Fast Algebraic Rewriting Based on And-Inverter Graphs

Cunxi Yu, *Student Member, IEEE*, Maciej Ciesielski, *Senior Member, IEEE*, and Alan Mishchenko, *Senior Member, IEEE*

***Abstract*—Constructing algebraic polynomials using computer algebra techniques is believed to be state-of-the-art in analyzing gate-level arithmetic circuits. However, the existing approach applies algebraic rewriting directly to a gate-level netlist, which has potential memory explosion problem. This paper introduces an algebraic rewriting technique based on the And-Inverter Graph (AIG) representation of gate-level designs. Using AIG-based cut-enumeration and truth table computation, an efficient order of algebraic rewriting is identified, resulting in dramatic simplifications of the polynomial under construction. An automatic approach, which further reduces the complexity of algebraic rewriting by handling redundant polynomials, is also proposed.**

I. Introduction

IMPORTANCE of arithmetic verification problem grows with an increased use of arithmetic modules in embedded systems to perform computation-intensive tasks in multimedia, signal processing, and cryptography applications. One of the remaining challenges in formal verification is formal verification of gate-level integer arithmetic circuits, such as multipliers, used extensively in those applications. Despite a considerable progress in verification of random and control logic, advances in formal verification of arithmetic designs have been slow. This can be attributed to the difficulty in the efficient modeling of arithmetic circuits and datapaths without resorting to computationally expensive Boolean methods, such as BDDs, SAT, SMT, etc., that require "bit blasting", i.e., flattening the design to a bit-level netlist. However, recently, formal techniques based on *computer algebra* have been successfully applied to the verification problems of gate-level arithmetic circuits.

Computer algebra techniques, which construct the polynomial representation of a gate-level arithmetic circuit, are believed to offer best solution for analyzing arithmetic circuits [1][2][3][4]. These works address the verification problems of Galois field arithmetic and integer arithmetic implementations, including abstractions and reverse engineering [3][4][1]. The verification problem is typically formulated as a proof that the implementation satisfies the specification, which is solved by polynomial division or algebraic rewriting. The results show that the computer algebra techniques provide several orders of magnitude performance improvement. The main advantage of computer algebra methods for verifying arithmetic circuits is that it provides a large number of polynomial reductions (by eliminating non-linear terms) while applying those techniques to a binary encoded specification polynomial. For example, let a polynomial expression be $E = 2x_1 + a + b - 2ab$, where x_1 is an output of an AND2 gate with inputs a and b. After rewriting the algebraic model of AND2$(a, b) = ab$, $E = 2ab + a + b - 2ab = a + b$. We can see that the non-linear term ab has been eliminated. Note that non-linear terms could explode exponentially after rewriting the variables in that term if they were not eliminated at the right time (order).

The order of rewriting or performing polynomial divisions has a significant impact on the performance of the computer algebra techniques [4][5]. However, computer algebra techniques may fail to find an efficient order of nodes in the gate-level arithmetic circuits. The main reason is that these techniques are applied to the original netlist. Yu et al. [5] compared the performance of algebraic methods on combinational gate-level multipliers when different topological orders are used. It showed that an efficient topological order may not exist in the post-synthesized gate-level netlist. Even if such an order exists, it may be difficult to identify because of the polynomial reductions hidden in the complex standard cells. In addition, redundant polynomials detected from combinational and sequential arithmetic circuits can provide significant polynomial reductions [6]. However, detecting such polynomials is limited by manual operations and depends on the structure of the circuits.

The approach presented in this paper aims at improving the efficiency of algebraic rewriting in the context of arithmetic verification. It addresses the problem by using a compact and uniform representation of the Boolean network called the *And-Inverter Graph* (AIG) [7]. Algebraic rewriting is performed by deriving the arithmetic function of the circuit from its low-level circuit representation. Instead of directly applying algebraic rewriting to the gate-level netlist, it is applied to an AIG. Additionally, this approach allows to automatically generate redundant polynomials, which significantly reduce the complexity of algebraic rewriting.

II. Background

A. Formal Verification of Arithmetic Circuits

Verification of arithmetic circuits is performed using a variation of *combinational equivalence checking* (CEC) referred to as *arithmetic combinational equivalence checking* (ACEC) [4]. Several approaches have been applied to equivalence check an arithmetic circuit against its functional specification, including *canonical diagrams*, *satisfiability* theories, *theorem proving*, etc. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams (BDDs) [8], Binary Moment Diagrams (BMDs) [9] [10], Taylor Expansion Diagrams (TED) [11], and other hybrid

C. Yu and M. Ciesielski are with the Department of Electrical and Computer Engineering at University of Massachusetts, Amherst, MA, 01003, USA (ycunxi@umass.edu, xiangyuzhang@umass.edu, ciesiel@ecs.umass.edu).

A. Mishchenko is with EECS Department at University of California, Berkeley, Berkeley, CA 94720 (alanmi@berkeley.edu)

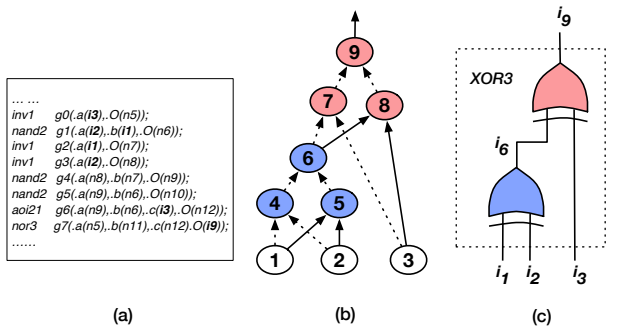

Fig. 1. a) Post-synthesized XOR3 gate-level netlist. b) AIG of the synthesized XOR3 gate-level netlist. (c) The extracted two XOR2 functions (nodes 6 and 9) and one XOR3 function (node 9).

diagrams. While BDDs have been used extensively in logic synthesis, their application to verification of arithmetic circuits is limited by the prohibitively high memory requirements for complex arithmetic circuits, such as multipliers. Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers have also been applied to solve ACEC problems [12]. Recently, several state-of-the-art SAT and SMT solvers have been applied to those problems, including MiniSAT[13], Lingeling[14], Boolector [15], Z3 [16], etc. However, the complexity of checking equivalence of large arithmetic circuits is extremely high [17][5]. Alternatively, the problem can be modeled as checking equivalence against the arithmetic function, e.g. checking whether the binary encoded output function is equivalent to the expected arithmetic function using bit-vector formulation of SMT. However, the complexity of this method is the same as the CEC method [5].

B. Computer Algebra Approaches

Computer algebra method is believed to be the best technique for solving arithmetic verification problems. Using computer algebra methods, the verification problem is typically formulated as a proof that the implementation satisfies the specification [1][2][3][4]. This task is accomplished by performing a series of divisions of the specification polynomial by a set of polynomials, representing components that implement the circuit. Techniques based on *Gröbner Basis* demonstrate that this approach can efficiently transform the verification problem to *membership testing* of the specification polynomial in the ideals [2]. A different approach to arithmetic verification of gate-level circuits has been proposed using the algebraic rewriting technique, which transforms the polynomial at the primary outputs to a polynomial in terms of primary inputs [1], called *function extraction*. This approach has successfully been applied to 512-bit multipliers, due to a large number of polynomial reductions gained by rewriting a binary encoded polynomial of the outputs [5]. A similar approach has been applied to arithmetic combinational equivalence checking [4]. Although those works showed good performance in solving arithmetic verification problems, they still suffer from potential polynomial (memory) explosion problem since they are applied to the original gate-level netlist.

C. Boolean network

A Boolean network is a directed acyclic graph (DAG) with nodes representing logic gates and directed edges representing wires connecting the gates. In the sequential network, the memory elements are assumed to be D flip-flops with known initial states. And-Inverter Graph (AIG) is a combinational Boolean network composed of two-input AND-gates and inverters [7]. In an AIG, each node has at most two incoming edges. A node with no incoming edges is a primary input (PI). Primary outputs are represented using specific output nodes. Each internal node in the AIG represents a two-input AND function. Based on DeMorgan's rule, the combinational logic of an arbitrary Boolean network can be transformed into an AIG [18], with the properly labeled edges to indicate the inversion of some signals. AIGs have been extensively used in logic synthesis, technology mapping [18] and formal verification [19].

AIGs have been used to detect unobserved Boolean functions such as *Multiplexer*-function [20] in an arbitrary gate-level circuits. This method is based on computing a *Cut* in the AIG. A cut *C* of node n is a set of nodes of the network called *leaves*, such that each path from PIs to n passes through the leaf nodes. Node n is the *root* of a *Cut*. A *Cut* is K-feasible if the number of leaves does not exceed K. The cut function is the function of node n in terms of the cut leaves. An AIG node n in an AIG structure that represents a Boolean function F, is called an F-node. Each node is an AND function and the edges indicate the inversions of Boolean signals[1]. An example of identifying XOR functions embedded in the AIG is shown in Figure 1. The AIG shown in Figure 1(b) represents a sub-circuit described in Figure 1(a). It includes a 3-feasible *Cut* of $node$ 9 and a 2-feasible *Cut* of $node$ 6, among other possible 3-feasible cuts. Let the function of the AIG nodes be i_x, and x be the index value of the node. The function of $node$ 6 is $i_1 \oplus i_2$, and the function of $node$ 9 is $i_1 \oplus i_2 \oplus i_3$. Hence, $node$ 6 is an *XOR2*-node, and $node$ 9 is an *XOR3*-node. This means that an embedded XOR3 function consisting of two XOR2s exists and can be detected in the sub-circuit shown in Figure 1(a). Similarly, an AIG can be applied to identify embedded *Majority* functions.

D. Computer Algebraic Model

In this approach, the circuit is modeled as an AIG containing the following gates: INV, AND, embedded MAJ3, and embedded XOR3. This is in contract to using a standard-cell network after synthesis and technology mapping [1]. The following algebraic equations describe the algebraic model used in this work.

$$\begin{aligned} &\neg a = 1 - a \\ &a \wedge b = ab \\ &MAJ3(a,b,c) = ab + ac + bc - 2abc \\ &XOR3(a,b,c) = a \oplus b \oplus c = a + b + c - 2ab - 2ac - 2bc + 4abc \end{aligned} \tag{1}$$

Similarly to [1], the algebraic rewriting for a circuit is based on two polynomials referred to as *output signature*

[1]In Fig.1, the dash edges are inversion signals, e.g. $i_4 = \overline{i_1}\ \overline{i_2}$, $i_5 = i_1 i_2$.

and *input signature*. The *input signature*, Sig_{in}, is a polynomial in terms of primary input variables that uniquely represents the integer function computed by the circuit, i.e., its *specification*. For example, an n-bit binary adder with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, is described by $Sig_{in} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. In our approach, the input specification need not be known; it will be derived from the circuit implementation as part of the verification process. The *output signature*, Sig_{out}, of the circuit is a polynomial in terms of the primary output variables. Such a polynomial is uniquely determined by the n-bit encoding of the output, provided by the designer. This means that the binary encoding of the primary output variables is assumed to be known.

E. Simplified Polynomial Construction

According to [5], efficiency of algebraic rewriting of Sig_{out} is determined by the amount of simplification during polynomial construction. This is because there is a large number of non-linear terms generated by *carry-out* (MAJ) and *sum*(XOR) functions, since multiplication is done by a series of additions. Finding the maximum polynomial cancellations has been previously addressed by improving the topological order of the gates [5]. For example, let a sub-polynomial expression be Nx_1+$2Nx_2$+..., x_1 = *XOR3(a, b, c)*, x_2 = *MAJ3(a, b, c)*, where *a, b, c* are the inputs of XOR3 and MAJ3 functions. According to Equation 1, rewriting x_1 and x_2 together, four non-linear terms are eliminated, namely $2Nab$, $2Nbc$, $2Nac$ and $4Nabc$, generated by the algebraic models of XOR3 and MAJ3. However, if rewriting is applied directly to the gate-level netlist, its efficiency is restricted when the MAJ3 and XOR3 functions are mapped into other standard cells by logic synthesis and technology mapping to reduce the delay and area of the hardware implementation. For example, the XOR3 function mapped using standard cells is shown in Figure 1(a). In this case, there is no ordering that provides the maximum polynomial reductions.

III. Approach

This section presents the algebraic rewriting approach based on AIGs. Similarly to [1], the algebraic rewriting process rewrites the output signature for all AIG nodes in a topological order. As discussed in Section III-E, the rewriting order that provides a large number of polynomial reductions, has significant impact on the performance of rewriting. However, there are many topological orders available in an AIG, since many nodes can have the same topological depth. This approach detects a topological order for algebraic rewriting that provides the maximum polynomial reduction. This is achieved by detecting pairs of MAJ3 and XOR3 nodes using AIG-based *cut enumeration*.

A. Outline of the Approach

The proposed flow is outlined in Algorithm 1. The inputs to the algorithm are: the gate-level netlist and the output signature Sig_{out}. The flow includes three basic steps: 1) converting the gate-level implementation into AIG; 2) detecting all pairs of XOR3 and MAJ3 nodes with identical inputs; topological sorting the AIG nodes while considering the detected pairs as one element; and 3) applying algebraic rewriting from POs to PIs following the topological order determined in step 2). Note that XOR2 and MAJ2(AND2) are the special cases of XOR3 and MAJ3, where one of the inputs is constant zero. The second step is performed as follows:

- Computing all 3-feasible (3-input) cuts of all AIG nodes.
- Computing truth tables of all cuts.
- Storing cuts in the hash table by their ordered set of inputs.
- Detecting pairs of 3-input cuts with identical inputs belonging to different nodes, such that the Boolean functions of the two cuts with the shared inputs belong to the *NPN* classes of XOR3 and MAJ3, respectively.

Note that, in this approach, matching the XOR3 and MAJ3 nodes does not require the inputs and outputs polarity to be the same. Instead, all the cut-points are matched without considering their complemented attributes. For example, instead of being an exact XOR3, the function of a 3-feasible cut can be either XOR3 or XNOR3. Similarly, instead of being exactly MAJ3, the function of can be one of the eight functions forming the *NPN* class of MAJ3 [21]. To compute the cuts, the 3-input cut enumeration is performed in a topological order as described in [22]. The truth tables of the cuts are obtained as a by-product of the cut enumeration. Thus, when two fanin cuts are merged during the cut computation and the resulting cut is 3-feasible, the truth tables of fanin cuts are permuted to match the fanin order of the resulting cut. These truth tables are then ANDed or XORed, depending on the node type, to get the resulting truth table. For the case of 3-input cuts, a dedicated pre-computation reduces the runtime of truth table computation to a small fraction of that of cut enumeration.

As soon as the XOR3 and MAJ3 pairs are detected, algebraic rewriting will be applied to the AIG network in a constrained topological order, in which each XOR3 and MAJ3 pair is considered as one element. This means that at one topological depth, whenever either XOR3 or MAJ3 node of a pair (or its complement) is rewritten, the subsequent rewritten node is of the other type. The AIG nodes with the same topological depth that do not belong to any pair are ordered in the decreasing order of their integer IDs. The algebraic rewriting ends when all elements in AIG network have been rewritten. The algorithm returns the extracted input signature.

Example 1 (2-bit CSA-multiplier): The mapped gate-level netlist of a 2-bit CSA-multiplier is shown in Figure 2(a). First, the gate-level netlist is converted to an AIG representation

Algorithm 1 Algebraic Rewriting in AIG

Input: Gate-level netlist, output signature Sig_{out}
Output: *Pseudo-Boolean* expression extracted by rewriting

```
Structural hashing the gate-level netlist into AIG, denoted G(V, E).
Detect all XOR3 and MAJ3 nodes in G(V, E).
Pair the XOR3 and MAJ3 if they have identical signals, denoted as P.
Topological sort G(V,E) considering each element in P as one node.
i = 0; F_i = Sig_out
while there are no elements remained in the topological order do
    Rewrite: F_{i+1} = F_i by substituting the variables with algebraic equations;
    i = i + 1
return F = F_i (to be compared with Sig_in)
```

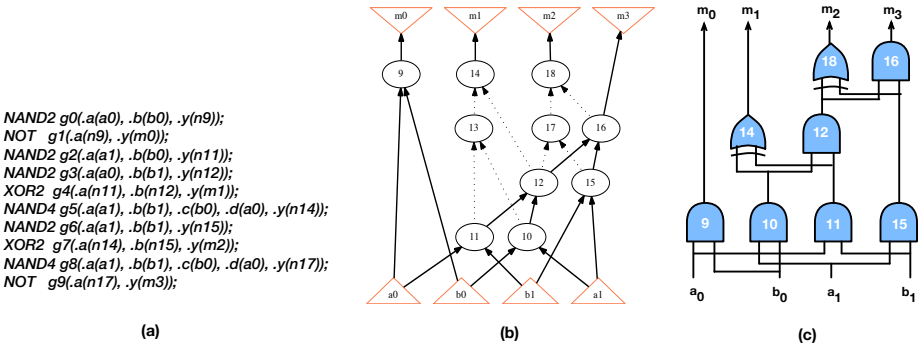

Fig. 2. (a) Post-synthesized 2-bit multiplier gate-level netlist; (b) The AIG of the 2-bit multiplier shown in Figure 2(a); (c) Detected unobserved functions from the AIG and the correspondences to AIG nodes. The index value in Figure 2(c) corresponds to the hash value of each node in Figure 2(b).

(Figure 2(b)). Then, a set of XOR3 nodes X, and a set of MAJ3 nodes M are detected. X = {*14, 18*}, M = {12, 16}. *Node 14* is *XOR3(10, 11, 1'b0)* and *node 12* is *MAJ3(10, 11, 1'b0)*, where *nodes 10 and 11* and constant zero *1'b0* are the inputs; *node 18* is *XOR3(12, 15, 1'b0)* and *node 16* is *MAJ3(12, 15, 1'b0)*; *1'b0* is Boolean *false*. Hence, two pairs of XOR3 and MAJ3 are generated, *(14, 12)* and *(18, 16)*. The order of rewriting is determined as follows: 1) *node 18* is the node with highest depth; it is detected as a XOR3 and paired with a MAJ3 *node 16*; hence, the first rewriting starts from node 18 and 16, and ends at node 12 and 15; 2) similarly to the first rewriting, the second rewriting starts from nodes 14 and 12, and ends at nodes 11 and 10; 3) the remaining AIG nodes are ordered by their index value in decreasing order. The logic network after detecting all XOR3 and MAJ3 functions are shown in Figure 2(c).

B. Detecting Redundant Polynomials

Significant simplification of algebraic polynomial construction can be achieved not only by performing algebraic rewriting using a reserve-topological order, as discussed above, but also by detecting redundant polynomials. For example, *don't-care* polynomials and *vanishing* polynomials are used for verifying sequential arithmetic circuits [6]. These polynomials are generated by observing that the signals removed to make design more efficient, contain algebraic information that is needed to cancel algebraic terms of the remaining output bits in the design. For example, the polynomial associated with the most significant bit (MSB) of an adder or a multiplier is such a polynomial. Such truncated designs are widely used in energy efficient applications by reducing critical paths and pruning the logic. However, automatically generating the redundant polynomials has not been addressed so far.

To efficiently apply algebraic rewriting to the multipliers with output bits truncated, an approach that automatically generates *don't-care* polynomials is presented. This approach is based on an observation that the logic obtained by removing output bits is either a carry-out function or a sum function of a 1-bit adder. It is known that MAJ3 and XOR3 are the components of a 1-bit adder. Hence, using the approach of detecting pairs of XOR3 and MAJ3 (Section III-A), the XOR3/MAJ3 nodes that do not belong to any such pairs are also identified. For example, a n-bit CSA-multiplier with *2n-1* output bits (with MSB removed), there is a missing MAJ3, i.e., the MAJ3s with identical inputs of an unpaired XOR3. Since one pair of XOR3 and MAJ3 is a 1-bit adder, removing the carry bit (MAJ3) makes the function an addition *modulo 2*. In this case, the algebraic model of XOR3 (Equation 1) is reduced to $a \oplus b \oplus c$ = *a+b+c mod 2*.

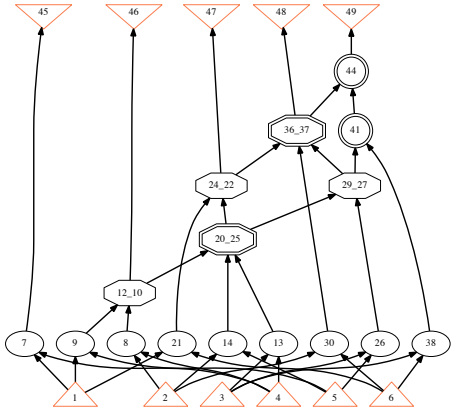

Fig. 3. Detecting *MAJ3-XOR3* of a 3-bit post-synthesized CSA-multiplier with MSB z_5 deleted.

Example 2 (3-bit CSA-multiplier with MSB z_5 detected): The AIG after detecting XOR3 and MAJ3 pairs of a 3-bit post-synthesized CSA-multiplier with MSB deleted is shown in Figure 3. The detected XOR3 and MAJ3 pairs are represented using the ID of the root node of the XOR3 and MAJ3 nodes. We can see that there is one XOR3 (composed of two XOR2 nodes, *41* and *44*) with inputs $i_{36,37}$, $i_{27,29}$ and i_{38}, that cannot be paired with any MAJ3. This is because synthesis process removed the redundant logic (last carry out) while the MSB has been removed. In this case, the algebraic model of that XOR3 is reduced to $2^4 \cdot z_4(i_{49}) = 2^4 \cdot (i_{36,37} + i_{27,29} + i_{38})$.

IV. Results

The technique described in this paper was implemented in ABC [18]. It applies algebraic rewriting to the AIG and generates the polynomial signature. The experiments include gate-level rewriting of Carry-Save-Adder (CSA) multipliers up to 512 bits. The results are compared with *functional extraction* [1]. The results show that the proposed technique is

TABLE I
RESULTS OF APPLYING AIG-BASED ALGEBRAIC REWRITING TO PRE- AND POST-SYNTHESIZED CSA MULTIPLIERS COMPARED TO *functional extraction* PRESENTED IN [1]. *t(s) IS THE RUNTIME IN SECONDS. *mem IS THE MEMORY USAGE IN MB.

#bits	*Pre-synthesized*				*Post-synthesized*			
	[1]		This approach		[1]		This approach	
	t(s)	mem	t(s)	mem	t(s)	mem	t(s)	mem
64	1.89	74	0.04	34	5.50	76	0.04	34
128	8.12	288	0.15	117	39.64	299	0.16	120
256	32.65	1157	0.82	441	285.22	1250	0.82	439
512[2]	130.22	4427	3.76	1695	-	-	-	-

TABLE II
RESULTS OF APPLYING AIG-BASED ALGEBRAIC REWRITING TO POST-SYNTHESIZED COMPLEX ARITHMETIC CIRCUITS COMPARED TO *functional extraction* PRESENTED IN [1]. *MO = MEMORY OUT OF 8 GB.

Benchmarks (256-bit)	[1]		This approach	
	runtime(s)	mem(MB)	runtime(s)	mem(MB)
$F=A\times B+C$	179.1	1182	5.1	447
$F=A\times(B+C)$	209.3	1120	5.1	451
$F=A\times B\times C$	-	MO	37.5	2871
$F=1+A+A^2+A^3$	-	MO	47.1	3331

more efficient than the state-of-the-art technique for extracting the polynomial expressions for the CSA multipliers. The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory.

We evaluate our AIG-based algebraic rewriting approach using pre-synthesized and post-synthesized CSA multipliers shown in Table I, and post-synthesized complex arithmetic circuits shown in Table II. The gate-level multipliers are taken from [1]. The runtime and memory usage are compared to *functional extraction* [1]. In Table II, the functions of the arithmetic circuits are shown in the first column. The bit-width varies between 64 and 512 bits[2]. We can see that the runtime of the proposing approach is lower than a second for both the pre- and post-synthesized CSA multipliers for any bit-width. The memory usage has been reduced on average 60%, compared to *function extraction* [1]. Also, the complexity of extracting the polynomial expression using functional extraction is increased when the multipliers are synthesized. For example, extracting post-synthesized 256-bit multiplier using functional extraction requires 9x more runtime and more memory. However, using the proposed approach, the runtime of extracting pre- or post-synthesized multipliers are almost the same. More importantly, we can see that our approach surpass *functional extraction* on complex arithmetic designs (Table II).

V. CONCLUSION

This paper presented a method to improve the efficiency of algebraic rewriting used in arithmetic verification. The method is based on the representation of the Boolean network called an And-Inverter Graph (AIG). This approach allows for formal verification of practical multipliers that are heavily optimized and mapped using 14nm technology library. Another contribution of the paper is a technique that automatically detects redundant polynomials to reduce the complexity of algebraic rewriting.

[2]512-bit post-synthesized multipliers are not reported in [1].

ACKNOWLEDGMENT

This work was supported by an award from National Science Foundation, No. CCF-1319496 and No. CCF-1617708. The co-author affiliated with UC Berkeley was supported in part by NSA grant "Enhanced equivalence checking in crypto-analytic applications".

REFERENCES

[1] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of Gate-level Arithmetic Circuits by Function Extraction," in *52nd DAC*. ACM, 2015, pp. 52–57.
[2] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *IEEE Trans. on CAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.
[3] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel, "STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.
[4] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal Verification of Integer Multipliers by Combining Grobner Basis with Logic Reduction," in *DATE'16*, 2016, pp. 1–6.
[5] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal Verification of Arithmetic Circuits using Function Extraction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.
[6] C. Yu and M. Ciesielski, "Formal Verification Using Don't-Care and Vanishing Polynomials," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2016, pp. 284–289.
[7] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG Rewriting A Fresh Look at Combinational Logic Synthesis," in *43rd DAC*. ACM, 2006, pp. 532–535.
[8] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. 100, no. 8, pp. 677–691, 1986.
[9] R. E. Bryant and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *DAC'95*.
[10] Y.-A. Chen and R. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Circuit Verification," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-97-134, 1997.
[11] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.
[12] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking," in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001, pp. 114–121.
[13] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.
[14] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," *Proceedings of SAT Competition*, pp. 51–52, 2013.
[15] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2015.
[16] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[17] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction From Combinational Circuits for Verification Over Finite Fields," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 7, pp. 1206–1218, 2016.
[18] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.
[19] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.
[20] C. Yu, M. J. Ciesielski, M. Choudhury, and A. Sullivan, "DAG-aware logic synthesis of datapaths," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 2016, pp. 135:1–135:6.
[21] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December, 2013*.
[22] P. Pan and C. Lin, "A New Retiming-Based Technology Mapping Algorithm for LUT-based FPGAs," in *FPGA*, 1998, pp. 35–42.