Boolean Decomposition Using Two-literal Divisors

Nilesh Modi
Universitat Politècnica de Catalunya
Barcelona, Spain

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

This paper is an attempt to answer the following question: how much improvement can be obtained in logic decomposition by using Boolean divisors? Traditionally, the existence of too many Boolean divisors has been the main reason why Boolean decomposition has had limited success. This paper explores a new strategy based on the decomposition of Boolean functions by means of two-literal divisors. The strategy is shown to derive superior results while still maintaining an affordable complexity. The results show improvements of 15% on average, and up to 50% in some examples, w.r.t. algebraic decomposition.

1. Introduction

Two main families of methods have been proposed for logic decomposition: *algebraic* and *Boolean*. Algebraic methods have had a widespread use due to their lower complexity, since logic expressions are manipulated as polynomials, without taking advantage of Boolean equivalences. Moreover, *don't care* sets are not used. On the other hand, Boolean methods are more powerful. For example, the function $F = \overline{a}be + bc + ac$ can be decomposed into $F = \overline{a}be + cD$, through the algebraic division by the divisor $D = a + b$. Through Boolean division, one can obtain the expression $F = (\overline{a}e + c)D$.

Given a divisor, Boolean division can be performed in different ways. One of the most used approaches is through two-level minimizers that can accept *don't care* information [2]. In the previous example, the substitution of $D = a + b$ into F can be done by incorporating the expression $D \oplus (a+b)$ into the *don't care* set of F. After two-level minimization, one would obtain $F = \overline{a}be + cD$.

If Boolean division is an affordable operation by using two-level minimizers, then why is Boolean decomposition not widely used? The answer is simple: there are too many Boolean divisors. Whenever $F \cdot D \neq 0$, D is a Boolean divisor of F and exploring all of them is prohibitively expensive.

We present an approach targetted at generating subject graphs, i.e. netlists with only 2-input gates and inverters. For that purpose, we claim that the set of "interesting" Boolean divisors is small and effective methods for Boolean decomposition can be devised.

The main goal of this paper is not to propose an efficient method for Boolean decomposition, even though one is proposed. The main goal is to explore the *limits* of Boolean decomposition and the potential improvements that can be obtained.

2. Overview and related work

This section presents an overview of the approach proposed in this work. The following function will be used as an example

$$F = ab\overline{e} + abg + \overline{a}ceg + \overline{a}cfg + bc\overline{e} + bd\overline{e} + deg + dfg$$

that can also be expressed with a 13-literal *factored* form:

$$F = b(\overline{e}(a + c + d) + ag) + g(e + f)(\overline{a}c + d)$$

The following trivial observation is the cornerstone of our strategy, which is in the same vein as in [16] for the calculation of algebraic divisors.

> *If the subject graph only has 2-input nodes, then at least one of them will have two primary inputs in its support. This node is a Boolean divisor of the function. In general, any Boolean divisor can be represented as the composition of 2-literal Boolean divisors.*

Now the crucial questions are:

- How many 2-literal divisors should be explored?
- Is the decomposition by 2-literal divisors effective?

The answer to the first question is simple: for an n-variable function, there are $4\begin{pmatrix} n \\ 2 \end{pmatrix} = 2n(n-1)$ different divisors (see Section 4.2), bearing in mind that Boolean division implicitly considers the complement of the divisor in the support of the remainder. For a 3-variable function with inputs a, b and c, following divisors should be explored: $\overline{a}\overline{b}, a\overline{b}, \overline{a}b, ab, \overline{a}\,\overline{c}, a\overline{c}, \overline{a}c, ac, \overline{b}\overline{c}, b\overline{c}, \overline{b}c, bc$. As an example, note that a divisor such as $a + \overline{c}$ corresponds to the complement of the divisor $\overline{a}c$.

The answer to the second question is what we would like to affirm with this paper. We propose an algorithm, described in Section 4 that gives very promising results (see Section 5). As an example, let us take the function F previously described. By using algebraic decomposition implemented in SIS [14], the netlist in Fig. 1(a) is obtained. Each gate corresponds to one of the nodes in the Boolean network after decomposition (`decomp -g` command in SIS). The decomposition to 2-input AND/OR gates leads to a netlist with 12 gates. The netlist in Fig. 1(b) has been obtained by BDS [18] using BDD-based methods for decomposition (10 gates). Finally, the netlist in Fig. 1(c) has been generated by the approach presented in this paper (9 gates).

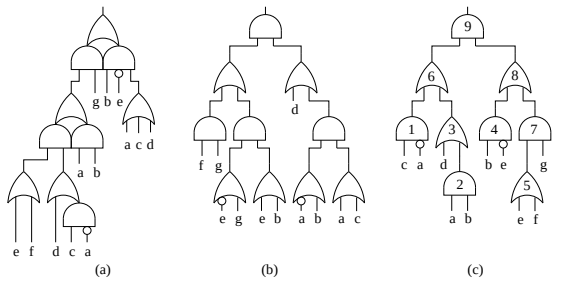

Figure 1. Different decomposition approaches.

Step(i)	#	Remainder	DC	Div.(x_i)
1	84	$b(\overline{e}(a+c+d)+ag)+g$ $(e+f)(\overline{a}c+d)$	0	$\overline{a}c$
2	84	$ab(\overline{e}+g)+(g(e+f)+b\overline{e})$ (x_1+d)	ax_1	ab
3	84	$x_2(\overline{e}+g)+(g(e+f)+b\overline{e})$ (x_1+d)	$x_2\overline{b}+$ x_2x_1	$d+x_2$
4	60	$(x_1+x_3)(g(e+f)+b\overline{e})$	0	$b\overline{e}$
5	60	$(x_1+x_3)(g(e+f)+x_4)$	ex_4	$e+f$
6	40	$(x_1+x_3)(gx_5+x_4)$	0	x_1+x_3
7	24	$x_6(gx_5+x_4)$	0	gx_5
8	12	$x_6(x_7+x_4)$	0	x_4+x_7
9	4	x_6x_8	0	x_6x_8

Table 1. Steps in the Boolean decomposition.

Each line of the Table 1 corresponds to one of the steps of our algorithm. The interpretation of each column is as follows:

# :	number of explored 2-literal divisors ($2n(n-1)$, n being the size of the support).
Remainder:	part of the function still to be decomposed.
DC:	*don't care* used for Boolean division.
Divisor:	extracted divisor at each step

To illustrate the power of the method, we will focus on two steps. In step 1, the divisor $x_1 = \overline{a}c$ is extracted. From the expression of the function, it is obvious that $\overline{a}c$ is an algebraic divisor. The algebraic division of F by $\overline{a}c$ gives the following *factored* form (12 literals):

$$F = b(\overline{e}(a+c+d)+ag)+g(e+f)(x_1+d)$$

On the other hand, the Boolean division derives the 11-literal expression shown in the table.

However, the main contribution of the approach in this paper is not the use of Boolean division, but the search of good Boolean divisors. And this is illustrated in step 3. The divisor $x_3 = d + x_2$ is not algebraic and it is not obvious how to find it by looking at the remainder. Here, the help of *don't cares* (DC) is crucial to find a good remainder after the Boolean division.

The DC for step 3 is calculated by accumulating the DC from step 2 (ax_1) and the *satisfiability don't care* (SDC) from divisor x_2, which is $x_2 \oplus ab$. Finally, those variables not in the support of the remainder (variable a in step 2) are *abstracted* out [13]. The expression to calculate the DC at step 2 is thus, $\forall a\ [ax_1 + (x_2 \oplus ab)] = x_2(\overline{b} + x_1)$.

2.1. Previous Work

The basic problem in decomposition has been identifying effective common subexpressions, or more formally divisors. Finding common algebraic divisors is a well-explored area.A bottom-up technique by Brayton and McMullen [3, 4] and an improvement to this technique in the form of a specialised algorithm [16] have been widely accepted in practice [14].

It is widely known, that Boolean divisors are more powerful than algebraic ones and theoretically lead to more compact, functionally equivalent circuits as shown by an example above in Fig. 1. Not surprisingly, very few attempts have been made in this direction because of the difficulty involved in finding effective Boolean divisors. The lack of an efficient way to generate Boolean divisors has also reduced the scope of some approaches [15]. Historically, the first attempt at this was by Karp [6]. Recently, there have been active attempts for finding common logic based on BDDs [1, 9–12, 17, 18] because of the fact that BDDs are an implicit decomposed representation of the function.

Yang, Ciesielski and Singhal proposed a BDD-based decomposition method based on dominators in [18], which is Boolean in nature. This approach, though being efficient, is applicable only to the decomposition of monolithic BDDs, relies on the variable ordering and does not necessarily give the best results in terms of area.

Kunz and Menon [8] have also proposed a recursive learning based technique for finding 2-input Boolean divisors. But the limitation of their approach, as compared to ours, is that they only consider divisors which are already present as nodes in the network, in other words their approach is highly dependent on the way circuit has been implemented.

The main difference between the technique in [7] and our approach is that in [7], the divisors are generated by using algebraic techniques (succesive factorization of sum-of-product covers). The decomposition is done by means of algebraic division augmented with the annihilation ($\overline{a} \cdot a = 0$) and idempotency ($a \cdot a = a$) laws.

3. Preliminaries

We begin by reviewing basic definitions and terminology.

The decomposition of a node in a Boolean network is its replacement by two or more nodes that form a subnetwork equivalent to the original node. The *support* of a Boolean function $sup(f)$ is the set of variables f effectively depends on. We would define $|F|$ to be an estimation of the *complexity* of function F. Here, we will take number of literals in the *factored* form as representation of the complexity of the given function, e.g. for the original function F from Section 2, $|F| = 13$.

Algebraic Division is an operation, given two Boolean functions F and D, returns Boolean functions Q and R such that, $F = Q \cdot D + R, Q \neq 0$ and $sup(Q) \bigcap sup(D) = \emptyset$. In *Boolean division* Q and D can have intersection of support-sets. Here, D is called a *Boolean divisor*. The assignment $x = f_x$ relates x with the variables in $sup(f_x)$. Conditions described by $x \neq f_x$, or equivalently by $x \oplus f_x$, which are never satisfiable, are called *satisfiability don't care* (SDC) conditions. If f_{x_i} is the *positive cofactor* and $f_{\overline{x}_i}$ is the *negative cofactor* of

$f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ w.r.t. x_i, then the *universal abstraction* $\forall$ of f w.r.t. a variable x_i is $\forall x_i\ f = f_{x_i} \cdot f_{\overline{x}_i}$.

4. Boolean Decomposition

4.1. Decomposition Procedure

In this section we present a simple, but effective, method for decomposition using 2-literal divisors.

Function `BOOL_DECOMP` (F, DC)
Inputs:
    Boolean Function F;
    Don't Care DC;
Returns:
    A netlist (represented as a set of 2-literal nodes);
begin
    x := new Variable; /* Output variable of a new node in the netlist */
    `if` ($|F| \leq 2$) `then`
        `return` $\{x = F\}$;
    DC_{proj} := Projection (DC, $sup(F)$);
    Best_R := F; Best_D := 0;
    `for each` pair of variables (a, b) in the fanin of F `do`
        `for each` divisor $D \in \{a\overline{b}, \overline{a}b, \overline{a}\overline{b}, ab\}$ `do`
            DC_{new} := $DC_{proj} + (x \oplus D)$; /* Satisfiability DC added */
            R := TwoLevelMinimization (F, DC_{new});
            `if` ($|R| <$ |Best_R|) `then`
                Best_D := D; Best_R := R;
    /* The best divisor and the decomp. of the remainder are returned */
    `return`$\{\{x =$ Best_D$\} \bigcup$
    `BOOL_DECOMP` (Best_R, $DC_{proj} + (x \oplus$ Best_D$))\}$;
end

Figure 2. Decomposition Algorithm.

The decomposition of a Boolean Function F is performed recursively. It attempts to find a netlist with 2-literal nodes for the given Boolean function F. The main algorithm is shown in Figure 2. [Note: $|F|$ stands for number of literals of F in *factored* form]. The output is a netlist with 2-literal nodes.

The recursive paradigm behind the algorithm is as follows:

1. Find the best 2-literal Boolean divisor by exhaustive search.
2. Find the remainder corresponding to this divisor by applying Boolean Division.
3. Recursively apply the algorithm to the remainder.

The algorithm stops when a function F, with at most 2 literals, is found.

4.2. Generating Boolean Divisors

Out of the 16 (2^{2^2}) possible Boolean functions between any two variables a and b, we just consider ($ab, a\overline{b}, \overline{a}b, \overline{a}\overline{b}$) as four distinct binate functions between a and b , as complementation of these functions is taken care by the two-level minimization in the algorithm. While, Exclusive-OR ($a\overline{b} + \overline{a}b$) and Equivalence ($ab + \overline{a}\overline{b}$) can be obtained by using two of these four basic functions. Rest of the functions are either constant($0, 1$) or unate functions($a, b, \overline{a}, \overline{b}$). The above mentioned four basic functions serve as potential Boolean divisors with the fanin variables a and b for the given Boolean function F in the algorithm.

4.3. DC Projection and Propagation

Each step of the algorithm extracts a Boolean divisor from the function with its associated SDC. After step i, the accumulated DC would be $DC = \bigvee_i (x_i \oplus D_i)$, where x_i is the new variable introduced by divisor D_i. Representing this DC-set would make the approach impractical after the extraction of few divisors, since the support of the incompletely specified function would become too large.

For this reason, at each step the support of the DC is reduced to the support of the remaining function. Even though this reduction can theoretically lead to inferior solutions (since one of the variables out of the support could help to minimize the function), we have observed that this does not occur in practice.

If $F(X)$ is the function under decomposition with support X and $DC(X, Y)$ is the *don't care* in which Y are variables not in the support of F, the projection of $DC(X, Y)$ onto X can be calculated as follows [13]: $DC_{proj}(X) = \forall Y\ DC(X, Y)$.

5. Experimental Results

The approach presented in this paper has been implemented in SIS [14]. The results have been obtained by running different algorithms on combinational circuits from the IWLS'93 benchmark set [5]. They have been compared with SIS (`script.rugged, script.algebraic`) and BDS [18].

The comparisons have been performed on the decomposition of single functions obtained by extracting the fanin cone of each primary output. After that, the cone was collapsed and simplified. The resulting sum-of-products cover was used as the starting point for decomposition. The reported results have been obtained after area-oriented technology mapping (`map -m0 -AF`) onto the library `lib2.genlib`.

Table 2 reports results on the largest 50 functions on which we applied our algorithm. The column `Bool decomp` corresponds to the results obtained by our approach. For some cases (*my_adder.q0* and *pair.a7*), the area is drastically reduced due to power of Boolean division. The last row of the table 2 shows the average improvements w.r.t. other techniques (15% in area w.r.t. `algebraic` and 10% in delay w.r.t. `rugged`).

However, Boolean decomposition does not always provide better results (e.g. *cm150a* when compared to `algebraic` script in terms of area). We conjecture these situations can occur due to the inaccuracy in the estimation of the function cost after Boolean division (size of *factored* forms).

After experimenting with 1000 outputs from different circuits of IWLS'93 benchmarks, the summary of the results is shown in Table 3 , which confirms the effectiveness of our approach over different types of circuits. The results are better when dealing with larger circuits, since the effectiveness of algebraic and Boolean methods tend to be similar when the circuits are small.

The results also confirm that BDD-based methods, such as BDS, are not very effective when the functions do not have clear decompositions with disjoint support. The dependence on variable ordering and possible discrepancies between BDD size and complexity of the function may also be the reasons for the inferior results.

The computational complexity of our approach is still not competitive with algebraic methods. The current approach takes

Circuit	Output	rugged		algebraic		Bool Decomp.	
		area	delay	area	delay	area	delay
apex6	RPTWIN_P	36192	6.88	36656	6.61	36656	5.75
	RXZ0_P	43616	9.08	67280	9.73	37120	7.8
	TD_P	45008	8.67	48256	10.37	47328	8.33
apex7	KBG_F	38048	9.79	29232	9.32	32016	8.84
	STAR1_P	44544	8.56	29232	10.36	31088	8.18
	STAR2_P	38512	9.35	29696	10.87	30160	8.54
	START3_P	38976	9.35	32480	9.97	31552	8.89
	VERR_F	45008	8.64	37584	9.57	37584	7.32
cm150a	v	45008	7.89	43152	8.15	45472	7.76
des	C_new0	52432	9.95	51968	9.29	44544	6.97
	C_new1	52432	9.95	51968	9.29	46400	8.52
	C_new14	52432	9.95	53824	9.23	49184	6.72
	C_new26	52432	9.95	51968	9.29	43616	6.97
	D_new26	52432	9.95	53824	9.23	42224	6.97
	D_new27	52432	9.95	51968	9.29	42688	6.97
f51m	44	54752	7.66	56608	7.38	31088	7.25
	45	48256	6.39	40368	6.56	27376	7.07
	46	32944	6.49	36192	7.47	33872	6.07
frg2	a9	58928	8.45	33408	9.41	38048	10.31
	m9	34336	7.96	34336	7.96	31552	7.18
	n9	32944	7.76	35728	8.09	33408	6.81
	p9	47328	10.15	70064	13.84	39440	7.34
	s9	70528	9.29	43616	11.76	45472	9.77
k2	f2	120176	11.06	108112	9	111360	10.28
	g2	89552	9.28	77488	9.83	83056	8.8
	h2	33872	6.24	32480	7.1	36656	5.36
	u1	52896	8.2	51040	7.64	48720	8.39
	w1	89552	10.05	75632	9.85	73312	10.73
	z0	83984	8.41	80272	10.26	84448	9.93
my_adder	p0	51504	9.4	68672	19.75	41760	10.82
	q0	65888	9.63	45936	19.14	34336	9.5
	r0	41296	8.11	65424	13.62	32480	8.65
pair	a10	47792	10.23	52432	14.46	31552	7.29
	a7	90944	13.44	69136	12.55	48256	9.16
	b9	52432	10.37	52896	10.18	36656	10.37
	d10	45936	10.77	45936	11.48	38976	8.29
	d6	39440	10.69	52432	9.47	30160	6.84
	e10	48256	10.77	51040	10.48	38512	7.78
	r7	32944	8.36	35728	8.45	35264	6.9
	s7	37120	9.41	38512	8.79	31088	8.18
	z6	52432	10.37	52896	10.18	37584	10.89
rot	n6	51968	8.42	64496	8.9	34800	7.27
	q7	27840	8.77	29696	6.28	31088	6.09
t481	v16.0	39440	6.65	51504	8.37	35728	6.24
term1	r0	56608	8.42	38976	9.72	45008	10.02
	s0	54288	10.19	39904	9.71	44544	9.42
vda	s	37120	8.44	36656	7.02	38512	6.64
	y	37584	8.92	46400	7.19	45472	7.32
x3	u5	54752	8.56	42688	10.38	46864	10.01
	z5	43616	9.08	67280	9.73	37120	7.8
Total		2548752	454.3	2493072	492.57	2111200	405.3
% Improvement of Bdec		17.16 %	10.79 %	15.32 %	17.72 %	-	-

Table 2. Results for the largest 50 functions.

between two and three orders of magnitude the time spent by algebraic decomposition. The next section discusses some of the improvements foreseen for the future to make this approach competitive in terms of CPU time.

	% improvement of **Bdec** w.r.t.		
Script	**rug**	**alg**	**BDS**
Area	8.95 %	5.70 %	20.08 %
Delay	9.04 %	10.10 %	14.98 %

Table 3. Summary of results.

6. Conclusions

A conceptually very simple as well as intuitive algorithm for Boolean decomposition using two-literal divisors has been presented. The main part of the algorithm is the search for effective Boolean divisors.

This work is a preliminary step towards proposing more effective techniques. The main conclusion is that finding Boolean divisors is not difficult, and promising results confirm that improvements that can be obtained by using them are relevant and worth the effort.

As future work, we plan to propose prunning techniques to reduce the search space for Boolean divisors by focusing on binate variables and by using dynamic programming methods. Reductions between one and two orders of magnitude are foreseen in run-time with these techniques.

Acknowledgements

This work has been partially supported by CICYT TIC2001-2476, a gift from Intel Corp. and a fellowship from IGSOC.

References

[1] V. Bertacco and M. Damiani. The Disjunctive decomposition of logic functions. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 78–82, Nov. 1997.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Acad. Publishers, 1984.

[3] R. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proc. of the Int. Symp. on Circuits and Systems*, pages 49–54, May 1982.

[4] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on CAD/ICAS, CAD-6*, November 1987.

[5] IWLS'93 benchmark set. `http://www.cbl.ncsu.edu/CBL_Docs/lgs93.html`.

[6] R. M. Karp. Functional decomposition and switching circuit design. *Journal of Society of Industrial App. Mathematics*, 11(2):291–335, June 1963.

[7] V. Kravets and K. Sakallah. M32: A constructive multilevel logic synthesis system. In *Proc. ACM/IEEE Design Automation Conf.*, pages 336–341, 1998.

[8] W. Kunz and P. R. Menon. Multi level logic optimization by implication analysis. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 6–13, Nov. 1994.

[9] Y.-T. Lai, K.-R. Pan, and M. Pedram. OBDD-based function decomposition: algorithms and implementation. *IEEE Trans. on Computer-Aided Design*, 15(8):977–990, Aug. 1996.

[10] A. Martinelli, R. Krenz, and E. Dubrova. Roth-Karp decomposition combining functional and structural techniques. In *Proc. Int. Workshop on Logic Synthesis*, California, May 2003.

[11] Y. Matsunaga. An exact and efficient algorithm for disjunctive decomposition. In *SASIMI'98*, pages 44–50, Oct. 1998.

[12] T. Sasao. FPGA design by generalized functional decomposition. In *Logic Synthesis and Optimization*, pages 233–258. Kluwer Acad. Publishers, 1993.

[13] H. Savoj and R. Brayton. The use of observability and external don't-cares for the simplification of multi-level networks. In *Proc. ACM/IEEE Design Automation Conf.*, pages 297–301, 1990.

[14] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[15] T. Stanion and C. Sechen. Boolean division and factorization using binary decision diagrams. In *IEEE Trans. on Computer-Aided Design*, volume 13, pages 1179–1184, Sept. 1994.

[16] J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 510–513, 1990.

[17] S. Yamashita, H. Sawada, and A. Nagoya. New methods to find optimal non-disjoint bi-decompositions. In *Proc. ACM/IEEE Design Automation Conf.*, pages 59–68, 1998.

[18] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proc. ACM/IEEE Design Automation Conf.*, pages 92–97, June 2000.