Towards Verifying Nonlinear Integer Arithmetic

Paul Beame⋆ and Vincent Liew∗

Computer Science & Engineering, University of Washington, Seattle WA 98195

Abstract. We eliminate a key roadblock to efficient verification of nonlinear integer arithmetic using CDCL SAT solvers, by showing how to construct short resolution proofs for many properties of the most widely used multiplier circuits. Such short proofs were conjectured not to exist. More precisely, we give $n^{O(1)}$ size regular resolution proofs for arbitrary degree 2 identities on array, diagonal, and Booth multipliers and $n^{O(\log n)}$ size proofs for these identities on Wallace tree multipliers.

1 Introduction

Recent decades have seen remarkable advances in our ability to verify hardware. Methods for hardware verification based on Ordered Binary Decision Diagrams (OBDDs) developed in the 1980s for hardware equivalence testing [16] were extended in the 1990s to produce general methods for symbolic model checking [18] to verify complex correctness properties of designs. More recently, several orders of magnitude of improvements in the efficiency of SAT solvers have brought new vistas of verification of hardware and software within reach.

Nonetheless, there is an important area of formal verification where roadblocks that were identified in the 1980s still remain: verification of data paths within designs for Arithmetic Logic Units (ALUs), or indeed any verification problem in hardware or software that involves the detailed properties of nonlinear arithmetic. Natural examples of such verification problems in software include computations involving hashing or cryptographic constructions. At the highest level of abstraction, nonlinear arithmetic over the integers is undecidable, but the focus of these verification problems is on the decidable case of integers of bounded size, which is naturally described in the language of bit-vector arithmetic (see, e.g. [32,30]).

In particular, a notorious open problem is that of verifying properties of integer multipliers in a way that both is general purpose and avoids exponential scaling in the bit-width. Bryant [17] showed that this is impossible using OBDDs since they require exponential size in the bit-width just to represent the middle bit of the output of a multiplier. This lower bound has been improved [11] and extended to include very tight exponential lower bounds for much more general diagrams than OBDDs, and FBDDs [36,10] and general bounded-length branching programs [38]. With the flexibility of CNF formulas, efficient representation

⋆ Research supported in part by NSF grant CCF-1524246.

of multipliers is no longer a problem but, even with the advent of greatly improved SAT solvers, there has been no advance in verifying multipliers beyond exponential scaling.

One important technique for verifying software and hardware that includes multiplication has been to use methods of uninterpreted functions to handle multipliers (see [14,32]) – essentially converting them to black boxes and hoping that there is no need to look inside to check the details. Another important technique has been to observe that it is often the case that one input to a multiplier is a known constant and hence the resulting computation involves linear, rather than nonlinear arithmetic. These approaches have been combined with theories of arithmetic (e.g. [12,35,15,13]), including preprocessors that do some form of rewriting to eliminate nonlinear arithmetic, but these methods are not able, for example, to check the details of a multiplier implementation or handle nonlinearity.

Though the above approaches work in some contexts, they are very limited. The approach of verifying code with multiplication using uninterpreted functions is particularly problematic for hashing and cryptographic applications. For example, using uninterpreted functions in the actual hash function computation inherently can never consider the case that there is a hash collision, since it only can infer equality between terms with identical arguments. Concern about such applications is real: longstanding errors in multiplication in OpenSSL have recently come to light [34].

Recent presentations at verification conferences and workshops have highlighted the problem of verifying nonlinear arithmetic, and multipliers in particular, as one of the key gaps in our current verification methods [6,7,29,9].

Since bit-vector arithmetic is not itself a representation in Boolean variables, in order to apply SAT solvers to verify the designs, one must convert implementations and specifications to CNF formulas based on specified bit-widths. The process by which one does this is called *flattening* [32], or more commonly *bit-blasting*. The resulting CNF formulas are then sent to the SAT solvers. While the resulting bit-blasted CNF formulas for a multiplier may grow quadratically with the bit-width, this growth is not a significant problem. On the other hand, a major stumbling block for handling even modest bit-widths is the fact that existing SAT solvers run on these formulas experience exponential blow-up as the bit-width increases. This is true even for the best of recent methods, e.g., Boolector [13], MathSAT [15], STP [27], Z3 [25], and Yices [24].

In verifying a multiplier circuit one could try to compare it to a reference circuit that is known to be correct. This introduces a chicken-and-egg problem: how do we know that the reference circuit is correct? Another approach to verifying a multiplier circuit is to check that it satisfies the right properties. A correct multiplier circuit must obey the multiplication identities for a commutative ring. If we check that each of these *ring identities* holds then the multiplier cannot have an error. This approach has the advantage that the specification of a multiplier circuit can be written *a priori* in terms of its natural properties, rather than in terms of an external reference circuit.

Empirically, however, modern SAT-solvers perform badly using either approach to problems of multiplier verification. Biere, in the text accompanying benchmarks on the ring identities submitted to the 2016 SAT Competition [8] writes that when given as CNF formulas, no known technique is capable of handling bit-width larger than 16 for commutativity or associativity of multiplication or bit-width 12 for distributivity of multiplication over addition. These observations lead to the question: is the difficulty inherent in these verification problems, or are modern SAT-solvers just using the wrong tools for the job?

Modern SAT-solvers are based on a paradigm called conflict-directed clause-learning (CDCL) [33] which can be seen as a way of breaking out of the backtracking search of traditional DPLL solvers [23]. When these solvers confirm the validity of an identity (by not finding a counterexample), their traces yield *resolution* proofs [4] of that identity. The size of such a proof is comparable to the running time of the solver; hence finding short resolution proofs of these identities is a necessary prerequisite for efficient verification via CDCL solvers. Although it is not known whether CDCL solvers are capable of efficiently simulating every resolution proof, all cases where short resolution proofs are known have also been shown to have short CDCL-style traces (e.g., [20,19,21]).

The extreme lack of success of general purpose solvers (in particular CDCL solvers) for verifying any non-trivial properties of bit-vector multiplication, recently led Biere to conjecture [9] that there is a fundamental proof-theoretic obstacle to succeeding on such problems; namely, verifying ring identities for multiplication circuits, such as commutativity, requires resolution proofs that are exponential in the bit-width n.

We show that such a roadblock to efficient verification of nonlinear arithmetic does not exist by giving a general method for finding short resolution proofs for verifying *any* degree 2 identity for Boolean circuits consisting of bit-vector adders and multipliers. This method is based on reducing the multiplier verification to finding a resolution refutation of one of a number of narrow *critical strips*. We apply this method to a number of the most widely used multiplier circuits, yielding $n^{O(1)}$ size proofs for array, diagonal, and Booth multipliers, and $n^{O(\log n)}$ size proofs for Wallace tree multipliers.

These resolution proofs are of a special simple form: they are *regular* resolution proofs[1]. Regular resolution proofs have been identified in theoretical models of CDCL solvers as one of the simplest kinds of proof that CDCL solvers naturally express [20]. Indeed, experience to date has been that the addition of some heuristics to CDCL suffices to find short regular resolution proofs that we know exist. The new regular resolution proofs that we produce are a key step towards developing such heuristics for verifying general nonlinear arithmetic.

[1] Some of these proofs are even more restricted *ordered* resolution proofs, also known as *DP* proofs, which are associated with the original Davis-Putnam procedure [22]. In contrast to the Davis-Putnam procedure, which eliminates variables one-by-one keeping all possible resolvents, ordered resolution (or DP) proofs only keep some minimal subset of these resolvents needed to derive a contradiction.

Related work: SAT solver-based techniques used in conjunction with case splitting previously were shown to achieve some success for certain multiplier verification problems in the work of Andrade et al. [3] improving on earlier work [2,37] which combined SAT solver and OBDD-based ideas for multiplier verification among other applications; however, there was no general understanding of when such methods will succeed.

Recently, two alternative approaches to multiplier verification have been considered: Kojevnikov [28] designed a mixed Boolean-algebraic solver, BASolver, that takes input CNF formulas in standard format. It uses algebraic rules on top of a DPLL solver. Though it can verify the equivalence of multipliers up to 32 bits in a reasonable time, in each instance it requires human input in order to find a suitable set of algebraic rules to help the solver. An alternative approach using Groebner basis algorithms has been considered [39]. This is a purely algebraic approach based on polynomials. Since the language of polynomials allows one to explicitly write down the algebraic specification for an n-bit multiplier, the verification problem is conveniently that of checking that the multiplier circuit computes a polynomial equivalent to the multiplier specification. [39] shows that Groebner basis algorithms can be used to verify 64-bit multipliers in less than ten minutes and 128-bit multipliers in less than two hours. However, this requires that the multipliers be identified and treated entirely separately from the rest of the circuit or software. Unfortunately, for the non-algebraic parts of circuits, Groebner basis methods can only handle problems several orders of magnitude smaller than can be handled by CDCL SAT-solvers and it remains to be seen whether it is possible to combine these to obtain effective verification for a general purpose software with nonlinear arithmetic or circuits that contain a multiplier as just one component of their design. In contrast, CDCL SAT solvers are already very effective for the non-algebraic aspects of circuits and are well-suited to handling the combination of different components; our work shows that there is no inherent limitation preventing them from being effective for verification of general purpose nonlinear arithmetic.

Roadmap: In Section 3 we introduce and give constructions of some standard multipliers, in particular the array multiplier and the Wallace tree multiplier. We give polynomial size regular resolution proofs for degree 2 identities for circuits constructed using array, diagonal, and Booth multipliers in Section 4 and give quasipolynomial size regular resolution proofs for circuits constructed using Wallace tree multipliers in Section 5.

2 Notation and Preliminaries

We represent Boolean variables in lowercase and denote clauses by uppercase letters and think of them as sets of literals, for example $C = \{x, \bar{y}, z\}$. We will work with length n *bit-vectors* of variables, denoted by $\mathbf{z} = z_{n-1} \dots z_1 z_0$.

We consider identities from the commutative ring of integers $\mathbb{Z}$. We use a set $\sigma = \sigma(x_0, x_1 \dots x_n) = \{x_0 = b_0, x_1 = b_1 \dots x_n = b_n\}$, where each $b_i \in \{0, 1\}$, to denote an assignment to the variables $x_0, x_1, \dots x_n$.

Definition 1. *A* commutative ring $(\mathcal{R}, \oplus, \otimes, 0, 1)$ *consists of a nonempty set* $\mathcal{R}$ *with addition* ($\oplus$) *and multiplication* ($\otimes$) *operators that satisfy the following properties:*

1. *($\mathcal{R}$,$\oplus$) is associative and commutative and its identity element is* 0.
2. *For each* $\mathbf{x} \in \mathcal{R}$ *there exists an* additive inverse*.*
3. *($\mathcal{R}$,$\otimes$) is associative and commutative and its identity element is* $1 \neq 0$.
4. *(distributivity) For all* $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathcal{R}$*,* $\mathbf{x} \otimes (\mathbf{y} \oplus \mathbf{z}) = (\mathbf{x} \otimes \mathbf{y}) \oplus (\mathbf{x} \otimes \mathbf{z})$.

A ring identity $L = R$ *denotes a pair of expressions* L, R *that can be transformed into each other using commutativity, distributivity and associativity.*

Note that both verifying integer $\oplus$ circuits and verifying that $\mathbf{x} \otimes \mathbf{1} = \mathbf{x}$ are easy in practice, so verifying an integer multiplier circuit $\otimes$ can be easily reduced to verifying its distributivity.

Definition 2. *A* resolution proof *consists of a sequence of clauses, each of which is either a clause of the input formula* ϕ*, or follows from two prior clauses via the* resolution rule *which produces clause* $C \vee D$ *from clauses* $C \vee x$ *and* $D \vee \overline{x}$*. We say that this inference* resolves *the clauses* on x*. The proof is a* refutation *of* ϕ *if it ends with the empty clause* $\perp$*. (With resolution we will use the terms "proof" and "refutation" interchangeably, since resolution provides proofs of unsatisfiability.)*

We can naturally represent a resolution proof P as a directed acyclic graph (DAG) of fan-in 2, with $\perp$ labelling the lone sink node. *Tree resolution* is the special subclass of resolution proofs where the DAG is a directed tree. Another restricted form of resolution is *regular resolution*: A resolution refutation is *regular* iff on any path in its DAG the inferences resolve on each variable at most once. The shortest tree resolution proofs are always regular. An *ordered* resolution refutation is a regular resolution refutation that has the further property that the order in which variables are resolved on along each path is consistent with a single total order of all variables. This is a very significant restriction and indeed the shortest tree resolution proofs do not necessarily have this property.

We will find it convenient to express our regular resolution proofs in the form of a *branching program* that solves the *conflict clause search problem.*

Definition 3. *Suppose that* ϕ *is an unsatisfiable formula. Then every assignment* σ *to its variables conflicts with some clause in* ϕ*. The* conflict clause search *problem is to map any assignment to some corresponding conflicting clause.*

Definition 4. *A branching program* B *on the Boolean variables* $X = \{x_0, x_1, \ldots\}$ *and output set* ϕ *(typically a set of clauses in this paper) is a finite directed acyclic graph with a unique source node and sink nodes at its leaves, each leaf labeled by an element from* ϕ*. Each non-sink node is labeled by a variable from* X *and has two outgoing edges, one labeled* 0 *and the other labeled* 1*. An assignment* σ activates *an edge labeled* $b \in \{0, 1\}$ *outgoing from a node labeled by the variable* x_i *if* σ *contains the assignment* $x_i = b$*. If* σ *activates a path from the source to a sink labeled* $C \in \phi$*, we say that the branching program* B outputs C*.*

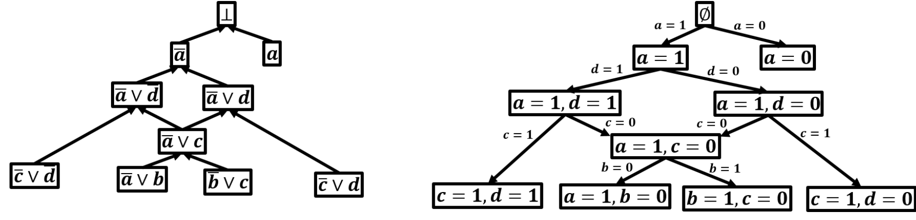

Fig. 1: A regular resolution refutation and the corresponding branching program.

A read-once branching program *(also known as a Free Binary Decision Diagram, or FBDD) is a branching program where each variable is read at most once on any path from source to leaf. An* Ordered Binary Decision Diagram (OBDD) *is a special case of an FBDD in which the variables read along any path are consistent with a single total order.*

The general case of the following theorem connecting regular resolution proofs and conflict clause search is due to Krajicek [31]; the special case connecting ordered resolution and OBDDs for the conflict clause search problem, which is a simple observation extending the original proof, does not seem to have been explicitly noted previously.

Theorem 1. *Let ϕ be an unsatisfiable formula. A regular resolution refutation R for ϕ of size s corresponds to a size s read-once branching program that solves the conflict clause search problem for ϕ.*

Suppose that B is a read-once branching program of size s solving the conflict clause search problem for ϕ. Then there is a regular resolution refutation for ϕ of size s.

Furthermore, if R is an ordered resolution refutation then the resulting branching program is an OBDD and if B is an OBDD then the resulting resolution refutation is an ordered resolution refutation.

Proof. (Sketch) In this equivalence, the nodes of the read-once branching program are in one-to-one correspondence with the clauses in the proof and the edges of the read-once branching program correspond to the inference connections in the proof. That is, clauses $A \vee x$ and $B \vee \overline{x}$ resolve to yield clause C if and only if the node corresponding to C is labeled by x, its 0-outedge goes to the node corresponding to $A \vee x$ and its 1-outedge goes to the node corresponding to $B \vee \overline{x}$.

Moving from the proof to the branching program is immediate. For the reverse direction we label each node with the maximal clause that is falsified by every assignment reaching that node. It is not hard to check that all the needed properties are satisfied.

In our proofs we write each clause as the assignment it forbids. For example we write the clause $\{x, \bar{y}\}$ as the assignment $\{x = 0, y = 1\}$. We will build up

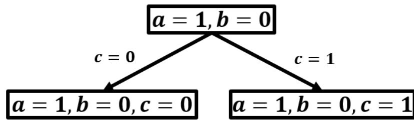

Fig. 2: Branching on c.

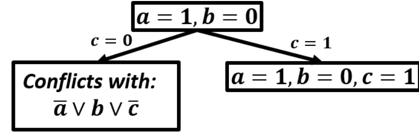

Fig. 3: Propagating to $c = 1$.

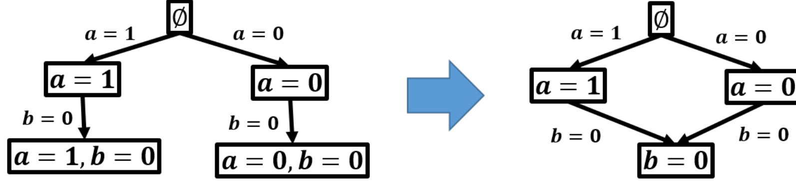

Fig. 4: Merging on the common assignment $\{b = 0\}$.

branching programs for conflict clause search in ϕ in terms of three types of action, shown in Figures 2, 3, 4. At a node labeled by an assignment $\sigma \not\ni z$, we *branch* on the variable z by adding a child node with assignment $\sigma \cup \{z = 0\}$, connected by a 0-labeled edge, and another child node $\sigma \cup \{z = 1\}$, connected by a 1-labeled edge. In the case that one of these children has an assignment conflicting with a clause $C \in \phi$, we say that we *propagated* the assignment σ to the other child's assignment. Lastly, for a set of leaf nodes with assignments $\sigma_0, \sigma_1, \ldots$ we can *merge* their branches based on a common assignment $\sigma \subset \cap_i \sigma_i$ by replacing these nodes with a single node labeled by σ.

3 Multiplier Constructions

We describe our SAT instances as a set of constraints, where each constraint is a set of clauses. We build circuits out of *adders* that output, in binary, the sum of three input bits. An adder is encoded as follows:

Definition 5. *Suppose a_0, a_1, a_2 are inputs to an adder A. The outputs c, d of the adder A are encoded by the constraints:*

$$d = a_0 \oplus a_1 \oplus a_2 \qquad c = MAJ(a_0, a_1, a_2)$$

We call c and d the sum-bit *and c* carry-bit *respectively. If an adder has two constant inputs 0 it acts as a* wire*. If it has precisely one constant input 0, we call it a* half adder*. If no inputs are constant, we call it a* full adder*.*

Each circuit variable in our constructions has a *weight* of the form 2^i. Each adder will take in three bits of the same weight 2^i and output a *sum-bit* of weight 2^i and a *carry-bit* of weight 2^{i+1}. The adder's definition ensures that the weighted sum of its input bits is the same as the weighted sum of its output bits. In the constructions that follow, we divide the adders up into columns so that the i-th column contains all the adders with inputs of weight 2^i.

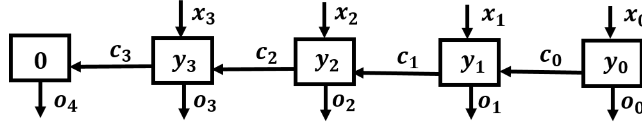

Fig. 5: 4-bit ripple-carry adder adding $\mathbf{x}, \mathbf{y}$. Each box represents a full adder with incoming arrows and outgoing arrows representing inputs and outputs.

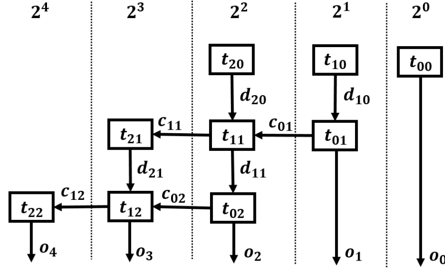

Fig. 6: Array multiplier tableau.

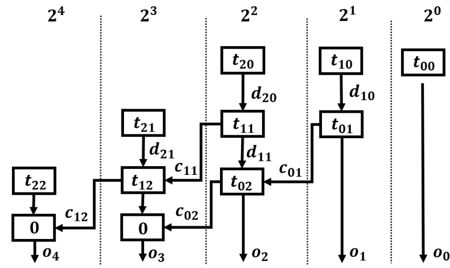

Fig. 7: Diagonal multiplier tableau.

Ripple-Carry Adder: A ripple-carry adder, shown in Figure 5, takes in two bitvectors $\mathbf{x}, \mathbf{y}$ and outputs their sum in binary. In the i-th column, for $i \leq n$, we place an adder A_i that takes the three variables c_{i-1}, x_i, y_i and outputs the adder's carry variable and sum variable to c_i and o_i respectively. In the $n+1$-st column we place a wire A_{n+1} taking c_n as input and outputting to o_{n+1}.

All the multipliers we describe perform two phases of computation to compute $\mathbf{xy}$ for length n bitvectors $\mathbf{x}, \mathbf{y}$. The first phase is the same in each multiplier: the circuit computes a *tableau* of values $x_i \wedge y_j$ for each pair of input bits x_i and y_j. In the second phase where the constructions differ, the circuit computes the weighted sum of the bits in the tableau.

Array Multiplier: An n-bit array multiplier works by arranging n ripple-carry adders in sequence in order to sum the n rows of the tableau. This multiplier has a simple gridlike architecture that is compact and easy to lay out physically. It has depth linear in its bitwidth. In the first phase, an array multiplier computes each tableau variable $t_{ij} = x_i \wedge y_j$, with associated weight 2^{i+j}.

Arrange a grid of full adders $A_{i,j}$, where $i, j \in [0, n]$, as shown in Figure 6. Adder $A_{i,j}$ occupies the j-th row and the $i+j$-th column, takes inputs $t_{i,j}, d_{i+i,j-1}, c_{i-1,j}$ (replacing nonexisting variables with the constant 0), and outputs the carry and sum bits $c_{i,j}$ and $d_{i,j}$. Finally, we add constraints equating the sum-bits $d_{0,0}, d_{0,1}, \ldots, d_{0,n-1}, d_{1,n-1}, \ldots, d_{n-1,n-1}$ with the corresponding output bits $o_0, o_1, \ldots, o_{2n-1}$.

Diagonal Multiplier and Booth Multiplier: A diagonal multiplier uses a similar idea to the array multiplier. The difference is that the diagonal multiplier

routes its carry bits to the next row instead of the same row as depicted in Figure 7.

A Booth multiplier uses a similar idea to the array multiplier, but uses a telescoping sum identity to skip consecutive digits in one multiplicand.

Wallace Tree Multiplier: A Wallace tree multiplier takes a different approach to summing the tableau. Using carry-save adders (parallel 1-bit adders), it iteratively finds a new tableau with the same weighted sum as the previous tableau, but with 1/3 fewer rows. Upon reducing the original tableau to just two rows, it uses a carry-lookahead adder to obtain the final result. In contrast to the array multiplier, a Wallace tree multiplier is complicated to lay out physically, but has only logarithmic depth.

4 Efficient Proofs for Degree Two Array Multiplier Identities

We give polynomial-size resolution proofs that commutativity, distributivity, and $x(x+1) = x^2 + x$ hold for a correctly implemented array multiplier. We go on to give polynomial-size resolution proofs for general degree two identities.

Proof Overview: The main idea, common to our proofs for each circuit family including Wallace tree multipliers, is to start by branching according to the lowest order disagreeing output bit between the two circuits. In each of these branches, the subcircuit known as a *critical strip* consists of the constraints on a small number of columns behind the disagreeing bit. For a large enough choice of width this critical strip is unsatisfiable since the removed section of the tableau on the right does not have enough total weight to cause the disagreeing output bit. It then remains to refute each critical strip.

Our proofs inside each critical strip cycle through three steps: branching on the values of input bits, propagating those values as far in the circuit as possible, then saving the resulting assignment to the variables on the boundary of the propagation. We call each of these boundaries a *cut* in the circuit. We can think of the branching program as scanning the input bits, saving its progress with assignments to small cuts.

These *cuts* are sets of variables that, under any assignment, split the strip into a satisfiable and an unsatisfiable region. If our branching program holds a cut assignment that was propagated from an earlier portion of the circuit, then this cut assignment is consistent with this earlier subcircuit. But since the critical strip as a whole is unsatisfiable, this cut assignment must be inconsistent with the rest of the circuit. Using these cuts, we isolate a small unsatisfiable region in the critical strip that is easily refuted.

One can view our proof as showing that the constraints within each strip form a graph of *pathwidth* $O(\log n)$ which, by [26], implies that there is a polynomial-size ordered resolution refutation of the strip. In the case of commutativity, our argument implies that the constraint graphs for the strips can be combined to

yield a single constraint graph of pathwidth $O(\log n)$. For the other identities, the orderings on the strips are different and the resulting constraint graphs only have small *branchwidth* which, by [1], still implies that there are small regular resolution proofs of the other identities. Rather than simply invoke these general arguments, we give the details of the resolution proofs, along with more precise size bounds.

4.1 Efficient Resolution Proofs for Commutativity

Definition 6. *We define a SAT instance* $\phi^{\text{Array}}_{\text{Comm}}(n)$*. The inputs are length* n *bitvectors* $\mathbf{x}, \mathbf{y}$*. Using the construction from Section 3, we define array multipliers* L *and* R *computing, respectively, the products* $\mathbf{xy}$ *and* $\mathbf{yx}$*. In this construction, the tableau variables in multipliers* L *and* R *are defined by the constraints*

$$t^L_{i,j} = x_i \wedge y_j, \qquad t^R_{i,j} = y_i \wedge x_j,$$

and in particular we can infer, through resolution, that $t^L_{i,j} = t^R_{j,i}$*.*

After specifying the subcircuits L *and* R*, we add a final subcircuit* E*,a set of* inequality-constraints *encoding that the two circuits disagree on some output bit:*

$$e_i = (\neg o^L_i \wedge o^R_i) \vee (o^L_i \wedge \neg o^R_i) \quad \forall i \in [0, 2n-1],$$

$$e_0 \vee e_1 \vee \ldots e_{2n-1}.$$

We give a small resolution proof for $\phi^{\text{Array}}_{\text{Comm}}(n)$ in the form of a labeled OBDD B, as described in Theorem 1. The variable order for B begins with the comparison bits in the order $e_0, e_1, \ldots$, followed by the output bits $o^R_0, o^R_1, \ldots$. Then B reads the variables associated with adders $A^L_{i,j}, A^R_{j,i}$ in order of increasing j, reading each row right to left. Finally, B reads the output bits $o^L_0, o^L_1, \ldots$, then the input bits $\mathbf{x}, \mathbf{y}$ in an arbitrary order.

At the root of B, we search for the first output bit that L and R disagree on by branching on the sequences of bits $e_k = 1, e_{k-1} = 0, \ldots e_0 = 0$ for each $k \in [0, 2n]$. We will show that on each branch we can prove that $\phi^{\text{Array}}_{\text{Comm}}(n)$ is unsatisfiable using only the constraints from L and R on the variables inside columns $[k - \log n, k]$.

Definition 7. *Let* $\delta = \log n$*. Let* $\phi_{\text{Strip}}(k) \subset \phi^{\text{Array}}_{\text{Comm}}(n)$ *hold the constraints containing a tableau variable* $t^L_{i,j}$ *or* $t^R_{i,j}$ *for* $i + j \in [k - \delta, k]$*. Further, add unit clauses to* $\phi_{\text{Strip}}(k)$ *that encode the assignment:* $e_0 = 0, e_1 = 0, \ldots, e_{k-1} = 0, e_k = 1$ *to the first* k *bits of* $\mathbf{e}$*. We call* $\phi_{\text{Strip}}(k)$ *a* critical strip *of* $\phi^{\text{Array}}_{\text{Comm}}(n)$*. We call the subset* $\phi_{\text{Strip}}(k) \cap L$ *the* critical strip *of circuit* L *and likewise for circuit* R*.*

Lemma 1. $\phi_{\text{Strip}}(k)$ *is unsatisfiable for all* k*.*

Proof. We interpret each critical strip as a circuit that outputs the weighted sum of the input variables in either circuit L, R. The assignment to $\mathbf{e}$ demands that the difference between the critical strip outputs is precisely 2^k. But by $t^L_{i,j} = t^R_{j,i}$,

the weighted sum of the tableau variables is the same in both critical strips. The difference in the critical strip outputs is then bounded by the sum of the input carry bits to column $k - \delta$ in either strip. There are fewer than n input carry bits for either critical strip, each of weight $2^{k-\delta} = 2^k/n$, therefore the difference in critical strip outputs is less than 2^k, violating the assignment to $\mathbf{e}$.

Observe that this proof only relied on the relation $t^L_{ij} = t^R_{ji}$ in the tableau variables. The additional requirement that the tableau variables came from an assignment to $\mathbf{x}, \mathbf{y}$ is unnecessary to refute $\phi_{\text{Strip}}(k)$.

Lemma 2. *There is an $O(n^7 \log n)$-sized ordered resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof. For simplicity we assume $k \leq n$; the case where $k > n$ is similar. We will also preprocess $\phi_{\text{Strip}}(k)$ by resolving on $\mathbf{x}, \mathbf{y}$ to obtain the tableau variable relations $t^R_{ji} = t^L_{ij}$, then replacing all the variables t^R_{ji} by t^L_{ij} in the clauses $\phi_{\text{Strip}}(k)$. Viewing the proof as a branching program, this amounts to querying $\mathbf{x}, \mathbf{y}$ at the end. We will not resolve on $\mathbf{x}, \mathbf{y}$ in the remainder of this proof.

We give this resolution proof in the form of a labeled read-once branching program B. We call the set of tableau variables of L, as well as the carry variables from column $k - \delta - 1$ of both L and R, the *input variables* to this critical strip. An assignment to the input variables, denoted by σ_{input}, determines the outputs of L and R.

The idea behind the branching program B is to verify circuit L by branching on input variables row-by-row, going from top-to-bottom, maintaining an assignment to a row of sum-variables. Since $t^L_{ij} = t^R_{ji}$, the tableau variables of circuit R simultaneously get revealed from bottom to top. In circuit R we maintain both a guess for its output values, and a row of sum-variables. From the proof of Lemma 1, if we have found that the outputs of L and R were computed correctly then they must violate one of the constraints $e_k = 0, \ldots, e_{k-\delta+1} = 0, e_{k-\delta} = 1$.

Definition 8. *Define* $\text{Cut}(0)$ *as the set of variables containing*

$$d^R_{0,i}, o^R_{i-1} \quad \textit{for} \quad i - 1 \in [k - \delta, k].$$

For $j \in [1, k - \delta - 1]$, we define $\text{Cut}(j)$ *to be the set containing the variables:*

$$\begin{aligned} d^L_{i,j-1}, d^R_{j,i-1} \quad &\textit{for} \quad i + j - 1 \in [k - \delta, k], \\ c^R_{j-1,i} \quad &\textit{for} \quad i + j - 1 \in [k - \delta, k - 1], \\ o^R_i \quad &\textit{for} \quad i \in [k - \delta, k]. \end{aligned}$$

Lastly, for $j \in [k - \delta, k]$, we define $\text{Cut}(j)$ *to be the set containing the variables, when the indices are in-range:*

$$\begin{aligned} o^L_i \quad &\textit{for} \quad i \in [k - \delta, j - 1] \\ d^L_{i+1,j-1}, d^R_{j,i}, c^R_{j-1,i} \quad &\textit{for} \quad i + j \in [k - \delta, k], \\ c^R_{j-1,i} \quad &\textit{for} \quad i + j - 1 \in [k - \delta, k - 1], \\ o^R_i \quad &\textit{for} \quad i \in [k - \delta, k]. \end{aligned}$$

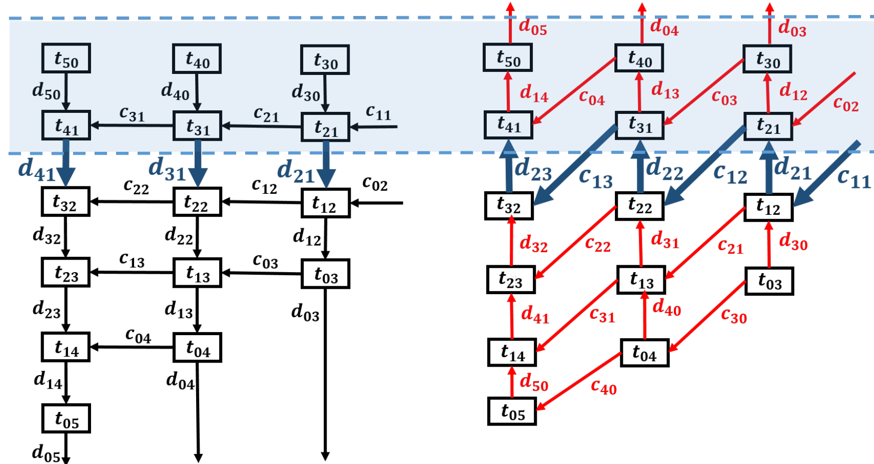

Fig. 8: The blue variables belong to Cut(2) of $\phi_{\text{Strip}}(5)$. This cut divides the critical strip into a shaded satisfiable region and an unshaded unsatisfiable region.

We will label each node of B by the pair $(\text{Cut}(j), \sigma)$ where $\text{Cut}(j)$ keeps track of the previously seen cut.

Initialization: Throughout, we work in terms of the tableau variables in circuit L, implicitly substituting t^L_{ij} for t^R_{ji}. We begin at the root node of the read-once branching program B, labeled $(\emptyset, \emptyset)$. For $i \in [k-\delta, k]$ we branch on the variable o^R_i, then propagate to $d^R_{0,i}$ using the constraint $o^R_i = d^R_{0,i}$. The surviving branches are those labeled by an assignment satisfying the constraints $o^R_i = d^R_{0,i}$. At this point we have reached nodes labeled Cut(0).

For each of the surviving branches, we branch on the tableau variables in the first row of L:

$$t^L_{i,0} \quad \text{for} \quad i \in [k-\delta, k].$$

Then we propagate to the variables, in sequence,

$$d^R_{1,i}, c^R_{0,i} \quad \text{for} \quad i+1 \in [k-\delta, k]$$

from Cut(1) (notice that this does not include the input carry-bit $c^R_{0,k-\delta-1}$). We then merge on Cut(1).

Inductive Step: We now describe the transition from $\text{Cut}(j)$ to $\text{Cut}(j+1)$ for $j \in [1, k]$. Suppose that the branching program B has reached an assignment to $\text{Cut}(j)$. From these nodes we branch on the next, j-th row of tableau variables

$$t^L_{i,j} \quad \text{for} \quad i+j \in [k-\delta, k]$$

and, when they exist, the pair of incoming input carry variables $c^L_{i,j}, c^R_{j-1,i}$ from column $k-\delta-1$, adjacent to the critical strip. We then propagate to the $\text{Cut}(j+1)$ and c^L variables in the sequence:

$$c^L_{i,j}, d^L_{i+1,j} \quad \text{for} \quad i+j+1 \in [k-\delta, k]$$

in circuit L. If $j \in [k-\delta, k]$ then we also propagate to o_{j-1}.

$$c^R_{j,i}, d^R_{j+1,i} \quad \text{for} \quad i+j+1 \in [k-\delta, k]$$

in circuit R. After branching on the last variable in $\text{Cut}(j+1)$ we start labeling nodes by $\text{Cut}(j+1)$ and merge branches on their assignment to $\text{Cut}(j+1)$. This completes the step from $\text{Cut}(j)$ to $\text{Cut}(j+1)$.

We repeat this step until we have reached $\text{Cut}(k+1)$. At this point we have an assignment to the critical strip output bits $\mathbf{o}^L, \mathbf{o}^R$. Furthermore, both output assignments were the result of, and therefore consistent with, propagating from a single assignment on the input variables σ_{inputs}. By the proof of Lemma 1, this implies that our assignment to $\mathbf{o}^L, \mathbf{o}^R$ conflicts with an inequality constraint.

Size Bound: We show that there are $O(n^7 \log n)$ nodes in B. Each $\text{Cut}(j)$ section of B begins with an assignment to at most $4\delta = 4\log n$ variables, so there are at most n^4 nodes labeled by an assignment to precisely $\text{Cut}(j)$. We branch on up to $\log(n) + 2$ input variables that propagate to the next cut's variables. So each cut has a full binary tree of $2 * (2^4 k^2)$ nodes branching on different configurations of input variables. For each leaf of this tree, B has a path of $O(\log n)$ nodes for propagating before the nodes get merged. Therefore each cut labels at most $O(n^6 \log n)$ nodes. There are $k+1$ different cuts, thus B has at most $O((n+1)n^6 \log n) = O(n^7 \log n)$ nodes.

Since the tableau variables were actually partial products of $\mathbf{x}$ and $\mathbf{y}$, we can make this proof smaller by branching on the bits of $\mathbf{x}, \mathbf{y}$ to determine the tableau variables in a row, maintaining a sliding window of δ bits of $\mathbf{x}$, yielding:

Corollary 1. *$\phi_{\text{Strip}}(k)$ has an $O(n^6 \log n)$-size regular resolution refutation.*

Theorem 2. *Let $N = |\phi^{\text{Array}}_{\text{Comm}}| = O(n^2)$. There is an $O(N^{7/2} \log N)$ size regular resolution proof that $\phi^{\text{Array}}_{\text{Comm}}$ is unsatisfiable. There is an $O(N^4 \log N)$ size* ordered *resolution proof that $\phi^{\text{Array}}_{\text{Comm}}$ is unsatisfiable.*

Proof. We can now describe the overall branching program B for $\phi^{\text{Array}}_{\text{Comm}}(n)$. The branching program branches on the inequality-constraint assignments $\sigma_e(k) = \{e_k = 1, e_{k-1} = 0, \ldots e_0 = 0\}$ for $k \in [0, 2n-1]$. The k-th branch contains the clauses $\phi_{\text{Strip}}(k)$ so we can use the read-once branching program from either Corollary 1 or Lemma 2 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. Corollary 1 yields the regular resolution proof and Lemma 2 yields the ordered resolution proof.

4.2 Efficient Resolution Proofs for Distributivity

Definition 9. *We define a SAT instance $\phi^{\text{Array}}_{\text{Dist}}(n)$ to verify the distributivity property*

$$x(y+z) = xy + xz$$

for an array multiplier in the natural way, using the constructions from Section 3. For the left hand expression we construct a ripple-carry adder L_{y+z}, outputting

$(\mathbf{y}+\mathbf{z})$, and array multiplier $L_{x(y+z)}$ outputting $\mathbf{x}(\mathbf{y}+\mathbf{z})$. For the right hand expression, we similarly define circuits R_{xz}, R_{xy} and R_{xy+xz}.

We define $L = L_{y+z} \cup L_{x(y+z)}$ and $R = R_{xz} \cup R_{xy} \cup R_{xy+xz}$. We let E contain the usual inequality constraints. The full distributivity instance is then $\phi^{\text{Array}}_{\text{Dist}}(n) = L \cup R \cup E$.

We follow a similar strategy to the one used to refute $\phi^{\text{Array}}_{\text{Comm}}$. We again divide the instance into critical strips.

Definition 10. *Define the constant $\delta = \log(2n)$. Define the subset $\phi_{\text{Strip}}(k) \subset \phi^{\text{Array}}_{\text{Dist}}(n)$ to include, firstly, the full ripple-carry adder circuit L_{y+z}. Secondly, include the constraints containing one of the tableau variables $t^{L_{x(y+z)}}_{i,j}, t^{R_{xy}}_{i,j}, t^{R_{xz}}_{i,j}$ for $i+j \in [k-\delta, k]$. Thirdly, include the constraints on the carry-bits and sum-bits $c^{R_{xy+xz}}_i, d^{R_{xy+xz}}_i$ for $i \in [k-\delta, k]$. Lastly, add constraints to $\phi_{\text{Strip}}(k)$ that encode the assignment the bits: $e_k = 1, e_{k-1} = 0, \ldots, e_0 = 0$.*

Lemma 3. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k*

Proof. (Sketch) Similarly to the proof of Lemma 1, the critical strip for $L_{x(y+z)}$ holds tableau bits with the same weighted sum (modulo 2^{k+1}) as those in R_{xz} and R_{xy} combined. The critical strip for $L_{x(y+z)}$ has at most n input carry-bits of weight $2^{k-\delta}$. The critical strips of the n-bit multipliers R_{xz} and R_{xy} each have at most $n-1$ input carry variables of weight $2^{k-\delta}$. The critical strip of the adder R_{xy+xz} has one input carry variable, so the critical strip for R has $2n-1$ input carry-bits. Since we set the width of the strip at $\delta = \log(2n)$, it is unsatisfiable.

Lemma 4. *For each k there is an $O(n^4 \log n)$ size regular resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof. (Sketch) We construct a labeled branching program B that solves the conflict clause search problem for $\phi_{\text{Strip}}(k)$. The idea is similar to the one for the commutativity instance. We branch row-by-row in the critical strips, maintaining an assignment to cuts of variables in each multiplier. For each strip we will select a (different) variable ordering for $\mathbf{x}, \mathbf{y}, \mathbf{z}$ that reveals the tableau variables row-by-row. Assume that $k < n$ for simplicity; the case where $k \geq n$ is similar.

For an array multiplier $C \in \{L_{x(y+z)}, R_{xz}, R_{xy}\}$ and $j \in [1, k-\delta]$ we define $\text{Cut}^C(j)$ to be the set of variables

$$d^C_{i,j-1} \quad \text{for} \quad i+j-1 \in [k-\delta, k],$$

and for $j \in [k-\delta+1, k]$ we define $\text{Cut}^C(j)$ as the set of variables

$$\begin{aligned} d^C_{i,j-1} &\quad \text{for} \quad i+j-1 \in [k-\delta, k], \\ o^C_i &\quad \text{for} \quad i \in [k-\delta, j-2] \end{aligned}$$

We define $\text{Cut}^{L_{y+z}}(j)$ to be the singleton set $\{c^{L_{y+z}}_{j-1}\}$. We also refer to a global cut, across the whole circuit: $\text{Cut}(j) = \cup_C \text{Cut}^C(j)$.

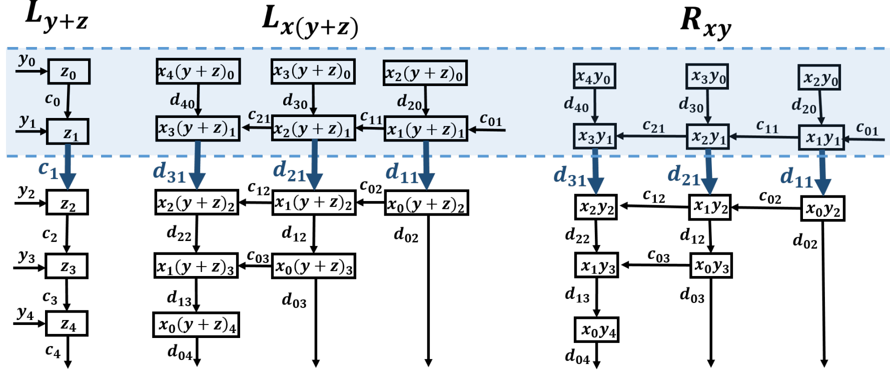

Fig. 9: Cut(2) for $\phi_{\text{Strip}}(4)$ consists of the blue variables.

As in the proof for commutativity, we sweep these cuts down each circuit. At row j, the branching program B will branch on x_{k-j}, y_j, z_j to reveal row j's tableau variables and then merge on the resulting propagated assignments to Cut$(j+1)$.

Size Bound: There are $k+1$ different global cuts Cut(j). Each Cut(j) section of B begins with an assignment to at most $3\delta+1$ variables. So each section Cut(j) is initialized with at most $2^{3\delta+1} = 8n^3$ branches. Each of these branches is a path with at most 10δ queried variables and therefore at most 20δ nodes. So there are at most $200\delta n^3$ nodes per cut and therefore at most $(k+1)200\delta n^3 = O(n^4 \log n)$ nodes in B.

Theorem 3. *Let* $N = |\phi^{\text{Array}}_{\text{Dist}}(n)| = O(n^2)$. *There is an* $O(N^{5/2} \log N)$ *size resolution proof that* $\phi^{\text{Array}}_{\text{Dist}}(k)$ *is unsatisfiable.*

Proof. At the root of this proof there are $2n$ branches each holding an assignment to $e_k, \ldots, e_1, e_0$. We refute each branch using the $O(n^4 \log n)$ size proof from Lemma 4.

4.3 Efficient Resolution Proofs for $x(x+1) = x^2 + x$

Define the SAT instance $\phi^{\text{Array}}_{x(x+1)}(n)$ for the identity $x(x+1) = x^2 + x$ in the usual way, with multipliers $L_{x(x+1)}$, R_{x^2} and ripple-carry adder L_{x+1}. While this identity looks like a special case of distributivity, its resolution proof is more complicated. This is because for distributivity: $x(y+z) = xy + xz$, the inputs to each multiplier were separate variables. This allowed us to scan the critical strip from one end to the other in a read-once fashion. If we try a similar strategy to scan the critical strip for the multiplier R_{x^2} from top to bottom, we will read each x_i twice. To avoid reading the same variable twice, we instead scan the critical strip from both ends, meeting in the middle. This strategy yields the following theorem, whose details can be found in the full version of this paper. [5]

Theorem 4. *Let* $N = |\phi^{\text{Array}}_{x(x+1)}(n)| = O(n^2)$*. There is a size* $O(N^4 \log N)$ *regular resolution proof that* $\phi^{\text{Array}}_{x(x+1)}(n)$ *is unsatisfiable.*

4.4 Efficient Resolution Proofs for Degree Two Identities

Let $\phi^{\text{Array}}_{L=R}(n)$ denote a SAT instance corresponding to verifying the ring identity $L = R$. Similarly define $\phi^{\text{Diag}}_{L=R}(n)$ and $\phi^{\text{Booth}}_{L=R}(n)$ denote the diagonal and Booth multiplier versions. With the insight from the earlier proofs in this section, we can prove the general theorem:

Theorem 5. *For any degree two ring identity* $L = R$*, there are polynomial size regular refutations for* $\phi^{\text{Array}}_{L=R}(n)$*.*

Proof. (Sketch) We divide $\phi^{\text{Array}}_{L=R}(n)$ into unsatisfiable critical strips of width $\delta = \log mn$, where m is the number of terms in the identity $L = R$. The ripple-carry adders that input to a multiplier remain intact, and for the rest we remove the columns outside the critical strip.

We begin by branching on guesses for the δ output bits from each multiplier and each truncated ripple-carry adder. In each multiplier we use a "meet-in-the-middle" strategy, similar to the proof for $x(x+1) = x^2 + x$. We read all the input bitvectors in parallel, each in the same order. This branch order for each input bitvector $\mathbf{x}$ is $x_0, x_n, x_1, x_{n-1}, \ldots$. We can propagate these assignments to diagonal cuts in each multiplier that scan from the top and bottom edges towards the middle, and likewise for the intact ripple-carry adders. In each input bitvector we remember the assignment to just the most recently queried 2δ variables. Because of the symmetry of this variable order, it is compatible with swapping the order of inputs to any multiplier, as well as multipliers squaring an input.

Corollary 2. *For any degree two ring identity* $L = R$*, there are polynomial size regular resolution proofs for* $\phi^{\text{Diag}}_{L=R}(n)$ *and* $\phi^{\text{Booth}}_{L=R}(n)$

Proof. The proofs for diagonal and Booth multipliers follow the same ideas as the proof for an array multiplier.

5 Wallace Tree Multipliers

Like the proofs for array multipliers, our proofs for Wallace tree multipliers divide the instance into critical strips. In fact, our proofs branch on the input tableau in the same row-by-row order in both array and Wallace tree multipliers. However the size of the resulting cuts is $O(\log^2 n)$ for Wallace tree multipliers rather than the $O(\log n)$ size cuts for array multipliers. These cuts result in quasipolynomial size regular resolution proofs. The details of these resolution proofs and our Wallace tree multiplier construction are described in the full version of the paper [5].

Definition 11. *Let the SAT instance $\phi^{\text{Wall}}_{\text{Comm}}(n)$ encode two n-bit Wallace tree multipliers computing* $\mathbf{xy}$ *and* $\mathbf{yx}$*, as well as the usual inequality constraints. Similarly let $\phi^{\text{Wall}}_{\text{Dist}}(n)$ correspond to verifying distributivity. Finally, let the SAT instance $\phi^{\text{Wall}}_{L=R}(n)$ correspond to verifying the ring identity $L = R$.*

Theorem 6. *The instance $\phi^{\text{Wall}}_{\text{Comm}}(n)$ has a size $O(n^{3\log n+3})$ regular resolution refutation. The instance $\phi^{\text{Wall}}_{\text{Dist}}(n)$ has a size $O(n^{4.5\log n+2})$ regular refutation.*

Using the same ordering on the input variables and ideas from the proof of Theorem 5, we can prove the analogous result for Wallace tree multipliers.

Theorem 7. *For any degree two ring identity $L = R$, there are quasipolynomial size regular refutations for $\phi^{\text{Wall}}_{L=R}(n)$.*

6 Proving Equivalence Between Multipliers

Given any two n-bit multiplier circuits $\otimes_1$ and $\otimes_2$ we can define a Boolean formula $\phi_{\otimes_1=\otimes_2}$ encoding the (negation of the) identity $\mathbf{x}\otimes_1\mathbf{y} = \mathbf{x}\otimes_2\mathbf{y}$ between length n bitvectors $\mathbf{x}$ and $\mathbf{y}$.

If they are both correct and compute using the typical tableau for multipliers then, as before, we can split $\phi_{\otimes_1=\otimes_2}$ into unsatisfiable critical strips. We can scan down both strips row-by-row, as in the proofs for commutativity and distributivity. If we have reached the outputs of both multipliers without finding an error, these outputs will disagree with the inequality-constraints for the critical strip. For our examples this method yields polynomial-size proofs if neither is a Wallace tree multiplier, and quasi-polynomial size proofs otherwise.

On the other hand, if one multiplier is incorrect and the other is not, then the proof search will yield a satisfying assignment in the appropriate critical strip.

In the more general case where a multiplier does not use the typical tableau, one can label each internal gate by the index of the smallest output bit to which it is connected and focus on comparing subcircuits labeled by $O(\log n)$ consecutive output bits, as we do with critical strips. The complexity of this equivalence checking will depend somewhat on the similarity of the circuits involved.

7 Discussion

Despite significant advances in SAT solvers, one of their key persisting weaknesses has been in equivalence checking of arithmetic circuits. This pointed towards the conjecture that that the corresponding resolution proofs are exponentially large; if true, this would have been a fundamental obstacle putting nonlinear arithmetic out of reach for any CDCL SAT solver. We have shown that no such obstacle exists by giving the first small resolution proofs for verifying any degree two ring identity for the most common multiplier designs. Given the historical success of CDCL SAT solvers for finding specific proofs, our results suggest a new path towards verifying nonlinear arithmetic.

We introduced a method of dividing each instance into narrow, but still unsatisfiable, critical strips that is sufficiently general to yield short proofs for a wide variety of popular multiplier designs. The proof size upper bounds we derived were conservative; we did not try to optimize the parameters. A more important direction than doing so is to find the right guiding information to add, either to the formulas derived from the circuits or to CDCL SAT solver heuristics, to help them find such short proofs.

It also remains open to find a small resolution proof verifying the last ring property, associativity $(xy)z = x(yz)$. Our critical strip idea alone does not seem to work: while we can divide the outer multipliers into narrow critical strips, the yz or xy multipliers remain intact. These critical strips do not seem to have small cuts. Finding efficient proofs of associativity, combined with our results for degree two identities, could yield small proofs of any general ring identity.

References

1. Michael Alekhnovich and Alexander A. Razborov. Satisfiability, branch-width and tseitin tautologies. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), Proceedings*, pages 593–603, Vancouver, BC, Canada, November 2002. IEEE Computer Society.
2. Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In *Proceedings of the 39th Design Automation Conference, DAC 2002*, pages 725–730, New Orleans, LA, USA, June 2002. ACM.
3. Fabrício Vivas Andrade, Márcia C. M. Oliveira, Antônio Otávio Fernandes, and Claudionor José Nunes Coelho Jr. Sat-based equivalence checking based on circuit partitioning and special approaches for conflict clause reuse. In Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak, and Tomasz Garbolino, editors, *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007)*, pages 397–402, Kraków, Poland, April 2007. IEEE Computer Society.
4. Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.
5. Paul Beame and Vincent Liew. Towards verifying nonlinear integer arithmetic. *CoRR*, abs/1705.0XXXX, 2017.
6. Armin Biere. Challenges in bit-precise reasoning. In *Formal Methods in Computer-Aided Design, FMCAD 2014*, page 3, Lausanne, Switzerland, October 2014.
7. Armin Biere. Where does SAT not work? In *BIRS Workshop on Theory and Applications of Applied SAT Solving*, January 2014. http://www.birs.ca/events/2014/5-day-workshops/14w5101/videos/watch/201401201634-Biere.html.
8. Armin Biere. Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. University of Helsinki, 2016.
9. Armin Biere. Weaknesses of CDCL solvers. In *Fields Institute Workshop on Theoretical Foundations of SAT Solving*, August 2016. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.

10. B. Bollig and P. Wooelfel. A read-once branching program lower bound of $\Omega(2^{n/4})$ for integer multiplication using universal hashing. In *Proceedings of the Thirty-Third Annual ACM Symposium on the Theory of Computing*, pages 419–424, Hersonissos, Crete, Greece, July 2001.
11. Beate Bollig. Larger lower bounds on the OBDD complexity of integer multiplication. *Inf. Comput.*, 209(3):333–343, 2011.
12. Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the ASPDAC 2002 / VLSI Design 2002*, pages 741–746, Bangalore, India, January 2002.
13. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009*, pages 174–177, 2009.
14. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT($\{\mathcal{BV}\}$) solver for hard industrial verification problems. In *Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007*, pages 547–560, Berlin, Germany, July 2007.
15. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4SMT solver. In *Proceedings, Computer Aided Verification, 20th International Conference, CAV 2008*, pages 299–303, 2008.
16. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
17. Randal E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.
18. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, 1994.
19. Samuel R. Buss and Maria Luisa Bonet. An improved separation of regular resolution from pool resolution and clause learning. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7313 of *Lecture Notes in Computer Science*, pages 244–57, Trento, Italy, June 2012.
20. Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. *Logical Methods in Computer Science*, 4(4), 2008.
21. Samuel R. Buss and Leszek Kolodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10(2), 2014.
22. M. Davis and H. Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.
23. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
24. Leonardo Mendonça de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 2005.
25. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, pages 337–340, 2008.
26. Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In Eric Horvitz and Finn Verner Jensen, editors, *UAI '96: Proceedings of the*

Twelfth Annual Conference on Uncertainty in Artificial Intelligence, pages 211–219, Portland, OR, USA, August 1996. Morgan Kaufmann.
27. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007*, pages 519–531, Berlin, Germany, July 2007.
28. Edward Hirsch, Dmitry Itsykson, Arist Kojevnikov, Alexander Kulikov, and Sergey Nikolenko. Report on the mixed boolean-algebraic solver. *Technical report, Laboratory of Mathematical Logic of St. Petersburg Department of Steklov Institute of Mathematics*, 2005.
29. Priyank Kalla. Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation. In *Proceedings, Formal Methods in Computer-Aided Design, FMCAD*, page 2, Austin, TX, September 2015.
30. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.
31. J. Krajíček. *Bounded Arithmetic, Propositional Logic and Complexity Theory.* Cambridge University Press, 1996.
32. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer, 2008.
33. João P. Marques-Silva, Ines Lynce, and Sharad Malik. CDCL solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 4, pages 131–154. IOS Press, 2009.
34. Openssl.org. Openssl bug cve-2016-7055, 2016.
35. Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. An efficient finite-domain constraint solver for circuits. In *Proceedings of the 41th Design Automation Conference, DAC*, pages 212–217, 2004.
36. S Ponzio. A lower bound for integer multiplication with read-once branching programs. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 130–139, Las Vegas, NV, May 1995.
37. Sherief Reda and A. Salem. Combinational equivalence checking using boolean satisfiability and binary decision diagrams. In Wolfgang Nebel and Ahmed Jerraya, editors, *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001*, pages 122–126, Munich, Germany, March 2001. IEEE Computer Society.
38. M. Sauerhoff and P. Woelfel. Time-space tradeoff lower bounds for integer multiplication and graphs of arithmetic functions. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 186–195, San Diega, CA, June 2003.
39. Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In Luca Fanucci and Jürgen Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016*, pages 1048–1053, Dresden, Germany, March 2016. IEEE.