PolyCleaner: Clean your Polynomials before Backward Rewriting to Verify Million-gate Multipliers

Alireza Mahzoon[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{mahzoon,grosse,drechsle}@informatik.uni-bremen.de

ABSTRACT

Nowadays, a variety of multipliers are used in different computationally intensive industrial applications. Most of these multipliers are highly parallelized and structurally complex. Therefore, the existing formal verification techniques fail to verify them.

In recent years, formal multiplier verification based on Symbolic Computer Algebra (SCA) has shown superior results in comparison to all other existing proof techniques. However, for non-trivial architectures still a monomial explosion can be observed. A common understanding is that this is caused by redundant monomials also known as vanishing monomials. While several approaches have been proposed to overcome the explosion, the problem itself is still not fully understood.

In this paper we present a new theory for the origin of vanishing monomials and how they can be handled to prevent the explosion during backward rewriting. We implement our new approach as the SCA-verifier PolyCleaner. The experimental results show the efficiency of our proposed method in verification of non-trivial million-gate multipliers.

1 INTRODUCTION

Arithmetic circuits are an inseparable part of many designs. They are getting even more attention due to the high demand for computationally intensive applications such as signal processing or cryptography. Integer multipliers are one of the most dominant components in arithmetic circuits. Designers have proposed a variety of multiplier architectures to satisfy the community demands for high speed, low power, and low area designs. These multipliers are usually highly parallelized and structurally complex which makes their verification a challenge for formal methods.

The evaluation of formal methods efficiency in verification of gate-level multipliers shows: (a) *Decision Diagrams* (DDs) (such as BDDs and *BMDs) suffer from memory blow-up when the multiplier is large, (b) *Boolean Satisfiability* (SAT) and *Satisfiability Modulo Theories* (SMT) experience exponential run-times as the bit-width of multiplier increases, (c) reverse engineering approaches [13, 8] using *Arithmetic Bit-Level* (ABL) representations can handle structurally simple multipliers, however they fail to verify non-trivial designs, and (d) term rewriting techniques [5, 14] rely on a database of rewrite rules to support a wide range of architectures, however for non-existing implementations a manual update of the database is required which makes it not fully automated. In contrast, *Symbolic Computer Algebra* (SCA) methods have shown very good results in comparison to the just mentioned methods (see e.g. [2, 11, 12, 9, 7]). In SCA-based verification, the specification is represented as a single polynomial SP describing the function of the circuit based on its inputs and outputs. The gate-level circuit is also captured as a set of suitably derived polynomials P_G employing the theory of Gröbner basis. Finally, the membership of SP in the ideal generated by P_G is tested. In other words, SP is step-wise divided by P_G to determine the remainder. The whole division process is called backward rewriting since the process is performed from the outputs to the inputs. Having the final division result, a zero remainder proves the correctness of the gate-level circuit.

In recent years, SCA has been successfully employed to verify finite field multipliers [6], as well as large but structurally simple integer multipliers [2, 4, 16, 9]. However, verification of non-trivial multipliers is still a big challenge for SCA methods as an explosion happens in the number of monomials during backward rewriting. A common understanding is that this explosion is caused by redundant monomials known as *vanishing monomials*. These monomials are generated during verification of a non-trivial arithmetic circuit, and reduce to zero after several steps. However, the huge number of vanishing monomials before cancellation causes a blow-up in computations. While there have been several recent attempts to attack this blow-up (for details see the related work section), the problem is still not fully understood and a global solution is missing.

In this paper, we propose **PolyCleaner, an approach to clean polynomials from vanishing monomials before global backward rewriting**. A driving force behind our approach comes from results of several conducted experiments: We raised the architectural complexity of multipliers step-by-step to clearly show the effect of vanishing monomials and hence the limits of the state-of-the-art SCA-verifiers. After analysis of the intermediate results of the divisions during backward rewriting, we come up with a new theory for the origin of vanishing monomials. In essence, the origin of vanishing monomials are gates where both output paths from a *Half Adder* (HA) converge. At these gates a monomial is formed when performing backward rewriting, which creates many new (vanishing) monomials in each following division step. They all last long and even worse make the current polynomial larger and larger with each new division step until the HA is reached. When substituting both gates of the HA, all these vanishing monomials reduce to zero. Our proposed SCA-verifier PolyCleaner finds all such converging gates and locally removes the vanishing monomials. Thus, the global backward rewriting process becomes vanishing-free and no explosion happens during backward rewriting. The experimental results confirm that PolyCleaner can verify non-trivial million-gate multipliers which was not possible before.[1]

Summarizing, the major contributions of this paper are:

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240837

[1]We make our tool PolyCleaner and all benchmarks available at https://github.com/amahzoon/PolyCleaner

- Presentation of the limits of SCA-based backward rewriting in verification of non-trivial multipliers
- Introduction of a new theory for vanishing monomials
- Presentation of the SCA-verifier PolyCleaner for local cancellation of vanishing monomials and global vanishing-free backward rewriting
- Verification of a wide range of different non-trivial multiplier architectures with up to 1.6 million gates

2 RELATED WORK

For verification of integer multipliers several SCA-based methods have been introduced. The authors of [2] and [16] proposed a method to capture the gate-level netlist as a set of polynomials, then substituting these polynomials step-by-step following the reverse topological order of the circuit into the specification polynomial. The work of [4] identifies fanout-free cones on the netlist and extracts polynomials for each of theses cones. By this, more monomials can be canceled in comparison to the initial introduced backward rewriting. The technique presented in [17, 10] finds half adders and full adders in an And-Inverter Graph (AIG) representation of the multiplier. However, the technique can only verify arithmetic circuits where the detection of adder-trees is possible. In summary, all mentioned approaches have reported very good results for simple multiplier architectures, but fail to verify non-trivial multipliers. The reason is that they have no solution for the growing number of vanishing monomials during backward rewriting.

Some works aim to attack the vanishing monomial explosion during backward rewriting. The method of [11] groups gates into cones based on XOR gates. Then, common and XOR rewriting for the cone polynomials are performed which removes some of the vanishing monomials near to HAs. The method works for some non-trivial architectures, but is not robust since it misses many vanishing monomials.

[9] introduced a column-wise method for multipliers with visible adder-trees. The method cuts the circuit into slices based on the column-structure and then verifies these slices incrementally. The method removes the product of HA's outputs whenever they appear during backward rewriting. While the paper contains clear theory for using Gröbner basis, the method is not able to verify any non-trivial multiplier as neither the origin of vanishing monomials is identified, nor are the effects of them handled.

3 PRELIMINARIES

In this section, the basic concepts of SCA and the verification process of an arithmetic circuit using SCA are explained.

Definition 1. *A* Monomial *is the power product of variables in the following form:*

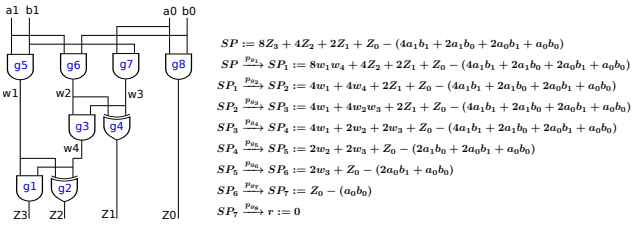
$$t = x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \quad with \quad \alpha_i \in \mathbb{N}_0 \tag{1}$$

A monomial with a coefficient is called a Term.

Definition 2. *A* Polynomial *is a finite sum of monomials with coefficients in field k:*

$$f = \sum_j c_j t_j \quad with \quad c_j \in k \tag{2}$$

A polynomial has a monomial order which facilitates the polynomial manipulations. This order is specified based on the ordering of variables and their powers. We use $A > B$ to show that A is in a higher order than B. For example, in $f = y^4z + y^2z^2 + xy$, if we assume that the ordering of the variables is $x > y > z$, then the monomial order will be $xy > y^4z > y^2z^2$. The first monomial



(a) 2-bit multiplier **(b) Backward rewriting steps**

Figure 1: 2-bit multiplier and backward rewriting steps

and the first term after ordering are called *leading monomial* and *leading term* and are denoted by $LM(f)$ and $LT(f)$, respectively.

In SCA, division is denoted by $p \xrightarrow{F} r$, where F is a set of polynomials and r is the remainder. For example, if $p = xy$, $f_1 = x - z$, and $f_2 = yz$, then $xy \xrightarrow{f_1} yz \xrightarrow{f_2} 0$. To perform the division of xy by f_1, first f_1 is multiplied by y to produce the same leading monomial xy as p, so $f_1y = xy - yz$. Subsequently, the subtraction is performed, i.e. $p - (f_1y) = xy - (xy - yz) = yz$, which is the result of the first division. Finally, yz is divided by f_2 to get remainder 0.

In SCA-based verification of arithmetic circuits, the gate-level netlist and the specification polynomial are given as inputs, and the task is to formally prove that the specification polynomial and the arithmetic circuit are equivalent. The *specification polynomial* is a polynomial determining the function of an arithmetic circuit based on its inputs and outputs. For example, the specification polynomial for the 2-bit multiplier of Fig. 1a is $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2a_1 + a_0)(2b_1 + b_0)$ where $8Z_3 + 4Z_2 + 2Z_1 + Z_0$ describes the 4-bit output, and $(2a_1 + a_0)(2b_1 + b_0)$ indicates the multiplication of the 2-bit inputs.

The gates of an arithmetic circuit can be modeled as polynomials determining the relation between output and inputs. The polynomials of basic Boolean gates are as follows:

$$\begin{aligned} z = \neg a \Rightarrow p_g &:= z - 1 + a, & z = a \vee b \Rightarrow p_g &:= z - a - b + ab, \\ z = a \wedge b \Rightarrow p_g &:= z - ab, & z = a \oplus b \Rightarrow p_g &:= z - a - b + 2ab \end{aligned} \tag{3}$$

The polynomials in (3) are in the form of $p_g = x - tail(p_g)$ where x is the gate's output, and $tail(p_g)$ is a function based on the gate's inputs.

The gate polynomials for the 2-bit multiplier of Fig. 1a are:

$$\begin{aligned} p_{g_1} &:= Z_3 - w_1w_4 & p_{g_5} &:= w_1 - a_1b_1 \\ p_{g_2} &:= Z_2 - w_1 - w_4 + 2w_1w_4 & p_{g_6} &:= w_2 - a_1b_0 \\ p_{g_3} &:= w_4 - w_2w_3 & p_{g_7} &:= w_3 - a_0b_1 \\ p_{g_4} &:= Z_1 - w_2 - w_3 + 2w_2w_3 & p_{g_8} &:= Z_0 - a_0b_0 \end{aligned} \tag{4}$$

Assume that the signals of an arithmetic circuit are ordered based on the reverse-topological order (i.e. from outputs toward inputs). The specification polynomial SP and the gate-level netlist are equivalent, iff the remainder of dividing SP by gate polynomials becomes zero. This division is known as Gröbner basis reduction. For the theory of Gröbner basis and its application to verification of arithmetic circuits we refer to [3, 9].

The steps of dividing SP by $p_{g_1}, \dots, p_{g_8}$ for the 2-bit multiplier of Fig. 1a are shown in Fig. 1b. The final remainder of the division is equal to zero, hence the multiplier is bug-free. Please note that all variables in the polynomials are Boolean. Thus, x^n can be replaced by x. Furthermore, for integer arithmetic circuits, dividing SP_i by a gate polynomial $p_{g_i} = x_i - tail(p_{g_i})$ is equivalent to *substituting* x_i by $tail(p_{g_i})$ in SP_i. For example, to obtain the result of

the first division step in Fig. 1b, Z_3 can be substituted by $w_1 w_4$ in SP. The process of dividing the specification polynomial by gate polynomials (or equivalently substituting gate polynomials in the specification polynomial) is called *backward rewriting*.

4 LIMITS OF SCA-BASED BACKWARD REWRITING

In this section we provide experimental evidence for the limits of SCA-based backward rewriting for multiplier circuits.

To achieve this goal, we first briefly review the general architecture of an integer multiplier. A multiplier consists of three stages: *Partial Product Generator* (PPG), *Partial Product Accumulator* (PPA), and *Final Stage Adder* (FSA). The PPG stage generates partial products from the multiplicand input and multiplier input. Then, the PPA stage performs multi-operand addition for all the generated partial products and computes their sum. Finally, this sum is "converted" to the corresponding binary output at the FSA.

For industrial multiplier designs, trading of area and performance leads to several options for the implementation of each stage. For example, in the PPG stage partial products can be computed in a straight forward way, or Booth encoding can be used to reduce the overall stages of the multiplier. Then in the PPA stage different alternatives to accumulate the partial products are available, ranging for example from simple adders formed in an array to different tree-like structures. Again, in the final FSA stage different choices to use parallelism can be made ranging from ripple carry adders to carry look-ahead adders. In the following we will use the **notation $[\alpha \circ \beta \circ \gamma]$** to refer to a multiplier consisting of the stages: PPG α, PPA β and FSA γ.

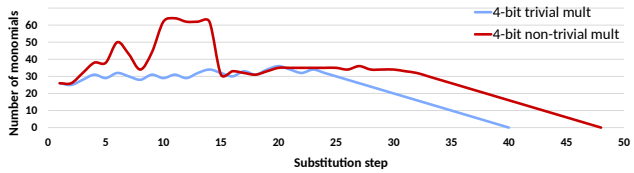
Coming back to SCA-based verification, we now run the following experiment: Perform backward rewriting as introduced in the previous section for

(i) a trivial multiplier architecture, i.e. *[simple partial product generator ∘ array ∘ ripple carry adder]*
(ii) a non-trivial architecture, i.e. *[simple partial product generator ∘ wallace tree ∘ carry look-ahead adder]*

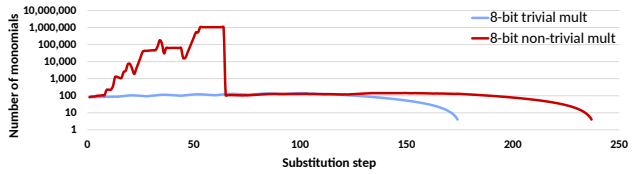
Results for different multiplier sizes (number of input bits per operand) are shown in Fig. 2a, Fig. 2b, and Fig. 2c, respectively. In the figures we plot the number of monomials in the consecutive substitution steps of backward rewriting. For the trivial multipliers (blue lines), the number of monomials remains almost constant during backward rewriting, and at a certain substitution step the number of monomials starts to decrease until it finally becomes one.[2] In contrast, during verification of the non-trivial multipliers (red lines) we can see after a few number of substitutions for each of them a sudden increase in the number of monomials. For example, the number of monomials during verification of the 4-bit (8-bit) non-trivial multiplier grows to 2.5x (10,000x) compared to the initial number of monomials. However, the situation is even worse for the 16-bit non-trivial multiplier. As can be seen in Fig. 2c, the number of monomials explode after about 50 substitution steps. In general, this exponential growth makes the verification of non-trivial multipliers with input bit-width larger than 8-bit practically impossible.

In the last three years, some papers have been proposed to overcome this monomial explosion problem. As a common understanding so-called *vanishing monomials* (redundant monomials which finally reduce to zero after several substitution steps; illustrating example see next section) have been identified as the root cause of the explosion [11, 9]. As already discussed in the related work section, the previous approaches either consider large but trivial architectures where no vanishing monomials appear, or carry out rewriting of the polynomials before performing backward rewriting but do not provide true inside into the vanishing monomial problem (which also becomes evident by their non-robustness).

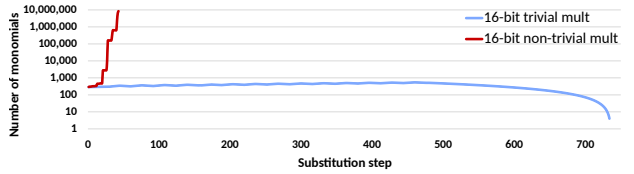
[2] All multipliers considered here are correct, hence the final result is the zero polynomial which means we have one monomial which is 0.



(a) 4-bit multipliers



(b) 8-bit multipliers



(c) 16-bit multipliers

Figure 2: Number of monomials at each step of substitution

In this paper we present a **new theory for the origin of vanishing monomials** and how they can be handled to prevent the explosion during backward rewriting. The core idea is to identify gates in the multiplier netlist where both output paths from a HA converge. At these gates a monomial is formed which creates many new monomials in each following substitution step. However, most of these new monomials can be canceled but this only when reaching the HA during backward rewriting after again many substitution steps (the vanishing monomial situation).

5 POLYCLEANER

In this section we first give an illustrative example for vanishing monomials in SCA-based backward rewriting of a non-trivial multiplier. Then, we make the general case of vanishing monomials, i.e. we come up with the basic theory for the origin of vanishing monomials. Next, we present the overview of our SCA-verifier PolyCleaner which overcomes polynomial explosion during backward rewriting. Finally, we discuss each phase of PolyCleaner in detail.

5.1 Vanishing Monomials Example

As a circuit example we consider a 3-bit non-trivial multiplier of type *[simple partial product generator ∘ wallace tree ∘ carry look-ahead adder]*. A subset of it's gate-level netlist is shown in

$$SP := 32Z_5 + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])$$

$$SP \xrightarrow{g_1 \Rightarrow Z_5 = w_{21} + w_{22} - w_{21}w_{22}} SP_1 := 32(w_{21} + w_{22} - w_{21}w_{22}) + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0]) = 32w_{21} + 32w_{22} - 32w_{21}w_{22} + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])$$

$$\ldots$$

$$SP_{13} \xrightarrow{g_{14} \Rightarrow Z_2 = w_9 + w_{13} - 2w_9w_{13}} SP_{14} := 4w_9 + 4w_{13} + 32w_{10} + 16w_{11} + 8w_3 + 8w_4 + 8w_{12} - 8w_4w_{12} + 2Z_1 + Z_0$$

$$\underbrace{+32\mathbf{w_4w_9w_{10}w_{11}w_{12}w_{13}} - 32\mathbf{w_3w_9w_{10}w_{11}w_{13}} - 32\mathbf{w_4w_9w_{10}w_{11}w_{13}} - 32\mathbf{w_9w_{10}w_{11}w_{12}w_{13}} - 32\mathbf{w_3w_{10}w_{11}w_{12}}}_{7\text{ monomials}}$$

$$\underline{-32\mathbf{w_3w_4w_{10}w_{11}} + 32\mathbf{w_3w_4w_{10}w_{11}w_{12}}} - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])$$

$$SP_{14} \xrightarrow{g_{15} \Rightarrow w_{10} = w_1w_2} SP_{15} := 32w_1w_2 + 16w_{11} + 8w_3 + 8w_4 + 8w_{12} + 4w_9 + 4w_{13} - 8w_4w_{12} + 2Z_1 + Z_0$$

$$\underbrace{+32\mathbf{w_1w_2w_4w_9w_{11}w_{12}w_{13}} - 32\mathbf{w_1w_2w_3w_9w_{11}w_{13}} - 32\mathbf{w_1w_2w_4w_9w_{11}w_{13}} - 32\mathbf{w_1w_2w_9w_{11}w_{12}w_{13}} - 32\mathbf{w_1w_2w_3w_{11}w_{12}}}_{7\text{ monomials}}$$

$$\underline{-32\mathbf{w_1w_2w_3w_4w_{11}} + 32\mathbf{w_1w_2w_3w_4w_{11}w_{12}}} - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])$$

$$SP_{15} \xrightarrow{g_{16} \Rightarrow w_{11} = w_1 + w_2 - 2w_1w_2} SP_{16} := 32w_1w_2 + 16w_1 + 16w_2 - 32w_1w_2 + 8w_{12} + 8w_4 + 8w_3 + 4w_{13} + 4w_9 - 8w_4w_{12} + 2Z_1 + Z_0 \underbrace{+32\mathbf{w_1w_2w_4w_9w_{12}w_{13}} + 32\mathbf{w_1w_2w_4w_9w_{12}w_{13}} - 64\mathbf{w_1w_2w_4w_9w_{12}w_{13}} + \ldots + 32\mathbf{w_1w_2w_3w_4w_{12}} + 32\mathbf{w_1w_2w_3w_4w_{12}}}_{21\text{ monomials}}$$

$$\underline{-64\mathbf{w_1w_2w_3w_4w_{12}}} - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])$$

$$= \mathbf{16w_1 + 16w_2 + 8w_{12} + 8w_4 + 8w_3 + 4w_{13} + 4w_9 - 8w_4w_{12} + 2Z_1 + Z_0 - (4A[2] + 2A[1] + A[0]) \times (4B[2] + 2B[1] + B[0])}$$

$$\ldots$$

Figure 3: Backward rewriting of 3-bit non-trivial multiplier

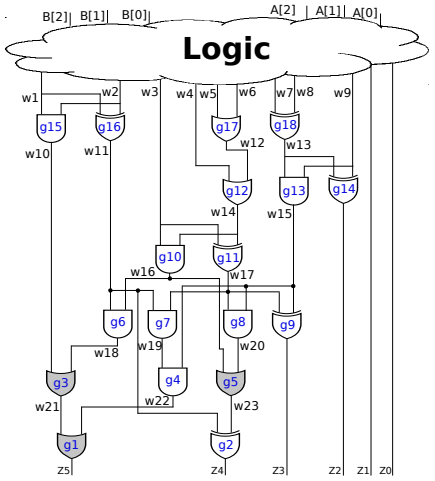


Figure 4: 3-bit non-trivial multiplier

Fig. 4. As can be seen in the figure, the inputs of the multiplier are $B = B[2]B[1]B[0]$ and $A = A[2]A[1]A[0]$, while the output is $Z = Z_5Z_4Z_3Z_2Z_1Z_0$. To increase the readability in the following, we are using different notations in referring to the inputs and output bits, respectively.

Also an excerpt of the substitution steps when performing backward rewriting for this multiplier is depicted in Fig. 3:

(a) SP is the specification polynomial for the 3-bit multiplier at hand. Performing backward rewriting in reverse topological order, i.e. dividing/substituting SP by the gate polynomials of the multiplier implementation of Fig. 4 will finally produce the remainder zero, since the considered gate-level implementation is correct.
(b) In the first step of backward rewriting, Z_5, which is the output of the OR gate g_1, is substituted by $w_{21} + w_{22} - w_{21}w_{22}$ (cf. OR gate polynomial in (3)). The result after substitution is shown as the new polynomial SP_1. Since the coefficient of Z_5 is 32, we have to perform this multiplication.
(c) The next 12 steps of backward rewriting are omitted due to page limitation.
(d) In Step 14, substitution of Z_2 (output of g_{14}) in SP_{13} is executed and as result the new polynomial SP_{14} is generated.
(e) The next two steps result in substitution of w_{10} (output of g_{15}) and w_{11} (output of g_{16}), respectively. The final result of both steps is SP_{16}, cf. bold line.

As can be seen, we have marked several monomials in red. The reason is that they finally reduce to zero, i.e. after the division of g_{15} and g_{16} they are canceled out completely. Hence, we call them vanishing monomials. Before we explain why these red monomials together form vanishing monomials, we provide some numbers: We find 7 red monomials (34 variables) in Step 14, 7 red monomials (41 variables) in Step 15, and 21 red monomials (102 variables) as intermediate result in Step 16. These numbers clearly illustrate an explosion in backward rewriting of a non-trivial multiplier. Please note when performing backward rewriting for the complete netlist much more vanishing monomials appear.

Now two major questions arise:

(1) Why do the red monomials finally reduce to zero in SP_{16}?
(2) What is the origin of the red monomials?

For answering (1), just take a look on all 7 red monomials in SP_{14}. They all contain the product $w_{10}w_{11}$. In the next two substitution steps this product is substituted by $(w_1w_2)(w_1 + w_2 - 2w_1w_2)$ based on the respective gate polynomials g_{15} and g_{16}. Please note this product equals zero as can be seen here:

$$w_1w_2(w_1 + w_2 - 2w_1w_2) = w_1w_2 + w_1w_2 - 2w_1w_2 = 0 \tag{5}$$

In addition, this is in line with the following observation: w_{10} and w_{11} are the outputs of a HA (see gates g_{15} and g_{16} in Fig. 4). Now computing the AND of these two HA outputs, we also get result 0:

$$w_{10} \cdot w_{11} = (w_1 \cdot w_2) \cdot (w_1 \oplus w_2) = 0 \tag{6}$$

In summary, this is the reason why the red monomials finally vanish in SP_{16}.

Now, we give an answer for (2): As just discussed in SP_{14} each red monomial contains the product $w_{10}w_{11}$. Traversing back all substitution steps (i.e. moving in the direction of the outputs on the netlist) this product originates from the multiplication of $w_{21}w_{22}$, formed via the substitution for gate g_1 as can be seen in the result SP_1. Interpreting this observation on the netlist means that there are two paths[3] starting from the two HA outputs – here w_{10} and w_{11} – and these paths finally converge at a gate (here the OR gate g_1 with output Z_5).

Overall, we conclude from this illustrating example that the origin of vanishing monomials is a gate where HA outputs converge, while the cancellation happens much later only after substitution of both HA gate polynomials.

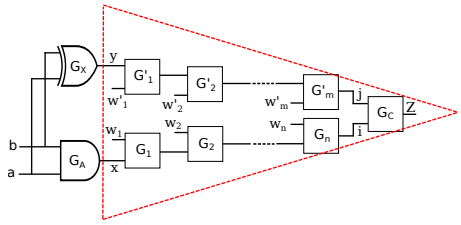
In the next section, we provide the underlying theory of vanishing monomials. We will show that these vanishing monomials can be handled efficiently such that raising the complexity of the multiplier architecture does not increase the maximal size of the current polynomial SP_i.

5.2 Basic Theory of Vanishing Monomials

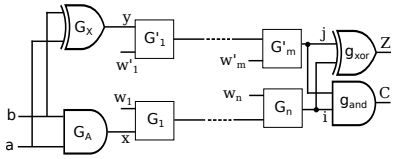
We now generalize the observation from the illustrating example of the previous section. Therefore, we formulate the following theorem:

THEOREM 1. *Assume that x and y are the outputs of a HA. The product of x and y appears during backward rewriting of an arithmetic circuit, if at least one path from x and one path from y converge to a gate G_C and G_C is not part of another HA.*

[3] Path 1: g_{15}, g_3, g_1; Path 2: g_{16}, g_7, g_4, g_1



(a) General case



(b) Converging to HA

Figure 5: Converging paths

Proof. Fig. 5a shows two paths starting from the HA outputs x and y converging to G_C with the output Z. The first path starting from x is a chain of gates which contains $G_1, G_2, \ldots, G_n, G_C$. The second path starting from y consists of $G'_1, G'_2, \ldots, G'_m, G_C$. Based on (3), we know that the polynomial of all 2-input gates contains the product of its inputs. Therefore, the polynomial of G_C can be written as:

$$G_C \Rightarrow Z = f(i, j) + cij \tag{7}$$

where $f(i, j)$ is equal to $i + j$ (for OR and XOR) or zero (for AND), and c is a coefficient. The functions for i and j can be extracted by substituting the polynomials of the gates located on the paths. Since one path depends on x and the other on y we get after the substitutions:

$$\begin{aligned} i &= f'(w_1, w_2, \ldots, w_n, x), \\ j &= f''(w'_1, w'_2, \ldots, w'_m, y) \end{aligned} \tag{8}$$

Based on (7) and (8), we conclude:

$$\begin{aligned} G_C \Rightarrow Z &= f(i, j) + cf'(w_1, w_2, \ldots, w_n, x)f''(w'_1, w'_2, \ldots, w'_m, y) \\ &= f(i, j) + \underbrace{cxyT'_1 + cxyT'_2 + \cdots + cxyT'_r + cT_1 + cT_2 + \cdots + cT_s}_{\text{comes from the product } ij} \end{aligned} \tag{9}$$

where $cxyT'_h$ denotes the monomials containing the product of x and y. Note that this xy product is generated because we multiply two polynomials one depending on x and the other on depending on y. On the other hand, assume the current polynomial SP_i before substituting Z by the G_C polynomial is:

$$SP_i := ZX'_1 + ZX'_2 + \cdots + ZX'_l + X_1 + X_2 + \cdots + X_q \tag{10}$$

where ZX'_i denotes the monomials which contain Z. Now, we distinguish between the two cases:

(1) *G_C is part of another HA (see Fig. 5b):* Assume G_C is the XOR (also holds for the AND), and the polynomials for the HA's gates are $g_{xor} \Rightarrow Z = i + j - 2ij$ and $g_{and} \Rightarrow C = ij$, respectively. Because of the functionality of the HA where G_C is part of, in SP_i one of the X_k monomials of (10) is $2C$, so after substituting g_{xor} and g_{and} polynomials in SP_i the result is:

$$\begin{aligned} SP_i &= Z + 2C + X_1 + \cdots + X_q \\ SP_i &\xrightarrow{g_{xor}} SP_{i+1} := i + j - 2ij + 2C + X_1 + \cdots + X_q \\ SP_{i+1} &\xrightarrow{g_{and}} SP_{i+2} := i + j - \cancel{2ij} + \cancel{2ij} + X_1 + \cdots + X_q \\ &= i + j + X_1 + \cdots + X_q \end{aligned} \tag{11}$$

As can be seen, since the product ij is not contained anymore in SP_{i+2}, we can conclude that the product xy is not generated in the following steps of backward rewriting.

(2) *G_C is not part of another HA:* Then, the product of ij remains as part of SP_i in (10) when substituting (9) and hence xy appears in the upcoming steps of backward rewriting.

□

Based on this theorem we make the following definitions.

Definition 3. *Let G_C be a gate fulfilling Theorem 1. Then G_C is called a* converging gate.

Definition 4. *Let G_C be a converging gate. Then, the monomials containing the product of the HA's outputs originating from G_C are* vanishing monomials *as they are reduced to zero after HA's gates substitution.*

For managing the size of the current polynomial SP_i during backward rewriting, it is essential to prevent the inclusion of vanishing monomials because for non-trivial architectures explosion occurs. Hence, the goal is to determine a vanishing-free polynomial representation for each converging gate. In other words, we are looking for the cones starting from a converging gate and ending in the related HA outputs. Such a cone is called *Converging Gate Cone (CGC)* in the rest of the paper (see also red area in Fig. 5a).

In the next section, we present our SCA-verifier PolyCleaner which finds all CGCs and locally removes the vanishing monomials. Thus, the global backward rewriting becomes vanishing-free and large non-trivial multipliers can be verified.

5.3 Overview of PolyCleaner

To alleviate the vanishing monomials explosion problem during backward rewriting of non-trivial multipliers, we propose our new verification method PolyCleaner. The core idea of the method is to locally remove the vanishing monomials generated by each converging gate before backward rewriting.

In our proposed method, first all CGCs are identified and the polynomial for each CGC is extracted by the substitution of the gate polynomials in the cone. The CGC polynomial determines the output of the cone (i.e. output of the converging gate) based on its inputs. We know that a vanishing monomial contains the product of HA's outputs, and these outputs are the inputs of CGCs. Therefore, the vanishing monomials appear in the extracted CGC polynomials. Local removal of vanishing monomials from these polynomials leads to a set of vanishing-free polynomials. Now the global backward rewriting can be performed by substituting vanishing-free polynomials in the specification polynomial without appearance of any new vanishing monomial.

Algorithm 1 shows the pseudo-code of PolyCleaner. In the first step, gates are put in the different levels. This levelization is done based on the distance from the primary inputs (PIs). It means that each gate should have the minimum possible distance from PIs. The main advantage of the levelization is that the gates with the same inputs would be always in the same levels which facilitates also HA detection. In the next step, HAs are identified. Then, the converging gates are found in the circuit (see Line 3). Subsequently, the CGCs are extracted based on the converging cones and the corresponding HAs. The rest of the gates, which are not part of any CGC, are grouped based on the fanout-free regions as it increases the chance of monomials cancellation during global backward rewriting. This technique is well-known and these cones are called *fanout free*

Algorithm 1 PolyCleaner

Input: Multiplier C, Specification polynomial SP
Output: TRUE if the circuit is bug-free, and FALSE otherwise
1: $C_L \leftarrow$ Levelization (C) ▷ C_L is the levelized circuit
2: $H \leftarrow$ FindHAs (C_L) ▷ H is the set of HAs
3: $CG \leftarrow$ FindConvergingGates (H, C_L) ▷ CG is the set of converging gates
4: $CN \leftarrow$ FindCones (CG, H, C_L) ▷ CN is the set of cones
5: $F \leftarrow$ ExtractPolys (CN) ▷ F is the set of cone polynomials
6: $F \leftarrow$ RemoveVanishingMonomials (F, H)
7: **if** $SP \xrightarrow{F} 0$ **then** ▷ Backward rewriting
8: **return** $TRUE$
9: **else**
10: **return** $FALSE$
11: **end if**

cones [4, 11]. In the next step, the polynomial for each cone is extracted by substitution, and the vanishing monomials are locally removed. Finally, the global backward rewriting is done by substituting extracted polynomials in the specification polynomial. If the final remainder is equal to zero, the circuit is bug-free, otherwise it is buggy (see Line 7 – Line 11).

In the next three sections, we explain finding of CGCs, local vanishing monomials removal, and global backward rewriting in detail.

5.4 Finding CGCs

Algorithm 2 shows the proposed algorithm for identifying the converging gates and finding CGCs in the multiplier. The algorithm receives the multiplier C and the set of half adders H as inputs, and returns set of converging gates SG and set of converging gate cones SC as outputs. First, for each HA in H all the paths from the AND and XOR output to the primary outputs (POs) are extracted and stored in P_{AND} and P_{XOR} (see Line 3 – Line 4 in Algorithm 2). In fact, P_{AND} and P_{XOR} contain all the possible gates' chains connecting the output of AND and XOR gates to POs. Then, the paths in P_{AND} and P_{XOR} are checked to find out whether there are paths in P_{AND} and P_{XOR} which lead to a common gate (i.e. converge; see Line 5 – Line 7). If so, the first common member (i.e. G_C) is a converging gate candidate because it is the first place where a path from an XOR gate and a path from an AND gate converge (see Line 8). Therefore, based on Theorem 1, if G_C is not a part of a HA, then G_C is a converging gate and is added to the set of the converging gates SG (see Line 9 – Line 10). In order to determine the CGC for the corresponding converging gate G_C, the union of two paths ($p_{AND} \cup p_{XOR}$) are subtracted by their intersection ($p_{AND} \cap p_{XOR}$) and G_C is added to the result to obtain all the gates from HA's outputs to the converging gate (see Line 11 – Line 12). This process is repeated for all HAs to achieve the complete set of converging gates and CGCs.

Consider the 3-bit non-trivial multiplier of Fig. 4. There are four HAs in the circuit which are $h_1 = \{g_{15}, g_{16}\}$, $h_2 = \{g_{13}, g_{14}\}$, $h_3 = \{g_{10}, g_{11}\}$, and $h_4 = \{g_8, g_9\}$. Based on Algorithm 2, first the paths from g_{15} and g_{16} output to POs are extracted. $p_1 = \{g_3, g_1\}$ is the only path for g_{15}, and $p'_1 = \{g_6, g_3, g_1\}$ and $p'_2 = \{g_7, g_4, g_1\}$ are the paths for g_{16}. After calculating the intersection of the paths, we observe $p_1 \cap p'_1 \neq \emptyset$, therefore the first common member, i.e. g_3, is a converging gate. Based on Line 11, the corresponding CGC would be $v_1 = \{g_3, g_6\}$. The members of a CGC are sorted based on the reverse topological order of the circuit. Hence, the first member of a CGC is always the converging gate. After applying Algorithm 2, the complete list of CGCs would be $v_1 = \{g_3, g_6\}$, $v_2 = \{g_1, g_3, g_4, g_7\}$, $v_3 = \{g_1, g_3, g_6, g_4, g_7\}$ and $v_4 = \{g_5, g_8\}$ where the two first CGCs are related to HA h_1, and the rest is related to HA h_2. The converging

Algorithm 2 Finding CGCs

Input: Multiplier C, Set of HAs H
Output: Set of converging gates SG, set of converging gate cones SC
1: $SC \leftarrow \emptyset, SG \leftarrow \emptyset$
2: **for each** $h \in H$ **do**
3: $P_{AND} \leftarrow$ Find all paths from h_{and} to PO
4: $P_{XOR} \leftarrow$ Find all paths from h_{xor} to PO
5: **for each** $p_{and} \in P_{AND}$ **do**
6: **for each** $p_{xor} \in P_{XOR}$ **do**
7: **if** $p_{and} \cap p_{xor} \neq \emptyset$ **then**
8: G_C = First common member of p_{and} and p_{xor}
9: **if** $G_C \notin H$ **then**
10: $SG \leftarrow SG \cup \{G_C\}$
11: $CGC \leftarrow [(p_{and} \cup p_{xor}) - (p_{and} \cap p_{xor})] \cup \{G_C\}$
12: $SC \leftarrow SC \cup \{CGC\}$
13: **end if**
14: **end if**
15: **end for**
16: **end for**
17: **end for**
18: **return** SG, SC

gates are shown in gray in Fig. 4. Please note that if the converging gates of two different CGCs are the same (i.e. two different HAs converge to the same gate), instead of two CGCs, the union of both cones is considered as a new single CGC. For example, the first member (i.e. converging gate) of v_2 and v_3 are the same, and hence the new CGC is $v'_2 = v_2 \cup v_3 = \{g_1, g_3, g_6, g_4, g_7\}$.

5.5 Local Removal of Vanishing Monomials in CGCs

In order to facilitate the local removal of vanishing monomials, we first propose a theorem which allows further optimization for a special type of converging gates.

Theorem 2. *If all members of a CGC except the converging gate G_C are AND gates, the product of the G_C inputs is zero.*

Proof. Assume that ij is the product of the G_C inputs in Fig. 5a. This product can be substituted by the gate polynomials in CGC:

$$ij = f(w_1, w_2, \ldots, w_n, x) f'(w'_1, w'_2, \ldots, w'_m, y) \tag{12}$$

All gates in CGC are AND gates, and the polynomial of an AND gate is just the product of its inputs. Therefore, f and f' can be shown as product of their inputs:

$$ij = [w_1 w_2 \ldots w_n x][w'_1 w'_2 \ldots w'_m y] \tag{13}$$

where x and y denote the outputs of the HA. Due to appearance of xy in (13), it can be concluded that the product ij equals zero. The converging gate whose product of inputs is zero is called *M-zero* gate. □

After finding CGCs as discussed in the previous section, first M-zero gates are identified based on Theorem 2. Subsequently, the monomials containing the product of inputs polynomials for the M-zero gate are removed. For example, if the M-zero gate is an XOR, its polynomial will be $Z = i + j - 2ij = i + j$. Based on Theorem 1, the product of the converging gate inputs (i.e. ij) is the source of the vanishing monomials in backward rewriting. However, for M-zero gates the monomial containing this product is already removed. Therefore, we do not need to find the polynomial and remove vanishing monomials for the CGC if its converging gate is an M-zero gate. In Fig. 4, g_3 and g_5 are M-zero gates because the corresponding CGCs after excluding converging gates , i.e. $v_1 - \{g_3\}$ and $v_4 - \{g_5\}$, are made of just AND gates. Thus, their polynomials become $g_3 \Rightarrow w_{21} = w_{10} + w_{18}$ and $g_5 \Rightarrow w_{23} = w_{16} + w_{20}$.

For the rest of CGCs which do not contain any M-zero gate, the polynomials are extracted by substitution. Then, the monomials

containing the product of HAs' outputs (i.e. vanishing monomials) are locally removed. For $v_2' = \{g_1, g_3, g_6, g_4, g_7\}$ the steps of polynomial extraction and local vanishing monomials removal are:

$$
\begin{aligned}
&Z_5 = w_{21} + w_{22} - w_{21}w_{22}\\
&w_{21} = w_{10} + w_{18} \Longrightarrow Z_5 = w_{10} + w_{18} + w_{22} - w_{10}w_{22} - w_{18}w_{22}\\
&w_{22} = w_{15}w_{19} \Longrightarrow Z_5 = w_{10} + w_{18} + w_{15}w_{19} - w_{10}w_{15}w_{19} - w_{15}w_{18}w_{19}\\
&w_{18} = w_{11}w_{16} \Longrightarrow Z_5 = w_{10} + w_{11}w_{16} + w_{15}w_{19} - w_{10}w_{15}w_{19} - w_{11}w_{15}w_{16}w_{19}\\
&w_{19} = w_{11}w_{17} \Longrightarrow Z_5 = w_{10} + w_{11}w_{16} + w_{11}w_{15}w_{17} \mathrm{-w_{10}w_{11}w_{15}w_{17}} - \mathrm{w_{11}w_{15}w_{16}w_{17}}\\
&\qquad\qquad = w_{10} + w_{11}w_{16} + w_{11}w_{15}w_{17} \qquad (14)
\end{aligned}
$$

The red monomials contain $w_{10}w_{11}$ and $w_{16}w_{17}$ which are the product of HAs' outputs in Fig. 4. As a result, they are removed at the end of the substitution.

The rest of the gates which are not part of a CGC are grouped into fanout-free cones. This grouping increases the chance of monomial cancellation during backward rewriting. For example, $\{g_2, g_5, g_8\}$ creates a fanout-free cone in Fig. 4. Then, the polynomials for these fanout-free cones are extracted by substitution. Finally, at the end of this step, we have a set of polynomials which are completely vanishing-free.

5.6 Global Vanishing-free Backward Rewriting

The final step of verification is substituting the just determined vanishing-free polynomials for the CGCs and fanout-free cones in specification polynomial. As a consequence, the global backward rewriting process is vanishing-free and no explosion happens. In the following experiments we show the advantages of our approach. In particular we provide verification data which confirms the effectiveness of our methods for removing vanishing monomials.

6 EXPERIMENTAL RESULTS

PolyCleaner has been implemented in C++. The experiments have been carried out on an Intel(R) Core(TM) i5-4300M CPU 2.60 GHz with 16 GByte of main memory. In order to evaluate the efficiency of PolyCleaner in verification of different trivial and non-trivial multipliers, we consider a variety of multiplier architectures generated by Arithmetic Module Generator [1] known as AOKI. The multipliers are named in the order of the three stages PPG, PPA, and FSA (see details in the legend below Table 1). The generated multipliers are in RTL Verilog format. We use *Yosys* [15] to convert them to a flattened gate-level Verilog netlist. Since AOKI cannot generate multipliers with more than 64 input bits for one input operand, we generated larger multipliers using *Yosys*.

Table 1 reports the run-times of different verification methods for the various multipliers. Please note that the *Time-Out (TO)* has been set to 200 hours. The first column of the table shows the type of multiplier (see below the table for abbreviations). The second column *Size* gives the size of multiplier based on the input bits.

The run-time (in seconds) of our proposed method is reported in detail in the third column **PolyCleaner**, which consists of five subcolumns: *Parsing & Levelization* reports the required time for parsing the gate-level netlist, converting it to the internal data structure, and finally levelizing the gates. *Finding Cones* refers to the time which is needed for finding CGCs and fanout-free cones, so Algorithm 2. *Local Van. Removal* presents the consumed time for generating the polynomials for each cone and removing the vanishing monomials. Global backward rewriting time is reported in *Glob. Backw. Rewriting*. Finally, the overall run-time of PolyCleaner which is the sum of four previous subcolumns is presented in *Overall*. The numbers show that Finding Cones phase time is 51% of total verification time in average, and therefore the most time-consuming phase of PolyCleaner. But as can be seen, this time investment pays off since PolyCleaner can verify all benchmarks.

The forth column **State-of-the-art methods** of Table 1 reports the run-times of the state-of-the-art verification methods. This column consists of five subcolumns: *Commercial* refers to the run-time of commercial formal verification tool OneSpin, while the remaining subcolumns report the run-times of the most recent SCA-based verification approaches. As can be seen the commercial tool verifies multipliers up to 16 input bits. The proposed methods [4], [17], and [9] are capable of verifying the trivial multipliers (i.e. $SP \circ AR \circ RC$ and $BP \circ AR \circ RC$) where no vanishing monomials are generated during backward rewriting. However, these methods return time-out in verification of even small non-trivial multipliers. The proposed method in [11] can verify some of the non-trivial multipliers as the authors presented a heuristic to detect and remove some of the vanishing monomials. Nevertheless, the verification method [11] is not robust as can be seen in column [11]: it fails for 12 non-trivial multipliers. Moreover, for the benchmarks where it was successful, the run-time is considerably larger than the run-time of PolyCleaner (sometimes event two orders of magnitudes).

Table 2 presents the verification data reported by PolyCleaner and by this gives very interesting insights. The first and the second column of the table show the type and size of the multiplier, respectively. The third column **#Gates** reports the number of gates in the multiplier. The via Yosys generated multiplier with 512 bits per input, which consists of approximately 1.6 million gates, is the largest multiplier in our experiments. The number of the converging gates is reported in the forth column **#Converg**. For non-trivial multipliers, the number of the converging gates varies based on the type of the architecture. The fifth column **#Cones** reports the number of CGCs and fanout-free cones in total. The number of the canceled vanishing monomials is shown in the sixth column **#Van**. In trivial multipliers, no vanishing monomial are generated, thus no cancellation happens. In contrast, in case of non-trivial multipliers, more than 2 million vanishing monomials have to be removed before global backward rewriting. Finally, the seventh column **#maxPoly** reports the maximum size of the current polynomial SP_i during backward rewriting by counting the number of monomials. Now consider a group of multipliers of the same size (column Size): The maximum polynomial size of trivial multipliers is approximately equal to that of the non-trivial multipliers, for example around 600 for the 16×16 multipliers. This result clearly demonstrates that PolyCleaner efficiently kills all vanishing monomials before backward rewriting.

7 CONCLUSION

In this paper, we have introduced the SCA-verifier PolyCleaner which allows formal verification of non-trivial million-gate multipliers. Based on a new theory for the origin of vanishing monomials, our approach allows for local cancellation of vanishing monomials in converging gates cones starting from half adders. As a consequence global vanishing-free backward rewriting can be performed by PolyCleaner. The experimental results showed the efficiency of PolyCleaner in verification for non-trivial million-gate multipliers where the other state-of-the-art methods failed.

ACKNOWLEDGMENTS

This work was supported by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative, and by the German Academic Exchange Service (DAAD).

Table 1: Run-times of verifying different types of multipliers (run-times in seconds)

Benchmark post-synth.	Size	PolyCleaner: Parsing & levelization	Finding cones	Local Van. Removal	Glob. Backw. Rewriting	Overall	State-of-the-art methods: Commercial	[4]	[17]	[9]	[11]
SP∘AR∘RC	16×16	0.20	0.23	0.07	0.09	0.59	63.00	0.38	0.01	1.49	3.01
SP∘CT∘BK		0.06	0.19	0.08	0.10	0.43	53.00	TO	TO	TO	4.00
SP∘DT∘LF		0.07	0.27	0.15	0.10	0.58	44.00	TO	TO	TO	3.21
SP∘WT∘CL		0.10	0.47	0.34	0.11	1.01	50.00	TO	TO	TO	7.63
BP∘AR∘RC	16×16	0.14	0.20	0.07	0.10	0.50	74.00	0.32	0.01	TO	6.34
BP∘CT∘BK		0.06	0.14	0.08	0.10	0.37	44.00	TO	TO	TO	10.47
BP∘DT∘LF		0.05	0.18	0.14	0.10	0.46	44.00	TO	TO	TO	11.50
BP∘WT∘CL		0.09	0.36	0.35	0.10	0.90	68.00	TO	TO	TO	TO
SP∘AR∘RC	32×32	1.25	3.94	0.74	1.19	7.12	TO	3.72	0.02	39.73	55.60
SP∘CT∘BK		0.40	2.74	0.86	1.39	5.38	TO	TO	TO	TO	81.09
SP∘DT∘LF		0.51	4.34	1.65	1.34	7.82	TO	TO	TO	TO	68.28
SP∘WT∘CL		1.01	8.73	5.24	1.44	16.43	TO	TO	TO	TO	1105.44
BP∘AR∘RC	32×32	0.95	2.50	0.57	1.11	5.13	TO	2.90	0.02	TO	56.71
BP∘CT∘BK		0.31	1.63	0.65	1.15	3.75	TO	TO	TO	TO	247.17
BP∘DT∘LF		0.31	2.23	1.24	1.09	4.86	TO	TO	TO	TO	265.08
BP∘WT∘CL		0.80	5.95	5.75	1.35	13.86	TO	TO	TO	TO	TO
SP∘AR∘RC	64×64	12.98	67.59	10.44	18.41	109.43	TO	49.91	0.14	TO	846.23
SP∘CT∘BK		3.55	42.99	11.22	21.66	79.44	TO	TO	TO	TO	1,952.06
SP∘DT∘LF		6.99	72.40	20.53	20.84	120.76	TO	TO	TO	TO	2,273.64
SP∘WT∘CL		16.21	210.41	84.59	24.48	335.70	TO	TO	TO	TO	TO
BP∘AR∘RC	64×64	9.21	38.45	6.95	15.84	70.43	TO	37.18	0.09	TO	911.07
BP∘CT∘BK		1.99	23.23	7.29	16.01	48.52	TO	TO	TO	TO	1,796.42
BP∘DT∘LF		2.66	32.68	13.07	15.11	63.53	TO	TO	TO	TO	TO
BP∘WT∘CL		9.78	112.73	83.64	16.51	222.66	TO	TO	TO	TO	TO
yosys_mult	128×128	105.52	1,445.77	183.39	650.05	2,384.73	TO	TO	TO	TO	TO
yosys_mult	192×192	538.15	8,360.23	951.12	3,759.72	13,609.22	TO	TO	TO	TO	TO
yosys_mult	256×256	1,324.88	20,325.90	2,344.55	8,664.24	32,659.57	TO	TO	TO	TO	TO
yosys_mult	320×320	3,269.63	53,332.48	5,964.00	24,602.92	87,169.02	TO	TO	TO	TO	TO
yosys_mult	384×384	10,150.71	137,212.26	17,032.26	66,466.43	230,861.66	TO	TO	TO	TO	TO
yosys_mult	448×448	12,301.13	221,666.97	22,706.62	105,182.33	361,857.05	TO	TO	TO	TO	TO
yosys_mult	512×512	21,041.23	385,253.07	37,795.14	190,075.10	634,164.54	TO	TO	TO	TO	TO

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out
Stage 2 ⇒ **AR**: Array **CT**: Compressor tree **DT**: Dadda tree **WT**: Wallace tree
Stage 3 ⇒ **RC**: Ripple carry adder **BK**: Brent-Kung adder **LF**: Lander-Fischer adder **CL**: Carry look-ahead adder

Table 2: Verification data for different types of multipliers

Benchmark post-synth.	Size	#Gates	#Converg	#Cones	#Van	#maxPoly
SP∘AR∘RC	16×16	1,856	0	736	0	540
SP∘CT∘BK		1,722	31	808	273	627
SP∘DT∘LF		1,924	66	789	1,587	646
SP∘WT∘CL		2,633	276	852	6,708	652
BP∘AR∘RC	16×16	1,691	0	588	0	1,004
BP∘CT∘BK		1,540	34	577	286	1,005
BP∘DT∘LF		1,665	70	553	2,045	1,002
BP∘WT∘CL		2,447	301	604	7,199	1,004
SP∘AR∘RC	32×32	7,808	0	3,008	0	2,108
SP∘CT∘BK		7,130	79	3,225	1,422	2,531
SP∘DT∘LF		8,046	175	3,124	15,200	2,652
SP∘WT∘CL		12,066	1,437	3,353	82,061	2,648
BP∘AR∘RC	32×32	6,314	0	2,063	0	4,050
BP∘CT∘BK		5,766	82	2,015	1,248	4,059
BP∘DT∘LF		6,263	180	1,895	14,498	4,050
BP∘WT∘CL		10,626	1,547	2,098	87,201	4,054
SP∘AR∘RC	64×64	32,000	0	12,160	0	8,316
SP∘CT∘BK		28,737	169	12,707	11,688	10,179
SP∘DT∘LF		32,680	428	12,403	94,805	10,764
SP∘WT∘CL		52,083	6,694	13,050	844,849	10,796
BP∘AR∘RC	64×64	24,442	0	7,727	0	16,290
BP∘CT∘BK		21,872	171	7,187	7,170	16,305
BP∘DT∘LF		24,006	434	6,884	96,023	16,290
BP∘WT∘CL		44,173	6,923	7,482	897,991	16,300
yosys_mult	128×128	99,722	472	66,896	44,237	32,770
yosys_mult	192×192	223,693	726	149,846	149,584	73,730
yosys_mult	256×256	396,997	997	265,747	248,859	131,074
yosys_mult	320×320	619,857	1,234	414,815	1,305,538	204,802
yosys_mult	384×384	891,114	1,489	595,899	817,426	294,914
yosys_mult	448×448	1,211,754	1,745	809,979	2,084,311	401,410
yosys_mult	512×512	1,582,385	1,995	1,057,680	1,253,955	524,290

REFERENCES

[1] Arithmetic module generator based on ACG. www.aoki.ecei.tohoku.ac.jp/arith/.
[2] M. Ciesielski, C. Yu, D. Liu, and W. Brown. Verification of gate-level arithmetic circuits by function extraction. In *DAC*, pages 52:1–52:6, 2015.
[3] D. A. Cox, J. Little, and D. O'Shea. *Ideals Varieties and Algorithms*. Springer, 1997.
[4] F. Farahmandi and B. Alizadeh. Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *MICPRO*, 39(2):83–96, 2015.
[5] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using rewrite rule laboratory. *Formal Methods in System Design: An International Journal*, 13(2):127–158, 1998.
[6] J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *TCAD*, 32(9):1409–1420, Sept 2013.
[7] A. Mahzoon, D. Große, and R. Drechsler. Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers. In *ISVLSI*, 2018.
[8] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, O. Wienand, and E. Karibaev. Modeling of custom-designed arithmetic components in ABL normalization. In *FDL*, pages 124–129, 2008.
[9] D. Ritirc, A. Biere, and M. Kauers. Column-wise verification of multipliers using computer algebra. In *FMCAD*, pages 23–30, 2017.
[10] D. Ritirc, A. Biere, and M. Kauers. Improving and extending the algebraic approach for verifying gate-level multipliers. In *DATE*, pages 1556–1561, 2018.
[11] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *DATE*, pages 1048–1053, 2016.
[12] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176, 2016.
[13] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *TCAD*, 23(5):586–597, 2004.
[14] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham. Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems. *TC*, 56(10):1401–1414, 2007.
[15] C. Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.
[16] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *TCAD*, 35(12):2131–2142, 2016.
[17] C. Yu, M. Ciesielski, and A. Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *TCAD*, 1(1):1–5, 2017.