Efficient Boolean Division and Substitution Using Redundancy Addition and Removing

Shih-Chieh Chang and David Ihsin Cheng

***Abstract*— Boolean division, and hence Boolean substitution, produces better result than algebraic division and substitution. However, due to the lack of an efficient Boolean division algorithm, Boolean substitution has rarely been used. We present an efficient Boolean division and Boolean substitution algorithm. Our technique is based on the philosophy of redundancy addition and removal. By adding multiple wires/gates in a specialized way, we tailor the philosophy onto the Boolean division and substitution problem. From the viewpoint of traditional division/substitution, our algorithm can perform substitution not only in sum-of-product form but also in product-of-sum form. Our algorithm can also naturally take all types of internal don't cares into consideration. As far as substitution is concerned, we also discuss the case where we are allowed to decompose not only the dividend but also the divisor. Experiments are presented and the result is promising.**

***Index Terms*— Boolean functions, circuit optimization, circuit synthesis, design automation, division, logic design.**

I. Introduction

In multilevel logic synthesis, an important step in minimizing the area of a circuit is *substitution* [6] (or *resubstitution* [15]). Substitution refers to the step where a function is simplified in complexity by using an additional input that was not previously in the function's immediate fanins. Substitution can reduce the complexity of a function because part of the function is replaced by the additional input that represents some existing function in the circuit. The expression of the existing function is, therefore, shared and reused. To perform substitution, the concept of *division* plays a major role. Given two Boolean functions F and D, if we can express F in the form $F = Q \cdot D + R$, where $\cdot$ and $+$, respectively, represent the Boolean AND and Boolean OR operators, then we say that F can be *divided* by D and that functions Q and R are, respectively, the *quotient* and the *remainder*.

Substitution can be algebraic or Boolean, depending on if the underlining division is algebraic or Boolean. In algebraic division [6], logic expressions are treated as algebraic polynomials, with some restrictions placed on the manipulations of the polynomials. In particular, the product of two functions $F \cdot G$ is algebraic only if no variable appears in both F and G. As a consequence of the restriction, certain Boolean identities such as $x \cdot \bar{x} = 0$ and $x \cdot x = x$ do not exist. As an example, given $F = \bar{a}be + bc + ac$ and divisor $D = a + b$, through algebraic division we obtain $F = c(a + b) + \bar{a}be$. Through Boolean division, which can exploit all the properties in Boolean algebra [2], we obtain $F = (\bar{a}e + c)(a + b)$. Assuming a node d with function $a + b$ exists in the circuit, with algebraic substitution we then have $F = cd + \bar{a}be$, while with Boolean substitution we have $F = (\bar{a}e + c)d$. In this example, function F has six literals[1] before substitution. Algebraic substitution reduces the number of literals to five, while Boolean substitution reduces it to four. Boolean division and, hence, Boolean substitution, in theory produces better results. However, there does not exist a general and efficient Boolean division algorithm. In terms of the above example, this means that the best result of reducing F to four literals is very difficult to achieve.

Although there does not exist a general and efficient algorithm to perform Boolean division, to certain degree a few approaches have been partially successful. The first technique, or actually an *ad-hoc* setup, is based on a good two-level optimizer. Since a good two-level optimizer, such as Espresso [3], is able to take don't cares into consideration, we can actually force it to achieve the effect of Boolean division. For example, given a function F and a divisor $d = a + b$, we can put F through Espresso with $d \oplus (a + b)$ as the don't cares, and furthermore force Espresso to take literal d into the final result, thereby achieving the effect of Boolean division. Another technique that is able to perform Boolean division is proposed in [9]. By adding two Boolean identities, $x \cdot \bar{x} = 0$ and $x \cdot x = x$, onto the traditional algebraic algorithm, the concept of *coalgebraic division* is introduced. The coalgebraic division algorithm exploits the two Boolean identities for possible modification of the quotient obtained through algebraic operations. For a simple example, if we perform abc divided by ab, algebraic algorithm would return the quotient as c. Adding the Boolean identities, coalgebraic division modifies the possible quotients to $\{c, ac, bc, abc\}$ and eventually chooses one of them that produces a good result. Another technique, based on the binary decision diagram (BDD) data structure, is proposed in [14]. Given a function F and a divisor D, the method is built on the fact that $F = DF_D + \bar{D}F_{\bar{D}}$, where the subscripts denote the generalized cofactor operator [6]. From the viewpoint of F divided by D, this fact means that the quotient is F_D and the remainder is $\bar{D}F_{\bar{D}}$. All the functions in this method are represented in

Manuscript received April 21, 1998; revised October 7, 1998. This work was supported in part by the Taiwan National Science Council (NSC) under Grant 88-2215-E-194-005. This paper was recommended by Associate Editor A. Saldanha.

S.-C. Chang is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Min-Hsiung, Chia-Yi 621, Taiwan (e-mail: scchang@cs.ccu.edu.tw).

D. I. Cheng is with Ultima Interconnect Technology, San Jose, CA 95136 USA (e-mail: ihsin@guitar.ece.ucsb.edu).

Publisher Item Identifier S 0278-0070(99)05680-8.

[1] In *factored form* [6], as opposed to sum-of-product form.

0278–0070/99$10.00 1999 IEEE

BDD's and the cost function of optimization is also based on some features on BDD's.

In this paper, we first present a new technique to perform Boolean division. Our technique is based on the concept of *redundancy addition and removal* (RAR) discussed in [4], [5], [7], and [12]. The basic philosophy of the RAR technique is to first add some redundancy and then remove other redundancies elsewhere, with the goal that the removed ones reduce the circuit size more than the added one. With a fixed setup that is specially configured, we tailor the RAR philosophy onto the Boolean division problem. Unlike traditional RAR techniques, which require redundancy checking on the potential wire to be added, our algorithm is tailored in a way that we know a priori that our interested potential wire is redundant. Also, although quite effective on adding one redundancy and then removing other redundancies, the traditional RAR techniques have little success on trying to add multiple wires/gates. In our algorithm, the traditional RAR philosophy is tailored to add multiple wires/gates in a specific way particularly for the Boolean division problem.

As far as substitution is concerned, knowing how to perform division is only the first step. The second step is to choose potential divisors. Traditionally, substitution on a function F is done by going through the existing nodes in the circuit and treating each of them as a potential divisor of F. Division is tried on each potential divisor and substitution is carried out when the trial is favorable. Since it is up to the underlining division algorithm to conclude whether a divisor is good or not, the algorithm may miss some "good" divisors. In the example mentioned earlier, let us say the node with function $a+b$ does not exist and, instead, a node with function $D = a + b + x$ exists. Since function F does not depend on variable x, a traditional division algorithm would quickly conclude that the quotient of function F divided by D is zero and, therefore, no substitution would occur. However, if we slightly change the circuit structure by decomposing $a + b + x$ to two nodes $n_1 = a+b$ and $n_2 = n_1+x$, function F can then be substituted with node n_1. We will use the term *basic division* to refer to the scenario where the given divisor is not allowed to be decomposed, and the term *extended division* for the scenario where the divisor is allowed to be decomposed, certainly with some purpose in mind. In the above example where function $F = \bar{a}be + bc + ac$ is divided by $D = a + b + x$, we would say that under basic division the quotient is zero. For the same F and D we would also say that under extended division the subexpression $a+b$ can be extracted out as a new divisor, and with the new divisor $a + b$ the quotient is $\bar{a}e + c$. From this viewpoint, all the traditional division algorithms perform only basic division, while our algorithm presented in this paper performs extended division.

Traditional substitution approaches operate on each node's internal sum-of-product data structure and, hence, can only perform substitution/division in the sum-of-product form. In contrast, our algorithm operates on circuit structure directly. Given an initial circuit, the first step of our algorithm is to decompose each node's internal sum-of-product form into a two-level AND and OR gates. The circuit then, in general, has a level of AND gates, followed by a level of OR gates, and so on. As a result, in addition to the traditional sum-of-product type of substitution, our algorithm can also perform substitution in the flavor of product-of-sum form. In other words, in two-level form, whether the dividend/divisor are a bunch of AND's followed by an OR, or a bunch of OR's followed by an AND are completely symmetric to us. For example, let $F = (\bar{a} + b + e)(b + c)(a + c)$ and $D = (a)(b)$ be existing nodes. With our algorithm we can quickly substitute D into F and obtain $F = D + (\bar{a} + e)c$, i.e., $F = ab+(\bar{a}+e)c$. Performing substitution in such a manner is completely not possible in the traditional approaches because of the strong attachment to the underlining sum-of-product expression, while in our technique performing substitution through sum-of-product form or product-of-sum form are basically the same.

Another feature of our algorithm is the ability to naturally handle don't cares. Traditional techniques either totally cannot handle don't cares or can only handle don't cares in an *ad-hoc* way. Since our algorithm is based on the RAR technique, which performs so called *implications*, we can take any internal don't cares into account naturally. Furthermore, since various types of implication algorithms exist [10], [11], [13], we can in fact adjust the tradeoff between the run time and the amount of don't cares we take into account.

The rest of the paper is organized as follows. Section II reviews the RAR technique. Section III provides a fundamental view of our algorithm focusing on basic division only. Section IV presents our complete algorithm. Section V shows some experimental results and, finally, Section VI concludes this paper.

II. Redundancy Addition and Removal

The most related work to our Boolean substitution algorithm is the technique of RAR. Here, we provide a detailed review. In [4], [5], [7], and [12], the technique of RAR is proposed and applied to general multilevel logic optimization. The basic philosophy in RAR is to add some redundancy first and then try to remove other redundancies elsewhere, with the goal that the removed ones reduce the circuit size more than the added one. We review the technique with an example circuit.

Fig. 1(a), without the dotted wire, shows an irredundant circuit. The dotted wire $g5 \rightarrow g9$ is a redundant wire, i.e., adding the wire does not change the circuit's functionality. However, once this wire is added, the two thick wires, $g1 \rightarrow g4$ and $g6 \rightarrow g7$, become redundant. In this case, we can remove these two redundant wires without changing the circuit's functionality. After removing these two wires, we then have the circuit shown in Fig. 1(b), which is smaller in size.

In general, the RAR technique first decides, based on some cost function, some existing irredundant wire that is the target to be removed. Then the technique searches for some nonexisting wire, sometimes called a candidate connection, that once added can remove the target wire. Finally, the technique checks whether the candidate connection is redundant, i.e., whether adding the nonexisting wire preserves the circuit's functionality. Only when the candidate connection is verified

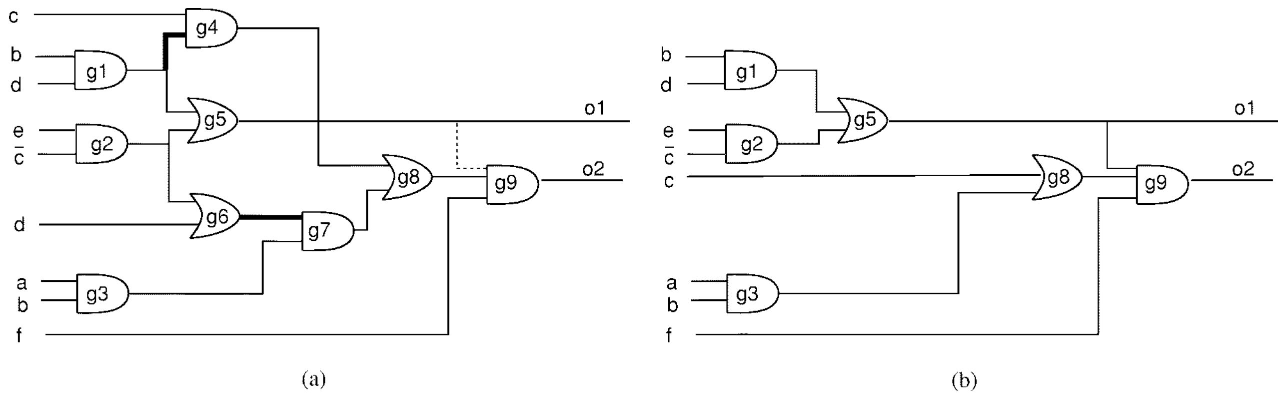

Fig. 1. The RAR technique.

as redundant, we can then add the connection and further remove the target wire. Note that most of the RAR techniques only try to incrementally add one wire at a time. Due to a large search space, efforts that try to add multiple wires/gates and remove even more wires/gates have only little success (e.g., [4]).

III. Basic Division

Given a function F and a divisor D, we use the term *basic division* to refer to the scenario where the divisor is not allowed to be decomposed, and use the term *extended division* to refer to the scenario where the divisor can be freely decomposed, with some optimization goal in mind. In this section, we focus on basic division.

A. SOS and POS of a Function

We first need some definitions. A *product term*, or *cube*, is a set of literals AND'ed together. A *sum term* is a set of literals OR'ed together. A function f_1 *contains* a function f_2 if the on-set of f_1 contains the on-set of f_2. As an example, function (cube) a contains function (cube) ab; function (sum term) $a+b$ contains function (sum term) a. Furthermore, we define SOS and POS of a function as follows:

SOS) Given a function F in two-level sum-of-product form, we say a function G, also in sum-of-product form, is a *sum-of-subproduct* (SOS) of F if every cube in F is contained by at least one cube in G.

POS) Given a function F in two-level product-of-sum form, we say a function G, also in product-of-sum form, is a *product-of-subsum* (POS) of F if every sum term in F contains at least one sum term in G.

For example, $D = a + b$ is a SOS of $F = \bar{a}be + bc + ac$ because every cube in F is contained by either cube a or cube b in D. For another example, $D' = a+b+x$ is also a SOS of the above F, since adding more cubes to D does not change the original containment relationship in F. On the other hand, function $E = ab + c$ is not a SOS of F, since cube $\bar{a}be$ is not contained in any cube in function E.

On the POS side, for example, $D = (a)(b)$ is a POS of $F = (\bar{a} + b + e)(b + c)(a + c)$ because every sum term in F contains either sum term a or sum term b in D. For another example, function $D' = (a)(b)(x + y)$ is also a POS of the above F, since adding more sum terms to D does not change the original containment relationship in F. On the other hand, function $E = (a + b)(c)$ is not a POS of F, since sum term $\bar{a} + b + e$ does not contain any sum term in function E.

The concepts of SOS and POS play a central role in our algorithm, and we now look at some of their simple properties.

Lemma 1: Let function G be a SOS of function F. Then $F = F \cdot G$.

Proof: Since $F \cdot G$ is an AND operation, and an AND operation can only reduce, but never increase, the set of minterms in F. The lemma holds if we can prove that all the cubes in F are still in the final sum-of-product of $F \cdot G$. Since each cube c_i in F is contained by at least one cube d_j in G and $c_i \cdot d_j = c_i$, all the cubes in F must be in the final sum-of-product of $F \cdot G$. □

Lemma 2: Let function G be a POS of function F. Then $F = F + G$.

Proof: By similar arguments of the proof in Lemma 1. □

These two lemmas establish the ground where we can tailor the technique of RAR onto our substitution problem. To illustrate the concept, we take the example of $F = \bar{a}be+bc+ac$ and $D = a + b$ from Section I. Since D is a SOS of F, by Lemma 1, the new function $F_{\rm new} = (a + b)(\bar{a}be + bc + ac)$ must be equivalent to the original function F. From the RAR viewpoint, we have successfully "added" a redundancy into the circuit. Focusing on the original F part inside $F_{\rm new}$, we then try to remove as many redundancies as possible, and can quickly arrive $F = (a + b)(\bar{a}e + c)$. Symmetric to the SOS case, we can perform similar operations on POS. Let $F = (\bar{a} + b + e)(b + c)(a + c)$ and $D = (a)(b)$. Since D is a POS of F, by Lemma 2, the new function $F_{new} = (a)(b)+(\bar{a}+b+e)(b+c)(a+c)$ must be equivalent to the original function F. Focusing on removing redundancies from the original F part inside $F_{\rm new}$, we then quickly have $F = ab + (\bar{a} + e)c$.

B. Performing Basic Division

Given a function F and a divisor D, in this section we present an algorithm that performs basic Boolean division, i.e., $F = Q \cdot D + R$. The best way to explain our algorithm is to discuss it with an example. Fig. 2(a) shows two nodes, which correspond to $F = \bar{a}be + bc + ac + y$ and $D = a + b$. Since

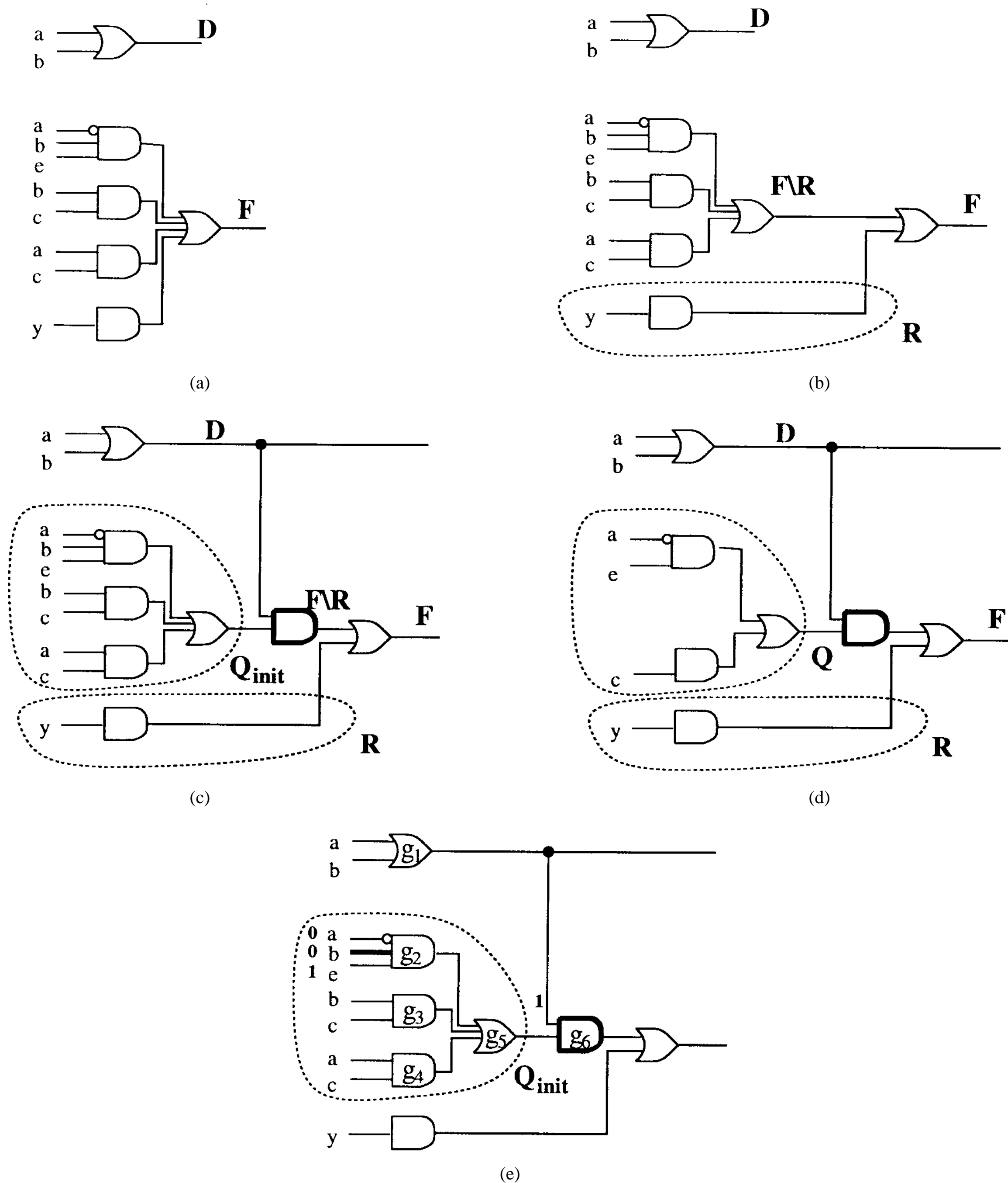

Fig. 2. Basic division.

our central idea is based on the SOS concept, the first step to perform F divided by D is to take out from F all the cubes that are not contained by any cube in D, and such cubes will be our final remainder term R. Among the four cubes in F, y is the only such cube since $y \not\subseteq a$ and $y \not\subseteq b$. Fig. 2(b) shows the circuit structure after we form the remainder y, where we use dotted circle R to indicate the remainder region and $F\backslash R$ to denote the resulting function with cube y taken out from F. Since every cube in $F\backslash R$ is now contained by at least one cube in D, D is a SOS of $F\backslash R$. By Lemma 1, $F\backslash R$ would stay unchanged if AND'ed with D. This fact is shown in Fig. 2(c) with an extra bold AND gate and the shift of $F\backslash R$ from before this AND gate to after this AND gate. From the viewpoint of the RAR technique, we have successfully added a redundancy and the circuit still has the same functionality. Now the region marked by the circle Q_{init} is highly redundant. The final step is to perform redundancy removal on the Q_{init} region and we reach the final result shown in Fig. 2(d), which is of the form $F = Q \cdot D + R$. To show how redundancy removal is done, we duplicate the circuit snapshot shown in Fig. 2(c) to (e) and remark some nodes. Let us illustrate how wire $b \rightarrow g_2$, the thick wire in Fig. 2(e), is detected as a redundant wire. For wire $b \rightarrow g_2$ stuck-at-one fault to be testable, b must be 0 to activate the fault. For the fault effect to propagate through gate g_2, a must be 0 and e must be 1. For the fault effect to propagate through gate g_6, g_1 must be

1. Since $a = 0$ and $b = 0$ implies $a + b = 0$, gate g_1 must be 0, which is a conflict. A conflict during the implication process means the fault $b \rightarrow g_2$ stuck-at-one is untestable and, therefore, wire $b \rightarrow g_2$ can be replaced by a constant 1. Our basic division algorithm works as illustrated by the above example. In summary, our algorithm consists of three steps. The first step of our algorithm is to decompose the dividend F so that the cubes that make the divisor D not a SOS of F form the remainder R. The second step is to AND D with $F \backslash R$, which does not change the functionality of $F \backslash R$ by Lemma 1. The third step is to remove all the redundancies inside the $F \backslash R$ region.

Note that it is the RAR steps that make our technique Boolean. Comparing to the traditional RAR techniques, however, a major difference lies on the fact that we know *a priori* that the added wires/gates are redundant because of the SOS property in Lemma 1. In other words, unlike the traditional RAR techniques, we do not need to check if the added wire/gate are redundant or not. Furthermore, as mentioned in Section II, there has been little success in works trying to generalize the RAR technique to adding multiple wires/gates. What our algorithm does is essentially a tailored version of the RAR philosophy onto the substitution problem, with a fixed configuration of multiple wires/gates addition. Also note that since the added wires/gates are known to be redundant a priori, the most time-consuming step in our algorithm is only on the redundancy removal step. As mentioned earlier, with different implication methods we can actually adjust the tradeoff between the run time and the quality of result. For example, we can limit our implication process only inside a small region, the $F \backslash R$ region plus the D region. As far as substitution is concerned, most of the reconvergences and implication conflicts would occur in this small region. Limiting the implication process inside this small region would greatly reduce the time required as opposed to a traditional redundancy removal process. On the other hand, we can certainly spend more time to perform implications to gates outside this small region, and thereby can naturally incorporate any internal don't cares into consideration. In the extreme case, we can even adopt some quite exhaustive implication technique such as recursive learning [11] to incorporate a large amount of internal don't cares. We do not discuss the details here but simply point out the existence of such a flexibility on various implication algorithms. Finally, as can be seen from the above example, our algorithm operates on circuit structure directly, rather than manipulating expressions like traditional approaches. As mentioned earlier, we are therefore not limited to performing substitutions in terms of the traditional sum-of-product viewpoint. With the POS concept, we can also perform substitutions on two functions when they are both in the product-of-sum form. Instead of using the SOS concept and Lemma 1, we can use the POS concept and Lemma 2, and the same philosophy as illustrated above would apply directly. As a simple example, imagine a circuit that is identical to the one shown in Fig. 2 with all the AND gates changed to OR gates and vice versa. With our algorithm it is as easy as was illustrated in this section, while in a traditional substitution technique all the sum-of-product expressions form a complete new problem whose result is difficult to predict. Since conceptually SOS and POS are symmetric, throughout the remaining of this paper we do not go into the details of the case for POS.

IV. Extended Division

Section III presented our algorithm that performs basic division, where a divisor is not allowed to be decomposed. Given a function F and a divisor D, under basic division we seek to reexpress F as $F = Q \cdot D + R$. This means we are allowed to decompose only on F but not on D. In this section we present an algorithm that performs what we call extended division. Given a function F and a divisor D, under extended division we are allowed to decompose not only F but also D, with the purpose of minimizing the number of literals in substitution. In essence, we first want to separate the cubes in D into two groups, the *core divisor* D_C and the *remaining divisor* D_R. Once this separation is determined, we decompose the original divisor D into two nodes such that $D = D_C + D_R$. Decomposing into a new node for the core divisor D_C means that D_C, a subexpression that was originally embedded in the given divisor D, is now exposed and can be used for substitution. We then apply our basic division algorithm in Section III on function F and core divisor D_C to obtain the result. For example, given function $F = \bar{a}bc + bc + ac + y$ and divisor $D = a + b + x$, we decompose the divisor D into the core divisor $D_C = a + b$ and the remaining divisor $D_R = x$. Applying our basic division algorithm on F and D_C, we then obtain the same result as illustrated in Section III. It should be clear that the most important thing here is to intelligently determine the core divisor D_C, since once D_C is determined an extended division reduces to a basic division.

Recall that during our basic division algorithm, it is the step of redundancy removal that really performs the minimization process. Looking back in Fig. 2(c), whenever we remove a wire from the cubes in the Q_{init} region, we effectively reduce a literal in the final quotient. What we would like to have is a core divisor that is able to remove the most wires. To determine the core divisor D_C with a given function F and a given divisor D, our basic idea is to have each wire in the cubes of F "vote" for a candidate core divisor. For each wire w in the cubes of F, we perform implications to see which cubes in divisor D are able to remove wire w. For example, let function $F = abd + \bar{a}c + cd + ae$ and divisor $D = ab + c + bd + e$, whose circuit structure is shown in Fig. 3(a). In Fig. 3(a), we name divisor D's four cubes y_1, y_2, y_3, and y_4; we also name function F's four cubes x_1, x_2, x_3, and x_4, which are, respectively, driven by gates g_1, g_2, g_3, and g_4. Consider wire $a \rightarrow g_1$ stuck-at-one fault. We have the following implications:

$a = 0$ (to activate the fault)	$\Rightarrow$	$y_1 = 0$
$b = 1$ and $d = 1$ (to allow fault effect thru g_1)	$\Rightarrow$	$y_3 = 1$
$x_2 = 0$ (to allow fault effect thru g_5) and $a = 0$	$\Rightarrow$	$c = 0$
$c = 0$	$\Rightarrow$	$y_2 = 0$

Assume that we somehow have determined a core divisor D_C. This core divisor, in our specialized configuration for basic division, feeds into a gate similar to the bold AND gate g_6 in Fig. 2(e) of Section III. This means that if we want

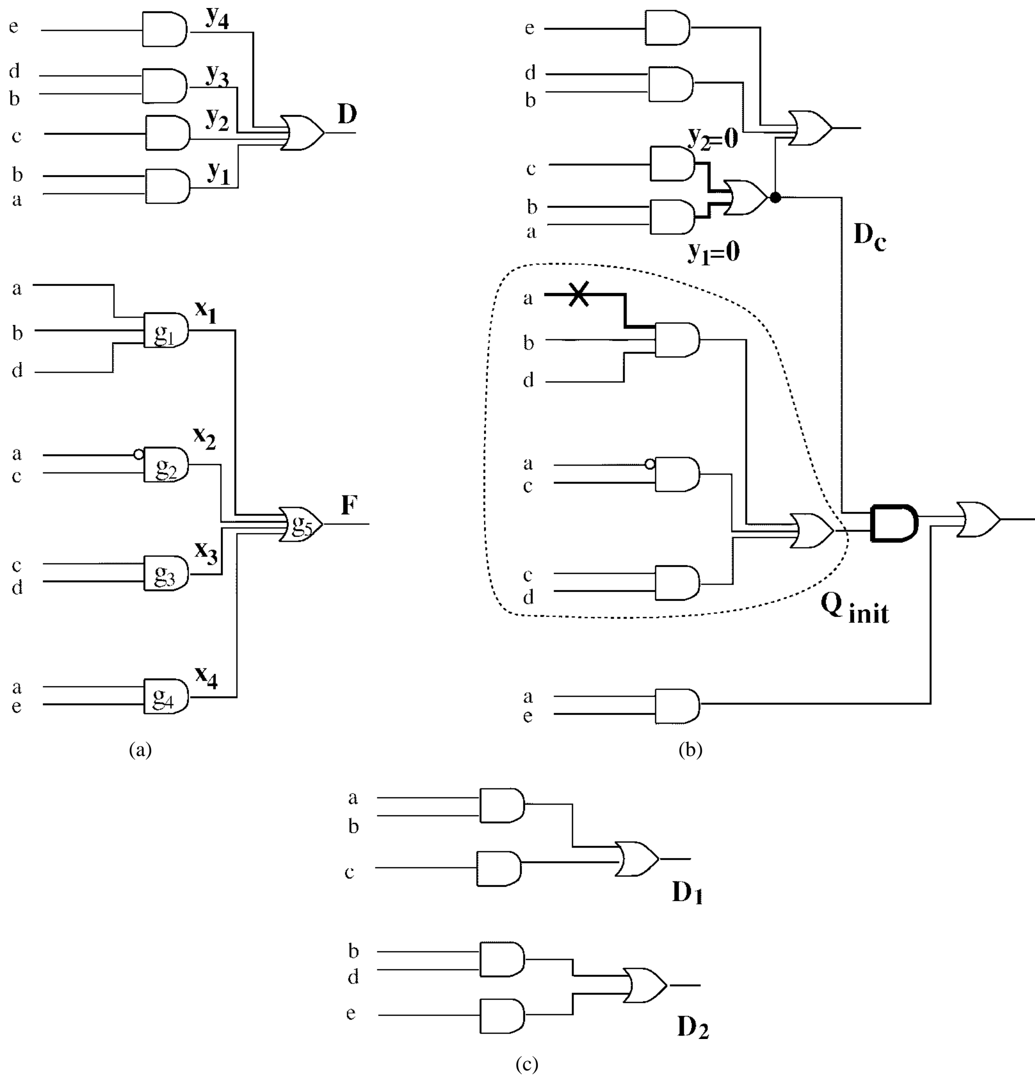

Fig. 3. Extended division.

any fault effect in the Q_{init} region to propagate through the bold AND gate, this core divisor D_C must have a value one during the fault's implication process. In the case of extended division, if the core divisor that we eventually determine has implication value zero for a particular fault, the fault must be untestable because a conflict will occur with the required assignment of one mentioned above. We illustrate this point by continuing the example for wire $a \to g_1$ stuck-at-one fault. We focus on the results that appear on the y_i's side whose implication values are zero. In this case, we have $y_1 = 0$ and $y_2 = 0$. Assuming we eventually choose $y_1 + y_2$ as our final core divisor, i.e., $D_C = y_1 + y_2 = ab + c$, then our basic division algorithm in Section V would change the circuit structure to the one shown in Fig. 3(b), where $D_C = ab + c$ is connected to the bold AND gate. Following the basic division algorithm in Section III, we would try to remove as many wires as possible in the Q_{init} region. When we again perform implications for the fault $a \to g_1$ stuck-at-one, shown with a cross in Fig. 3(b), we know that y_1 and y_2, and hence D_C, all have implication value 0. This creates a conflict because, as stated earlier, for the fault effect of $a \to g_1$ stuck-at-one to propagate through the bold AND gate, D_C must be assigned one. In other words, if we do choose $y_1 + y_2$ as our core divisor, we expect wire $a \to g_1$ to be removed in the subsequent basic division. Now, in determining the core divisor, different wires have different implication values on the y_i's side in Fig. 3(a). In some sense, this means that each wire "votes" for a candidate core divisor. In the above example, wire $a \to g_1$ votes for candidate core divisor $y_1 + y_2$. This

TABLE I
VOTE TABLE

wire	$y_i = 0$
$a \to g_1$	y_1, y_2
$b \to g_1$	y_1, y_2, y_3, y_4
$d \to g_1$	y_3, y_4
$a \to g_2$	
$c \to g_2$	y_1, y_2
$c \to g_3$	y_1, y_2
$d \to g_3$	y_3, y_4
$a \to g_4$	y_1
$e \to g_4$	y_3, y_4

(a)

wire	$y_i = 0$
$a \to g_1$	y_1, y_2
$b \to g_1$	y_1, y_2, y_3, y_4
$d \to g_1$	y_3, y_4
$a \to g_2$	
$c \to g_2$	y_1, y_2
$c \to g_3$	y_1, y_2
$d \to g_3$	
$a \to g_4$	
$e \to g_4$	y_3, y_4

(b)

TABLE II
EXPERIMENTAL RESULTS (SCRIPT A)

circuit	init	sis		basic		ext.		ext.+GDC	
	lit.	lit.	cpu	lit.	cpu	lit.	cpu	lit.	cpu
9symml	244	244	0.90	234	1.95	234	1.90	231	13.59
C1355	562	562	1.31	558	1.99	558	2.06	558	7.28
C1908	678	631	2.55	597	1.64	597	1.68	595	3.29
C2670	928	843	3.09	831	3.20	829	3.31	817	12.83
C3540	1462	1452	7.26	1261	12.27	1261	12.06	1258	171.53
C432	260	260	0.51	214	1.13	223	1.14	214	1.83
C499	562	562	0.75	558	2.01	558	2.07	558	6.62
C5315	1995	1961	7.67	1910	8.32	1914	8.62	1900	35.85
C6288	4212	4212	37.27	3747	20.59	3747	21.31	3747	21.62
C7552	2605	2522	16.13	2413	14.33	2410	15.40	2398	123.71
C880	416	416	1.08	412	0.76	413	0.77	413	1.44
alu2	448	446	19.70	412	5.40	417	5.35	413	56.86
alu4	831	829	80.96	779	16.23	784	15.08	790	302.55
apex6	807	807	1.36	796	3.67	795	3.86	792	40.52
apex7	278	278	0.39	263	0.77	263	0.82	259	2.62
dalu	1998	2003	19.45	1753	53.28	1768	52.22	1737	1922.85
des	6048	6048	80.81	5712	168.37	5697	190.48	5693	6091.68
example2	361	362	0.65	336	1.57	339	1.65	336	9.54
frg2	1321	1285	8.17	1217	22.23	1215	24.19	1180	666.11
i10	2875	2861	13.63	2627	46.26	2619	46.22	2587	594.15
i8	1817	1817	13.46	905	64.62	915	65.78	908	1579.56
i9	750	750	2.98	729	15.13	729	17.82	733	400.09
rot	733	751	1.59	701	1.99	698	1.93	696	5.34
t481	2611	2608	78.61	2009	32.28	1958	28.45	1976	406.52
term1	341	325	2.87	302	1.73	302	1.60	295	9.39
too_large	1052	1052	14.04	999	7.35	1004	7.17	980	66.92
ttt2	261	259	0.84	233	1.15	238	1.19	228	6.29
x1	346	346	0.91	340	1.47	345	1.53	339	10.80
x3	1013	999	3.20	980	5.45	974	6.11	970	77.81
x4	549	526	1.54	538	3.34	526	3.83	518	44.40
s1196	612	614	3.03	577	6.11	573	6.11	569	117.55
s1238	679	682	3.55	599	7.56	607	7.87	602	94.57
s13207	3891	3855	33.42	2894	61.07	2836	63.15	2828	2746.19
s1423	644	644	2.31	621	1.88	624	1.95	621	7.22
s1488	767	764	4.89	722	21.25	714	19.75	710	536.32
s1494	777	773	4.94	724	22.31	713	20.04	709	608.37
s15850	4358	4339	30.89	3789	36.33	3760	38.64	3751	695.82
Total	50092	49688	507	44292	677	44157	703	43909	17500
%		0.81%		11.58%		11.85%		12.34%	

should become clear if we look at the complete situation after each wire performs implications on the example circuit shown in Fig. 3(a). Table I(a) lists all the y_i's that have implication value zero for each wire.

We explain the interpretation of Table I(a) by examples. The meaning of the second row is that we expect wire $b \rightarrow g_1$ to be removed if we choose $y_1+y_2+y_3+y_4$ as the core divisor. For simplicity, we also say that wire $b \rightarrow g_1$ *votes* for candidate core divisor $y_1+y_2+y_3+y_4$. Similarly, the meaning of the fourth row is that we do not expect wire $a \rightarrow g_2$ to be removed, regardless of whatever core divisor we choose. The remaining entries of Table I(a) can be interpreted in a similar way.

The above voting scheme demonstrates our criteria for choosing a good core divisor. However, from the RAR technique's viewpoint, one more thing we need to make sure is that a candidate core divisor is indeed a redundant wire which we can eventually "add" to the circuit. This is done by checking if the candidate core divisor voted by a wire w is a SOS of the cube that is connected to wire w. For example, from the first entry in Table I(a), the candidate core divisor of wire $a \rightarrow g_1$ is $y_1+y_2 = ab+c$. The cube that is connected to wire $a \rightarrow g_1$ is $x_1 = abd$. Since the candidate core divisor $ab+c$ is a SOS of cube abd, we know eventually if we add core divisor $ab+c$ into the circuit, the added wire will be a redundant wire and, therefore, the circuit functionality would not change. In Table I(a), the only candidate core divisors that do not hold for this condition are wires $d \rightarrow g_3$ and $a \rightarrow g_4$. The candidate core divisor for wire $d \rightarrow g_3$ is $y_3+y_4 = bd+e$, which is not a SOS of the corresponding cube $x_3 = cd$. On the case of wire $a \rightarrow g_4$, candidate core divisor $y_1 = ab$ is not a SOS of the corresponding cube $x_4 = ae$. We therefore need to delete these two entries in Table I(a), and we have our final *vote table*, shown in Table I(b).

To finalize the choice of the core divisor, various heuristics can be used. We reduce the above choice problem to a maximal clique problem [8] in graph theory. First we construct a graph. For each wire we create a vertex; there is an edge between two vertices v_1 and v_2 if the intersection of the corresponding candidate core divisors are not empty. For example, the intersection of the candidate core divisors between wires $a \rightarrow g_1$ and $b \rightarrow g_1$ is $\{y_1, y_2\}$, and hence there is an edge

TABLE III
EXPERIMENTAL RESULTS (SCRIPT B)

circuit	init	sis		basic		ext.		ext.+GDC	
	lit.	lit.	cpu	lit.	cpu	lit.	cpu	lit.	cpu
9symml	251	251	0.95	237	1.64	243	1.69	243	12.51
C1355	560	560	1.35	560	1.96	558	1.98	558	7.59
C1908	679	632	2.61	598	1.65	598	1.70	596	3.32
C2670	939	849	3.65	840	2.63	831	2.80	828	10.03
C3540	1635	1618	7.86	1300	17.39	1345	16.82	1332	315.26
C432	261	261	0.55	214	1.02	223	1.12	214	1.63
C499	560	560	0.82	560	1.94	558	1.96	558	6.50
C5315	1961	1927	8.02	1870	5.94	1864	6.44	1856	24.11
C6288	4212	4212	37.19	3747	20.64	3747	21.10	3747	21.71
C7552	2627	2534	15.94	2382	13.13	2390	14.38	2367	113.62
C880	414	414	1.10	411	0.70	412	0.74	412	1.30
alu2	466	465	19.85	433	5.43	438	5.40	441	58.16
alu4	864	857	81.63	821	16.57	833	15.70	829	328.24
apex6	817	817	1.46	803	3.57	802	3.79	800	24.81
apex7	281	281	0.44	261	0.80	263	0.79	259	2.48
dalu	1809	1765	17.07	1671	37.09	1609	38.07	1587	895.06
des	4920	4918	80.63	4728	110.30	4734	115.31	4733	3325.34
example2	366	367	0.66	344	1.63	343	1.71	344	9.18
frg2	1177	1135	6.82	1066	14.63	1071	15.04	1058	253.36
i10	2772	2757	13.23	2567	44.18	2558	46.17	2518	608.61
i8	1491	1491	14.04	1211	34.82	1221	39.21	1216	850.84
i9	749	749	3.07	730	10.77	730	14.16	687	413.90
rot	758	763	1.73	710	1.95	706	1.90	703	5.00
t481	1395	1313	58.80	1114	17.37	1111	15.66	1108	195.26
term1	357	349	2.93	319	1.40	323	1.31	309	5.45
too_large	1304	1302	36.40	1183	10.11	1124	10.26	1115	149.46
ttt2	260	257	0.90	247	1.04	251	1.06	242	4.67
x1	385	384	1.11	369	1.59	368	1.45	363	7.41
x3	975	966	3.36	932	4.77	946	5.01	930	27.62
x4	522	492	1.56	488	2.21	469	2.62	457	16.81
s1196	626	624	3.10	577	5.80	573	5.89	569	83.57
s1238	696	692	3.62	599	7.62	607	7.60	602	100.90
s13207	3569	3552	28.84	2894	37.57	2836	36.54	2828	931.91
s1423	652	652	2.58	621	1.87	624	1.88	621	6.55
s1488	762	758	4.97	722	18.48	714	17.25	710	467.52
s1494	756	754	5.05	724	16.97	713	16.09	709	453.01
s15850	4306	4269	29.81	3789	34.16	3760	35.71	3751	605.06
Total	47134	46547	503	42642	511	42496	526	42200	10347
%		1.25%		9.53%		9.84%		10.47%	

between the vertices corresponding to these two wires. For another example, since the intersection of the candidate core divisors between wires $a \rightarrow g_1$ and $d \rightarrow g_1$ is empty, there is no edge between the vertices corresponding to these two wires. The complete graph is shown in Fig 4. Each clique in this graph represents a core divisor D_C that, if chosen, is expected to remove all the wires corresponding to the vertices in the clique. As we can see from Fig. 4, one clique, marked with a dotted circle, consists of four vertices $a \rightarrow g_1$, $b \rightarrow g_1$, $c \rightarrow g_2$, and $c \rightarrow g_3$, with the corresponding intersected candidate core divisor being $y_1 + y_2 = ab + c$. In this case, we expect to remove the four wires, $a \rightarrow g_1$, $b \rightarrow g_1$, $c \rightarrow g_2$, and $c \rightarrow g_3$, if we choose $y_1 + y_2$ as the final core divisor. Another clique, marked by the three bold edges in Fig. 4, consists of three vertices $b \rightarrow g_1$, $d \rightarrow g_1$, and $e \rightarrow g_4$, with the corresponding intersected candidate core divisor being $y_3 + y_4 = bd + e$. This means we expect to remove the three wires, $b \rightarrow g_1$, $d \rightarrow g_1$, and $e \rightarrow g_4$, if we choose $y_3 + y_4$ as the final core divisor. The problem of finding the best core divisor that would potentially remove most wires

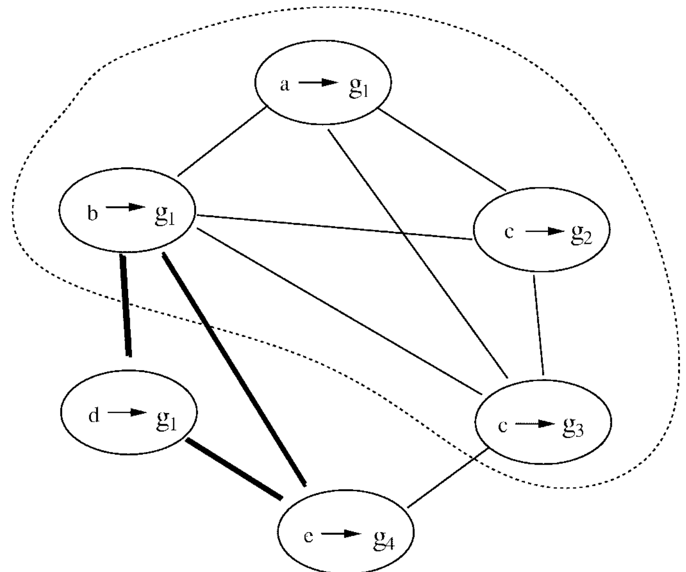

Fig. 4. Graph that represents the intersection of candidate core divisors.

TABLE IV
EXPERIMENTAL RESULTS (SCRIPT C)

circuit	init	sis		basic		ext.		ext.+GDC	
	lit.	lit.	cpu	lit.	cpu	lit.	cpu	lit.	cpu
9symml	258	258	0.98	237	2.14	241	2.08	241	16.12
C1355	562	562	1.34	558	2.01	558	2.08	558	6.77
C1908	677	628	2.65	584	1.76	584	1.83	583	3.86
C2670	934	846	3.35	822	3.15	815	3.62	815	12.45
C3540	1519	1499	7.85	1261	11.09	1261	11.05	1258	153.52
C432	261	261	0.55	214	1.12	223	1.13	215	1.68
C499	562	562	0.78	558	1.99	558	2.07	558	6.66
C5315	1996	1962	7.95	1910	7.92	1914	8.66	1900	36.07
C6288	4212	4212	37.46	3762	20.35	3762	20.20	3762	21.25
C7552	2605	2522	16.54	2413	14.34	2410	15.45	2398	125.08
C880	415	415	1.15	412	0.73	413	0.77	413	1.41
alu2	468	455	19.91	408	5.63	419	5.55	417	61.81
alu4	855	842	81.96	783	16.30	788	15.23	789	278.46
apex6	810	809	1.59	796	3.74	794	4.09	792	27.01
apex7	279	279	0.45	260	0.79	262	0.75	259	2.40
dalu	1774	1734	16.91	1581	35.08	1584	36.49	1536	882.73
des	6100	6069	91.28	5716	153.83	5706	172.52	5707	5997.51
example2	364	363	0.67	336	1.54	338	1.62	336	9.33
frg2	1299	1209	8.09	1168	19.32	1173	21.63	1137	545.20
i10	2872	2851	13.50	2617	46.30	2608	46.40	2574	610.70
i8	1575	1559	14.66	835	45.91	839	45.88	835	1088.36
i9	745	745	3.04	724	15.98	724	17.56	728	435.38
rot	747	727	1.84	702	1.95	705	1.85	703	5.63
t481	2485	2356	75.03	1956	21.73	1936	20.14	1920	244.70
term1	336	304	2.91	272	1.44	268	1.35	270	6.53
too_large	917	840	24.11	872	3.71	848	3.60	847	24.31
ttt2	273	243	1.02	232	1.15	236	1.20	235	9.10
x1	351	321	1.13	331	1.28	330	1.34	328	9.78
x3	965	887	3.79	909	5.57	900	5.94	897	75.50
x4	541	501	1.66	513	3.21	494	3.48	488	42.14
s1196	630	629	3.20	577	6.74	573	6.11	569	69.87
s1238	693	692	3.71	599	7.77	607	7.77	602	91.30
s13207	3826	3700	34.14	2894	41.88	2836	42.86	2828	1248.70
s1423	644	644	2.38	621	1.90	624	1.97	621	7.19
s1488	796	791	5.06	722	21.30	714	20.52	710	562.94
s1494	803	798	5.14	724	21.01	713	19.54	709	591.35
s15850	4355	4306	32.87	3789	35.91	3760	40.36	3751	687.62
Total	49504	48381	530	43668	587	43518	614	43289	14000
%		2.27%		11.79%		12.09%		12.55%	

is, therefore, reduced to a maximal clique problem. In this example, since the maximal clique is the one of size four, we determine the core divisor to be $D_C = y_1 + y_2 = ab + c$, with which we change the circuit structure to the one shown in Fig. 3(b). After performing redundancy removal, the four wires $a \rightarrow g_1$, $b \rightarrow g_1$, $c \rightarrow g_2$, and $c \rightarrow g_3$ are removed and, finally, we have $F = abd + \bar{a}c + cd + ae = D_C(\bar{a} + d) + ae = (ab + c)(\bar{a} + d) + ae$.

Applying our extended division algorithm to the substitution problem, we want to point out that we can actually do more than what the above discussion shows. In the above formulation, we focus only on one existing node D. In the case of substitution, we actually have freedom to select our core divisor from many circuit nodes. As an example of how this generalization works, imagine the given divisor in the above example, $D = ab + c + f + bd + e$, does not exist in our circuit and instead, two nodes $D_1 = ab + c + f$ and $D_2 = bd + e$ exist, as shown in Fig. 3(c). When function $F = abd + \bar{a}c + cd + ae$ is given and we want to search for a good divisor between D_1 and D_2 with extended division, we can temporarily pretend that all the five cubes are from the same node and, therefore, the flow is identical to the example shown in this section. Each wire in the cubes of D_1 and D_2 votes for a candidate core divisor and we have an identical vote table as shown in Table I(b). The only slight modification we need is in the final maximal clique formulation, where we need to model the fact that some cubes in the second column of Table I originally come from a different node. Since the situation is very similar to the situation when we only have one node, we do not go into details here. Note that, as is also the case for basic division, we can perform extended division in terms of sum-of-product form as well as product-of-sum form. Instead of focusing on the cubes that have implication value zero, we would then focus on the sum terms that have implication value one. The rest of the algorithm applies similarly.

V. EXPERIMENTAL RESULTS

We have implemented our algorithm and applied it to the substitution problem. Our implementation has three configurations:

1) basic division;
2) extended division without global internal don't cares;

TABLE V
EXPERIMENTAL RESULTS (script.algebraic)

circuit	sis		basic		ext.		ext.+GDC	
	lit.	cpu	lit.	cpu	lit.	cpu	lit.	cpu
9symml	267	1.53	268	22.40	256	28.52	256	98.44
C1355	670	3.54	526	32.32	526	31.77	595	80.44
C1908	564	4.95	563	37.03	553	34.67	557	109.72
C2670	840	7.31	767	52.18	753	52.93	755	126.00
C3540	1486	16.25	1460	201.39	1465	218.18	1514	4168.66
C432	252	1.73	203	10.41	220	12.36	214	18.71
C499	558	2.20	550	28.27	550	26.46	550	62.55
C5315	2008	22.21	1861	138.43	1869	141.53	1851	388.77
C6288	3787	35.22	3317	223.10	3316	227.13	3317	320.58
C7552	2584	38.25	2356	229.58	2223	227.47	2335	1400.14
C880	473	3.19	450	21.84	425	22.74	414	32.53
alu2	478	5.46	406	54.38	426	68.53	419	393.79
alu4	917	20.60	838	157.18	789	186.81	824	1835.95
apex6	854	5.23	777	55.44	806	56.69	772	275.78
apex7	286	1.65	240	12.17	237	11.66	239	27.56
dalu	1500	31.24	1417	307.81	1201	303.83	1250	8493.04
des	3816	119.12	3720	1062.60	3595	1006.89	3738	34068.09
example2	375	2.30	346	22.56	329	24.26	340	93.06
frg2	1118	17.53	1011	160.26	854	155.59	1041	2386.93
i8	1143	19.93	1052	298.18	1046	307.37	1039	8277.33
i9	623	6.39	606	140.93	600	122.37	604	2714.39
i10	2658	44.96	2427	412.45	2392	446.46	2375	5324.80
rot	803	6.44	680	62.03	690	34.82	694	62.03
t481	1028	66.34	961	142.42	917	104.18	685	1229.88
term1	271	3.05	264	21.17	223	22.75	162	58.47
too_large	491	316.66	437	174.02	429	173.97	413	2178.73
ttt2	242	1.62	217	13.57	170	12.75	206	36.26
x1	357	2.97	337	21.46	333	23.81	326	84.48
x3	890	8.75	775	75.82	765	73.05	771	487.81
x4	424	4.22	398	32.05	375	29.14	402	169.09
s1196	677	7.60	599	69.29	589	81.90	573	636.54
s1238	714	8.18	587	73.48	576	91.11	602	548.91
s13207	2518	50.66	2244	313.61	2144	302.72	2258	10108.98
s1423	723	6.04	618	33.67	624	38.07	632	100.85
s1488	751	9.58	731	146.66	670	212.78	707	2123.05
s1494	762	10.45	697	149.65	686	209.46	704	1941.85
s15850	4113	396.60	3764	933.7	3702	1249.65	3661	17892.23
Total	42021	1310	38470	5943	37324	6375	37795	108356
%			8.5%		11.2%		10.1%	

3) extended division with global internal don't cares.

To clarify what we mean by global internal don't cares, we refer to Fig. 3(b) as an example. As explained earlier, after adding a redundancy most of the internal don't cares would occur within the Q_{init} region, the D_c region, and the bold AND gate. With the second configuration we limit our implications search within these interested regions, while with the third configuration we allow the implication search to go outside these regions.

We performed experiments on MCNC and ISCAS benchmarks within SIS [15] environment. For each benchmark, we first run the following script to obtain the initial circuit:

Script A: eliminate 0; simplify.

The purpose of "eliminate zero" is to create complex gates by collapsing gates with single fanout since complex gates are more suitable for substitution. After running the above script, we then compare our algorithm with the algebraic resubstitution "resub -d" in SIS. Table II shows the comparison between SIS and our result.

The first column shows the name of the circuit. The second column shows the initial literal count after running Script A above. The columns labeled "sis" is the result of running the "resub -d" command in SIS, with subcolumns "lit." and "cpu" reporting the number of literals and CPU time, respectively. The column labeled "basic" is the result of our basic division algorithm. The column labeled "ext." shows the result of our extended division without global don't cares (GDC's); while column "ext.+GDC" shows the result with GDC's taken into consideration. All literal counts are in factor form. Take the circuit *C2670* as an example, after running the above script, initially the circuit had 928 literals, shown in the second column. After running "resub -d" the circuit reduced to 843 literals. With our basic division the literal count was reduced to 831. The extended division reduced it to 829, while the extended division with GDC's taken into account brought it down to 817. The last two rows show the summation of each column and the percentage of improvement compared to the initial literal count. As the table indicates, all three configurations of our division algorithms outperformed the traditional division and substitution. In general, the "resub -d" command, the basic division, and the extended division without GDC's spent similar CPU times, while the extended

division with GDC's, in spite of the best result, spent much more time. The much larger CPU time was spent in performing implications throughout the whole circuit, as opposed to restricting the implication only within the interested area discussed earlier.

To further explore the scenario of different initial circuits, we note that the commands "gcx" and "gkx" are also typically good steps before applying the "resub" command.[2] We therefore repeated the experiment with the following two scripts:

Script B: eliminate 0; simplify; gcx;
Script C: eliminate 0; simplify; gkx.

Tables III and IV, respectively, show the results, which are consistent with the observations drawn from Table II.

The above three scripts were created so that we may directly compare the traditional resubstitution with our algorithms in one single run. To test our algorithm in a complete flow, we have also performed an experiment which replaces all the occurrences of the "resub" command in *script.algebraic*[3] [15] by our algorithm, keeping the rest of the script intact. Table V shows the result. Again we see that our division algorithms outperformed the traditional division and substitution. One anomaly we see in Table V is that "ext.+GDC" on average underperformed "ext." We believe this is due to the locally greedy nature of our implementation. In other words, since our implementation takes the first division that has a positive gain on literal count, which can be marginal, we may have neglected the other potential better divisors.

In summary, our division and substitution algorithm consistently had about 10% improvement over SIS "resub" on a variety of script setups and/or starting points. As we discussed earlier, the better result is due to our consideration of Boolean substitutions instead of algebraic substitutions. Moreover, we consider additional POS structure and GDC's during optimization. Among the three configurations we setup for the experiments, extended divisions ("ext.") seems to have the best balance between the run time and the quality of result.

VI. Conclusion

In this paper, we first presented an efficient algorithm, based on the philosophy of RAR, for performing Boolean division. With the concept of SOS and POS, we tailored the RAR philosophy to the Boolean division problem. The tailoring enables us to add multiple wires/gates in a specialized configuration and remove more wires/gates. Applying our Boolean division algorithm, our algorithm can perform substitution not only in the traditional sum-of-product form, but also in product-of-sum form. We then generalize our basic division to what we call extended division. Extended division allows us to decompose not only on the dividend but also on the divisor. Furthermore, our technique is able to naturally incorporate all types of internal don't cares into consideration. We also presented some experimental results to verify the effectiveness of our algorithm.

[2] In almost all the scripts shipped with the SIS package, "resub" is run after either "simply", "gcx", or "gkx."

[3] We choose *script.algebraic* because it is one of the scripts that contain the most "resub"s among the many scripts included in the distribution of SIS.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ: IEEE Press, 1994.
[2] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. IEEE ISCAS-82*, 1982, pp. 49–54.
[3] R. K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer Academic, 1984.
[4] S. C. Chang and M. Marek-Sadowska, "Perturb and simplify: Multi-level Boolean network optimizer," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 2–6.
[5] S. C. Chang, L. VanGinneken, and M. Marek-Sadowska, "Fast Boolean optimization by rewiring," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 262–269.
[6] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.
[7] L. A. Entrena and K. T. Cheng, "Combination and sequential logic optimization by redundancy addition and removal," *IEEE Trans. Computer-Aided Design,* vol. 14, pp. 909–916, July 1995.
[8] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Francisco, CA: Freeman, 1979.
[9] W. J. Hsu and W. Z. Shen, "Coalgebraic division for multilevel logic synthesis," in *Proc. ACM/IEEE Design Automation Conf.*, June 1992, pp. 438–442.
[10] T. Kirkand and M. R. Mercer, "A Topological search algorithm for ATPG," in *Proc. ACM/IEEE Design Automation Conf.*, June 1987, pp. 502–508.
[11] W. Kunz and D. K. Pradhan, "Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits," in *Proc. Int. Test Conf.*, 1992, pp. 816–825.
[12] ——, "Multi-level logic optimization by implication analysis," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 6–13.
[13] M. Schulz and E. Auth, "Advanced automatic test pattern generation and redundancy identification techniques," in *Proc. Fault Tolerant Computing Symp.*, June 1988, pp. 30–34.
[14] T. Stanion and C. Sechen, "Boolean division and factorization using binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1179–1184, Sept. 1994.
[15] 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," Univ. California, Berkeley, memo UCB/ERL M92/41.

Shih-Chieh Chang received the B.S. degree in electrical engineering from National Taiwan University, Taiwan, in 1987 and the Ph.D. degree in electrical engineering from the University of California, Santa Barbara, in 1994.

He worked at Synopsys, Inc., Mountain View, CA, from 1995 to 1996. He then joined the faculty at the Institute of Computer Science and Information Engineering, National Chung Cheng University, Taiwan. His current research interests include VLSI logic synthesis, layout, and field-programmable gate array (FPGA) related applications.

Dr. Chang received a Best Paper Award at the 1994 Design Automation Conference.

David Ihsin Cheng received the B.S. degree in computer engineering from National Chiao-Tung University, Taiwan, in 1986 and the Ph.D. degree in electrical engineering from the University of California, Santa Barbara, in 1995.

He worked at Mentor Graphics Corporation, San Jose, CA, and Exemplar Logic Corporation, San Jose, CA, from 1995 to 1998. He is currently Manager of System Integration at Ultima Interconnect Technology, Sunnyvale, CA, working on physical aspects of VLSI design automation. During the summers of his student years he had worked with Daimler-Benz Research Institute in Ulm, Germany, AT&T Bell Labs in Murray Hill, NJ, and IBM Watson Research Center in Yorktown Heights, NY. His research interests include logic synthesis, physical design, formal verification, and error diagnosis.