Fast Algebraic Rewriting based on And-Inv-Graphs

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

***Abstract*—Based on the successes of arithmetic verification using computer algebra method, this paper presents a fast computer algebraic rewriting technique based on And-Inv-Graphs (AIGs). This technique significantly improves the performance of the existing methods. Computer algebra method with polynomial representations is believed be the state-of-the-art technique for analyzing gate-level arithmetic circuits. However, the existing works are all restricted to applying algebraic rewriting directly on the gate-level netlist, which has potential memory explosion problem. In this paper, an AIGs-based algebraic rewriting technique that rewrites from the primary outputs (POs) to primary inputs (PIs), through the AIGs of the gate-level design, has been introduced. Using the AIG-based structural hashing, an efficient order of rewriting can be identified, which provides the maximum polynomial reductions during the rewriting process. This approach is based on detecting the pairs of Majority3 and XOR3 functions in AIGs. Also, an automatic approach which reduces the complexity of algebraic rewriting and generates redundant polynomials is proposed.**

I. Introduction

IMPORTANCE of arithmetic verification problem grows because of an increased use of arithmetic modules in embedded systems to perform computation-intensive tasks for 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. Despite a considerable progress in verification of random and control logic, advances in formal verification of arithmetic designs have been lagging. This can be attributed mostly 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. Until recently, formal techniques based on *computer algebra* have been successfully applied to the verification problems of gate-level arithmetic circuits.

Computer algebra techniques with polynomial representations is 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 proving that the implementation satisfies the specification, by polynomial division or algebraic rewriting. The results show that the computer algebra techniques provide several orders of magnitude performance improvements. 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 $E = 2x_1 + a + b - 2ab$, and 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.

The order of rewriting or polynomial divisions has a significant impact on the performance of computer algebra techniques [4][5]. However, computer algebra techniques may be limited to finding the efficient orders in the gate-level arithmetic circuits. The main reason is that those techniques are restrictedly applied to the original netlist. Yu et al. [5] compared the performance of different topological orders of the combinational gate-level multipliers. It showed that the efficient topological order might not exist in the post-synthesized gate-level netlist are difficult to be identified because of the polynomial reductions are hidden in the complex standard cells. In addition, redundant polynomials detected from combinational and sequential arithmetic circuits provide significant polynomial reductions [6]. However, detecting such polynomials is limited to manual operations and the structure of the circuits.

The approach presented in this paper aims at improving the efficiency of algebraic rewriting for arithmetic verification. It addresses the problem by using a *Direct-acyclic-graph* (DAG) representation of Boolean network, called *And-inverter-graph* (AIGs) [7]. The algebraic rewriting is done by deriving arithmetic function computed by the circuit from its low-level circuit implementation. Instead of directly applying on the gate-level netlist, the algebraic rewriting is performed on AIGs. Also, this approach is used to automatically generate redundant polynomials, which significantly reduce the complexity of algebraic rewriting.

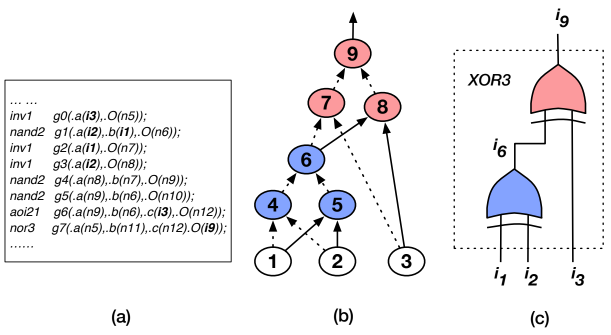
II. Background

A. Formal Verification of Arithmetic Circuits

Verification of arithmetic circuits is solved using *combinational equivalence checking* (CEC), typically also refers to *arithmetic combinational equivalence checking* (ACEC) [4]. Several approaches have been applied to check an arithmetic circuit against its functional specification, such as *canonical diagrams*, *satisfiability* theories, *theorem proving*, etc. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams

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@eecs.berkeley.edu)



```
… …
inv1     g0(.a(i3),.O(n5));
nand2   g1(.a(i2),.b(i1),.O(n6));
inv1     g2(.a(i1),.O(n7));
inv1     g3(.a(i2),.O(n8));
nand2   g4(.a(n8),.b(n7),.O(n9));
nand2   g5(.a(n9),.b(n6),.O(n10));
aoi21   g6(.a(n9),.b(n6),.c(i3),.O(n12));
nor3    g7(.a(n5),.b(n11),.c(n12).O(i9));
……
```

Fig. 1. A 3-feasible Cut of node 9. a) A Cut includes two XOR2-nodes (6, 9) and one XOR3-node (9). b) The corresponding Boolean function.

(BDDs) [8], Binary Moment Diagrams (BMDs) [9] [10], Taylor Expansion Diagrams (TED) [11], and other hybrid diagrams. While BDDs have been used extensively in logic synthesis, their application to verification of arithmetic circuits is limited by the prohibitively high memory requirement for complex arithmetic circuits, such as multipliers. Boolean satisfiability (SAT) or 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, this 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 in solving the arithmetic verification problems. Using computer algebra methods, the verification problem is typically formulated as proving that the implementation satisfies the specification [3][2][1][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 of primary inputs [1], called *function extraction*. This approach successfully has been applied to 512-bit multipliers, because of 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 great performance in solving arithmetic verification problems, there is potential polynomial (memory) explosion problem since they all applied restrictedly to the original gate-level netlist.

C. *Boolean network*

A Boolean network can be represented using 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 the AIGs, each node has either 0 or two incoming edges. A node with no incoming edges is a primary input. Primary outputs are represented using specific output nodes. Each node in an AIGs represents a Boolean AND function. Based on DeMorgan's rule, the combinational logic of an arbitrary Boolean network can be factored and transformed into an AIG [18], with the edges indicate the inversion of the signals.

AIG has been applied to search unobserved Boolean function such as searching *Multiplexer*-function [19] in an arbitrary gate-level circuits. This is based on computing the *Cut* of AIGs. 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 at least one leaf. Node n is the *root* of the *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 defined as F-node. Each node is an AND function and the edges indicate the inversions of Boolean signals[1]. An example of identifying unobserved XOR functions using AIGs 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 each node. The function of *node* 6 equals to $i_1 \oplus i_2$, and the *node* 9 equals to $i_1 \oplus i_2 \oplus i_3$. Hence, *node* 6 is an *XOR2*-node, and *node* 9 is an *XOR3*-node. This means that an unobserved XOR3 function consisting of two XOR2s, is detected in the sub-circuit shown in Figure 1(a). Similarly, AIGs can be applied to identify unobserved *Majority* functions.

D. *Computer Algebraic model*

In this approach, the circuit is modeled as an AIG, including (AND, INV, and implicit MAJ3, XOR3), instead of a network of standard cells after synthesis and technology mapping [1]. The following algebraic equations are used to describe the algebraic model used in this work.

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

Similarly to the work of [1], the algebraic rewriting process of the circuits is specified by two polynomials, referred to

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

as *output signature* and *input signature*. The *input signature*, Sig_{in}, is a polynomial in 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 defined as a polynomial in the primary output signals. 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 output bits is assumed to be known.

E. Polynomial Reductions

According to [5], the reason why algebraic rewriting from POs to PIs is efficient is that rewriting the Sig_{out} provides significant internal polynomial reductions. This is because there are a large number of non-linear terms generated by *carry out* (MAJ) function and *sum*(XOR), since multiplication is done by a series of additions. Find the maximum polynomial cancellations were previously addressed by improving the topological order and property ordering the gates [5]. For example, let a sub-polynomial expression be Nx_1+$2Nx_2$+..., x_1 = *XOR3(a, b, c)*, x_2 = *MAJ(a, b, c)*. According to Equation 1, rewriting x_1 and x_2 together, four non-linear terms will be eliminated, $2Nab$, $2Nbc$, $2Nac$ and $4Nabc$, that are generated by the XOR3 and MAJ3 algebraic models. However, if rewriting is applied directly to the gate-level netlist, its efficiency is limited to when the MAJ and XOR3 functions are technology mapped using standard cells, e.g. it maps XOR3 function as 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 through all the elements in the AIGs in a topological order. As discussed in Section III-E, the ordering of rewriting that provides a large number of polynomial reductions, has significant impacts on the performance of rewriting. However, mostly, there are many topological orders available in an AIG, since many nodes can have the same topological depth. The proposed approach is applied to detect a topological order for algebraic rewriting that provides the maximum polynomial reductions. This is achieved by searching the pairs of MAJ3 and XOR3 nodes by computing the AIG *Cuts*.

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 basically three steps: 1) convert the gate-level implementation into AIGs; 2) detect all pairs of XOR3 and MAJ3 nodes with identical inputs; topologically sort the AIG nodes while considering the detected pairs as one element; 3) apply 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, which one of the input is constant zero. The second step specifically processes as following:

- Compute all 3-feasible (3-input) cuts of all AIG nodes.
- Compute truth tables of all cuts.
- Hash the cuts by their ordered set of inputs.
- Find pairs of 3-input cuts with identical inputs, w.r.t to different nodes, such that the Boolean functions of the two cuts regarding the shared inputs are in *NPN* classes of XOR3 and MAJ3, respectively.

Algorithm 1 Algebraic Rewriting in AIGs

```
Input: Gate-level netlist, output signature $Sig_{out}$
Output: Pseudo-Boolean expression extracted by rewriting
1: Structural hashing the gate-level netlist into AIGs, denoted G(V, E).
2: Detect all XOR3 and MAJ3 nodes in G(V, E).
3: Pair the XOR3 and MAJ3 if they have identical signals, denoted as $P$.
4: Topological sort G(V,E) considering each element in $P$ as one node.
5: $i$ = 0; $F_i$ = $Sig_{out}$
6: while there are no elements remained in the topological order do
7:     Rewrite: $F_{i+1}$ = $F_i$ by substituting the variables with algebraic equations;
8:     $i$ = i + 1
9: return $F$ = $F_i$ (to be compared with $Sig_{in}$)
```

Note that in this approach, matching the XOR3 and MAJ3 nodes do not require the inputs and outputs polarity to be the same. Instead, all the cut-points are matched without considering the complement. 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 [20]. To compute the cuts, the 3-input cut enumeration is performed in a topological order as described in [21]. 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.

Moreover, when detecting the pairs of XOR3 and MAJ3 nodes, the algebraic coefficient of MAJ3 node is not required to be $2\times$ of the coefficient of the XOR3 node. Although the non-linear terms reductions between those XOR3 and MAJ3 happen only if the coefficients satisfy this condition, there are significant polynomial reductions provided by the detected XOR3 and MAJ3 pairs. This is based on an important observation that the XOR3 and MAJ3 with identical inputs, are likely to construct a one-bit addition (full-adder or half-adder).

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 are 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 following rewriting element is of the other type. The AIG nodes with the same

```
NAND2 g0(.a(a0), .b(b0), .y(n9));
NOT   g1(.a(n9), .y(m0));
NAND2 g2(.a(a1), .b(b0), .y(n11));
NAND2 g3(.a(a0), .b(b1), .y(n12));
XOR2  g4(.a(n11), .b(n12), .y(m1));
NAND4 g5(.a(a1), .b(b1), .c(b0), .d(a0), .y(n14));
NAND2 g6(.a(a1), .b(b1), .y(n15));
XOR2  g7(.a(n14), .b(n15), .y(m2));
NAND4 g8(.a(a1), .b(b1), .c(b0), .d(a0), .y(n17));
NOT   g9(.a(n17), .y(m3));
```

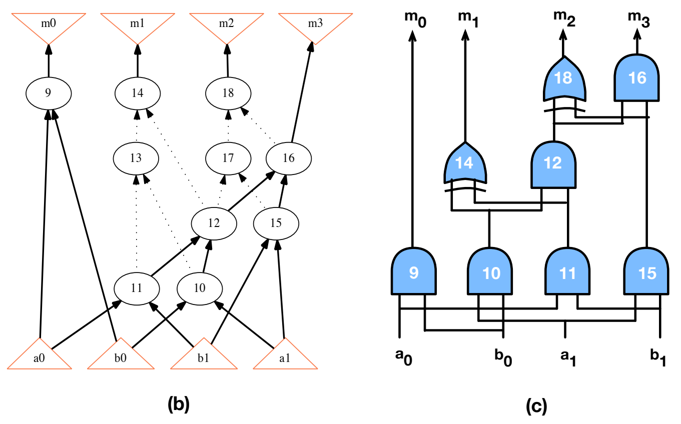


Fig. 2. (a) Post-synthesized 2-bit multiplier gate-level netlist; (b) The AIGs of the 2-bit multiplier shown in Figure 2(a); (c) Detected unobserved functions from the AIGs and the correspondences to AIG nodes. The index value in Figure 2(c) corresponds to the hash value of each node in Figure 2(b).

topological depth that do not exist in any pairs are ordered by their hashing value (decreasing order). The algebraic rewriting ends when all elements in AIG network have been rewritten and 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 (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)*; *node 18* is *XOR3(12, 15, 1'b0)* and *node 16* is *MAJ3(12, 15, 1'b0)*; *1'b0* is Boolean *false*. Hence, the 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) similar to the first rewriting, the second rewriting starts from node 14 and 12, and ends at node 11 and 10; 3) the order of the remaining AIGs nodes are ordered by their index value by decreasing order. The logic network after detecting all XOR3 and MAJ3 functions are shown in Figure 2(c).

B. Redundant Polynomial Determination

In addition to the order of algebraic rewriting, identifying and using redundant polynomials could also provide significant polynomial reductions. For example, *don't-care* polynomials and *vanishing* polynomials are used for verifying sequential arithmetic circuits [6]. Those polynomials are mostly generated by observing the signals that are removed for design efficiency but contain algebraic information that cancels algebraic terms of output bits remaining in the design, e.g. the most significant bit (MSB) of adder or multiplier. Those 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 yet.

To efficiently apply algebraic rewriting on 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 removed logic by removing output bits is either a carry-out function or a sum function of one-bit addition. It is known that MAJ3 and XOR3 are the components that construct a 1-bit addition. Hence, using the approach of detecting pairs of XOR3 and MAJ3 (Section III-A), the XOR3/MAJ3 nodes that do not belong to any pair can be detected. 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 the identical inputs of an unpaired XOR3. Since one pair of XOR3 and MAJ3 constructs a 1-bit addition, removing the carry bit (MAJ3) makes the function be an addition and *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*. The negation of the removed terms by modular 2, $-2ac - 2bc + 4abc$, are the redundant polynomials detected for each unpaired XOR3.

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 hash value of the root node of the XOR3 and MAJ3 nodes. We can see that there is one XOR3 (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 C, and embedded in the ABC system. It applies algebraic rewriting on AIG network and generates the polynomial signature. The experimental results include rewriting on gate-level Carry-save-adder (CSA) multipliers up to 512-bit, and compared to *function extraction* [1]. It shows that our technique is more efficient than the state-of-the-art technique in extracting the polynomial expressions of 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.

The results of applying AIG-based algebraic rewriting on pre-synthesized and post-synthesized CSA-multipliers are

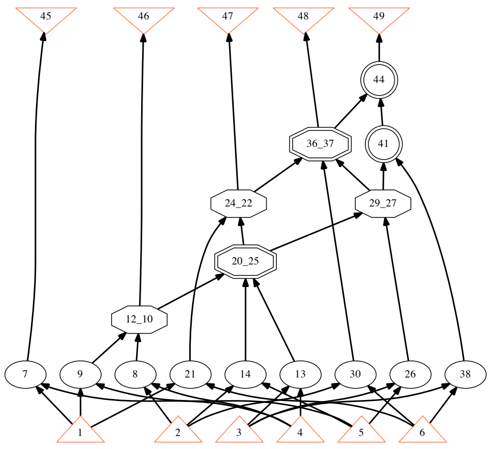


Fig. 3. Detecting *MAJ3-XOR3* of a 3-bit post-synthesized CSA-multiplier with MSB z_5 deleted.

TABLE I
RESULTS OF APPLYING AIG-BASED ALGEBRAIC REWRITING ON PRE-SYNTHESIZED CSA-MULTIPLIERS COMPARED TO *function extraction* [1].

# bits	[1]		This approach	
	runtime(s)	mem(MB)	runtime(s)	mem(MB)
8	0.02	4.9	0.01	9.8
16	0.10	8.2	0.01	10.4
64	1.89	73.9	0.04	34.2
128	8.12	288.4	0.15	117.2
256	32.65	1157.3	0.82	440.5
512	130.22	4427.5	3.76	1695.1

shown in Table I and Table II. The gate-level multipliers are taken from [1]. Runtime and memory usage of *function extraction* [1] are shown in columns 2 and 3, and the results of the proposing approach are shown in columns 4 and 5. The bit-width varies from 8-bit to 256-bit[2]. First, we can see that the runtime of the proposing approach is lower than a second on both the pre- and post-synthesized CSA-multipliers from 8-bit to 256-bit. And, the memory usage has been reduced average 60% compared to *function extraction* [1]. Finally, the complexity of extracting the polynomial expression using function extraction is increased while the multipliers have been synthesized. For example, extracting post-synthesized 256-bit multiplier using function 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.

TABLE II
RESULTS OF APPLYING AIG-BASED ALGEBRAIC REWRITING ON POST-SYNTHESIZED CSA-MULTIPLIERS COMPARED TO *function extraction* [1].

# bits	[1]		This approach	
	runtime(s)	mem(MB)	runtime(s)	mem(MB)
8	0.04	2.9	0.01	9.7
16	0.14	6.1	0.01	10.4
64	5.50	76.3	0.04	34.3
128	39.64	299.2	0.16	120.0
256	285.22	1250.6	0.82	438.9

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

ACKNOWLEDGMENT

This work was supported by an award from National Science Foundation, No. CCF-1319496 and No. CCF-1617708.

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 (2007)," *URL http://www. eecs. berkeley. edu/alanmi/abc*, 2010.

[19] 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.

[20] 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*.

[21] P. Pan and C. Lin, "A New Retiming-Based Technology Mapping Algorithm for LUT-based FPGAs," in *FPGA*, 1998, pp. 35–42.