Parallel Gröbner Basis Rewriting and Memory Optimization for Efficient Multiplier Verification

Hongduo Liu[1], Peiyu Liao[1], Junhua Huang[2], Hui-Ling Zhen[2], Mingxuan Yuan[2], Tsung-Yi Ho[1], Bei Yu[1]
[1]Chinese University of Hong Kong [2]Huawei Noah's Ark Lab

***Abstract*—Formal verification of integer multipliers is a significant but time-consuming problem. This paper introduces a novel approach that emphasizes the acceleration of symbolic computer algebra (SCA)-based verification systems from the perspective of efficient implementation instead of traditional algorithm enhancement. Our first strategy involves leveraging parallel computing to accelerate the rewriting process of the Gröbner basis. Confronting the issue of frequent memory operations during the Gröbner basis reduction phase, we propose a double buffering scheme coupled with an operator scheduler to minimize memory allocation and deallocation. These unique contributions are integrated into a state-of-the-art verification tool and result in substantial improvements in verification speed, demonstrating more than 15× speedup for a 1024×1024 multiplier.**

I. Introduction

Integer multipliers serve a crucial role in digital systems, including microprocessors, digital signal processors, and communication systems. Formally verifying the correctness of multipliers is essential to ensure the reliability and safety of digital systems and prevent the issue of the famous Pentium FDIV bug [1]. Architects have proposed complex architectures for integer multipliers to meet power, performance, and area demands. As the size of operands increases, verifying these multipliers can become significantly challenging and time-consuming.

Verification techniques can be categorized into three main groups. The first one is the decision diagram approach [2], which could suffer from memory explosion issues and heavily relies on the structural information of multipliers. The second is SAT-based [3], which, despite being versatile, struggles with scalability when it comes to larger designs. Currently, the most effective methods draw from symbolic computer algebra (SCA) [4]–[7]. Here, the multiplier's specification and each circuit gate are represented using polynomials. Consequently, the verification process is reduced to a membership test to confirm if the specification polynomial belongs to the ideal derived from the circuit polynomial. The ideal test reduces the specification polynomial with the Gröbner basis generated from the circuit and checks if the remainder equals zero.

Various techniques have been proposed to cut down the time required for SCA-based verification. For instance, Yu et al. [8] reduced the verification complexity by detecting redundant polynomials. Mahzoon et al. [9] presented a method to allow for local cancellation of vanishing monomials in converging gates cones starting from half adders. Furthermore, Kaufmann et al. [7] substituted complex final-stage adders with simple ripple-carry adders and used SAT solvers to verify the equivalence of substitution. These efforts, while valuable, often approach the issue from the perspective of enhancing the algorithm itself. However, this paper opens up an opportunity to accelerate verification by enabling parallel computing and memory footprint optimization.

In the verification process, merely reducing the specification polynomial with the Gröbner basis generated from the circuit often produces an exponential volume of monomials. Thus in practice, the Gröbner basis is rewritten to a simplified version before reducing the specification polynomial. During this stage, some polynomial computations in the Gröbner basis are independent of others. In this work, we leverage the independence to parallelize Gröbner basis rewriting. When reducing the specification polynomial using the simplified Gröbner basis, there can be a significant surge in the number of monomials, leading to massive memory operations. To mitigate memory expenditure, we propose a double buffering scheme to avoid fresh memory allocation. Moreover, we present an operator scheduler to enable concurrent reduction of multiple polynomials from the simplified Gröbner basis, which can further decrease memory cost.

The key contributions of this paper are summarized as follows.

- We observe that the polynomial computation in the Gröbner basis rewriting stage can be processed simultaneously and develop a parallel execution system for the Gröbner basis simplification.
- To combat the substantial increase in memory operations during polynomial reduction, a double buffering strategy is proposed to avoid unnecessary new memory allocation and deallocation. Also, we introduce an operator scheduling system to allow multiple polynomials from the simplified Gröbner basis to be processed concurrently.
- Experiments show that our acceleration framework can achieve more than 15× speedup when verifying a 1024×1024 multiplier.

II. Preliminaries

SCA-based verification can be transformed into a membership test to validate whether the given specification polynomial is a part of the ideal generated from the circuit polynomials. Our initial focus will be on introducing the algebraic foundations based on [10]. We then establish a connection between multiplier verification and ideal membership testing, building on the work of [6]. Finally, we present an illustrative example to demonstrate the testing algorithm.

979-8-3503-4859-0/DATE24/© 2024 EDAA

A. Algebra Basis

Definition 1 (Monomial). *A monomial x^α in $x_1, \cdots, x_n$ is a product of the form $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where $\alpha = (\alpha_1, \cdots, \alpha_n)$ is an n-tuple of nonnegative integers.*

Definition 2 (Polynomial). *A polynomial f in $x_1, \cdots, x_n$ with coefficients in a field K is a finite linear combination (with coefficients in K) of monomials. We can write f in the form $f = \sum_\alpha a_\alpha x^\alpha, a_\alpha \in K$. The set of all polynomials in $x_1, \cdots, x_n$ with coefficients in K is denoted by $K[x_1, \cdots, x_n]$.*

Given a nonzero polynomial $f \in K[x]$, let $f = c_0 x^m + c_1 x^{m-1} + \cdots + c_m$, where $c_i \in K$ and $c_0 \neq 0$. Then we say $c_0 x^m$ is the leading term of f, written as $LT(f) = c_0 x^m$.

Definition 3 (Ideal). *A subset $I \subseteq K[x_1, \cdots, x_n]$ is an ideal if it satisfies: (i) $0 \in I$; (ii) If $f, g \in I$, then $f + g \in I$; (iii) If $f \in I$ and $h \in K[x_1, \cdots, x_n]$, then $hf \in I$.*

Let $f_1, \cdots, f_s$ be polynomials in $K[x_1, \cdots, x_n]$. Then we set $\langle f_1, \cdots, f_s \rangle = \{\sum_{i=1}^s h_i f_i \mid h_1, \cdots, h_s \in K[x_1, \cdots, x_n]\}$. The crucial fact is that $\langle f_1, \cdots, f_s \rangle$ is an ideal and we call it the ideal generated by $f_1, \cdots, f_s$.

Definition 4 (Gröbner Basis). *Fix a monomial order on the polynomial ring $K[x_1, \cdots, x_n]$. A finite subset $G = \{g_1, \cdots, g_t\}$ of an ideal $I \subseteq K[x_1, \cdots, x_n]$ different from $\{0\}$ is said to be a Gröbner basis if $\langle LT(g_1), \cdots, LT(g_t) \rangle = \langle LT(I) \rangle$, where $LT(I)$ is the set of leading terms of nonzero elements of I.*

B. Verification to Ideal Membership Testing

An effective way to model a boolean circuit is through an And-Inverter Graph (AIG). An AIG consists of two-input nodes representing logical conjunction, terminal nodes, and edges optionally marked as dashed lines indicating logical negation. A polynomial can be used to encapsulate the connections between the inputs and output by assigning a variable to each gate's input and output wires. Thus, an AIG node with 'z' as the output and 'a' and 'b' as inputs can be represented as one of the following polynomial relations

$$\begin{array}{lcl} z = a & \Rightarrow & 0 = -z + a \\ z = \neg a & \Rightarrow & 0 = -z + 1 - a \\ z = a \wedge b & \Rightarrow & 0 = -z + ab \\ z = \neg a \wedge b & \Rightarrow & 0 = -z + b - ab \\ z = \neg a \wedge \neg b & \Rightarrow & 0 = -z + 1 - a - b + ab. \end{array}$$

We consider circuits C with two bit-vectors $a_0, \cdots, a_{n-1}$ and $b_0, \cdots, b_{n-1}$ of size n as inputs, and a bit-vector $s_0, \cdots, s_{2n-1}$ of size $2n$ as output. Also, we assign a variable $t_1, \cdots, t_k$ to each internal gate output.

Definition 5 (Polynomial Circuit Constraint). *A polynomial p is called a polynomial circuit constraint (PCC) for a circuit C if for every choice of $(a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}) \in \{0,1\}^{2n}$ and the resulting values $t_1, \cdots, t_k, s_0, \cdots, s_{2n-1}$ which are implied by the gates of the circuit C, the substitution of all these values into the polynomial p gives zero. The set of all PCCs for C is denoted by $I(C)$.*

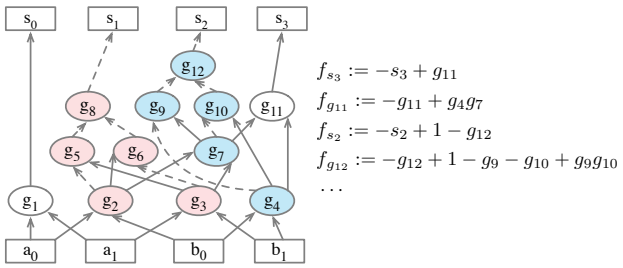

Fig. 1 The AIG representation of a 2×2 multiplier and polynomials inferred by the AIG.

The multiplier verification can be formulated as an ideal membership testing problem stated below.

Problem 1. *A circuit C is called a correct multiplier if the word-level specification*

$$\sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \in I(C).$$

Define $G = \{g_1, \cdots, g_m\} \cup \{x_i(x_i - 1) \mid x \in X\}$, where $\{g_1, \cdots, g_m\}$ is the set that contains the polynomials indicated by corresponding AIG and X is the set of all variables in C. According to the proofs in [6], we have the following facts.

Theorem 1. *For an acyclic circuit, we have $I(C) = \langle G \rangle$.*

Theorem 2. *G is a Gröbner basis for $\langle G \rangle$.*

Theorem 3. *Let $H = \{f_1, \cdots, f_n\} \subseteq K[X]$ be a Gröbner basis, and let $f \in K[X]$. Then $f \in \langle H \rangle$ if and only if the remainder of f with respect to G is zero.*

Integrating the theorems mentioned above, we know that the correctness of the multiplier can be established by reducing the specification polynomial by polynomials in G and checking if the result is zero. Let $p \xrightarrow{f} r$ denote that polynomial p divides polynomial f and results in a remainder of r. Then the verification of the multiplier shown in Fig. 1 can be done following the steps:

- Get the specification polynomial $sp := 8s_3 + 4s_2 + 2s_1 + s_0 - (2a_1 + a_0)(2b_1 + b_0)$.
- Divide the specification polynomial with respect to polynomials indicated by AIG gates following a reverse topological order.

$$\begin{aligned} sp \xrightarrow{f_{s_3}} r_1 :=& 8g_{11} + 4s_2 + 2s_1 + s_0 \\ & - (2a_1 + a_0)(2b_1 + b_0) \\ r_1 \xrightarrow{f_{g_{11}}} r_2 :=& 8g_4 g_7 + 4s_2 + 2s_1 + s_0 \\ & - (2a_1 + a_0)(2b_1 + b_0) \\ r_2 \xrightarrow{f_{s_2}} r_3 :=& 8g_4 g_7 + 4 - 4g_{12} + 2s_1 + s_0 \\ & - (2a_1 + a_0)(2b_1 + b_0) \\ & \cdots \end{aligned}$$

- Check the remainder. If the remainder is zero, then the multiplier is correct. Otherwise, it is incorrect.

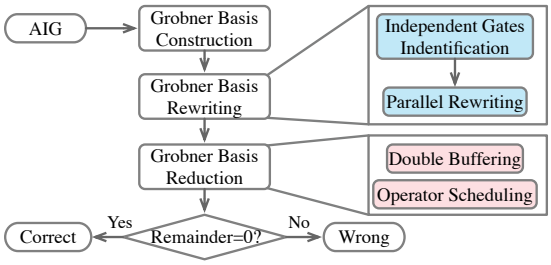

Fig. 2 The overall flow of our acceleration framework.

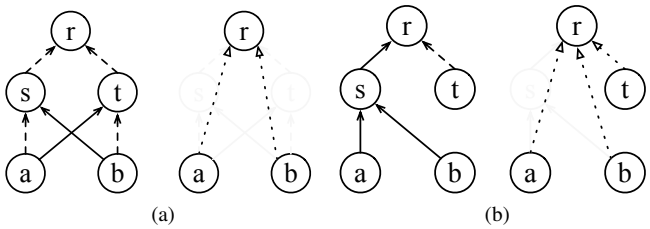

Fig. 3 Examples of Gröbner basis rewriting. (a) XOR-Rewriting (b) Common-Rewriting.

III. Acceleration Framework

The entire flow of our acceleration framework is illustrated in Fig. 2. Within this section, we will provide a comprehensive overview of both the parallel Gröbner basis rewriting and the memory optimization techniques during the Gröbner basis reduction phase.

A. Parallel Gröbner Basis Rewriting

The objective of Gröbner basis rewriting is to acquire a new basis with fewer variables, which aids in preventing the blow-up of monomials. In [11], fanout-free cones of the circuit are extracted and represented as corresponding polynomials. [4] further enhanced fanout-free rewriting by XOR-Rewriting and Common-Rewriting. XOR-Rewriting removes all variables that are neither an input nor an output of an XOR-gate. Notably, primary inputs and primary outputs are left unaltered. As shown in Fig. 3(a), s and t are neither an input nor an output of an XOR gate and are therefore eliminated. Before rewriting, the polynomial representation of gate r is $f_r := -r+1-r-s+st$, which depends on s and t. After rewriting, the polynomial representation becomes $f_r := -r+a+b-2ab$, which depends on a and b. Afterward, Common-Rewriting further simplifies the Gröbner basis by eliminating all gates that only possess a single fanout. In Fig. 3(b), gate r is the only fanout of s. Thus s is eliminated from Gröbner basis. The polynomial representation of gate r changes from $f_r := -r+s-st$ to $f_r := -r+ab-abt$, which no longer depends on s. Other rewriting methods include extracting half-adder and full-adder specifications mentioned in [12].

After Gröbner basis rewriting, some nodes in the AIG are eliminated and a new graph describing the dependencies of the remaining variables is obtained. Here we give a definition of the newly derived graph.

Definition 6. *A Rewritten Gröbner Basis Dependency Graph (RGBDG) is a directed graph describing the dependency of variables after Gröbner basis rewriting.*

The second graph in Fig. 3(a) and Fig. 3(b) are both examples of RGBDG. The dashed lines in an AIG indicate complement, while the dashed lines in an RGBDG indicate the dependency of variables.

All existing rewriting strategies aim to derive a simplified Gröbner basis beneficial for subsequent specification polynomial reduction. However, none emphasize accelerating the time-consuming rewriting phase, which can become particularly burdensome when dealing with millions of gates. We observe that the elimination of certain variables can operate independently of others. We take the circuit in Fig. 1 as an example in which two XOR gates are highlighted. The elimination of (g_5, g_6) and (g_9, g_{10}) are independent of each other and thus can be done in parallel. Consequently, applying multiple parallel threads for simultaneous Gröbner basis rewriting could significantly decrease runtime.

To avoid data race and inconsistent results, the gates that need to be eliminated will be divided into independent groups before parallel Gröbner basis rewriting. The whole algorithm is shown in Algorithm 1, which can be generalized to any rewriting technique.

Algorithm 1 Independent Gates Identification

Input: Multiplier AIG $\mathcal{G}$.
Output: Set of independent groups of gates $\mathcal{S}$.
$\mathcal{S} \leftarrow \emptyset$;
for gate $n \in \mathcal{G}$ **do**
    `set_visited`$(n) \leftarrow 0$;
for $n \in \mathcal{G}$ following reverse topological order **do**
    $\mathcal{D} \leftarrow \emptyset$;
    Initialize an empty queue Q;
    if !`need_elim`(n) or `get_visited`(n) **then**
        `continue`;
    Q.`push`(n);
    while !Q.`empty`() **do**
        $g \leftarrow Q$.`pop`();
        if `get_visited`(g) **then** `continue`;
        `set_visited`$(g) \leftarrow 1$;
        if `need_elim`(g) **then**
            $\mathcal{D} \leftarrow \mathcal{D} \cup g$;
            for each $p \leftarrow$ `parent`(g) **do**
                if !`is_output`(p) **then** `continue`;
                Q.`push`(p);
        for each $c \leftarrow$ `child`(g) **do**
            if !`is_input`(c) **then** `continue`;
            Q.`push`(c);
    `sort`$(\mathcal{D})$ following reverse topological order;
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{D}$;

Here we assume the gates that require eliminating have already been identified before conducting the algorithm. All gates within the given AIG $\mathcal{G}$ are initially flagged as not visited. This step prepares the algorithm to track which gates have been examined during the search procedure. We subsequently iterate over the gates following reverse topological order. When a gate, say n, is marked for elimination and has not been visited, we

employ a set $\mathcal{D}$ to hold gates that should be eliminated and are dependent on n. Simultaneously, a queue Q is initialized with n and used to monitor interrelated gates. We then process each gate in Q until it gets empty. If the front gate, denoted as g, has been visited, the algorithm proceeds to the next gate. If not, it is tagged as visited. If gate g is flagged for elimination, we incorporate it into $\mathcal{D}$ and add its parent gates, excluding outputs, to Q. We also add the children gates of g into $\mathcal{D}$ if they are not inputs. Upon emptying Q, we know that gates in $\mathcal{D}$ are related and cannot be eliminated in parallel. As the gates in $\mathcal{D}$ are unsorted, we still need to sort the gates by reverse topological order before adding to $\mathcal{S}$. At last, the gate groups $\mathcal{D}_1, \mathcal{D}_2, \cdots$ in $\mathcal{S}$ are independent, allowing for their parallel elimination.

It's important to note that we only consider the parent gates of g if g needs to be eliminated (line 15). However, regardless of whether g is flagged for removal, we still examine its child gates. Consider the first graph of Fig. 3(b). Assume we're tracing the interrelated gates of a, and we've just dequeued s. We also suppose gates a, b, and r are tagged for elimination. Even though r is set for elimination, it remains independent of a if s does not need to be eliminated. However, we should always take into account gate b, irrespective of whether s requires elimination or not, because the polynomial representation of s relies on both a and b.

After independent gates are identified, we can leverage multiple threads to rewrite the Gröbner basis in parallel.

B. Memory Optimization for Gröbner Basis Reduction

Procedure of Gröbner Basis Reduction: Consider a node α in the RGBDG, and let T_α be the set of its predecessors. Then this node can be mapped to a polynomial in the simplified Grönber basis $f_\alpha := -\alpha + h(T_\alpha)$, where $h(T_\alpha)$ is a function of T_α. As mentioned in Section II-B, we utilize polynomial division for Gröbner basis reduction. Due to the special property of f_α , dividing it from sp can be done by substituting α with $h(T_\alpha)$ in sp. For example, suppose we desire to reduce $sp = c_1x_1x_2x_3 + \cdots + c_2x_2x_4$ through a polynomial in Gröbner basis $f_{x_2} = -x_2 + c_3x_5x_6$. The reduction process can be accomplished by the following steps:

- Identify all terms containing the variable x_2 in sp, divide x_2 from those terms and add them together to get $quo := c_1x_1x_3 + c_2x_4$.
- Multiply the obtained quo with f_{x_2}, resulting in a polynomial $mul := -c_1x_1x_2x_3 + c_1c_3x_1x_3x_5x_6 - c_2x_2x_4 + c_2c_3x_4x_5x_6$.
- Finally, add mul to sp to cancel all terms containing the variable x_2, which are $c_1x_1x_2x_3$ and $c_2x_2x_4$.

The reduction phase can be illustrated using a computation graph, where the operations of term division, polynomial multiplication, and polynomial addition are represented as nodes. In contrast, the connections between nodes depict the dataflow. Fig. 4(a) demonstrates how to reduce sp_1 to sp_3 through two polynomials f_1 and f_2 from the simplified Gröbner basis.

Double Buffering: The specification polynomial is updated for every iteration during the Gröbner basis reduction phase.

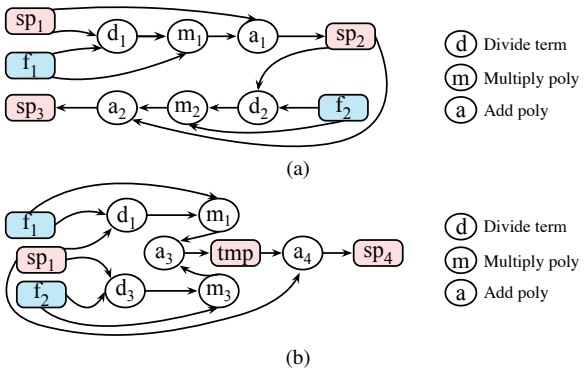

Fig. 4 The procedure of the global reduction phase can be represented by a computation graph. (a) Original computation graph. (b) Computation graph after rescheduling.

Therefore, memory allocation and deallocation are required for the newly derived polynomial after each polynomial division, which can be memory intensive. To address this issue, we implement a double buffering scheme that uses two already-allocated memory buffers to store the reduced specification polynomials.

At the beginning of the reduction process, we cache the first polynomial, sp_1, in the first buffer. After reducing it by f_1, the derived polynomial, sp_2, is stored in the second buffer. At this point, sp_1 is no longer needed, so the first buffer can be used to store the newly derived polynomial, sp_3. By alternately storing the reduced polynomials in the two buffers, we avoid the need for frequent memory allocation and deallocation.

Operator Scheduling: After the Gröbner basis rewriting, each polynomial in the simplified basis is related to a node in RGBDG, and the reduction process also follows a reverse topological order of RGBDG. In a node list following a reverse topological order, two consecutive nodes being adjacent in the list does not necessarily imply a direct edge between them. Similarly, the reduction of two polynomials in Gröbner basis does not have to follow chronological order because the first can be independent of the second one. Suppose we have $f_1 := -\alpha + h(T_\alpha)$ and $f_2 := -\beta + h(T_\beta)$ in Fig. 4. If $\alpha \notin T_\beta$, $\beta \notin T_\alpha$ and $\forall u \in sp_1, u \xrightarrow{\alpha\beta} r \neq 0$, where u is a monomial in sp_1, then the reduction of f_1 and f_2 can be performed concurrently. More specifically, the computation graph can be rescheduled as Fig. 4(b).

Theorem 4. *The two computation graphs in Fig. 4 are equivalent.*

Proof. Without losing generality, we assume $sp_1 = g_1 + c_i\alpha p + g_2 + c_j\beta q + g_3$, where g_1, g_2 and g_3 are sub-polynomials of sp_1 that do not contain neither variable α nor β. p and q are monomials. c_i and c_j are coefficients. Since $\forall u \in sp_1, u \xrightarrow{\alpha\beta} r \neq 0$, we know that $\beta \notin p$ and $\alpha \notin q$. It is easy to see that

$$sp_3 = sp_2 + m_2 = sp_1 + m_1 + m_2. \tag{1}$$

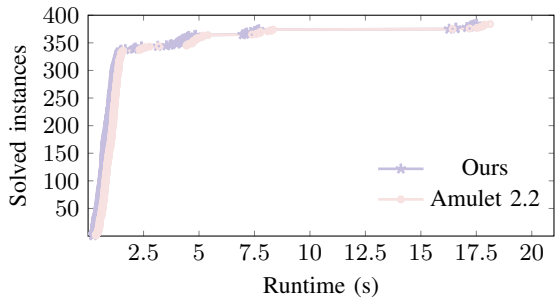

Fig. 5 Verification runtime comparison with Amulet 2.2 on AOKI benchmarks.

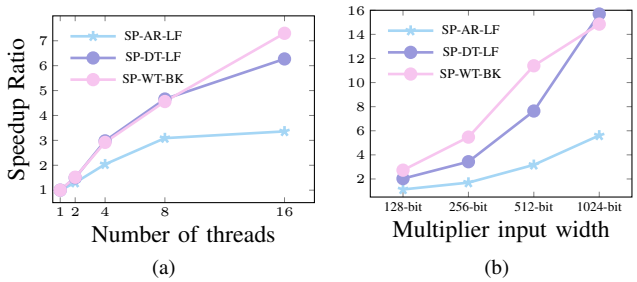

Fig. 6 Scalability of our acceleration framework as number of threads and multiplier input width increases.

Also, we have

$$sp_4 = sp_1 + a_3 = sp_1 + m_1 + m_3. \tag{2}$$

To prove $sp_3 = sp_4$, we only need to show $m_2 = m_3$. Following the Gröbner basis reduction procedure, we can get

$$sp_2 = g_1 + c_i[h(T_\alpha)]p + g_2 + c_j\beta q + g_3. \tag{3}$$

As $\beta \notin h(T_\alpha)$ and $\beta \notin p$, we conclude that β only appears in $c_j\beta q$. Thus, $d_2 = c_j q$ and

$$m_2 = d_2 f_2 = -c_j\beta q + c_j[h(T_\beta)]q. \tag{4}$$

Similarly, as $\alpha \notin T_\beta$ and $\alpha \notin q$, from Fig. 4(b) we can derive that

$$m_3 = d_3 f_2 = -c_j\beta q + c_j[h(T_\beta)]q, \tag{5}$$

which is equal to m_2. □

In the Gröbner basis reduction phase, the number of monomials in a reduced specification polynomial is generally much larger than that in a Gröbner basis polynomial. Therefore, most computation cost and memory footprint come from operations whose operands contain a reduced specification polynomial. The most significant difference between the two computation graphs originates from operations a_1 and a_3. While for other operators in Fig. 4(a), we can find a corresponding operator in Fig. 4(b) with similar computation cost and memory footprint. For instance, a_2 and a_4, d_2 and d_3 have similar cost. The operands of a_3 are both small polynomials, while one of the operands of a_1 is a large polynomial, sp_1. As a result, the computation cost of a_3 is smaller than a_1. Also, the derived polynomial tmp has a smaller memory overhead than sp_2. The given computation graphs only investigate two polynomials from the simplified Gröbner basis. However, it can be easily extended to scenarios when we can apply multiple polynomials from the simplified Gröbner basis to reduce the specification polynomial simultaneously.

IV. Results

A. Setup

Currently, no single verification tool outperforms others in all benchmarks. For example, RevSAC 2.0 [9] is more robust against design optimization, while Amulet 2.2 [13] performs better when final stage adders can be detected. Our acceleration framework works with any state-of-the-art verification tool as long as it contains the Gröbner basis rewriting and reduction stage. In order to assess the effectiveness of our techniques, we have specifically integrated our acceleration framework into Amulet 2.2 due to its availability as an open-source tool. Amulet 2.2 uses a global hash table for internal term sharing and global buffers to store intermediate results of terms and polynomials. Consequently, these initial data structures were not designed for parallel computing. To rectify this, we revised the implementations by incorporating local buffers, which enabled parallel polynomial calculations. We conducted all experiments on a Linux server outfitted with 125G DRAM and two Intel(R) Xeon(R) Silver 4210R CPUs. Each CPU clocks at 2.40 GHz and has 10 physical cores.

B. Experimental Results

First, our experiments utilized the AOKI benchmarks, consisting of 384 mixed signed and unsigned integer multipliers, reaching up to 64-bit. The comparative verification time with Amulet 2.2 is depicted in Fig. 5. We only applied the memory optimization techniques because small multipliers don't benefit from multithreading. Both Amulet 2.2 and our optimized tool were able to verify all the multipliers. Nonetheless, our enhanced acceleration framework verified a higher number of multipliers given an identical time budget.

We also evaluated the effectiveness of the proposed techniques using up to 1024-bit size multipliers produced by the GenMul multiplier generator [14]. The benchmarks include three different multiplier architectures, where detailed information can be found in [14]. We only conducted experiments on unsigned multipliers because the verification time is similar as long as the architecture and the input width are equal. The generated designs were synthesized to gate-level netlists through Yosys [15] before being transformed into AIG via ABC [16]. The experimental results are documented in TABLE I without considering the adder substitution time. Considering we applied different acceleration methods for Gröbner basis rewriting and Gröbner basis reduction, we listed the runtime separately. The total runtime slightly surpasses the combined time of rewriting and reduction because pre-processing and post-processing time are also counted. We can see that for the 1024×1024 multiplier with architecture `SP-DT-LF`, the speedup ratio can reach more than $15\times$.

TABLE I Verification runtime comparison with Amulet 2.2 on large multipliers.

architecture	size	#gates	Amulet 2.2 [13]			Ours (16 threads)		
			rewriting (s)	reduction (s)	overall (s)	rewriting (s)	reduction (s)	overall (s)
SP-AR-LF	128×128	194314	0.98	0.16	1.30	0.43	0.57	1.15
SP-DT-LF		193806	2.26	0.38	2.82	0.45	0.82	1.39
SP-WT-BK		197774	2.31	2.65	5.24	0.48	1.26	1.92
SP-AR-LF	256×512	781834	6.20	0.87	7.72	2.37	1.52	4.55
SP-DT-LF		780814	17.85	2.09	20.80	1.79	3.57	6.06
SP-WT-BK		790610	17.84	25.85	44.66	1.86	5.63	8.16
SP-AR-LF	512×512	3136522	55.42	5.70	63.81	10.94	6.97	20.13
SP-DT-LF		3134478	185.53	12.02	201.13	8.28	15.22	26.33
SP-WT-BK		3157890	186.46	322.87	512.96	8.84	33.05	45.02
SP-AR-LF	1024×1024	12564490	506.55	39.39	573.05	54.26	37.41	102.11
SP-DT-LF		12560398	1817.96	92.74	1940.23	38.32	73.96	123.68
SP-WT-BK		12606714	1807.25	3519.13	5356.14	37.65	311.19	360.71
Average Ratio			15.64	2.80	6.24	1.00	1.00	1.00

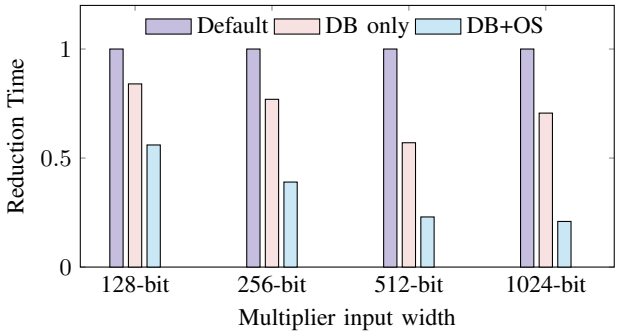

Fig. 7 Gröbner basis reduction time with/without memory optimization techiniques. "Default" refers to the original implementation. "DB" is short for double buffering and "OS" is short for operator scheduling.

C. Ablation Study

Scalability: To begin with, we increased the number of threads and recorded the speedup ratio for Gröbner basis rewriting on the largest 1024-bit multipliers. The results are displayed in Fig. 6(a). Additionally, we explored how the overall speedup ratio scales as the multiplier input width increases. As depicted in Fig. 6(b), we can see that our proposed techniques perform better as the multipliers become larger with 16 threads.

Effectiveness of Memory Optimization: We also investigated the impact of double buffering and operator scheduling on the time required for Gröbner basis reduction. The normalized runtime shown in Fig. 7 demonstrates that both memory optimization techniques can significantly decrease the Gröbner basis reduction time.

V. Conclusion and Future Work

This paper proposes a novel approach to accelerate SCA-based verification systems for integer multipliers. By leveraging parallel computing and a double buffering scheme with an operator scheduler, the proposed techniques result in a significant speedup compared to the original implementation. Our future work includes integrating our acceleration framework into more verification tools like RevSCA 2.0 [9] and exploring a new Gröbner basis rewriting method that can be beneficial for parallel computing.

References

[1] H. Sharangpani and M. Barton, "Statistical analysis of floating point flaw in the pentiumtm processor (1994)," *Intel Corporation*, vol. 30, 1994.

[2] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits using binary moment diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 3, pp. 137–155, 2001.

[3] A. Biere, "Collection of combinational arithmetic miters submitted to the SAT competition 2016," *SAT Competition*, vol. 2016, p. 1, 2016.

[4] 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 *Proc. DATE*, 2016, pp. 1048–1053.

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

[6] D. Kaufmann, A. Biere, and M. Kauers, "Incremental column-wise verification of arithmetic circuits using computer algebra," *Formal Methods in System Design*, vol. 56, pp. 22–54, 2020.

[7] ——, "Verifying large multipliers by combining SAT and computer algebra," in *Proc. FMCAD*, 2019, pp. 28–36.

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

[9] A. Mahzoon, D. Große, and R. Drechsler, "Revsca-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *IEEE TCAD*, vol. 41, no. 5, pp. 1573–1586, 2021.

[10] D. Cox, J. Little, and D. OShea, *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*, 2013.

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

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

[13] D. Kaufmann and A. Biere, "Fuzzing and Delta Debugging And-Inverter Graph Verification Tools," in *International Conference on Tests and Proofs*, 2022, pp. 69–88.

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

[15] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013, p. 97.

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