Symbolic Computer Algebra for Multipliers Revisited - It's All About Orders and Phases

Alexander Konrad Christoph Scholl
University of Freiburg, Freiburg, Germany
{konrada, scholl}@informatik.uni-freiburg.de

***Abstract*—Using Symbolic Computer Algebra (SCA) enabled a huge progress in formal verification of arithmetic circuits in recent years. Several different approaches have been proposed showing great success especially for the verification of multipliers. Some of them are based on precomputing and simplifying polynomials for specific circuit structures like converging cones while others take advantage of known or detected hierarchy information to replace and simplify particular sub-circuits of the design. In this paper we propose a new method that avoids the use of such methods and applies only two dynamic approaches: (1) choosing a good substitution order for the backward rewriting process and (2) adjusting the phases of signals occurring in the intermediate polynomials during the verification process. Both methods are simply based on a greedy local search taking the sizes of intermediate polynomials into account. Our experimental results show that this method is very competitive with already existing tools and it improves their robustness, e.g. against optimizations of the verified circuits using logic synthesis.**

I. Introduction

Arithmetic circuits account for an important part in circuit designs, be it general-purpose processors or specialized hardware aimed for computationally intensive applications like cryptography, signal processing or machine learning. The infamous Pentium bug [1] from 1994 raised the communities' awareness about the need of formal methods to verify the correctness of arithmetic circuit designs. Today, the design of arithmetic circuits is not limited to the major processor vendors only, but is also done by various different suppliers of special-purpose embedded hardware who cannot afford to employ large teams of specialized verification engineers being able to provide human-assisted theorem proofs. This results in a growing interest for *fully automatic formal verification* of arithmetic circuits.

Especially the verification of multiplier and divider circuits remained a challenging problem for long time. While BDD-based methods [2], [3] suffer from exponential space complexity, SAT-based methods [4], [5] face exponential run times for larger bit-widths. *BMDs [6]–[8] were presented as a suitable verification data structure for multipliers, but unfortunately *BMDs based verification approaches did not fulfill expectations in practice. Nevertheless, methods based on *Symbolic Computer Algebra (SCA)* have shown great progress for the automatic formal verification of gate-level multipliers and dividers in recent years. They enabled the verification of large and complex arithmetic circuit structures, including finite field multipliers [9], integer multipliers [10]–[25], modular multipliers [26] and divider circuits [27]–[31]. Here the verification task has been reduced to an ideal membership test for the specification polynomial based on so-called backward rewriting, proceeding from the outputs of the circuit in the direction of the inputs. For integer multipliers, SCA-based methods are closely related to verification methods based on word-level decision diagrams like *BMDs, since polynomials can be seen as "flattened" *BMDs [29]. In addition, rewriting based approaches [32], [33] have also shown to be able to verify complex multipliers as well as arithmetic modules with embedded multipliers at the register transfer level.

Most multiplier architectures are basically composed of three stages: (1) Partial Product Generator (PPG), (2) Partial Product Accumulator (PPA) and (3) Final Stage Adder (FSA). Surprisingly, difficulties with exponential polynomial sizes in SCA-based multiplier verification often occurred when using fast adders [34] as the FSAs, and not so often when using complex PPAs. A first hint in this direction was already given by the theoretical analysis for *BMDs in the work of Keim et al. [35]. Most recently three major approaches were published to tackle this problem:

- [18], [19], [21], [24] use reverse engineering and detection of converging cones to precompute polynomials for sub-circuits and simplify those polynomials early on by avoiding so-called *vanishing monomials*.
- [20], [22], [25], [36] use heuristics to detect a parallel prefix adder, replace it by a simple ripple-carry adder and use SAT to prove the soundness of this replacement, changing the multiplier circuit into a structure for which rewriting is much easier.
- [23] uses the adder detection from [20] to determine a parallel prefix adder, but it does not replace the adder by a simpler form. Instead, it introduces *dual variables* and uses a new approach of *carry rewriting* to avoid the occurrence of exponential peak polynomials in the rewriting steps of the parallel prefix adder.

In this paper, we present a new method which consists of two dynamic approaches. First, we advance the idea from [21] of dynamically finding a good substitution order for the backward rewriting process. For this we partition the circuit into blocks and use a hierarchical approach to select a good candidate block for the next substitution step as well as a

This work was supported by the German Research Foundation (DFG) within the project VerA (SCHO 894/5-1).

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_32

good substitution order for the block itself, with the aim of keeping intermediate polynomials as small as possible. Second, we adjust the phases of signals occurring in intermediate polynomials during the backward rewriting with the same goal. Both of our new dynamic approaches work as greedy local search algorithms that only take the current polynomial size into account. As a result, we are not as dependent as other approaches on the detection of specific sub-circuits. This makes our method more robust to circuit optimizations. The simplicity of the approach additionally paves the way for easier certifiability.

The experimental results show that our simple method is very competitive with other existing approaches, being able to verify almost all unoptimized 64-bit multiplier circuits which are at our disposal. However, the main advantage of our method is seen in the verification of optimized benchmarks where we outperform other tools by a large margin.

The paper is structured as follows: In Sect. II we provide background on the basic SCA-method and multiplier circuits. In Sect. III we summarize and discuss existing methods to motivate the need of the novel approach presented in Sect. IV. We evaluate our new approach in Sect. V and conclude with final remarks in Sect. VI.

II. Preliminaries

A. SCA for Verification

For the presentation of SCA we basically follow [29]. SCA-based approaches work with polynomials and reduce the verification task to an ideal membership test using a Gröbner basis representation of the ideal. The ideal membership test is performed using polynomial division. While Gröbner basis theory is very general and, e.g., can be applied to finite field multipliers [9] and truncated multipliers [20] as well, for integer arithmetic it boils down to substitutions of variables for gate outputs by polynomials over the gate inputs (in reverse topological order), if we choose an appropriate "term order" (see [14] or [17], e.g.). Here we restrict ourselves to exactly this view.

For integer arithmetic we consider polynomials over binary variables (from a set $X = \{x_1, \ldots, x_n\}$) with integer coefficients from $\mathbb{Z}$, i. e., a polynomial is a sum of terms, a term is a product of a monomial with an integer, and a monomial is a product of variables from X. Polynomials represent *pseudo-Boolean functions* $f : \{0,1\}^n \mapsto \mathbb{Z}$.

As a simple example consider the full adder from Fig. 1. The full adder defines a pseudo-Boolean function $f_{FA} : \{0,1\}^3 \mapsto \mathbb{Z}$ with $f_{FA}(a_0, b_0, c) = a_0 + b_0 + c$. We can compute a polynomial representation for f_{FA} by starting with a weighted sum $2c_0 + s_0$ (called the "output signature" in [13]) of the output variables. Step by step, we replace the variables in polynomials by the so–called "gate polynomials". This replacement is performed in reverse topological order of the circuit, see Fig. 1. We start by replacing c_0 in $2c_0 + s_0$ by its gate polynomial $h_2 + h_3 - h_2h_3$ (which is derived from the Boolean function $c_0 = h_2 \vee h_3$). Finally, we arrive at the polynomial $a_0 + b_0 + c$ (called the "input signature" in [13]) representing the pseudo-Boolean function defined by the circuit. During this procedure (which is called *backward rewriting*) the polynomials are simplified by reducing powers v^k of variables v with $k > 1$ to v (since the variables are binary), by combining terms with identical monomials into one term, and by omitting terms with leading factor 0. We can also consider $a_0 + b_0 + c = 2c_0 + s_0$ as the "specification" of the full adder. The circuit implements a full adder iff backward rewriting, now starting with $2c_0 + s_0 - a_0 - b_0 - c$ instead of $2c_0 + s_0$, reduces the "specification polynomial" to 0 in the end. (This is the notion usually preferred in SCA-based verification.)

The correctness of the method relies on the fact that polynomials (with the above mentioned simplifications resp. normalizations) are canonical representations of pseudo-Boolean functions (up to reordering of the terms). (This is proven in [29], e.g..)

B. Multiplier Circuits

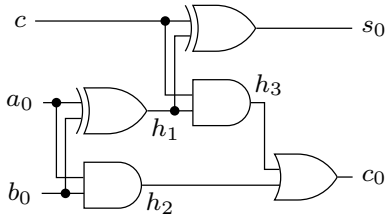
In the following, we briefly summarize textbook knowledge on multipliers. For more details, see [34], e.g.. Most integer multipliers are composed of three stages: The first stage is the *Partial Product Generator (PPG)* which generates partial products from the bits of the two input operands. Examples are *Simple PPGs*, which just compute the logical AND of all bits of the first input and all bits of the second input, or PPGs with *Booth Encoding* which reduce the number of generated partial products using Booth's Algorithm [37]. The second stage is the *Partial Product Accumulator (PPA)* which sums up all the partial products until they are reduced to two numbers only. Well-known accumulation structures are array accumulation, Wallace trees [38] or Dadda trees [39]. The third stage consists of the *Final Stage Adder (FSA)* which converts the resulting two numbers from the PPA stage into the final binary representation of the output product. Any two operand adder networks can be used here, ranging from simple examples such as the well-known ripple-carry adder to more complex structures such as various implementations of parallel prefix adders. Such implementations include the Kogge-Stone adder [40], the Brent-Kung adder [41] and the Ladner-Fischer adder [42], to name just a few.

C. Specification Polynomial for Unsigned Multipliers

In this paper, we focus on unsigned gate-level integer multipliers with input bits $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1}$ of multiplier and multiplicand and output bits $p_0, \ldots, p_{2n-1}$ of the product. The corresponding specification polynomial, which is the starting point of the backward rewriting process, is

$$P_{spec}(p_0, \ldots, p_{2n-1}, a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}) = \sum_{i=0}^{2n-1} 2^i p_i - (\sum_{j=0}^{n-1} 2^j a_j) \cdot (\sum_{k=0}^{n-1} 2^k b_k). \quad (1)$$

As explained in Sec.II-A the multiplier circuit is correct iff backward rewriting reduces P_{spec} to 0.



$$
\begin{array}{rcl}
c_0 & = & h_2 + h_3 - h_2 h_3 \\
s_0 & = & c + h_1 - 2ch_1 \\
h_3 & = & ch_1 \\
h_2 & = & a_0 b_0 \\
h_1 & = & a_0 + b_0 - 2a_0 b_0
\end{array}
$$

$$
\begin{array}{rl}
 & 2c_0 + s_0 \\
\xrightarrow{c_0} & 2h_2 + 2h_3 - 2h_2 h_3 + s_0 \\
\xrightarrow{h_3} & 2h_2 + 2ch_1 - 2ch_1 h_2 + s_0 \\
\xrightarrow{s_0} & 2h_2 - 2ch_1 h_2 + c + h_1 \\
\xrightarrow{h_2} & 2a_0 b_0 - 2a_0 b_0 ch_1 + c + h_1 \\
\xrightarrow{h_1} & a_0 + b_0 + c
\end{array}
$$

Fig. 1: Full adder circuit with series of substitutions.

In the SCA-based verification for integer arithmetic we use terms with coefficients from $\mathbb{Z}$ in the polynomials. However, for n-bit integer multipliers the polynomial computations can be performed in $\mathbb{Z}_{2^{2n}}$ instead of $\mathbb{Z}$ which is desirable, since it improves efficiency by reducing the maximal coefficient size. In this case the specification polynomial can be defined as

$$P_{spec,mod}(p_0,\ldots,p_{2n-1},a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1}) = \left(\sum_{i=0}^{2n-1} 2^i p_i - (\sum_{j=0}^{n-1} 2^j a_j)\cdot(\sum_{k=0}^{n-1} 2^k b_k)\right) \mod 2^{2n}. \quad (2)$$

In the following paragraph, we give the sketch of a proof that a circuit fulfills the specification from Eqn. (1) iff it fulfills the specification from Eqn. (2) (a similar proof is given in [20]): Consider the polynomial P_{spec} from Eqn. (1). For all possible assignments to $p_i, a_j, b_k \in \{0,1\}$ it holds that P_{spec} evaluates to a value in $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ (which is easy to see since the upper bound is reached for $p_0,\ldots,p_{2n-1}=1$ and $a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1}=0$, and the lower bound is reached for the opposite case). By replacing variables by their gate polynomials (without modulo 2^{2n}) we obtain functions depending on a different set of variables, but their range is still a subset of the range of P_{spec}. In the end (after all substitutions) we get a function whose range is still a subset of $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ (regardless of whether the circuit is correct or not). If we now apply a modulo 2^{2n} operation on those values, they might change, but still 0 will be mapped to 0 and values different from 0 will be mapped to values different from 0, since the absolute values in $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ are all smaller than 2^{2n}. Therefore it holds: All assignments consistent with some circuit C evaluate P_{spec} to 0 iff all assignments consistent with C evaluate $P_{spec,mod}$ to 0. This means that we can perform all polynomial computations in $\mathbb{Z}_{2^{2n}}$ instead of $\mathbb{Z}$ during backward rewriting.

III. Review of Existing Approaches

A large number of excellent and non-trivial methods has been presented in the literature to enable SCA-based multiplier verification and increase its efficiency. Most of those methods are tailored towards certain structural properties of existing multiplier circuits. Here we will review and analyze existing approaches from the literature. Some of those techniques mentioned here have been described in the literature based on computations with Gröbner bases (like the elimination of certain polynomials from Gröbner bases). Here we prefer a description based on backward rewriting, as already mentioned in Sect. II-A. Note however that the difference is only in the representation, not in the actual contents.

The first approach is the *detection of so-called "atomic blocks"* in the multiplier design [19]. Atomic blocks may be XOR gates, half adders (HAs), full adders (FAs), or Compressors (CMs). In [19], [24] and [17] the information about atomic blocks is used to enable a *hierarchical polynomial computation*: Polynomials for atomic blocks are computed first and during the backward rewriting the local polynomials for atomic blocks are used for replacements in the global specification polynomial. If atomic blocks have several outputs, then either polynomials for the outputs are computed separately and are handled as mentioned above, or the "word-level" polynomial for the atomic block (like $2c_0 + s_0 = a_0 + b_0 + c$ for a FA) is used for rewriting the global specification polynomial, provided that it has an appropriate form (in the case of the FA mentioned above: if the only terms containing c_0 and s_0 in the specification polynomial are $2^{k+1}c_0$ and $2^k s_0$).

Note that in our work we use atomic block detection as well, but we always perform replacements by backward rewriting on the gate level, not on the block level. Atomic blocks are only used to guide the order in which we replace the gates.

Sayed-Ahmed et al. [14] observed that polynomials can be simplified based on the observation that the sum outputs s and the carry outputs c of HAs can never be 1 at the same time. Therefore, terms containing both c and s (called "*vanishing monomials*") can be removed from the polynomial immediately. The authors of [14], [18] argue that vanishing monomials are the main cause of polynomial size explosions during backward rewriting of multipliers. This technique is used in other works like [15] as well. In [18] the observation was used in the context of a hierarchical polynomial computation. [18] computes so-called *Convergent Gate Cones* (CGCs) which are logic cones where paths from outputs of the same half adder (HA) converge in a common node. The polynomials of CGCs are precomputed, vanishing monomials are removed

during this computation, and the "cleaned" polynomials for CGCs are used during the final backward rewriting. If large CGCs occur, the advantage of this method is not clear at first sight, since the vanishing monomials can only be removed at the end of backward rewriting of a CGC. However, in fast final stage adders (FSAs) like carry-lookahead or parallel prefix adders there are usually overlapping CGCs of different sizes. It is essential that the method from [18] starts with the smaller CGCs, cleans them, and uses the resulting polynomials when polynomials for larger, overlapping CGCs are computed. In [26] the approach has been generalized to arbitrary pairs of contradicting signals.

The idea of a hierarchical polynomial computation has also been applied in [21], [24] to *fanout-free cones* of gates which are not included in atomic blocks and CGCs. This technique has been called "*Common Term Rewriting*" in [14].

The special treatment of CGCs was motivated by difficulties of SCA-based methods when the FSA is a parallel prefix adder. Kaufmann et al. [20], [23] propose other techniques to tackle the same problem. They substitute a final parallel prefix adder by a simple ripple-carry adder and then verify the resulting simplified multiplier. In [20] the equivalence of the substituted parallel prefix adder with a ripple-carry adder is proven by SAT solving. [23] introduces the concepts of *dual variables* and *tail substitution* and uses them during "*carry rewriting*" inside the detected FSA. For both [20] and [23] certain circuit structures inside the FSA as well as the FSA bounds have to be structurally detected. Those methods work well and are fast for clean multipliers (as the circuit rewriting from [32], [33] as well), but they become problematic, if logic synthesis steps at the gate level have destroyed the clear structure. This observation is clearly confirmed by our experiments in Sect. V.

Finally, the order of traversing the circuit during backward rewriting is of utmost importance. Reverse topological orderings are usually not unique, but leave a lot of degrees of freedom. Different orderings may lead to totally different sizes of the intermediate polynomials during backward rewriting. Some methods try to find good static orders for the traversal (and sometimes need structural information about the circuit to find them). [13] performs a *"row-wise" traversal* of the multiplier (which resembles a breadth-first search in the circuit, augmented with information on hierarchy bounds), [15] performs a *"column-wise" traversal* (which resembles a depth-first search). In [15] a decomposition of the multiplier specification polynomial into column-wise "sub-specifications" is used in addition (needing an additional multiplier-specific reasoning for correctness). In contrast, in [21] a dynamic substitution order was proposed. Here the order in which backward rewriting processes the circuit is not determined beforehand, but it is adjusted dynamically based on the sizes of intermediate polynomials. It is important to note that the dynamic substitution order is used in the context of hierarchical polynomial computation on the level of "components" (atomic blocks, CGCs, fanout-free cones) here. The substitution orders to compute polynomials for the components are still chosen statically.

Most of the approaches mentioned above work well, when they are applied to clean multipliers at the gate level where logic synthesis has not been applied, but they become weaker, when they are applied to multipliers which are optimized by logic synthesis. This does not only hold for methods which rely on the detection of FSA boundaries and FSA structures as mentioned above [20], [23], but also for other methods: The hierarchical polynomial computation becomes weaker, if the boundaries of atomic blocks are destroyed by logic synthesis; the removal of vanishing monomials based on CGCs gets into trouble, if HAs at the origin of CGCs cannot be detected anymore due to logic restructuring; the computation of static replacement orders may suffer as well, if the circuit structures are destroyed on which the order computation relies (like XOR-skeletons for the computation of column-wise slices in [25]).

For this reason, we investigate in this paper whether it is possible to increase the robustness of SCA-based verification of multipliers by avoiding complex approaches which are vulnerable against changes to the clean multiplier structure. Our goal is to simplify the approach and at the same time to increase its robustness. We restrict ourselves to a flat and non-hierarchical polynomial computation based on backward rewriting with gate polynomials, but we invest considerable effort into the computation of good substitution orders. Moreover, we deeply integrate the technique of phase optimization into our order optimization to make the approach more robust against the selection of unfavourable orders.

IV. Backward Rewriting with Phase and Order Optimization

In this section we present our SCA-based verification method which is based on two simple ingredients: The optimization of "phases" of signals and the optimization of the order of gate replacements during backward rewriting. Both methods are integrated into an overall phase and order optimization, but we begin with the description of phase optimization.

A. Phase Optimization

The idea of phase optimization is to adjust the phases of occurring signals during backward rewriting to keep intermediate polynomial sizes small and to make backward rewriting more robust, e.g., against different orderings. The approach is based on the observation that replacing variables by their negation in intermediate polynomials may reduce the sizes of those polynomials.

Example 1. *Let us consider the function* $f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$ *which computes the disjunction of 3 inputs. It is easy to see that the polynomial for* f *is* $p = x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3$ *with 7 terms. Now we change the phase of one variable, let's say* x_1*, i.e., we introduce a "new input variable" called* $\overline{x_1}$ *representing the negation of* x_1 *and replace* x_1 *in* p *with* $1 - \overline{x_1}$*. This results in* $p' = 1 - \overline{x_1} + \overline{x_1}x_2 + \overline{x_1}x_3 - \overline{x_1}x_2x_3$ *with 5 terms. Changing the phase of* x_2 *(i.e. replacing* x_2 *with* $1 - \overline{x_2}$*) results in* $p'' =$

Algorithm 1 Phase Optimization.

Input: Polynomial P, Set of candidate signals S
Output: Polynomial P with optimized phases
1: **for each** $s \in S$ **do**
2: $old\_size \leftarrow$ size(P);
3: $P \leftarrow$ Flip-Phase(s, P); ▷ flip signal s in P
4: $new\_size \leftarrow$ size(P);
5: **if** $old\_size < new\_size$ **then** $P \leftarrow$ Flip-Phase(s, P) ▷ flip s back
6: **return** P;

$1-\overline{x_1}\,\overline{x_2}+\overline{x_1}\,\overline{x_2}x_3$ *and changing the phase of* x_3 *finally results in* $p''' = 1 - \overline{x_1}\,\overline{x_2}\,\overline{x_3}$ *with 2 terms. The example shows that phase optimization is able to reduce the sizes of polynomials. If* $f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$ *is part of a larger circuit and* x_1 *has to be replaced with the polynomial of another gate* g*, then the phase flipping of* x_1 *has of course to be reverted before replacing* x_1 *with the polynomial of* g*. Anyway, our hope is that phase optimization is able to reduce the size of intermediate polynomials. If* g *is constant 1, e.g., then reverting the phase flipping of* x_1 *results in* $1 - \overline{x_2}\,\overline{x_3} + x_1\overline{x_2}\,\overline{x_3}$ *and after replacing* x_1 *with constant 1 we arrive at* $p'''' = 1$*. The experimental results in Sect. V show that our hope for the benefits of phase optimization is well founded.*

We perform phase optimization by a simple greedy algorithm, see Alg. 1. The algorithm is a general function of the polynomial package and, therefore, does not need any circuit information, but only uses the polynomial which should be optimized and a set of candidate signals which are objective to the optimization. For every candidate signal s the following is done: Saving the current size of the polynomial P (which is just the number of terms in P, line 2), then flipping the phase of the signal s in P (line 3). If the now achieved polynomial size is larger than before, the phase change was not beneficial and s is flipped again to restore the previous polynomial (line 5). Only if the polynomial size could have been reduced by the phase change, the flipped phase is kept in the polynomial and the algorithm continues with the next candidate signal. In the end, a smaller polynomial with optimized phases is found or (in case no phase changes led to a smaller size) the original polynomial remains. Therefore our phase optimization never increases the polynomial size. In our implementation, we perform phase optimization after each rewriting step. To save computation time, we restrict the search space of the phase optimization by choosing the set S of candidate signals in Alg. 1 as the set of variables which were newly introduced into the polynomial in the last rewriting step.

The correctness of flipping phases of signals during backward rewriting can be seen easily: Consider some arbitrary gate g of a circuit computing the signal x_i. Phase flipping of x_i can be simulated by inserting two consecutive inverters immediately at the output of g (which apparently does not change the function of the circuit). The signal after the first inverter is called $\overline{x_i}$, the signals before the first inverter and after the second inverter are called x_i. Phase flipping for x_i corresponds to backward rewriting of the second inverter (introducing $\overline{x_i}$ into the polynomial). Reverting phase flipping before replacement of g corresponds to backward rewriting of the first inverter (introducing x_i into the polynomial again).

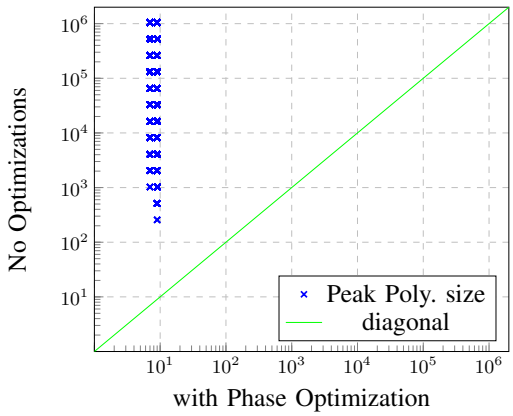


Fig. 2: Peak polynomial sizes for different orderings.

Before showing how we integrate phase optimization into our dynamic order optimization approach, we present an example which shows that phase optimization makes backward rewriting much more robust against changes of replacement orders. The example is a toy example chosen for illustration, but the demonstrated effect is similar to the effects occurring during backward rewriting of parallel prefix adders which contain large OR trees in their implementation. Large OR trees may be a problem for SCA-based methods due to their exponential polynomial representation, see also [20] and [23].

Example 2. *Consider a circuit with 32 input signals* $i_0, \ldots, i_{31}$*. The 32 input signals are connected to 32 inverters and the outputs of inverters are the inputs of a balanced OR tree. It is clear that this circuit basically implements a NAND function with 32 inputs. Thus, backward rewriting starting with the polynomial* $p_o = o$ *for the output variable* o *leads to the final polynomial* $p = 1 - i_0 \cdot \ldots \cdot i_{31}$*. The final polynomial has size 2 (number of terms).*

Backward rewriting proceeds in reverse topological order and of course there is a huge number of different possible reverse topological orders to traverse the circuit. In our experiment, we randomly choose one of the possible reverse topological orders and perform backward rewriting with and without phase optimization. We repeat the experiment 10.000 times with different random choices. Fig. 2 shows a scatter plot with each data point representing one possible traversal order. The y-axis indicates the peak polynomial size during backward rewriting using this order without *phase optimization and the x-axis* with *phase optimization (note that the axes are scaled logarithmically). Fig. 2 shows that with phase optimization the peak polynomial size is always between 7 and 9. Without phase optimization, the peak polynomial sizes vary considerably and in 698 out of 10.000 cases our chosen upper bound of 1.000.000 terms is exceeded. The large intermediate sizes of the polynomials in the version without phase optimization can be easily explained by the fact that the intermediate polynomials often represent disjunctions of a large number of inputs*

which have exponential representations as polynomials [23]. Apparently, phase optimization keeps intermediate polynomial sizes small and makes the backward rewriting much more robust.

Our phase optimization is related to the method from [23] using dual variables. In contrast to [23] we do not introduce dual phases x_i and $\overline{x_i}$ of variables, but we only enable flipping the phases of variables, i.e., we flip *all* occurrences of a signal in the polynomial. This makes the approach much simpler and does not need additional handling and simplification steps in the polynomial (like the merging algorithm in [23] which checks if two terms can be merged into one because they only differ in one dual variable).

Interestingly, as also observed in [43], phase optimization is strongly related to replacing the so-called *positive Davio decomposition* by *negative Davio decomposition* in K*BMDs [44]. In contrast to K*BMDs, *BMDs [6] only allow positive Davio decomposition. The positive Davio decomposition wrt. variable x_1 decomposes a function $f : \{0,1\}^n \mapsto \mathbb{Z}$ according to $f = f_{x_1=0} + x_1 \cdot (f_{x_1=1} - f_{x_1=0})$. $f_{x_1=0}$ and $f_{x_1=1}$ are the functions resulting from f by replacing the variable x_1 by 0 and 1, respectively. The correctness of the decomposition can be easily shown by case distinction wrt. $x_1 = 0$, $x_1 = 1$. *BMDs are graphs which basically represent the "sub-functions" $f_{x_1=0}$ and $f_{x_1=1} - f_{x_1=0}$ by nodes. The representation of Boolean polynomials without phase flipping corresponds to positive Davio decomposition: If p is the polynomial for f, then the polynomial for $f_{x_1=0}$ contains all terms which do not include x_1 (in Example 1 $x_2 + x_3 - x_2x_3$) and $x_1 \cdot (f_{x_1=1} - f_{x_1=0})$ contains all terms which do include x_1 (in Example 1 $x_1 - x_1x_2 - x_1x_3 + x_1x_2x_3 = x_1 \cdot (1 - x_2 - x_3 + x_2x_3)$). Thus, *BMDs can be seen as a *factored form* of Boolean polynomials. The *negative* Davio decomposition wrt. x_1 decomposes f according to $f = f_{x_1=1} + (1-x_1) \cdot (f_{x_1=0} - f_{x_1=1})$. By replacing $1-x_1$ with $\overline{x_1}$ it is easy to see that negative Davio decomposition corresponds to Boolean polynomials with phase flipping for x_1. In Example 1 $f_{x_1=1} = 1$, $f_{x_1=0} - f_{x_1=1} = -1 + x_2 + x_3 - x_2x_3$, and $f_{x_1=1} + \overline{x_1} \cdot (f_{x_1=0} - f_{x_1=1}) = 1 + \overline{x_1} \cdot (-1 + x_2 + x_3 - x_2x_3) = p'$.

Whereas [43] performs phase optimization as well, its optimization approach is pretty complex: It translates polynomials into K*BMDs, then uses a K*BMD minimization method [45] that changes the decomposition types (positive or negative Davio) of the variables, and finally it translates the K*BMDs back into polynomials – in the hope that size reductions in K*BMDs translate into size reductions of polynomials. As already mentioned above, our implementation of phase optimization is much simpler: It greedily optimizes the phases of variables and additionally restricts the phase optimization to variables newly introduced into the polynomial in the last rewriting step.

B. Backward Rewriting with Dynamic Order Optimization

Now we introduce our new method of backward rewriting with Dynamic Order Optimization. It can be seen as an improvement of the algorithm from [21] which was the first work to introduce dynamic rewriting in a SCA context.

Algorithm 2 Rewriting with Dynamic Order Optimization.

Input: Specification polynomial SP^{init}; Circuit CUV
Output: TRUE iff specification holds

```
1: SP_i ← SP^init;
2: A ← Detect-Atomic-Blocks(CUV);
3: B ← Compute-EABs(A, CUV);
4: for each b ∈ B do
5:     Penalty[b] ← 1;
6: C ← Get-First-Candidate-EABs(B);
7: while C is not empty do
8:     for each c ∈ C do
9:         Score[c] ← Count-Occurrence(c, SP_i) · Penalty[c];
10:    sortedC ← Sort-Candidates-Ascending(C, Score);
11:    SP_old ← SP_i;  chosen ← 0;
12:    j ← 0;  upB_fac ← 1;  best_j ← −1;
13:    while chosen = 0 do
14:        SP_i ← SP_old;
15:        (success, SP_i) ← Rewrite(SP_i, sortedC[j], upB_fac);
16:        if success = TRUE then
17:            threshold ← 0.01 × Get-Node-Count(sortedC[j]);
18:            growth ← (size(SP_i) − size(SP_old))/size(SP_old);
19:            if growth < threshold then
20:                chosen ← sortedC[j];
21:            else
22:                best_j ← Save-Best-Candidate-So-Far(SP_i, j);
23:                Penalty[sortedC[j]] ← Penalty[sortedC[j]] × 2;
24:        else
25:            Penalty[sortedC[j]] ← Penalty[sortedC[j]] × 2;
26:        j ← j + 1;
27:        if chosen = 0 and j = size(sortedC) then
28:            if best_j ≥ 0 then
29:                chosen ← sortedC[best_j];
30:                (success, SP_i) ← Rewrite(SP_i, chosen, upB_fac);
31:            else
32:                j ← 0;  upB_fac ← upB_fac × 2;
33:    C ← Update-Candidates(B, C, chosen);
34: if size(SP_i) = 0 then return TRUE else return FALSE;
```

The observation is that during backward rewriting, usually, there are several candidates (be it on the lower level of individual nodes or on a higher "component" level like atomic blocks or fan-out-free cones) to choose from for the next substitution step. The goal is always to find a substitution order which keeps intermediate polynomials small. While it is easier to find "good" *static orderings* (which are computed before the rewriting process started) for clean multiplier circuits, this task gets hard for optimized circuits where the clean boundaries between different functional blocks might vanish, e.g. due to applied logic synthesis. The idea of Dynamic Order Optimization is to take the current polynomial into account to choose good candidates for the next substitution step. We combine this idea with our new phase optimization approach to achieve a stronger dynamic ordering.

We start with a rough overview of our Dynamic Order Optimization: Basically, we use *two* dynamic procedures on different hierarchy levels to obtain a good dynamic order for backward rewriting. On the higher "component level" (see Alg. 2) we use dynamic ordering on the level of components, choosing a good candidate component in every step. In our case the components are so-called Extended Atomic Blocks

(EABs) [31]. On the lower level (see Alg. 3) we use dynamic ordering as well, but now on the level of individual circuit nodes inside the components. This is an important difference to the approach from [21] which uses a dynamic ordering approach on the component level, but a static approach for precomputing polynomials of those components (which are used later on in a global rewriting procedure).

Now we come to a more detailed description of our method: Our algorithm for backward rewriting with Dynamic Order Optimization is depicted in Algs. 2 and 3. We start in Alg. 2 with the specification polynomial SP^{init} and the circuit under verification CUV in AIG format. In the beginning we detect atomic blocks (XORs, HAs, FAs, as well as single sum and carry outputs of FAs) (see line 2). Next, we combine these atomic blocks and the remaining gates of the circuit into Extended Atomic Blocks (EABs) [31] (line 3). The idea of EABs is to combine atomic blocks and remaining gates into fanout-free cones to partition the circuit into functionally and structurally related subcircuits. While [31] uses EABs to compute don't cares on their inputs to optimize polynomials during backward rewriting and [21] uses a similar concept called "components" with the purpose of locally precomputing polynomials for those components before using them in a global rewriting procedure, we use EABs only for the purpose of helping to find good substitution orders.

For every EAB we initialize a penalty factor of 1 (line 5). Next the initial list of candidate EABs is computed (line 6). To do substitutions in reverse topological order, this list contains all EABs which only have fan-outs into primary outputs but not into any other EABs. Our method can be seen as a two-leveled, hierarchical ordering approach. The first, upper level, working on the level of EABs, is described by the while loop from lines 7 to 33. In each round one candidate EAB for substitution is picked in a dynamic fashion and the candidate set is adjusted, until all EABs and therefore the complete circuit has been substituted. In detail this works as follows: A score is assigned to each candidate which is computed as the number of occurrences of the candidate's output signals in SP_i multiplied by the individual penalty factor of the candidate (lines 8 and 9). Afterwards, the candidates are sorted by their scores in ascending order (line 10). The intuition is to prefer candidates with small numbers of output signal occurrences in the polynomial, since from a worst-case perspective many occurrences of EAB output signals in the current polynomial lead to a higher risk of a large polynomial after substitution of this EAB, see also [21], Example 6. Here this idea is adjusted by an additional penalty factor which will be explained later.

In the inner while loop from lines 13 to 32 the actual selection of the candidate EAB happens by iterating through the sorted candidates until a suitable one has been chosen. First, the old polynomial is restored. Next, the actual backward rewriting steps are performed for the current candidate (line 15). The details of how an EAB is rewritten are shown in Alg. 3 and will be explained later. At this point it is only important that rewriting may succeed or fail due to size limitations, which will be indicated by the return value $success$. If it succeeded, a polynomial is returned where all nodes from the current candidate have been rewritten. Otherwise the rewriting has been aborted. In the successful case a threshold (line 17) defines how much growth of SP_i is acceptable for the current candidate. This threshold increases with the number of AIG nodes in the candidate EAB to avoid a bias towards the selection of small EABs (whose replacement cannot increase the polynomial too much). If the growth of SP_i stays below the threshold the candidate is accepted (line 20). Otherwise it is checked if the current candidate produced the smallest polynomial size so far and is therefore saved as best candidate until now (line 22). The penalty factor of the current candidate EAB is increased, if it either has not been accepted (line 23) or the rewriting of the candidate even failed due to the size limit (line 25). The idea of a penalty factor is to avoid that the same candidates get checked unsuccessfully over and over again, because their occurrence count is small, but their substitution is very costly all the same.

In case all candidates have been checked and none of them stayed below the threshold (line 27), the best found candidate is rewritten again. This way we do not need to repeat the complete iteration as long as one candidate could have been rewritten successfully. Only in the case that none of the candidates could have been rewritten without aborting, the upper bound factor upB_fac used for the internal rewriting of Alg. 3 is increased (line 32) and the iteration starts with the same set of candidates again. By this increasing it is guaranteed that eventually some candidate can be rewritten successfully (assuming unlimited resources). After a candidate was chosen, the set of candidate blocks is adjusted (line 33): The chosen candidate is removed and new candidate EABs, which are fan-ins of the just chosen one might get included, if all of their fan-out EABs have already been substituted. At the end the algorithm returns TRUE if SP_i has been reduced to 0, meaning the specification is fulfilled by the circuit.

Next, we explain the actual rewriting process for an EAB which can be seen as the second, lower level of our hierarchical ordering approach working on the level of individual nodes inside of EABs. The algorithm is shown in Alg. 3. As inputs it takes the current polynomial SP_i, an EAB E and some factor upB_fac. An upper bound for intermediate polynomial sizes is computed (line 2) as follows: If $10 \times \text{size}(SP_i) < 100,000$ the upper bound is set to $10 \times \text{size}(SP_i)$, otherwise it is set to $\text{size}(SP_i) + 100,000$. Additionally the upper bound is multiplied by the upB_fac factor afterwards.

Before computing an order dynamically, the algorithm first tries rewriting with two predefined orders based on breadth-first-search and depth-first-search (line 3), with applying phase optimization after every node rewriting. It is history dependent (based on successful rewriting for the current EAB in the past) which order is tested first, and if the first order fails (due to exceeding the upper bound limit) the second is tried. In case one of the orders was successful, TRUE is returned together with the rewritten polynomial SP_i.

Only if none of the two predefined orders was successful,

Algorithm 3 Dynamic Rewriting of an EAB.

Input: Polynomial SP_i, EAB E, Factor upB_fac
Output: ((TRUE if successful, FALSE otherwise), Polynomial SP_i)

```
 1: SP_start ← SP_i;
 2: upperBound = Compute-Upper-Bound(size(SP_i), upB_fac);
 3: (success, SP_i) ← Try-Rewriting-With-BFS-DFS-Orders(SP_i, E);
 4: if success = FALSE then
 5:     SP_i ← SP_start;
 6:     success ← TRUE;
 7:     N ← Get-First-Candidate-Nodes(nodes(E));
 8:     while N is not empty and success = TRUE do
 9:         for each n ∈ N do
10:             Score[n] ← Count-Occurrence(n, SP_i);
11:         sortedN ← Sort-Candidates-Ascending(N, Score);
12:         SP_old ← SP_i;   chosen ← 0;   j ← 0;
13:         while chosen = 0 do
14:             SP_i ← SP_old;
15:             SP_i ← Rewrite(SP_i, sortedN[j]);
16:             SP_i ← Phase-Opt(SP_i, Get-Inputs(sortedN[j]));
17:             threshold ← 0.1;
18:             growth ← (size(SP_i) − size(SP_old))/size(SP_old);
19:             if growth < threshold then
20:                 chosen ← sortedN[j];
21:             else
22:                 best_j ← Save-Best-Candidate-So-Far(SP_i, j);
23:             j ← j + 1;
24:             if chosen = 0 and j = size(sortedN) then
25:                 chosen ← sortedN[best_j];
26:                 SP_i ← Rewrite(SP_i, chosen);
27:                 SP_i ← Phase-Opt(SP_i, Get-Inputs(chosen));
28:         N ← Update-Candidates(nodes(E), N, chosen);
29:         if size(SP_i) > upperBound then
30:             success ← false;
31:             SP_i ← SP_start;
32: return (success, SP_i)
```

dynamic ordering is started. The dynamic ordering on the node level works very similar to the dynamic ordering on EAB level of Alg. 2, thus we will keep the description short here. Again candidates, which are individual circuit nodes here, are sorted based on a scoring and we look for a candidate which can keep the relative polynomial growth below a threshold (which is a fixed value of 0.1 here) or, if such candidate does not exist, the best found so far is picked. This is repeated until all nodes have been rewritten. After each (tentative or final) node rewriting we immediately apply phase optimization to the newly introduced signals (lines 16, 27). There is one special case for nodes of the EAB being part of an atomic block (XORs, HAs, FAs, single FA outputs) that is not explicitly represented in Alg. 3: They are only rewritten if *all* output nodes of the atomic block are in the candidate set N and whenever one of the output nodes of the atomic block should be rewritten, then all the nodes in the atomic block are rewritten in a fixed precomputed order. The major difference to Alg. 2 is that after every chosen candidate node it is checked whether the current polynomial size exceeds the upper bound (line 29). If this is the case, the dynamic rewriting of this EAB E is aborted and returns FALSE together with the starting polynomial SP_{start}. Therefore, the rewriting of an EAB is not always completed. This is used in Alg. 2 to avoid investing too much space and time in the rewriting of one specific candidate EAB, while there may be other suitable EABs to choose from.

C. Simple Certifiability

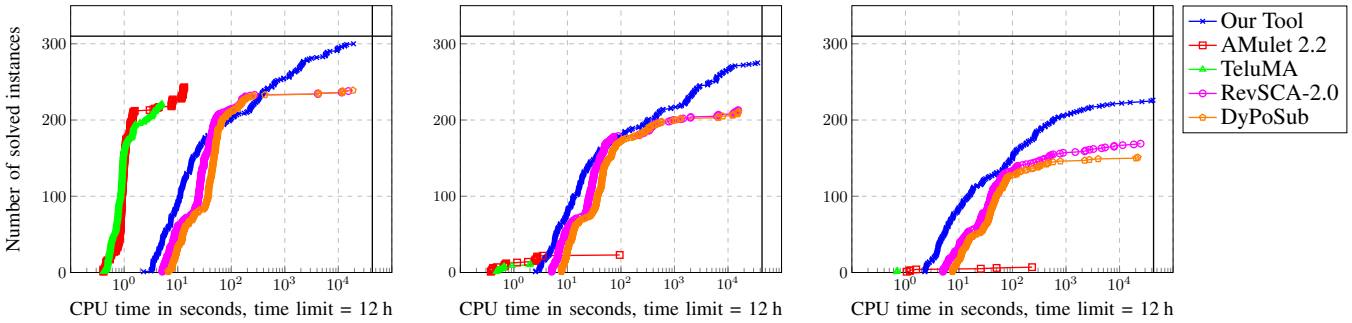
Certification is an important aspect to increase the trust in fully automatic tools. To this day, most SCA-based verification tools lack certification. Kaufmann et al. have made efforts to provide (easily checkable) certificates for different versions of their rewriting tools so far [20], [22], [25], [36]. They even showed in [36] unsoundness for an existing SCA-based tool by using fuzzing techniques. An alternative approach to guarantee the soundness of an automatic verification is to formally verify the verification tool itself. This approach is taken by Temel et al. [32], [33], [46] for the automatic verification of multipliers (which is not based on SCA, however). They verified the verification tool with the ACL2 theorem prover [47].

We want to highlight that the simplicity of our new approach facilitates certification, although our prototype tool does not yet produce certificates. Whereas the intensive search process for a good substitution order and for a good phase assignment for the intermediate polynomials may be expensive, it is easy to write out the final set of signals to virtually insert double inverters for phase optimization (see Sect. IV-A) as well as the finally computed substitution order. The substitutions are only performed at the gate level and not in a hierarchical manner and we do not use any properties derived by SAT or other techniques which would need a separate proof method. Thus, a simple dedicated and formally verified proof checker could be used whose memory requirements are limited by the memory requirements of the verifier or the certificate could be simply mapped to the existing practical algebraic calculus (PAC) format [48].

V. Experimental Results

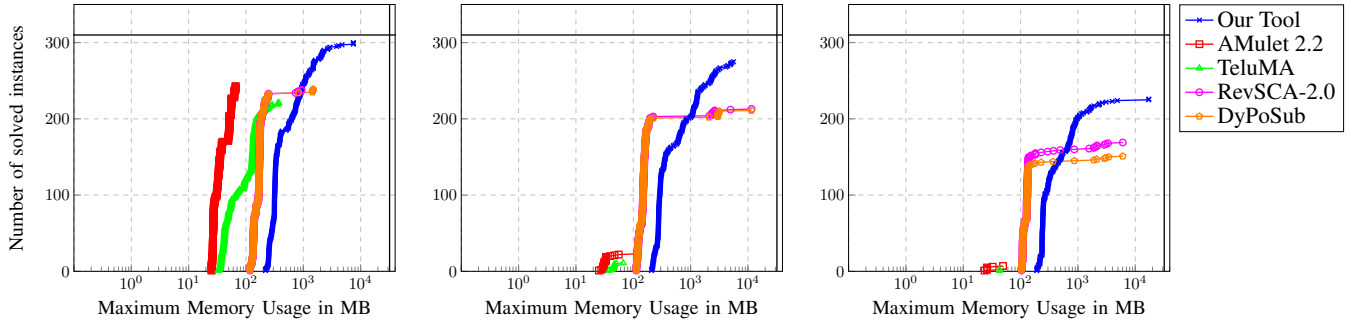
We have implemented the new method from this paper in our tool DynPhaseOrderOpt. Tests have been run on a single core of an Intel Xeon CPU E5-2650v2 with 2.60GHz. Resources were limited to 32GB main memory and 12 hours of CPU run time. For comparison we also run the following SCA-based multiplier verification tools on the same setup: AMulet 2.2 [25], [36], TeluMA [23], RevSCA-2.0 [24] and DyPoSub [21]. First experiments with the VeSCMul tool by Temel et al. [32], [33], [46] have shown that it is not suitable for flat AIG-based designs as we consider them here, since their rewriting method relies on hierarchy information and in case of flat AIG-based designs it fails even for multipliers with very small bitwidths. Therefore, the comparison with VeSCMul was omitted. The examined benchmark set contains 310 different unsigned 64-bit multiplier circuits and is composed of:

- *all* 192 64-bit unsigned multipliers from the aoki-benchmark set [49] (which unfortunately is no longer available online and therefore was obtained from the artifact data of [22])
- *all* 28 possible 64-bit unsigned multipliers obtainable from the multiplier generator GenMul [50], [51]
- *all* 90 possible 64-bit stand-alone unsigned multipliers obtainable from the multiplier generator multgen [52].



(a) No optimization. (b) Optimized with *resyn3*. (c) Optimized with *dc2*.

Fig. 3: Verification times for different tools and optimizations.



(a) No optimization. (b) Optimized with *resyn3*. (c) Optimized with *dc2*.

Fig. 4: Maximum memory usage for different tools and optimizations.

The multiplier circuits cover a wide range of different implementation options for PPGs, PPAs, and FSAs (see Sect. II-B). We provide the benchmark set, our tool and experimental data at [53].

The experimental results are shown in Tab. I. Col. 1 states the used tool. For each tool, we differentiate between five types of results (Col. 2): "Solved" means that the tool has successfully verified that the multiplier is correct within the given resource limits. "TO" indicates a time out, i.e. exceeding the time limit, while "MO" indicates a memory out, i.e. exceeding the available main memory. "SegFault" means the program was terminated by a segmentation fault and "F.Buggy" states that the multiplier circuit has been erroneously reported as buggy. For testing the robustness of the tools we consider different optimizations on the benchmark set given in Cols. 3 to 5 which are either *none* or the ABC [54], [55] commands *resyn3* and *dc2*. The numbers in Cols. 3 to 5 indicate how many results of each type a specific tool (indicated by the row) has produced for a given optimization variant (indicated by the column). In Col. 6 we sum up the results over all benchmarks. For AMULET 2.2 we first used its standard method which substitutes possible complex FSAs and verifies the modified circuits afterwards. Since AMULET 2.2 produced several segmentation faults when trying to substitute complex FSAs in *optimized* benchmarks, we have chosen the following approach: Whenever AMULET 2.2 produced a segmentation fault when trying to substitute complex FSAs, we omitted the substitution and ran the verification on the original circuit.

It can be seen that none of the tools is able to successfully verify all 310 unoptimized benchmarks (Col. 3). Our tool solves the most with 300 benchmarks, followed by AMULET 2.2 with 243, DYPOSUB with 239, REVSCA-2.0 with 238 and TELUMA with 221. The advantage of our tool can be seen even better in the results for optimized circuits. With logic optimization, we are still able to solve 275 benchmarks for *resyn3* and 226 for *dc2*. Here we also see the large deficit of AMULET 2.2 and TELUMA which can only solve up to 23 for any optimized benchmark set. While AMULET 2.2 runs into time outs for most instances, TELUMA also produces up to 143 segmentation faults for the optimized benchmarks. REVSCA-2.0 and DYPOSUB perform better on optimized benchmarks, but still lag behind our tool. They solve at least 62 benchmarks less for *resyn3* and 57 less for *dc2*. In total, our tool is able to solve 801 benchmarks while the second best tool, REVSCA-2.0, only solves 620 (Col. 6). In contrast to our tool, all comparison tools produce segmentation faults and even erroneously reported buggy results on some of the benchmark.

TABLE I: Verification results for different tools.

Tool	Result	Optimization none	resyn3	dc2	sum
Our tool DynPhase-OrderOpt	Solved	300	275	226	801
	TO	10	34	76	120
	MO	0	1	8	9
	SegFault	0	0	0	0
	F.Buggy	0	0	0	0
AMulet 2.2 [25], [36]	Solved	243	23	7	273
	TO	57	282	303	642
	MO	2	5	0	7
	SegFault	0	0	0	0
	F.Buggy	8	0	0	8
TeluMA [23]	Solved	221	11	2	234
	TO	89	219	165	473
	MO	0	1	0	1
	SegFault	0	77	143	220
	F.Buggy	0	2	0	2
RevSCA-2.0 [24]	Solved	238	213	169	620
	TO	21	20	53	94
	MO	21	70	73	164
	SegFault	12	0	0	12
	F.Buggy	18	7	15	40
DyPoSub [21]	Solved	239	211	151	601
	TO	18	23	98	139
	MO	24	69	54	147
	SegFault	12	0	0	12
	F.Buggy	17	7	7	31

More details on the results are shown in Fig. 3 and Fig. 4 where we show cactus plots for the required run times and the maximum memory usage, respectively, for all tools and all optimizations (but only for solved instances). Fig. 3 shows that AMulet 2.2 and TeluMA are excellent wrt. time efficiency. All instances that could be solved needed 233 CPU seconds or less. A similar picture emerges for memory efficiency (Fig. 4). However, this advantage is paid by a much lower robustness; other tools show significantly better results for optimized benchmarks. This can be explained by the fact that AMulet 2.2 and TeluMA are tailored towards detecting certain structural pecularities in the circuit implementations. They are very fast, if those characteristics are found in the benchmarks. If logic synthesis has destroyed those structural properties, the other tools (and in particular our tool DynPhaseOrderOpt) can demonstrate their robustness and their consistent performance for the general case.

In summary, the presented results show that our new tool DynPhaseOrderOpt is not only able to solve almost all unoptimized benchmarks within reasonable times, but it also performs better than the other tools especially on optimized benchmarks, confirming the higher overall robustness of our method.

Finally, we investigated for our tool DynPhaseOrderOpt the detailed question of whether it makes sense to try two precomputed orders based on breadth-first-search (BFS) and depth-first-search (DFS) first, before computing a dynamic order in Alg. 3 on the level of individual nodes within an EAB. The goal of this approach is to avoid time-intensive dynamic order computations for simple cases where BFS or DFS are sufficient. Tab. II clearly shows that the approach does make sense. For the table we consider all successful cases where the ordering for an EAB was not discarded later on by choosing another EAB to be processed before it on the higher "component level" of Alg. 2. We count how often BFS, DFS, and dynamic ordering was used, separately for each optimization (none, resyn3, dc2). Tab. II gives the fraction for each ordering method. It shows that most EABs are ordered by BFS. This fact may seem surprising at first sight, but can be explained by the fact that BFS is the default first order to try and it is chosen in all simple cases such as very small EABs with only a few nodes, EABs consisting of only one atomic block (like an XOR gate, an HA, or an FA which is anyway ordered according to a fixed precomputed order), or EABs which are just not very sensitive to different rewriting orders. Even though the number of dynamic orders applied on the level of individual nodes within an EAB is only up to $0.05\ \%$, it is still important to use dynamic ordering to avoid exponential blowups while rewriting the individual nodes of an EAB in cases where neither BFS nor DFS are successful, since even the occurrence of just one such EAB in the entire circuit can lead to a failed verification attempt. Moreover, note that the precomputed BFS and DFS orders are used only on the level of individual nodes, while on the higher level of ordering EABs (see Alg. 2) a dynamic approach is always used.

TABLE II: Statistic on successful orders used in Alg. 3.

Optimization	% BFS	% DFS	% Dynamic
none	97.63	2.32	0.05
resyn3	98.49	1.47	0.04
dc2	98.40	1.58	0.02

VI. Conclusions and Future Work

We have discussed the latest SCA-based approaches to fully automatic verification of multiplier circuits and presented a new, particularly simple method for this task. The new method consists of two major contributions. The first is our *Phase Optimization* algorithm which dynamically adjusts the phases of variables in occurring polynomials to reduce intermediate polynomial sizes. The second is our backward rewriting with dynamic *Order Optimization*, which uses several heuristics to create a dynamic order for backward rewriting that keeps intermediate polynomial sizes as small as possible. Our experiments show that our simpler method does not only compete well with latest tools on clean benchmarks but also adds more robustness, e.g. for the verification of optimized circuits. We believe that our dynamic approaches will be crucial for the verification of multipliers as well as other arithmetic circuits in the future.

REFERENCES

[1] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs J.*, vol. 20, no. 4, pp. 129—-135, 1995.

[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[3] J. R. Burch, "Using BDDs to verify multipliers," in *DAC*, 1991, pp. 408–412.

[4] J. P. M. Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *DATE*. IEEE Computer Society / ACM, 1999, pp. 145–149.

[5] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*. IEEE Computer Society, 2001, pp. 114–121.

[6] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC*, 1995, pp. 535–541.

[7] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits using binary moment diagrams," *STTT*, vol. 3, no. 2, pp. 137–155, 2001.

[8] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *ICCAD*, 1995, pp. 78–82.

[9] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, 2013.

[10] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV*, 2008, pp. 473–486.

[11] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[12] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.

[13] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[14] 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*, 2016, pp. 1048–1053.

[15] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.

[16] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.

[17] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.

[18] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.

[19] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[20] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.

[21] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.

[22] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits," in *TACAS*. Springer, 2021, pp. 357–364.

[23] D. Kaufmann, P. Beame, A. Biere, and J. Nordström, "Adding dual variables to algebraic reasoning for gate-level multiplier verification," in *DATE*. IEEE, 2022.

[24] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *TCAD*, vol. 41, no. 5, pp. 1573–1586, 2022.

[25] D. Kaufmann and A. Biere, "Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra," *STTT*, vol. 25, no. 2, pp. 133–144, 2023.

[26] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," in *DAC*, 2022.

[27] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, "Formal verification of integer dividers: Division by a constant," in *ISVLSI*, 2019, pp. 76–81.

[28] ——, "Functional verification of hardware dividers using algebraic model," in *VLSI-SoC*, 2019, pp. 257–262.

[29] C. Scholl and A. Konrad, "Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification," in *DAC*, 2020.

[30] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *DATE*. IEEE, 2021, pp. 1110–1115.

[31] A. Konrad, C. Scholl, A. Mahzoon, D. Große, and R. Drechsler, "Divider verification using symbolic computer algebra and delayed don't care optimization," in *FMCAD*. IEEE, 2022, pp. 1–10.

[32] M. Temel, A. Slobodová, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *CAV*, 2020, pp. 485–507.

[33] M. Temel and W. A. Hunt, "Sound and automated verification of real-world RTL multipliers," in *FMCAD*. IEEE, 2021, pp. 53–62.

[34] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.

[35] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial formal verification of multipliers," *Form Methods Syst. Des.*, vol. 22, no. 1, pp. 39–58, 2003.

[36] D. Kaufmann and A. Biere, "Fuzzing and delta debugging and-inverter graph verification tools," in *TAP@STAF*. Springer, 2022, pp. 69–88.

[37] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 01 1951.

[38] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Comp.*, vol. EC-13, pp. 14–17, 1964.

[39] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[40] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Computer*, vol. 22, no. 8, pp. 786–793, 1973.

[41] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Computer*, vol. 31, no. 3, pp. 260–264, 1982.

[42] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[43] A. A. R. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using gröbner bases," in *FMCAD*. IEEE, 2016, pp. 169–176.

[44] R. Drechsler, B. Becker, and S. Ruppertz, "K*BMDs: A new data structure for verification," in *European Design & Test Conf.* IEEE Computer Society, 1996, pp. 2–8.

[45] S. Höreth and R. Drechsler, "Dynamic minimization of word-level decision diagrams," in *DATE*. IEEE Computer Society, 1998, pp. 612–617.

[46] M. Temel, "Vescmul: Verified implementation of S-C-Rewriting for multiplier verification," in *TACAS*. Springer, 2024, pp. 340–349.

[47] W. Hunt, M. Kaufmann, J. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philos. Trans. R. Soc. A*, vol. 375, p. 20150399, 2017.

[48] D. Kaufmann, M. Fleury, and A. Biere, "The proof checkers pacheck and pastèque for the practical algebraic calculus," in *FMCAD*. IEEE, 2020, pp. 264–269.

[49] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Formal design of arithmetic circuits based on arithmetic description language," *IEICE Trans. Fundamentals*, vol. 89-A, pp. 3500–3509, 2006.

[50] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Springer International Publishing, 2021, pp. 177–191.

[51] ——, "Genmul," 2023. [Online]. Available: https://ics.jku.at/research/sca-verification/genmul/

[52] M. Temel, "Fast multplier generator multgen," 2019. [Online]. Available: https://github.com/temelmertcan/multgen

[53] A. Konrad and C. Scholl, "Benchmarks, binaries and experimental data," 2024. [Online]. Available: https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=24

[54] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[55] "ABC: A system for sequential synthesis and verification," available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2019.