a novel synthesis paradigm that performs bottom-up association of Boolean functions as opposed to top-down functional decomposition. By performing bottom-up process, FC has a better control of the implementation cost of the final function. By relying on bonded-pair representation, FC can perform a more complete search of the solution space, yielding better results.

FC is based on the following principles: (1) representation of logic functions as a bonded pair of functional/structural representations; (2) it starts from a set of initial functions; (3) simpler functions are associated to create more complex functions; (4) a partial order that enables dynamic programming is respected; (5) a set of allowed functions is maintained to reduce execution time/memory consumption. In this work, we present FC algorithms variants for Boolean factoring, AIG rewriting, minimum decision chain computation and SOP generation.

This paper is organized as follows. Section 2 presents general principles of FC. The general flow of FC is shown in Section 3. Some applications for the FC paradigm are described in Section 4, to illustrate how the particularization of the general principles can lead to FC algorithms for different applications. The final section discusses the conclusions.

2. General Principles

The FC paradigm is based on some general principles. These principles include the use of bonded-pair representation, the use of a set initial functions to start the process, the association between simpler functions to create more complex functions, the control of costs achieved by using a partial order that enables dynamic programming, and the restriction of allowed functions to reduce execution time/memory consumption. These general principles are discussed below.

2.1. Bonded-Pair Representation

FC uses bonded pairs to represent logic functions. The bonded pair contains one functional and one structural representation of the same Boolean function. The functional representation is used to avoid the structural bias, making FC to be a highly Boolean method. Normally the functional representation needs to be a canonical representation like a truth table or a ROBDD node. The structural element in bonded-pair is related to the final implementation of the target function to be synthesized. The structural element in the bonded pair is used to control costs in the final implementation and by nature it is not a canonical implementation, as costs may vary.

2.2. Initial Functions

The FC paradigm computes new functions by the associations of known functions. As a consequence, a set of initial functions is needed before the algorithm starts. The set of initial functions needs to have two characteristics. First, the bonded-pairs for the initial functions have to be simple to compute, to allow efficient initialization. Second, the initial functions have to have known (preferable minimum) costs for each function, to allow the computation of the cost for derived functions. The set of initial functions can vary depending on the specific FC algorithm, as it will be discussed later.

2.3. Bonded-Pair Association

The FC is done using bonded pairs, represented by a functional representation and a structural representation of the same function. When a logic operation (e.g. logic and) is applied to bonded pairs, the operation is applied independently to the functional and to the structural part of the bonded pair are associated. This way, the correspondence between the functional and the structural representation is still valid after the bonded pair association. The main advantage of the bonded pair association is the fact that it is much faster to compute the operations between the representations of the same type than it would be to convert a functional representation into a structural one or vice-versa. Fig. 1a presents the association of bonded pairs. The bonded pair $<F_3, S_3>$ is obtained from bonded pairs $<F_1, S_1>$ and $<F_2, S_2>$. The computation of the functional part ($F_3=F_1+F_2$) is independent of the computation of the structural part ($S_3=S_1+S_2$). The concept can be expanded to do complex operations, both in function and structure, as seen in Fig. 1b.

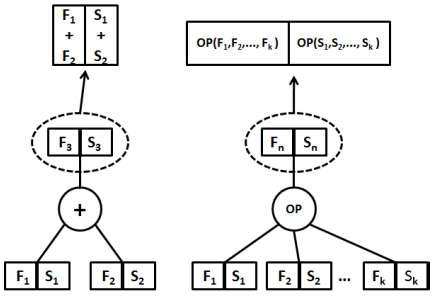

Figure 1: Examples of bonded-pair association: (a) using primitive operations (OR operation) and 2 simpler elements; (b) using complex operations with k elements in a k-ary operation.

2.4. Partial Order and Dynamic Programming

The key idea behind dynamic programming (DP) is to solve a problem in which an optimal solution is obtained by combining optimal sub-solutions. This can be done for problems that have a so called optimal sub-structure. DP starts by solving sub-problems and then combines the sub-problem solutions to obtain a complete solution. In FC, DP is used associated to the concept of partial ordering. The partial ordering is used to classify costs of intermediate solutions. This is done to ensure that implementations (the structural elements in the bonded pairs) with minimum costs are used for the sub-problems. Different partial orders can be used depending on the costs to be minimized. To use the concept of partial order, intermediate solutions of sub problems are classified into 'buckets' that separate them in an increasing order of costs for structural element of the bonded-pair representation. The initial functions are stored into the respective buckets. This concept is illustrated in Fig. 2. The buckets are computed in the order of growing costs, so that the first solution found has minimum cost.

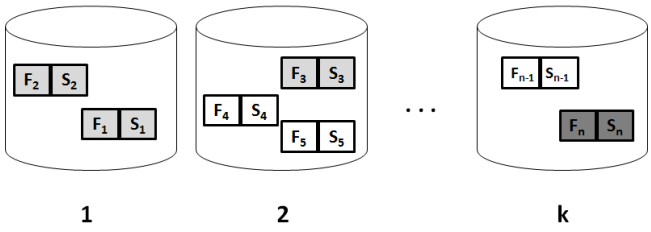

Figure 2: Illustration of buckets in FC: light gray bonded-pairs are the initial functions, the white bonded-pairs are intermediate functions and dark gray bonded-pair is the target function, located in the k-bucket.

2.5. Allowed Sub-Functions

The great number of intermediate functions created by exhaustive combination can make the FC approach unfeasible. For performance optimization, a hash table of allowed functions can be pre-computed before the algorithm starts. The allowed sub-functions are found by applying a heuristic algorithm that selects useful functions, given a problem. Functions that are not present in the allowed functions hash table are discarded during the processing. The use of the allowed functions hash table helps to control the execution time of the algorithms. FC can (in some cases) achieve better results according to the number of allowed functions. For some cases, solutions can be guaranteed optimal even with a very limited set of allowed functions. This is the case of read once factoring [10], for instance. Several effort levels can be implemented for memory/execution time vs. quality tradeoff control. These effort levels can vary from a limited set of functions to an exhaustive effort including all possible functions.

3. General Flow

Fig. 3 shows a general flow chart for algorithms following the FC principles. First step is to parse the target function. Then the initial bonded-pairs are generated and checked against the target function. The allowed functions are computed and inserted in a separated set which is used to discard unwanted intermediate functions, reducing the solution space (memory) and the execution time. The initial bonded-pairs will be inserted in the buckets. The bonded-pairs are associated to compose new elements that will be inserted in the next bucket (according to the cost). These new bonded-pairs will be used in the sequence of the associations. The process will continue until the target function is found.

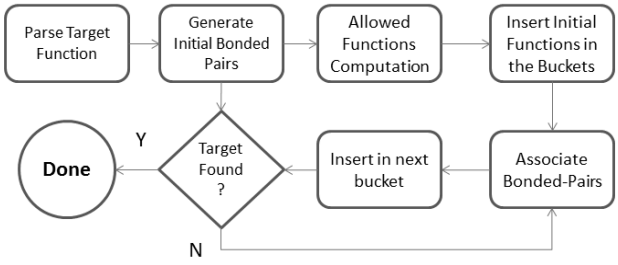

Figure 3: General flow chart for the FC approach.

4. Applications

In this section, we describe four distinct applications of FC: (1) Minimum Decision Chain computation, (2) Minimum SOP computation, (3) Boolean factoring with multiple objective goals and (4) reduced AIG construction. All these methods are particularizations of the general FC procedure presented in Fig. 3. Each specific algorithm is obtained by choosing adequately the bonded-pair representation, the initial functions, the bonded pair associations, an adequate partial order and associated dynamic programming as well as the allowed functions. The four approaches are described in the following.

4.1. Minimum Decision Chain Computation

The minimum decision chain (MDC) of a logic function is related to the number of transistors in series in switch networks that implement such function [8]. A prime implicant of an arbitrary function *f* can be viewed as a variable assignment for which two conditions are respected: (1) the output of the function f is true; and, (2) the removal of any assigned variable from the assignment makes *f* to become undetermined. This way, a set of prime implicants that cover the function can be viewed as a set of variable assignments that decide a function. The largest number of variables in a single assignment among a set of assignments that decide a function is the Decision Chain (DC) of the set. Different sets of assignments that decide a function are possible, and each set of assignments has its own DC. In the following we present a method to compute the Minimum Decision Chain (MDC) among all possible Decision Chains of a function. The adaption of FC to compute MDCs is described in [8].

MDC can be calculated using a top-down approach through a modified Quine-McCluskey algorithm (QMC-MDC), or a bottom-up process considering the FC strategy (FC-MDC) [8]. Some comparison results demonstrate that the FC method is more efficient than the Quine-McCluskey based method, as shown in Table 1.

Table 1: Execution time of on-set MDC computation [8].

Function set	Number of functions	Average MDC	QMC-MDC	**FC-MDC**
4-NPN	221	3.46	34 ms	**9 ms**
5-NPN	616,125	4.67	92 s	**104.3 s**
44-6.genlib	3,503	3.87	> 4 h	**5.9 s**

4.2. SOP Generation

We can use FC to compute a Sum-of-Products (SOP). The prime computation using FC takes advantage from the MDC computation to generate a SOP respecting the MDC with some algorithm modifications. A SOP that has the primary goal of respecting the MDC while minimizing literals is useful as the start point for algorithms of transistor network generation. The algorithm proposed by Kagaris [9] needs a SOP that respects the MDC as input, in order to generate a supergate that respects the MDC of the target function.

As the prime computation using FC has almost the same algorithm of the MDC computation, the execution times tend to be similar to those in Table I. One advantage is that the generated SOP can be forced to respect the MDC. The algorithm proposed by Kagaris [9] to generate supergates needs a SOP that respects the MDC as input, in order to generate a supergate that respects the MDC.

4.3. Boolean Factoring

In [18] we present an algorithm to compute minimum factored forms using FC. Factoring based on FC starts from literals and constructs larger functions by association. Costs are computed from the previously known simpler functions, which allow the method to minimize multi-objective cost functions [10]. Keep track of the number of series transistors (besides number of literals) is important for library free approaches [11. 12] aimed to use a cell generator [13].

The factoring process using FC was compared with QuickFactor (QF) and GoodFactor (GF) methods described in [14]; with the factoring available in ABC [15] and with the Xfactor method [16, 17]. FC based factoring has some important advantages against other methods: (1) FC factoring can minimize expressions with multiple objective goals beyond literal minimization, e.g. factored forms respecting the MDC value of the target function; (2) the results of FC factoring were always equal or better than the other methods; (3) for read-once functions, it always give the optimal result. Table 2 shows results comparing FC factoring approach to other factoring methods. However, FC factoring can be slightly slower as a method, but still completes in a reasonable time.

Table 2: Results of some MCNC benchmarks [18].

Logic function	SOP	QF	GF	ABC	XF	**FC**
b9_a1	56	12	12	12	12	**12**
rd53_0	20	14	14	14	12	**12**
rd53_1	80	28	28	28	46	**28**
cm162a_o	29	16	16	16	13	**12**
cm162a_p	36	18	18	16	14	**14**
cm162a_q	43	20	20	18	16	**16**
cm163a_r	31	13	13	13	13	**12**

4.4. AIG Rewriting

In [7], is presented an And-Inverter-Graph (AIG) rewriting approach using FC. This approach constructs AIGs from simpler graphs, while minimizing a cost function (e.g., the number of nodes or the graph height) [7].

AIG rewriting using FC has been compared to two different approaches: using the GF factoring [11] + FRAIG [19] and the FC factoring [10] + FRAIG [19]. The set chosen for the analysis is the 3,982 representative functions [20] of permutation equivalent classes of four input functions. Table 3 shows the sum of all nodes generated in the test and the average number of nodes. There is an improvement of around 5% in the number of nodes when compared to the ABC + FRAIG, and 2.2 % when compared to the FC factoring + FRAIG approach. Table 4 shows the sum of all logical depth in the experiment and the average logical depth. FC AIG rewriting has an improvement of around 16.88% in the logical depth when compared to the FC Factoring + FRAIG. For depth minimization, the partial order is based on graph depth.

Table 3: Comparison of AIGs number of nodes generated using different methods [7].

	ABC + FRAIG	*FC Factoring + FRAIG*	***FC AIG rewriting***
Number of nodes	*32,813*	*31,904*	***31,25***
Average number of nodes	*8.24*	*8.01*	***7.84***

Table 4: Comparison of AIGs Logical Depth using FRAIG and FC AIG rewriting [7].

	FC Factoring + FRAIG	***FC AIG rewriting***
Sum of Logical Depth	*17,933*	***15,356***
Average Logical Depth	*4.50*	***3.85***

5. Conclusions

This paper proposed a novel paradigm for performing logic synthesis, called functional composition (FC). FC is based in a bottom up approach using extensive composition of Boolean functions to have an efficient cost control, while enhancing quality by exploiting a large portion of the solution space. Four applications methods have been presented with promising results, demonstrating the potential for generalization of functional composition to be used for new applications in logic synthesis.

6. References

[1] R. L. Ashenhurst. The decomposition of switching functions. Computation Lab, Harvard University, vol. 29, pp.74-116, 1959.

[2] H. A. Curtis. A New Approach to the Design of Switching Circuits. Von Nostrand, 1962.

[3] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, T. Kam. "Reducing structural bias in technology mapping," ICCAD-2005, pp. 519- 526.

[4] T.Stanion, C.Sechen; A Method for Finding Good Ashenhurst Decompositions and its Application to FPGA Synthesis. DAC95, pp.60-64.

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

[6] A. Mishchenko, R. Brayton, S.Jang, V.Kravets, "Delay optimization using SOP balancing", IWLS 2011.

[7] T.Figueiro, R.P. Ribas, A.I. Reis. Constructive AIG optimization considering input weights; ISQED 2011, pp. 1-8

[8] M.G.A. Martins, V. Callegaro, R. P. Ribas, A. I. Reis. Efficient method to compute minimum decision chains of Boolean functions. GLSVLSI 2011, pp. 419-422.

[9] D. Kagaris, T. Haniotakis. A methodology for transistor-efficient supergate design. IEEE TVLSI. Vol. 15, N. 4, pp. 488-492.

[10] M.G.A.Martins, L.S.Rosa Jr, A.B. Rasmussen, R.P.Ribas, A.I. Reis. "Boolean factoring with multi-objective goals," ICCD 2010, pp.229-234.

[11] A.I.Reis. Covering strategies for library free technology mapping. SBCCI'99, pp. 180-183.

[12] V.Correia, A.Reis. "Advanced technology mapping for standard-cell generators," SBCCI 2004, pp. 254-259.

[13] J.D.Togni, F.R.Schneider, V.P.Correia, R.P.Ribas, A.I.Reis. "Automatic generation of digital cell libraries," SBCCI 2002, pp. 265- 270.

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

[15] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. December 2005 Release. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[16] M.C.Golumbic, A.Mintz. Factoring logic functions using graph partitioning. ICCAD '99, pp. 195-199.

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

[18] S. Yang, Logic Synthesis and Optimization Benchmarks User Guide Version 3.0, Technical Report 1991-IWLS-UG-Saeyang, MCNC Research Triangle Park, NC, January 1991.

[19] 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.
[20] V.P.Correia, A.I.Reis. "Classifying n-Input Boolean Functions". IBERCHIP 2001, pp. 58-66.