Divider Verification Using Symbolic Computer Algebra and Delayed Don't Care Optimization

Alexander Konrad[1] Christoph Scholl[1] Alireza Mahzoon[2] Daniel Große[3] Rolf Drechsler[2]
[1]University of Freiburg, Germany [2]University of Bremen, Germany [3]Johannes Kepler University Linz, Austria
{konrada, scholl}@informatik.uni-freiburg.de, {mahzoon, drechsle}@informatik.uni-bremen.de, daniel.grosse@jku.at

***Abstract*—Recent methods based on Symbolic Computer Algebra (SCA) have shown great success in formal verification of multipliers and – more recently – of dividers as well. In this paper we enhance known approaches by the computation of *satisfiability don't cares for so-called Extended Atomic Blocks (EABs)* and by *Delayed Don't Care Optimization (DDCO)* for optimizing polynomials during backward rewriting. Using those novel methods we are able to extend the applicability of SCA-based methods to further divider architectures which could not be handled by previous approaches. We successfully apply the approach to the fully automatic formal verification of large dividers (with bit widths up to 512).**

I. Introduction

Arithmetic circuits are important components in processor designs as well as in special-purpose hardware for computationally intensive applications like signal processing and cryptography. At the latest since the famous Pentium bug [1] in 1994, where a subtle design error in the divider had not been detected by Intel's design validation (leading to erroneous Pentium chips brought to the market), it has been widely recognized that incomplete simulation-based approaches are not sufficient for verification and formal methods should be used to verify the correctness of arithmetic circuits. Nowadays the design of circuits containing arithmetic is not only confined to the major processor vendors, but is also done by many 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. Therefore the interest in *fully automatic formal verification* of arithmetic circuits is growing more and more.

In particular the verification of multiplier and divider circuits formed a major problem for a long time. Both BDD-based methods [2], [3] and SAT-based methods [4], [5] for multiplier and divider verification do not scale to large bit widths. Nevertheless, there has been great progress during the last few years for the automatic formal verification of gate-level multipliers. Methods based on *Symbolic Computer Algebra (SCA)* were able to verify large, structurally complex, and highly optimized multipliers. In this context, finite field multipliers [6], integer multipliers [7]–[19], and modular multipliers [20] have been considered. 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 direction of the inputs. For integer multipliers, SCA-based methods are closely related to verification methods based on word-level decision diagrams like *BMDs [21]–[23], since polynomials can be seen as "flattened" *BMDs [24]. Moreover, rewriting based approaches [25], [26] have recently shown to be able to verify complex multipliers as well as arithmetic modules with embedded multipliers at the register transfer level.

Research approaches for divider verification were lagging behind for a long time. Attempts to use Decision Diagrams for proving the correctness of an SRT divider [27] were confined to a single stage of the divider (at the gate level) [28]. Methods based on word-level model checking [29] looked into SRT division as well, but considered only a special abstract and clean sequential (i.e., non-combinatorial) divider without gate-level optimizations. Other approaches like [30], [31], or [32] looked into fixed division algorithms and used semi-automatic theorem proving with ACL2, Analytica, or Forte to prove their correctness. Nevertheless, all those efforts did not lead to a fully automated verification method suitable for gate-level dividers.

A side remark in [23] (where actually multiplier verification with *BMDs was considered) seemed to provide an idea for a fully automated method to verify integer dividers as well. Hamaguchi et al. start with a *BMD representing $Q \times D + R$ (where Q is the quotient, D the divisor, and R the remainder of the division) and use a backward construction to replace the bits of Q and R step by step by *BMDs representing the gates of the divider. The goal is to finally obtain a *BMD representation for the dividend $R^{(0)}$ which proves the correctness of the divider circuit. Unfortunately, the approach has not been successful in practice: Experimental results showed exponential blow-ups of *BMDs during the backward construction.

Recently, there have been several approaches to fully automatic divider verification that had the goal to catch up with successful approaches to multiplier verification: Among those approaches, [33] is mainly confined to division by constants and cannot handle general dividers due to a memory explosion problem. [34] works at the gate level, but assumes that hierarchy information in a restoring divider is present. Using this hierarchy information it decomposes the proof obligation $R^{(0)} = Q \times D + R$ into separate proof obligations for each level of the restoring divider. Nevertheless, the approach scales

This work was supported by the German Research Foundation (DFG) within the project VerA (SCHO 894/5-1, GR 3104/6-1 and DR 297/37-1) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_17

only to medium-sized bit widths (up to 21 as shown in the experimental results of [34]).

The approaches of [24], [35] work on the gate level as well, but they do not need any hierarchy information which may have been lost during logic optimization. They prove the correctness of non-restoring dividers by "backward rewriting" starting with the "specification polynomial" $Q \times D + R - R^{(0)}$ (similar to [23], with polynomials instead of *BMDs as internal data structure). Backward rewriting performs substitutions of gate output variables with the gates' specification polynomials in reverse topological order. They try to prove dividers to be correct by finally obtaining the 0-polynomial. The main insight of [24], [35] is the following: The backward rewriting method definitely needs "forward information propagation" to be successful, otherwise it provably fails due to exponential sizes of intermediate polynomials. Forward information propagation relies on the fact that the divider needs to work only within a range of allowed divider inputs (leading to input constraints like $0 \leq R^{(0)} < D \cdot 2^{n-1}$). [24] uses SAT-based information propagation (SBIF) of the input constraint in order to derive information on equivalent and antivalent signals, whereas [35] uses BDDs to compute satisfiability don't cares which result from the structure of the divider circuit as well as from the input constraint. (Satisfiability don't cares [36] at the inputs of a subcircuit describe value combinations which cannot be produced at those inputs by allowed assignments to primary inputs.) The don't cares are used to minimize the sizes of polynomials. In that way, exponential blowups in polynomial sizes which would occur without don't care optimization could be effectively avoided. Since polynomials are only changed for input values which do not occur in the circuit if only inputs from the allowed range are applied, the verification with don't care optimization remains correct. In [35] the computation of optimized polynomials is reduced to suitable *Integer Linear Programming* (ILP) problems.

In this paper we make two contributions to improve [24] and [35]: First, we modify the computation of don't cares leading to increased degrees of flexibility for the optimization of polynomials. Instead of computing don't cares at the inputs of "atomic blocks" like full adders, half adders etc., which were detected in the gate level netlist, we combine atomic blocks and surrounding gates into larger fanout-free cones, leading to so-called *Extended Atomic Blocks (EABs)*, prior to the don't care computation. Second, we replace local don't care optimization by *Delayed Don't Care Optimization (DDCO)*. Whereas local don't care optimization immediately optimizes polynomials wrt. a don't care cube as soon as the polynomial contains the input variables of the cube, DDCO only adds don't care terms to the polynomial, but delays the optimization until a later time. This method has two advantages: First, by looking at the polynomial later on, we can decide whether exploitation of certain don't cares is needed *at all*, and secondly, the later (delayed) optimization will take the effect of following substitutions into account and thus uses a more global view for optimization. Using those novel methods we are able to extend the applicability of SCA-based methods from [24], [35] to further optimized non-restoring dividers and restoring dividers which could not be handled by previous approaches.

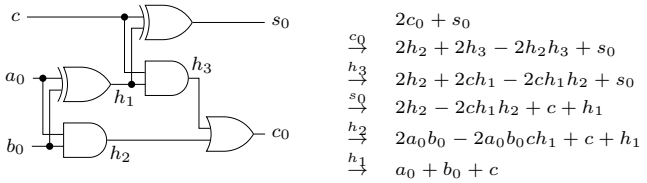

Fig. 1. Circuit with series of substitutions.

The paper is structured as follows: In Sect. II we provide background on SCA and divider circuits. We motivate the need for novel optimizations by analyzing the existing approaches in Sect. III, and in Sect. IV we present the novel approach. The approach is evaluated in Sect. V and we conclude with final remarks in Sect. VI.

II. Preliminaries

A. SCA for Verification

For the presentation of SCA we basically follow [24]. 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 [6] and truncated multipliers [17] 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 [11] or [14], 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, 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 [10]) 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 [10]) 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

Algorithm 1 Restoring division.

1: **for** $j = 1$ to n **do**
2: $\quad R^{(j)} := R^{(j-1)} - D \cdot 2^{n-j}$;
3: $\quad$ **if** $R^{(j)} < 0$ **then**
4: $\quad\quad q_{n-j} := 0$; $R^{(j)} := R^{(j)} + D \cdot 2^{n-j}$;
5: $\quad$ **else**
6: $\quad\quad q_{n-j} := 1$;
7: $R := R^{(n)}$;

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 substitution, 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 formulated as Lemma 1 in [35] and proven in [24], e.g..)

B. Divider Circuits

In the following we briefly review textbook knowledge on dividers. For more details, see [37], e.g.. We use $\langle a_n, \ldots, a_0 \rangle := \sum_{i=0}^{n} a_i 2^i$ and $[a_n, \ldots, a_0]_2 := (\sum_{i=0}^{n-1} a_i 2^i) - a_n 2^n$ for interpretations of bit vectors $(a_n, \ldots, a_0) \in \{0,1\}^{n+1}$ as unsigned binary numbers and two's complement numbers, respectively. The leading bit a_n is called the sign bit. An unsigned integer divider is a circuit with the following property:

Definition 1. *Let $(r^{(0)}_{2n-2} \ldots r^{(0)}_0)$ be the dividend with sign bit $r^{(0)}_{2n-2} = 0$ and value $R^{(0)} := \langle r^{(0)}_{2n-2} \ldots r^{(0)}_0 \rangle = [r^{(0)}_{2n-2} \ldots r^{(0)}_0]_2$, $(d_{n-1} \ldots d_0)$ be the divisor with sign bit $d_{n-1} = 0$ and value $D := \langle d_{n-1} \ldots d_0 \rangle = [d_{n-1} \ldots d_0]_2$, and let $0 \leq R^{(0)} < D \cdot 2^{n-1}$. Then $(q_{n-1} \ldots q_0)$ with value $Q = \langle q_{n-1} \ldots q_0 \rangle$ is the quotient of the division and $(r_{n-1} \ldots r_0)$ with value $R = [r_{n-1} \ldots r_0]_2$ is the remainder of the division, if $R^{(0)} = Q \cdot D + R$ (verification condition 1 = "vc1") and $0 \leq R < D$ (verification condition 2 = "vc2").*

Note that we consider here the case that the dividend has twice as many bits as the divisor (without counting sign bits). This is similar to multipliers where the number of product bits is two times the number of bits of one factor. If both the dividend and the divisor are supposed to have the same lengths, we just set $r^{(0)}_{2n-2} = \ldots = r^{(0)}_{n-1} = 0$ and require $D > 0$. Then $D > 0$ immediately implies $0 \leq R^{(0)} < D \cdot 2^{n-1}$.

The simplest algorithm to compute quotient and remainder is *restoring division* which is the "school method" to compute quotient bits and "partial remainders" $R^{(j)}$. Restoring division is shown in Alg. 1. In each step it subtracts a shifted version of D. If the result is less than 0, the corresponding quotient bit is 0 and the shifted version of D is "added back", i. e., "restored". Otherwise the quotient bit is 1 and the algorithm proceeds with the next smaller shifted version of D.

Non-restoring division optimizes restoring division by combining two steps of restoring division in case of a negative

Algorithm 2 Non-restoring division.

1: $R^{(1)} := R^{(0)} - D \cdot 2^{n-1}$;
2: **if** $R^{(1)} < 0$ **then** $q_{n-1} := 0$ **else** $q_{n-1} := 1$;
3: **for** $j = 2$ to n **do**
4: $\quad$ **if** $R^{(j-1)} \geq 0$ **then**
5: $\quad\quad R^{(j)} := R^{(j-1)} - D \cdot 2^{n-j}$
6: $\quad$ **else**
7: $\quad\quad R^{(j)} := R^{(j-1)} + D \cdot 2^{n-j}$;
8: $\quad$ **if** $R^{(j)} < 0$ **then** $q_{n-j} := 0$ **else** $q_{n-j} := 1$;
9: $R := R^{(n)} + (1 - q_0) \cdot D$;

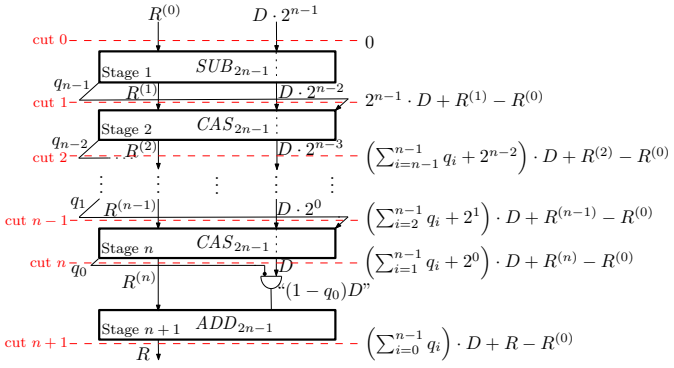

Fig. 2. Non-restoring divider.

partial remainder: adding the shifted D back and (tentatively) subtracting the next D shifted by one position less. These two steps are replaced by just adding D shifted by one position less (which obviously leads to the same result). More precisely, non-restoring division works according to Alg. 2.

SRT dividers are most closely related to non-restoring dividers, with the main differences of computing quotient bits by look-up tables (based on a constant number of partial remainder bits) and of using redundant number representations which allow to use constant-time adders. Other divider architectures like Newton and Goldschmidt dividers rely on iterative approximation. In this paper we restrict our attention to restoring and non-restoring dividers.

For dividers it is near at hand to start backward rewriting not with polynomials for the binary representations of the output words (which is basically done for multiplier verification), but with a polynomial for $Q \cdot D + R$. For a correct divider one would expect to obtain a polynomial for $R^{(0)}$ after backward rewriting. As an alternative one could also start with $Q \cdot D + R - R^{(0)}$ and one would expect that for a correct divider the result after backward rewriting is 0. This would be a proof for verification condition (vc1). (Then it remains to show that $0 \leq R < D$ (vc2) which we postpone until later.) This idea was already proposed by Hamaguchi in 1995 [23] in the context of verification using *BMDs [21]. As already mentioned in the introduction, Hamaguchi et al. observed exponential blow-ups of *BMDs in the backward construction and thus the approach did not provide an effective way for verifying large integer dividers.

However, this basic approach seems to be promising at first sight. As an example, Fig. 2 shows a *high level* view of a circuit for non-restoring division. Stage 1 implements a subtractor, stages j with $j \in \{2, ..., n\}$ implement conditional

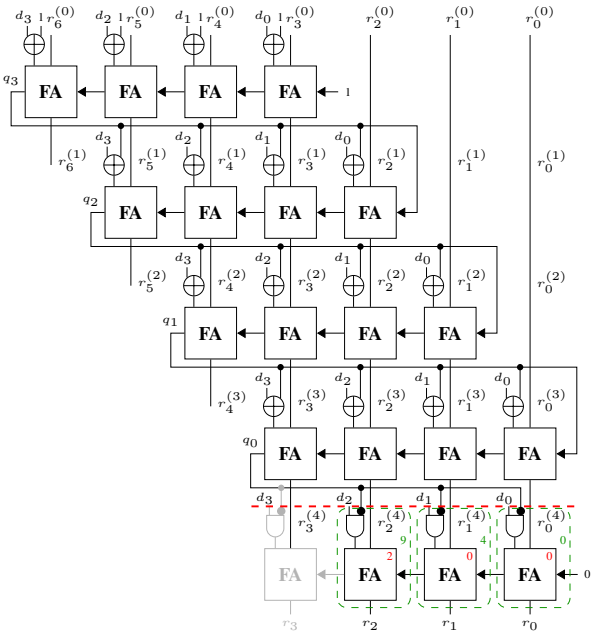

Fig. 3. Optimized non-restoring divider, $n = 4$.

adders / subtractors depending on the value of q_{n-j+1}, and stage $n+1$ implements an adder. If we start backward rewriting with the polynomial $Q \cdot D + R - R^{(0)}$ (which is quadratic in n) and if backward rewriting processes the gates in the circuit in a way that the stages shown in Fig. 2 are processed one after the other, then we would expect the following polynomials on the corresponding cuts (see also Fig. 2):

We would expect $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot D + R^{(n)} - R^{(0)}$ for the polynomial at cut n which is obtained after processing stage $n+1$, since stage $n+1$ enforces $R = R^{(n)} + (1 - q_0) \cdot D$. For $j = n$ to 2 we would (by induction) expect $(\sum_{i=n-j+2}^{n-1} q_i 2^i + 2^{n-j+1}) \cdot D + R^{(j-1)} - R^{(0)}$ for the polynomial at cut $j-1$ after processing stage j, since stage j enforces $R^{(j)} = R^{(j-1)} - q_{n-j+1}(D \cdot 2^{n-j}) + (1 - q_{n-j+1})(D \cdot 2^{n-j}) = R^{(j-1)} + (1 - 2q_{n-j+1})(D \cdot 2^{n-j})$. Finally, the polynomial at cut 0 after processing stage 1 using the equation $R^{(1)} = R^{(0)} - D \cdot 2^{n-1}$ would reduce to 0.

There may be two obvious reasons why backward rewriting might fail in practice all the same: (1) It could be the case that backward rewriting does not exactly hit the boundaries between the stages of the divider. (2) There may be significant peaks in polynomial sizes in between the mentioned cuts.

[24] and [35] show that there are additional obstacles apart from those obvious potential problems: In fact, with usual optimizations in implementations of non-restoring dividers the polynomials represented at the cuts between stages are different from this high-level derivation. The reason lies in the fact that the stages do not really implement signed addition / subtraction. In general, signed addition / subtraction of two $(2n-1)$-bit numbers leads to a $2n$-bit number. The leading bit of the result can only be omitted, if "no overflow occurs". The fact that no overflow occurs results from the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ of the divider and from the way the results of the different stages are computed [24]. Usual implementations even go one step further: By additional arguments using the input constraint and the circuit functionality it can be shown that it is not only possible to omit overflow bits of the adder / subtractor stages, but it is even possible to omit the computation of one further most significant bit. For a detailed analysis see [35]. These considerations lead to an optimized implementation shown in Fig. 3 for $n = 4$, e.g.. (For simplicity, we present the circuit before propagation of constants which is done however in the real implemented circuit.) In summary, it is important to note that (1) the stages in Fig. 3 cannot be seen as real adder / subtractor stages as shown in the high-level view from Fig. 2, (2) backward rewriting leads to polynomials at the cuts which are different from the ones shown in Fig. 2, and (3) unfortunately those polynomials have (provably) exponential sizes.

The conclusion drawn in [35] was that verification of (large) dividers using backward rewriting is infeasible, if there is no means to make use of "forward information" obtained by propagating the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ in forward direction through the circuit. This idea indeed made it possible to verify large non-restoring dividers with bit widths up to 512 bits.

III. Analysis of Existing Approach

In this section we motivate our approach by analyzing weaknesses of the method from [35]. The algorithm from [35] starts with a gate level netlist and detects atomic blocks [16] like full adders and half adders. This results in a circuit with non-trivial atomic blocks (full adders, half adders etc.) and trivial atomic blocks (original gates not included in non-trivial atomic blocks). The method computes a topological order $\prec_{top}$ on the atomic blocks with heuristics from [15], [16], computes satisfiability don't cares [36] at the inputs of the atomic blocks, and performs backward rewriting starting with the specification polynomial $Q \cdot D + R - R^{(0)}$ by replacing atomic blocks in reverse topological order. During backward rewriting two optimization methods are used, if they are needed to keep polynomial sizes small: The first method uses information on equivalent and antivalent signals (which is derived by SAT-based information propagation (SBIF) using the input constraint and the don't cares at the inputs of atomic blocks), the second method optimizes polynomials modulo don't cares by reducing the problem to Integer Linear Programming (ILP).

A. Insufficient don't care conditions

Let us start by considering stage $n+1$ of the non-restoring divider (see Figs. 2 and 3). Analyzing the method from [35] applied to optimized n-bit non-restoring dividers, we can observe that it does not make use of don't cares at the inputs of atomic blocks corresponding to stage $n+1$ (although there exist some don't cares), but it makes use of the (only existing) antivalence of q_0 and $r_{n-1}^{(n)}$ which is shown by SAT taking already proven satisfiability don't cares into account (as already described above). If we only consider the circuit of stage $n+1$ (i.e., the circuit below the dashed

line in Fig. 3), replace $r_{n-1}^{(n)}$ by $\neg q_0$ (i.e. if we make use of the mentioned antivalence), and start backward rewriting with $(\sum_{i=0}^{n-1} q_i 2^i) \cdot (\sum_{i=0}^{n-1} d_i 2^i) + (\sum_{i=0}^{n-2} r_i 2^i - r_{n-1} 2^{n-1}) - (\sum_{i=0}^{2n-2} r_i^{(0)} 2^i)$, then we indeed obtain exactly the polynomial $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot (\sum_{i=0}^{n-1} d_i 2^i) + (\sum_{i=0}^{n-2} r_i^{(n)} 2^i - (1 - q_0) 2^{n-1} - (\sum_{i=0}^{2n-2} r_i^{(0)} 2^i)$ which corresponds (with $(1-q_0) = r_{n-1}^{(n)}$) to $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot D + R^{(n)} - R^{(0)}$ as shown in Fig. 2, cut n. Fig. 4 shows the size of the final polynomial for stage $n+1$ with increasing bit width n, with and without using the antivalence $r_{n-1}^{(n)} = \neg q_0$. Fig. 4 clearly shows that it is essential to make use of the mentioned antivalence.

Now we consider another version of the non-restoring divider which is slightly further optimized. It is clear that in a correct divider the final remainder is non-negative, i.e, $r_{n-1} = 0$. Therefore there is actually no need to compute r_{n-1} and the full adder shown in gray in Fig. 3 can be omitted. The verification condition vc1 is then replaced by $R^{(0)} = Q \cdot D + \sum_{i=0}^{n-2} r_i 2^i$. Whereas in the original circuit making use of antivalences was essential for keeping the polynomial sizes small, in stage $n+1$ of the further optimized version there are neither equivalent nor antivalent signals anymore. The only don't cares in the last stage (after constant propagation) are two value combinations at the inputs of the now leading full adder. However, making use of those don't cares does not help in avoiding an exponential blow up as Fig. 5 shows. Intuitively it is not really surprising that removing the full adder shown in gray potentially makes the verification problem harder, since the partial remainders $R, R^{(n)}, \ldots, R^{(1)}$ in the high-level analysis of polynomials at cuts (see Fig. 2) represent signed numbers, but now R does not introduce a sign bit anymore.

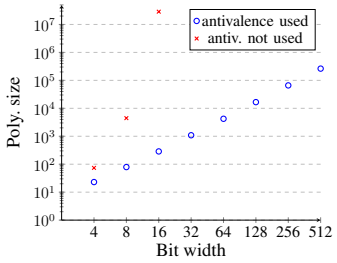

Fig. 4. Polynomial sizes, stage $n+1$, optimized non-restoring divider.

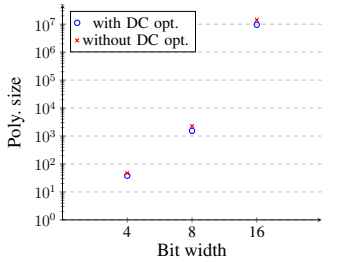

Fig. 5. Polynomial sizes, stage $n+1$, further optimized non-restoring divider.

Nevertheless, this raises the question whether the derivation of don't care conditions may be improved in a way that don't care optimization can avoid exponential blow ups like the one shown in Fig. 5.

B. Don't care optimization with backtracking

The method from [35] does not make use of don't care optimizations immediately, but stores a backtrack point after backward rewriting was applied to an atomic block which has don't cares at its inputs or has input signals with equivalent / antivalent signals. Whenever the polynomial grows too much, the method backtracks to a previously stored backtrack point and performs an optimization. Alg. 3 shows a simplified

Algorithm 3 Backward rewriting with backtracking.

Input: Specification polynomial SP^{init}, Input constraint IC, Circuit CUV with atomic blocks $a_1 \prec_{top} \ldots \prec_{top} a_m$ in topological order $\prec_{top}$
Output: 1 iff specification holds for all inputs satisfying IC

```
1:  SP_m := SP^init; oldsize := size(SP_m); i := m; ST := ∅;
2:  (dc(a_1), ..., dc(a_m)) := Compute_DC(CUV, IC);
3:  while i > 0 do
4:      SP_{i-1} := Rewrite(SP_i, a_i);
5:      if size(SP_{i-1}) > threshold · oldsize and ST ≠ ∅ then
6:          (SP, j) = pop(ST);
7:          i := j; SP_{i-1} := SP;
8:          SP_{i-1} := Opt_DC(SP_{i-1}, dc(a_i));
9:      else
10:         if dc(a_i) ≠ ∅ then push(ST, (SP_{i-1}, i)); oldsize := size(SP_{i-1});
11:     i := i − 1;
12: return evaluate(SP_0);
```

overview of the approach.* For ease of exposition we omitted handling of equivalences / antivalences here.

As shown in [35], the approach works surprisingly well. It tries to restrict don't care optimizations (which are illustrated later on in Example 1, for more details see [35]) to situations where they are really needed. Only if the size threshold in line 5 is exceeded, backtracking is used and don't care optimization comes into play. A further analysis shows that the success of the approach in [35] is partly due to the following reasons: (1) In the non-restoring dividers used as benchmarks, atomic blocks that have any satisfiability don't cares grow only linearly with the bit width. (2) Only a linear amount of backtrackings is needed. (3) On the other hand, if backtrackings have to be used, don't care assignments have an essential effect in keeping the polynomials small (the size of the polynomials is quadratic in n just like the specification polynomial we start with).

Let us now consider a very simple example which does *not* have the mentioned characteristics.

Example 1. *Consider a circuit which contains (among others) $2n+1$ atomic blocks $a_0, \ldots a_{2n}$. Those blocks are the last atomic blocks in the topological order and $a_{2n} \prec_{top} \ldots \prec_{top} a_0$. The initial polynomial is $SP^{init} = 8a + 4b + 2c + i_0$. a_0 has inputs x_1, i_1, output i_0, defines the function $i_0 = x_1 \vee i_1 = x_1 + i_1 - x_1 i_1$, and we assume that it has the satisfiability don't care $(x_1, i_1) = (0,0)$. Correspondingly, for $j = 1, \ldots, n$, a_j defines $i_j = x_{j+1} i_{j+1}$ with assumed satisfiability don't care $(x_{j+1}, i_{j+1}) = (0,0)$, and for $j = n+1, \ldots, 2n$, a_j defines $i_j = x_{j+1} \vee i_{j+1} = x_{j+1} + i_{j+1} - x_{j+1} i_{j+1}$. We compute size$(p)$ as the number of terms in the polynomial p and assume threshold $= 1.5$ in line 5 of Alg. 3. Then Alg. 3 computes the following series of polynomials*

$$
\begin{aligned}
SP_m &= 8a + 4b + 2c + i_0 \\
SP_{m-1} &= 8a + 4b + 2c + x_1 + i_1 - x_1 i_1 \\
SP_{m-2} &= 8a + 4b + 2c + x_1 + x_2 i_2 - x_1 x_2 i_2 \\
&\ldots
\end{aligned}
$$

*SP_0 in Alg. 3 does not have to be 0 for correct dividers, it is sufficient that SP_0 evaluates to 0 for all inputs in the allowed input range $0 \leq R^{(0)} < D \cdot 2^{n-1}$. This can be checked by evaluate(SP_0) in polynomial time [35].

$$SP_{m-n-1} = 8a + 4b + 2c + x_1 + x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1}$$
$$SP_{m-n-2} = 8a + 4b + 2c + x_1 + x_2 \dots x_{n+2} + x_2 \dots x_{n+1} i_{n+2} - x_2 \dots x_{n+2} i_{n+2} - x_1 \dots x_{n+2} - x_1 \dots x_{n+1} i_{n+2} + x_1 \dots x_{n+2} i_{n+2}$$

with sizes 4, 6, ..., 6, 10. SP_{m-n-2} is the first polynomial exceeding the size limit. For each of the $n+1$ preceding atomic blocks there was a satisfiability don't care at the inputs, the size limit was not exceeded, and the corresponding polynomial has been pushed to the backtracking stack ST. Now backtracking to SP_{m-n-1} takes place. (Note that it is easy to see that without backtracking using don't care optimization the following $n-1$ backwriting steps would quickly lead to a blowup in the polynomial sizes finally resulting in a polynomial with size $2^{n+2}+2$.) SP_{m-n-1} is optimized with the don't care $(x_{n+1}, i_{n+1}) = (0,0)$. Let us explain the idea of don't care optimization using this example: Don't care optimization adds $v \cdot (1-x_{n+1}) \cdot (1-i_{n+1})$ for the don't care $(x_{n+1}, i_{n+1}) = (0,0)$ to SP_{m-n-1} with a fresh integer variable v. For all valuations $(x_{n+1}, i_{n+1}) \neq (0,0)$, $v \cdot (1-x_{n+1}) \cdot (1-i_{n+1})$ evaluates to 0, thus we may choose an arbitrary integer value for v without changing the polynomial "inside the care space". The choice of v is made such that the size of SP_{m-n-1} is minimized. So the task is to choose v such that the size of $8a+4b+2c+x_1+x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1} + v - v i_{n+1} - v x_{n+1} + v x_{n+1} i_{n+1}$ is minimal. We achieve this by using an ILP solver to get a solution for v which maximizes the number of terms with coefficients 0 and therefore minimizes the polynomial. It is easy to see that the best choice is $v = 0$ in this case. This means that we arrive at an unchanged polynomial SP_{m-n-1} and the don't care did not help. Then we do the replacement of a_{n+1} again, detect an exceeded size limit again, backtrack to SP_{m-n} and so on. Exactly as for SP_{m-n-1}, don't care assignment does not help for $SP_{m-n}, \dots, SP_{m-2}$. The first really interesting case occurs when backtracking arrives at SP_{m-1}. Adding $v \cdot (1-x_1) \cdot (1-i_1)$ with a fresh variable v to SP_{m-1} results in $8a+4b+2c+v+(1-v)x_1+(1-v)i_1+(v-1)x_1 i_1$ and choosing $v = 1$ leads to the minimal polynomial $8a+4b+2c+1$ which is even independent from i_1. Now replacing $a_1, \dots, a_{2n}$ does not change the polynomial anymore and we finally arrive at $SP_{m-2n-1} = 8a+4b+2c+1$ (without further don't care assignments).

The example shows that the backtracking method works in principle, but it comes at huge costs: Backtracking potentially explores all possible combinations of assigning or not assigning don't cares for atomic blocks with don't cares by storing backtrack points again in line 10 of Alg.3 after successful as well as unsuccessful don't care optimizations. In the example this leads to 2^{n+1} rewritings for atomic blocks and $2^{n+1}-1$ unsuccessful don't care optimizations, before we finally backtrack to SP_{m-1} where we do the relevant don't care optimization.

Our goal is to come up with a don't care optimization method which is robust against situations like the one illustrated in Example 1 where we have many blocks with don't cares, but only a few of those don't cares are really useful for minimizing the sizes of polynomials. As we will show in Sect. V, we run into such situations when we verify restoring dividers using the method from [35].

Algorithm 4 Computation of satisfiability don't cares.

Input: Input constraint IC, Circuit CUV with EABs $ea_1 \prec_{top} \dots \prec_{top} ea_l$ in topological order $\prec_{top}$, $dc\_cand(ea_j) \forall j \in \{1, \dots, l\}$
Output: Satisfiability don't cares at inputs of EABs resulting from IC

```
1: I = {j ∈ {1,...,l} | dc_cand(ea_j) ≠ ∅}; i_old = 1; χ = IC;
2: dc(ea_1) = ∅; ...; dc(ea_l) = ∅;
3: while I ≠ ∅ do
4:     i = min(I); slice = {ea_{i_old}, ..., ea_{i-1}};
5:     χ = compute_image(χ, slice);
6:     for (ε_1, ..., ε_n) ∈ dc_cand(ea_i) do ▷ x_1, ..., x_n: input signals of ea_i
7:         if χ|_{x_1=ε_1,...,x_n=ε_n} = 0 then dc(ea_i) = dc(ea_i) ∪ {(ε_1, ..., ε_n)};
8:     I = I \ {i}; i_old = i;
9: return (dc(ea_1), ..., dc(ea_l));
```

IV. Don't Care Computation and Optimization

A. Don't care computation for extended atomic blocks

This section is motivated by [8], [11] which combine several gates and atomic blocks into fanout-free cones, compute polynomials for the fanout-free cones first and use those precomputed polynomials for "macro-gates" formed by the fanout-free cones during backward rewriting. Whereas in [8], [11] the purpose of forming those fanout-free cones is avoiding peaks in polynomial sizes during backward rewriting without don't care optimization, the motivation here is different: Here we aim at detecting more and better don't cares.

First of all, we detect atomic blocks for fixed known functions like full adders and half adders as already mentioned in Sect. III. The result is a circuit with non-trivial atomic blocks and the remaining gates. Now we want to combine those atomic blocks and remaining gates into "*extended atomic blocks (EABs)*" which are fanout-free cones of atomic blocks and remaining gates. To do so, we compute a directed graph $G = (V, E)$ where the nodes correspond to the non-trivial atomic blocks, the remaining gates, and the outputs. There is an edge from a node v to a node w iff there is an output of the atomic block / gate corresponding to v which is connected to an input of the atomic block / gate / output node corresponding to w. We compute the coarsest partition $\{P_1, \dots, P_l\}$ of V such that for all sets P_i and all $v \in P_i$ with more than one successor it holds that all successors of v are not in P_i. We combine all gates / atomic blocks in P_i into an EAB ea_i.

The computation of satisfiability don't cares at the inputs of EABs that result from the input constraint IC (for dividers according to Def. 1 $IC = 0 \leq R^{(0)} < D \cdot 2^{n-1}$) is performed for EABs as described in [35] for atomic blocks. First of all, an intensive simulation (taking IC into account) excludes candidates for satisfiability don't cares. Value combinations at inputs of EABs that are seen in the simulation are excluded, finally resulting in a set $dc\_cand(ea_j)$ for each EAB ea_j. Satisfiability don't cares at inputs of EABs are then computed by a series of BDD-based image computations [38] as shown

in Alg. 4, starting with IC. In the end we have classified all don't care candidates to be real don't cares or not.[†]

If we apply the method to the optimized divider in Fig. 3, the EABs below the dashed line are shown by dashed boxes. The number of satisfiability don't cares at the inputs of the dashed boxes (*after* constant propagation!) are shown at the right sides of the boxes just above the full adders. For the first EAB, the number of don't cares is 9, e.g., whereas for the atomic block (full adder) included in the EAB the number is only 2. At first sight, it is not clear that more don't cares really help during don't care based optimization, but we will show in Sect. V that this is definitely the case and that the use of *extended* atomic blocks is essential for a successful verification of large dividers.

B. Delayed Don't Care Optimization

In this section we introduce *Delayed Don't Care Optimization (DDCO)*. DDCO is based on the observation that don't care optimization as introduced in [35] is a *local* optimization that does not take its global effects into account. If backtracking goes back to a backtrack point with don't cares, then it backtracks to a situation where backward rewriting for an (extended) atomic block with don't cares at its inputs has taken place and the inputs of this block have been brought into the polynomial. The optimization locally minimizes the size of the polynomial using those don't cares immediately and the results of the optimization do not depend on rewriting steps which take place in the future. However, it is obvious that the future sizes of polynomials depend on the future substitutions during backward rewriting and therefore a local don't care optimization may go into the wrong direction. For that reason we propose a *delayed* don't care optimization taking future steps into account, which are performed after rewriting of the block for which the don't cares are defined. Before we will introduce DDCO, we illustrate the effect by an example.

Example 2. *Consider the polynomial*

$$p = x_1x_4x_5x_6 + x_2x_4x_5x_6 + x_3x_4x_5x_6 - x_1x_2x_4x_5x_6 - x_1x_3x_4x_5x_6 - x_2x_3x_4x_5x_6 + x_1x_2x_3x_4x_5x_6$$

with size 7. Assume that the valuation $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ *is a don't care. By using the don't care optimization method from [35] which was already illustrated in Example 1, we arrive at a polynomial*

$$q = p + vx_4x_5 - vx_1x_4x_5 - vx_2x_4x_5 - vx_3x_4x_5 + vx_1x_2x_4x_5 + vx_1x_3x_4x_5 + vx_2x_3x_4x_5 - vx_1x_2x_3x_4x_5$$

with a new integer variable v. *Since there is no pair of terms in* q *with the same monomials,* $v = 0$ *leads to the polynomial with the smallest number of terms. For all* $v \neq 0$ q *has the size 15 instead of 7. This shows that a local don't care optimization with don't care* $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ *does not help in this example. Now assume that we perform a replacement of* x_6 *by* $x_4 \cdot x_5$ *in the polynomial* q, *resulting in*

$$q' = vx_4x_5 + (1-v)x_1x_4x_5 + (1-v)x_2x_4x_5 + (1-v)x_3x_4x_5 + (v-1)x_1x_2x_4x_5 + (v-1)x_1x_3x_4x_5 + (v-1)x_2x_3x_4x_5 + (1-v)x_1x_2x_3x_4x_5$$

Here it is easy to see that choosing $v = 1$ *reduces* q' *to* $q' = x_4x_5$. *I.e., performing local don't care optimization* before *rewriting with* $x_6 = x_4 \cdot x_5$ *did not help and leads to a polynomial with 7 terms after the rewriting step, but don't care optimization* after *the rewriting step reduces the polynomial to a single term. By generalizing the example from 6 to an arbitrary number of* n *variables, we obtain* $2^{n-3} - 1$ *terms with don't care optimization before rewriting and one term with don't care optimization after rewriting, which shows that* delayed *don't care optimization can be exponentially better than local don't care optimization (even for a delay by one step only).*

Algorithm 5 Rewriting with DDCO.

Input: Specification polynomial SP^{init}; Input constraint IC; Circuit CUV with EABs $ea_1 \prec_{top} \ldots \prec_{top} ea_m$ in topological order $\prec_{top}$; EABs ea_i with input signals $x_1^{(i)}, \ldots, x_{n_i}^{(i)}$; don't cares $dc(ea_i) = \{(\varepsilon_{1,1}^{(i)}, \ldots, \varepsilon_{1,n_i}^{(i)}), \ldots, (\varepsilon_{l_i,1}^{(i)}, \ldots, \varepsilon_{l_i,n_i}^{(i)})\}$; "delay" d
Output: 1 iff specification holds for all inputs satisfying IC

1: $SP_m := SP^{init}$; $i := m + 1$;
2: **while** $i - 1 > 0$ **do**
3: $\quad i := i - 1$;
4: $\quad SP_{i-1}$:= Rewrite(SP_i, ea_i);
5: $\quad$ **for** $j = 1$ to l_i **do**
6: $\quad\quad SP_{i-1} := SP_{i-1} + v_j^{(i)} \cdot \prod_{\varepsilon_{j,k}^{(i)}=1} x_k^{(i)} \cdot \prod_{\varepsilon_{j,k}^{(i)}=0} (1 - x_k^{(i)})$;
7: $\quad$ **if** $i + d > m$ **then continue**;
8: $\quad SP_{i-1}^{tmp} := assign\_dc(SP_{i-1}, v_1^{(i+d-1)} = 0, \ldots, v_{l_i}^{(i)} = 0)$;
9: $\quad dc0\_size :=$ size$(assign_dc(SP_{i-1}^{tmp}, v_1^{(i+d)} = 0, \ldots, v_{l_{i+d}}^{(i+d)} = 0))$;
10: $\quad$ **if** $dc0_size \leq increase($size$(SP_{i+d}))$ **then**
11: $\quad\quad$ **for** $j = i - 1$ to $i + d - 1$ **do**
12: $\quad\quad\quad SP_j := assign\_dc(SP_j, v_1^{(i+d)} = 0, \ldots, v_{l_{i+d}}^{(i+d)} = 0)$;
13: $\quad$ **else**
14: $\quad\quad (z_1^{i+d}, \ldots, z_{l_{i+d}}^{i+d}) := DC\_opt(SP_{i-1}^{tmp})$;
15: $\quad\quad$ **for** $j = i - 1$ to $i + d - 1$ **do**
16: $\quad\quad\quad SP_j := assign\_dc(SP_j, v_1^{(i+d)} = z_1^{i+d}, \ldots, v_{l_{i+d}}^{(i+d)} = z_{l_{i+d}}^{i+d})$;
17: $SP_0 := assign\_dc(SP_0, v_1^{(d)} = 0, \ldots, v_{l_1}^{(1)} = 0)$;
18: **return** evaluate(SP_0);

Alg. 5 shows an integration of DDCO into backward rewriting. In contrast to Alg. 3, it does not use backtracking and it always "delays" don't care optimization by d EAB rewriting steps. In the while loop from lines 2 to 16, don't care terms with fresh integer variables $v_j^{(i)}$ are immediately added to the polynomial SP_{i-1} for each don't care of the current EAB ea_i (line 6), but those don't cares may only be used with a delay of d EAB rewritings, i.e., in the iteration replacing ea_i only don't cares coming from ea_{i+d} may be used. Therefore, younger don't care variables are temporarily assigned to 0 in line 8, leading to a polynomial SP_{i-1}^{tmp}. Now the size of SP_{i+d} (which is the polynomial before rewriting with ea_{i+d}) is compared to the size $dc0\_size$ of SP_{i-1} where the don't care variables from ea_{i+d} are assigned to 0 as well (i.e., they are not used). If $dc0\_size$ did not increase too much compared to the size of SP_{i+d} ("too much" is specified by a monotonically increasing

[†]It is easy to see that the don't care computation from Alg. 4 can be extended to a verification of vc2 (similar to [35]) just by adding a final step computing the image χ at the outputs. This way we obtain the image of the input constraint produced by the whole circuit. Then it has only to be checked whether χ implies $0 \leq R < D$.

function $increase$), then the don't care variables from ea_{i+d} are permanently assigned to 0 (lines 11 and 12) in the current as well as all previous polynomials containing those variables. Otherwise, the known ILP based don't care optimization is used and its results are inserted into SP_{i-1} and again also in all previous polynomials containing the don't care variables from ea_{i+d} (lines 14 to 16).

V. Experimental Results

Our experiments have been carried out on one core of an Intel Xeon CPU E5-2643 with 3.3 GHz and 62 GiB of main memory. The run time of all experiments was limited to 24 CPU hours. All run times in Tables I, II and III are given in CPU seconds. We used the ILP solver Gurobi [39] for solving the ILP problems for don't care optimization of polynomials. For image computations we used the BDD package CUDD 3.0.0 [40]. For benchmarks and binaries see [41].

In our experiments we consider verification of three different types of divider benchmarks with different bit widths (Cols. 1 in Tabs. I to III). Tab. I shows results for non-restoring dividers "non-restoring$_1$" as seen in Fig. 3 (with the gray full adder included), which were also used in [35]. Table II contains results for further optimized non-restoring dividers "non-restoring$_2$" that omit the gray full adder shown in Fig. 3. Table III gives results for restoring dividers. All three tables share the same column labels. Note that we did not make use of any hierarchy information during verification, but only used the flat gate-level netlist (numbers of gates are shown in Cols. 2) and employed heuristics for detecting atomic blocks as well as for finding good substitution orders [15], [16].

We begin with three experiments for comparison where we check the equivalence of the divider circuits with a "golden specification". In those experiments we restrict counterexamples to the allowed range $0 \leq R^{(0)} < D \cdot 2^{n-1}$ of inputs.

In the first experiment we used a SAT-solver (MiniSat 2.2.0 [42]) to solve the corresponding satisfiability problems. The results from Cols. 3 in Tabs. I, II, and III show that SAT-solving is hard for non-trivial arithmetic circuits and none of the benchmarks with bit widths larger than 8 could be solved in the specified time limit. In the second experiment we considered the combinational equivalence checking (CEC) approach of ABC [43], [44]. Since it is based on And-Inverter-Graph (AIG) rewriting via structural hashing, simulation, and SAT, the equivalence checking between two designs is reduced to finding equivalent internal AIG nodes. As for SAT-solving, ABC cannot verify the dividers with bit widths larger than 8, see Cols. 4 in Tabs. I, II, and III. In a third experiment we used a commercial verification tool. As Cols. 5 in Tabs. I, II, and III show, the commercial tool is able to verify also 16-bit dividers, for the restoring dividers it even verifies the 32-bit divider in about 15 CPU hours, but does not finish within the time limit for larger dividers.

From Col. 6 in Tab. I we can see that the method from [35] performs very well for the verification of the non-restoring$_1$ dividers. Col. 7 ("#bt") shows how many backtrack operations were actually performed. For the non-restoring$_2$ benchmarks considered in Tab. II the method exceeds the available memory for 16 bits and larger, for the restoring ones from Tab. III even already for 8 bits. As already shown by our analysis from Sect. III (see Fig. 5), equivalence/antivalence computation and don't care optimizations on atomic blocks as used in [35] are not strong enough to avoid exponential blowups of polynomials for the non-restoring$_2$ dividers. For restoring dividers the situation is similar.

In the next experiment we evaluate our new approach of using EABs for don't care computation instead of atomic blocks as used in [35] (at first without DDCO). For non-restoring$_1$ dividers (where the method from [35] already performed very well) this approach is somewhat slower than the original method, see Cols. 6 and 8 of Tab. I. The reason for this is that using EABs instead of atomic blocks as in [35] leads to more blocks where don't cares are applicable whereas the number of don't care optimizations which are really necessary stays the same. This can be seen in Cols. 7 and 9 of Tab. I which compare the number of performed backtracks. The version with EABs performs additional backtracks to backtrack points where optimization does not help and it has to store a larger amount of backtrack points. This even leads to running out of available memory for the 512-bit instance of non-restoring$_1$. But on the other hand already the usage of EABs enables to verify the non-restoring$_2$ dividers from Table II up to 256 bits in about 2 hours. Since don't care optimizations on atomic blocks as used in [35] are not strong enough to avoid exponential blowups for the non-restoring$_2$ dividers (as already mentioned above), using EABs is inevitable. However, the approach is not able to verify restoring dividers with bit widths larger than 64, see Col. 8 in Table III, due to increasing run times and memory consumption. This can be explained by the larger number of EABs with non-empty don't care sets for restoring dividers compared to non-restoring dividers. These numbers are given in Cols. 10 ("#EABs with DCs") of Tabs. I and II for the non-restoring dividers and in Col. 10 of Tab. III for restoring dividers. The numbers grow only linearly for non-restoring dividers, but quadratically for restoring dividers. More EABs with non-empty don't care sets lead to an increased memory consumption by storing more backtrack points and to increased run times consumed by extensive backtracking. The effect occurring here has already been illustrated in Example 1 of Sect. III-B where we have to perform an exponential amount of unsuccessful backtracks before finally arriving at the relevant don't care optimization. For the 64-bit non-restoring$_2$ divider, e.g., the approach needs less than 50 seconds with 205 backtracks (Cols. 8, 9 of Tab. II) whereas the corresponding restoring divider only finishes in about 15 minutes with 3047 backtracks (Cols. 8, 9 of Tab. III).

Cols. 12 of Tabs. I, II, and III show that those difficulties can be overcome by using our novel DDCO method. It turned out that already the simplest possible parameter choice of $d = 1$ and $increase(size) = size+1$ in Alg. 5 is successful. We were even able to verify the 256-bit restoring divider in less than 9.5 CPU hours and both 512-bit instances of non-restoring$_1$ and non-restoring$_2$ could be verified in about 7.5 hours. Comparing

TABLE I
VERIFYING DIVIDERS NON-RESTORING$_1$ FROM [35], TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35] time	[35] #bt	[35]+EABs time	[35]+EABs #bt	Our method = [35]+EABs+DDCO #EABs with DCs	#DC opt.	time	peak poly.
4	100	0.22	0.01	1.23	0.15	7	0.44	12	12	5	0.23	128
8	404	68.58	17.65	1.33	0.39	11	1.21	37	28	9	0.94	199
16	1,588	TO	TO	165.87	1.59	19	3.26	83	60	17	1.87	407
32	6,260	TO	TO	TO	5.06	35	12.10	166	124	33	6.78	1,207
64	24,820	TO	TO	TO	21.88	67	96.15	365	252	65	28.24	4,343
128	98,804	TO	TO	TO	114.73	131	1,434.11	909	508	129	153.71	16,759
256	394,228	TO	TO	TO	825.11	259	13,656.97	2,077	1,020	257	1,985.05	66,167
512	1,574,900	TO	TO	TO	9,183.28	515	MO	-	2,044	513	27,370.60	263,287

TABLE II
VERIFYING DIVIDERS NON-RESTORING$_2$, TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35] time	[35] #bt	[35]+EABs time	[35]+EABs #bt	Our method = [35]+EABs+DDCO #EABs with DCs	#DC opt.	time	peak poly.
4	96	0.23	0.01	1.21	0.17	8	0.26	17	11	5	0.23	61
8	400	31.83	16.78	1.86	2,486.89	31	0.99	21	27	9	0.95	117
16	1,584	TO	TO	108.23	MO	-	2.68	51	59	17	2.17	325
32	6,256	TO	TO	TO	MO	-	9.36	102	123	33	7.25	1,125
64	24,816	TO	TO	TO	MO	-	49.41	205	251	65	26.87	4,261
128	98,800	TO	TO	TO	MO	-	340.85	397	507	129	149.75	16,677
256	394,224	TO	TO	TO	MO	-	7,341.86	1,053	1,019	257	1,691.72	66,085
512	1,574,896	TO	TO	TO	MO	-	MO	-	2,043	513	27,351.10	263,205

TABLE III
VERIFYING RESTORING DIVIDERS, TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35] time	[35] #bt	[35]+EABs time	[35]+EABs #bt	Our method = [35]+EABs+DDCO #EABs with DCs	#DC opt.	time	peak poly.
4	140	0.27	0.01	1.21	2.59	17	0.47	35	16	8	0.38	61
8	700	14.88	14.27	1.49	MO	-	1.77	45	64	16	1.42	117
16	3,068	TO	TO	16.39	MO	-	8.41	171	256	32	6.63	325
32	12,796	TO	TO	53,277.73	MO	-	65.99	727	1,024	64	29.02	1,125
64	52,220	TO	TO	TO	MO	-	885.71	3,047	4,096	128	193.40	4,261
128	210,940	TO	TO	TO	MO	-	MO	-	16,384	256	2,244.24	16,677
256	847,868	TO	TO	TO	MO	-	MO	-	65,536	512	33,593.30	66,085
512	3,399,676	TO	TO	TO	MO	-	MO	-	262,144	-	TO	-

the numbers of EABs with non-empty don't care sets (Col. 10, "#EABs with DCs") with the actual numbers of don't care optimizations performed (Col. 11, "#DC opt.") in Tab. III, we observe that in particular for restoring dividers DDCO performs don't care optimizations only for a small fraction of the EABs with non-empty don't care sets. The effect is visible especially for larger instances. For the 256-bit divider this percentage is less than 1%, e.g..

Finally, Cols. 13 give the peak polynomial sizes during backward rewriting, counted in number of monomials. It can be observed that these peak sizes grow quadratically with the bit width. This shows that our methods are really successful in keeping the polynomial sizes small, since already the specification polynomial is quadratic in n.

In summary, the presented results show that our new method is able to successfully verify not only the divider benchmarks from [35], but also new divider architectures for which the previous approach fails.

VI. CONCLUSIONS AND FUTURE WORK

We analyzed weaknesses of previous approaches that enhanced backward rewriting in a SCA approach with forward information propagation and we presented two major contributions to overcome those weaknesses. The first contribution is the usage of Extended Atomic Blocks to enable stronger don't care computations. The second one is the new method of Delayed Don't Care Optimization which has two benefits: First, it performs don't care optimizations in a more global rewriting context instead of seeking for only local optimizations of polynomials, and second it is able to effectively minimize the number of don't care optimizations compared to considering all possible combinations of using / not using don't cares of EABs which can potentially occur in a backtracking approach. We showed that our new method is able to verify large divider designs as well as different divider architectures. For the future, we believe that the general approach of combining backward rewriting with forward information propagation will be a key concept to verify further divider architectures as well as other arithmetic circuits at the gate level.

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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[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*, 2016, pp. 1048–1053.

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

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

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

[15] 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.

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

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

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

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

[20] 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, to appear.

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

[22] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits using binary moment diagrams," *Int. J. Softw. Tools Technol. Transf.*, vol. 3, no. 2, pp. 137–155, 2001.

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

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

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

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

[27] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, vol. 7, no. 3, pp. 218–222, 1958.

[28] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *DAC*, 1996, pp. 661–665.

[29] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking - avoiding the Pentium FDIV error," in *DAC*, 1996, pp. 645–648.

[30] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal Comput. Math.*, vol. 1, pp. 148–200, 1998.

[31] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," *Form Methods Syst. Des.*, vol. 14, no. 1, pp. 7–44, 1999.

[32] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating point hardware," *Intel Technology Journal*, vol. Q1, pp. 1–10, 1999.

[33] 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.

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

[35] 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.

[36] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization," in *ICCAD*, 1991, pp. 514–517.

[37] I. Koren, *Computer arithmetic algorithms*. Prentice Hall, 1993.

[38] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD*, 1990, pp. 126–129.

[39] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com

[40] F. Somenzi, "Efficient manipulation of decision diagrams," *STTT*, vol. 3, no. 2, pp. 171–181, 2001.

[41] A. Konrad, C. Scholl, A. Mahzoon, D. Große, and R. Drechsler, "Benchmarks and binaries," 2022. [Online]. Available: https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=24

[42] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.

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

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