Toward Verifying Nonlinear Integer Arithmetic

PAUL BEAME and VINCENT LIEW, University of Washington

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.

CCS Concepts: • **Theory of computation → Proof complexity**; • **Hardware → Theorem proving and SAT solving**; *Equivalence checking*;

Additional Key Words and Phrases: Multiplier verification, SAT solvers, resolution proofs

ACM Reference format:
Paul Beame and Vincent Liew. 2019. Toward Verifying Nonlinear Integer Arithmetic. *J. ACM* 66, 3, Article 22 (June 2019), 30 pages.
https://doi.org/10.1145/3319396

1 INTRODUCTION

The last few decades have seen remarkable advances in our ability to verify hardware and software. Methods for hardware verification based on Ordered Binary Decision Diagrams (OBDDs) developed in the 1980s for hardware equivalence testing (Bryant 1986) were extended in the 1990s to produce general methods for symbolic model checking (Burch et al. 1994) 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

A preliminary version of this work appeared in *Proceedings of the 29th International Conference on Computer-Aided Verification (CAV 2017), Part II*, pp. 238–258, 2017.
This research was supported by NSF-AF grant CCF-1524246 and NSF-SHF grant CCF-1714593.
Authors' addresses: P. Beame and V. Liew, Paul G. Allen School of Computer Science & Engineering, University of Washington, Box 352350, Seattle, Washington, 98195-2350; emails: {beame, vliew}@cs.washington.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0004-5411/2019/06-ART22 $15.00
https://doi.org/10.1145/3319396

in the language of bit-vector arithmetic (see, e.g., Kroening and Strichman (2008) and Kovásznai et al. (2016)).

In particular, a notorious open problem is that of verifying properties of integer multipliers in a way that is both general enough to handle a wide variety of multiplier implementations, and avoids exponential scaling in the bit-width. Bryant (1991) 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 (Bollig 2011) and extended to include very tight exponential lower bounds for much more general diagrams than OBDDs, including FBDDs (Ponzio 1995; Bollig and Wooelfel 2001) and general bounded-length branching programs (Sauerhoff and Woelfel 2003). On the other hand, CNF formulas can efficiently represent multipliers, 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 Bruttomesso et al. (2007) and Kroening and Strichman (2008))—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., Brinkmann and Drechsler (2002), Parthasarathy et al. (2004), Bruttomesso et al. (2008), and Brummayer and Biere (2009)), 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 the correctness of the arithmetic in such applications is real: for example, longstanding errors in multiplication in OpenSSL have recently come to light (Openssl.org 2016).

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 (Biere 2014a, 2014b; Kalla 2015; Biere 2016b).

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* (Kroening and Strichman 2008), 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 seem to experience nearly exponential blow-up as the bit-width increases (Figure 1). As a result, the best of recent bit-vector solvers, e.g., Boolector (Brummayer and Biere 2009), MathSAT (Bruttomesso et al. 2008), STP (Ganesh and Dill 2007), Z3 (de Moura and Bjørner 2008), and Yices (de Moura 2005) all rely on multiple rounds of preprocessing to reduce the expense of bit-blasting as much as possible.

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

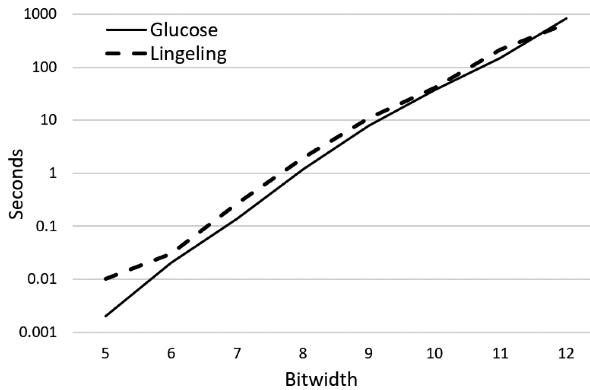

Fig. 1. Time to verify multiplier commutativity versus the bit-width of the multiplier for SAT solvers Glucose and Lingeling (Biere 2016b).

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 (Biere 2016a) writes that when given as CNF formulas, no known technique was 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) (Marques-Silva and Sakallah 1996; Moskewicz et al. 2001), which can be seen as a way of breaking out of the backtracking search of traditional DPLL solvers (Davis et al. 1962). When these solvers confirm the validity of an identity (by not finding a counterexample), their traces yield *resolution* proofs (Beame et al. 2004) 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 (polynomial size) resolution proofs are known have also been shown to have short CDCL-style traces (e.g., Buss et al. (2008), Buss and Bonet (2012), and Buss and Kolodziejczyk (2014)).

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 (Biere 2016b) 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 (Buss et al. 2008). 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 toward developing such heuristics for verifying general nonlinear arithmetic.

Related Work. SAT solver-based techniques used in conjunction with case splitting previously were shown to achieve some success for multiplier verification in the work of Andrade et al. (Andrade et al. 2007) improving on earlier work (Andersson et al. 2002; Reda and Salem 2001) 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: Hirsch et al. (2005) 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 (Sayed-Ahmed et al. 2016). 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. Sayed-Ahmed et al. (2016) shows that Groebner basis algorithms can be used to verify 64-bit multipliers in less than 10 minutes and 128-bit multipliers in less than 2 hours. One drawback of algebraic methods is that they require 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 non-linear 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.

Finally, independently of and in parallel with our results, there has also been further work on refining Groebner basis methods (Ritirc et al. 2017). We postpone discussing that refinement until after we have presented our results.

Roadmap. Section 3 gives our polynomial size regular resolution proofs for array multipliers. Section 4 describes how to extend these ideas to obtain short proofs for diagonal and Booth multipliers. Section 5 gives our quasipolynomial size regular resolution proofs for Wallace tree multipliers.

[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 (Davis and Putnam 1960). 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.

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 $\mathbf{z} = z_{n-1} \ldots z_1 z_0$.

Ring Identities

We consider identities from the commutative ring of integers $\mathbb{Z}$. A variable assignment is denoted by a set $\sigma = \sigma(x_0, x_1 \ldots x_n) = \{x_0 = b_0, x_1 = b_1 \ldots x_n = b_n\}$, where each $b_i \in \{0, 1\}$. $x_0, x_1, \ldots x_n$.

Definition 2.1. 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 the correctness of an integer multiplier circuit $\otimes$ can be easily reduced to verifying its distributivity.

Proposition 2.2. *Given a circuit $\oplus$ and another circuit $\otimes$, if $\oplus$ correctly implements integer addition, $\mathbf{x} \otimes \mathbf{1} = \mathbf{x}$, and $\otimes$ is distributive over $\oplus$, then $\otimes$ correctly implements integer multiplication.*

Proof. $\mathbf{x} \otimes \mathbf{y} = \mathbf{x} \otimes \underbrace{(\mathbf{1} \oplus \mathbf{1} \ldots \oplus \mathbf{1})}_{\text{y additions}} = \underbrace{\mathbf{x} \oplus \mathbf{x} \ldots \oplus \mathbf{x}}_{\text{y additions}} = (\mathbf{xy})$. □

Resolution Proofs and Branching Programs

Definition 2.3. 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 $\perp$ labeling 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 find it convenient to express our regular resolution proofs in the form of a *branching program* that solves the *conflict clause search problem.*

Definition 2.4. 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.

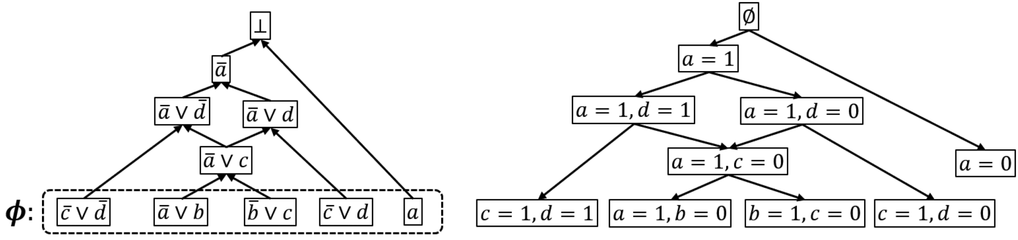

Fig. 2. A regular resolution refutation for ϕ and the corresponding branching program.

Definition 2.5. 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 article) 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.

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 proposition connecting regular resolution proofs and conflict clause search is due to Krajíček (1996) (see Figure 2 for an example); the special case connecting ordered resolution and OBDDs for the conflict clause search problem was first observed in Lovász et al. (1995). We include its proof for completeness.

Proposition 2.6. *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. Suppose that R is a regular resolution refutation of size s for ϕ. Each clause C appearing in R is a node of B. If two clauses $C_0 \vee x, C_1 \vee \bar{x}$ in R resolve on a variable x to produce the clause C, then in the branching program B we branch from the node C on the variable x to reach $C_0 \vee x$ on the $x = 0$ branch, and $C_1 \vee \bar{x}$ on the $x = 1$ branch. The resulting branching program B solves the conflict clause search problem for ϕ and has the same size as the refutation R. The fact that no variable is branched on more than once on any path is immediate from the definition; the fact that this results in an OBDD in the case of ordered resolution is also immediate.

In the other direction, we obtain a regular refutation R from the specified read-once branching program B. We will label each node v with the maximal clause C_v that is falsified by every assignment reaching v. These clauses form the regular resolution refutation. If v is a leaf, then C_v is the conflicting clause from ϕ found by B. If B branches from node v on a variable x to nodes v_0, v_1, then in R we resolve the clauses C_{v_0}, C_{v_1} on x to obtain C_v. Again, the number of clauses in the refutation R is the same as the number of nodes in the branching program B. The fact that the resolution is regular follows immediately from the fact that the branching program is read-once; if the branching program is an OBDD,then it is immediate that the resolution refutation is ordered. □

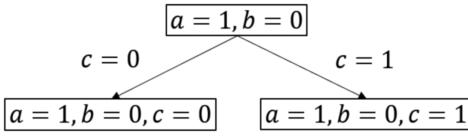

Fig. 3. Branching on c.

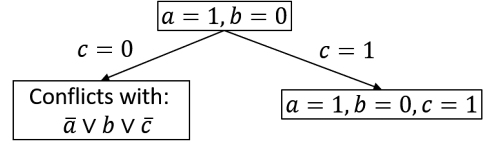

Fig. 4. Propagating to $c = 1$.

Fig. 5. Merging on the common assignment $\{b = 0\}$.

In our proofs, we represent each clause with the partial assignment it forbids. For example, we write the clause $x \vee \bar{y}$ as the partial assignment $\{x = 0, y = 1\}$. A branching program for conflict clause search in ϕ consists of three types of actions, shown in Figures 3–5. At a node labeled by a partial assignment σ that does not include variable z, we *branch* on z by connecting a child node with assignment $\sigma \cup \{z = 0\}$ using 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 $\sigma \subseteq \cap_i \sigma_i$ by replacing these nodes with a single node labeled by σ.

3 ARRAY MULTIPLIERS

3.1 Array Multiplier Construction

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

Definition 3.1. Let a_0, a_1, a_2 be 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 *carry-bit* and d the *sum-bit*. If an adder has two constant 0 inputs, 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 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.

Ripple-Carry Adder. A ripple-carry adder, shown in Figure 6, takes in two bit-vectors $\mathbf{x}, \mathbf{y}$ 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}. While the implementation is simple, it has depth n.

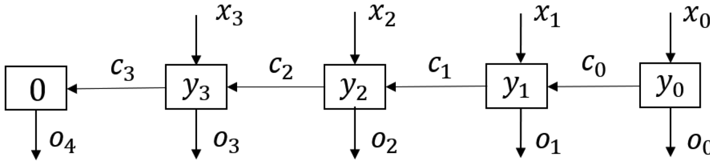

Fig. 6. 4-bit ripple-carry adder adding $\mathbf{x}, \mathbf{y}$. Each box represents a full adder with incoming arrows and outgoing arrows representing inputs and outputs.

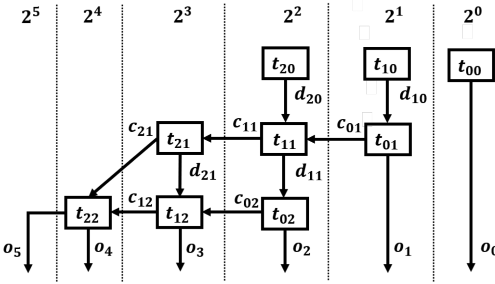

Fig. 7. 3-bit array multiplier.

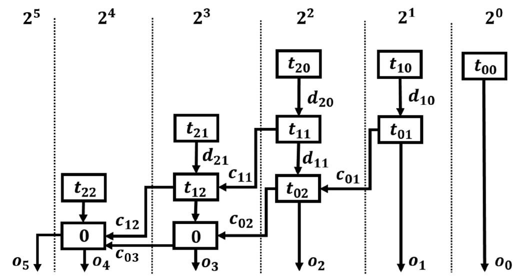

Fig. 8. 3-bit diagonal multiplier.

All the multipliers we describe perform two phases of computation to compute $\mathbf{xy}$. 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. These multipliers differ in the second phase, where 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 order to sum the n rows of the tableau. This multiplier has a simple grid-like 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}.

For the second phase, arrange full adders $A_{i,j}$, where $i, j \in [0, n]$, into a grid as shown in Figure 7. Adder $A_{i,j}$ occupies the j-th row and the $(i + j)$-th column and outputs the carry and sum bits $c_{i,j}$ and $d_{i,j}$. For $i < 0$, adder $A_{i,j}$ takes inputs $t_{i,j}, d_{i+i,j-1}, c_{i-1,j}$ (replacing nonexistent variables with the constant 0). Adders of the form $A_{n,j}$ take input $c_{n,j-1}$ instead of $c_{n-1,j}$. 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 $o_0, o_1, \ldots, o_{2n-1}$.

3.2 Overview: Efficient Proofs for Degree Two Array Multiplier Identities

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

Proof Overview. The first step in 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 L and R. This output bit has no dependence on the circuitry in the higher order columns to the left, so those columns can be removed while preserving the unsatisfiability of the remaining subcircuit.

The key insight is that almost all of the columns to the right can also be removed while preserving unsatisfiability, reducing the problem to a narrow subcircuit that we call a *critical strip*. After removing the low-order columns, the carry-bits feeding into the strip become unconstrained. If

the critical strip is too thin, then these unconstrained carry-bits could have enough total weight to "cause" the disagreeing output bit, making the instance satisfiable. But for a large enough choice of width, this cannot happen: each additional column on the right roughly halves the maximum possible total weight of these carry-bits, so as the width of the strip increases, the weight of the unconstrained carry-bits quickly becomes too small to cause the large disagreement in the leading output bit. It then remains to refute each critical strip.

Our proofs inside each critical strip repeat three steps: (1) Branch on some of the input bits. Typically these will correspond to a row of the tableau. (2) Propagate those values as far in the circuit as possible. (3) Save the resulting assignment to the boundary of the propagation. We call each of these boundaries a *cut* in the circuit. Because the critical strip is narrow, we will only need to remember an assignment to a small number of variables as we move along these cuts in the critical strip.

These *cuts* are sets of variables that, under any assignment, split the strip into a satisfiable and an unsatisfiable region. If a cut assignment was propagated from an already-queried portion of the circuit, then this cut assignment is consistent with the assignment given by those queries. But since the critical strip as a whole is unsatisfiable, this cut assignment must be inconsistent with any assignment to the unqueried portion of the circuit. Walking these cuts down the critical strip, row-by-row, we reduce the unsatisfiable, unqueried region in the critical strip until it is trivially refuted.

One can view our proof as showing that the constraints within each strip form a graph of *pathwidth* $O(\log n)$ which, by Dechter (1996), 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 Alekhnovich and Razborov (2002), 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.

3.3 Proofs of Array Multiplier Commutativity

Definition 3.2. We define a SAT instance $\phi^{\text{Array}}_{\text{Comm}}(n)$. The inputs are length n bit-vectors $\mathbf{x}, \mathbf{y}$. Using the construction from Section 3.1, we define array multipliers L^{xy} and R^{yx}. The tableau variables are defined by the constraints

$$t^{xy}_{i,j} = x_i \wedge y_j, \qquad t^{yx}_{i,j} = y_i \wedge x_j,$$

and in particular we can infer, through resolution, that $t^{xy}_{i,j} = t^{yx}_{j,i}$.

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

$$e_i = \left[o^{xy}_i \neq o^{yx}_i\right] \quad \forall i \in [0, 2n-1],$$
$$e_0 \vee e_1 \vee \ldots e_{2n-1}.$$

We give a small resolution proof for $\phi^{\text{Array}}_{\text{Comm}}(n)$ in the form of a labeled OBDD B, as described in Proposition 2.6. The variable order for B begins with $e_0, e_1, \ldots,$ followed by the output bits $o^{yx}_0, o^{yx}_1, \ldots$. Then, B reads the variables associated with adders $A^{xy}_{i,j}, A^{yx}_{j,i}$ in order of increasing j, reading each row right to left. Finally, B reads the output bits $o^{xy}_0, o^{xy}_1, \ldots,$ then the input bits $\mathbf{x}, \mathbf{y}$ in an arbitrary order.

At the root of B, we search for the first output bit on which L^{xy} and R^{yx} disagree 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^{xy} and R^{yx} on the variables inside columns $[k-\Delta, k]$.

Definition 3.3. Let $\Delta = \log n$. Let $\phi_{\text{Strip}}(k)$ hold the constraints from $\phi^{\text{Array}}_{\text{Comm}}(n)$ containing any tableau variable $t^{xy}_{i,j}$ or $t^{yx}_{i,j}$ for $i+j \in [k-\Delta, k]$. Then add unit clauses to $\phi_{\text{Strip}}(k)$ to encode the assignment: $e_0 = 0, e_1 = 0, \ldots, e_{k-1} = 0, e_k = 1$. 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 3.4. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k.*

Proof. We interpret each critical strip as a circuit that outputs the weighted sum of the input variables in circuits L^{xy} and R^{yx}. The assignment to $\mathbf{e}$ demands that the difference between the critical strip outputs is precisely 2^k. But by $t^{xy}_{i,j} = t^{yx}_{j,i}$, 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 larger of the sums of the input carry bits to column $k-\Delta$ in the two strips. There are fewer than n input carry bits for each 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 $t^{xy}_{ij} = t^{yx}_{ji}$ in the tableau variables. The additional requirement that the tableau variables came from an assignment to $\mathbf{x}, \mathbf{y}$ is unnecessary to refute $\phi_{\text{Strip}}(k)$.

Also observe that if one of the array multipliers has a bug, then at least one of the $2n$ critical strips will be satisfiable.

Lemma 3.5. *There is an $O(n^6 \log n)$-sized ordered resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof. For simplicity, we assume that $k \le n$; the case where $k > n$ is similar. We will also preprocess $\phi_{\text{Strip}}(k)$ by resolving on the variables in $\mathbf{x}, \mathbf{y}$ to obtain the tableau variable relations $t^{yx}_{j,i} = t^{xy}_{i,j}$, then replacing all the variables $t^{yx}_{j,i}$ by $t^{xy}_{i,j}$ 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 define the *input variables* σ_{input} as the set of tableau variables of circuit L^{xy}, together with the carry variables from column $k-\Delta-1$ of both L^{xy} and R^{yx}. We say σ_{input} contains the *input variables* to this critical strip, since their values determine an output assignment.

The idea behind the branching program B is to verify circuit L^{xy} by branching on its input variables row-by-row, going from top-to-bottom, remembering an assignment to a row of sum-variables. Since $t^{xy}_{i,j} = t^{yx}_{j,i}$, the tableau variables of circuit R^{yx} simultaneously are revealed from bottom to top. In circuit R^{yx}, we maintain both a guess for its output values, and a row of sum-variables. From the proof of Lemma 3.4, if we have found that the outputs of L^{xy} and R^{yx} were computed correctly, then they must violate one of the constraints $e_k = 1, \ldots, e_{k-\Delta+1} = 0$, $e_{k-\Delta} = 0$. □

Definition 3.6. Define Cut(0) as the set of variables containing

$$d^{yx}_{0,i}, o^{yx}_{i-1} \quad \text{for} \quad i-1 \in [k-\Delta, k].$$

For $j \in [1, k-\Delta]$, we define Cut(j) to be the set containing the variables:

$$\begin{aligned} d^{xy}_{i,j-1}, d^{yx}_{j,i-1} &\quad \text{for} \quad i+j-1 \in [k-\Delta, k], \\ c^{yx}_{j-1,i} &\quad \text{for} \quad i+j-1 \in [k-\Delta, k-1], \\ o^{yx}_{i} &\quad \text{for} \quad i \in [k-\Delta, k]. \end{aligned}$$

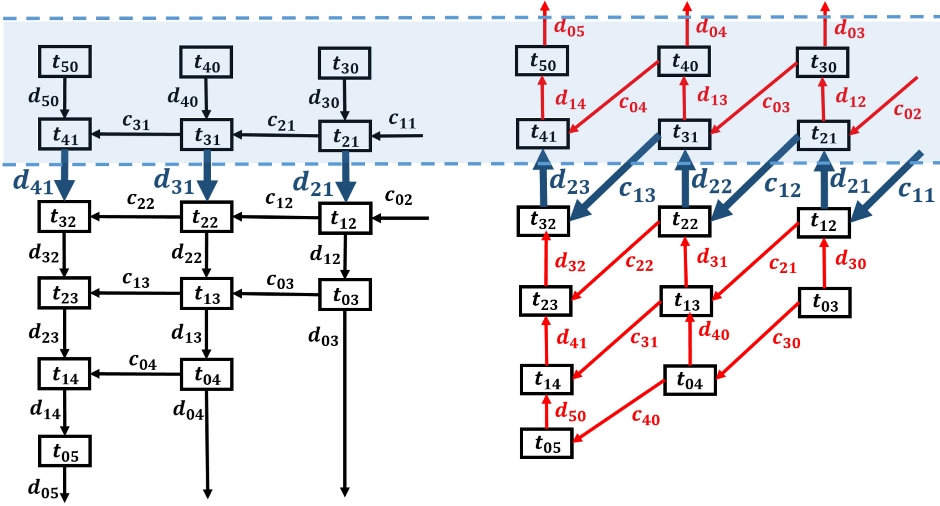

Fig. 9. The critical strip $\phi_{\text{Strip}}(5)$ for checking commutativity. The enlarged variables belong to Cut(2) of $\phi_{\text{Strip}}(5)$. This cut divides the critical strip into a shaded satisfiable region and an unshaded unsatisfiable region.

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

$$\begin{aligned} o_i^{xy} &\quad \text{for} \quad i \in [k-\Delta, j-1], \\ d_{i+1,j-1}^{xy}, d_{j,i}^{yx}, c_{j-1,i}^{yx} &\quad \text{for} \quad i+j \in [k-\Delta, k], \\ c_{j-1,i}^{yx} &\quad \text{for} \quad i+j-1 \in [k-\Delta, k-1], \\ o_i^{yx} &\quad \text{for} \quad i \in [k-\Delta, k]. \end{aligned}$$

We will label each node of B by the pair (Cut(j), σ) where Cut(j) keeps track of the previously seen cut. See Figure 9.

Initialization. Throughout, we work in terms of the tableau variables in circuit L^{xy}, implicitly substituting t_{ij}^{xy} for t_{ji}^{yx}. We begin at the root node of the read-once branching program B, labeled with an empty cut and an empty partial assignment $(\emptyset, \emptyset)$. For $i \in [k-\Delta, k]$, we branch on the variable o_i^{yx}, then propagate to $d_{0,i}^{yx}$ using a clause from the constraint $o_i^{yx} = d_{0,i}^{yx}$. The surviving branches are those labeled by an assignment satisfying the constraints $o_i^{yx} = d_{0,i}^{yx}$. 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 xy:

$$t_{i,0}^{xy} \quad \text{for} \quad i \in [k-\Delta, k].$$

Then we propagate to the variables, in sequence,

$$d_{1,i}^{yx}, c_{0,i}^{yx} \quad \text{for} \quad i+1 \in [k-\Delta, k]$$

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

Inductive Step. We now describe the transition from Cut(j) to Cut($j+1$) for $1 \leq j \leq k$. Suppose that the branching program B has reached an assignment to Cut(j). From these nodes we branch

on the next, j-th row's tableau variables

$$t^{xy}_{i,j} \quad \text{for} \quad i + j \in [k - \Delta, k]$$

and, when they exist, the pair of incoming input carry variables $c^{xy}_{i,j}, c^{yx}_{j-1,i}$ from column $k - \Delta - 1$. We then propagate to the Cut$(j + 1)$ and c^{xy} variables in the sequence:

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

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

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

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

We repeat this step until we have reached Cut$(k + 1)$. At this point, we have an assignment to the critical strip output bits $\mathbf{o}^{xy}, \mathbf{o}^{yx}$. 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 3.4, this implies that our assignment to $\mathbf{o}^{xy}, \mathbf{o}^{yx}$ conflicts with an inequality constraint.

Size Bound. We show that there are $O(n^6 \log n)$ nodes in B. Each 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 Cut(j). We branch on up to $\Delta + 2$ input variables, so each cut has a full binary tree of $8n$ nodes. For each leaf of this tree, B has a path of $O(\Delta)$ nodes for propagating before the nodes get merged. Therefore, each cut labels at most $O(n^5\Delta)$ nodes. There are $k + 1$ different cuts, thus B has at most $O((k + 1)n^5\Delta) = O(n^6 \log n)$ 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 the following.

Corollary 3.7. *$\phi_{\text{Strip}}(k)$ has an $O(n^5 \log n)$-size regular resolution refutation.*

We note that the alternative strategy of directly branching on the cuts to perform binary search on the critical strip yields the same size bound as Corollary 3.7.

Theorem 3.8. *Let $N = |\phi^{\text{Array}}_{\text{Comm}}| = O(n^2)$. There is an $O(N^3 \log N)$ size regular resolution proof that $\phi^{\text{Array}}_{\text{Comm}}$ is unsatisfiable. There is an $O(N^{7/2} \log N)$ 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 3.7 or Lemma 3.5 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. Corollary 3.7 yields the regular resolution proof and Lemma 3.5 yields the ordered resolution proof. □

3.4 Proofs of Array Multiplier Distributivity

Definition 3.9. 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. For the left-hand expression, we construct a ripple-carry adder L^{y+z}, outputting $\mathbf{o}^{(y+z)}$, and array multiplier $L^{x(y+z)}$ outputting $\mathbf{o}^{x(y+z)}$. 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 again divide the instance into critical strips, following the strategy previously used to refute $\phi^{\text{Array}}_{\text{Comm}}$.

Definition 3.10. Define the constant $\Delta = \log(2n)$. Let $\phi_{\text{Strip}}(k)$ contain the following constraints from $\phi^{\text{Array}}_{\text{Dist}}(n)$: first, the full ripple-carry adder circuit L^{y+z}. Second, include the constraints containing one of the tableau variables $t^{x(y+z)}_{i,j}, t^{xy}_{i,j}, t^{xz}_{i,j}$ for $i + j \in [k - \Delta, k]$. Third, include the ripple-carry adder constraints on the carry-bits and sum-bits c^{xy+xz}_i, o^{xy+xz}_i for $i \in [k - \Delta, k]$. Lastly, add constraints to $\phi_{\text{Strip}}(k)$ that assign $e_k = 1, e_{k-1} = 0, \ldots, e_0 = 0$.

Lemma 3.11. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k.*

Proof. Like the proof of Lemma 3.4, 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 3.12. *For each k there is an $O(n^5 \log n)$ size regular resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof. We construct a labeled branching program B that solves the conflict clause search problem for $\phi_{\text{Strip}}(k)$. 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 computing an expression $C \in \{x(y + z), xz, xy\}$ 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 $\text{Cut}^{y+z}(j)$ as the singleton set $\{c^{y+z}_{j-1}\}$ and define $\text{Cut}^x(j)$ as the set

$$x_i \quad : \quad i \in [k - j - \Delta, k - j].$$

We also refer to a global cut, across the whole circuit: $\text{Cut}(j) = \cup_C \text{Cut}^C(j)$. See Figure 10.

Initialization: Getting to Cut(1). At the root node $(\emptyset, \emptyset)$ of B, we branch on the circuit input variables y_0, z_0 and

$$x_i \quad \text{for} \quad i \in [k - \Delta, k].$$

We propagate these assignments to variables c^{y+z}_0 and o^{y+z}_0, giving us an assignment to $\text{Cut}^{y+z}(0)$. The assignment to o^{y+z}_0, in turn, propagates to an assignment to the first row of tableau and sum

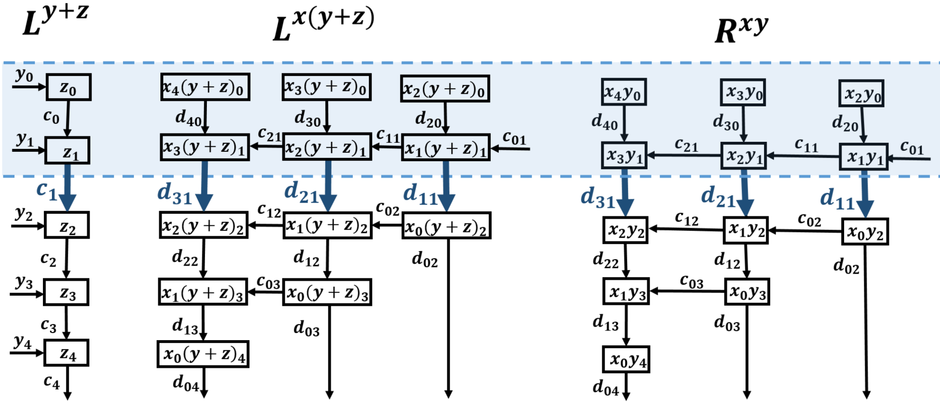

Fig. 10. The critical strip $\phi_{\text{Strip}}(4)$ for distributivity. Cut(2) consists of the enlarged variables.

variables from the critical strip for $L^{x(y+z)}$:

$$t_{i,0}^{x(y+z)}, d_{i,0}^{x(y+z)} \quad \text{for} \quad i \in [k-\Delta, k].$$

At this point, we have an assignment to $\text{Cut}^{x(y+z)}(0)$.

We then propagate the input variable assignments through the multipliers R^{xy} and R^{xz}:

$$t_{i,0}^{xy}, d_{i,0}^{xy} \quad : \quad i \in [k-\Delta, k],$$

$$t_{i,0}^{xz}, d_{i,0}^{xz} \quad : \quad i \in [k-\Delta, k],$$

obtaining assignments to $\text{Cut}^{xy}(0)$ and $\text{Cut}^{xz}(0)$, thus completing an assignment to Cut(0). At this point we merge nodes on assignment to Cut(0).

Inductive Step: Cut(j) *to* Cut($j+1$). Suppose we have merged branches and are at a node labeled with an assignment to Cut(j). If this assignment contains a variable $d_{0,i}^C$ we propagate to o_i^C. We branch on input variables $x_{k-\Delta-j}, y_j, z_j$. We then propagate these assignments to $c_{j+1}^{y+z}, o_{j+1}^{y+z}$, followed by the next row of tableau, carry, and sum variables in each multiplier:

$$c_{i-j-2,j+1}^C, t_{i-j-1,j+1}^C, d_{i-j-1,j+1}^C \quad : \quad i \in [k-\Delta, k].$$

At this point, we have reached an assignment to all of the variables in Cut($j+1$) so we merge nodes based on Cut($j+1$). We repeat this step until reaching an assignment to Cut($k+1$), which consists of each multiplier's output bit-vector $\mathbf{o}^C$.

End: Beyond Cut($k+1$). Suppose that we have reached Cut($k+1$) and merged nodes. We branch on the input carry variable $c_{k-\Delta-1}^{xy+xz}$, that goes into the critical strip of ripple-carry adder R^{xy+xz}. We can then propagate to the outputs $\mathbf{o}^{xy+xz}$. We now have an assignment to both $\mathbf{o}^{x(y+z)}, \mathbf{o}^{xy+xz}$ that was propagated from one assignment to the input variables to the critical strip. By Lemma 3.11, this assignment conflicts with an inequality constraint from E.

Size Bound. There are $k+1$ different global cuts Cut(j). Each Cut(j) section of B begins with an assignment to at most $4\Delta + 1$ variables, and then branches on three input variables. So each section Cut(j) is initialized with at most $8 * 2^{4\Delta+1} = O(n^4)$ branches. Each of these branches then propagates in a path with at most $O(\Delta)$ nodes. So there are at most $O(n^4 \log n)$ nodes per cut and therefore at most $O((k+1)n^4 \log n) = O(n^5 \log n)$ nodes in B. □

THEOREM 3.13. *There is an $O(n^6 \log n)$ size resolution proof that $\phi_{\text{Dist}}^{\text{Array}}(k)$ is unsatisfiable.*

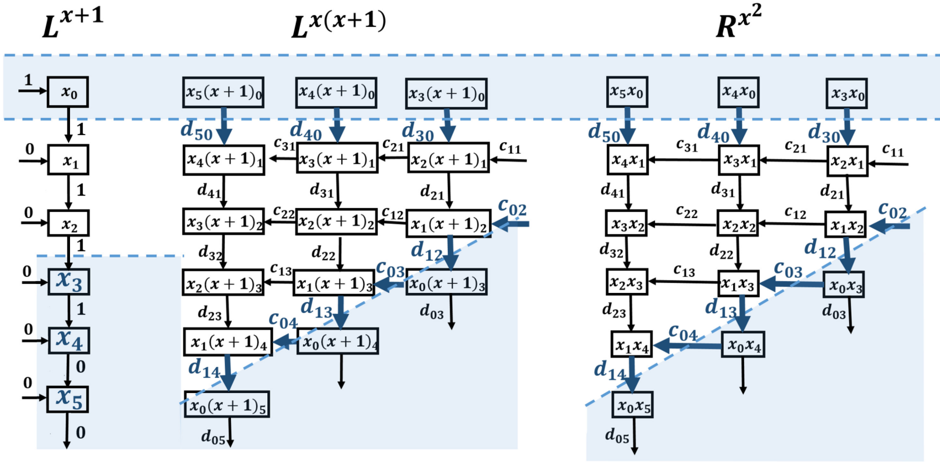

Fig. 11. The critical strip $\phi_{\text{Strip}}(5)$ for checking $x(x+1) = x^2 + x$. The shaded region is satisfiable. The enlarged variables belong to Cut(1).

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 $O(n^5 \log n)$ size proof from Lemma 3.12. □

3.5 Proofs of $x(x+1) = x^2 + x$ for Array Multipliers

Definition 3.14. We define a SAT instance $\phi^{\text{Array}}_{x(x+1)}(n)$. Circuit L is composed of circuits L^{x+1}, consisting of a ripple-carry adder taking inputs $\mathbf{x}$ and $\mathbf{1}$ and outputting their sum $(\mathbf{x}+\mathbf{1})$, and $L^{x(x+1)}$, an array multiplier outputting the product $\mathbf{x}(\mathbf{x}+\mathbf{1})$. Similarly, circuit R is composed of circuits R^{x^2} and R^{x^2+x}.

We let E contain the usual inequality constraints. The instance is then

$$\phi^{\text{Array}}_{x(x+1)}(n) = L \cup R \cup E.$$

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.

Definition 3.15. Define the constant $\Delta = \log(2n-1)$. Let $\phi_{\text{Strip}}(k)$ contain the full ripple-carry adder circuit L^{x+1} from $\phi^{\text{Array}}_{x(x+1)}(n)$. Also include the constraints containing one of the multiplier tableau variables $t^{x(x+1)}_{i,j}, t^{x^2}_{i,j}$ for $i + j \in [k-\Delta, k]$. Further include the constraints on the ripple-carry adder carry-bits and sum-bits $c^{x^2+x}_i, d^{x^2+x}_i$ for $i \in [k-\Delta, k]$. Lastly, add constraints to $\phi_{\text{Strip}}(k)$ that encode the values of the bits: $e_k = 1, e_{k-1} = 0, \ldots, e_0 = 0$.

We refer to the subcircuit $\phi_{\text{Strip}}(k) \cap C$ as the *critical strip* for C. Figure 11 shows an example of a critical strip.

Lemma 3.16. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k.*

Proof. The proof is the same as the proof for Lemma 3.11. □

Definition 3.17. For an array multiplier computing the expression $C \in \{x(x+1), x^2\}$ and $j \in [1, (k-\Delta)/2]$, we define $\text{Cut}^C(j)$ to be the set of variables

$$d^C_{i,j-1} \quad : \quad i+j-1 \in [k-\Delta, k] \quad \text{(upper cut)},$$

$$c^C_{j-1,i}, d^C_{j,i} \quad : \quad i+j \in [k-\Delta, k] \quad \text{(lower cut)}.$$

We define $\text{Cut}^{x+1}(j)$ to contain x_{j-1} and the set of variables

$$x_i \quad : \quad i \in [k-j-\Delta, k-j].$$

Theorem 3.18. *There is a size $n^7 \log n$ regular resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof.

Initialization. We give our proof in the form of a labeled read-once branching program B. We begin by branching on a guess for the critical strip outputs $\mathbf{o}^{x(x+1)}, \mathbf{o}^{x^2+x}$. For the branches that do not conflict with an inequality constraint, we branch on the values

$$o^{x^2}_i, x_i \quad : i \in [k-\Delta, k],$$

then merge to erase the assignment to $\mathbf{o}^{x^2+x}$.

We observe that the carry variables in L^{x+1} must be a sequence of 1's followed by 0's. If, on the contrary, we observe the assignments $c_i = 0$ and $c_j = 1$ for $i < j$, then we can efficiently find a conflict by propagating $c_i = 0$ through columns $[i, j]$. So we can begin this proof by branching on the at most n valid carry-bit assignments

$$c^{x+1}_0 = 1, \ldots, c^{x+1}_i = 1, c^{x+1}_{i+1} = 0, \ldots, c^{x+1}_k = 0.$$

Our branch order begins on the input variables x_0 and $x_k, x_{k-1}, \ldots, x_{k-\Delta}$. We propagate the resulting assignment to the upper and lower cuts in each circuit, then merge on the assignment to Cut(1).

Inductive Step. To get from Cut(j) to Cut($j+1$), we branch on input variables $x_j, x_{k-j-\Delta+1}$, then propagate to and merge on Cut($j+1$).

We have two cases: the upper and lower cuts of Cut($j+1$) either intersect or they do not. In either case, we branch on input variables $x_{j-1}, x_{k-\Delta-j+1}$ and the input carry variables to rows j and $(k-j-\Delta+1)$. If the cuts do not intersect, we propagate to, then merge on, all the Cut($j+1$) variables. Otherwise, suppose that the upper and lower cuts of Cut($j+1$) intersect on $d_{i,j}$. The upper and lower cuts of Cut(j) either propagate to conflicting values of $d_{i,j}$, in which case we have found a conflict, or they agree on the value of $d_{i,j}$, in which case we delete column $i+j$ from our cuts.

Size Bound. Each cut belongs to one of up to n branches for the carry variables in L^{x+1} and holds an assignment to at most $7 \log n$ variables so there are at most n^8 initial nodes for each cut. Each of these nodes propagates for $O(\log n)$ steps to get to the next cut, so our branching program has size $O(n^9 \log n)$. ☐

We can now obtain a refutation for $\phi^{\text{Array}}_{x(x+1)}(n)$ by branching on sequences of variables in $\mathbf{e}$ and using the refutation for $\phi_{\text{Strip}}(k)$ on each branch.

Theorem 3.19. *There is a size $n^{10} \log n$ regular resolution proof that the SAT instance $\phi^{\text{Array}}_{x(x+1)}(n)$ is unsatisfiable.*

3.6 Degree Two Identity Proofs for Array Multipliers

Let $\phi^{\text{Array}}_{L=R}(n)$ denote a SAT instance checking that the array multiplier obeys the ring identity $L = R$. With the insight from the earlier proofs in this section, we can prove the general theorem as follows.

Theorem 3.20. *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 bit-vectors in parallel, each in the same order. This branch order for each input bit-vector $\mathbf{x}$ is $x_0, x_n, x_1, x_{n-1}, \ldots$. We branch on the input carry-bits as needed to propagate the cuts. We can propagate the resulting input variable assignments to diagonal cuts in each multiplier that scan from the top and bottom edges toward the middle, and likewise for the intact ripple-carry adders. In each input bit-vector, 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. □

4 DIAGONAL MULTIPLIERS AND BOOTH MULTIPLIERS

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 8.

A Booth multiplier uses a similar idea to the array multiplier, but uses two's complement notation and a telescoping sum identity to skip consecutive digits in one multiplicand. To add the terms of this sum, the Booth multiplier uses a grid of full adders similarly to the array multiplier, but with some small modifications to accommodate signed integers.

Like with the array multiplier, we can divide the diagonal and Booth multipliers into $O(\log n)$-width unsatisfiable critical strips. Using the same input variable orderings from Section 3, we can verify each of these critical strips with a polynomial-size regular resolution proof.

Definition 4.1. Let $\phi^{\text{Diag}}_{L=R}(n)$ denote the SAT instance checking that an n-bit diagonal multiplier obeys the ring identity $L = R$. Likewise, let $\phi^{\text{Booth}}_{L=R}(n)$ denote the SAT instance checking that an n-bit Booth multiplier obeys the ring identity $L = R$.

Theorem 4.2. *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 (Sketch). We divide $\phi^{\text{Diag}}_{L=R}(n)$ or $\phi^{\text{Booth}}_{L=R}(n)$ into unsatisfiable critical strips of width $\Delta = \log mn$, where m is the number of terms in the identity $L = R$. This is the same width as in the array multiplier since the number of input carry-bits in each multiplier's critical strip is at most n. The ripple-carry adders that input to a multiplier remain intact, and for the rest we remove the columns outside the critical strip. We note that although the Booth multiplier uses two's complement signed integers, this does not materially affect our critical strip proofs.

We begin by branching on guesses for the Δ output bits from each multiplier and each truncated ripple-carry adder. We use the same branch order as in the array multiplier proof: each input bit-vector $\mathbf{x}$ is read in parallel, in the order $x_0, x_n, x_1, x_{n-1}, \ldots$. We branch on the input carry-bits as

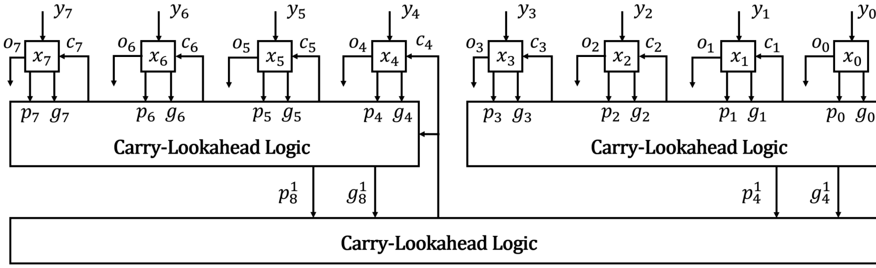

Fig. 12. 8-bit, two-layer CLA adding $\mathbf{x}, \mathbf{y}$.

needed to propagate the cuts. We can propagate the input variable assignments to diagonal cuts in each multiplier that scan from the top and bottom edges toward the middle, and likewise for the intact ripple-carry adders. In each input bit-vector, we remember the assignment to just the most recently queried 2Δ variables. □

5 WALLACE TREE MULTIPLIERS

5.1 Wallace Tree Multiplier Construction

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.

Carry-Lookahead Adder. A carry-lookahead adder (CLA) uses a tree structure to add two bit-vectors $\mathbf{x}, \mathbf{y}$ with only logarithmic depth. The 4-bit CLA computes, for each pair x_i, y_i, the values

$$g_i = x_i y_i \qquad p_i = x_i \oplus y_i.$$

Then, writing c_i for the carry bit in the i-th column, we have

$$c_{i+1} = g_i \oplus (p_i c_i).$$

We can use this to derive the following equations, which we can use to compute each carry digit in parallel from the values g_i, p_i, and c_0:

$$\begin{aligned} c_1 &= g_0 \oplus p_0 c_0, \\ c_2 &= g_1 \oplus g_0 p_1 \oplus c_0 p_{0,0} p_1, \\ c_3 &= g_2 \oplus g_1 p_2 \oplus g_0 p_1 p_2 \oplus c_0 p_0 p_1 p_2, \\ c_4 &= g_3 \oplus g_2 p_3 \oplus g_1 p_2 p_3 \oplus g_0 p_1 p_2 p_3 \oplus c_0 p_0 p_1 p_2 p_3. \end{aligned}$$

These values are used to compute the outputs: $o_i = c_i \oplus x_i \oplus y_i$. It additionally computes the *group propagate* and *group generate*:

$$\begin{aligned} p_{1,4} &= p_3 p_2 p_1 p_0, \\ g_{1,4} &= g_3 \oplus g_2 p_3 \oplus g_1 p_3 p_2 \oplus g_0 p_3 p_2 p_1, \end{aligned}$$

where the first index indicates the layer.

We construct a 16-bit CLA with two layers, whose first half of is shown in Figure 12. At the zero-th layer we arrange four 4-bit CLAs, the k-th CLA taking inputs $x_i, y_i, i \in [4k, 4k+3]$ and outputting to $p_{0,i}, g_{0,i}, i \in [4k, 4k+3]$, where the superscript indicates the layer. We denote the k-th CLA group propagate and generate by $p_{1,4k} g_{1,4k}$. Then the carries $c_4, c_8, c_{12}, \ldots$ can be computed

by the equations

$$\begin{aligned}
c_4 &= g_{1,0} \oplus p_{1,0}c_0, \\
c_8 &= g_{1,4} \oplus g_{1,0}p_{1,4} \oplus c_0p_{1,0}p_{1,4}, \\
c_{12} &= g_{1,8} \oplus g_{1,4}p_{1,8} \oplus g_{1,0}p_{1,4}p_{1,8} \oplus c_0p_{1,0}p_{1.4}p_{1,8}, \\
c_{16} &= g_{1,12} \oplus g_{1,8}p_{1,12} \oplus g_{1,4}p_{1,8}p_{1,12} \oplus p_{1,0}p_{1,4}p_{1,8}p_{1,12} \oplus c_0p_{1,0}p_{1,4}p_{1,8}p_{1,12}.
\end{aligned}$$

Notice that these equations are isomorphic to the previous equations for computing carries within each 4-bit CLA. We can reuse the same circuitry from the 4-bit CLA to compute these carries, as well as the group propagate and generate for the next layer. We can repeat this process to construct larger CLAs, with each iteration able to handle four times the bitwidth.

Wallace Tree Multiplier. We construct a Wallace tree multiplier taking input $(\mathbf{x}, \mathbf{y})$. We compute a tableau of partial products like in the array multiplier. We then go through $h \approx \log n$ steps to reduce the n-row starting tableau to an equivalent two-row tableau.

We define *tableau variables* $t_{\ell,i,j}$ where ℓ is the layer of the tableau, i is the index of the column containing the adder, and j is the row. We will denote the set of tableau variables in a column by

$$\mathrm{Col}(i) = \{t_{\ell,i,j} \quad \text{for all} \quad \ell, j\},$$

and call the subset of a column within a layer l a *subcolumn*, denoted by

$$\mathrm{Col}(\ell, i) = \{t_{\ell,i,j} \quad \text{for all} \quad j\}.$$

In the zero-th layer, the tableau variables represent the partial products:

$$t_{0,i,j} = x_{i-j} \wedge y_j \quad \text{for} \quad i < n,$$
$$t_{0,i,j} = x_{n-1-j} \wedge y_{i-n+j+1} \quad \text{for} \quad i \geq n.$$

We now specify how to construct layer $\ell + 1$ from layer ℓ. We partition the rows of layer ℓ into sets of three, from top to bottom. Adder $A_{\ell,i,j}$ will take input from the i-th column of the j-th set of three rows. For each row of adders $j = 0, 1, \ldots$, for each $i \in [0, 2n]$, we append adder $A_{\ell,i,j}$'s sum-bit to subcolumn $\mathrm{Col}(\ell + 1, i)$. Then for each i, we append adder $A_{\ell,i,j}$'s carry-bit to subcolumn $\mathrm{Col}(\ell + 1, i + 1)$.

Each layer reduces the number of rows in the tableau from N to $\lceil 2N/3 \rceil$. The tableau for the last layer $h < \log_{3/2}(n) < 2\log n$, will only have two rows. We use a $2n$-bit[2] CLA to sum the two rows in logarithmic depth, outputting the final sum in the output bits o_i.

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. This cut size results in quasipolynomial size regular resolution proofs.

When analyzing the cuts in a Wallace tree multiplier, we will find the following property useful.

Definition 5.1. For layer ℓ of a Wallace tree multiplier, if for each $j \leq k$, the outputs of the j-th row of adders, $\{A_{\ell,i,j}\}_i$, map to and cover the rows $2j, 2j + 1$ of the next layer $\ell + 1$'s tableau, we say that layer ℓ is *row-friendly* up to its k-th row of adders. If layer ℓ is row-friendly up to its last row of adders, we say that layer ℓ is *row-friendly.*

LEMMA 5.2. *In a Wallace tree multiplier, each layer $\ell \in [0, h - 2]$ is row-friendly.*

In terms of the dot diagram in Figure 13, this lemma simply states that no two bits are connected with a line of slope greater than one.

[2] This is not a $(2n - 1)$-bit adder because the top summand may have $2n$ bits.

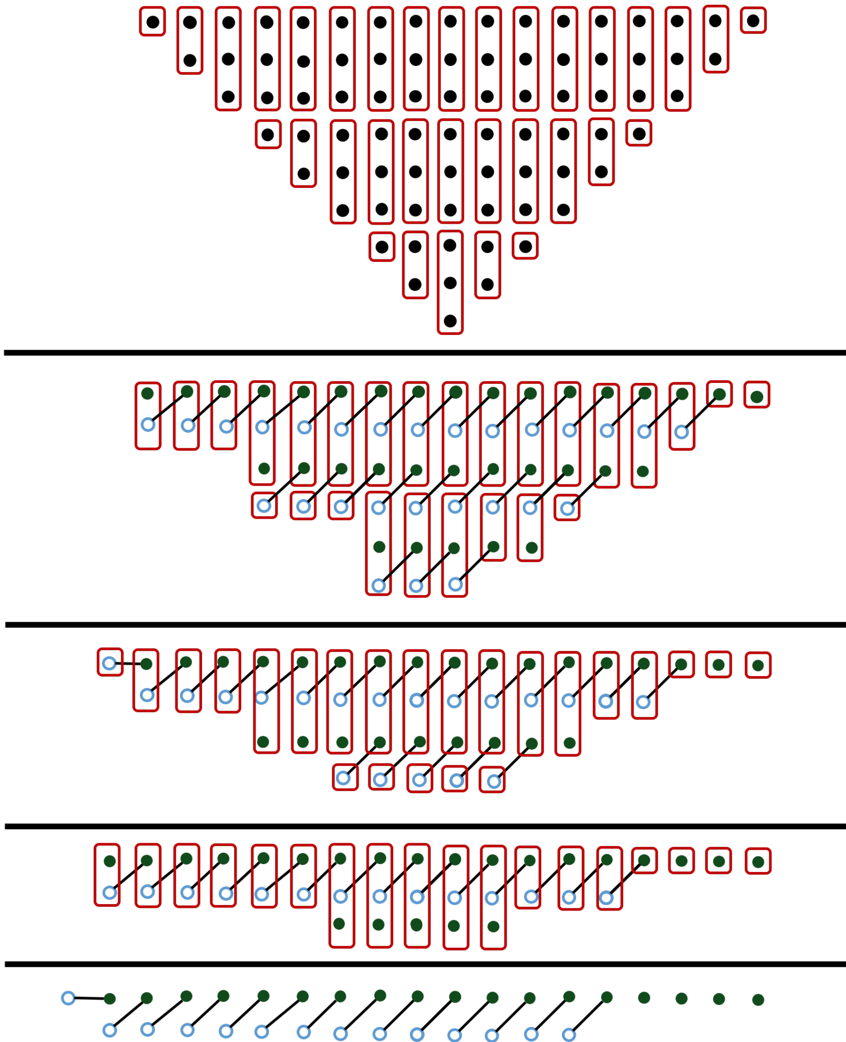

Fig. 13. Dot diagram for a 9×9 Wallace tree multiplier. Hollow dots represent carry-bits and solid dots represent sum-bits. Dots connected by an edge are output by the same adder.

5.2 Proofs of Wallace Tree Multiplier Commutativity

Definition 5.3. We define a SAT instance $\phi^{\text{Wall}}_{\text{Comm}}(n)$. The inputs to the multipliers are n-bit integers $\mathbf{x}, \mathbf{y}$. Using the construction from Section 5, we define Wallace tree multipliers L, computing $\mathbf{xy}$, and R, computing $\mathbf{yx}$ (reversing the order of multiplier inputs).

After specifying the circuits L and R, we add a circuit E, of *inequality constraints* encoding that the two circuits disagree on some output bit.

Definition 5.4. Define $\delta = \log(n+2)$. Let $\phi_{\text{Strip}}(k)$ contain the constraints from $\phi^{\text{Wall}}_{\text{Comm}}(n)$ that contain a tableau variable $t^{xy}_{\ell,i,j}$ or $t^{yx}_{\ell,i,j}$ for $i \in [k-\delta, k]$, and also the constraints for the full CLAs at the end of the Wallace tree multipliers. Also add unit clauses to $\phi_{\text{Strip}}(k)$ for the assignment: $e_0 = 0, e_1 = 0, \ldots, e_{k-1} = 0, e_k = 1$.

We call the newly unconstrained tableau bits in column $k - \delta$, that were carry-bits output by adders from the removed column $k - \delta - 1$, the *input carry-bits* to $\phi_{\text{Strip}}(k)$.

Lemma 5.5. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k.*

Proof. We reason similarly to the proof of Lemma 3.4. Again, we interpret the critical strip as a circuit that computes the weighted sum, in both L and R, of the tableau variables within the strip. The assignment to $\mathbf{e}$ asserts that the outputs of L and R differ by precisely 2^k. We bound the admissible difference in outputs by counting the number of input carry-bits in either L or R. Since each layer of a Wallace tree multiplier has $\lceil 2/3 \rceil$ fewer rows than the previous layer, the total number of tableau rows past the initial layer is at most $2n$. At most half of these rows are composed of carry-bits, so circuits L and R each have at most n input carry-bits coming from the removed column $k - \delta - 1$. Additionally, the newly unconstrained inputs to the final CLA from the removed columns can contribute a total weight of at most $2^{k-\delta}$ to the final output. Since we set $\delta = \log(n + 2)$, the total difference between the final outputs is at most $2^{k-\delta}(n + 2) < 2^k$. □

Lemma 5.6. *There is a regular resolution proof of size $2^{8 \log^2 n + O(\log n)}$ that $\phi_{\text{Strip}}(k)$ is unsatisfiable.*

Proof. The idea of this proof is to read the initial layer of the critical strip row-by-row. If we have assigned all of the inputs to a row of adders, we propagate to their output bits. In this way, an input assignment to $\mathbf{x}$ and $\mathbf{y}$ will propagate through the layers of the Wallace tree multiplier in parallel, then finally reach an assignment to the output bits of both circuits. From the proof of 5.5, the result will contradict one of the inequality constraints from $\phi^{\text{Wall}}_{\text{Comm}}(n)$.

Each node of the branching program will only keep track of a constant number of variables in each subcolumn. This will ensure that the cuts have $O(\log^2 n)$ variables, so that the branching program has at most $2^{O(\log^2 n)}$ nodes.

We first preprocess the constraints to obtain the equalities $t^{xy}_{0,i,j} = t^{yx}_{0,i,i-j}$. Like in the array multiplier case, as we branch from the top tableau row downward in circuit L, we will reveal the bottom row upward in circuit R. We will first describe how the branching program B propagates an assignment from the initial tableau to an assignment to the last layer in circuit L. The propagation in circuit R works symmetrically, going from the bottom row of adders to the top in each layer. Then we will describe how to propagate an assignment to the last layer through the CLA to finally reach an assignment to the output bits. □

ALGORITHM 1: Propagates from the initial layer $\ell = 0$ to the final layer $\ell = h$ of the critical strip L while assigning at most a constant number of bits per subcolumn.

```
1  for j = 0, 1, ..., ⌈n/3⌉ do
2      Branch on the inputs to the j-th row of adders {A^{xy}_{0,i,j}}_i
3      for each layer ℓ = 0, 1, ..., h − 1 before the last layer do
4          if layer ℓ has a fully assigned row of adders {A^{xy}_{ℓ,i,j'}}_i then
5              Propagate to tableau rows 2j', 2j' + 1 of layer ℓ + 1.
6              Merge to forget the assignment to the row of adders {A^{xy}_{ℓ,i,j'}}_i
7              Branch on any input carry-bits in tableau rows 2j', 2j' + 1 of layer ℓ + 1
8          end
9      end
10 end
```

The branching program B begins by following the Algorithm 1 on circuit L. We use the propagation loop in lines 3–9 for circuit R, leaving the branching steps to circuit L. We claim that at the end,

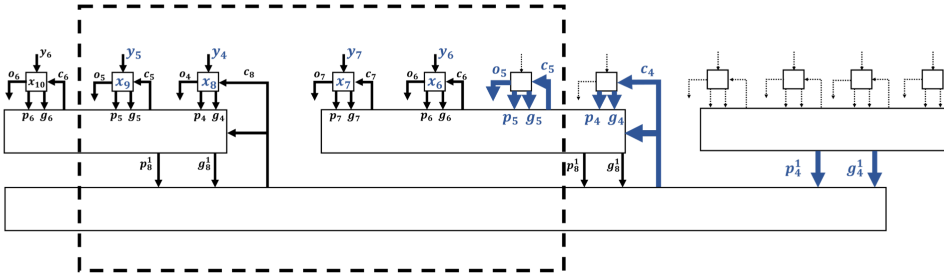

Fig. 14. An intermediate state in the CLA after scanning up to the sixth column. The box contains the columns of the critical strip. The blue variables are assigned while the blank variables were previously assigned, but then erased. Notice that we remember the assignment to the output variables in the strip and forgot the assignment outside.

B will reach an assignment to just the last layer of circuits L and R. This will follow immediately from Lemma 5.7.

Lemma 5.7. *During the execution of Algorithm 1, the tableau variables within each layer of circuit L get assigned in row order from top to bottom. Furthermore, each tableau variable eventually receives an assignment.*

Likewise, the tableau variables in each layer $\ell > 0$ of circuit R get assigned in row order from bottom to top, and each tableau variable eventually receives an assignment.

Proof. We prove both properties in circuit L by induction, making use of the row-friendliness of Wallace tree multipliers from Lemma 5.2. It is clear that the initial layer satisfies both properties. Suppose that layer $\ell - 1$ satisfies both properties. Then its rows of adders $\{A^{xy}_{\ell-1,i,j'}\}_i$ get assigned to in ascending order with $j' = 0, 1, \ldots$. For each increment of j', by row friendliness the steps 5 and 7 yield an assignment to all the variables in tableau rows $2j', 2j' + 1$ of layer ℓ. So layer ℓ gets assigned in row order from top to bottom, and each tableau variable in ℓ eventually receives an assignment.

The proof for circuit R is symmetric, except the initial tableau is not assigned in horizontal rows, but rather diagonal rows. Nevertheless, the subsequent layer $\ell = 1$ will still satisfy both desired properties and the induction argument may be used from there. □

Corollary 5.8. *At the end of Algorithm 1, the branching program B reaches an assignment to precisely both rows in the last layer of circuits L and R.*

To propagate an assignment to the last layer of L or R through the CLA, we will follow Algorithm 2. This algorithm will essentially perform a post-order traversal of the full CLA tree. While it is not technically necessary to include the components of the CLA to the right of the critical strip, we have retained them for clarity.

After running Algorithm 2 in both circuits L and R, we have an assignment to the outputs of both critical strips. By Lemma 5.5, this assignment violates an inequality constraint in E.

Size Bound. We claim that in the first phase, where the branching program B is executing Algorithm 1, each node in B is labeled by an assignment to at most four rows of tableau variables within each layer ℓ of L, and likewise for each layer $\ell > 1$ for R. By Lemma 5.7, the tableau variables within each layer are assigned in row order from top to bottom in L. So if four rows are assigned in a layer ℓ, they form a fully assigned row of adders $\{A^{xy}_{0,i,j}\}_i$. Algorithm 1 will propagate that assignment to the next layer, erasing the assignment to the row of adders $\{A^{xy}_{0,i,j}\}_i$. The same

ALGORITHM 2: Propagates from the inputs to the critical strip outputs of the CLA while assigning at most a constant number of bits per CLA layer.

```
1 for i = 0, 1, ..., 2n do
2 |   Branch on any unassigned inputs to the i-th column: t_{h,i,0}, t_{h,i,1}
3 |   while there is a pair of propagate and generate variables p_{ℓ,i'}, g_{ℓ,i'} with all their input variables
  |   assigned do
4 |   |   Propagate to p_{ℓ,i'}, g_{ℓ,i'} while merging to forget their input propagate and generate bits.
5 |   |   Merge to forget the carry-bits computed by the CLA that output p_{ℓ,i'}, g_{ℓ,i'}.
6 |   |   Propagate to each carry-bit with all its input variables assigned.
7 |   |   Propagate to each critical strip output bit with all its inputs assigned.
8 |   end
9 end
```

proof works to show that at most four rows of tableau variables are assigned within each layer $\ell > 1$ of R.

Each node in the first phase of B then holds an assignment to at most $8\delta h$ variables of the critical strip. Both L and R have at most $2n$ rows of tableau variables, so the number of tableau variables in the critical strip is upper bounded by $4nh$. Therefore, the execution of Algorithm 1 will take at most $4nh$ steps. As this algorithm is also oblivious, each node gets labeled by an assignment to one of $4nh$ sets of at most $8\delta h$ tableau variables. So the total number of nodes in the first phase of B is at most $4nh2^{\delta h} = 2^{16\log^2 n+O(\log n)}$.

We can obtain a more efficient version of Algorithm 1 by immediately propagating when an individual adder becomes fully assigned. This modified algorithm will only store at most two variables per subcolumn, except for a single "working" subcolumn in each layer that may hold three variables. This modification results in a size bound of $2^{8\log^2 n+O(\log n)}$.

We give a polynomial bound for the second phase, where the branching program B is executing Algorithm 2. Observe that this algorithm only keeps an assignment to variables within the sub-CLAs intersecting the i-th column. At most one sub-CLA in each of the $\log_4 n$ layers will intersect the i-th column, so there are $O(\log n)$ assigned variables in any step of Algorithm 2. The whole CLA has $O(n)$ variables, therefore B uses a polynomial number of nodes to execute Algorithm 2.

The total size of the branching program B is then $2^{8\log^2 n+O(\log n)}$.

Theorem 5.9. *There is a regular resolution proof of size $2^{8\log^2 n+O(\log n)}$ that $\phi^{\text{Wall}}_{\text{Comm}}(n)$ is unsatisfiable.*

Proof. As usual, we initially branch on the assignments $\sigma_e(k) = \{e_0 = 0, e_1 = 0, \dots e_k = 1\}$ 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 Lemma 5.6 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. □

5.3 Proofs of Wallace Tree Multiplier Distributivity

Our proof of commutativity for Wallace tree multipliers used Algorithms 1 and 2 to efficiently propagate an assignment from the initial layer of L's critical strip to the outputs. We will modify the branching step in these algorithms to verify the distributivity of Wallace tree multipliers.

Definition 5.10. Define a SAT instance $\phi^{\text{Wall}}_{\text{Dist}}(n)$ encoding the identity $x(y+z) = xy + xz$ in the usual way, with subcircuits $L^{y+z}, L^{x(y+z)}$ forming circuit L, $R^{xy}, R^{xz}, R^{xy+xz}$ forming circuit R, and inequality constraints E.

Theorem 5.11. *There is a regular resolution proof of size $2^{O(\log^2 n)}$ that $\phi^{\text{Wall}}_{\text{Dist}}(n)$ is unsatisfiable*

Proof (Sketch). We sketch the proofs for distributivity as they are simpler than the proofs for commutativity. The main difference is that we branch on the input variables $\mathbf{x}, \mathbf{y}, \mathbf{z}$ rather than the tableau variables in the initial layer.

We define critical strips in the usual way for each multiplier. There are at most $n + 2$ unconstrained carry bits in the $n + 1$-bit multiplier $L^{x(y+z)}$ and one unconstrained carry bit from the adder L^{y+z} for $n + 3$ total in L's critical strip. Together, the two n-bit multipliers R^{xy}, R^{xz} have $2n + 2$ unconstrained carry bits. The adder R^{xy+xz} contributes one more for a total of $2n + 3$ unconstrained carry bits in R's critical strip. So if our critical strip has width $\delta = \log(2n + 4)$, it will be unsatisfiable.

We now describe a branching program B that proves a given critical strip $\phi_{\text{Strip}}(k)$ is unsatisfiable. We begin the branching program B by running Algorithm 1 with the following modification: instead of branching on a row of initial tableau variables in some multiplier $\{t_{0,i,j}\}_i$, branching program B will instead branch on the input variables $\mathbf{x}, \mathbf{y}, \mathbf{z}$ and propagate to that row of tableau variables $\{t_{0,i,j}\}_i$. To reveal the rows from top to bottom in the initial layer of each multiplier's critical strip, we only need to assign a sliding window of δ bits in each input bit-vector $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The resulting branch order on $\mathbf{x}, \mathbf{y}, \mathbf{z}$ is the same as in our proof of array multiplier distributivity.

At the end of Algorithm 1, the branching program B reaches an assignment to the last layer of each multiplier $R^{xy}, R^{xz}, L^{x(y+z)}$. By using Algorithm 2, we propagate this assignment to the multiplier outputs $\mathbf{xy}, \mathbf{xz}$ and $\mathbf{x}(\mathbf{y} + \mathbf{z})$. Lastly, we propagate from $\mathbf{xy}, \mathbf{xz}$, through the CLA circuit L^{xy+xz}, to the final output $\mathbf{xy} + \mathbf{xz}$. Since the critical strip was unsatisfiable, the resulting assignment to $\mathbf{x}(\mathbf{y} + \mathbf{z})$ and $\mathbf{xy} + \mathbf{xz}$ must violate some equality constraint from E. □

5.4 Degree Two Identity Proofs for Wallace Tree Multipliers

Using the same ordering on the input variables and ideas from the proof of Theorem 3.20, we can prove the analogous result for Wallace tree multipliers.

Theorem 5.12. *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 bit-vectors $\mathbf{x}$ and $\mathbf{y}$.

If both $\otimes_1$ and $\otimes_2$ are 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 quasipolynomial 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 verifying arithmetic circuits containing multipliers. This pointed toward 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.

Thus, much of the recent research on multiplier verification has focused on using algebraic reasoning, in particular Groebner basis methods. The recent work of Ritirc et al. (2017) has improved the Groebner basis approach by dividing a multiplier into columns, and then incrementally checking that each column receives and transmits its carry-bits correctly. They find that this incremental method allows off-the-shelf computer algebra software to verify “simple” multiplier designs of up to 64 bits, though “optimized” multipliers still pose some difficulty.

We have shown that the conjectured resolution proof size barrier does not hold by giving the first small resolution proofs for verifying any degree two ring identity for the most common multiplier designs. 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. In light of our results and Ritirc et al. (2017), it seems that for verifying multipliers at the bit-level, the column-wise view is most natural. This is in contrast to the row-wise view taken, for example, in verifying multipliers at the word level. We remark that the critical strip decomposition is not only useful in the domain of resolution proofs. Other verification methods may find critical strips a useful testing ground, or could even benefit from checking each strip instead of the full multiplier all at once.

Given the historical success of CDCL SAT solvers for finding specific proofs, our results suggest a new path toward verifying nonlinear arithmetic. The proof size upper bounds we derived were conservative; we did not try to optimize the parameters. Nevertheless, the observed scaling of SAT solver performance on these problems suggests that they do not currently find proofs matching even these upper bounds. An important direction for improving SAT solvers 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 shorter 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.

APPENDIX

A PROOF OF LEMMA 5.2

In order to prove Lemma 5.2, we will first prove the following *smooth* and *singly peaked* properties of each layer of a Wallace tree multiplier. Define $\#\,\mathrm{Col}(\ell, i)$ to be the number of variables in subcolumn $\mathrm{Col}(\ell, i)$.

Definition A.1. We say that a layer l is *smooth* if for all pairs of adjacent subcolumns $\mathrm{Col}(\ell, i)$ and $\mathrm{Col}(\ell, i-1)$, we have

$$|\#\,\mathrm{Col}(\ell, i) - \#\,\mathrm{Col}(\ell, i-1)| \leq 2.$$

We say that a function $f : [0, N] \to \mathbb{Z}$ is *singly peaked* if there exists an integer k such that the following inequalities hold:

$$f(0) \leq f(1) \leq \cdots \leq f(k),$$
$$f(k) \geq f(k+1) \geq \cdots \geq f(N).$$

For any k satisfying these inequalities, we say that f attains its *peak* at k.

We say a layer ℓ of a Wallace tree Multiplier is *singly peaked* if the function $\#\operatorname{Col}(\ell, i)$ of the variable i is singly peaked. If $\#\operatorname{Col}(\ell, i)$ attains its peak at k, we also say that layer ℓ attains its peak at k.

Proposition A.2. *All layers of a Wallace tree multiplier are smooth.*

Proof. It is clear that the initial layer is smooth. Assume, for induction, that layer $\ell - 1$ is smooth. From our construction, for layers $\ell > 0$ we have the recurrence

$$\#\operatorname{Col}(\ell, i) = \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i) \right\rceil + \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i - 1) \right\rceil, \tag{1}$$

where the left term counts the number of sum-variables in $\operatorname{Col}(l, i)$ and the right term counts the number of carry-variables. Therefore, we can write

$$|\#\operatorname{Col}(\ell, i) - \#\operatorname{Col}(\ell, i - 1)| = \left| \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i) \right\rceil - \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i - 2) \right\rceil \right|.$$

To bound this expression, we use that by the smoothness of layer $\ell - 1$:

$$|\#\operatorname{Col}(\ell - 1, i) - \#\operatorname{Col}(\ell - 1, i - 2)| \leq 4.$$

This implies the bound

$$\left| \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i) \right\rceil - \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i - 2) \right\rceil \right| \leq 2. \qquad \square$$

Proposition A.3. *All layers of a Wallace tree multiplier are singly peaked.*

Proof. The initial layer is singly peaked. Assume, for induction, that layer $\ell - 1$ is singly peaked and attains a maximum at k. We show that layer ℓ is singly peaked and attains its peak at k or $k + 1$. Roughly, this follows because, from Equations (1), $\#\operatorname{Col}(\ell, i)$ is a sum of two singly peaked functions, one attaining its peak at k and the other attaining its peak at $k + 1$.

Consider the case where $i < i' \leq k$. We will show that $\#\operatorname{Col}(\ell, i) \leq \#\operatorname{Col}(\ell, i')$. We have, from Equations (1), that

$$\begin{aligned} \#\operatorname{Col}(\ell, i) &= \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i) \right\rceil + \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i - 1) \right\rceil, \\ \#\operatorname{Col}(\ell, i') &= \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i') \right\rceil + \left\lceil \frac{1}{3} \#\operatorname{Col}(\ell - 1, i' - 1) \right\rceil. \end{aligned} \tag{2}$$

Since $\#\operatorname{Col}(\ell - 1, x)$ was singly peaked in x, attained its peak at k, and $i - 1 < i \leq i' - 1 < i' \leq k$, we have

$$\begin{aligned} \#\operatorname{Col}(\ell - 1, i)\rceil &\leq \#\operatorname{Col}(\ell - 1, i'), \\ \#\operatorname{Col}(\ell - 1, i - 1)\rceil &\leq \#\operatorname{Col}(\ell - 1, i' - 1), \end{aligned}$$

implying, through Equations (2), that $\#\operatorname{Col}(\ell, i) \leq \#\operatorname{Col}(\ell, i')$.

Consider the other case where $k + 1 \leq i < i'$. We will show that $\#\operatorname{Col}(\ell, i) \geq \#\operatorname{Col}(\ell, i')$. Since $\#\operatorname{Col}(\ell - 1, x)$ is singly peaked in x, attained its peak at k, and $k \leq i - 1 < i \leq i' - 1 < i' \leq k$, we have

$$\begin{aligned} \#\operatorname{Col}(\ell - 1, i)\rceil &\geq \#\operatorname{Col}(\ell - 1, i'), \\ \#\operatorname{Col}(\ell - 1, i - 1)\rceil &\geq \#\operatorname{Col}(\ell - 1, i' - 1), \end{aligned}$$

implying, through Equations (2), that $\#\operatorname{Col}(\ell, i) \geq \#\operatorname{Col}(\ell, i')$.

Lastly, either $\#\operatorname{Col}(\ell - 1, k) \geq \#\operatorname{Col}(\ell - 1, k + 1)$ in which case $\#\operatorname{Col}(\ell, i)$ is singly peaked, attaining its peak at k, or $\#\operatorname{Col}(\ell - 1, k) < \#\operatorname{Col}(\ell - 1, k + 1)$ and $\#\operatorname{Col}(\ell, i)$ attains its peak at $k + 1$. $\square$

Together, these two properties imply that the rows of adders in each layer are arranged in a shallow pyramid.

Proposition A.4. *In each layer l of a Wallace tree multiplier*

(1) each row j of adders occupies a contiguous interval of columns between $minCol(j)$ to $maxCol(j)$;
(2) each row j of adders is arranged from the middle-out with full adders in the middle, then half adders, then wires at the ends;
(3) for row $j + 1 > 0$, we have the strict inequalities

$$minCol(j) < minCol(j + 1), \tag{3}$$

$$maxCol(j) > maxCol(j + 1). \tag{4}$$

Proof. Properties (1) and (2) follow from Proposition A.3: layer j is singly peaked
(Proof of 3). The (non-strict) inequalities

$$\text{minCol}(j) \leq \text{minCol}(j + 1), \tag{5}$$

$$\text{maxCol}(j) \geq \text{maxCol}(j + 1) \tag{6}$$

follow from layer j being singly peaked. The strictness of these inequalities follows from Proposition A.2 which states that layer j is smooth. For contradiction, suppose that

$$\text{minCol}(j) = \text{minCol}(j + 1).$$

Then we have at least two more adders in $\#\,\text{Col}(\text{minCol}(j))$ compared to the previous column. These are either full adders or half adders; therefore,

$$\#\,\text{Col}(\text{minCol}(j)) - \#\,\text{Col}(\text{minCol}(j) - 1) \geq 4$$

contradicting the smoothness of layer j. □

Having established this structure on the arrangement of adders in each layer, we prove Lemma 5.2. Recall the definition and statement of lemma as follows.

Definition A.5. For layer l, if for all $j \leq k$ the j-th row of adders

$$P_{l,i,j} \quad \text{for all} \quad i$$

outputs only to rows $2j, 2j + 1$ of the next layer $\ell + 1$'s tableau, we say that layer ℓ is *row-friendly* up to its k-th row of adders. If layer ℓ is row-friendly up to its last $h_l - 1$-th row of adders, we say that layer ℓ is *row-friendly*.

Lemma A.6. *In a Wallace tree multiplier, each layer $\ell \in [0, h - 2]$ is row-friendly.*

Proof of Lemma 5.2. Fix a layer ℓ. We show that if ℓ satisfies the properties in Proposition A.4, then it is row-friendly. We will use induction on the rows of adders in ℓ. □

Definition A.7. An interval of columns from $[i, i']$ is *flat* if they all contain the same number of variables.

It is clear from the Wallace tree multiplier construction 5 that the zero-th row of adders

$$P_{\ell,i,0} \quad \text{for all} \quad i \in [\text{minCol}(0), \text{maxCol}(0)]$$

outputs only to rows 0 and 1, so it is row-friendly. Furthermore, from our construction and Proposition A.4, after placing this row of adder outputs in layer $\ell + 1$, the resulting columns are flat from $i \in [\text{minCol}(0) + 1, \text{maxCol}(0)]$.

Assume, as an induction hypothesis, that ℓ was row-friendly up to row j and furthermore, that after the outputs for the j-th row of adders have been placed, layer $\ell + 1$ is flat from columns

$$i \in [\text{minCol}(j) + 1, \text{maxCol}(j)].$$

By the third property of Proposition A.4, we can restate this, saying that layer $\ell + 1$ is flat from columns

$$i \in [\text{minCol}(j + 1), \text{maxCol}(j + 1) + 1].$$

Since we added j rows of adders to these columns, each column contains $2j$ variables. These are precisely the columns in which we place the outputs for the $j + 1$-th row of adders. They occupy rows $2j$ and $2j + 1$ so that ℓ is row-friendly up to row $j + 1$. Additionally, we observe that if we append the $j + 1$-th row of adder outputs to the flat interval of columns

$$[\text{minCol}(j + 1), \text{maxCol}(j + 1) + 1],$$

the resulting columns are flat from

$$[\text{minCol}(j + 1) + 1, \text{maxCol}(j + 1)].$$

REFERENCES

Michael Alekhnovich and Alexander A. Razborov. 2002. Satisfiability, branch-width and tseitin tautologies. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS'02)*. IEEE Computer Society, 593–603. DOI: https://doi.org/10.1109/SFCS.2002.1181983

Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. 2002. A proof engine approach to solving combinational design automation problems. In *Proceedings of the 39th Design Automation Conference (DAC'02)*. ACM, 725–730. DOI: https://doi.org/10.1145/513918.514101

Fabrício Vivas Andrade, Márcia C. M. Oliveira, Antônio Otávio Fernandes, and Claudionor José Nunes Coelho, Jr. 2007. SAT-based equivalence checking based on circuit partitioning and special approaches for conflict clause reuse. In *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS'07)*, Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak, and Tomasz Garbolino (Eds.). IEEE Computer Society, 397–402. DOI: https://doi.org/10.1109/DDECS.2007.4295319

Paul Beame, Henry A. Kautz, and Ashish Sabharwal. 2004. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)* 22 (2004), 319–351.

Armin Biere. 2014a. Challenges in bit-precise reasoning. In *Formal Methods in Computer-Aided Design (FMCAD'14)*, 3. http://dx.doi.org/10.1109/FMCAD.2014.6987584

Armin Biere. 2014b. Where does SAT not work? In *BIRS Workshop on Theory and Applications of Applied SAT Solving*. http://www.birs.ca/events/2014/5-day-workshops/14w5101/videos/watch/201401201634-Biere.html.

Armin Biere. 2016a. Collection of combinational arithmetic miters submitted to the SAT competition 2016. In *Proceedings of SAT Competition 2016 – Solver and Benchmark Descriptions (Department of Computer Science Series of Publications B)*, Tomáš Balyo, Marijn Heule, and Matti Järvisalo (Eds.), Vol. B-2016-1. University of Helsinki, 65–66.

Armin Biere. 2016b. Weaknesses of CDCL solvers. In *Fields Institute Workshop on Theoretical Foundations of SAT Solving*. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.

Beate Bollig. 2011. Larger lower bounds on the OBDD complexity of integer multiplication. *Inf. Comput.* 209, 3 (2011), 333–343. http://dx.doi.org/10.1016/j.ic.2010.11.007

Beate Bollig and Philipp Wooelfel. 2001. A read-once branching program lower bound of $\Omega(2^{n/4})$ for integer multiplication using universal hashing. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing*. Hersonissos, 419–424.

Raik Brinkmann and Rolf Drechsler. 2002. RTL-datapath verification using integer linear programming. In *Proceedings of the ASPDAC 2002/VLSI Design 2002*, 741–746. http://dx.doi.org/10.1109/ASPDAC.2002.995022

Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*. 174–177. http://dx.doi.org/10.1007/978-3-642-00768-2_16

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. 2007. A lazy and layered SMT($\{\mathcal{BV}\}$) solver for hard industrial verification problems. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, 547–560. http://dx.doi.org/10.1007/978-3-540-73368-3_54

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. 2008. The MathSAT 4SMT solver. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*. 299–303. http://dx.doi.org/10.1007/978-3-540-70545-1_28

Randal E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* 35, 8 (1986), 677–691.

Randal E. Bryant. 1991. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.* 40, 2 (1991), 205–213. DOI: https://doi.org/10.1109/12.73590

Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. 1994. Symbolic model checking for sequential circuit verification. *IEEE Trans. Comput.-Aided Des. Integr Circuits Syst.* 13, 4 (1994), 401–424.

Samuel R. Buss and Maria Luisa Bonet. 2012. An improved separation of regular resolution from pool resolution and clause learning. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, Lecture Notes in Computer Science, Vol. 7313, 244–57.

Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. 2008. Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. *Log. Meth. Comput. Sci.* 4, 4 (2008). http://dx.doi.org/10.2168/LMCS-4(4:13)2008

Samuel R. Buss and Leszek Kolodziejczyk. 2014. Small stone in pool. *Log. Meth. Comput. Sci.* 10, 2 (2014). http://dx.doi.org/10.2168/LMCS-10(2:16)2014

Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *Commun. ACM* 7 (1960), 201–215.

Leonardo Mendonça de Moura. 2005. *System Description: Yices 0.1.* Technical Report. Computer Science Laboratory, SRI International.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24

Rina Dechter. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence (UAI'96)*, Eric Horvitz and Finn Verner Jensen (Eds.). Morgan Kaufmann, 211–219. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=370&proceeding_id=12.

Vijay Ganesh and David L. Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, 519–531. http://dx.doi.org/10.1007/978-3-540-73368-3_52

Edward Hirsch, Dmitry Itsykson, Arist Kojevnikov, Alexander Kulikov, and Sergey Nikolenko. 2005. Report on the Mixed Boolean-Algebraic Solver. *Technical Report, Laboratory of Mathematical Logic of St. Petersburg Department of Steklov Institute of Mathematics.* http://logic.pdmi.ras.ru/~basolver/basolver-firstreport.pdf.

Priyank Kalla. 2015. Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation. In *Proceedings on Formal Methods in Computer-Aided Design (FMCAD'15)*, 2.

Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2016. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.* 59, 2 (2016), 323–376. http://dx.doi.org/10.1007/s00224-015-9653-1

Jan Krajíček. 1996. *Bounded Arithmetic, Propositional Logic and Complexity Theory.* Cambridge University Press.

Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View.* Springer.

László Lovász, Moni Naor, Ilan Newman, and Avi Wigderson. 1995. Search problems in the decision tree model. In *SIAM J. Discrete Math.* 107 (1995), 119–132.

J. P. Marques-Silva and K. A. Sakallah. 1996. GRASP—A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design.* 220–227. DOI: https://doi.org/10.1109/ICCAD.1996.569607

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232).* 530–535. DOI: https://doi.org/10.1145/378239.379017

Openssl.org. 2016. OpenSSL Bug CVE-2016-7055. Retrieved from https://www.openssl.org/news/secadv/20161110.txt.

Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. 2004. An efficient finite-domain constraint solver for circuits. In *Proceedings of the 41st Design Automation Conference (DAC'04).* 212–217. http://doi.acm.org/10.1145/996566.996628

Stephen Ponzio. 1995. A lower bound for integer multiplication with read-once branching programs. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, 130–139.

Sherief Reda and A. Salem. 2001. Combinational equivalence checking using Boolean satisfiability and binary decision diagrams. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'01)*, Wolfgang Nebel and Ahmed Jerraya (Eds.). IEEE Computer Society, 122–126. DOI: https://doi.org/10.1109/DATE.2001.915011

Daniela Ritirc, Armin Biere, and Manuel Kauers. 2017. Column-wise verification of multipliers using computer algebra. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD'17)*. 23–30.

Martin Sauerhoff and Philipp Woelfel. 2003. Time-space tradeoff lower bounds for integer multiplication and graphs of arithmetic functions. In *Proceedings of the 35th Annual ACM Symposium on the Theory of Computing*, 186–195.

Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. 2016. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE 2016)*, Luca Fanucci and Jürgen Teich (Eds.). IEEE, 1048–1053. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7459464.

Received April 2018; revised March 2019; accepted March 2019