A Path Paved by Proof Complexity Towards Verifying Nonlinear Integer Arithmetic

Vincent Liew

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Paul Beame, Chair

Emina Torlak

Dan Suciu

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

A Path Paved by Proof Complexity
Towards Verifying Nonlinear Integer Arithmetic

Vincent Liew

Chair of the Supervisory Committee:

Professor Paul Beame

Computer Science and Engineering

Automated theorem provers have long struggled to efficiently reason about bit-precise properties of integer multiplication. Despite major advances in the efficiency of automated reasoning, from the Binary Decision Diagrams of the 1980s to the SAT solvers of today, integer multiplication has persisted as a major bottleneck in hardware and software verification.

In this thesis, we use proof complexity to pave a new path towards verifying nonlinear integer arithmetic. We propose that pseudo-Boolean solvers equipped with cutting planes reasoning have the potential to combine the complementary strengths of the existing SAT and algebraic approaches while avoiding their weaknesses.

We present several results on the proof complexity of fundamental multiplier identities. In the resolution proof system, we construct polynomial size proofs for degree two ring identities, refuting a widely believed conjecture that such proofs must be exponentially large. In the polynomial calculus proof system, we give optimal, $O(n^2)$ length proofs for word-level ring identities. But we also show that extracting simple bit-level consequences from a word-level property can require an exponentially large polynomial calculus derivation. In the cutting

planes proof system, we give optimal, $O(n^2)$ length proofs for a large class of degree two identities, at both the word-level and the bit-level.

We present experiments testing the CDCL SAT solving approach (corresponding to resolution) and the pseudo-Boolean approach (corresponding to cutting planes) that uncover the potential of using pseudo-Boolean solvers to efficiently reason with mixtures of arithmetic and bit-level constraints. We demonstrate that pseudo-Boolean solvers can verify, at both the word-level and bit-level, the commutativity of a multiplier as well as the equivalence of different multiplier architectures. We also find examples of simple nonlinear bit-vector inequalities that are intractable for current bit-vector and SAT solvers but easy for pseudo-Boolean solvers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Paul Beame, for his guidance and patience over the years. My growth into an independent researcher was only possible thanks to his generously-shared wisdom, careful effort, and compassionate spirit. One of the many valuable lessons that Paul has taught me over the years is the power of narrative, for both communication and understanding.

I would also like to thank Emina Torlak and Dan Suciu for serving on my reading committee, and Anup Rao and Matt Lorig for serving on my general dissertation committee.

Thank you to the UW Theory group, for being my academic home and a continuing source of inspiration. Math is best served with friends. I also must thank the UW PLSE group, for welcoming me into the world of verification with abundant enthusiasm and energy.

My fellow graduate students in the larger CSE community provided me with more support and camaraderie than I could have ever asked for. Thank you Yun-En, Isaac, Suchin, Shana, Dimitrios and Cyrus, and many others for the memories and support.

Thank you to Nick Porter for providing a great deal of assistance with implementing many of the SAT solver modifications and experiments presented in Chapter 4.

Many thanks to Jakob Nordström and the BARC and MIAO groups for hosting me at the University of Copenhagen and Lund University. This period of collaboration and the opportunity to bounce thoughts and ideas with Jan Elffers, Jo Devriendt, and Stephan Gocht, was one of the highlights of my time as a PhD student.

Going back to my undergraduate years at MIT, I want to thank Scott Aaronson for welcoming me into the world of research in theoretical computer science. As he would say, computer science is really "quantitative epistemology".

I must also thank my friends Max, Edward, and Gabe, who were always by my side throughout a tumultuous Course 8/18 journey . The memories make the math all the more meaningful.

Going back a bit further than that, I must thank Mr. Wilder, first for believing in me, and second for truly teaching me *What is Mathematics?* Thank you Zev, for your unyielding patience in showing me the way, and for lending me your copy of Rudin. Thank you Dr. Gaffney, for showing me physics from a theorist's point of view. Thank you Mr. Eldridge, for introducing me to Hume. Thanks Nehemiah and David, for always keeping it real.

Outside of the academic world, I want to thank the powerlifting and weightlifting communities for providing a second home where I can always fit in. Perhaps it takes a certain kind of personality to do this for fun! To the friends I have made in the IMA and at RCF— you have all helped me, more than you can know, to stay grounded.

Finally, to my family. Thanks Mom and Dad, for giving me the freedom to explore my curiosity and find my path. Thanks Austin, for your principled support and like-minded commiseration.

Thanks Virginia, for bringing color to my life. We made it!

DEDICATION

To my ancestors, for connecting us all.

To Mom, Dad, Austin, for your infinite patience, support, and wisdom.

To my radiant prism in a world of grey.

Chapter 1

INTRODUCTION

> "All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason."
>
> — Immanuel Kant, *Critique of Pure Reason* (1781)

1.1 Formal Verification: The Quest for Certainty

The world runs on computation. Hardware and software have become intricately woven into almost every thread of modern life. Every day we put our trust in millions, if not billions, of lines of code to control our banks, cars, phones, logistics and medical devices. Though this increasingly algorithmic landscape has undoubtedly brought along many benefits, our increasing dependence on computer systems has magnified the consequences of their inevitable defects, bugs, and security flaws.

A lesson that every programmer quickly realizes is that creating bug-free programs is extraordinarily difficult. The engineering required to write *correct, robust* and *secure* software is fundamentally more complicated than the engineering involved in building a bridge, airplane, or skyscraper. This is because modern programs typically contain so many interacting components and modules that, even if one had the will, the sheer number of possible paths that a program could follow makes testing every real-world scenario completely infeasible.

Unreliability and defects in computer programs can have disastrous consequences in industry. In 2012 the firm Knight Capital Group lost $440M in just half an hour due to a glitch in

its high frequency trading algorithms [123]. In 2018 and 2019, malfunctioning flight control software on the newly-developed Boeing 737 MAX airplane was linked to two crashes that killed 346 people, costing Boeing an estimated $18 billion over the next year [63]. As recently as March 2020, both the U.S. Department of Homeland Security and the USA Food and Drug Administration issued alerts [43,57] on a suite of microchip security vulnerabilities known as "SweynTooth" [62], that could allow malicious agents to take control of implanted medical devices such as pacemakers and insulin pumps.

The aim of formal verification is to provide *mathematical certainty* that a program behaves correctly. Though formal methods have been studied since the 1970s, algorithmic advances from just the last two decades have brought about a tidal wave of new, and highly practical applications. Formal verification has been applied to software of all kinds, including C compilers like CompCert [90], security monitors (Serval [110]), operating system kernels (Hyperkernel [111]), and hypervisors (Phidias [116]).

The increased reliability of formally verified software is not merely a theoretical nicety. Indeed, empirical studies have demonstrated that formally verified software is far less prone to errors when compared to unverified code. For example, the randomized testing tool Csmith [146] found nearly 300 bugs in the compilers GCC and LLVM. In comparison, only a handful of bugs were found in CompCert, and they were all located in the unverified front-end code. No bugs were found in verified code. Another study [56], which focused on testing distributed systems, found no protocol errors in the verified systems, whereas such errors were found in all of the real-world, unverified systems that were studied.

1.2 Mathematical Certainty Through Symbolic Logic

Ever since the publication of Euclid's *Elements* (300 BCE), natural philosophers and modern scientists alike have held up the field of *mathematics* as the paragon of certain knowledge. The empirical rate of success enjoyed by mathematical deduction is compelling: to date, no

counterexample has ever been found for a correctly-proven theorem of mathematics.[1]

Formal verification uses *symbolic logic* as a framework to increase our certainty in the correctness of computer systems. The goal is to produce a mathematical proof of correctness that uses only a trusted set of deduction rules. These rules comprise a *proof system*.

The deduction rules should be as simple as possible for two main reasons: we must be confident that they are correct, and it must be easy to check that the rules were applied correctly. For example, if A, B and C are *propositions*, i.e. true/false statements, then the following inference, known as the *resolution rule*, satisfies both criteria.

1. If A or B is true, and
2. not-B or C is true, then

---

3. A or C is true.

This rule is simple enough that we can confirm its logical validity by listing all the possible assignments of true/false to the statements A, B and C satisfying both premises (1) and (2), and then observing that either A or C is true in each case regardless of the value of B.

The resolution rule is also a *syntactic* rule, meaning that we can reduce its description down to a rule for formatting symbols. This is particularly important for formal verification: no matter the "meaning" behind the true/false statements A, B and C, we can verify that a syntactic rule was applied correctly through a purely mechanical format-checking procedure.

At the beginning of this section, we said that no counterexample has ever been found for a correctly-proven theorem. The reader may have noticed that the accuracy of this statement depends on what exactly we mean by the phrase "correctly-proven theorem". After all, there are many examples in the history of mathematics where a widely accepted proof was

[1]While there are cases of counterexamples being discovered for theorems that were "proven," in the sense that a "proof" was accepted by the mathematical community, flaws in the proof were always later identified. A classic example is the analysis "theorem" that any continuous function on an interval has a differentiable point, which was "proven" by Ampere in 1806. Much later, in 1872, Weierstrass produced a startling counterexample.

later discovered to contain a fundamental flaw. The standard of proof required in formal verification is high enough to exclude these flawed proofs. We only consider a theorem "correctly-proven" once we have produced a *formal proof* (that is, a proof written in symbolic logic) that follows a set of *logically valid* and *syntactic* inference rules.

1.3 Three Steps to Proving Correctness

Now that we have clarified what we mean by "proof", we can describe the main steps of formally verifying the correctness of a computer program P.

1. Write a *specification* S, using symbolic logic, that defines what it means for the program P to be "correct".

2. Create a mathematical model M, using symbolic logic, that describes the behavior of the program P.

3. (Formal Verification) Generate and check a formal proof that the model M implies the specification S.

The first two steps intend to accurately capture the *real-world* behavior of the program P with mathematical precision. Whether the mathematical descriptions S and M are "correct" or not is always, to some extent, a subjective and empirical matter, and hence cannot be adjudicated through purely mathematical reasoning.[2]

This thesis focuses on the last step, "formal verification," which takes place after we have mathematically defined "correctness". If the verification step succeeds, then the remaining possibilities for error are in either the specification S, or in the logical model M (or, more rarely, in the proof-checking program). In such cases, analyzing the specification or model (or proof-checker) is usually far simpler than analyzing a large program.

---

[2]Ultimately, programs run on physical hardware, which is subject to all kinds of sources of error. For example, cosmic rays hitting a physical RAM chip can cause bits to flip unpredictably [149].

1.4 Tools of the Trade: SAT and SMT Solvers

Once we have translated the specification and description of a program into symbolic logic, we are faced with a purely mathematical problem: find and check a formal proof that the logical model M implies the specification S. Unfortunately, while we gain a great deal of certainty by expressing proofs in symbolic logic, a (literally) inhuman effort is necessary to find and check these low-level proofs. Deductions that might be "obvious" from our vantage point may require a formal proof of pages upon pages of densely packed symbols.[3]

Modern computers are indispensable tools for grappling with the complexity of symbolic logic. Verifying proofs by computer is easy both in theory and in practice. Low-level proofs that might take a human several years to check can now be verified in seconds by a simple computer program. Furthermore, we can examine the source code of a proof checking program, but we have no such access to the inner workings of a proof checking mathematician. Shifting the burden of proof-checking to computers does not just avoid human error, it also transforms the process of checking a proof into a perfectly reproducible experiment.

On the other hand, *finding* these formal proofs is theoretically intractable. The shadow of *computational complexity* looms large: the widely believed conjecture $P \neq NP$ implies that, in a rather general sense, efficient proof search is impossible. Despite this, it turns out that many of the problems facing practitioners in real life are actually easy for *automated theorem provers*, in large part due to fundamental advances in proof search algorithms.

In particular, recent advances in *Satisfiability* (SAT) solvers and *Satisfiability Modulo Theories* (SMT) solvers have transformed these tools into efficient, general-purpose reasoning engines that can be applied to many of the verification problems that arise in practice. A feature of these solvers that is particularly useful for finding bugs in software is their ability to produce an explicit *counterexample* when asked to prove a false statement.

---

[3] An infamous early example is the many pages of symbols used to define and prove $1+1=2$ in Whitehead and Russel's *Principia Mathematica*.

As one would expect, SAT solvers solve the SAT problem, which asks: given a Boolean formula in CNF form, is there an assignment to the variables that satisfies the formula? The following is an example of a Boolean formula in CNF form:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3).$$

Each x_i above is a Boolean (i.e. true/false) variable. We have used the notation of symbolic logic, so that $\vee$ means OR, $\wedge$ means AND, and $\neg$ means NOT. This formula is *satisfied* (i.e. made true) by the assignment $x_1 = \texttt{true}, x_2 = \texttt{true}$, and $x_3 = \texttt{false}$.

If no satisfying assignment exists, the SAT solver will output "unsatisfiable". In the process, the SAT sover will have effectively traced out a *proof* of unsatisfiability. Otherwise, if a satisfying assignment exists, then the solver outputs such an assignment, which itself serves as the proof of satisfiability.

SMT solvers decide the satisfiability of *SMT formulas*, which generalize SAT formulas. Just like a SAT solver, an SMT solver outputs "unsatisfiable" if no satisfying assignment exists (in the process, tracing out a proof), and otherwise outputs a satisfying assignment. While the atoms of a Boolean formula are Boolean variables, the atoms of an SMT formula are *predicates*, that is, true/false statements about non-Boolean variables, for example integers, real numbers, or arrays. We interpret each statement in the context of a background *theory*, for example the theory of integers, the theory of real numbers, or the theory of arrays. Each theory specifies a set of true statements about a particular domain of non-Boolean variables. For example, the following is an SMT formula whose atoms belong to the *theory of bit-vectors*:

$$(\mathbf{x} \leq 011) \wedge (\mathbf{y} + \mathbf{z} \leq 111).$$

Each $\mathbf{x}, \mathbf{y}, \mathbf{z}$ above is a *bit-vector* variable of some fixed length, which in this example is 3. One satisfying assignment for the above formula is $\mathbf{x} = 010$, $\mathbf{y} = 110$, $\mathbf{z} = 001$.

In general, a length n bit-vector $\mathbf{x}$ is an array of 0/1 variables $x_0, x_1, \ldots, x_{n-1}$. We can

view an n-bit-vector $\mathbf{x}$ in two ways: as the number represented by the n-bit binary string "$x_{n-1}x_{n-2}\ldots x_1x_0$", on which we can perform arithmetic operations such as addition and multiplication, or as a bit-string on which we can perform bit-level operations such bit-wise AND, or extracting a sub-string.

Peering inside the workings of an SMT solver, one will find a SAT solver, as well as dedicated solvers for each theory. The efficiency of today's SMT solvers hinges on the efficiency of each of these individual solvers. Accordingly, a major direction of SMT research focuses on maximizing the efficiency of each solver.

Of the many solvers in the SMT arsenal, the SAT solver plays a central role. First, SAT solvers are used to reason with the logical structure of an SMT formula. Second, many theory-solvers make heavy use of SAT-solvers as a subroutine. In particular, all bit-vector solvers rely on converting the problem into SAT, and then letting the SAT-solver do the rest of the work.

1.5 Verification with Bit-Vectors

The overarching goal of this thesis is to improve solvers for the theory of *fixed-width bit-vectors*. The theory of bit-vectors is particularly useful for hardware and software verification because it naturally captures both the bit-level and word-level behavior of the integer representations used by most modern CPUs and programming languages. For example, we can directly model the ubiquitous 32-bit `int` data type by using a length 32 bit-vector. And most `int` operations, for example `+` (addition) and & (bit-wise AND), have a direct bit-vector counterpart.

We now illustrate an example where we use a bit-vector solver to verify a snippet of real-life C code. Consider the standard implementation of the "minimum" function, `min`, shown in Figure 1.1, and the branchless version `branchless_min` shown in Figure 1.2. While the standard implementation seems correct almost by definition, it is much less clear whether the strange series of bit-wise operations in the branchless implementation is equivalent. Yet chances

```c
int min(int x, int y) {
    return (x > y) ? y : x ;
}
```

Figure 1.1: C implementation of the function $\min(x, y)$. In words, it says "If x is greater than y, return x. Otherwise, return y."

```c
int branchless_min(int x, int y) {
    return y ^ ((x ^ y) & -(x < y));
}
```

Figure 1.2: "Branchless" C implementation of $\min(x, y)$ using only arithmetic and bit-wise operations. The symbol ^ is bit-wise XOR and & is bit-wise AND.

are that you, in your everyday life, use programs that rely on the equivalence of branchless programs. Compilers often optimize performance by replacing standard implementations of common functions, such as `min`, with a branchless version.[4]

To verify that `branchless_min` is equivalent to `min`, we will show that the outputs are always equal. More precisely, we will show that with the same inputs x, y, the outputs of the two programs cannot ever be *unequal*.

In terms of the "three steps to proving correctness" from Section 1.3, this approach goes as follows. For Step 1, our specification is a logical formula ϕ_{equals} that asserts that the outputs are equal. For Step 2, our mathematical model M is the AND of two bit-vector formulas $\phi_{\text{branchless}} \wedge \phi_{\text{min}}$, where $\phi_{\text{branchless}}$ is a bit-vector formula that models the program `branchless_min`, and ϕ_{min} is a formula modeling the program `min`. For Step 3 we will use an SMT solver to prove that the model $M = \phi_{\text{branchless}} \wedge \phi_{\text{min}}$ implies the specification $S = \phi_{\text{equals}}$.

Step 1: Let the 32-bit-vectors $\mathbf{s}_{\text{branchless}}$ and $\mathbf{s}_{\text{min}}$ represent the outputs of `branchless_min` and `min` respectively. We define the bit-vector formula $\phi_{\text{equals}} := \left(\mathbf{s}_{\text{min}} = \mathbf{s}_{\text{branchless}}\right)$.

Step 2: We first write a bit-vector formula ϕ_{min} to model the standard implementation `min`. Representing the inputs using the 32-bit-vectors $\mathbf{x}$ and $\mathbf{y}$, we model the code from

[4]For example, the compiler `gcc`, on its most commonly used optimization setting `-O2`, attempts to transform functions such as `min, max, abs`, into branchless equivalents.

Figure 1.1 with the following formula:

$$\phi_{\text{min}} := \big((\mathbf{x} > \mathbf{y}) \rightarrow (\mathbf{s}_{\text{min}} = \mathbf{y})\big) \ \wedge \ \big(\neg(\mathbf{x} > \mathbf{y}) \rightarrow (\mathbf{s}_{\text{min}} = \mathbf{x})\big).$$

Next, we construct a bit-vector formula $\phi_{\text{branchless}}$ to capture the behavior of `branchless_min`. Note that the syntax `-(x < y)` in Figure 1.2 implicitly converts from a Boolean to a 32-bit `int`. We handle this type conversion in the first line of the following bit-vector formula, where the 32-bit-vector t represents the term `-(x < y)`:

$$\begin{aligned}\phi_{\text{branchless}} := \big((\mathbf{x} < \mathbf{y}) \rightarrow (\mathbf{t} = 1)\big) \ \ &\wedge \big(\neg(\mathbf{x} < \mathbf{y}) \rightarrow (\mathbf{t} = 0)\big) \ \wedge \\ &\big(\mathbf{s}_{\text{branchless}} = \mathbf{y} \ \wedge \big((\mathbf{x} \ \oplus \ \mathbf{y}) \ \& \ \mathbf{t}\big)\big).\end{aligned}$$

In the second line we have directly modeled the remainder of `branchless_min`.[5]

Step 3: We can now use an SMT solver to prove that the model M implies the specification S. More precisely, we will prove that the formula $\phi := (\phi_{\text{branchless}} \wedge \phi_{\text{min}}) \rightarrow \phi_{\text{equal}}$ is always true. This is equivalent to proving that the negation of ϕ:

$$\neg\phi = \phi_{\text{branchless}} \wedge \phi_{\text{min}} \wedge \neg\phi_{\text{equal}},$$

is unsatisfiable (always false)— a perfect problem for an SMT solver.

If the SMT solver outputs that $\neg\phi$ is "unsatisfiable", then it has proven the equivalence of the two implementations. There are no inputs x, y such that $\mathbf{s}_{\text{min}} \neq \mathbf{s}_{\text{branchless}}$. Otherwise, the solver will produce a concrete pair of values for $\mathbf{x}$ and $\mathbf{y}$ (and the rest of the bit-vectors in the formula) on which the two implementations disagree.

It does turn out that the $\neg\phi$ is unsatisfiable, but you, the reader, do not need to take our

[5] In order to avoid mixing up with the AND symbol $\wedge$, we have represented bit-wise XOR using the symbol $\oplus$ instead of the symbol $\hat{}$ used in C.

```
(set-logic QF_BV)
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const s_min (_ BitVec 32))
(declare-const s_branchless (_ BitVec 32))
(declare-const t (_ BitVec 32))

; Standard implementation of min.
(assert (=> (bvsgt x y) (= s_min y)))
(assert (=> (not (bvsgt x y)) (= s_min x)))

; Conversion from Boolean to int for term -(x < y).
(assert (=> (bvslt x y) (= t (_ bv1 32))))
(assert (=> (not (bvslt x y)) (= t (_ bv0 32))))
; Rest of branchless_min.
(assert (= s_branchless (bvxor y (bvand (bvxor x y) (bvneg t)))))

(assert (not (= s_min s_branchless)))
; If z3 outputs unsat, then we have successfully verified equivalence.
(check-sat)
```

Figure 1.3: The bit-vector formula $\neg\phi$ written in the SMT-LIB2 format.

word for this. You may see for yourself that the bit-vector formula $\neg\phi$ is unsatisfiable by following the link: https://rise4fun.com/Z3/EhvX, where we have written this formula in the standard SMT-LIB2 [7] format, as shown in Figure 1.3. From your browser, you can try running the SMT solver z3 [47] on $\neg\phi$ and confirm that it outputs "`unsat`". In this way, we can convince you that the two implementations `min(x,y)` and `branchless_min(x,y)` are indeed equivalent, and you need not even think about how the latter implementation even works!

1.6 The Multiplication Bottleneck

This seems almost too good to be true. Using bit-vectors, we were able to describe our program and its specification, and the SMT solver was able to fill in a proof of correctness with no further human input. In the small example above, the solver had no trouble finding a proof quickly. But how does this efficiency scale as we approach larger and more complicated

problems?

Though deciding bit-vector formulas is NEXPTIME-complete in general [86], current bit-vector solvers are highly efficient on problems arising in practice [23,47,59,79,99,104,132], and they continue to improve year-over-year alongside SAT solvers. Bit-vector solvers primarily solve formulas by *bit-blasting*, which converts a given bit-vector formula into a SAT formula, which is then fed into a SAT solver. This is often a very effective method due to the sheer efficiency of current SAT solvers, which can typically handle SAT formulas with millions of bits.

However, SAT and bit-vector solvers face a glaring weakness: *multiplication*. Proving even simple properties of multiplication, for example *commutativity* ($\mathbf{xy} = \mathbf{yx}$), for merely 16-bit-vectors, is already intractable for SAT solvers [15, 16]. In comparison, these solvers have no problem proving commutativity of addition ($\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$) for bit-widths into the thousands. This weakness of SAT-solvers translates to bit-vector solvers having a particularly difficult time reasoning with bit-vector formulas containing multiplication.

So far, bit-vector solvers have dealt with this obstacle of multiplication by applying sophisticated preprocessing and simplification procedures before ultimately handing off the resulting formula to a SAT-solver. For example, the bit-vector expression $\mathbf{xy} + \mathbf{yx}$ might be rewritten to $2(\mathbf{xy})$ by applying commutativity and distributivity. Though these word-level methods certainly help to alleviate the burden of multiplication on the SAT solver, there will always be formulas that evade simplification. In this thesis, we focus on the root of the problem: the SAT-solver.

1.7 Proof Complexity

We will draw heavily from the perspective of *proof complexity*, the field studying the *size* of formal proofs. Recall that when a SAT or SMT solver finds that a formula is unsatisfiable, it has effectively traced out a *proof* of unsatisfiability. So the shorter the proof, the more quickly the solver can finish running.

At first, one might expect SAT solvers to use a complicated, long list of proof rules, since the more rules you have, the more choices you have to construct a small proof. But the drawback of having more choices is that the correct choice may become much harder to find. For the moment, simplicity has won. For all their might, SAT solvers only use a single inference rule: *resolution*. (The same rule that we gave as an example earlier in Section 1.2.) It turns out that the proof of unsatisfiability that SAT solvers trace out is a *resolution proof*.

While there are solvers based on more complicated proof systems, for example the *cutting planes* proof system or the *polynomial calculus* proof system (these will be formally defined in Chapter 2), in practice the additional complexity of the proof rules usually outweighs the potential benefits of finding smaller proofs.

However, there are fundamental gaps in the resolution-based proof search of SAT solvers. From proof complexity, we know of many SAT formulas that require exponentially large resolution proofs of unsatisfiability. As we would expect, even small instances of these problems, containing only a few hundred variables, can quickly become intractable for SAT-solvers.

The observed inability of SAT solvers to verify any non-trivial properties of bit-vector multiplication[6] led many to suspect the following *multiplication barrier conjecture*[7], which provided the starting point for this work:

Multiplication Barrier Conjecture Checking nonlinear properties of bit-vector multiplication, such as *commutativity*, require exponentially long resolution proofs.

It was natural to expect this multiplication barrier conjecture since previously, in the 90s, exponential lower bounds for multiplication were shown for pre-SAT verification algorithms based on *decision diagrams* [27].

---

[6]Several conference and workshop presentations from 2014-2016 [13,14,16] highlighted this weakness. [14] pointed out the difficulty of proving multiplier commutativity, and [16] gave data showing that SAT solving times scale exponentially for this problem.

[7]This conjecture was made explicit in a 2016 talk by Armin Biere [16].

1.8 Our Contributions

1.8.1 Polynomial Size Resolution Proofs

We refute the above conjecture by constructing polynomial size resolution proofs for arbitrary degree 2 identities involving bit-vector addition and multiplication. As special cases, for bit-width n we give $O(n^6 \log n)$ size resolution proofs for multiplier commutativity as well as distributivity. This work appears in the following publications.

> Paul Beame, Vincent Liew. Towards Verifying Nonlinear Integer Arithmetic. In *Computer Aided Verification*, volume 10427 of *LNCS*, pages 238-258. Springer, 2017. [11]
>
> Paul Beame, Vincent Liew. Towards Verifying Nonlinear Integer Arithmetic. In *J. ACM*, 66(3):22:1-22:30, 2019. [12]

The key idea leading to these polynomial size proofs is to divide each multiplier up into thin, unsatisfiable strips. Each of these strips has a polynomial-size resolution proof of unsatisfiability. By combining the proofs for each strip, we can produce a polynomial-size proof for the full multiplier.

The existence of these polynomial size proofs suggested a new path towards verifying non-linear arithmetic: improve the proof search heuristics of SAT-solvers so that they can find these proofs.

1.8.2 SAT Experiments and the Need for Stronger Reasoning

With our polynomial-size proofs in hand, the next step was to try and experimentally reproduce these proofs, using real-life SAT-solvers. Because of the apparent exponential scaling of off-the-shelf SAT solvers on multiplier problems [16], it was already long known that they were not finding polynomial size proofs. Thus, we needed to look for modifications to the SAT solving algorithm that could guide a real-life solver towards polynomial size proofs such

as ours.

We examined several modifications to the SAT-solving algorithm, evaluating the impact on performance, and on the qualitative structure of the underlying resolution proof. Though we were able to, in some cases, guide a SAT solver towards one of our proofs, the resulting runtimes, despite scaling polynomially, were too large for practice. These experiments indicated that although we could find a polynomial size proof, the degree of this polynomial bound was too large for practical purposes, even if a SAT-solver could find our proof perfectly. This suggested that we needed stronger methods of reasoning than resolution.

Two natural approaches for strengthening resolution-based reasoning are embodied by the proof systems *polynomial calculus* [38], which generalizes from clauses to polynomials, and *cutting planes* [41], which generalizes from clauses to linear 0/1 inequalities. Both of these proof systems can directly simulate resolution, and in some cases are exponentially better.

Polynomial calculus is suited towards "algebraic" forms of reasoning, so the problem of verifying multipliers is a natural fit for this proof system. In the last decade, algebraic methods based on polynomial calculus have emerged as the state-of-the-art for verifying the correctness of multipliers in isolation [36, 80, 81, 128, 135, 147, 148]. A key reason for the effectiveness of these methods is that polynomial calculus admits significantly smaller multiplier proofs than we know for resolution. Properties like correctness and commutativity of a multiplier circuit have short, $O(n^2)$-length polynomial calculus proofs, which is optimal since a multiplier circuit already takes $O(n^2)$ lines to describe.

Unfortunately, for non-algebraic problems, searching for a polynomial calculus proof is typically orders of magnitude slower than using a SAT solver to search for a resolution proof. To some extent, Boolean reasoning is an inherent weakness in the polynomial calculus proof system. One of the side-contributions of our work in Chapter 5, is showing that even if we derive the equation $\sum_i 2^i (x_i - y_i) = 0$, which says that the two n-bit-vectors $\mathbf{x}, \mathbf{y}$ represent the same binary value, an exponentially large polynomial calculus proof is required

to extract any of the individual bit-level equalities $x_i = y_i$. Because of obstacles like this, algebraic methods are unlikely to supplant the role of SAT for solving bit-vector formulas.

1.8.3 Optimal Length Cutting Planes Proofs

Instead, we propose that *pseudo-Boolean solvers* that search for *cutting planes* proofs have the potential to achieve the "best of both worlds", combining the strengths of algebraic methods with the efficiency of SAT solvers for Boolean reasoning. We give theoretical and experimental results that uncover the potential of the pseudo-Boolean approach. These contributions appear in the following publication.

> Vincent Liew, Paul Beame, Jo Devriendt, Jan Elffers, Jakob Nordström. Verifying Properties of Bit-vector Multiplication Using Cutting Planes Reasoning. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2020.

Our main theoretical result is the construction of optimal, $O(n^2)$-length cutting planes proofs for a wide range of degree two bit-vector identities, including commutativity and distributivity. We are able to prove these bit-vector identities not only at the word level, but also at the level of the individual bits. While $O\big(n^2\big)$-length polynomial calculus proofs are known for some of these properties at the word-level [82], we showed in this work that polynomial calculus cannot efficiently extract the individual bit-equalities.

Along the way to constructing these cutting planes proofs, we introduce a convenient new format called (k, d)-*cutting planes* for writing down these cutting planes proofs. Instead of writing a linear inequality in each line, as in standard cutting planes, the (k, d)-cutting planes format allows us to write *polynomial* $0/1$ inequalities of degree at most d that contain at most k nonlinear terms. We show that cutting planes can simulate a (k, d)-cutting planes derivation line-for-line with roughly a d^k factor of overhead in length.

Experimentally, we are able to use pseudo-Boolean solvers to verify the word-level equivalence of several different multiplier circuits of up to 256 bits in similar times to those of the best

algebraic methods. We find that these solvers can be particularly efficient at extracting all of the bit-level equalities from a word-level equality, which neither SAT solvers nor polynomial calculus can do efficiently.

We also show that pseudo-Boolean solvers are able to efficiently verify a number of bit-vector inequalities combining multiplication with bit-wise operations. In contrast, these inequalities are much harder or intractable for the top bit-vector solvers Boolector [23, 112], Z3 [47], Yices2 [51] and CVC4 [6]. These examples demonstrate some of the potential of pseudo-Boolean solvers for reasoning with nonlinear, bit-precise systems that are out of reach of current methods.

1.9 Roadmap

The rest of this thesis is organized as follows. In Chapter 2 we give some technical background material on SAT-solvers, bit-vector solvers, proof systems, and multiplier verification. In Chapter 3 we present our polynomial size resolution proofs for multiplier properties. In Chapter 4 we discuss and present data for our SAT solver experiments. In Chapter 5 we present our optimal length cutting planes proofs for multiplier properties. In Chapter 6 we discuss and present data for our pseudo-Boolean solver experiments. In Chapter 7, we present some partial results on the problem of whether there are small regular resolution proofs for associativity. Finally, Chapter 8 concludes this thesis with a discussion of future directions and remaining open problems.

Chapter 2

BACKGROUND

In this chapter we cover two important background topics for the rest of the thesis. First we present some of the standard circuit designs for addition and multiplication and set up the associated notation. Then we define some of the most important proof systems studied in proof complexity, resolution, polynomial calculus and cutting planes, and discuss their relationships to different algorithms for proof search.

2.1 Circuit Notation and Construction

We represent circuits as a set of constraints. In the context of SAT and resolution proofs, each constraint is encoded by a set of clauses. In polynomial calculus, each constraint is encoded by a set of polynomial constraints. And in cutting planes proofs, each constraint is encoded by a set of linear integer inequalities.

We write the i-th entry of a bit-vector $\mathbf{x}$ as a Boolean variable x_i. We typically refer to a circuit by the output bit-vector that it produces— for example we use $\mathbf{C}$ to refer to both a circuit and its output bit-vector, depending on the context. Often we write this output bit-vector in terms of the inputs, so that a multiplier circuit denoted by $\mathbf{xy}$ is understood to take input bit-vectors $\mathbf{x}, \mathbf{y}$ and output a bit-vector labeled $\mathbf{xy}$. We label the internal variables of a circuit $\mathbf{C}$ using the superscript C, for example: $t^C_{i,j}$.

2.1.1 Addition Circuits

Our circuits are built using *full adders* that output, in binary, the sum of three input bits. An adder is encoded as follows:

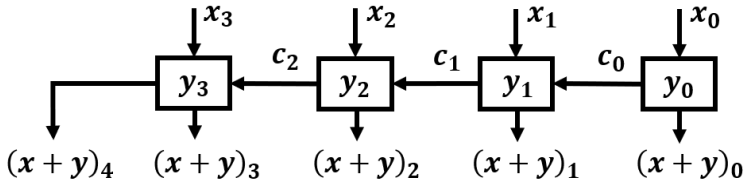

Figure 2.1: A 4-bit ripple-carry adder $\mathbf{x} + \mathbf{y}$. Each box represents a full adder with incoming arrows and outgoing arrows representing inputs and outputs.

Definition Let a_0, a_1, a_2 be inputs to a *full 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*.

Ripple-Carry Adder: A *ripple-carry adder* $\mathbf{x} + \mathbf{y}$, shown in Figure 2.1, takes in two bitvectors $\mathbf{x}, \mathbf{y}$ and outputs their sum in binary. In the i-th column, for $i < n$, 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 $(x + y)_i$ respectively. In the n-th column we place a wire A_n taking c_{n-1} as input and outputting to $(x + y)_n$. While the implementation is simple, it has depth n.

Carry-Lookahead Adder: A *carry-lookahead adder* uses a tree structure to add two bitvectors $\mathbf{x}, \mathbf{y}$ with only logarithmic depth. In contrast, an n-bit ripple-carry adder has linear depth. Notably, this circuit does not use full-adder components. We defer the full description of a carry-lookahead adder to Section 3.5.

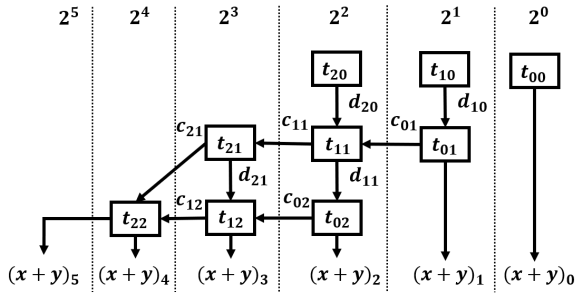

Figure 2.2: A 3-bit array multiplier.

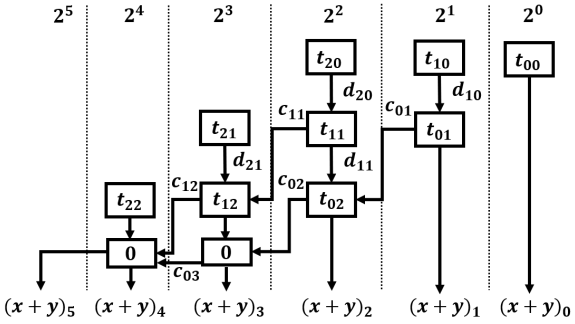

Figure 2.3: A 3-bit diagonal multiplier.

2.1.2 Multiplication Circuits

The multiplier circuits we describe all perform two phases of computation to compute **xy**. The first phase is identical in each multiplier: the circuit computes a *tableau* of partial products $x_i \wedge y_j$ for each pair of input bits x_i and y_j. These multiplier implementations 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 **xy** 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 $t_{i,j} = x_i \wedge y_j$, 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 2.2. 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 $t_{i,j}, d_{i+1,j-1}, c_{i-1,j}$ (replacing nonexistent variables with the constant 0). Adders of the form $A_{n,j}$ take input $c_{n,j-1}$ instead of $d_{i+1,j-1}$. Finally, we add constraints equating the sum-bits $d_{0,0}, d_{0,1}, \ldots, d_{0,n-1}, d_{1,n-1}$ with the corresponding output bits $(xy)_0, \ldots, (xy)_{n-1}$, and $\ldots, d_{n-1,n-1}$ with the corresponding output bits $(xy)_n, \ldots, (xy)_{2n-1}$.

Diagonal Multiplier: A diagonal multiplier uses a similar idea to the array multiplier. The difference is that the diagonal multiplier routes its carry bits to the next row instead of the same row as depicted in Figure 2.3.

Wallace Tree Multiplier: A Wallace tree multiplier takes a tree-based approach to summing the tableau. Using carry-save adders (parallel full 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. We defer the full description of a Wallace tree multiplier to Section 3.5.

Booth Encoding: This is a way to use a standard unsigned multiplier circuit to implement signed multiplication in two's complement notation. With this encoding, the tableau entries $t_{i,j}$ no longer represent partial products $x_i \wedge y_j$. There is also typically some additional logic that lets the circuit "skip" over consecutive strings of 1 or 0 digits in an input. An explicit example of a Booth encoded array multiplier is given by Hirsch, Itsykson, Kojevnikov and Kulikov in [73].

2.2 Proof Systems and Solvers

In this section we introduce three of proof complexity's most important proof systems: resolution, polynomial calculus and cutting planes, and their connections to automated theorem provers. See the survey [30] by Buss and Nordström for an excellent survey of these topics in greater depth.

2.2.1 Resolution Proofs and SAT Solvers

Definition 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:*

$$\frac{C \vee x \qquad D \vee \overline{x}}{C \vee D},$$

where C, D are clauses and x is a Boolean variable. We say that this inference *resolves* the clauses *on* x. A resolution proof *refutes* ϕ if it ends with the empty clause $\bot$. (We will use the terms "proof" and "refutation" interchangeably throughout this thesis, since each proof system provides proofs of unsatisfiability.)

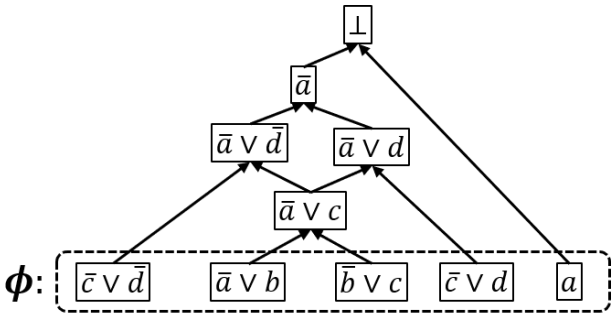

Figure 2.4: A bottom-up resolution proof that the SAT formula ϕ is unsatisfiable.

Figure 2.4 gives an example of a resolution refutation for the formula

$$\phi = (\bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{c} \vee d) \wedge a.$$

The resolution rule is *complete*, meaning that for any unsatisfiable formula ϕ, there is a resolution proof that derives the empty clause $\bot$. Since its appearance in the thesis of Blake [19], resolution has been generally considered to be one of the simplest complete, yet nontrivial, proof rules. Accordingly, some of the earliest automated theorem provers from the 1960s [45, 46, 131] were based on the resolution rule.

These early implementations converged towards the *DPLL algorithm* [45, 46], named after Davis, Putnam, Logemann and Loveland, which works by performing a depth-first search for satisfying assignments. A DPLL solver maintains a partial assignment ρ at all times, while simplifying the input SAT formula ϕ according to ρ. While ρ does not force the formula to be true or false, the solver selects an unassigned variable x and branches into the two cases where either the assignment $x = \texttt{true}$ or $x = \texttt{false}$ is added to ρ. This x is the *decision variable* for this step. In each case, the formula ϕ is simplified accordingly and the algorithm recurses. If the search ends without finding a satisfying assignment ρ, then the formula is unsatisfiable.

An important part of simplifying ϕ according to a new assignment is *unit propagation*. For a clause $C = l_1 \vee l_2 \vee \ldots \vee l_k$ in ϕ, if the current partial assignment ρ sets each literal to false except for an unassigned literal l_i, then unit propagation immediately sets l_i to true, since setting l_i to false immediately falsifies ϕ. Unit propagation can be viewed as a variable selection heuristic for DPLL. As we will see shortly, propagation plays a much more important role in modern SAT solvers.

The trace of a DPLL solver's execution on an unsatisfiable SAT formula corresponds to a restricted form of resolution proof known as *tree-like* resolution, in which the underlying proof structure is a tree (as opposed to a directed acyclic graph in general resolution). When the DPLL algorithm was introduced in the 1960s, it was capable of solving SAT formulas containing a few hundred variables, but struggled to scale much further. Around this time, the $P \neq NP$ conjecture rose to prominence, casting some pessimism on the possibility of practical SAT solving algorithms.

Remarkably, SAT solvers have made a resurgence in the last two decades. A long series of major (and minor) refinements to the DPLL framework have taken SAT solvers from problems with hundreds or thousands of variables to problems with millions of variables.

Perhaps the most significant improvement is the addition of *Conflict-Driven Clause Learning*

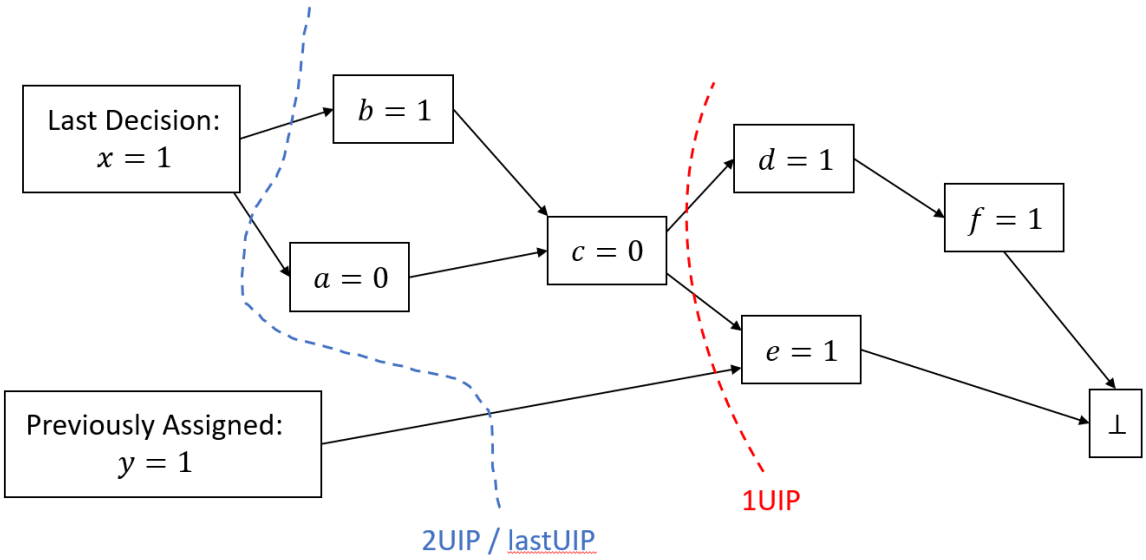

Figure 2.5: Example of a conflict graph. The blue cut corresponds to the 2UIP/lastUIP clause $\overline{x} \vee \overline{y}$ and the red cut corresponds to the 1UIP clause $c \vee \overline{y}$.

(CDCL) [100–102] to infer new clauses from the SAT formula ϕ. When a CDCL solver reaches an assignment ρ that falsifies ϕ, it runs a *clause learning* algorithm to learn a new clause C to block part of the falsifying assignment ρ. Adding the clause C to the formula ϕ prunes the search space, saving the SAT solver from repeating steps.

Unit propagation plays a central role in selecting an effective learned clause C. When a CDCL SAT solver branches on the value of a variable x, it keeps track of the resulting sequence of propagation steps using a *conflict graph*, as shown in Figure 2.5.

Each cut in the conflict graph that separates the assignments prior to and including the most recent decision x represents a possible learned clause. This clause will block the partial assignment along the cut. We can learn this clause because we know from the conflict graph that this partial assignment propagates to a conflict.

So which cut(s) do we choose? The most popular choice is to learn the clause corresponding to the *first unique implication point* (1UIP) of the conflict graph. This is the closest cut to

the conflict that contains only one assignment (node) that is either the most recent decision x, or is propagated after the decision x. The next-closest cut is the 2UIP, then the next is the 3UIP and so on, ending with the lastUIP.

Most clause learning schemes learn a subset of these UIP clauses because they are *asserting*: when the SAT solver undoes the last decision, the UIP clause will contain one unassigned literal which will immediately become assigned by unit propagation. In the example shown in Figure 2.5, if the SAT solver learns the 1UIP clause $c \vee \overline{y}$ and then undoes the decision $x = 1$, then we can immediately propagate $c = 1$. Learning the 1UIP is particularly popular because it is intuitively "closest" to the explanation for the conflict, and in practice it typically outperforms any other choice.

In terms of proof systems, the addition of clause learning to the DPLL framework upgrades the underlying proof system from tree-like resolution proofs to dag-like resolution proofs. It has been shown that if a CDCL solver can nondeterministically make all of the "right" decisions, it can polynomially simulate a special subsystem of resolution called *regular resolution* [32]. If the solver can also restart frequently enough, it can simulate general resolution up to a polynomial factor of overhead [5, 9, 121]. Of course, nondeterminism is not a realistic assumption for actual SAT solvers.

One crucial improvement of modern CDCL solvers is the *watched literals* scheme for much more efficient unit propagation. Even with watched literals, CDCL solvers typically spend $80 - 90\%$ of their time propagating.

There are also several important heuristics used in modern SAT solvers, including *variable state independent decaying sum* (VSIDS) [105] for selecting the next decision variable, *phase-saving* [120] to decide whether to assign a decision variable `true` or `false`, and restart heuristics to decide when to start the search over while keeping the learned clauses. For most of these, there is not a clear single choice that works best for every SAT instance. In many cases, finding the right combination of heuristics and parameters can dramatically

improve SAT solver performance.

The close connections between the resolution rule and SAT solving has made this proof system a prime target for finding upper and lower bounds on proof complexity. If a formula has a small resolution refutation, then SAT-solvers can potentially perform perform well— if its proof search heuristics guide it to a small proof. On the other hand, if we can prove large lower bounds on resolution proof size, then no matter how good a SAT solver's heuristics are, it cannot escape the task of walking through a large resolution proof.

2.2.2 Polynomial Calculus Proofs and Gröbner Basis Reduction

Definition Given a set of polynomials Φ over a set of variables $\{x_1, x_2, \ldots, x_n\}$ and a field K, each polynomial $p \in \phi$ represents the constraint $p = 0$. A *polynomial calculus* refutation of Φ is a sequence of polynomials ending with the polynomial 1 such that each line is either in Φ or is derived from the previous lines using the inference rules of linear combination and multiplication by a monomial m:

$$\frac{p \qquad q}{\alpha p + \beta q}\ (\alpha, \beta \in K), \qquad \frac{p}{m \cdot p}\ .$$

The polynomials $x^2 - x$ are also included as axioms for each variable x so that it only takes Boolean values. The polynomial p is interpreted to mean the equation $p = 0$.

Polynomial calculus is the proof system underlying *Gröbner basis reduction algorithms* [38]. Polynomial calculus can efficiently simulate a resolution step (as long as we allow for separate variables to represent the literals x and $\bar{x}$). Hard examples for polynomial calculus proofs include mod-p Tseitin tautologies [29].

To model a gate-level circuit implementation algebraically, each gate input and output is represented by a variable, and each gate is described by a polynomial constraint in the polynomial ring $K[X]$, where the field K is usually chosen to be the rational numbers $\mathbb{Q}$.[1]

[1]Other valid choices include $\mathbb{Z}$ and $\mathbb{R}$, but choosing $\mathbb{Q}$ seems to work best for verifying multiplier cir-

For example, an AND gate encoding $u = v \wedge w$ corresponds to the polynomial constraint $-u + vw = 0$. This encoding with polynomials allows polynomial calculus to express the following word-level specification for a multiplier circuit $\mathbf{xy}$, with input bit-vectors $\mathbf{x}, \mathbf{y}$:

$$\sum_{i=0}^{2n-1} 2^i (xy)_i - \Big(\sum_{i=0}^{n-1} 2^i x_i\Big)\Big(\sum_{i=0}^{n-1} 2^i y_i\Big) = 0. \tag{2.1}$$

If we can deduce this specification polynomial from the polynomial constraints modeling the gates of the multiplier circuit $\mathbf{xy}$, then the circuit is correct.

In general, the set of polynomials $\{f_0, f_1, \ldots\}$ encoding the gates of a circuit C generate an *ideal* denoted by $J(C)$. Due to its closure properties, this ideal contains all of the constraints derivable from $\{f_0, f_1, \ldots\}$ using polynomial calculus. So, the circuit C implies a given specification polynomial p_{spec} if and only if $p_{\text{spec}} \in J(C)$. Therefore, deciding whether a circuit C follows its specification reduces to deciding whether its specification polynomial p_{spec} is contained in the ideal $J(C)$.

For ideals of polynomial rings, the main tool for *ideal membership testing* is the theory of *Gröbner bases*. A Gröbner basis is a type of generating set for a polynomial ideal that can be used to decide ideal membership. This is done using a multivariate version of polynomial division. In a process termed *Gröbner basis reduction*, the specification polynomial p_{spec} is divided by each Gröbner basis polynomial $\{f_0, f_1, \ldots f_k\}$ for the ideal $J(C)$. There is no remainder afterwards if and only if $p_{\text{spec}} \in J(C)$.

2.2.3 Applications of Polynomial Calculus and Gröbner bases to Multiplier Verification

At this time, the most efficient methods for verifying isolated gate-level multiplier circuits use computer algebra [18,36,97,98,128,129,135,148]. Sayed-Ahmed, Große, Kühne, Soeken, and Drechsler leveraged Gröbner basis reduction to verify 64-bit integer multipliers in less than ten minutes and 128-bit multipliers in less than two hours [135] . Kaufmann, Biere

cuits [128].

and Kauers further refined the Gröbner basis approach in the series of papers [81, 128, 130], eventually combining it with SAT solving to produce independently certifiable polynomial calculus proofs of correctness for multipliers of up to 2048 bits. The length of these proofs was empirically found in [130] to scale as $O(n^2)$, which is optimal, since the size of a multiplier circuit is also quadratic in n. Shortly afterwards, Kaufmann et. al. confirmed in [82] that polynomial calculus does indeed have $O(n^2)$ length proofs of multiplier correctness. This efficient scaling gives some proof complexity theoretic explanation for how Gröbner basis methods have recently attained their efficiency: these methods can now find these short polynomial calculus proofs.

While the algebraic approach has demonstrated great efficiency for verifying multipliers in isolation, it remains to be seen whether this ability can be leveraged to verify larger circuits and formulas that contain multipliers as a smaller component. Unfortunately, for the non-algebraic parts of circuits, Gröbner basis methods are typically orders of magnitude slower than CDCL SAT-solvers. As mentioned in the Introduction, we give some theoretical explanation for this by proving an exponential polynomial calculus size lower bound in Chapter 5, Corollary 5.2.3, for the *bit-extraction* problem (where size is defined as the number of monomials appearing in the proof). This lower bound implies that an exponentially large polynomial calculus derivation is required to extract an individual bit-equality $x_k = y_k$ from the polynomial constraint $\sum_i 2^i (x_i - y_i) = 0$. Hence, most Gröbner basis algorithms require exponential time to make such bit-level inferences.

2.2.4 Cutting Planes Proofs and Pseudo-Boolean Solvers

Definition Given a *pseudo-Boolean formula*, a set of linear integer inequalities Φ over a set of variables $\{x_1, x_2, \ldots, x_n\}$, a *cutting planes* refutation of Φ is a sequence of linear integer inequalities ending with the inequality $0 \geq 1$ such that each line is either in Φ or is derived

from the previous lines using the inference rules of positive linear combination

$$\frac{\sum_i a_i x_i \geq b \qquad \sum_i a'_i x_i \geq b'}{\sum_i (\alpha a_i + \beta b_i) x_i \geq \alpha b + \beta b'}$$

where $\alpha, \beta \geq 0$, and the *division rule*

$$\frac{\sum_i (c \cdot a_i) x_i \geq b}{\sum_i a_i x_i \geq \lceil \frac{b}{c} \rceil} .$$

The *literal axioms* $-x \geq -1$ and $x \geq 0$ are also included for each variable x.

The cutting planes proof system [41] is the proof system underlying *pseudo-Boolean* solvers. Cutting planes is strictly stronger than resolution. It can efficiently simulate a resolution step, and can also efficiently refute the so-called *pigeonhole principle* formulas. Haken showed that these formulas require exponentially large resolution proofs in 1985 [70].

A significant advantage of using linear inequalities instead of clauses is that they can be much more expressive. Linear inequalities can write everything that clauses can: $l_1 \vee l_2 \vee \ldots \vee l_k$ translates directly into the linear inequality $\sum_{i=1}^{k} l_i \geq 1$. Linear inequalities are also able to write down word-level properties of bit-vectors. For example, we can directly set the value of a bit-vector $\mathbf{x} = b$ using the two inequalities $\sum_i 2^i x_i \geq b$ and $\sum_i 2^i x_i \leq b$. We can use this encoding to write a bit-vector equality $\mathbf{x} = \mathbf{y}$ using the two inequalities $\sum_i 2^i (x_i - y_i) \geq 0$ and $\sum_i 2^i (x_i - y_i) \leq 0$. In comparison, clauses can only write these properties in terms of blocked partial assignments to the individual bits.

There are currently two main approaches that are competitive for solving pseudo-Boolean formulas. One approach is to convert the formula into a CNF and then run a SAT solver to solve it. This approach is limited to finding resolution proofs. Examples of SAT based pseudo-Boolean solvers include MiniSat+ [52], Open-WBO [103] and NaPS [133].

The other approach is to generalize the CDCL algorithm to the pseudo-Boolean setting using cutting planes reasoning. Examples of conflict-driven pseudo-Boolean solvers include

Pueblo [136], Sat4j [89], and RoundingSat [54]. These solvers are still a comparatively young technology. Each uses the framework introduced by [34] to search for cutting planes proofs in a different way, with its own strengths and weaknesses. This is in contrast with CDCL solvers where the best ideas for conflict analysis have largely converged on a single method that is used by all of the best solvers.

An important weakness shared by each of these conflict-driven pseudo-Boolean solvers is that they fail to take full advantage of cutting planes reasoning. A manifestation of this is that when run on CNF inputs, pseudo-Boolean solvers collapse to CDCL solvers, only much less efficient ones because of the more advanced data structures needed. Further theoretical and experimental shortcomings in the ability of current solvers to perform cutting planes reasoning are reported in [53, 65, 141]. At this moment, there is substantial scope for developing new methods and heuristics for pseudo-Boolean solving that can carry out much more of this cutting planes reasoning in practice.

Chapter 3

POLYNOMIAL SIZE RESOLUTION PROOFS

3.1 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 [26] were extended in the 1990s to produce general methods for symbolic model checking [28] 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, as we discussed in the introduction to this thesis, there is an important area of formal verification where roadblocks that were identified in the 1980s still remain: verification of data paths within designs for Arithmetic Logic Units (ALUs), or indeed any verification problem in hardware or software that involves the detailed properties of nonlinear arithmetic. Natural examples of such verification problems in software include computations involving hashing or cryptographic constructions. At the highest level of abstraction, nonlinear arithmetic over the integers is undecidable, but the focus of these verification problems is on the decidable case of integers of bounded size, which is naturally described in the language of bit-vector arithmetic (see, e.g. [86, 88]).

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 showed in [27] 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 [20] and extended to include very tight exponential lower bounds for much more general diagrams than OBDDs, including FBDDs [21, 122] and general bounded-length branching programs [134]. On the other hand, CNF formulas can efficiently represent multipliers, but as we discussed in the Introduction, the advent of greatly improved SAT solvers has not advanced the verification of 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 [24, 88]) – 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. [22, 23, 25, 118]), 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 [117].

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 [13, 14, 16, 78].

As we discussed in the Introduction to this thesis, the dominant approach to bit-vector solving is to transform the given bit-vector formula into an equivalent CNF formula, and then dispatch the result to a SAT solver. The transformation process from bit-vector formula to CNF is called *flattening* [88], or more commonly *bit-blasting*. 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 (See Figure 3.1, and more detailed data in Chapter 4). As a result, the best of recent bit-vector solvers, e.g., Boolector [23],

Figure 3.1: Time to verify multiplier commutativity versus the bitwidth of the multiplier for SAT solvers Glucose and Lingeling [16]. See Table 6.1 and Chapter 4 for more detailed data and discussion.

MathSAT [25], STP [61], Z3 [47], and Yices [48] all rely on multiple rounds of preprocessing to reduce the expense of bit-blasting as much as possible.

When attempting to verify a multiplier circuit written as a CNF, a basic question arises: How do we write down its specification? The specification equation

$$\Big(\sum_{i=0}^{n-1} x_i\Big)\Big(\sum_{i=0}^{n-1} y_i\Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i\Big) = 0$$

does not have an obvious natural translation to clauses. A common approach is to try and compare it to a reference circuit that is known to be correct, in effect using this reference circuit to express the specification equation. This introduces a chicken-and-egg problem: how do we know that the reference circuit is correct?

Another approach to verifying a multiplier circuit is to check that it satisfies the right properties. A correct multiplier circuit must obey the multiplication identities for a commutative ring. If we check that each of these *ring identities* holds then the multiplier cannot have an error. This approach has the advantage that the specification of a multiplier circuit can be written *a priori* in terms of its natural properties, rather than in terms of an external reference circuit.

Empirically, however, modern SAT-solvers perform badly using either approach to problems of multiplier verification. Biere, in the text accompanying benchmarks on the ring identities submitted to the 2016 SAT Competition [15] 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?

As we discussed in Chapter 2, Modern SAT-solvers are based on a paradigm called conflict-directed clause-learning (CDCL) [100, 105] which can be seen as a way of breaking out of the backtracking search of traditional DPLL solvers [45]. When these solvers confirm the validity of an identity (by not finding a counterexample), their traces yield *resolution* proofs [9] 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., [31–33]).

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 make the following multiplication barrier conjecture that we stated in the Introduction [16]:

Multiplication Barrier Conjecture Checking nonlinear properties of bit-vector multiplication, such as commutativity, require exponentially long resolution proofs.

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 that we mentioned in Chapter 2: 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 [32]. Indeed, experience to date has been that the addition of some heuristics to CDCL suffices to find short regular resolution proofs that we know exist. The new regular resolution proofs that we produce are a key step towards developing such heuristics for verifying general nonlinear arithmetic.

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. [4] improving on earlier work [3, 126] which combined SAT solver and OBDD-based ideas for multiplier verification among other applications; however, there was no general

[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 [46]. 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.

understanding of when such methods will succeed.

Recently, alternative approaches to multiplier verification have been considered using computer algebra. Hirsch, Itsykson, Kojevnikov, Kulikov and Nikolenko designed a mixed Boolean-algebraic solver, BASolver [73], 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.

In Chapter 2, we discussed another approach using Gröbner basis algorithms that has recently come into prominence. 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. Unfortunately, for the non-algebraic parts of circuits, Gröbner basis methods can only handle problems several orders of magnitude smaller than can be handled by CDCL SAT-solvers and it remains to be seen whether it is possible to combine these to obtain effective verification for a general purpose software with nonlinear arithmetic or circuits that contain a multiplier as just one component of their design. In contrast, CDCL SAT solvers are already very effective for the non-algebraic aspects of circuits and are well-suited to handling the combination of different components; our work shows that there is no inherent limitation preventing them from being effective for verification of general purpose nonlinear arithmetic.

Roadmap: Section 3.3 gives our polynomial size regular resolution proofs for array multipliers. Section 3.4 describes how to extend these ideas to obtain short proofs for diagonal and Booth multipliers. Section 3.5 gives our quasipolynomial size regular resolution proofs for Wallace tree multipliers.

3.2 Notation and Preliminaries

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

Ring identities

We consider n-bit-vector identities from the commutative ring of integers $\mathbb{Z}_{2^n}$. A variable assignment is denoted by a set $\sigma = \sigma(x_0, x_1 \dots x_n) = \{x_0 = b_0, x_1 = b_1 \dots x_n = b_n\}$, where each $b_i \in \{0, 1\}$. $x_0, x_1, \dots x_n$.

Definition 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 3.2.1. *Let C_+ be an n-bit addition circuit corresponding to the operation $\oplus$ in $\mathbb{Z}_{2^n}$, and let $C_\times$ be an n-bit multiplier circuit $C_\times$ corresponding to the operation $\otimes$ in $\mathbb{Z}_{2^n}$. If C_+ correctly implements addition, $\mathbf{x} \otimes \mathbf{1} = \mathbf{x}$, and $\otimes$ is distributive over $\oplus$, then $C_\times$ implements multiplication correctly.*

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

Resolution proofs and branching programs

For convenience, we restate the definition of a resolution proof from Chapter 2.

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

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

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

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

Definition 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 thesis) 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 Krajicek [87]; the special case connecting ordered resolution and OBDDs for the conflict clause search problem was first observed in [95]. We include its proof for completeness.

Proposition 3.2.2. *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

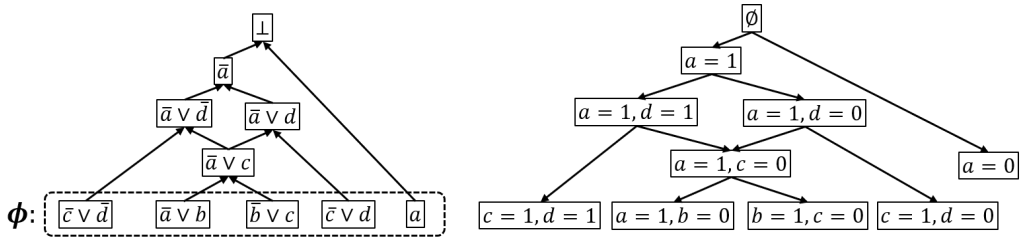

Figure 3.2: A regular resolution refutation for ϕ and the corresponding branching program.

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. □

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 3.3, 3.4, 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

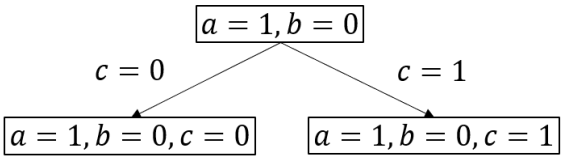

Figure 3.3: Branching on c.

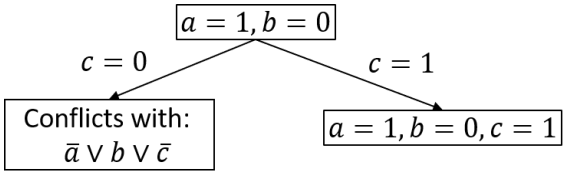

Figure 3.4: Propagating to $c = 1$.

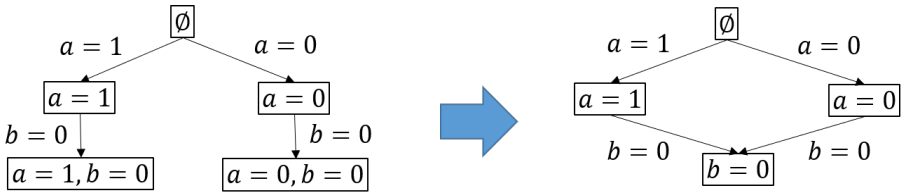

Figure 3.5: Merging on the common assignment $\{b = 0\}$.

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.

3.3 Array Multipliers

In this section 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 [49], implies that there is a polynomial-size ordered resolution refutation of the strip. In the case of commutativity, our argument implies that the constraint graphs for the strips can be combined to yield a single constraint graph of pathwidth $O(\log n)$. For the other identities, the orderings on the strips are different and the resulting constraint graphs only have small *branchwidth* which, by [1], still implies that there are small regular

resolution proofs of the other identities. Rather than simply invoke these general arguments, we give the details of the resolution proofs, along with more precise size bounds.

3.3.1 Proofs of Array Multiplier Commutativity

Definition We define a SAT instance $\phi^{\text{Array}}_{\text{Comm}}(n)$. The inputs are length n bitvectors $\mathbf{x}, \mathbf{y}$. Using the construction from Section 2.1, we define array multipliers $\mathbf{xy}$ and $\mathbf{yx}$. 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 $\mathbf{xy}$ and $\mathbf{yx}$, we add a final subcircuit E, a set of *inequality-constraints* encoding that the two circuits disagree on some output bit:

$$e_i = \left[(xy)_i \neq (yx)_i\right] \quad \forall i \in [0, 2n-1],$$

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

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 3.2.2. The variable order for B begins with $e_0, e_1, \ldots$, followed by the output bits $(yx)_0, (yx)_1, \ldots$. 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 $(xy)_0, (xy)_1, \ldots$, 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 $\mathbf{xy}$ and $\mathbf{yx}$ 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 $\mathbf{xy}$ and $\mathbf{yx}$ on the variables lying inside columns $[k - \Delta, k]$.

Definition 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 $\phi_{\text{Strip}}(k) \cap \mathbf{xy}$ the *critical strip* of circuit $\mathbf{xy}$ and likewise for circuit $\mathbf{yx}$.

Lemma 3.3.1. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k.*

Proof. We interpret each critical strip as a circuit that outputs the weighted sum of the input variables in circuits $\mathbf{xy}$ and $\mathbf{yx}$. 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 3.3.2. *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 $k \leq n$; 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 $\mathbf{xy}$, together with the carry variables from column $k - \Delta - 1$ of both $\mathbf{xy}$ and $\mathbf{yx}$. 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 $\mathbf{xy}$ 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 $\mathbf{yx}$ simultaneously are revealed from bottom to top. In circuit $\mathbf{yx}$ we maintain both a guess for its output values, and a row of sum-variables. From the proof of Lemma 3.3.1, if we have found that the outputs of $\mathbf{xy}$ and $\mathbf{yx}$ 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 Define Cut(0) as the set of variables containing

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

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

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

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

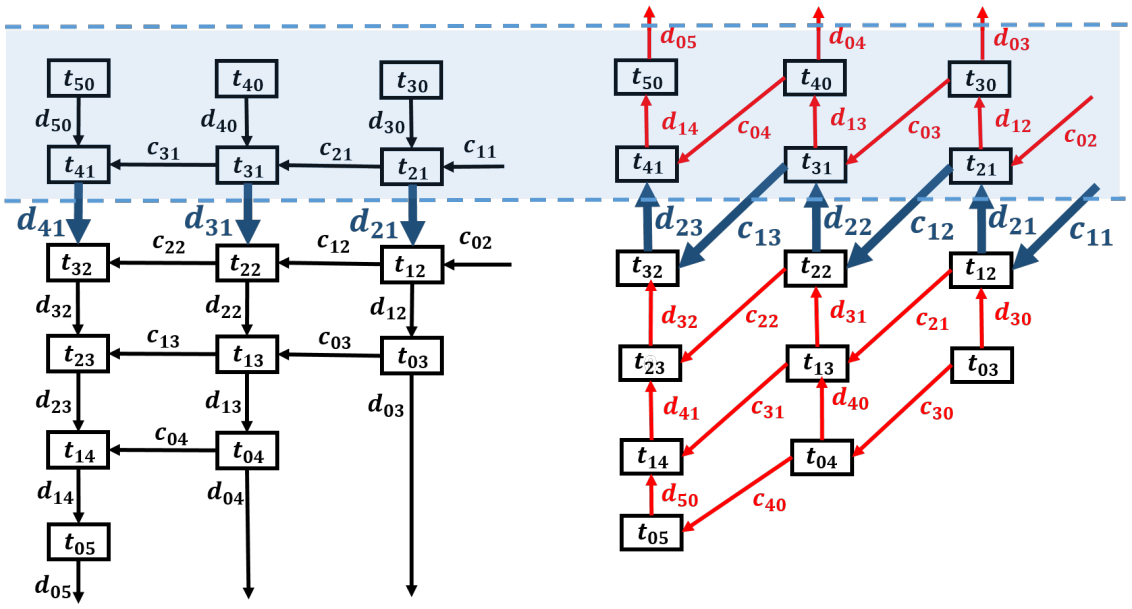

Figure 3.6: The critical strip $\phi_{\text{Strip}}(5)$ for checking commutativity. The enlarged variables belong to Cut(2) of $\phi_{\text{Strip}}(5)$. This cut divides the critical strip into a shaded satisfiable region and an unshaded unsatisfiable region.

indices are in-range:

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

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

Initialization: Throughout, we work in terms of the tableau variables in circuit **xy**, implicitly substituting t^{xy}_{ij} for t^{yx}_{ji}. 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 $(yx)_i$, then propagate to $d^{yx}_{0,i}$ using a clause from the constraint $(yx)_i = d^{yx}_{0,i}$. The surviving branches are those labeled by an assignment satisfying the constraints $(yx)_i = d^{yx}_{0,i}$. At this point we have reached nodes labeled Cut(0).

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

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

Then we propagate to the variables, in sequence,

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

from Cut(1) (notice that this does not include the input carry-bit $c^{yx}_{0,k-\Delta-1}$). 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 **xy**. If $j \in [k-\Delta, k]$ then we also propagate to $(xy)_{j-1}$.

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

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

We repeat this step until we have reached $\mathrm{Cut}(k+1)$. At this point we have an assignment to the critical strip output bits $\mathbf{xy}, \mathbf{yx}$. 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 3.3.1, this implies that our assignment to $\mathbf{xy}, \mathbf{yx}$ conflicts with an inequality constraint.

Size Bound: We show that there are $O(n^6 \log n)$ nodes in B. Each $\mathrm{Cut}(j)$ 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 $\mathrm{Cut}(j)$. 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:

Corollary 3.3.3. *$\phi_{\mathrm{Strip}}(k)$ 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 3.3.3

Theorem 3.3.4. *Let $N = |\phi^{\mathrm{Array}}_{\mathrm{Comm}}| = O(n^2)$. There is an $O(N^3 \log N)$ size regular resolution proof that $\phi^{\mathrm{Array}}_{\mathrm{Comm}}$ is unsatisfiable. There is an $O(N^{7/2} \log N)$ size* ordered *resolution proof that $\phi^{\mathrm{Array}}_{\mathrm{Comm}}$ 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 3.3.3 or Lemma 3.3.2 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. Corollary 3.3.3 yields the regular resolution proof and Lemma 3.3.2 yields the ordered resolution proof. □

3.3.2 Proofs of Array Multiplier Distributivity

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

We let E contain the usual inequality constraints. The full distributivity instance is then the union of the constraints in the above circuits: $\phi^{\text{Array}}_{\text{Dist}}(n) = \mathbf{x}(\mathbf{y} + \mathbf{z}) \cup \mathbf{xy} + \mathbf{xz} \cup E$.

We again divide the instance into critical strips, following the strategy previously used to refute $\phi^{\text{Array}}_{\text{Comm}}$.

Definition 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 $\mathbf{y} + \mathbf{z}$. 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 $c^{xy+xz}_i, (xy + xz)_i$ 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 3.3.5. *$\phi_{\text{Strip}}(k)$ is unsatisfiable for all k*

Proof. Like the proof of Lemma 3.3.1, the critical strip for $\mathbf{x}(\mathbf{y} + \mathbf{z})$ holds tableau bits with the same weighted sum (modulo 2^{k+1}) as those in $\mathbf{xz}$ and $\mathbf{xy}$ combined. The critical strip for $\mathbf{x}(\mathbf{y} + \mathbf{z})$ has at most n input carry-bits of weight $2^{k-\Delta}$. The critical strips of the n-

bit multipliers $\mathbf{xz}$ and $\mathbf{xy}$ each have at most $n-1$ input carry variables of weight $2^{k-\Delta}$. The critical strip of the adder $\mathbf{xy}+\mathbf{xz}$ has one input carry variable, so the critical strip decomposition of the two multipliers xy and xz has $2n-1$ input carry-bits. Since we set the width of the strip at $\Delta = \log(2n)$, it is unsatisfiable. □

Lemma 3.3.6. *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 $\mathbf{C} \in \{\mathbf{x}(\mathbf{y}+\mathbf{z}), \mathbf{xz}, \mathbf{xy}\}$ and $j \in [1, k-\Delta]$, we define $\text{Cut}^C(j)$ to be the set of variables

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

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

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

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: $\text{Cut}(j) = \cup_C \text{Cut}^C(j)$.

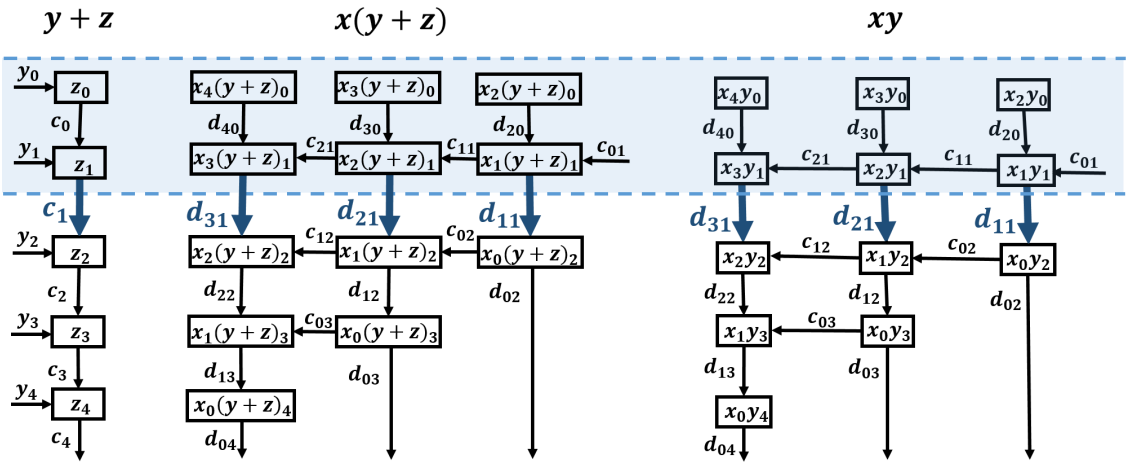

Figure 3.7: The critical strip $\phi_{\text{Strip}}(4)$ for distributivity. Cut(2) consists of the enlarged variables.

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_0^{y+z} and $(y+z)_0$, giving us an assignment to $\text{Cut}^{y+z}(0)$. The assignment to $(y+z)_0$, in turn, propagates to an assignment to the first row of tableau and sum variables from the critical strip for $\mathbf{x}(\mathbf{y}+\mathbf{z})$:

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

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

We then propagate the input variable assignments through the multipliers $\mathbf{xy}$ and $\mathbf{xz}$:

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

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

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

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

$$c^C_{i-j-2,j+1}, t^C_{i-j-1,j+1}, d^C_{i-j-1,j+1} \quad : \quad i \in [k-\Delta, k].$$

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

End: Beyond $\text{Cut}(k+1)$ Suppose that we have reached $\text{Cut}(k+1)$ and merged nodes. We branch on the input carry variable $c^{xy+xz}_{k-\Delta-1}$, that goes into the critical strip of ripple-carry adder $\mathbf{xy+xz}$. We can then propagate to the output bit-vector $(\mathbf{xy+xz})$. We now have an assignment to both $\mathbf{x(y+z)}, (\mathbf{xy+xz})$ that was propagated from one assignment to the input variables to the critical strip. By Lemma 3.3.5, this assignment conflicts with an inequality-constraint from E.

Size Bound: There are $k+1$ different global cuts $\text{Cut}(j)$. Each $\text{Cut}(j)$ section of B begins with an assignment to at most $4\Delta+1$ variables, and then branches on 3 input variables. So each section $\text{Cut}(j)$ 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 3.3.7. *There is an $O(n^6 \log n)$ size resolution proof that $\phi^{\text{Array}}_{\text{Dist}}(k)$ is unsatisfiable.*

Proof. At the root of this proof there are $2n$ branches each holding an assignment to $e_k, \ldots, e_1, e_0$. We refute each branch using the $O(n^5 \log n)$ size proof from Lemma 3.3.6. □

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

Definition We define a SAT instance $\phi^{\text{Array}}_{x(x+1)}(n)$. For the expression $x(x+1)$, we have the circuits $\mathbf{x} + \mathbf{1}$, consisting of a ripple-carry adder taking inputs $\mathbf{x}$ and $\mathbf{1}$ and outputting their sum, and $\mathbf{x}(\mathbf{x}+\mathbf{1})$, an array multiplier outputting the product $\mathbf{x}(\mathbf{x}+\mathbf{1})$. We construct a circuit for the expression $x^2 + x$, similarly.

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

$$\phi^{\text{Array}}_{x(x+1)}(n) = \mathbf{x}(\mathbf{x}+\mathbf{1}) \cup \mathbf{x^2} + \mathbf{x} \cup E.$$

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 $\mathbf{x^2}$ 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 Define the constant $\Delta = \log(2n-1)$. Let $\phi_{\text{Strip}}(k)$ contain the full ripple-carry adder circuit $\mathbf{x} + \mathbf{1}$ 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 3.8 shows an example of a critical strip.

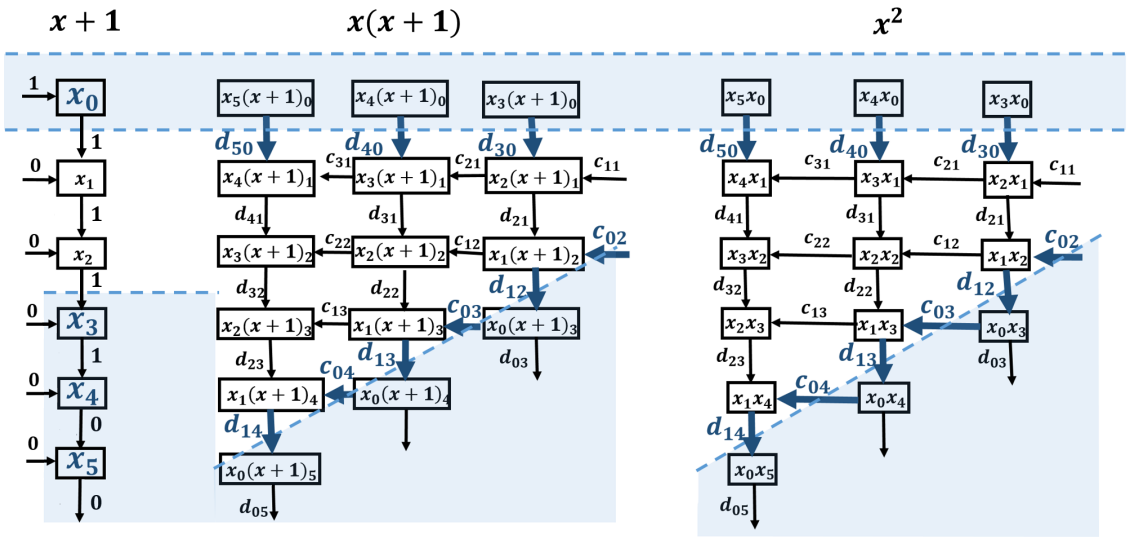

Figure 3.8: The critical strip $\phi_{\text{Strip}}(5)$ for checking $x(x+1) = x^2 + x$. The shaded region is satisfiable. The enlarged variables belong to Cut(1).

Lemma 3.3.8. *$\phi_{Strip}(k)$ is unsatisfiable for all k*

Proof. The proof is the same as the proof for Lemma 3.3.5. □

Definition 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 3.3.9. *There is a size $n^7 \log n$ regular resolution proof that $\phi_{Strip}(k)$ is unsatisfi-*

able.

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 $\mathbf{x}(\mathbf{x}+\mathbf{1}), \mathbf{x^2}+\mathbf{x}$. For the branches that don't conflict with an inequality-constraint, we branch on the values

$$(x^2)_i, x_i \quad : i \in [k-\Delta, k],$$

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

We observe that the carry variables in $\mathbf{x}+\mathbf{1}$ must be a sequence of 1s followed by 0s. 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

$$c_0^{x+1} = 1, \ldots, c_i^{x+1} = 1, c_{i+1}^{x+1} = 0, \ldots, c_k^{x+1} = 0.$$

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 the 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 $\mathbf{x} + \mathbf{1}$ 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 3.3.10. *There is a size $n^{10} \log n$ regular resolution proof that the SAT instance $\phi^{Array}_{x(x+1)}(n)$ is unsatisfiable.*

3.3.4 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:

Theorem 3.3.11. *For any degree two ring identity $L = R$, there are polynomial size regular refutations for $\phi^{\text{Array}}_{L=R}(n)$.*

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

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

3.4 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 2.3.

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 3.3 we can verify each of these critical strips with a polynomial-size regular resolution proof.

Definition 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 3.4.1. *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 bitvector $\mathbf{x}$ 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 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 towards the middle, and likewise for the intact ripple-carry adders. In each input bitvector we remember the assignment to just the most recently queried 2Δ variables. □

3.5 Wallace Tree Multipliers

3.5.1 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 bitvectors $\mathbf{x}, \mathbf{y}$ with only logarithmic depth. The 4-bit CLA computes, for each pair x_i, y_i, the values

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

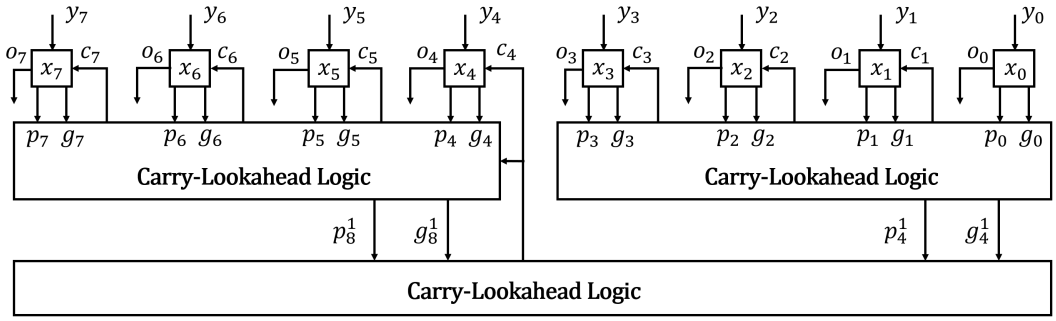

Figure 3.9: 8-bit, two-layer CLA adding $\mathbf{x}, \mathbf{y}$. We have shortened $(x+y)_i$ to o_i in the diagram.

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: $(x + y)_i = c_i \oplus x_i \oplus y_i$. 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 2 layers, whose first half of is shown in Figure 3.9. 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

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

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 2-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

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

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

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

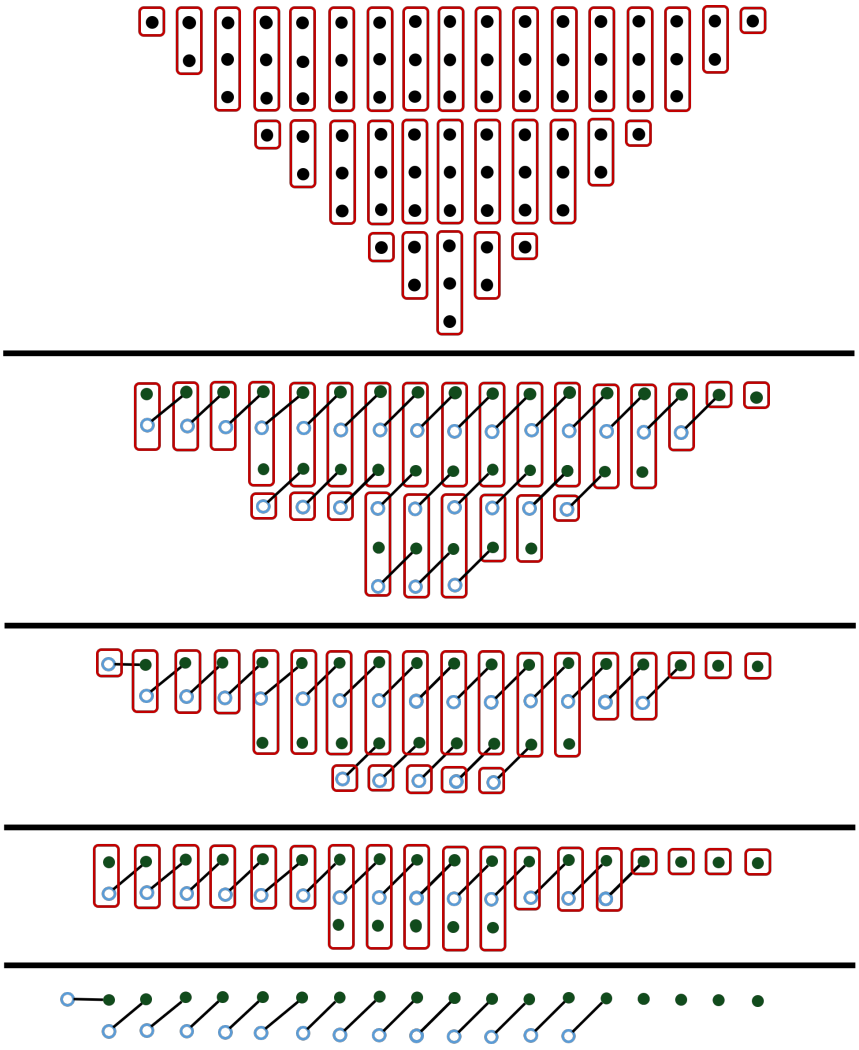

Figure 3.10: Dot diagram for a 9×9 Wallace tree multiplier. Hollow dots represent carry-bits and solid dots represent sum-bits. Dots connected by an edge are output by the same adder.

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] carry-lookahead adder (CLA) to sum the two rows in logarithmic depth, outputting the final sum to the output bit-vector $\mathbf{xy}$.

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 quasi-polynomial size regular resolution proofs.

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

Definition For layer ℓ of a Wallace tree multiplier, if for each $j \leq k$, the outputs of 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*.

---

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

Lemma 3.5.1. *In a Wallace tree multiplier, each layer $\ell \in [0, h-2]$ is row-friendly.*

In terms of the dot diagram in Figure 3.10, this Lemma simply states that no two bits are connected with a line of slope greater than one.

3.5.2 Proofs of Wallace Tree Multiplier Commutativity

Definition 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 3.5, we define Wallace tree multipliers $\mathbf{xy}$ and $\mathbf{yx}$. We then add a circuit E, of of *inequality-constraints* encoding that the two circuits disagree on some output bit.

Definition 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 3.5.2. *$\phi_{Strip}(k)$ is unsatisfiable for all k.*

Proof. We reason similarly to the proof of Lemma 3.3.1. Again, we interpret the critical strip as a circuit that computes the weighted sum, in both $\mathbf{xy}$ and $\mathbf{yx}$, of the tableau variables within the strip. The assignment to $\mathbf{e}$ asserts that the outputs of $\mathbf{xy}$ and $\mathbf{yx}$ differ by precisely 2^k. We bound the admissible difference in outputs by counting the number of input carry-bits in either $\mathbf{xy}$ or $\mathbf{yx}$. 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 $\mathbf{xy}$ and $\mathbf{yx}$ 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 3.5.3. *There is a regular resolution proof of size* $2^{8\log^2 n + O(\log n)}$ *that* $\phi_{Strip}(k)$ *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 3.5.2, 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 downwards in circuit $\mathbf{xy}$, we will reveal the bottom row upwards in circuit $\mathbf{yx}$. 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 $\mathbf{xy}$. The propagation in circuit $\mathbf{yx}$ 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.

The branching program B begins by following Algorithm 1, shown in Figure 3.11, on circuit $\mathbf{xy}$. We use the propagation loop in lines 3-9 for circuit $\mathbf{yx}$, leaving the branching steps to circuit $\mathbf{xy}$. We claim that at the end, B will reach an assignment to just the last layer of circuits $\mathbf{xy}$ and $\mathbf{yx}$. This will follow immediately from Lemma 3.5.4.

ALGORITHM 1:

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

Figure 3.11: Algorithm 1 propagates from the initial layer $\ell = 0$ to the final layer $\ell = h$ of the critical strip **xy** while assigning at most a constant number of bits per subcolumn.

Lemma 3.5.4. *During the execution of Algorithm 1, the tableau variables within each layer of circuit* **xy** *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* **yx** *get assigned in row order from bottom to top, and each tableau variable eventually receives an assignment.*

Proof. We prove both properties for the circuit **xy** by induction, making use of the row-friendliness of Wallace tree multipliers from Lemma 3.5.1. 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 **yx** 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 3.5.5. *At the end of Algorithm 1, the branching program B reaches an assignment to precisely both rows in the last layer of circuits* **xy** *and* **yx**.

To propagate an assignment to last layer of **xy** or **yx** 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 **xy** and **yx**, we have an assignment to the outputs of both critical strips. By Lemma 3.5.2, 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 **xy**, and likewise for each layer $\ell > 1$ for **yx**. By Lemma 3.5.4, the tableau variables within each layer are assigned in row order from top to bottom in **xy**. 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 proof works to show that at most four rows of tableau variables are assigned within each layer $\ell > 1$ of **yx**.

ALGORITHM 2:

```
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
```

Figure 3.12: 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.

Each node in the first phase of B then holds an assignment to at most $8\delta h$ variables of the critical strip. Both **xy** and **yx** 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 $2^{8\log^2 n + O(\log n)}$.

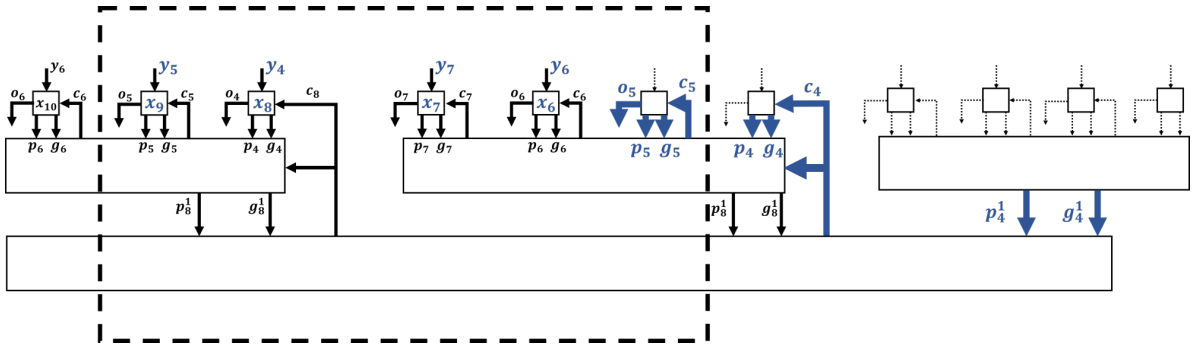

Figure 3.13: An intermediate state in the CLA after scanning up to the sixth column. The dashed box contains the columns of the critical strip. The blue variables are assigned while the blank variables were previously assigned, but then erased.

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 3.5.6. *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 $\sigma_e(k) = \{e_0 = 0, e_1 = 0, \ldots e_k = 1\}$ 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 3.5.3 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. □

3.5.3 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 $\mathbf{L}$'s critical strip to the outputs. We will modify the branching step in these algorithms to verify the distributivity of Wallace tree multipliers.

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

Theorem 3.5.7. *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 $\mathbf{x}(\mathbf{y} + \mathbf{z})$ and one unconstrained carry bit from the adder $\mathbf{y} + \mathbf{z}$ for $n + 3$ total in $\mathbf{L}$'s critical strip. Together, the two n-bit multipliers $\mathbf{xy}, \mathbf{xz}$ have $2n + 2$ unconstrained carry bits. The adder $\mathbf{xy} + \mathbf{xz}$ contributes one more for a total of $2n+3$ unconstrained carry bits in $\mathbf{R}$'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 bitvector $\mathbf{x}, \mathbf{y}, \mathbf{z}$. 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 $\mathbf{xy}, \mathbf{xz}, \mathbf{x}(\mathbf{y}+\mathbf{z})$. 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 $\mathbf{xy}+\mathbf{xz}$, 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. □

3.5.4 Degree Two Identity Proofs for Wallace Tree Multipliers

Using the same ordering on the input variables and ideas from the proof of Theorem 3.3.11, we can prove the analogous result for Wallace tree multipliers.

Theorem 3.5.8. *For any degree two ring identity $L = R$, there are quasipolynomial size regular refutations for $\phi^{\text{Wall}}_{L=R}(n)$.*

3.6 Proving Equivalence Between Multipliers

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

If 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 quasi-polynomial size proofs otherwise.

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

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

3.7 Discussion

Despite significant advances in SAT solvers, one of their key persisting weaknesses has been in verifying arithmetic circuits containing multipliers. This pointed towards the conjecture that that the corresponding resolution proofs are exponentially large; if true, this would have been a fundamental obstacle putting nonlinear arithmetic out of reach for any CDCL SAT solver.

Thus, much of the recent research on multiplier verification has focused on using algebraic reasoning, in particular Gröbner basis methods. The recent work of Kaufmann, Biere and Kauers [128] has improved the Gröbner 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 [128], 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 towards 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. If there are small proofs of associativity, it may be possible to combine these with our results for degree two identities to obtain small proofs of any general ring identity. On the other hand, associativity may require exponentially large resolution proofs, in which case it is an example of bit-vector reasoning that is out of reach of current SAT solving methods. We discuss the proof complexity of associativity further in Chapter 7.

Chapter 4

SAT EXPERIMENTS

4.1 Introduction

The polynomial-size resolution proofs that we constructed in the previous chapter tell us that, in principle, CDCL SAT-solvers can verify properties like multiplier commutativity in polynomial time by walking through our proof. However, polynomial time algorithms are not necessarily practical. Constants matter. The real-life applicability of our proofs hinges on the absolute (rather than asymptotic) proof size associated with the specific bit-widths of $n = 32$ or $n = 64$ (these bit-widths capture the behavior of the standard integer representations used in most programming languages). In this chapter we present several sets of experiments that aimed to determine the feasibility of scaling SAT-solvers to solve the problems containing 32-bit or 64-bit multiplication.

We divide this chapter into three main sets of experiments. In the first set of experiments, we present the baseline performance of two unmodified SAT solvers for verifying algebraic properties of multiplier and adder circuits. These experiments demonstrate that while verifying *linear* properties, such as the commutativity of an adder circuit, is easy for SAT solvers for bit-widths of over 1024, verifying *nonlinear* properties, such as the commutativity of a multiplier circuit, already become intractable at 11 bits.

In the second set of experiments, we run modified solvers on critical strips for commutativity, aiming to get a CDCL SAT-solver to find resolution proofs as small, or ideally even smaller, than our proofs. We focus our discussion on the quantitative and qualitative effects of modifying the decision variable order and the clause learning scheme. Unfortunately, we

were unable to coax solvers into convincingly finding our proof due to the combined diagonal and grid-like structure of a critical strip.

In the third set of experiments, we ran modified solvers on an idealized, "rectangular" version of a critical strip, aiming to estimate the performance of a SAT solver that follows our critical strip proof. We found that with the purely grid-like structure of these rectangular strips, using a fixed variable order was sufficient to get the CDCL algorithm very close to our proof.

Finally, in the context of our experimental data, we will discuss the feasibility of using CDCL SAT solvers to verify properties of 32 or 64 bit multipliers.

4.2 Experimental Setup

In our experiments, we used an Intel Core i7-6700K CPU at 4.00GHz with a memory limit of 8GB. The wall-clock time limit was set to 1200 seconds. We list experiment times in seconds (wall-clock time) and write TO if the time limit of 1200 seconds was exceeded. Our benchmarks and generators were written from scratch, and are (for the moment) available at https://github.com/vliew/nonlinear.

For the first set of experiments with unmodified solvers, we used the solvers CaDiCaL, one of the top solvers in the 2019 SAT competition, and MiniSAT, an influential and particularly well-understood CDCL solver from 2005. For the second and third sets of experiments, we focused our attention on modifying MiniSAT.

4.3 Unmodified Solver Experiments

This section presents experiments comparing the performance of unmodified SAT solvers for verifying algebraic properties of adder and multiplier circuits. We start by demonstrating that properties of adders are easy to verify with SAT-solvers whereas properties of multipliers are hard to verify. Then we examine whether we gain any immediate benefit from breaking the multipliers up into critical strips and solving each strip independently compared to handling the multipliers all at once.

	CaDiCaL		MiniSAT	
n	Comm	Assoc	Comm	Assoc
32	0.01	0.07	0	2.4
64	0.02	0.15	0.01	6.9
128	0.03	0.33	0.04	44
256	0.09	0.77	0.19	188
512	0.17	3.3	0.63	TO
1024	0.17	11.2	2.9	

Table 4.1: Time (seconds) for off-the-shelf versions of SAT solvers CaDiCaL and MiniSAT to verify the commutativity or associativity of n-bit ripple-carry adders.

We verify commutativity, distributivity and associativity for both adders and multipliers. We used either array or Wallace tree multipliers to model multiplication, and we modeled addition using ripple-carry adders.

Table 4.1 shows how SAT-solvers scale when checking commutativity and associativity of ripple-carry adders. We observe that while the older 2005 solver MiniSAT was able to verify commutativity reasonably quickly, it started to struggle to verify associativity at 512 bits (which is far more than enough bit-width: for most applications, 64 bits suffice). The modern 2019 solver CaDiCaL makes short work of verifying both commutativity and associativity up to even 1024 bits. The last decade of refining CDCL solvers has noticeably improved their ability to reason about linear bit-vector arithmetic.

On the other hand, we can see from Table 4.2 that problems involving *nonlinear* bit-vector arithmetic are an entirely different beast. Any attempt to verify a nontrivial property of bit-vector multiplication seems to inevitably run into a wall of exponentially scaling solve times. Simply checking the commutativity of a 16-bit multiplier is already hopelessly out of reach for current SAT solvers. The performance improvement going from MiniSAT to CaDiCaL was also much narrower here, compared to the improvement we saw when verifying linear properties. Both solvers timed out at 11 to 12 bits for all seven problems.

We see the same scaling in Figure 3.1, which shows commutativity solve times for the top 2016

CaDiCaL							
n	Ar-Comm	Ar-Dist	Ar-Assoc	Wa-Comm	Wa-Dist	Wa-Assoc	Ar-Wa
5	0.02	1.3	2.4	0.02	1.4	2.6	0.02
6	0.13	15	31	0.2	18	38	0.15
7	0.72	417	TO	1	407	TO	1
8	4.0	TO		8	TO		5
9	16			38			26
10	116			204			172
11	969			TO			TO
MiniSAT							
n	Ar-Comm	Ar-Dist	Ar-Assoc	Wa-Comm	Wa-Dist	Wa-Assoc	Ar-Wa
5	0.01	1.1	2.4	0.02	1.7	2.8	0.01
6	0.15	33	39	0.1	75	71	0.12
7	0.52	668	TO	0.8	587	TO	0.7
8	11	TO		6	TO		5.6
9	43			95			62
10	743			257			466
11	TO			TO			TO

Table 4.2: Time for CaDiCaL and MiniSAT to verify commutativity (Comm), distributivity (Dist) and associativity (Assoc) of n-bit array (Ar) or Wallace (Wa) tree multipliers. The last problem (Ar-Wa) checks that array and Wallace tree multipliers are equivalent.

solvers Lingeling and Glucose. The times listed for these solvers are actually roughly a factor of 10 faster than the times we obtained for CaDiCaL and MiniSAT, which reflects a difference in the computing hardware used. Unfortunately, our ability to solve these multiplier problems does not seem to have materially benefited from the last 15 years of improvements to SAT solving.

The apparently exponential scaling of solve times shown in Table 4.2 indicate that out-of-the-box SAT solvers were failing to find our polynomial-size proofs. The most immediate reason was that our proofs, as described in Theorem 5.3.2, begin with a fairly specific branch order on the e variables in order to split a multiplier into critical strips. As a SAT-solver starts with no knowledge about the meaning of any variable, it is unlikely to immediately stumble upon this branch order by chance.

An early hope was that simply breaking up these verification problems would allow solvers to efficiently verify these properties by finding our proof. In Table 4.3 we show the effect of breaking up each of the multiplier problems into critical strips. Using this critical strip decomposition did end up producing faster results, allowing us to verify commutativity up to 13 bits instead of 11 bits. However, the solve times still appeared to scale exponentially in bit-width. Even starting from the critical strips, SAT solvers were not finding our polynomial size proofs.

4.4 Critical Strips and Ordering

In this section we describe our second set of experiments, which attempted modify the solver MiniSAT to guide it towards our strip-based proofs for array multiplier commutativity.

The most direct way to influence a SAT-solver towards a known *regular* (i.e. fixed order) resolution proof is to fix a specific branch ordering. Our polynomial size critical strip refutations branched on variables in a fixed order that revealed the entries of the critical strip row-by-row. We can force a CDCL solver to follow the same branch ordering, with the hopes that its conflict analysis subroutine learns the same clauses that appear in the proof.

CaDiCaL						
	Ar-Comm		Wal-Comm		Ar-Dist	
Bit-width	Full	Strips	Full	Strips	Full	Strips
6	0.13	0.15	0.2	0.12	15	8
7	0.7	0.7	1	0.6	417	56
8	4	3	8	2	TO	394
9	16	11	38	5		TO
10	116	45	204	19		
11	969	165	TO	187		
12	TO	605		536		
13		TO		1009		
MiniSAT						
	Ar-Comm		Wal-Comm		Ar-Dist	
Bit-width	Full	Strips	Full	Strips	Full	Strips
6	0.2	0.1	0.1	0.1	33	21
7	0.5	0.7	0.8	0.4	668	393
8	11	3	6	3	TO	TO
9	43	26	95	14		
10	743	146	257	133		
11	TO	1055	TO	788		

Table 4.3: Time for CaDiCaL and MiniSAT to verify multiplier properties by decomposing the multiplier into strips, compared to the time listed in Table 4.2 to verify the property all at once. We check the commutativity and distributivity of array multipliers (Ar-Comm and Ar-Dist respectively) and commutativity of Wallace tree multipliers (Wa-Comm).

Bit-width	Unmodified	x, y-order	t-order
5	0.03	0.03	0.07
6	0.1	0.3	0.3
7	0.7	2	2
8	3	14	13
9	26	165	155
10	146	TO	TO
11	1055		

Table 4.4: Time for a fixed order version of MiniSAT to refute critical strips.

For each critical strip, we either forced MiniSAT to branch on the tableau variables in the order specified by Lemma 3.3.1, or to branch on the input x, y variables in the order given by Corollary 3.3.3. Table 4.4 shows the effect of both orderings. Fixing the ordering seemed to actually harm overall performance compared to MiniSAT's standard variable selection heuristics. Consequently, the scaling appeared to remain exponential, certainly outpacing the $O(n^6 \log n)$ resolution proof size. This scaling indicated that even with the "correct" fixed branch order, MiniSAT's conflict analysis did not learn the "correct" clauses. Hence, our next step was to additionally try changes to the clause learning algorithm.

To recall Chapter 2, the standard CDCL conflict analysis algorithm, used by most SAT solvers including MiniSAT and CaDiCaL, learns a clause corresponding to the '*first unique implication point* (1UIP) associated with a conflict. As the name suggests, a conflict may have a second unique implication point (2UIP) or a third one (3UIP), and so on. We investigated the effect of learning these alternative UIPs. Some key benefits of using a UIP-based learning scheme are that UIPs are easy to find, and they typically correspond to short learned clauses.

We focused our modified UIP experiments on the critical strips of a 9-bit array multiplier. Table 4.5 compares performance on these strips between the 1UIP, 2UIP, 3UIP and lastUIP learning schemes. We found that overall, there was little difference. The 1UIP scheme slightly outperformed the 2UIP and 3UIP methods. Learning the lastUIP caused a large

Strip	Unmodified	fixed x, y order				fixed t order			
	1UIP	1UIP	2UIP	3UIP	lastUIP	1UIP	2UIP	3UIP	lastUIP
4 to 7	0.2	0.8	0.5	0.6	0.5	0.5	0.5	0.5	0.6
5 to 8	4	3	4	4	4	3	4	4	4
6 to 9	9	11	12	12	18	12	12	13	17
7 to 10	9	33	35	36	67	31	33	39	59
8 to 11	3	28	31	33	126	27	30	35	122
8 to 12	0.5	73	88	93	451	69	78	83	412
9 to 13	0.1	15	15	18	126	12	13	12	137
11 to 14	0.1	0.8	0.9	1	2	0.3	0.3	0.4	1

Table 4.5: Effect of learning different UIPs in fixed-order MiniSAT on critical strips for $n = 9$.

drop in performance.

The small difference between the performance of the first, second and third UIP learning schemes was largely explained when we examined how often conflicts actually were associated with more than the first UIP. We observed that the vast majority, 99.5%, of conflicts in our critical strip experiments only had a first UIP. Hence, the 2UIP and 3UIP learning schemes only changed the learned clause for about 0.5% of conflicts. On the other hand, this small fraction seemed to be enough for the lastUIP scheme to significantly slow down the solver.

These results indicated that even with the a fixed variable order, UIP-based learning schemes are not able to find the "right" clauses for our proof. The next section shows and discusses, at a qualitative level, which clauses are actually learned.

4.5 Qualitative Performance

We implemented a visualization tool in order to qualitatively assess how close unmodified and fixed-order SAT-solvers were getting to our polynomial size critical strip proofs. Recall from Figure 3.6 that the clauses appearing in our critical strip refutation for commutativity consist of variables along two cuts in the circuit: one cut along the output bits and one across a row of the strip. Hence, if a CDCL SAT-solver were following our proofs, then it

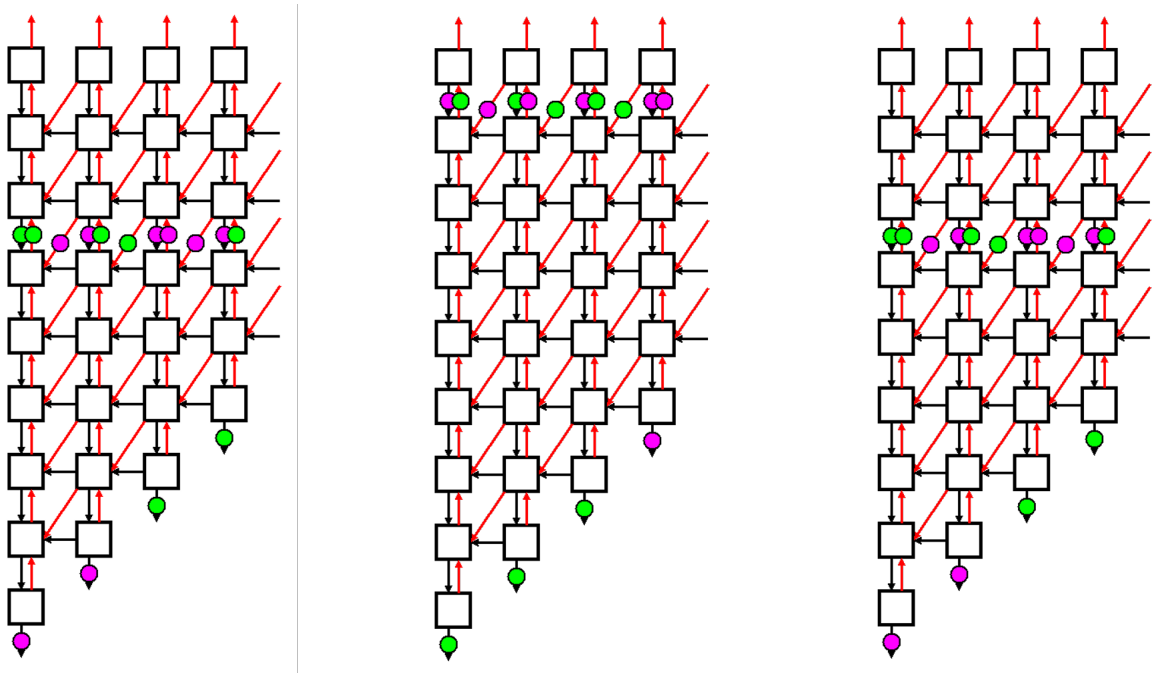

Figure 4.1: Examples of three "good" learned clauses for the critical strip spanning columns 5 to 8. For clarity we have overlaid the two circuits xy (black) and yx (red). Dots label the variables appearing in the learned clause. Green dots correspond to positive literals and purple dots correspond to negative literals.

should learn clauses similar to the examples depicted in Figure 4.1.

Our visualization tool revealed, in agreement with our interpretation of the data in Table 4.3, that the refutations produced by unmodified SAT-solvers were very different in appearance from the target proofs. As we can see from Figure 4.2 depicts some examples of clauses learned by the unmodified version of MiniSAT. We can see that the learned clauses were highly varied, often involving many disconnected regions of the circuit.

The refutations that MiniSAT produced with the t-variable ordering were closer to looking like "good cuts". Some examples of these learned clauses are shown in Figure 4.3. By branching on the bottom output bits first, we ensured that the bottom cut appears correctly in the learned clauses. However, the top cuts generally looked very different, and more

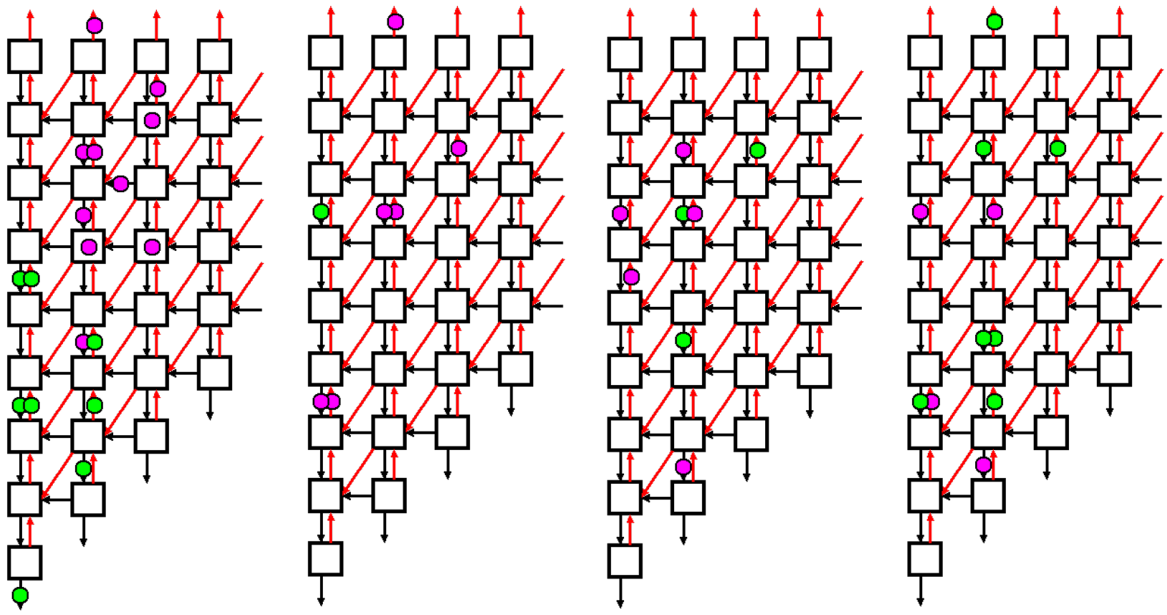

Figure 4.2: Examples of four clauses learned by MiniSAT using default settings.

chaotic, than the top cuts in our target "good cuts".

We observed that the key feature of these critical strips leading to these chaotic clauses is the presence of both rectangular and diagonal grid structures. The "cuts" in our proofs should mainly follow one of the two structures. However, propagation occurs along both, leading a SAT solver to learn clauses mixing both structures. Learning these mixed clauses seemed to cascade into learning more and more chaotic-looking clauses later in the solve.

To conclude this section, we were unable to reproduce our polynomial-size critical strip proofs by modifying the branch order and the conflict analysis of MiniSAT. Other modifications that we tried, but do not discuss, included changes to the restart frequency, phase saving policy, clause deletion, and propagation order. Each of these changes had either a negligible, or highly negative impact on performance. It remains an open challenge to get a CDCL SAT solver to closely reproduce our critical strip proofs.

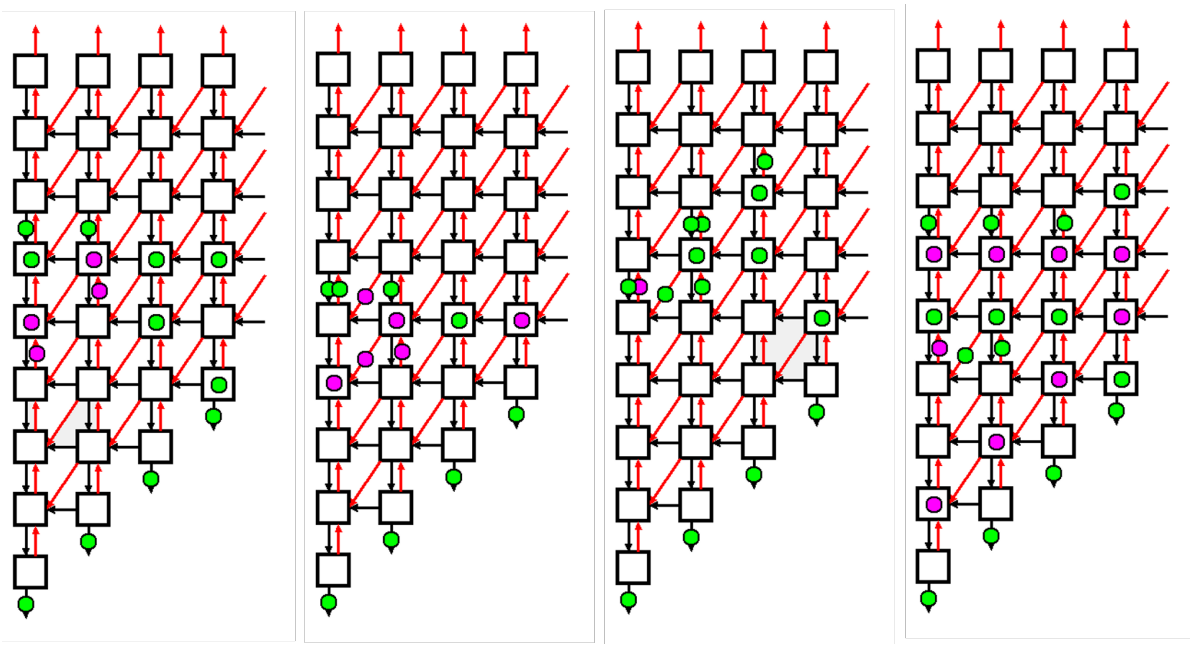

Figure 4.3: Examples of four clauses learned when MiniSAT uses the t-ordering.

4.6 Rectangles: An Idealized Model

As we discussed in the previous section, the mixture of rectangular and diagonal grid structures in multiplier critical strips seemed to throw our fixed-order version of MiniSAT off track. In order to get an empirical estimate of the real-life scaling of our proofs, we defined a purely rectangular analogue of the critical strip problem that we call a $w \times h$ *rectangle*. Without the interfering diagonal structure, a fixed-order SAT solver should be able to cleanly learn clauses corresponding to the horizontal cuts of our proof.

We will begin this section by defining $w \times h$ *rectangles*. We then describe the analogue of our critical strip proof for a critical rectangle. Then we show how to use the size of this rectangle proof as an overestimate for the size of our critical strip refutations from Theorem 5.3.2. Through this size relationship, we can use empirical data for a SAT solver that finds our critical rectangle proof to estimate the real-world performance of a SAT solver that faithfully follows our critical strip proof.

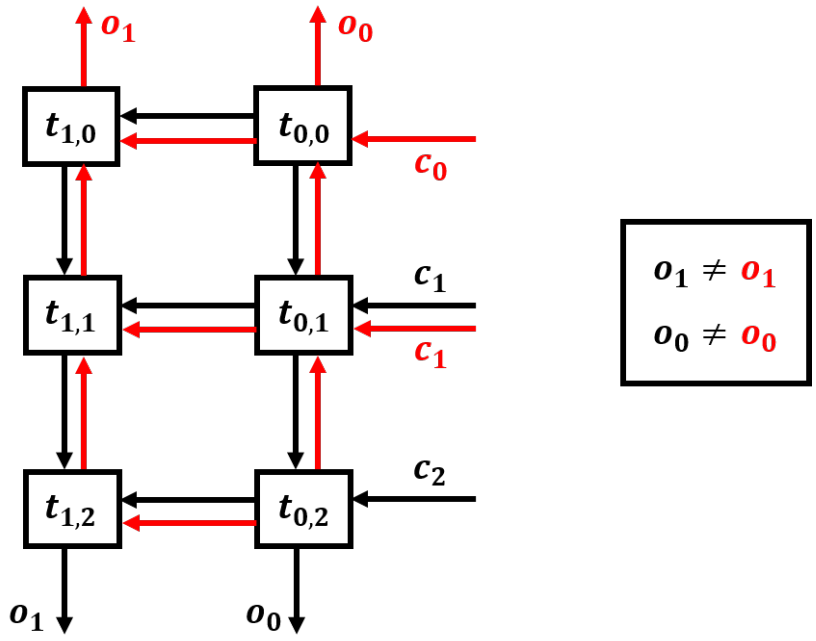

Figure 4.4: Labeled example of a 2×3 rectangle. Circuit variables for L are black and circuit variables for R are red. The box on the right contains the constraints on the output bits: $o_1^L \neq o_1^R$ and $o_0^L = o_0^R$.

4.6.1 Rectangles and Critical Strips

Definition A $w \times h$ *rectangle* is a circuit consisting of two $w \times h$ rectangular grids, **L** and **R**, of full adders, along with constraints asserting that for each $i < w - 1$, we have $o_i^L = o_i^L$ and for the most significant bit $i = w - 1$, we have $o_i^L \neq o_i^L$. Grid L computes, in order of increasing j, the weighted sum of the tableau variables $t_{i,j}$, where $(i, j) \in [w] \times [h]$ and $\text{weight}(t_{i,j}) = 2^{i+j}$, and the carry-in bits c_j^L, where $h > j \geq 1$ and $\text{weight}(c_j^L) = 1$. Grid R computes, in order of decreasing j, the weighted sum of the tableau variables $t_{i,j}$ and the carry-in bits c_j^R.

Definition A $w \times h$ *critical strip* is a critical strip that is at most w adders wide and at most h adders tall.

Figure 4.4 shows a labeled example of a 2×3 rectangle. The circuit **L** sums each row of

t-variables from top to bottom while circuit $\mathbf{R}$ sums them from bottom to top. This 2×3 rectangle turns out to be satisfiable if we assign $c_1^L = 1, c_2^L = 1$ and $c_0^R = 0, c_1^R = 0$ so that the difference in carry-in between the two grids is precisely 2. In the following proposition we will adapt the proof of Lemma 3.3.1 to determine which $w \times h$ rectangles are unsatisfiable.

Proposition 4.6.1. *A $w \times h$ rectangle is unsatisfiable if and only if $h \leq 2^{w-1}$.*

Proof. For a $w \times h$ rectangle, the total difference Δ in carry-in weight between the circuits L and R can take any (integer) value in the open interval $(-h, h)$. The output constraints assert that the absolute value of this difference, $|\Delta|$, is precisely 2^{w-1}. □

For an unsatisfiable $w \times h$ rectangle, the analogue of our critical strip resolution proof uses the same row-by-row branch order. It begins by branching on the 2^w possible assignments to the w output values o_i^L. We will refute each of these 2^w branches separately. Propagate the assignment to o_i^L to the outputs o_i^R. For rows $j = 0$ to $j = w - 1$, branch on the j-th row of tableau variables $t_{i,0}$ as well as the carry-in variable c_0^R. Propagate the effects of these decisions through the circuit, then "learn" the assignment to variables contained in the cut between the j-th and $j+1$-th rows (i.e. forget the assignment to variables outside of the cut). After iterating these branch-propagate-save steps for each row and reaching the bottom, we will reach an assignment contradicting one of the original circuit constraints.

Definition Let $C_{\text{rectangle}}(w, h)$ denote the number of horizontal cuts appearing in the resolution refutation for $w \times h$ rectangles described above, and let $C_{\text{strip}}(w, h)$ denote the number of horizontal cuts appearing in the critical strip refutation described in Theorem 5.3.2 for a critical strip that is w adders wide and h adders tall.

Proposition 4.6.2. *For unsatisfiable $(w \times h)$ critical strips and $(\frac{3}{2}w \times h)$ rectangles:*

$$C_{strip}(w, h) \leq C_{rectangle}\left(\frac{3}{2}w, h\right)$$

Sketch. We first calculate an upper-bound for the quantity $C_{\text{strip}}(j, w, h)$, defined as the

number of $(w \times h)$ critical strip cuts (i.e. learned clauses) that our proof generates between consecutive rows j and $j+1$. Each critical strip cut corresponds to an assignment to its $\leq 4w$ variables. Moreover, the remaining assignment to a critical strip cut is fully determined after freely choosing an assignment of the $\leq 3w$ cut variables (interpreted as bit-vectors): $\mathbf{o}, \mathbf{c}^{yx}, \mathbf{d}^{yx} \in [0, 2^w - 1]$, and choosing a $\Delta \in (-j+1, j-1)$ for the carry-in difference above row j. Therefore, $C_{\text{strip}}(j, w, h) \leq (2j-3)2^{3w}$.

We now calculate a lower-bound for the quantity $C_{\text{rectangle}}(j, w, h)$, the number of $(w \times h)$ rectangle cuts that our proof generates between consecutive rows j and $j+1$. We can generate the horizontal cuts that appear in our proof by assigning the $2w$ cut variables $\mathbf{o}, \mathbf{d^L} \in [0, 2^w - 1]$, and choosing a $\Delta \in (-j+1, j-1)$ for the difference in carry-in above row j. Therefore, $C_{\text{rectangle}}(j) \geq (2j-3)2^{2w}$.

From these bounds, we see that $C_{\text{rectangle}}(j, \frac{3}{2}w, h) \geq C_{\text{strip}}(j, w, h)$. This implies the proposition since $C_{\text{rectangle}}(w, h) = \sum_j C_{\text{rectangle}}(j, w, h)$ and $C_{\text{strip}}(w, h) = \sum_j C_{\text{strip}}(j, w, h)$. □

The quantities $C_{\text{rectangle}}(w, h)$ and $C_{\text{strip}}(w, h)$ are closely proportional to the resolution proof size. So the above proposition says, roughly, that our resolution proof size for a $\frac{3}{2}w \times h$ rectangle is an overestimate for the proof size of a $w \times h$ strip.

Recall that our target was to refute critical strips for 32 or 64 bit multipliers. The largest critical strip for a 32-bit multiplier has a width of $w = 6$ and a height of $h = 32$, which, from Proposition 4.6.2, roughly corresponds to a 9×32 rectangle. So if SAT-solvers are able to efficiently solve a 9×32 rectangle by following our proof, that would be excellent evidence that with well-tuned heuristics, we would also be able to efficiently solve the critical strips of a 32-bit multiplier. On the other hand, if we observe the solver following our proof but it is nowhere near solving a 9×32 rectangle, then that is evidence that our resolution proofs are not small enough for practice.

For the purpose of evaluating our experiments in the next section, we will now estimate how our proof size scales in rectangles of fixed width w and varying height h.

Height h	Time (seconds)	# Learned clauses (thousands)
3	48	160
4	141	317
5	247	450
6	457	631
7	1056	967
8	1900	1284
9	2740	1625
10	5211	2209
11	8372	2869
12	15738	3957
13	17302	4175

Table 4.6: Time and number of learned clauses for MiniSAT to refute a $5 \times h$ rectangle.

Proposition 4.6.3. *For fixed w and varying $h \leq 2^{w-1}$, we have that $C_{rectangle}(w, h)$ is proportional to h^2.*

Proof. The proof size should scale proportionally to h^2 because the number of cuts we need to learn between rows $j-1$ and j is proportional to $2j-1$, the number of possible values for $\Delta \in [-j+1, j-1]$, the difference in carry-in above row j. So the total number of cuts we need to learn to refute a rectangle of fixed width w and h rows using is proportional to the sum $3 + 5 + \ldots + (2h-1) = O(h^2)$. □

4.6.2 Rectangle Experiments

Figures 4.5 and 4.6 show the performance, in terms of both time and total number of learned clauses, for a SAT solver using a fixed row-by-row branch order to solve rectangles of width 5. We increased the timeout limit to $20,000$ seconds (from to the 1200 second timeout of the previous experiments) to better estimate how performance scaled asymptotically.

We observed that both metrics, time and number of learned clauses, seemed to scale as relatively low degree polynomials. This was not the case for our experiments on critical strips. Notably, the best power-law fit ($y = ax^b$) for the number of learned clauses scaled

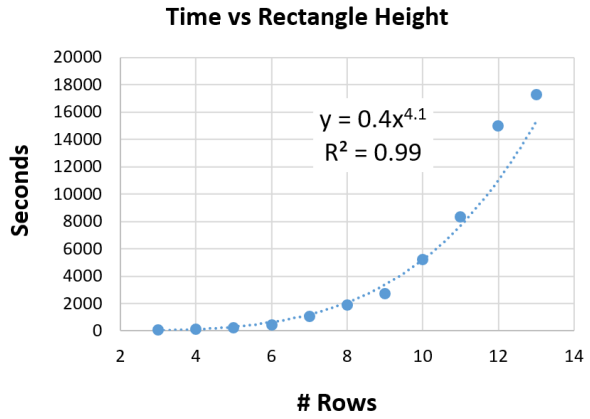

Figure 4.5: Time to refute a $5 \times h$ rectangle.

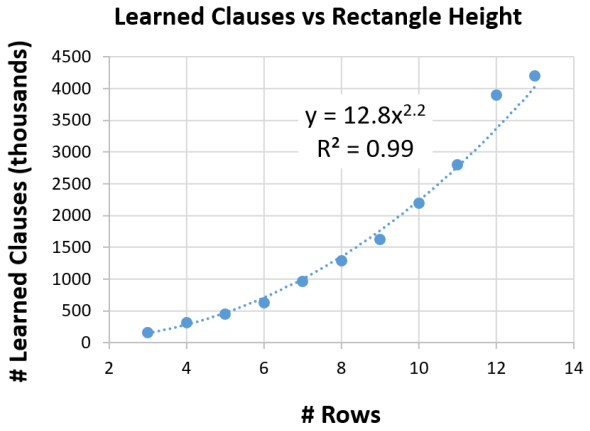

Figure 4.6: Learned clauses to refute a $5 \times h$ rectangle.

as $O(h^{2.2})$ and had an R^2 of 0.99— only slightly more than the $O(h^2)$ scaling of our target proof. This scaling suggested that the fixed-order SAT solver was actually getting fairly close to our rectangle proofs.

Examining the fixed order SAT solver's learned clauses more closely using our visualization tool, the solver was indeed learning clauses corresponding to exactly the cuts that should appear. Some examples of these learned cuts are shown in Figure 4.7. While there were some occasional "imperfect" cuts, that we show in Figure 4.8, where extra variables appeared alongside a cut, the solver was largely faithful to our target proof.

Although we achieved a polynomial runtime, scaling roughly as $O(h^4)$, this polynomial turned out to be far too large for practice. As we discussed in the previous section, the largest critical strip for a 32-bit multiplier is approximately equivalent, in terms of proof size, to a 9×32 rectangle. Even when with a fixed order SAT-solver that followed our rectangle proof, we were unable to solve 6×10 rectangles within the extended $20,000$ second timeout.

In the end, our experiments with these rectangle instances suggested that even if SAT-solvers were to carry out our polynomial-size proof, the proof sizes required are still simply too large

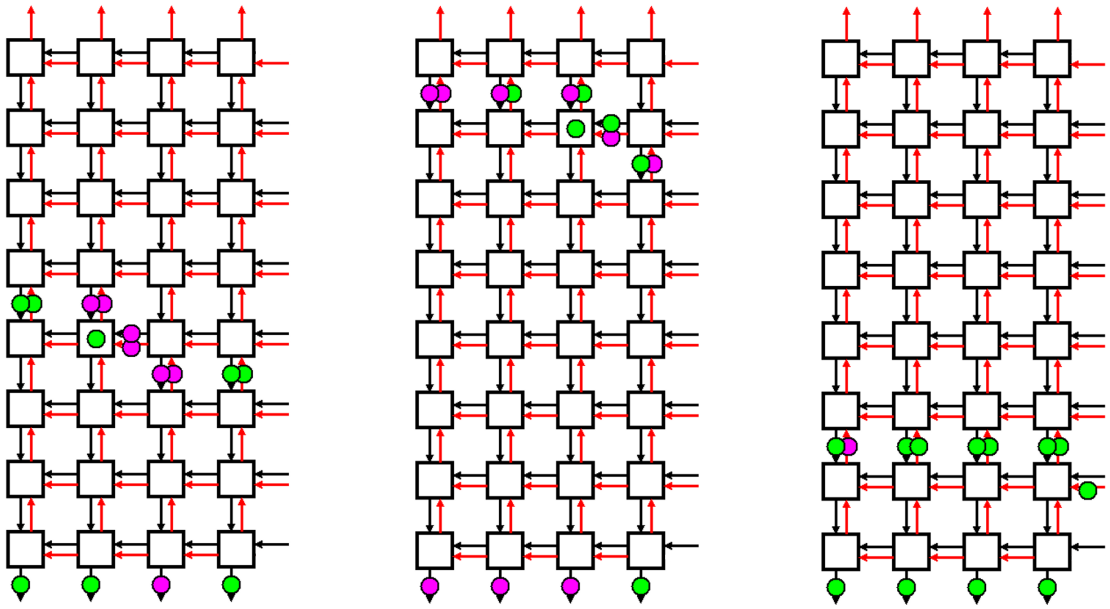

Figure 4.7: Examples of typical "good" cuts learned by fixed order MiniSAT running on a 4×8 rectangle.

to efficiently handle sufficiently large enough multipliers for practice. Unless even smaller resolution proofs than ours exist, resolution-based reasoning is just too weak.

In order to handle bit-precise properties of multipliers in practice, we need to upgrade to a stronger, more expressive proof system that is a more natural fit for reasoning with arithmetic properties. In the next chapter, we turn our focus away from resolution and towards the *cutting planes* proof system. We will show that the cutting planes proof system is capable of dramatically smaller proofs than resolution is. We will construct cutting planes proofs that scale *optimally* in the bit-width n for a large class of degree two ring properties.

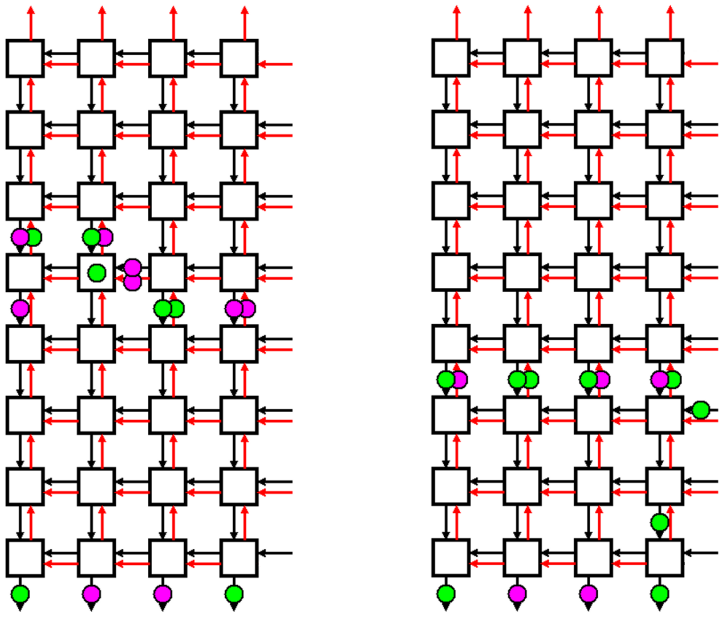

Figure 4.8: Examples of some "imperfect" cuts learned by fixed order MiniSAT running on a 4×8 rectangle.

Chapter 5

VERIFYING PROPERTIES OF BIT-VECTOR MULTIPLICATION USING CUTTING PLANES REASONING

5.1 Introduction

Although we were able to construct polynomial size resolution proofs in Chapter 3, the results of our SAT solver experiments in Chapter 4 indicate that the degree of these polynomial size bounds is likely too large for practice. In order to unlock the ability to solve formulas that mix bit-level reasoning with multiplication, we need to fundamentally improve the back-end reasoning.

Two natural approaches for strengthening resolution-based reasoning are embodied by the proof systems *polynomial calculus* [38], which generalizes from clauses to polynomials, and *cutting planes* [41], which generalizes from clauses to linear 0-1 inequalities. Both of these proof systems can directly simulate resolution, and in some cases are exponentially better.

As we discussed in the Introduction and Chapter 2, computer algebra has recently emerged as a powerful tool for verifying isolated gate-level multiplier circuits. Since our work in Chapter 3, there has been a continued effort towards tailoring these methods towards verifying larger and more complicated multiplier circuits [36, 81, 97, 98, 148].

A major advantage of *Gröbner basis* methods, which perform algebraic reasoning that is captured by the polynomial calculus proof system, is that they operate with polynomials instead of disjunctive clauses. This makes it possible to encode the *correctness* of a multiplier with input bit-vectors $\mathbf{x}, \mathbf{y}$ and output bit-vector $(\mathbf{xy})$ through the word-level *specification*

equation:

$$\Big(\sum_{i=0}^{n-1} 2^i x_i\Big)\Big(\sum_{i=0}^{n-1} 2^i y_i\Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i\Big) = 0.$$

Unfortunately, for the non-algebraic parts of circuits, Gröbner basis methods are typically orders of magnitude slower than SAT solvers and scale poorly on general reasoning. We provide an explanation for this by showing, drawing on the work of Impagliazzo, Pudlak and Sgall [75], that Gröbner basis methods require an exponential number of steps to derive bit-level consequences of word-level properties. Hence, these methods are unlikely to supplant the role of SAT solvers for bit-vector arithmetic.

We propose instead that *conflict-driven pseudo-Boolean solvers* [34] that take advantage of the *cutting planes* method for 0-1 linear inequalities [41] have the potential to achieve the "best of both worlds", combining the strengths of Gröbner basis methods for polynomials with the efficiency of CDCL SAT solvers for Boolean reasoning. Cutting planes reasoning can easily express word-level properties and does not suffer the same obstacles as polynomial calculus, since only a linear number of steps are needed to derive *all* of the individual bit-equalities from a word-level equality.

An essential aspect of this approach in improving on SAT-based methods is that one can express the correctness of 1-bit adders, basic building blocks of arithmetic circuits, directly via pairs of inequalities, instead of using sets of clauses, and one can similarly directly express word-level properties of circuit outputs. Together, these yield a higher-level fully precise form of "bit-blasting".

In this chapter, we will construct optimal, $O(n^2)$ length cutting planes proofs for a large class of n-bit ring identities, including commutativity and distributivity.

Our proofs are most naturally expressed using an extended form of cutting planes that we call the (k, d)-*cutting planes* proof system. While the lines of a cutting planes proof are linear inequalities, a (k, d)-cutting planes proof allows for inequalities with up to k nonlinear "terms" of a certain form, each of degree at most d. We show that any (k, d)-cutting planes

proof can be simulated by a standard cutting planes proof that is only a factor of $(k+4)d^k$ larger. When proving a particular ring identity, we make sure to limit the number of bounded degree nonlinear terms on each line so that the corresponding cutting planes proof is only a constant factor larger.

We emphasize that cutting planes can prove these identities not only at the word level, but also at the level of the individual bits. while $O\big(n^2\big)$-length polynomial calculus proofs are known for some of these properties at the word-level, as shown by Kaufmann, Biere and Kauers in [82], polynomial calculus proofs cannot efficiently extract the bit-equalities. As a consequence, for example, the best known polynomial calculus proof for the bit-level property "the middle bit of xy equals the middle bit of yx" is still the $O\big(n^5 \log n\big)$-length resolution proof given by Beame and Liew in [12], which is much larger than our $O\big(n^2\big)$-length cutting planes proof.

These ring identities appeared previously as testbed instances representing the gap between word-level and bit-level methods of reasoning. For example, it was observed in 2016 that proving the commutativity of a multiplier circuit is already intractable for SAT solving at 16 bits [16]. While bit-vector solvers try to overcome this shortcoming of SAT by implementing word-level preprocessing and inprocessing, the verification of larger systems containing multiplication and bit-logic (that appear for instance, in cryptography) remains a key weakness. The ability to verify these ring identities at the bit level, rather than through preprocessing, is a good test for the potential of any method for verifying these more complex systems.

In the next chapter, we show experimentally that we are able to use pseudo-Boolean solvers to verify the word-level equivalence of several different multiplier circuits of up to 256 bits in similar times to those of the best algebraic methods. We find that these solvers can be particularly efficient at extracting all of the bit-level equalities from a word-level equality, which neither CDCL solvers nor Gröbner basis reduction can do efficiently.

In the next chapter we will also show that pseudo-Boolean solvers can be used to efficiently

verify a number of bit-vector inequalities combining multiplication with bit-wise operations. In contrast, these inequalities are much harder or intractable for the top bit-vector solvers Boolector [23, 112], Z3 [47], Yices2 [51] and CVC4 [6]. Our examples demonstrate the potential of pseudo-Boolean solvers for reasoning with nonlinear, bit-precise systems that are out of reach of current methods.

5.2 Notation and Preliminaries

We recall and restate the definitions of the polynomial calculus and cutting planes proof systems from Chapter 2.

Definition Given a set of polynomials Φ over a set of variables $\{x_1, x_2, \ldots, x_n\}$ and a field K, a *polynomial calculus* refutation of Φ is a sequence of polynomials ending with the polynomial 1 such that each line is either in Φ or is derived from the previous lines using the inference rules of linear combination and multiplication by a monomial m:

$$\frac{p \qquad q}{\alpha p + \beta q}\ (\alpha, \beta \in K), \qquad \frac{p}{m \cdot p}\ .$$

The polynomials $x^2 - x$ are also included as axioms for each variable x so that it only takes Boolean values. The polynomial p is interpreted to mean the equation $p = 0$.

Definition Given a set of linear integer inequalities Φ over a set of variables $\{x_1, x_2, \ldots, x_n\}$, a *cutting planes* refutation of Φ is a sequence of linear integer inequalities ending with the inequality $0 \geq 1$ such that each line is either in Φ or is derived from the previous lines using the inference rules of positive linear combination

$$\frac{\sum_i a_i x_i \geq b \qquad \sum_i a'_i x_i \geq b'}{\sum_i (\alpha a_i + \beta b_i) x_i \geq \alpha b + \beta b'}$$

where $\alpha, \beta \geq 0$, and the *division rule*

$$\frac{\sum_i (c \cdot a_i) x_i \geq b}{\sum_i a_i x_i \geq \lceil \frac{b}{c} \rceil} .$$

The *literal axioms* $-x \geq -1$ and $x \geq 0$ are also included for each variable x. We will use "=" as shorthand for the two equivalent "$\leq, \geq$" inequalities.

5.2.1 A polynomial calculus lower bound for bit-extraction

The bit-extraction lower bound discussed in the introduction follows directly from the following polynomial calculus lower bound for *subset-sum equations*.

Theorem 5.2.1 (Impagliazzo, Pudlak and Sgall [75])**.** *Let $c_1, \ldots, c_n$ be nonzero real numbers such that no subset sums to the real number m. Then the equation $m - \sum_{i=1}^{n} c_i x_i = 0$ has no polynomial calculus refutation of degree $\lceil n/2 \rceil$ in the field of real numbers.*

Theorem 5.2.2 (Impagliazzo, Pudlak and Sgall [75])**.** *Suppose that Φ is a set of polynomials of degree at most $\sqrt{n}$, where n is the number of variables appearing in Φ. Let d denote the minimum refutation degree of Φ, and M denote the minimum number of monomials in a refutation of Φ, and assume that $M \geq 3$. Then $M \geq \exp\left((d-1)^2/4n\right)$*

We combine Theorems 5.2.1 and 5.2.2 to demonstrate the weakness of polynomial calculus in extracting bit-level properties from word-level ones.

Corollary 5.2.3. *For a fixed integer k, any polynomial calculus refutation of the system of two polynomials:*

$$f := \sum_{i=0}^{n-1} 2^i (s_i - s'_i)$$
$$g := s_k - s'_k - 1$$

contains at least $e^{n/4-1} \approx 2^{0.36n}$ monomials.

Proof. Define the polynomial $f' := \sum_{i \neq k} 2^i(s_i - s'_i) + 2^k$. Observe that Theorem 5.2.1 gives us a degree lower bound of $n-1$ on refutations of the polynomial $\{f'\}$. Theorem 5.2.2 translates this into a monomial size lower bound of $e^{n/4-1}$. The reduction below lifts this lower bound on $\{f'\}$ to the polynomials $\{f, g\}$.

We will show that a length l polynomial calculus refutation of the polynomials $\{f, g\}$ may be converted into a length l refutation of the polynomial $\{f'\}$ without increasing the number of monomials in each line. First notice that the polynomials f, f' are equivalent modulo the polynomial $g = s_k - s'_k - 1$. Given a PC refutation of $\{f, g\}$, we reduce each line by g (which effectively sets $s_k = 1$ and $s'_k = 0$), only reducing the number of monomials, to produce a refutation of $\{f'\}$. □

As a consequence of this corollary, polynomial calculus cannot derive $s_k = s'_k$ from the first equation using fewer than $e^{n/4-1}$ monomials. In comparison, cutting planes has small derivations that produce all of the bit-equalities.

Proposition 5.2.4. *There is an $O(n)$-length cutting planes derivation of all n bit-equalities $s_i = s'_i$ from the equation $\sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} 2^i s'_i = 0$.*

Proof. We extract the individual bit-equalities in the low-to-high sequence $s_0 = s'_0, s_1 = s'_1, \ldots s_{n-1} = s'_{n-1}$. Recall that in cutting planes, the equation $\sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} 2^i s'_i = 0$ is represented by two inequalities. Take the inequality $\sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} 2^i s'_i \geq 0$, and use the literal axioms on s_0, s'_0 to get $\sum_{i=1}^{n-1} 2^i s_i - \sum_{i=1}^{n-1} 2^i s'_i \geq -1$. Divide this by 2 to get $\sum_{i=1}^{n-1} 2^{i-1} s_i - \sum_{i=1}^{n-1} 2^{i-1} s'_i \geq 0$. Finally, use linear combination to multiply this by 2 and add it to the equation $\sum_{i=0}^{n-1} 2^i s'_i - \sum_{i=0}^{n-1} 2^i s_i \geq 0$ to obtain the result $s'_i - s_i \geq 0$. A symmetric derivation gives $s_i - s'_i \geq 0$. □

5.3 Array Multiplier Commutativity in $O(n^2)$ Steps

In this section, we give $O(n^2)$-length derivations for the word-level equivalence of the output bit-vectors **xy** and **yx** for both polynomial calculus and cutting planes. For polynomial

calculus, this proof was, in essence, previously written down in by Kaufmann et. al. in [128].

Swapping the order of inputs $\mathbf{x}, \mathbf{y}$ to a multiplier has the effect of reversing the order of tableau values in each column. In particular we have the equalities $t_{i,j}^{xy} = t_{j,i}^{yx}$ between tableau variables. The next lemma shows that from these bit-level equalities we can derive the word-level equality of the output bit-vectors $\mathbf{xy}$ and $\mathbf{yx}$ using only $O(n^2)$ linear combination steps. As both polynomial calculus and cutting planes can carry out such steps (recall that cutting planes represents "=" using two inequalities), they can both perform this proof.

Lemma 5.3.1. *Suppose that we have two n-bit array multipliers $\mathbf{xy}$ and $\mathbf{yx}$ implementing the two sides of the commutativity relation $xy = yx$. Further, suppose that we are given the n^2 equalities between the tableau variables $t_{i,j}^{xy} = t_{j,i}^{yx}$. Then there is a derivation in degree 1 and length $3n^2 + 1$ of the equation $\sum_{i=0}^{n-1} 2^i (xy)_i - \sum_{i=0}^{n-1} 2^i (yx)_i = 0$ that only uses linear combinations.*

Proof. We first derive two "conservation of weight" equations for the circuits $\mathbf{xy}$ and $\mathbf{yx}$ that state that the total weight of a multiplier's output bits is the same as the total weight of its tableau bits. We obtain these by adding up the adder constraints, weighting them so that the internal circuit variables cancel. For an adder in column i corresponding to a constraint $a_0 + a_1 + a_2 - 2c - d = 0$, this weighting simply scales the constraint up by a factor of 2^i. Once all the n^2 adder equations for an array multiplier $\mathbf{xy}$ have been summed together, we will arrive at an equation stating that the weight of tableau variables $t_{i,j}^{xy}$ is the same as the weight of the output variables $\mathbf{xy}$. After repeating the same steps for the multiplier $\mathbf{yx}$, we arrive at the two equations

$$\Big(\sum_{i,j=0}^{n-1} 2^{i+j} t_{i,j}^{xy} \Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i \Big) = 0$$
$$\Big(\sum_{i,j=0}^{n-1} 2^{i+j} t_{j,i}^{yx} \Big) - \Big(\sum_{i=0}^{2n-1} 2^i (yx)_i \Big) = 0$$

having used $2n^2$ linear combination steps. We then use a total of n^2 further derivation steps to replace $t^{yx}_{j,i}$ by $t^{xy}_{i,j}$ for each pair $i, j \in [0, n-1]$ in the latter equation. Finally, we subtract the two equations to finish the derivation. □

Theorem 5.3.2. *There is a polynomial calculus derivation in length $4n^2 + 1$, and also a cutting planes derivation in length $14n^2 + 2$, of the equation $\sum_{j=0}^{n-1} 2^i (xy)_i - \sum_{j=0}^{n-1} 2^i (yx)_i = 0$ from the array multiplier circuits* **xy** *and* **yx**.

Proof. Given the previous lemma, to complete our derivation we need to obtain the tableau equalities $t^{xy}_{i,j} = t^{yx}_{j,i}$. In polynomial calculus, we get each equality with one subtraction step with the equations $t^{xy}_{i,j} = x_i y_j$ and $t^{yx}_{j,i} = y_j x_i$. So deriving these equalities takes an additional n^2 polynomial calculus steps.

In cutting planes, it takes 3 linear inequalities (clauses) to represent a constraint $t^{xy}_{i,j} = x_i y_j$. From these, we can derive that $t^{xy}_{i,j} = t^{yx}_{j,i}$ in eight steps. Hence, deriving the tableau equalities takes $8n^2$ cutting planes steps. Afterwards, it takes two cutting planes steps to carry out each of the $3n^2 + 1$ linear combination steps of Lemma 5.3.1. □

Cutting planes also allows us to use Proposition 5.2.4 to prove bit-level equality from the equation $\sum_{i=0}^{n-1} 2^i (xy)_i - \sum_{i=0}^{n-1} 2^i (yx)_i = 0$, which gives the following corollary.

Corollary 5.3.3. *There is a length-$O(n^2)$ cutting planes derivation yielding all of the $2n$ equalities $(xy)_i = (yx)_i$ from the array multiplier circuits* **xy** *and* **yx**.

For other ring identities such as distributivity, we no longer have straightforward equalities between the tableau variables on either side of the identity. For distributivity, the natural generalization of these tableau variable equalities contains nonlinear terms. Before we give our cutting planes proofs, we introduce the (k, d)-cutting planes proof system in the next section as a convenient way to work with nonlinear terms within cutting planes.

5.4 Conservation of Weight for Adder Networks

In this section we use cutting planes to derive *conservation of weight* equations for networks of full adders: the weight of the output bits is the same as the weight of the input bits. This will allow us to derive the equation $\sum_{i,j} 2^{i+j} t_{i,j}^{xy} - \sum_i 2^i (xy)_i = 0$ for "adder network based" multiplier architectures, such as those discussed in Section 2.1. Our proof will automatically give us derivations of conservation of weight for larger compositions of the adder-based $+$ and $\times$ circuits that we described in Section 2.1, as well as for smaller subsets of full adders within a circuit. We start by defining some formalism to capture these multiplier designs.

Definition An *adder network* is a bipartite directed acyclic graph $G = (A, W, E)$, where A is a set of *adder* nodes, each with in-degree 2 or 3 and out-degree 2, and W is a set of *wire* nodes with in-degree and out-degree at most 1. The adders A and wires W are partitioned into columns $0, 1, 2, \ldots$, constraining outgoing edges E as follows. (1) A wire node w in a column i can only have an outgoing edge to an adder node a in the same column i. (2) An adder in column i has one outgoing edge (for the sum-bit) directed to a wire node in the same column i and another outgoing edge (for the carry-bit) to a wire node in the next column $i + 1$.

Definition An *adder network based multiplier* **xy** is a Boolean circuit with two phases. The first phase computes the *tableau variables* $t_{i,j}^{xy} = x_i y_j$ (also known as *partial products*). The second phase uses an adder network to compute the weighted sum of these tableau variables.

Ripple-carry adders are an example of an adder network. Examples of adder network based multipliers include array and diagonal multipliers, as well as Wallace tree multipliers using a final stage ripple-carry adder (as opposed to the standard carry-lookahead adder, which has non-adder components).

Definition Let $G = (A, W, E)$ be an adder network. Define the set of Boolean constraints $\Phi[G]$ on the variables (nodes) $A \cup W$ as follows. $\Phi[G]$ contains, for each adder $a \in A$, the constraint $a_0 + a_1 + a_2 - 2c - d = 0$, where a_0, a_1, a_2 are the adder a's input variables (parent

wires), c is its carry-bit (child wire) and d is its sum-bit (child wire). Note that we have represented each adder using a linear constraint rather than through clauses.

Definition Let $G = (A, W, E)$ be an adder network. For a variable (node) v in column i of the adder network G, define $\text{weight}(v) = 2^i$. Define the weight of a set of variables V as $\text{weight}(V) = \sum_{v \in V} \text{weight}(v) \cdot v$.

Define the *input boundary* $\partial_{\text{in}} G$ of the adder network as the subset of wires W with in-degree 0 and its *output boundary* as the subset of W with out-degree 0. Define the *conservation of weight constraint* for the adder network G, as $\text{weight}(\partial_{\text{in}} G) - \text{weight}(\partial_{\text{out}} G) = 0$.

We now state our main lemma for this section, which gives a small derivation, using only linear combination of constraints, of the conservation of weight equation for an adder network.

Lemma 5.4.1. *Let $G = (A, W, E)$ be an adder network. There is a length $|A|$ derivation of the conservation of weight constraint* $\text{weight}(\partial_{in} G) - \text{weight}(\partial_{out} G) = 0$. *that uses only linear combination of the constraints in $\Phi[G]$.*

Sketch. Add up the constraints corresponding to the adders A', multiplying each adder-constraint in a column i by 2^i. The non-boundary variables will cancel, leaving precisely the equation: $\text{weight}(\partial_{\text{in}} G) - \text{weight}(\partial_{\text{out}} G) = 0$.. □

As both polynomial calculus and cutting planes can perform these linear combination steps line-for-line, for both proof systems this lemma furnishes $O(n^2)$ length derivations of the equation $\sum_{i,j} 2^{i+j} t_{i,j}^{xy} - \sum_i 2^i (xy)_i = 0$ for array and diagonal multipliers.

5.5 Full Proof of Lemma 5.4.1

The following proposition is simple to verify.

Proposition 5.5.1. *Let $G = (A, W, E)$ be an adder network and let $A' \subseteq A$ be a subset of adders. The neighborhood subgraph $N(A')$ is also an adder network.*

Definition Let $G = (A, W, E)$ be an adder network. For a subset of adders $A' \subseteq A$, define its *input boundary* and *output boundary* as, respectively, the following subsets of the wires W:

$$\partial_{\text{in}} A' = \{w \in N(A') \cap W \mid w \text{ has in-degree 0 in } N(A')\},$$
$$\partial_{\text{out}} A' = \{w \in N(A') \cap W \mid w \text{ has out-degree 0 in } N(A')\}.$$

Define the *boundary* of A' as $\partial A = \partial_{\text{in}} A' \cup \partial_{\text{out}} A'$. Recall that the weight of a set of variables V is $\text{weight}(V) = \sum_{v \in V} \text{weight}(v) \cdot v$. Define the *conservation of weight constraint* for the subset A', as

$$\text{weight}(\partial_{\text{in}} A') - \text{weight}(\partial_{\text{out}} A') = 0.$$

Definition Let $G = (A, W, E)$ be an adder network. We call a subset of adders $A' \subseteq A$ *connected* if its neighborhood $N(A')$ is connected.

Proof of Lemma 5.4.1. We prove the lemma in the case where the adders A form one connected component. This suffices for the general case, as we can add together each component's conservation of weight equation.

We inductively show that the lemma holds for any connected subset $A' \subseteq A$. For the base case, notice that for a single adder $a \in A$ in column i, scaling its corresponding constraint up by 2^i gives the desired conservation of weight constraint for the subset $\{a\}$.

Assume the induction hypothesis that the lemma holds for a connected subset of adders $A' \subseteq A$, so that we have derived the conservation of weight constraint $\psi[A']$ in $|A'|$ steps. Let a be any adder outside of A' such that $A' \cup \{a\}$ is connected. Then a must be connected to A' through a nonempty set of wires $W_a = \partial A' \cap \partial\{a\}$. These wires in W_a do not appear in the boundary $\partial(A' \cup \{a\})$, so they will not appear in the conservation of weight constraint $\psi[A' \cup \{a\}]$. We will show that when we add $\psi[A'] + \psi[\{a\}]$, the terms for wires in W all cancel.

Assume that for some $w \in W_a$, we have $w \in \partial_{\text{in}} A'$. Then the term $\text{weight}(w) \cdot w$ appears in $\psi[A']$. Furthermore, since $w \in \partial_{\text{in}} A'$, the wire w is an output of the adder a. So the term $-\text{weight}(w) \cdot w$ appears in $\psi[\{a\}]$. Symmetrically, if $w \in \partial_{\text{out}} A'$, then $-\text{weight}(w) \cdot w$ appears in $\psi[A']$ and $\text{weight}(w)$ appears in $\psi[\{a\}]$. So adding the constraints, the terms containing wire variables in W_a cancel and we obtain

$$\begin{aligned}
\psi[A'] + \psi[\{a\}] &= [\big(\text{weight}(\partial_{\text{in}} A) - \text{weight}(\partial_{\text{out}} A)\big) \\
&\qquad + \big(\text{weight}(\partial_{\text{in}}(a)) - \text{weight}(\partial_{\text{out}}(a))\big) = 0] && (5.1) \\
&= [\text{weight}(\partial_{\text{in}}(A) \setminus W_a) + \text{weight}(\partial_{\text{in}}(a) \setminus W_a) - \text{weight}(\partial_{\text{out}} A \setminus W_a) \\
&\qquad - \text{weight}(\partial_{\text{out}}(a) \setminus W_a) = 0] && (5.2) \\
&= [\text{weight}(\partial_{\text{in}}(A' \cup \{a\})) - \text{weight}(\partial_{\text{out}}(A' \cup \{a\}) = 0] && (5.3) \\
&= \psi[A' \cup \{a\}], && (5.4)
\end{aligned}$$

where from lines 5.2 and 5.3 we have used that

$$\begin{aligned}
\partial_{\text{in}}(A' \cup \{a\}) &= (\partial_{\text{in}}(A) \setminus W_a) \cup (\partial_{\text{in}}(a) \setminus W_a), \\
\partial_{\text{out}}(A' \cup \{a\}) &= (\partial_{\text{out}}(A) \setminus W_a) \cup (\partial_{\text{out}}(a) \setminus W_a).
\end{aligned}$$

The inductive step takes one linear combination step to add together $\psi[A'] + \psi[\{a\}]$. So the derivation of $\psi[A' \cup a]$ took $|A'|+1$ steps. Deriving the final conservation of weight constraint $\psi[A]$ then takes $|A|$ steps. □

5.6 Polynomial Calculus Proofs

In this section, we will take a brief detour away from cutting planes. We will use Lemma 5.4.1 to give $O(n^2)$ length polynomial calculus derivations of word-level equivalence for circuits implementing any ring identity $L = R$. More precisely, we will prove the following theorem.

Theorem 5.6.1. *Fix a degree d ring identity $L = R$. Let* **L** *and* **R** *denote the sets of*

polynomials encoding the circuits L and R respectively, using a ripple-carry adder for $+$ and an array or diagonal multiplier or Wallace tree multiplier with a final stage ripple-carry adder for $\times$. There is a degree d, length $O(n^2)$ polynomial calculus derivation of the equation $\sum_i 2^i(L_i - R_i) = 0$.

Although Theorem 5.6.1 gives short polynomial calculus proofs of word-level circuit equivalence, we showed in Section 5.2 that polynomial calculus is very inefficient at deducing the bit-level consequences of equations of the form $\sum_i 2^i(s_i - s'_i) = 0$.

5.7 Proof of Theorem 5.6.1

5.7.1 Multiplier correctness

Let $\phi_{\text{spec}}(\mathbf{xy})$ denote the following degree two specification equation for the correctness of an n-bit multiplier $\mathbf{xy}$.

$$\Big(\sum_{i=0}^{n-1} 2^i x_i\Big)\Big(\sum_{i=0}^{n-1} 2^i y_i\Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i\Big) = 0.$$

If the multiplier circuit implies that $\phi_{\text{spec}}(\mathbf{xy})$ holds, then the multiplier is correct. In general, we will write the specification equation relating the inputs and outputs for a circuit $\mathbf{C}$ as $\phi_{\text{spec}}(\mathbf{C})$.

Theorem 5.7.1. *For an n-bit adder network based multiplier $\mathbf{xy}$ that uses $A(n)$ adders to implement the summation phase, there is a a degree 2 polynomial calculus derivation of $\phi_{spec}(\mathbf{xy})$ of length $A(n) + n^2$.*

Proof. In the first step of the proof, we apply Lemma 5.4.1 to the adder network used in the summation phase of the multiplier $\mathbf{xy}$. The input boundary of this adder network consists of the tableau variables $\{t_{i,j} \mid i, j \in [n]\}$, where each variable (wire) $t_{i,j}$ is in column $i + j$. The output boundary is the set of output bits $\{(xy)_i \mid i \in [2n]\}$. By Lemma 5.4.1, there is

length $2A(n) - 1$ polynomial calculus derivation of the conservation of weight equation

$$\Big(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2^{i+j} t_{i,j}\Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i\Big) = 0.$$

The second step of the proof is to substitute each $t_{i,j}$ by $x_i y_j$. Each of these substitutions takes one linear combination step. This yields

$$\Big(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2^{i+j} x_i y_j\Big) - \Big(\sum_{i=0}^{2n-1} 2^i (xy)_i\Big) = 0$$

which, after factoring the left term, can be seen to be the specification equation ϕ_{spec}. There are n^2 substitutions, so in total this proof has length $A(n) + n^2$. □

Corollary 5.7.2. *For an n-bit array or diagonal multiplier or Wallace tree multiplier with a final stage ripple-carry adder, there is a degree 2, length $2n^2$ polynomial calculus derivation of ϕ_{spec}.*

5.7.2 General ring identities

We generalize the above derivation to arbitrary ring expressions L through the following lemma.

Lemma 5.7.3. *Let $\mathbf{L}$ denote the circuit corresponding to a degree d ring expression L containing $A(n)$ adders. There is a degree d, length $O(A(n))$ polynomial calculus derivation for the specification equation $\phi_{spec}(\mathbf{L})$ that expresses $\mathbf{L}$ as a function of the input bit-vectors.*

Proof. Let $\mathbf{X}$ be the set of input bit-vectors for circuit $\mathbf{L}$. Let C, C' be sub-expressions of L with degrees d, d' and denote their output bit-vectors by $\mathbf{C}, \mathbf{C}'$ respectively. Let $|\mathbf{C}|$ denote the number of adders in a subcircuit C.

Assume the induction hypothesis that we have obtained, in $O(|\mathbf{C}| + |\mathbf{C}'|)$ lines, the degree d, d' specification equations $\phi_{\text{spec}}(\mathbf{C})$ and $\phi_{\text{spec}}(\mathbf{C}')$, that relate the inputs $\mathbf{X}$ to the outputs

$\mathbf{C}, \mathbf{C}'$. We will show that: (1) The specification equation $\phi_{\text{spec}}(\mathbf{C} + \mathbf{C}')$ for the circuit $\mathbf{C} + \mathbf{C}'$ has degree $\max(d, d')$ and has a derivation of length $O(|\mathbf{C} + \mathbf{C}'|)$, and (2) The specification equation $\phi_{\text{spec}}(\mathbf{C} \times \mathbf{C}')$ for the circuit $\mathbf{C} \times \mathbf{C}'$ has degree $d + d'$ and has a derivation of length $O(|\mathbf{C} \times \mathbf{C}'|)$.

For the first statement, apply Lemma 5.4.1 to the circuit $\mathbf{C} + \mathbf{C}'$ to obtain the equation $\sum_i 2^i (C + C')_i = \left(\sum_i 2^i C_i\right) + \left(\sum_i 2^i C'_i\right)$. Then use the specification equation $\phi_{\text{spec}}(\mathbf{C})$ to replace the term $\left(\sum_i 2^i C_i\right)$ with a degree d polynomial in the input variables $\mathbf{X}$, and similarly use the specification equation $\phi_{\text{spec}}(\mathbf{C}')$ to replace the term $\left(\sum_i 2^i C'_i\right)$ with a degree d' polynomial in the input variables $\mathbf{x}$. The result is the degree $\max(d, d')$ specification equation $\phi_{\text{spec}}(\mathbf{C} + \mathbf{C}')$. The number of lines used is linear in the number of new adders from the ripple-carry adder computing $\mathbf{C} + \mathbf{C}'$.

The second statement is proven similarly. Apply Lemma 5.4.1 to the outer multiplier of circuit $\mathbf{C} \times \mathbf{C}'$ to obtain the equation $\sum_i 2^i (C + C')_i = \left(\sum_i 2^i C_i\right)\left(\sum_i 2^i C'_i\right)$. Using the equations $\phi_{\text{spec}}(\mathbf{C})$ and $\phi_{\text{spec}}(\mathbf{C}')$ to express the right hand side in terms of $\mathbf{X}$, we get the degree $d + d'$ specification equation $\phi_{\text{spec}}(\mathbf{C} \times \mathbf{C}')$. □

Given circuits $\mathbf{L}$ and $\mathbf{R}$ corresponding to a ring identity $L = R$, applying Lemma 5.7.3 to both circuits and subtracting the specification equations gives yields short proofs of equivalence for circuits $\mathbf{L}$ and $\mathbf{R}$.

Theorem 5.7.4. *Let $L = R$ be a degree d ring identity. Let $A(n)$ denote the total number of adders contained in both the corresponding circuits $\mathbf{L}$ and $\mathbf{R}$. There is a degree d, length $O(A(n))$ polynomial calculus derivation of the equation*

$$\sum_i 2^i (L_i - R_i) = 0,$$

which asserts the word-level equality of the bit-vectors $\mathbf{L}$ and $\mathbf{R}$.

Corollary 5.7.5. *There are $O(n^2)$ length polynomial calculus derivations for any fixed word-*

level ring identity $L = R$ where the circuits $\mathbf{L}$ and $\mathbf{R}$ are implemented with array or diagonal multipliers, or Wallace tree multipliers with a final stage ripple-carry adder.

5.8 (k, d)*-Cutting Planes Proofs*

We write our cutting planes multiplier proofs in a more convenient format that we call (k, d)-*cutting planes*. Although cutting planes proofs only allow the use of linear inequalities, we will also be able to efficiently represent a large class of *nonlinear* Boolean inequalities using sets of linear inequalities.

Definition We say that a polynomial inequality ϕ on the Boolean variables X is (k, d)-nonlinear if is written in the form

$$\ell(X) + \sum_{i=1}^{k} \ell_i m_i \geq b$$

where $\ell(X)$ is an integer linear form (i.e., $\ell(X) = \sum_i c_i x_i$), each $\ell \in \{\ell_1, \ldots, \ell_k\}$ is a non-negative integer linear form (i.e., $\ell = \sum_i c_i x_i$ and each $c_i \geq 0$), each m_i is a monomial of degree at most $d - 1$ and with coefficient $+1$ or -1, containing only variables disjoint from ℓ_i, and lastly, b is an integer.

Definition We define the (k, d)-*cutting planes* proof system as follows. Each line is a (k, d)-nonlinear inequality on a set of Boolean variables $\{x_i\}$. Each variable x_i follows the *literal axioms* $x_i \geq 0$ and $-x_i \geq -1$. The proof rules are as follows.

Division rule: Writing $\ell(X) = \sum_i (c \cdot a_i) x_i$:

$$\frac{\sum_i (c \cdot a_i) x_i + \sum_i (c \cdot \ell_i) m_i \geq b}{\sum_i a_i x_i + \sum_i \ell_i m_i \geq \lceil \frac{b}{c} \rceil.}$$

Linear combination rule: Suppose that we have already derived the (k, d)-nonlinear inequalities ϕ and ϕ'. Let $\widetilde{m_1}, \widetilde{m_2}, \ldots, \widetilde{m_s}$ denote the monomial terms that ϕ and ϕ' have in common, and that we would like the resulting inequality to collect. So ϕ and ϕ' have the

form

$$\phi = \Big[\ell(X) + \sum_{i=1}^{s} \widetilde{\ell}_i \widetilde{m_i} + \sum_{i=1}^{k_1} \ell_i m_i \geq b\Big], \quad \phi' = \Big[\ell'(X) + \sum_{i=1}^{s} \widetilde{\ell}'_i \widetilde{m_i} + \sum_{i=1}^{k_2} \ell'_i m'_i \geq b'\Big],$$

where $s + k_1 + k_2$ is at most k so that the resulting inequality is (k, d)-nonlinear. Then we can make the following inference for any $\alpha, \beta \in \mathbb{N}$:

$$\frac{\ell(X) + \sum_{i=1}^{s} \widetilde{\ell}_i \widetilde{m_i} + \sum_{i=1}^{k_1} \ell_i m_i \geq b, \quad \ell'(X) + \sum_{i=1}^{s} \widetilde{\ell}'_i \widetilde{m_i} + + \sum_{i=1}^{k_2} \ell'_i m'_i + \geq b'}{\alpha\ell(X) + \beta\ell'(X) + \sum_{i=1}^{s} (\alpha\widetilde{\ell}_i + \beta\widetilde{\ell}'_i)\widetilde{m_i} + \sum_{i=1}^{k_1} \alpha\ell_i m_i + \sum_{i=1}^{k_2} \beta\ell'_i m'_i \geq \alpha b + \beta b'}$$

Factoring rule: If the (k, d)-nonlinear inequality ϕ contains two terms ℓm and $\ell' m$ with the same monomial m, then we can derive an inequality ϕ' that replaces these terms with the term $(\ell + \ell')m$. Syntactically:

$$\frac{\ell(X) + \sum_i \ell_i m_i + \ell m + \ell' m \geq b}{\ell(X) + \sum_i \ell_i m_i + (\ell + \ell')m \geq b}\,.$$

Distributing rule: This rule allows us to distribute a factored term appearing in an inequality. For a $(\ell + y_r)m \mapsto \ell m + y_r m$. Because of a technical detail related to the simulation size, we require that the two inequalities $\max(\ell) \geq \ell \geq 0$ have been derived from the literal axioms before making this inference.

$$\frac{\ell(X) + \sum_i \ell_i m_i + (\ell + y_r)m \geq b \qquad \max(\ell) \geq \ell \geq 0}{\ell(X) + \sum_i \ell_i m_i + \ell m + y_r m \geq b}\,.$$

Multiplication rule: This rule permits the multiplication of an inequality ϕ by a variable z, provided that the resulting inequality ϕz is (k, d)-nonlinear. Decomposing $\ell(X) = \ell(X)^+ -$

$\ell(X)^-$ into a sum of positive terms $\ell(X)^+$ and negative terms $\ell(X)^-$:

$$\frac{\ell(X)^+ - \ell(X)^- + \sum_i \ell_i m_i \geq b}{\ell(X)^+ \, z - \ell(X)^- \, z + \sum_i \ell_i m_i z - bz \geq 0} \, .$$

Remark. We emphasize that the (k, d)-cutting planes proof system distinguishes between inequalities ϕ and ϕ' that are semantically equivalent, but are syntactically different due to different factorizations. For example, the inequality $(x_1 + x_2)y_1 \geq b$ is *not* considered to be the same as the inequality $x_1y_1 + x_2y_1 \geq b$. The first inequality is $(1, 2)$-nonlinear while the second is $(2, 2)$-nonlinear. Our simulation will represent these two inequalities using two different (though semantically equivalent) sets of linear inequalities.

We also point out that in the distributing rule, we are limited to distributing out terms with coefficient ± 1. For example, fully expanding $(10y_1 + 3y_2)x_1x_2x_3 \geq b$ would take the following 3 applications of the distributing rule:

$$\begin{aligned}
(10y_1 + 3y_2)x_1x_2x_3 &\geq b \\
(10y_1 + 2y_2)x_1x_2x_3 + y_2x_1x_2x_3 &\geq b \\
(10y_1 + y_2)x_1x_2x_3 + 2y_2x_1x_2x_3 &\geq b \\
10y_1x_1x_2x_3 + 3y_2x_1x_2x_3 &\geq b.
\end{aligned}$$

Theorem 5.8.1. *Fix a pair of positive integers $k \geq 1$ and $d \geq 2$. A (k, d)-cutting planes proof of s lines can be simulated by a standard cutting planes proof of at most $(k + 4)d^k s$ lines.*

We spend the rest of this section describing how our simulation represents a (k, d)-nonlinear inequality using a set of linear inequalities. The remainder of the proof, the simulation for each (k, d)-cutting planes inference rule, is rather tedious, so we defer this to Section 5.10.

We begin with the following lemma showing that we may represent the integer-valued function computed by a degree d term using a set of at most d linear upper bounds.

Lemma 5.8.2. *Let ℓ be a non-negative linear form in the Boolean variables X and let $x_1x_2\ldots x_{d-1}$ be a degree $d-1$ monomial. Let $f(X)$ be an arbitrary integer-valued function of X and let b be an integer. The inequality $\ell x_1x_2\ldots x_{d-1}+f(X)\geq b$ has an equivalent set of d linear inequalities in the variables X and the function $f(X)$. The inequality $-\ell x_1x_2\ldots x_{d-1}+f(X)\geq b$ has an equivalent set of two linear inequalities in the variables X and the function $f(X)$.*

Proof. For the non-negative linear form $\ell=\sum_i c_ix_i$, let $\ell_{\max}$ denote the maximum value (i.e., $\ell_{\max}=\sum_i c_i$) for the non-negative linear form ℓ. Then $\ell x_1x_2\ldots x_{d-1}$ is exactly bounded from above by the d inequalities

$$\left\{\begin{array}{c}\ell\\ \ell_{\max}x_1\\ \vdots\\ \ell_{\max}x_{d-1}\end{array}\right\}\geq \ell x_1x_2\ldots x_{d-1}$$

since for any $0-1$ assignment to the variables in the monomial $x_1x_2\ldots x_{d-1}$, at least one of these inequalities is tight. Similarly, we can see that a negative term $-\ell x_1x_2\ldots x_{d-1}$ is exactly bounded from above by the two inequalities

$$\left\{\begin{array}{c}0\\ \ell_{\max}(d-1)-\ell_{\max}\big(\sum_{i=1}^{d-1}x_i\big)-\ell\end{array}\right\}\geq -\ell x_1x_2\ldots x_{d-1},$$

since the bottom upper bound is $-\ell$ when all the x_i's are 1, and non-negative if any $x_i=0$. So the inequality $\ell x_1x_2\ldots x_{d-1}+f(X)\geq b$ is equivalent to the d inequalities

$$\left\{\begin{array}{c}\ell+f(X)\\ \ell_{\max}x_1+f(X)\\ \vdots\\ \ell_{\max}x_{d-1}+f(X)\end{array}\right\}\geq b.$$

Likewise, the inequality $-\ell x_1 x_2 \dots x_{d-1} + f(X) \geq b$ is equivalent to the two inequalities

$$\left\{ \begin{array}{c} f(X) \\ \ell_{\max}(d-1) - \ell_{\max}\big(\sum_{i=1}^{d-1} x_i\big) - \ell + f(X) \end{array} \right\} \geq b. \qquad \square$$

Proposition 5.8.3. *Let $\phi = \sum_{i=1}^{k} \ell_i m_i + \ell(X) \geq b$ be a (k,d)-nonlinear inequality on the Boolean variables X. There exists an equivalent set $\widehat{\phi}$ of at most d^k linear integer inequalities on X.*

Proof. Use k applications of Lemma 5.8.2 to construct $\widehat{\phi}$. Each application increases the number of inequalities by a factor of at most d. $\square$

5.9 Optimal Cutting Planes Proofs

In the proof of commutativity, we were able to give cutting planes proofs without including nonlinear terms. However, when giving proofs for distributivity and other larger identities, nonlinear terms are difficult to avoid. This is where the (k,d)-cutting planes format is convenient for expressing $O(n^2)$ length cutting planes proofs of distributivity.

We then generalize the proofs for distributivity to a large class of degree two ring identities. For this larger class of identities, finding a (k,d)-cutting planes proof requires using the factoring rule to limit the number of nonlinear terms that appear in each inequality.

In the first half of these proofs, we apply Lemma 5.4.1 to each ripple-carry adder circuit $\mathbf{x}+\mathbf{y}$ to derive the conservation of weight equation $\sum_i 2^i(x_i + y_i) = \sum_i 2^i(x+y)_i$, and also to each adder network based multiplier circuit $\mathbf{xy}$ to derive the conservation of weight equation $\sum_{i,j} 2^{i+j} t_{i,j}^{xy} = \sum_i 2^i (xy)_i$.

This section focuses on the second half of these proofs, where the goal is to derive an equation stating that both sides hold equal weight in their multiplier tableau variables. The main idea is to derive an equation $\rho(i,j)$ relating the (i,j)-th tableau entry of each multiplier. Summing

these equations along i gives an equation $\rho(j)$ relating the j-th rows of each multiplier. Finally, adding together the equations $\rho(j)$ yields the desired equation for the full multiplier tableaus.

5.9.1 Distributivity

Theorem 5.9.1. *There is a length $O(n^2)$ cutting planes proof that the circuits $(\mathbf{x}+\mathbf{y})\mathbf{z}$ and $\mathbf{xz}+\mathbf{yz}$ for length n bit-vectors $\mathbf{x},\mathbf{y},\mathbf{z}$ have equal outputs.*

Proof. We will give a length $O(n^2)$ proof in $(5,2)$-cutting planes. By Theorem 5.8.1, this implies that there is an equivalent cutting planes proof that is only a constant factor larger. We begin with the following lemma, which gives a small derivation that the weight of the j-th row of the multiplier $(\mathbf{x}+\mathbf{y})\mathbf{z}$ is the same as the combined weight of the j-th rows of multipliers $\mathbf{xz}$ and $\mathbf{yz}$.

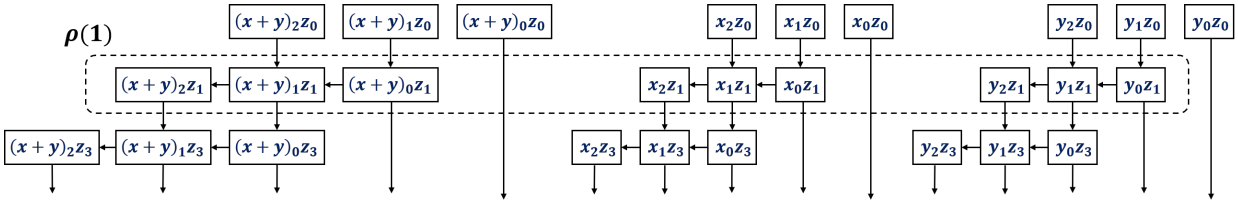

Figure 5.1: The equation $\rho(1)$ says that the weight held in row 1 of the multiplier $(x+y)z$ is the same as the weight held in row 1 of multipliers xz and yz.

Lemma 5.9.2. *For each $j \in [0,n-1]$ there is a length $O(n)$ derivation in $(5,2)$-cutting planes of the equality $\rho(j)$, defined as: $\sum_{i=0}^{n} 2^{i+j} \cdot t_{i,j}^{(x+y)z} = \sum_{i=0}^{n-1} 2^{i+j} \cdot (t_{i,j}^{xz} + t_{i,j}^{yz})$. from the circuits $(\mathbf{x}+\mathbf{y})\mathbf{z}$ and $\mathbf{xz}+\mathbf{yz}$.*

Proof. Fix $j \in [0,n-1]$. We give a constant length derivation of the cell-wise constraint $\rho(i,j)$, defined for $i \in [1,n-1]$ as $t_{i,j}^{(x+y)z} = t_{i,j}^{xz} + t_{i,j}^{yz} + c_{i-1}^{x+y} z_j - 2c_i^{x+y} z_j$. Define constraints $\rho(0,j)$ and $\rho(n,j)$ the same way absent the non-existing variables c_{-1}^{x+y}, c_n^{x+y}, $t_{n,j}^{xz}$ and $t_{n,j}^{yz}$. Adding up the constraints $\rho(i,j)$ will yield $\rho(j)$.

Start with the equation $x_i + y_i + c_{i-1}^{x+y} - 2c_i^{x+y} - (x+y)_i = 0$, given by the i-th adder in the ripple-carry adder $(\mathbf{x} + \mathbf{y})$. Multiplying this equation by z_j, we obtain the $(5, 2)$-nonlinear equation

$$x_i z_j + y_i z_j + c_{i-1}^{x+y} z_j - 2c_i^{x+y} z_j - (x+y)_i z_j = 0.$$

Substituting in the tableau variables $t_{i,j}^{(x+y)z}, t_{i,j}^{xz}, t_{i,j}^{yz}$ gives us $\rho(i, j)$.

To derive $\rho(j)$ we will add together the constraints $\rho(i, j)$ so that the carry terms telescope. We start with $\rho(n, j)$. Use linear combination to derive the equation $2\rho(n, j) + \rho(n-1, j)$:

$$2t_{n,j}^{(x+y)z} + t_{n-1,j}^{(x+y)z} = t_{n-1,j}^{xz} + t_{n-1,j}^{yz} + c_{n-1}^{x+y} z_j.$$

Repeating this step for $\rho(n-2, j), \ldots, \rho(0, j)$ gives $\rho(j)$. □

The rest of the proof combines equations $\rho(j)$ given by Lemma 5.9.2 with the conservation of weight equations, each derived in $O(n^2)$ steps by summing the adder constraints for the circuits $\mathbf{xz} + \mathbf{yz}$ and $(\mathbf{x} + \mathbf{y})\mathbf{z}$. For the first circuit $\mathbf{xz} + \mathbf{yz}$, note that the summation phases of $\mathbf{xz}$ and $\mathbf{yz}$, along with the ripple-carry adder computing the sum $\mathbf{xz} + \mathbf{yz}$, together form an adder network. The inputs of this adder network are the tableau variables of $\mathbf{xz}$ and $\mathbf{yz}$, and the output is the bit-vector $\mathbf{xz} + \mathbf{yz}$. Summing the adder constraints gives the conservation of weight equation

$$\sum_{i=0}^{2n} 2^i \cdot (xz + yz)_i = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} 2^{i+j} \cdot (t_{i,j}^{xz} + t_{i,j}^{yz}). \tag{5.5}$$

Summing the adder constraints of the multiplier circuit $(\mathbf{x} + \mathbf{y})\mathbf{z}$, we get the conservation of weight equation:

$$\sum_{i=0}^{2n} 2^i \cdot ((x+y)z)_i = \sum_{j=0}^{n-1} \sum_{i=0}^{n} 2^{i+j} \cdot t_{i,j}^{(x+y)z}. \tag{5.6}$$

Sum all of the equalities $\rho(j)$ to derive the equation ρ, stating that both sides have equal

weight in their tableau variables: $\sum_{i,j} 2^{i+j} \cdot (t_{i,j}^{xz} + t_{i,j}^{yz}) = \sum_{i,j} 2^{i+j} \cdot t_{i,j}^{(x+y)z}$. Combine this with equations 5.5 and 5.6 to obtain the final result: $\sum_i 2^i \cdot (xz + yz)_i = \sum_i 2^i \cdot ((x+y)z)_i$. $\square$

Notice that we only used the structure of the multipliers $(\mathbf{x}+\mathbf{y})\mathbf{z}$, $\mathbf{xz}$ and $\mathbf{yz}$ to derive the conservation of weight equations relating the sum of tableau variables to the output of the multiplier. The above proof is thereby compatible with any integer multiplier for which we can efficiently derive these conservation of weight equations. For example, we obtain $O(n^2)$ length proofs for Wallace tree multipliers using a final stage ripple-carry adder. In comparison, the best prior proof known for, say, checking that the middle pair of bits of an array multiplier and a Wallace tree multiplier are equal, was the quasi-polynomial size $n^{O(\log n)}$ resolution proof given in Chapter 3.

Reversing the order of multiplier inputs only has the effect of permuting the order of tableau variables, so the above proof also immediately generalizes to identities like $z(x+y) = zx+xz$ that mix distributivity and commutativity.

5.9.2 2-Colorable identities

We now generalize the ideas behind the proofs for the identity $(x+y)z = xz+yz$ to provide $O(n^2)$ length cutting planes proofs for larger instances of distributivity. We first show how to find small proofs for the equivalence of arbitrary expansions of the bit-vector expression $(x_1 + x_2 + \ldots + x_s)(y_1 + y_2 + \ldots + y_{s'})$. First we show the proof of the case where $s, s' = 2$.

This will give us $O(n^2)$ proofs for ring identities that can be written as the sum of independent bit-vector distributing or factoring steps. However, there exist identities such as $x(y+z) + wz = xy+(x+w)z$ which cannot be decomposed into a sum of independent distributing and factoring components. Nevertheless, we can still give an $O(n^2)$ length proof of this identity. We define the notion of a *2-colorable* degree two identity, a criterion that identifies the class of ring identities for which our technique can derive $O(n^2)$ length proofs.

Theorem 5.9.3. *There is a length $O(n^2)$ cutting planes proof that the left and right circuits*

corresponding to the identity $(x+y)(w+z) = xw + yw + xz + yz$ *on length* n *bit-vectors have equal outputs, for any order of addition on the right-hand side.*

Proof. This proof follows a similar idea as the proof of distributivity shown in Theorem 5.9.1. The difference is that each row of the circuit no longer has exactly the same tableau weight on both sides. Two extra nonlinear terms will appear in $\rho(j)$, containing the carry-bits c_{j-1}^{w+z} and c_j^{w+z} from the ripple-carry adder $w+z$. Our (k,d)-cutting planes proof will need to use the factoring and distributing rules to limit the accumulation of these nonlinear terms.

We obtain the following definition of $\rho(j)$ by taking the constraint $(w+z)_j = w_j + z_j + c_{j-1}^{w+z} - 2c_j^{w+z}$ for the j-th adder in the ripple-carry adder $\mathbf{w}+\mathbf{z}$, and multiplying both sides by the weight of the bit-vector $(\mathbf{x}+\mathbf{y})$, which is $\sum_i 2^i(x+y)_i$.

Definition For $j \in [1, n-1]$, define $\rho(j)$ as the constraint

$$\sum_{i=0}^{n} 2^{i+j} t_{i,j}^{(x+y)(w+z)} = \sum_{i=0}^{n-1} 2^{i+j}(t_{i,j}^{xw} + t_{i,j}^{xz} + t_{i,j}^{yw} + t_{i,j}^{yz}) + \Big(\sum_{i=0}^{n} 2^{i+j}(x+y)_i\Big) c_{j-1}^{w+z} - \Big(\sum_{i=0}^{n} 2^{i+j+1}(x+y)_i\Big) c_j^{w+z}.$$

Define $\rho(0)$ and $\rho(n)$ the same way absent non-existing variables such as c_{-1}^{w+z}, c_n^{w+z} and $t_{n,k}^{xw}$.

Like in the proof of distributivity, we will derive constraints $\rho(i,j)$ that, when added up for $i \in [0,n]$, sum to $\rho(j)$. For $i \in [1, n-1]$, define $\rho(i,j)$ as:

$$t_{i,j}^{(x+y)(w+z)} = t_{i,j}^{xw} + t_{i,j}^{yw} + t_{i,j}^{xz} + t_{i,j}^{yz} + (w_j + z_j)c_{i-1}^{x+y} - 2(w_j+z_j)c_i^{x+y} + (x+y)_i(c_{j-1}^{w+z} - 2c_j^{w+z}).$$

For $i = 0$ or $i = n$, define $\rho(i,j)$ the same way absent non-existing circuit variables.

Each $\rho(i,j)$ derivation will take a constant number of $(7,2)$-cutting planes lines. We give the derivation for the case where $i \in [1, n-1]$. The cases $i = 0$ and $i = n$ use essentially the same

derivation. The j-th adder in the ripple-carry adder $(\mathbf{w}+\mathbf{z})$ corresponds to the constraint $(w+z)_j = w_j+z_j+c_{j-1}^{w+z}-2c_j^{w+z}$. Multiply both sides by the variable $(x+y)_i$ and factor the first two terms to get the equation $(x+y)_i(w+z)_j = (x+y)_i(w_j+z_j)+(x+y)_i c_{j-1}^{w+z}-2(x+y)_i c_j^{w+z}$. Then use the equation $(x+y)_i = x_i+y_i+c_{i-1}^{x+y}-2c_i^{x+y}$ to expand the term $(x+y)_i(w_j+z_j)$, obtaining

$$\begin{aligned}(x+y)_i(w+z)_j =& x_i(w_j+z_j)+y_i(w_j+z_j)+(w_j+z_j)c_{i-1}^{x+y}-2(w_j+z_j)c_i^{x+y}\\ &+(x+y)_i c_{j-1}^{w+z}-2(x+y)_i c_j^{w+z}.\end{aligned}$$

We can then replace terms by their corresponding tableau variables to obtain $\rho(i,j)$ while using a maximum of 7 nonlinear terms per line.

To derive $\rho(j)$, we will add up each $\rho(i,j)$ in the same way as in the proof of Theorem 5.9.1. Again we start with $\rho(n,j)$. Use linear combination to derive $2\rho(n,j)+\rho(n-1,j)$ and then factor to obtain the $(3,2)$-nonlinear constraint:

$$\begin{aligned}2t_{n,j}^{(x+y)(w+z)}+t_{n-1,j}^{(x+y)(w+z)} =& t_{n-1,j}^{xw}+t_{n-1,j}^{yw}+t_{n-1,j}^{xz}+t_{n-1,j}^{yz}\\ &+(2(x+y)_n+(x+y)_{n-1})c_{j-1}^{w+z}-(4(x+y)_n\\ &+2(x+y)_{n-1})c_j^{w+z}+(w_j+z_j)c_{n-2}^{x+y}.\end{aligned}$$

Notice the cancellation of the term $2(w_j+z_j)c_{n-1}^{x+y}$. We repeat these linear combination and factoring steps for $\rho(n-2,j),\ldots,\rho(0,j)$, each time cancelling the nonlinear term $(w_j+z_j)c_i^{x+y}$. The result is $\rho(j)$.

Finally, we sum all the equations $\rho(j)$ to obtain the equation

$$\sum_{i,j=0}^{n} 2^{i+j}t_{i,j}^{(x+y)(w+z)} = \sum_{i,j=0}^{n-1} 2^{i+j}(t_{i,j}^{xw}+t_{i,j}^{xz}+t_{i,j}^{yw}+t_{i,j}^{yz}).$$

Regardless of the order of addition chosen for the circuit $(\mathbf{xw}+\mathbf{yw}+\mathbf{xz}+\mathbf{yz})$, we can sum

the adder constraints in the resulting network of full adders to derive its conservation of weight equation, like in the earlier proof of Theorem 5.9.1. Along with the conservation of weight equation for the multiplier $(\mathbf{x}+\mathbf{y})(\mathbf{w}+\mathbf{z})$, this gives the final result:

$$\sum_{i=0}^{2n+2} 2^i \cdot (xw + yw + xz + yz)_i = \sum_{i=0}^{2n+2} 2^i \cdot ((x+y)(w+z))_i.$$

This proof was $O(n^2)$ lines in $(7,2)$-cutting planes, since each of the n^2 equations $\rho(i,j)$ had a constant length derivation with at most 7 degree two terms per line, it took $O(n^2)$ steps to add them all up, and $O(n^2)$ steps to derive and apply the necessary conservation of weight equations. □

Theorem 5.9.4. *Let* $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_s}$ *and* $\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_{s'}}$ *be length* n *bit-vectors. Define the circuit* $\mathbf{L}$ *as* $(\mathbf{x_1} + \mathbf{x_2} + \ldots + \mathbf{x_s})(\mathbf{y_1} + \mathbf{y_2} + \ldots + \mathbf{y_{s'}})$. *Define the circuit* $\mathbf{R}$ *as* $\left(\sum_{\alpha,\beta} \mathbf{x}_\alpha \mathbf{y}_\beta\right)$, *representing the fully expanded version of* $\mathbf{L}$. *There is a length* $O(n^2)$ *proof that circuits* $\mathbf{L}$ *and* $\mathbf{R}$ *have equal outputs.*

Sketch. We will give a $(2(s+s'),2)$-cutting planes proof. We will interchangeably use the shorthand notation $\mathbf{x} = (\mathbf{x_1} + \mathbf{x_2} + \ldots + \mathbf{x_s})$ and $\mathbf{y} = (\mathbf{y_1} + \mathbf{y_2} + \ldots + \mathbf{y_{s'}})$.

The circuit $(\mathbf{x_1} + \mathbf{x_2} + \ldots + \mathbf{x_s})$ is built from $s-1$ ripple-carry adders adding the bit-vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_s}$ in some order. In order to express $(\mathbf{x_1} + \mathbf{x_2} + \ldots + \mathbf{x_s})$ in terms of the input bit-vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_s}$, we use the conservation of weight equation for the slice of adders belonging to the i-th column of $\mathbf{x}$. Let C_i^x denote the linear expression for the weight of the carry-in bits minus the weight of the carry-out bits for column i of the circuit $\mathbf{x}$ (see Figure 5.2 for an example). There are $s-1$ adders appearing in the i-th column of $\mathbf{x}$, hence C_i^x contains at most $2s-2$ carry-bits. Conservation of weight gives $(x_1+x_2+\ldots+x_s)_i = (x_1)_i + (x_2)_i + \ldots + (x_s)_i + C_i^x$. Symmetrically define C_j^y; this expression will contain at most $2s'-2$ carry-bits. Conservation of weight gives $(y_1+y_2+\ldots+y_s)_j = (y_1)_j + (y_2)_j + \ldots + (y_{s'})_j + C_j^y$.

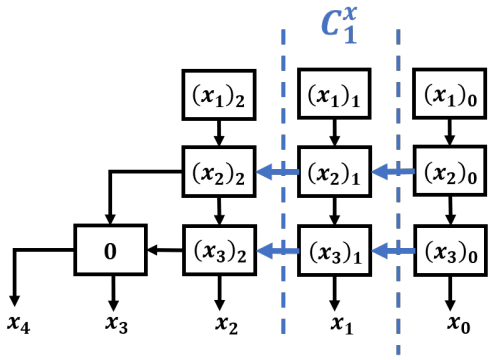

Figure 5.2: Example of the set of carry-bits appearing in the expression C_1^x for the circuit $(\mathbf{x_1} + \mathbf{x_2}) + \mathbf{x_3}$. In this case $C_1^x = c_0^{x_1+x_2} + c_0^{(x_1+x_2)+x_3} - 2c_1^{x_1+x_2} - 2c_1^{(x_1+x_2)+x_3}$.

The proof will derive the following $(2s' - 2, 2)$-nonlinear constraints $\rho(j)$ relating the j-th row of both sides of the identity: $\sum_i t_{i,j}^{xy} = \sum_{i,\alpha,\beta} t_{i,j}^{x_\alpha y_\beta} + ((\mathbf{x_1}) + (\mathbf{x_2}) + \ldots + (\mathbf{x_s}))C_j^y$ where $(\mathbf{x}_\alpha) = \sum_i 2^i (x_\alpha)_i$. Adding all these $\rho(j)$ equations together and applying conservation of weight yields the equality of the two circuits.

Like in the previous proofs, we derive the $(2(s+s')-4, 2)$-nonlinear constraints $\rho(i,j)$ defined as $t_{i,j}^{xy} = \sum_{\alpha,\beta} t_{i,j}^{x_\alpha y_\beta} + C_i^x y_j + C_j^y((x_1)_i + (x_2)_i + \ldots + (x_s)_i)$ that relate the tableau variable $t_{i,j}^{xy}$ to the tableau variables $t_{i,j}^{x_\alpha y_\beta}$. Adding up each $\rho(i,j)$ yields a $(2(s+s'), 2)$-nonlinear derivation of $\rho(j)$ of length $O(n)$.

At a high level, the derivation of $\rho(i,j)$ consists of the following steps.

$$t_{i,j}^{xy} = (x_1 + x_2 + \ldots + x_s)_i y_j \tag{5.7}$$

$$= ((x_1)_i + (x_2)_i + \ldots + (x_s)_i + C_i^x) y_j \tag{5.8}$$

$$= ((x_1)_i + (x_2)_i + \ldots + (x_s)_i) y_j + C_i^x y_j \tag{5.9}$$

$$= \sum_{\alpha,\beta} t_{i,j}^{x_\alpha y_\beta} + C_i^x y_j + C_j^y((x_1)_i + (x_2)_i + \ldots + (x_s)_i). \tag{5.10}$$

To go from equation 5.9 to equation 5.10 without creating too many nonlinear terms, while in the process of expanding y_j, we replace nonlinear terms $(x_\alpha)_i (y_\beta)_j$ by $t_{i,j}^{x_\alpha y_\beta}$ whenever they

arise. We also factor nonlinear terms containing carry-bits into the expression $C_j^y((x_1)_i + (x_2)_i + \ldots + (x_s)_i)$ when possible. In this way we can limit the total number of nonlinear terms in a line to $2(s+s')$. Each derivation of $\rho(i,j)$ has length $O(s+s')$, which is constant in the bit-width n. □

Theorem 5.9.4 yields optimal size proofs for any identity that can be split into a sum of distributing and factoring components. The remaining degree two identities, such as $x(y+z)+wz = xy+(x+w)z$, correspond to some interleaving of factoring and distributing steps. We show that we can still find $O(n^2)$ size proofs in many of these cases.

Definition Let $L = R$ be a degree two ring identity. A *2-coloring* for $L = R$ is an assignment of either the color red or blue to each bit-vector, with multiplicity (so a bit-vector may appear twice with different colors), such that: (1) each bit-vector in a sub-expression $(\mathbf{x_1} + \mathbf{x_2} + \ldots + \mathbf{x_r})$ has the same color as the bit-vector representing the sub-expression, (2) two sub-expressions that are multiplied together have opposite colors, and (3) the colored version of $L = R$, where a blue input bit-vector colored blue $\mathbf{x_i}$ is distinguished from its red counterpart $\mathbf{x_i}$, is still a valid ring identity.

For example, $(\mathbf{x}+\mathbf{y})\mathbf{z} = \mathbf{xz}+\mathbf{zy}$ has the 2-coloring $(\mathbf{x}+\mathbf{y})\mathbf{z} = \mathbf{xz}+\mathbf{zy}$. The more general form of distributivity in Theorem 5.9.4 clearly always has an 2-coloring. Lastly, the identity $\mathbf{x}(\mathbf{y}+\mathbf{z})+\mathbf{wz} = \mathbf{xy}+\mathbf{z}(\mathbf{x}+\mathbf{w})$ has the 2-coloring $\mathbf{x}(\mathbf{y}+\mathbf{z})+\mathbf{wz} = \mathbf{xy}+\mathbf{z}(\mathbf{x}+\mathbf{w})$. An example of an identity without a 2-coloring is $\mathbf{x}(\mathbf{y}+\mathbf{z})+\mathbf{w}(\mathbf{x}+\mathbf{y}) = \mathbf{y}(\mathbf{x}+\mathbf{w})+\mathbf{x}(\mathbf{z}+\mathbf{w})$.

Theorem 5.9.5. *Let $L = R$ be a 2-colorable degree two ring identity on length n bit-vectors $\mathbf{x_1}, \ldots, \mathbf{x_r}$. There is a length $O(n^2)$ cutting planes proof that the circuits $\mathbf{L}$ and $\mathbf{R}$ have equivalent outputs.*

Sketch. The idea is that since $L = R$ is 2-colorable, we will be able to derive equations $\rho(i,j)$ expressing the (i,j)-th tableau variables in the multipliers of $\mathbf{L}$ in terms of the (i,j)-th tableau variables in the multipliers of $\mathbf{R}$. These equations $\rho(i,j)$ will also contain a constant number

of nonlinear terms containing carry-bits from ripple-carry adders. We then sum up these equations in the same way as the previous proofs so that these nonlinear terms telescope. The result is an equation stating that **L** and **R** have equal weight in their multiplier tableaus.

We use the 2-coloring in the following way. If we attach the index i to each blue-colored input bit-vector and the index j to each red-colored input bit-vector, the result is still a valid ring identity. Therefore, we can derive an analagous identity to $L = R$ among the i-th and j-th bits of the input bit-vectors. For example, the 2-coloring of the ring identity $\mathbf{x}(\mathbf{y}+\mathbf{z})+\mathbf{w}\mathbf{z} = \mathbf{x}\mathbf{y}+\mathbf{z}(\mathbf{x}+\mathbf{w})$ gives rise to the valid identity $x_i(y_j+z_j)+w_iz_j = x_iy_j+z_j(x_i+w_i)$. Since the ring identity $L = R$ is fixed, this identity will have a constant number of nonlinear terms. Furthermore, it has a constant length derivation from the literal axioms.

This identity is the beginning of our derivation of the equation $\rho(i, j)$. Each equation $\rho(i, j)$ contains a constant number of nonlinear terms and relates the (i, j)-th tableau variables in the left and right circuits. For each multiplier that takes, as input, a blue bit-vector $(\mathbf{x_1}+\mathbf{x_2}+\ldots+\mathbf{x_s})$, we use the conservation of weight equation $(x_1)_i+(x_2)_i+\ldots+(x_s)_i = (x_1+x_2+\ldots+x_s)_i - C_i^x$ that we used in Theorem 5.9.4 to convert the left expression to the right expression. Apply the same steps to the red bit-vectors as well, then distribute out a constant number of nonlinear terms containing the carry-bits. We arrive at an expression where every nonlinear term in the resulting expression that does not contain a carry-bit has an equivalent tableau variable. Substituting in these tableau variables gives us the equation $\rho(i, j)$.

Like before, we can sum the equations $\rho(i, j)$ so that the terms containing carry-bits indexed by i telescope. Then summing the resulting equations $\rho(j)$ yields the equation stating that **L** and **R** have equal weight in their multiplier tableaus. □

5.10 Cutting Planes Simulation of (k, d)-Cutting Planes Rules

In this section we complete the proof of Theorem 5.8.1, restated below, by giving the simulation of each (k, d)-cutting planes rule in standard cutting planes.

Theorem 5.8.1. *Fix a pair of positive integers $k \geq 1$ and $d \geq 2$. A (k,d)-cutting planes proof of s lines can be simulated by a standard cutting planes proof of at most $(k+4)d^k s$ lines.*

We start with some definitions that will be useful in these simulations.

Definition Let ϕ be a (k,d)-nonlinear inequality. Let $\widehat{\phi}$ denote the equivalent set of at most d^k linear inequalities constructed in the proof of Proposition 5.8.3. Recall that for a non-negative linear form $\ell = \sum_i c_i x_i$, we define $\ell_{\max}$ as $\max(\ell) = \sum_i c_i$.

For a term ℓm, if $m = x_1 x_2 \dots x_{d-1}$ is a positive monomial, let $\widehat{\ell m}$ denote the set of linear upper bounds $\{\ell, \ell_{\max} x_1, \dots, \ell_{\max} x_{d-1}\}$. If $m = -x_1 x_2 \dots x_{d-1}$ is a negative monomial, then $\widehat{\ell m}$ is the set of two linear upper bounds $\{0, \ell_{\max}(d-1) - \ell_{\max}\big(\sum_{i=1}^{d-1} x_i\big) - \ell\}$.

Definition For a linear constraint $\psi \in \widehat{\phi}$ and a nonlinear term $\ell_i m_i$ in ϕ, let $\psi(\ell_i m_i)$ denote the expression that ψ uses to replace $\ell_i m_i$.

For example, consider the $(2,3)$-nonlinear inequality $\phi : (2y_1 + y_2 + y_3)x_1 x_2 - (y_3 + y_4)x_3 \geq b$. The set of linear inequalities $\widehat{\phi}$ consists of the following set of six linear inequalities:

$$\left\{ \begin{matrix} 2y_1 + y_2 + y_3 \\ 4x_1 \\ 4x_2 \end{matrix} \right\} + \left\{ \begin{matrix} 0 \\ 2 - 2x_3 - y_3 - y_4 \end{matrix} \right\} \geq b.$$

For, say, the linear inequality $\psi : 4x_1 \geq b$ from the set $\widehat{\phi}$, we have that $\psi((2y_1 + y_2 + y_3)x_1 x_2) = 4x_1$ and $\psi(-(y_3 + y_4)x_3) = 0$.

Recall that our simulation represents each (k,d)-nonlinear inequality ϕ by its corresponding set $\widehat{\phi}$ of at most d^k linear inequalities. We simulate each (k,d)-cutting planes proof rule $\phi \to \phi'$ by using at most $(k+4)d^k$ steps in standard cutting planes to derive $\widehat{\phi} \to \widehat{\phi'}$.

Division: Suppose that the (k,d)-cutting planes proof applies the division rule to the (k,d)-nonlinear inequality

$$\phi = \left[\sum_i (c \cdot a_i)x_i + \sum_i (c \cdot \ell_i)\boldsymbol{m}_i \geq b\right]$$

to obtain

$$\phi' = \left[\sum_i a_i x_i + \sum_i \ell_i \boldsymbol{m}_i \geq \left\lceil \frac{b}{c} \right\rceil\right].$$

We can get from $\widehat{\phi}$ to $\widehat{\phi'}$ within d^k cutting planes inferences by dividing each of the linear inequalities from $\widehat{\phi}$ by c.

Linear Combination: Suppose that the (k,d)-cutting planes proof has derived the (k,d)-nonlinear inequalities ϕ and ϕ' and uses linear combination to produce the inequality $\alpha\phi+\beta\phi'$. Let $\widetilde{\boldsymbol{m}_1}, \widetilde{\boldsymbol{m}_2}, \ldots, \widetilde{\boldsymbol{m}_s}$ denote the monomial terms that ϕ and ϕ' have in common, and that the inequality $\alpha\phi + \beta\phi'$ combines. So ϕ and ϕ' have the form

$$\phi = \big[\ell(X) + \sum_{i=1}^{s} \widetilde{\ell_i}\widetilde{\boldsymbol{m}_i} + \sum_{i=1}^{k_1} \ell_i \boldsymbol{m}_i \geq b\big], \qquad \phi' = \big[\ell'(X) + \sum_{i=1}^{s} \widetilde{\ell'_i}\widetilde{\boldsymbol{m}_i} + \sum_{i=1}^{k_2} \ell'_i \boldsymbol{m}'_i \geq b'\big],$$

where the number of terms $s + k_1 + k_2$ is at most k so that the linear combination $\alpha\phi + \beta\phi'$ is (k,d)-nonlinear. The inequality $\alpha\phi + \beta\phi'$ has the form

$$\alpha\phi + \beta\phi' = \big[\alpha\ell(X) + \beta\ell'(X) + \sum_{i=1}^{s} (\alpha\widetilde{\ell_i} + \beta\widetilde{\ell'_i})\widetilde{\boldsymbol{m}_i} + \sum_{i=1}^{k_1} \alpha\ell_i \boldsymbol{m}_i + \sum_{i=1}^{k_2} \beta\ell'_i \boldsymbol{m}'_i \geq \alpha b + \beta b'\big].$$

We will show that starting from the sets of linear inequalities $\widehat{\phi}$ and $\widehat{\phi'}$, each of the linear inequalities in the set $\widehat{\alpha\phi + \beta\phi'}$ has a one-step cutting planes derivation. More precisely, let $\psi \in \widehat{\alpha\phi + \beta\phi'}$. We will show how to select a pair of linear inequalities $\rho \in \widehat{\phi}$ and $\rho' \in \widehat{\phi'}$ such that $\alpha\rho + \beta\rho' = \psi$. Adding all the necessary pairs takes one line for each constraint in $\widehat{\alpha\phi + \beta\phi'}$. So in total deriving the set of linear constraints $\widehat{\alpha\phi + \beta\phi'}$ will take at most d^k

lines.

We select inequalities ρ and ρ' according to the choices of $\psi(\ell m)$ that ψ uses to replace each of its nonlinear terms ℓm. We have two cases that each term ℓm of ψ can fall under: the monomial m is either one of the collected monomials $\widetilde{m_i}$ or it is not. We cover the first case; the case where m is not a collected monomial is similar.

Let $(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})\widetilde{m_i}$ be a positive term in $\alpha\phi+\beta\phi'$. Suppose that ψ replaces a positive monomial term $(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})\widetilde{m_i}$ by the linear expression $\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'}$. Then we will have ρ and ρ' make the replacements

$$\rho(\widetilde{\ell_i}\widetilde{m_i}) = \widetilde{\ell_i} \qquad \text{and} \qquad \rho'(\widetilde{\ell_i'}\widetilde{m_i}) = \widetilde{\ell_i'}.$$

Otherwise, suppose that

$$\psi(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})\widetilde{m_i} = \max(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})x_{m_i}$$

for some choice of $x_{m_i} \in m_i$. Then our inequalities ρ and ρ' will make the replacements

$$\rho(\widetilde{\ell_i}\widetilde{m_i}) = \max(\widetilde{\ell_i})x_{m_i} \qquad \text{and} \qquad \rho'(\widetilde{\ell_i'}\widetilde{m_i}) = \max(\widetilde{\ell_i'})x_{m_i}.$$

For our next case, let $-(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})x_1x_2\ldots x_{d-1}$ be a negative degree d term in $\alpha\phi+\beta\phi'$. If $\psi \in \widehat{\alpha\phi+\beta}\phi'$ replaces this term by the approximation 0, then we choose ρ and ρ' that replace their corresponding terms $-\widetilde{\ell_i}x_1x_2\ldots x_{d-1}$ and $-\widetilde{\ell_i'}x_1x_2\ldots x_{d-1}$ by 0. Otherwise if ψ replaces this term by the approximation

$$\max(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})(d-1) - \max(\alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'})(\sum_{i=1}^{d-1} x_i) - \alpha\widetilde{\ell_i}+\beta\widetilde{\ell_i'},$$

then we choose ρ and ρ' that make the replacements

$$\rho(\widetilde{\ell_i}\widetilde{m_i}) = \max(\widetilde{\ell_i})(d-1) - \max(\widetilde{\ell_i})(\sum_{i=1}^{d-1} x_i) - \widetilde{\ell_i}$$

and

$$\rho(\widetilde{\ell_i'}\widetilde{m_i}) = \max(\widetilde{\ell_i'})(d-1) - \max(\widetilde{\ell_i'})(\sum_{i=1}^{d-1} x_i) - \widetilde{\ell_i'}.$$

Factoring: Suppose that the (k,d)-cutting planes proof applies the factoring rule to the (k,d)-nonlinear inequality

$$\phi = \Big[\ell(X) + \sum_{i=1}^{k-2} \ell_i \boldsymbol{m}_i + \ell_{k-1} m + \ell_k m \geq b\Big]$$

to obtain the $(k-1,d)$-nonlinear inequality

$$\phi' = \Big[\ell(X) + \sum_{i=1}^{k-2} \ell_i \boldsymbol{m}_i + (\ell_{k-1} + \ell_k) m \geq b\Big].$$

We do not need any cutting planes steps to simulate this because the set of linear inequalities $\widehat{\phi'}$ is a subset of $\widehat{\phi}$.

Distributing rule: This is the trickiest (k,d)-nonlinear proof rule to simulate. Our cutting planes simulation of this rule uses division, which turns out to by why the coefficient of the distributed term is limited to ± 1.

There is a subtlety to simulating distributing efficiently: we need to already have derived the inequalities $\ell_{\max} \geq \ell \geq 0$. Otherwise, since the linear form ℓ could contain up to n variables, deriving these inequalities from the literal axioms could take $O(n)$ steps. We give our simulation in the following lemma.

Lemma 5.10.1. *Let ϕ be an inequality of the form*

$$\ell(X) \pm (\ell + a_r)\boldsymbol{m} \geq b,$$

where $\ell(X)$ is a linear form, $\boldsymbol{m} = x_1 \dots x_d$ is a degree d monomial, and $\ell = c_1 a_1 + \ldots + c_{r-1} a_{r-1} + (c_r - 1) a_r$ is a non-negative linear form. In the case that ϕ contains $+(\ell + a_r)\boldsymbol{m}$,

define the target inequality

$$\phi' = \ell(X) + \ell m + a_r m \geq b.$$

Otherwise, if ϕ contains $-(\ell + a_r)m$, define the target inequality

$$\phi' = \ell(X) - \ell m - a_r m \geq b.$$

Starting from the set of linear inequalities $\widehat{\phi}$ and the inequalities $\ell_{max} \geq \ell \geq 0$, each target inequality in $\widehat{\phi'}$ has a length 4 cutting planes derivation.

Applying this lemma to the set of linear inequalities $\widehat{\phi}$ corresponding to the $(k-1, d)$-nonlinear inequality

$$\phi = \left[\ell(X) + \sum_i \ell_i m_i + (\ell + a_r)m \geq b\right]$$

takes us, in at most $4d^k$ cutting planes steps, to the set of at most d^k linear inequalities $\widehat{\phi'}$ corresponding to the (k, d)-nonlinear inequality

$$\phi' = \left[\ell(X) + \sum_i \ell_i m_i + \ell m + a_r m \geq b\right].$$

Proof of Lemma 5.10.1. Let $\ell = c_1 a_1 + \ldots + c_{r-1} a_{r-1} + (c_r - 1)a_r$. We first consider the case where $(\ell + a_r)x_1 \ldots x_d$ is a positive term. We need to go from the set of inequalities

$$\widehat{\phi} = \ell(X) + \left\{\begin{array}{c} c_1 a_1 + \ldots c_r a_r \\ (\ell_{\max} + 1)x_1 \\ \vdots \\ (\ell_{\max} + 1)x_{d-1} \end{array}\right\} \geq b$$

to the set of inequalities

$$\widehat{\phi'} = \ell(X) + \left\{ \begin{array}{c} c_1a_1 + \ldots + c_{r-1}a_{r-1} + (c_r - 1)a_r \\ \ell_{\max}x_1 \\ \vdots \\ \ell_{\max}x_{d-1} \end{array} \right\} + \left\{ \begin{array}{c} a_r \\ x_1 \\ \vdots \\ x_{d-1} \end{array} \right\} \geq b.$$

First notice that $\widehat{\phi} \subseteq \widehat{\phi'}$. The new inequalities in ϕ' that are not contained in ϕ are either of the form

$$\psi = \ell(X) + c_1a_1 + \ldots + c_{r-1}a_{r-1} + (c_r - 1)a_r + x_i \geq b,$$

or of the form

$$\rho = \ell(X) + \max(\ell)x_i + z \geq b$$

for some $z \in \{a_r, x_1, \ldots, x_{d-1}\}$. In the former case, we obtain ψ as follows in 4 steps.

$$\frac{\dfrac{\ell(X) + c_1a_1 + \cdots + c_ra_r \geq b \qquad 1 \geq a_r}{\ell(X) + c_1a_1 + \cdots + (c_r - 1)a_r \geq b - 1} \qquad \ell(X) + (\ell_{\max} + 1)x_i \geq b}{(\ell_{\max} + 1)\ell(X) + \ell_{\max}(c_1a_1 + \cdots + (c_r - 1)a_r) + (\ell_{\max} + 1)x_i \geq (b-1)(\ell_{\max} + 1) + 1}$$

Combining the last line with $\ell \geq 0$, we get

$$(\ell_{\max} + 1)\ell(X) + (\ell_{\max} + 1)(c_1a_1 + \cdots + (c_r - 1)a_r) + (\ell_{\max} + 1)x_i \geq (b-1)(\ell_{\max} + 1) + 1.$$

(If the coefficient of the distributed variable a_r was not ± 1, then the right hand side becomes too small for the following division step.) Dividing by $(\ell_{\max} + 1)$, we get the final result:

$$\ell(X) + c_1a_1 + \cdots + (c_r - 1)a_r + x_i \geq b.$$

For the latter case, if $z = a_r$ so that we need to derive an inequality $\rho \in \widehat{\phi'}$ of the form

$$\rho = \ell(X) + \max(\ell)x_i + a_r \geq b,$$

then we perform the following 4 step derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\ell(X)+c_1a_1+\cdots+c_ra_r\geq b \qquad -\ell\geq -1}{\ell(X)+a_r\geq b-\ell_{\max}} \qquad \ell(X)+(\ell_{\max}+1)x_i\geq b}{(\ell_{\max}+1)\ell(X)+\ell_{\max}(\ell_{\max}+1)x_i+a_r\geq(\ell_{\max}+1)b-\ell_{\max}} \qquad a_r\geq 0}{(\ell_{\max}+1)\ell(X)+\ell_{\max}(\ell_{\max}+1)x_i+(\ell_{\max}+1)a_r\geq(\ell_{\max}+1)b-\ell_{\max}}}{\ell(X)+\ell_{\max}x_i+a_r\geq b}$$

If $z=x_j$ and $j\neq i$ so that we need to derive an inequality $\rho\in\widehat{\phi'}$ of the form

$$\rho=\ell(X)+\max(\ell)x_i+x_j\geq b,$$

then we perform the following 2 step derivation.

$$\dfrac{\dfrac{\ell(X)+(\ell_{\max}+1)x_i\geq b \qquad \ell(X)+(\ell_{\max}+1)x_j\geq b}{(\ell_{\max}+1)\ell(X)+\ell_{\max}(\ell_{\max}+1)x_i+(\ell_{\max}+1)x_j\geq(\ell_{\max}+1)b}}{\ell(X)+\ell_{\max}x_i+x_j\geq b}$$

Now consider the case where ϕ contains the negative nonlinear term $-(\ell+a_r)x_1\dots x_{d-1}$. We need to go from the two inequalities

$$\widehat{\phi}=\ell(X)+\left\{\begin{array}{c}0\\(\ell_{\max}+1)(d-1)-(\ell_{\max}+1)\big(\sum_{i-1}^{d-1}x_i\big)-(\ell+a_r)\end{array}\right\}\geq b$$

to the four inequalities

$$\widehat{\phi'}=\ell(X)+\left\{\begin{array}{c}0\\\ell_{\max}(d-1)-\ell_{\max}\big(\sum_{i-1}^{d-1}x_i\big)-\ell\end{array}\right\}+\left\{\begin{array}{c}0\\(d-1)-\big(\sum_{i-1}^{d-1}x_i\big)-a_r\end{array}\right\}\geq b.$$

The two new inequalities in $\widehat{\phi'}$ that are not already contained in $\widehat{\phi}$ are

$$\ell_{\max}(d-1)-\ell_{\max}\left(\sum_{i-1}^{d-1}x_i\right)-\ell$$

and

$$(d-1) - \left(\sum_{i-1}^{d-1} x_i \right) - a_r.$$

To derive the former inequality, we use the following 4 step derivation:

$$\dfrac{\dfrac{\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - (\ell + a_r) \geq b \qquad 0 \geq -a_r}{\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - \ell \geq b} \qquad -\ell \geq -\ell_{\max}}{\ell_{\max}\ell(X) + \ell_{\max}(\ell_{\max}+1)(d-1) - \ell_{\max}(\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - (\ell_{\max}+1)\ell \geq \ell_{\max} b - \ell_{\max}}$$

Combine this with $\ell(X) \geq b$ to get

$$(\ell_{\max}+1)\ell(X) + \ell_{\max}(\ell_{\max}+1)(d-1) - \ell_{\max}(\ell_{\max}+1)\left(\sum_{i-1}^{d-1} x_i \right) - (\ell_{\max}+1)\ell \geq (\ell_{\max}+1)b - \ell_{\max}.$$

Dividing by $(\ell_{\max}+1)$, we get the final result:

$$\ell(X) + \ell_{\max}(d-1) - \ell_{\max}\left(\sum_{i-1}^{d-1} x_i \right) - \ell \geq b.$$

To derive the latter inequality we perform the following 4-step derivation.

$$\dfrac{\dfrac{\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - (\ell + a_r) \geq b \qquad \ell \geq 0}{\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - a_r \geq b} \qquad \ell(X) \geq b}{(\ell_{\max}+1)\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\big(\sum_{i-1}^{d-1} x_i\big) - a_r \geq (\ell_{\max}+1)b}$$

Combine this with $-a_r \geq -1$ to get

$$(\ell_{\max}+1)\ell(X) + (\ell_{\max}+1)(d-1) - (\ell_{\max}+1)\left(\sum_{i-1}^{d-1} x_i \right) - (\ell_{\max}+1)a_r \geq (\ell_{\max}+1)b - \ell_{\max}.$$

Dividing by $(\ell_{\max}+1)$, we get the final result:

$$\ell(X) + (d-1) - \left(\sum_{i-1}^{d-1} x_i \right) - a_r \geq b. \qquad \square$$

Multiplication rule: To check multiplication by a variable, suppose that the (k, d)-cutting planes proof has derived the inequality ϕ, and then uses multiplication by z to derive a (k, d)-nonlinear inequality ϕz. We will show that given the set of inequalities $\widehat{\phi}$, each of the inequalities in $\widehat{\phi z}$ has a constant-length cutting planes derivation.

Let $\psi \in \widehat{\phi z}$. Write $\ell(X) = \ell(X)^+ - \ell(X)^-$, where $\ell(X)^+$ contains the positive terms of $\ell(X)$ and $\ell(X)^-$ is a non-negative linear form containing the (negated) negative terms. Let

$$\phi = \big[\ell(X)^+ - \ell(X)^- + \sum_i \ell_i \boldsymbol{m}_i \geq b\big],$$

so that

$$\phi z = \big[\ell(X)^+ z - \ell(X)^- z + \sum_i \ell_i \boldsymbol{m}_i z - bz \geq 0\big].$$

We will obtain ψ by replacing each positive term $\ell_i \boldsymbol{m}_i z$ from ϕz by either ℓ, or a variable $x_{\boldsymbol{m}_i}$ contained in $\boldsymbol{m}_i$, or by the variable z. and then replacing each negative term $\ell_i x_1 x_2 \dots x_{d-2} z$ by either 0 or $\max(\ell_i)(d) - \max(\ell_i)\big(z + \sum_{i=1}^{d-2} x_i\big) - \ell_i$.

To derive a constraint $\psi \in \widehat{\phi z}$, we start from a constraint $\rho \in \widehat{\phi}$ that is selected according to the choices for term replacement in ψ. If ψ replaces a positive term $\ell_i \boldsymbol{m}_i z$ by ℓ_i, then ρ replaces its corresponding positive term $\ell_i \boldsymbol{m}_i$ by ℓ_i as well. If ψ replaces a positive term $\ell_i \boldsymbol{m}_i z$ by $\max(\ell_i) x_{\boldsymbol{m}_i}$, then ρ replaces its corresponding positive term by $\max(\ell_i) x_{\boldsymbol{m}_i}$ as well. Otherwise, if $z \notin \boldsymbol{m}_i$ and ψ replaces $\ell_i \boldsymbol{m}_i z$ by $\max(\ell_i) z$, then ρ replaces its corresponding term by $\max(\ell_i) x_{\boldsymbol{m}_i}$ for some variable $x_{\boldsymbol{m}_i} \in \boldsymbol{m}_i$. Lastly, if ψ replaces a negative monomial $\boldsymbol{m}_i z$ by 0, then ρ correspondingly replaces $\boldsymbol{m}_i$ by 0.

Let N denote the set of negative terms $\ell_i \boldsymbol{m}_i z$ in ϕz for which $\psi(\ell_i \boldsymbol{m}_i z) \neq 0$. The linear

inequalities $\psi \geq 0 \in \widehat{\phi z}$ and $\rho \geq 0 \in \widehat{\phi}$ then have the following forms, where each $c_i > 0$:

$$\psi = \sum_{\psi(m_i z)=z} c_i z + \sum_{\psi(m_i z)=x_{m_i}} c_i x_{m_i} - \sum_{m_i \in N} c_i \Big(\sum_{v \in m_i} v + z - |m_i| \Big) - bz$$

$$\rho = \sum_{\psi(m_i z)=z} c_i x_{m_i} + \sum_{\psi(m_i z)=x_{m_i}} c_i x_{m_i} - \sum_{m_i \in N} c_i \Big(\sum_{v \in m_i} v - |m_i| + 1 \Big) - b.$$

(For ease of reading this simulation of the multiplication rule, we abuse notation and use ψ and ρ to represent expressions instead of inequalities in the remainder of this section.) As shorthand, set the constants $C_1 = \sum_{\psi(m_i z)=z} c_i$ and $C_2 = \sum_{m_i \in N} c_i$. Observe that if $C_1 - C_2 - b > 0$, then $\psi \geq 0$ is a tautology. So without loss of generality, assume that $-C_1 + C_2 + b \geq 0$. From this inequality we will derive that $\psi - \rho \geq 0$. The first step goes as follows.

$$(1 - z) \geq 0 \tag{5.11}$$

$$(-C_1 + C_2 + b)(1 - z) \geq 0 \tag{5.12}$$

$$(C_1 - b)z + (b - C_1) - C_2(z - 1) \geq 0. \tag{5.13}$$

Going from Equation 5.11 to 5.13 takes one cutting planes step as the last step only rearranges terms. We use up to k linear combination steps (since in order for ϕz to be (k, d)-nonlinear, there were at most k terms in ϕ) using the literal axioms $1 - x_i \geq 0$ to show that $-\sum_{\psi(m_i z)=z} c_i x_{m_i} \geq -\sum_{\psi(m_i z)=z} c_i$. Adding this to Equation 5.13 and plugging in the definitions of C_1 and C_2 gives

$$\Big(\sum_{\psi(m_i z)=z} c_i - b \Big) z + \Big(b + \sum_{\psi(m_i z)=z} c_i x_{m_i} \Big) - \sum_{m_i \in N} c_i (z - 1) \geq 0.$$

Rearranging, we see that this is inequality $\psi - \rho \geq 0$:

$$\sum_{\psi(m_i z)=z} c_i (z - x) - \sum_{m_i \in N} c_i (z - 1) + b(1 - z) \geq 0.$$

Finally, adding $\rho \geq 0$ to $\psi - \rho \geq 0$ gives us $\psi \geq 0$. This derivation of ψ required at most $k + 2$ linear combination steps.

Chapter 6

PSEUDO-BOOLEAN EXPERIMENTS

6.1 Experiments

The goal of our experiments was to evaluate the potential of using cutting planes solvers to reason with mixtures of multiplication and bit-level logic. Such problems are a key weakness of using a SAT-based approach to "bit-blasting". We found several types of problems where pseudo-Boolean solvers performed well out-of-the-box. These include checking the word-level equivalence, commutativity, or correctness of different multipliers, extracting bit-equalities from word-level equalities, and verifying nonlinear bit-vector inequalities.

In our experiments, we used an Intel Core i7-6700K CPU at 4.00GHz with a memory limit of 8GB. The wall-clock time limit was set to 1200 seconds. We list experiment times in seconds (wall-clock time) and write TO if the time limit of 1200 seconds was exceeded. Our benchmarks are available at [93].

We used the two pseudo-Boolean solvers: Sat4j-CP [89], and RoundingSat [54]. These were chosen to represent the current state-of-the-art among solvers using native cutting planes reasoning (rather than reducing to SAT). The first solver, Sat4j-CP, uses the framework of Chai and Kuehlman [34] to generalize the CDCL algorithm that we explained in Chapter 2 to algorithms for conflict-driven pseudo-Boolean solving. This solver uses the *saturation* rule, instead of the division rule, in its conflict analysis:

$$\frac{\sum_i a_i x_i \geq b}{\sum_i \min\{a_i, b\} x_i \geq b}.$$

The second solver, RoundingSat [54], is a newer solver that also performs division-based

conflict analysis.[1]

While the conflict analysis routines found in RoundingSat and Sat4j-CP are based on similar frameworks, there are other more significant differences between these solvers. They use different internal data structures, and also tune the variable selection and restart heuristics differently. At a high level, Sat4j is oriented towards modularity, and is thus written in Java, whereas RoundingSat is oriented towards performance, and is accordingly written in C++.

The precise relationship between the saturation and division rules in both proof complexity and pseudo-Boolean solving is still largely unknown. As we will see in our experiments, some problems see much better performance with saturation-based conflict analysis, and for other problems division is clearly the superior choice. The two rules seem to be incomparable in terms of proof complexity. Gocht, Nördstrom and Yehudayoff showed that pseudo-Boolean solvers equipped with division can be exponentially stronger than solvers with saturation, and they also showed that simulating a single saturation step can require exponentially many division steps [65]. Research is still ongoing as to the best way to use these two rules in conflict analysis. In contrast, modern SAT solvers have largely converged to 1UIP clause learning for conflict analysis.[2] For a more detailed discussion of the differences between division and saturation in proof complexity and pseudo-Boolean solvers, see the previously mentioned survey [30].

Our experiments focused on integer multipliers with n-bit inputs and $2n$ bits of output. We report results on three different adder-based circuits to represent multiplication: array, diagonal, and Wallace-tree multipliers with final stage ripple-carry adder. As we discussed in Chapter 5, one of the limitations of current pseudo-Boolean solvers is that when they are given a CNF, they are limited to SAT-based reasoning. In order to benefit from pseudo-Boolean reasoning, it is important to represent multiplier circuits using the adder constraints

[1] RoundingSat actually uses a slightly more general form of the division rule in which the variable coefficients do not all need to share a common factor.

[2] See Chapter 2 to find our discussion on UIP-based learning schemes.

Instance	n	Sat4j-CP Word-level	RoundingSat Extract	Bit-level
array $x \cdot y = y \cdot x$	32	6	1	7
	64	8	6	14
	128	25	41	66
	256	171	158	329
diagonal $x \cdot y = y \cdot x$	32	7	1	8
	64	7	6	13
	128	25	41	66
	256	172	158	330
array spec-eqn	32	6	1	7
	64	18	6	24
	128	135	41	176
	256	TO	N/A	TO
diagonal spec-eqn	32	4	1	5
	64	18	6	24
	128	129	41	170
	256	TO	N/A	TO
diagonal $\equiv$ array	32	2	1	3
	64	5	6	11
	128	16	41	57
	256	102	158	260
		Gröbner [80]		
Instance	n	Word-level	Extract	Bit-level
gate-array $x \cdot y = y \cdot x$	32	1	N/A	N/A
	64	3	N/A	N/A
	128	27	N/A	N/A
	256	273	N/A	N/A
gate-array spec-eqn	32	1	N/A	N/A
	64	2	N/A	N/A
	128	14	N/A	N/A
	256	136	N/A	N/A

Table 6.1: Time to prove equivalences between multipliers using Sat4j and RoundingSat. We give the time to prove equivalence at the word-level, the time to extract the individual bits of the word-level equivalence, and the sum of these gives the total time to prove bit-level equivalence. We compare performance to Kaufmann's algebraic approach [80]

.

Instance	n	Sat4j-CP Word-level	RoundingSat Extract	Bit-level
Wallace $x \cdot y = y \cdot x$	16	1	1	2
	32	5	1	6
	48	TO	N/A	TO
	64	TO	N/A	TO
Wallace $\geq$ spec-eqn	16	1	N/A	N/A
	32	5	N/A	N/A
	48	65	N/A	N/A
	64	360	N/A	N/A
array $\equiv$ Wallace	16	1	1	2
	32	2	1	3
	48	45	3	48
	64	41	6	47

Table 6.2: Time to prove properties of Wallace tree multipliers using Sat4j and RoundingSat.

.

of the form $a_0 + a_1 + a_2 - 2c - d = 0$ using two inequalities instead of as a set of clauses, or to represent multiplication directly, without referencing a circuit, by writing the specification equation

$$\sum_{i,j=0}^{n-1} 2^{i+j} t_{i,j}^{xy} - \sum_{i=0}^{2n-1} 2^i (xy)_i = 0$$

along with the partial product constraints $t_{i,j}^{xy} = x_i y_j$. In these two ways, the pseudo-Boolean format allows us to "bit-blast" multiplication, along with other word-level functions, to a higher-level description than CNF while maintaining full bit-precision.

Our first set of experiments, presented in Table 6.1, uses the pseudo-Boolean solvers Sat4j-CP and RoundingSat to verify the word-level and bit-level equivalence of different multiplier circuits. More precisely, we use Sat4j-CP to prove an equation of the form $\sum_i 2^i (s_i - s'_i) = 0$ stating that the total weight of the outputs $\mathbf{s}, \mathbf{s}'$ is the same for the two multipliers. Then we have RoundingSat deduce, from this equation, each equality $s_i = s'_i$ individually in order to prove equivalence at the bit-level. Performance on bit-extraction scaled particularly well with the right choice of pseudo-Boolean solver, as shown in Table 6.3, which also includes

a comparison with the theoretical lower bound we showed for algebraic methods. Using these two steps, we can efficiently check the commutativity of array, diagonal, and Wallace-tree multipliers, as well as several equivalences between array, diagonal, and spec-equation multipliers. We can also check some of these properties of Wallace-tree multipliers for up to 32 or 64 bits.

We chose to use the saturation-based solver Sat4j-CP for the first step of deriving an equation $\sum_i 2^i(s_i - s'_i) = 0$ because it performed much better than the division-based solver RoundingSat. On the other hand, we will see that for the second step of deducing the bit-equalities $s_i = s'_i$, RoundingSat significantly outperformed Sat4j-CP.

An important step for showing word-level equivalence was to do some basic preprocessing to find equivalent partial products ($t_{i,j}^{xy}$ variables). Adding these equivalences was key to obtaining efficient solve times in Sat4j-CP. In contrast, we found that adding these equivalences did not help SAT-based solvers. We note that most bit-vector solvers, and many SAT solvers, already perform preprocessing to find equivalent variables; current pseudo-Boolean solvers based on cutting planes do not yet have such preprocessing.

To provide some context for these experimental results, we compared the performance of our pseudo-Boolean approach to Kaufmann's algebraic approach [80], which is currently the fastest method for verifying these properties. We replicated their verification of the commutativity and correctness of a simple gate-level array multiplier "btor", generated by Boolector, by using their tool, AMulet, in our environment to obtain the solve times at the bottom of Table 6.1. We note that AMulet is also capable of similarly fast solve times for more complicated gate-level multipliers such as Booth-encoded Wallace-tree multipliers. We direct interested readers to [80] for further experiments using the algebraic approach to verify commutativity, correctness, and equivalence of these other gate-level multiplier architectures.

Current pseudo-Boolean solvers have limited reasoning capabilities for these lower level multipliers. In particular, these solvers degenerate to SAT-based reasoning when given a CNF

n	RS	Sat4j-CP	Sat4j-Res	NaPS	#monomials
12	.001	7	.4	.1	7
16	.001	TO	3	2	20
20	.001		81	39	54
24	.001		TO	208	148
28	.002			Error	403
32	.002				1096
64	.009				3×10^6
128	.04				2×10^{13}
256	.2				2×10^{27}
512	.4				1×10^{55}

Table 6.3: Time in seconds to prove the equality $s_0 = s'_0$ from the equation $\sum_{i=0}^{n-1} 2^i(s_i - s'_i) = 0$ for the cutting planes solvers RoundingSat (RS) and Sat4j-CP, compared to the SAT-based solvers Sat4j-Res and NaPS [133]. We also compare with the polynomial calculus lower bound given by Corollary 5.2.3.

input. Our focus is not so much on verifying a large spectrum of multiplier circuits as on bit-vector solving, where we are free to choose the most efficient way to represent bit-vector multiplication.

We see that for array and diagonal multipliers, our approach (on adder-level multipliers) achieves comparable times to the algebraic approach (on gate-level multipliers) for proving commutativity and word-level equivalence. Furthermore, we are able to efficiently extract each of the individual bit-level equalities that a word-level equality implies.

For Wallace-tree multipliers with a final stage ripple-carry adder (wt-rca), we could check its equivalence with an array for 64 bits within 1 minute. We could also check commutativity for 32 bits in 5 seconds. However, we hit time-out on larger instances of 48 or 64 bits. We were also unable to completely verify the equivalence of a wt-rca and spec equation multiplier for 32-bit instances, though we could show that the the output of the wt-rca is at least as large as the output of the spec equation in 5 seconds. We see that Sat4j-CP has a harder time with these more complicated multiplier architectures.

Although we were able to check properties like commutativity and equivalence of multipliers, we were not able to use either pseudo-Boolean solver to efficiently verify more complicated identities such as distributivity, despite the fact that small cutting planes proofs exist, as shown in Chapter 5. Even with assistance, such as providing pre-computed lemmas, we were not able to check distributivity for more than 10 bits.

Our other experiments, presented in Table 6.4, use the solver RoundingSat to verify some nonlinear bit-vector inequalities involving untruncated multiplication and the operations "|" for bit-wise OR, "&" for bit-wise AND. We use these bit-wise operations to apply the *bit masks* "$\mid k$" and "$\&k$", where k is set to the constant alternating bit-string $(10)^{(n/2)}$. (This value was an arbitrary choice that contains a mix of 1s and 0s; we observed similar performance across all solvers with other values of k.) The inequalities listed follow from thinking of "|" and "&" as, respectively, computing the bit-wise maximum and minimum.

We compare RoundingSat's performance on these inequalities against the bit-vector solvers Boolector, Yices2, Z3 and CVC4. Our inputs to these bit-vector solvers used the word-level format SMT-LIB2 [7] to allow for full use of word-level reasoning and other non-SAT capabilities. We found that the bit-vector solvers (with the exception of Boolector) generally exceeded the time limit at 20 bits. On the other hand, when we "bit-blasted" multiplication using the spec-equation, RoundingSat outperformed all of the bit-vector solvers, with the exception of last inequality $(x \mid k)(y+1) \geq ky + x$, where Boolector won out by a few bits.

We were able to achieve this performance on inequalities where k is set to a constant number, but performance became no better than the bit-vector solvers when, in these inequalities, k was replaced a bit-vector variable z.

6.2 Conclusions & Directions

In the last two chapters, we have described a new approach to deciding nonlinear bit-vector formulas: include 1-bit adders among the set of essential building blocks along with the usual Boolean operations and express properties using pseudo-Boolean formulas rather than

Inequality	n	RS	Btor	Z3	Yices2	CVC4
$(x \mid k)z \geq kz$	16	17	14	21	31	44
	20	11	136	TO	TO	TO
	24	16	TO			
	28	501				
	32	TO				
$kz \geq (x\&k)z$	16	.06	10	15	172	31
	20	.5	117	1154	TO	TO
	24	.7	TO	TO		
	28	.6				
	32	.6				
$(x \mid k)z \geq (x\&k)z$	16	.2	14	22	31	44
	20	7	TO	TO	TO	TO
	24	2				
	28	629				
	32	TO				
$(x \mid z)(z \mid k) \geq kx$	16	.008	19	43	114	50
	20	.05	351	TO	TO	TO
	24	.2	TO			
	28	.2				
	32	.2				
$kx \geq (x\&z)(z\&k)$	16	.04	10	32	100	48
	20	.07	243	TO	TO	TO
	24	.1				
	28	23				
	32	7				
$(x \mid k)(y+1)$ $\geq ky + x$	16	.4	25	29	38	118
	20	TO	342	TO	TO	TO
	24		TO			

Table 6.4: Time to prove bit-vector inequalities containing both multiplication and bit-level operations. We compare RoundingSat (RS), Boolector 3.2.0 (Btor), Z3 4.8.7, Yices 2.6.2 and CVC4. The bit-vector k is the value $(10)^{(n/2)}$. & is bit-wise AND, | is bit-wise OR.

SAT formulas during "bit-blasting". We have shown, both experimentally and in principle, how pseudo-Boolean solvers based on cutting planes reasoning, when given these new bit-blasted formulas, can achieve levels of performance comparable to, or better than, the best alternative methods on a number of natural multiplier verification examples.

In particular, we have given $O(n^2)$-length cutting planes proofs for a broad class of properties of multipliers, matching the optimal efficiency of the best Gröbner basis algorithms for these properties at the word level, while also being able to extract bit-level properties. Importantly, Gröbner basis algorithms are not known to be able to extract such bit-level properties efficiently: We have shown that such methods require exponential time to extract bit-level consequences from word-level properties.

We also have shown experimentally that for several of these properties on inputs of up to 256 bits — namely, commutativity, correctness, and equivalence — pseudo-Boolean solvers can achieve performance comparable to that of the best algebraic solvers at the word-level, and, in contrast to algebraic methods, also solve these problems at the bit-level.

Finally, we have experimentally verified a number of crafted bit-vector inequalities, each involving a mixture of multiplication and bit-wise operations and have shown that our pseudo-Boolean approach can achieve much better verification performance than several of the best current bit-vector solvers.

The idea of using pseudo-Boolean solving for verifying nonlinear bit-vector formulas appears not to have been explored previously. One possible explanation for this is that when pseudo-Boolean solvers are run purely on CNF inputs, their reasoning collapses to that of CDCL SAT solvers, only much less efficient ones because of the more involved data structures and algorithms required in the pseudo-Boolean case. Our use of 1-bit adders as fundamental structures is critical to achieving the performance that we obtain.

Conflict-driven pseudo-Boolean solvers are still at a relatively early stage of development, especially compared to the 25+ years of concerted effort directed at optimizing Gröbner

basis algorithms and CDCL solvers. In particular, there is quite some variation in the different forms of conflict analysis methods used, and some of these methods have been shown to be quite weak. In fact, many solvers, such as NaPS [133] and Open-WBO [103], do not use any cutting planes reasoning and instead reduce the problem to SAT. Other shortcomings in the cutting planes reasoning used in current solvers are discussed in [53, 65, 141]. In our experiments, different conflict analysis methods worked best on different problems. For example, we found that the saturation-based solver Sat4j-CP worked much better than RoundingSat for checking word-level equalities. On the other hand, the division-based solver RoundingSat significantly outperformed Sat4j-CP when tasked with extracting bit-equalities, and also for checking bit-vector inequalities. This is in contrast with CDCL solvers where the best ideas for conflict analysis have largely converged on a single method that is used by all of the currently best solvers.

We view this work as providing a "call to arms" for pseudo-Boolean solver development, focusing especially on features that will be useful in verification of these kinds of bit-vector problems. In particular, though our experiments validate the pseudo-Boolean approach in principle, none of the solvers we used allowed us to verify the properties for which we provided more complex cutting planes proofs in Section 5.9. Thus, there is substantial scope for developing new methods and heuristics for pseudo-Boolean solving that can carry out much more of this cutting planes reasoning in practice.

Chapter 7

THE PROOF COMPLEXITY OF ASSOCIATIVITY?

At the end of Chapter 3, the last ring identity for which we did not know of polynomial size proofs was associativity, $\mathbf{x}(\mathbf{yz}) = (\mathbf{xy})\mathbf{z}$. In fact, we were not able to find polynomial size proofs in the stronger cutting planes proof system. We showed in Section 5.6 that polynomial calculus can derive the following equation for word-level associativity in $O(n^2)$ lines:

$$\left(\sum_i 2^i (x(yz))_i\right) = \left(\sum_i 2^i ((xy)z)_i\right).$$

But due to our bit-extraction lower bound (Corollary 5.2.3), it is still open as to whether there are polynomial size proofs for the equality of any pair of individual bits. It may still require an exponentially large polynomial calculus proof to prove, for instance, that the n-th output bits $(x(yz))_n$ and $((xy)z)_n$ are equal.

Since cutting planes *can* efficiently extract bits, we know that if a proof system is strong enough to perform the inferences of both polynomial calculus (over $\mathbb{Q}$), and cutting planes, it would be able to prove the equality of every pair of output bits $(x(yz))_i$ and $((xy)z)_i$ within $O(n^2)$ lines.

In the case of resolution, we attempted to find polynomial size proofs for associativity by using our critical strip decomposition from Chapter 3 to divide the outer multipliers $\mathbf{x}(\mathbf{yz})$ and $(\mathbf{xy})\mathbf{z}$ into narrow unsatisfiable strips. A polynomial size refutation of each strip would yield a polynomial size associativity proof. However, unlike the case of degree two identities, it is no longer straightforward to "walk down" the narrow strip of, say, the multiplier $(\mathbf{xy})\mathbf{z}$,

by branching on the tableau variables $t_{i,j}^{(xy)z}$, or by branching on the multiplier inputs $\mathbf{xy}, \mathbf{z}$. The issue is that the input bit-vector $\mathbf{xy}$ is the output of a multiplier. When we have a partial assignment to the bits $(xy)_i$, it is not clear how to enforce that these bits are consistent with the "true" values of the inputs $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The upcoming main theorem of this chapter, Theorem 7.1.1, provides some theoretical evidence that a polynomial size regular resolution proof cannot enforce this consistency.

Even in the more powerful cutting planes proof system, where we can efficiently derive the specification equation for each multiplier, we were not able to find polynomial size proofs for associativity. In Chapter 5, our strategy for proving identities such as commutativity and distributivity, was to derive (k, d)-nonlinear, local equations relating the tableau variables on both sides of the identity. For associativity, the relationship between the sets of tableau variables $t_{i,j}^{x(yz)}$ and $t_{i,j}^{(xy)z}$ is much more complicated. Within the (k, d)-cutting planes proof system, we were not able to express these relationships using fewer than n nonlinear terms. Unfortunately, having $k = n$ nonlinear terms in a line would cause the conversion into cutting planes from Theorem 5.8.1 to have an exponential overhead factor of over d^n.

Because of the difficulty of finding polynomial upper bounds in any of these proof systems, we make the following conjecture.

Associativity Conjecture Multiplier associativity requires exponentially large regular resolution proofs.

We conjecture this for regular resolution in particular for two reasons. First, despite our efforts, we were not able to extend our methods from Chapter 3, for finding small regular resolution proofs for degree two identities, to the case of associativity. And second, we will show in this chapter that checking the consistency of an assignment to the tableau variables of a multiplier requires exponentially large read-once branching programs, a model of computation that is closely related to regular resolution.

This may be a difficult lower bound to prove. The difficulty lies in understanding the

information encoded by partial assignments to the circuit variables. These circuit variable assignments constrain the bits of the input values x, y, z in a complicated way that is difficult to analyze. For instance, if we assign some values to a few output bits $(xy)_i$ of a multiplier **xy**, how does that constrain the possible values of **x** and **y**? Some kind of answer to questions like this seems necessary in order to prove a lower bound in resolution.

7.1 The Tableau Checking Problem

A potentially important theoretical piece of evidence that associativity requires exponentially large regular resolution proofs is that representing the following decision problem requires an exponentially large read-once branching program.

Definition For each n, the following CNF formula encodes the *tableau checking problem*:

$$T_n = \bigwedge_{i,j \in [n]} (\overline{x_i} \vee \overline{y_j} \vee t_{ij}) \wedge (x_i \vee \overline{t_{ij}}) \wedge (y_i \vee \overline{t_{ij}}).$$

This *tableau checking CNF* evaluates to 1 on total assignments θ satisfying the n^2 equalities $t_{ij} = x_i y_j$.

Theorem 7.1.1. *Every read-once branching program $\mathcal{B}$ computing T_n has at least $\frac{2^n-1}{n^2+2n}$ nodes.*

Notice that the tableau checking CNF T_n appears as a subset of the clauses encoding an n-bit multiplier circuit **xy**.

Recall from Chapter 3, Proposition 3.2.2 that a size s regular resolution refutation for a CNF formula ϕ can be converted into a size s read-once branching problem for the conflict clause search problem on ϕ and vice-versa. There are subtle, but important differences between a read-once branching program computing a CNF ϕ, and a read-once branching program searching for a conflict clause in the CNF ϕ. Thus, the hardness of tableau checking did not prevent us from finding polynomial size regular resolution proofs for degree two multiplier

identities such as commutativity. Our critical strip decomposition allowed us to side-step this obstacle.

For degree three identities such as associativity, it may no longer be possible to avoid the hardness of tableau checking for the following reasons. When we apply the critical strip decomposition to the outermost multiplier circuits $(\mathbf{xy})\mathbf{z}$ and $\mathbf{x}(\mathbf{yz})$, it is no longer straightforward to "walk down" the narrow strip of, say, the multiplier $(\mathbf{xy})\mathbf{z}$. Doing so requires partially assigning the tableau variables $t_{i,j}^{(xy)z)}$ and the multiplier inputs $\mathbf{xy}, \mathbf{z}$. The issue is that the input bit-vector $\mathbf{xy}$ is the output of a multiplier. If we cannot somehow guarantee that there are no tableau errors in the multiplier $\mathbf{xy}$, it seems that we cannot "trust" that a partial assignment to the bits $(xy)_i$ is consistent with the "true" values of the inputs $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The presence of just one incorrect tableau entry in the k-th column of multiplier $\mathbf{xy}$ can potentially cause bit-flips in any of the output bits $(xy)_i$ where $i \geq k$.

On the other hand, it is also possible that the presence of a multiplier's circuit variables could help somehow with tableau checking. Extending Theorem 7.1.1 to a lower bound on computing the CNF for a multiplier circuit $\mathbf{xy}$ (i.e. *multiplier circuit checking*) may provide a stepping stone towards a resolution lower bound for associativity.

In the rest of this chapter, we prove Theorem 7.1.1. Our proof closely follows the proof of Theorem 3.6 from [10], where Beame, Li, Roy and Suciu show exponential read-once branching program size lower bounds for monotone formulas corresponding to database queries. We generalize some of their definitions in order to extend these lower bounds to the non-monotone tableau checking CNFs T_n.

7.2 Proof of the Hardness of Tableau Checking

In this section we prove Theorem 7.1.1. The idea of the proof is to keep track of the "core" decisions made in each root-to-leaf path of the read-once branching program $\mathcal{B}$ computing T_n. We will show that

Definition Let Φ be a Boolean function. If the restriction $\Phi\restriction_{x=\bar{b}}$ is a constant function while the restriction $\Phi\restriction_{x=b}$ is nonconstant, then we say that $x = b$ is a *forced assignment* in Φ, the variable x is a *forced variable* in Φ, and b is the *forced value* of x.

Definition Let $\mathcal{B}$ be a read-once branching program for a Boolean function Φ. For a node u, let Φ_u denote the function represented by u in $\mathcal{B}$. Call a node u in $\mathcal{B}$ a *forced node* if the variable tested at u is a forced variable in Φ_u and a *decision node* otherwise. We say that $\mathcal{B}$ follows the *forcing rule* if, for every node u such that Φ_u has a forced variable, the node u tests a forcing variable.

7.2.1 Read-Once Branching Programs and the Forcing Rule

In this subsection, following [10], we show that any read-once branching program can be transformed into an equivalent read-once branching program of similar size that follows the forcing rule.

Lemma 7.2.1. *If Φ is a Boolean function $\{0,1\}^N \to \{0,1\}$ with a read-once branching program $\mathcal{B}$ of size s and containing N variables, then Φ has a read-once branching program $\mathcal{B}'$ of size at most sN that follows the forcing rule.*

Definition For a Boolean function Φ with forced variables, a *forcing sequence* π is a sequence of variable assignments generated by iteratively setting a forced variable to its forced value until reaching the unique function Φ^* containing no forced variables. We call Φ^* the *closure* of Φ. Let $U(\Phi)$ denote the (unique) set of variable assignments contained in a forcing sequence for Φ. We call $U(\Phi)$ the set of *forced assignments* for Φ.

Let $\mathcal{B}$ be a read-once branching program for the function Φ. For each read-once branching program edge $e = (u, v)$, define $U(e) = U(\Phi_v) - U(\Phi_u)$.

We observe that restricting a function cannot flip one of its forced assignments:

Proposition 7.2.2. *Let Φ be a Boolean function. If $U(\Phi)$ contains the assignment $x = b$, then for no restriction $\Phi\restriction_\rho$ of Φ does $U(\Phi\restriction_\rho)$ contain the assignment $x = \bar{b}$. Furthermore,*

the assignment $x = b$ is contained in $U(\Phi\restriction_\rho)$ if and only if ρ does not restrict x.

This proposition will be useful because the descendants of a node u correspond to restrictions of the function Φ_u computed at u.

We also observe that the closure Φ^* of a Boolean function Φ is invariant under setting forced variables to their forced values. This implies the following proposition.

Proposition 7.2.3. *If Φ is a Boolean function and $U(\Phi)$ contains the assignment $x = b$, then $U(\Phi\restriction_{x=b}) \subseteq U(\Phi)$.*

We now give the construction of $\mathcal{B}'$.

Definition Let $\mathcal{B} = (V, E)$ be a read-once branching program for the function Φ with nodes V and edges E. The set of nodes V' of $\mathcal{B}'$ is given by:

$$V' = V \cup \{(e, i) | e = (u, v) \in E, u \in V, 1 \leq i \leq |U(e)|\}.$$

For each original edge $e = (u, v) \in E$ such that $U(e)$ is nonempty:

1. The new vertices $(e, 1), \ldots, (e, |U(e)|)$ form a path from u to v that replaces the edge e.

2. The node v in $\mathcal{B}'$ is labeled by the same variable as node v in $\mathcal{B}$. Likewise, the edge $(u, (e, 1))$ in $\mathcal{B}'$ is labeled by the same value from $\{0, 1\}$ as the edge e in $\mathcal{B}$.

3. Let $\{x_1 = b_1, x_2 = b_2, \ldots, x_{|U(e)|} = b_{|U(e)|}\}$ be a forcing sequence for Φ_u. The variable x_i labels the new vertex (e, i) in $\mathcal{B}'$ and the value b_i labels the edge from (e, i) to $(e, i+1)$. The opposite outedge of (e, i), labeled by value $\overline{b_i}$, leads to the sink labeled by the constant $\Phi_v\restriction_{x_1=b_1, x_2=b_2, \ldots, x_i=\overline{b_i}}$.

Lastly, suppose that a node $w \in V$ is labeled by a variable x that appears in an assignment $x = b$ contained in $U(e)$ for some edge $e = (u, v)$ such that there is a path in $\mathcal{B}$ from v to w. We convert node w in $\mathcal{B}$ into a no-op node (i.e. a single-input, single-output node

representing precisely the same function as its parent node) in $\mathcal{B}'$ by removing its labeling variable x and its $\bar{b}$-outedge, and retaining its b-outedge.

To see that the above choice of retained edge is well-defined, by Proposition 7.2.2, if the assignment $x = b$ is in $U(e)$, then $x = \bar{b}$ does not appear in $U(e')$ for any other edge e' also appearing on a path in $\mathcal{B}$ to w. To see that the conversion to no-op nodes does not conflict with the conversion of edges to paths, observe that since $x = b$ appeared in $U(\Phi_v)$, by Proposition 7.2.2 and the read-once property of $\mathcal{B}$, we also have $x = b \in U(\Phi_w)$. Therefore by Proposition 7.2.3, we have $U(e_w) = \emptyset$ for each outedge e_w of node w in $\mathcal{B}$.

Lemma 7.2.4. *Let the read-once branching program $\mathcal{B}$ compute the function Φ. Then $\mathcal{B}'$ is a read-once branching program with no-op nodes for Φ that follows the forcing rule.*

Proof. We first show that $\mathcal{B}'$ is a read-once branching program. In particular, we show that every root-leaf path P in $\mathcal{B}'$ tests each variable at most once. The path P contains old nodes $u \in V$ and new nodes (e, i). Suppose that the variable x is tested twice along a path. The two tests cannot both occur at old nodes since $\mathcal{B}$ was read-once. The first test cannot be at an old node u and the second at a new node (e, i) because for any descendant node v of u in $\mathcal{B}$, the function Φ_v does not depend on x, and hence $x \notin U(e)$. It cannot first be tested by a new node (e, i) and then later by an old node u since the construction would have converted u to a no-op node. Finally, suppose that the two tests are done by two new nodes $((u_1, v_1), i)$ and $((u_2, v_2), j)$ in the path P. Then we must have $x \in U(v_1)$ and $x \notin U(u_2)$ where there is a path from v_1 to u_2 in $\mathcal{B}$. Therefore x is tested on this path from v_1 to u_2 so Φ_{u_2} does not depend on x, contradicting the requirement that $x \in U(v_2)$.

By construction, $\mathcal{B}'$ clearly follows the forcing rule. It remains to prove that Ψ, the function computed by $\mathcal{B}'$, is the same as Φ. We make the stronger claim, by induction, that for all original nodes $v \in V$, if θ' labels a path in $\mathcal{B}'$ from the root to v, then $\Psi[\theta'] = \Phi^*_v$, and $\theta' = \theta \cup U(\Phi_v)$ for some θ that labels a path in $\mathcal{B}$ from the root to v. This is trivially true for the root. If the claim holds true for the output nodes, then $\mathcal{B}$ correctly computes Ψ since

constant functions have no forced variables.

Let $v \in V$ and suppose that the claim is true for all vertices u such that there is some path θ' from the root to v in $\mathcal{B}'$ for which u is the last vertex in V on θ'. Suppose that the edge (u, v) is labeled b and the variable tested at u in $\mathcal{B}$ is x. We consider two cases: u either becomes a no-op node in $\mathcal{B}'$ or it does not.

In the case where u did not become a no-op node in $\mathcal{B}'$, every path θ' from the root to v through u is of the form $\theta' = \theta \cup \{x = b\} \cup U(e)$ for some θ that labels a path from the root to u in $\mathcal{B}'$. By induction, $\Psi[\theta] = \Phi_u^* = \Phi_u[U(\Phi_u)]$. Therefore $\Psi[\theta'] = \Psi[\theta \cup \{x = b\} \cup U(e)] = \Phi_u[\{x = b\} \cup U(\Phi_u) \cup U(e)]$. By definition, we had $U(\Phi_u) \cup U(e) = U(\Phi_v)$, so we finally obtain $\Psi[\theta'] = \Phi_u[\{x = b\} \cup U(\Phi_v)] = \Phi_v[U(\Phi_v)] = \Phi_v*$, as needed.

In the case where u became a no-op node in $\mathcal{B}'$, the original read-once branching program $\mathcal{B}$ had an ancestor w of u at which $(x = b) \in U((w', w))$ for some parent w' of w. By the read-once property of $\mathcal{B}$, it does not test x between w and u. Therefore, either Φ_u is constant or $x = b \in U(\Phi_u)$. In the former case, $\Phi_v = \Phi_u$. In the latter case, $U(\Phi_u) = U(\Phi_v) \cup \{x = b\}$ so that $\Phi_u^* = \Phi_v^*$. In both cases, the correctness of Φ_u^* implies the correctness of Φ_v^*.

□

7.2.2 Tableau Checking Lower Bound

In this subsection we show that any read-once branching program $\mathcal{B}$ computing the tableau checking CNF T_n and following the forcing rule must be exponentially large.

Lemma 7.2.5. *Any read-once branching program $\mathcal{B}$ computing T_n that follows the forcing rule has size $\geq 2^n - 1$.*

Theorem 7.1.1 then follows from combining Lemma 7.2.5 and Lemma 7.2.1.

We first use a standard method to extend $\mathcal{B}$ to another equivalent read-once branching program $\mathcal{B}^{\mathcal{X}}$ that is easier to work with. We will take $\mathcal{X} = \{x_i, y_j | i, j \in [n]\}$ in the following

definition.

Definition Let $\mathcal{B}$ be a read-once branching program and $\mathcal{X}$ a subset of its variables. For each node u in $\mathcal{B}$, let $\mathcal{X}_u$ be the subset of variables in $\mathcal{X}$ that appear as labels on paths from the root to u. Define $\mathcal{B}^{\mathcal{X}}$ as follows: The nodes of $\mathcal{B}^{\mathcal{X}}$ include the nodes of $\mathcal{B}$ plus some extra dummy nodes: For every edge (u, v) of $\mathcal{B}$, if u tests variable x and (u, v) is labeled by $b \in \{0, 1\}$, then

1. if $\mathcal{X}_v = \mathcal{X}_u$ or $\mathcal{X}_v = \mathcal{X}_u \cup \{x\}$ then $\mathcal{B}^{\mathcal{X}}$ has edge (u, v) exactly as $\mathcal{B}$ does.

2. otherwise, let $\{x_1, \ldots, x_a\} = \mathcal{X}_v \setminus \mathcal{X}_u \cup \{x\}$. The read-once branching program $\mathcal{B}^{\mathcal{X}}$ has dummy nodes $(u, v, 1), \ldots, (u, v, a)$ between u and v, with an edge from u to $(u, v, 1)$ labeled b. Each node (u, v, i) tests x_i and has both out-edges pointing to $(u, v, i + 1)$ for $i < a$ and pointing to v for (u, v, a). Further, define $\mathcal{X}_{(u,v,i)} = \mathcal{X}_u \cup \{x_1, \ldots, x_{i-1}\}$.

The following is immediate.

Proposition 7.2.6. *For any subset $\mathcal{X}$ of the variables in $\mathcal{B}$, the construction $\mathcal{B}^{\mathcal{X}}$ is a read-once branching program that computes the same function as $\mathcal{B}$ does, and $\mathcal{B}^{\mathcal{X}}$ satisfies the forcing rule if and only if $\mathcal{B}$ does. Furthermore, for each node u in $\mathcal{B}^{\mathcal{X}}$, every path in $\mathcal{B}^{\mathcal{X}}$ from the root to u tests precisely the same subset $\mathcal{X}_u$ of the variables in $\mathcal{X}$ (possibly testing additional variables outside $\mathcal{X}$).*

Let $\mathcal{B}$ compute T_n. We will prove the lower bound by considering a special class of paths in $\mathcal{B}^{\mathcal{X}}$ that end at nodes of $\mathcal{B}$; following [10], we call these *admissible* paths. We give such a definition and prove two properties: There are at least $2^n - 1$ distinct admissible paths, and no two admissible paths can lead to the same node. These two properties prove Lemma 7.2.5. We will base our definition of admissible paths on the set $\mathcal{A}$ of admissible assignments, which is defined to contain all total assignments θ to the variables of T_n such that $t_{ij} = x_i y_j$.

Definition For $i \in [n]$, define the set $\mathrm{Row}(i)$ of *variables in row i* to be x_i, t_{ij} for all $j \in [n]$. For $j \in [n]$, define the set $\mathrm{Col}(j)$ of *variables in column j* to be y_j, t_{ij} for all $i \in [n]$. Finally,

define the set $\mathrm{Cell}(i,j)$ of *variables in the cell* (i,j) to be $\{x_i, y_j, t_{ij}\}$.

Definition For $i \in [n]$, define the set $\mathrm{Row}(i)$ of *variables in row* i to be x_i, t_{ij} for all $j \in [n]$. For $j \in [n]$, define the set $\mathrm{Col}(j)$ of *variables in column* j to be y_j, t_{ij} for all $i \in [n]$. Finally, define the set $\mathrm{Cell}(i,j)$ of *variables in the cell* (i,j) to be $\{x_i, y_j, t_{ij}\}$.

Definition Let the set of *admissible assignments* $\mathcal{A}$ contain the assignments θ such that $T_n[\theta] = 1$.

The following properties of $\mathcal{A}$ are clear.

Proposition 7.2.7. *1. For every admissible assignment $\theta \in \mathcal{A}$, we have $T_n[\theta] = 1$.*

2. For each pair of values $b_x, b_y \in \{0,1\}$, there is precisely one value t_{ij} that is agreed on by all extensions of $x_i = b_x$, $y_j = b_y$ in $\mathcal{A}$.

3. The admissible assignments are symmetric with respect to rows and columns.

4. For any $i, j \in [n]$,

(a) For any assignment $\theta \in \mathcal{A}$, there is an assignment $\theta' \in \mathcal{A}$ that agrees with θ everywhere except possibly in the variables in $\mathrm{Row}(i)$ and has the opposite value of x_i from θ.

(b) For any assignment $\theta \in \mathcal{A}$, there is an assignment $\theta' \in \mathcal{A}$ that agrees with θ everywhere except possibly in the variables in $\mathrm{Col}(j)$ and has the opposite value of y_j from θ.

(c) For any assignment $\theta \in \mathcal{A}$ and variable w in $\mathrm{Cell}(i,j)$, there is an assignment $\theta' \in \mathcal{A}$ that agrees with θ everywhere except possibly in the variables in $\mathrm{Row}(i) \cup \mathrm{Col}(j)$ and has the opposite value of $t_{i,j}$ from θ.

Definition Let π be a partial assignment to the variables of T_n. We view π as a set of assignments to individual variables and so write $\pi \subseteq \pi'$ iff partial assignment π' extends π. We write $\pi || \pi'$ iff π and π' are consistent partial assignments, and $\pi \cap \pi'$ for the partial

assignment where they agree. If π is a partial assignment consistent with some (total) admissible assignment in $\mathcal{A}$, then we define the partial assignment *forced* by π to be $\pi^* = \bigcap_{\theta \in \mathcal{A},\ \theta || \pi} \theta$.

Proposition 7.2.8. *Let $i, j \in [n]$. Suppose that π^* sets t_{ij}. If π does not set any variable in* Row(i)*, then π^* sets $y_j = 0$. Likewise, if π does not set any variable in* Col(j)*, then π^* sets $x_i = 0$.*

Proof. Let π be an assignment that does not set any variable in Row(i), and for which π^* sets t_{ij}. That is, π^* sets the value of $x_i \wedge y_j$. Suppose that π^* does not set $y_j = 0$. Then there is some assignment $\theta \in \mathcal{A}$ consistent with π^* such that θ sets $y_j = 1$. By Proposition 7.2.7(4a) there is another assignment $\theta' \in \mathcal{A}$ that flips the value of x_i and agrees with θ everywhere except in the variables in Row(i). Since π does not set any variable in Row(i) and θ is consistent with π, the assignment θ' is also consistent with π. Therefore θ' is also consistent with π^*. However, while $y_j = 1$ in both θ and θ', they disagree on the value of x_i so π^* does not fix the value of $x_i \wedge y_j$, a contradiction.

By Proposition 7.2.7(3), the second case can be proven symmetrically. □

Definition Let P be a path in the read-once branching program $\mathcal{B}^{\mathcal{X}}$ and u be a node in P. Identify P with the partial assignment defined by its edges. Let P_u be the partial assignment defines by the prefix of P ending at u and x_u be the variable tested at node u.

Suppose now that P is consistent with some assignment in $\mathcal{A}$. We say that $x_u = b$ is *forced* in P iff P_u^* sets X_u to b; otherwise, we say that $X_u = b$ is a *free* assignment to X_u. Define the partial assignment Core(P) as the union of all free assignments in P.

Proposition 7.2.9. *If P is consistent with some assignment in $\mathcal{A}$, then* $(\text{Core}(P))^* = P^*$.

Note that the definition of Core(P) depends on the order in which variables are tested in P. Moreover, Core(P) may not even be a minimal set of free assignments along P; for example, a path that first sets $t_{ij} = 0$ and then $y_j = 0$ has both assignments in Core(P), but assigning

$y_j = 0$ first makes $t_{ij} = 0$ a forced assignment.

Definition Suppose that $\mathcal{B}$ computes T_n and follows the forcing rule, and let $\mathcal{B}^{\mathcal{X}}$ be defined as in Definition 7.2.2, with $\mathcal{X} = \{x_i, y_j | i, j \in [n]\}$. A path P in $\mathcal{B}^{\mathcal{X}}$ is called *admissible* iff it ends at a node of $\mathcal{B}$ that is not a forced node and is consistent with some assignment in $\mathcal{A}$ and $|\operatorname{Core}(P)| \leq n - 1$. Thus, in particular, every admissible path is consistent with some admissible total assignment.

Lemma 7.2.5 follows from two lemmas, which we state here and prove in the rest of the section.

Lemma 7.2.10. *Let P be an admissible path in $\mathcal{B}^{\mathcal{X}}$. If $|\operatorname{Core}(P)| < n - 1$, then there are two distinct admissible paths P' and P'' extending P such that $|\operatorname{Core}(P')| = |\operatorname{Core}(P'')| = |\operatorname{Core}(P)| + 1$.*

Since the empty path from the root of $\mathcal{B}^{\mathcal{X}}$ to itself is admissible, the following Corollary is immediate by induction.

Corollary 7.2.11. *There are at least $2^n - 1$ admissible paths in $\mathcal{B}^{\mathcal{X}}$.*

The second lemma is as follows:

Lemma 7.2.12. *If P_0 and P_1 are admissible paths in $\mathcal{B}^{\mathcal{X}}$ ending at some node v (of $\mathcal{B}$), then $P_0 = P_1$.*

Corollary 7.2.11 and Lemma 7.2.12 immediately imply Lemma 7.2.5.

To prove Lemma 7.2.10 we first show the following two lemmas:

Lemma 7.2.13. *Let P be an admissible path in the read-once branching program $\mathcal{B}^{\mathcal{X}}$ that computes T_n. Then there is some $i_0, j_0 \in [n]$ such that no variable in $\operatorname{Row}(i_0)$ or $\operatorname{Col}(j_0)$ is set by $|\operatorname{Core}(P)|$. For any such i_0, j_0, P^* does not set any variables in $\operatorname{Cell}(i_0, j_0)$. In particular, $T_n[P^*]$ is not a constant function.*

Proof. Since $|\operatorname{Core}(P)| \leq n - 1$, there must be some pair (i, j) such that $\operatorname{Core}(P)$ does not

set any variable in $\mathrm{Row}(i) \cup \mathrm{Col}(j)$. By Proposition 7.2.7(4c), for any variable $w \in \mathrm{Cell}(i,j)$. there are two assignments θ and θ' in $\mathcal{A}$ that agree with $\mathrm{Core}(P)$ but have different values for w so $(\mathrm{Core}(P))^*$ does not set any variable in $\mathrm{Cell}(i,j)$. We can extend P^* by setting $t_{ij} \neq x_i \wedge y_j$ to make T_n evaluate to 0. On the other hand, since P^* is consistent with an assignment in $\mathcal{A}$, by Proposition 7.2.7, T_n also evaluates to 1 on some extension of P^* and hence $T_n[P^*]$ is not constant. □

Lemma 7.2.14. *Suppose that $\mathcal{B}^{\mathcal{X}}$ computes T_n. Let P be an admissible path in $\mathcal{B}^{\mathcal{X}}$. All of the assignments of* $\mathrm{Core}(P)$ *are set at nodes of $\mathcal{B}$ and none of the variable assignments in* $\mathrm{Core}(P)$ *involves the forcing rule.*

Proof. We need to show that every assignment in P at a dummy node of $\mathcal{B}^{\mathcal{X}}$ is forced. The dummy node must test either x_i or y_j. First, suppose that variable x_i is tested at a dummy node (u,v,l) of $\mathcal{B}^{\mathcal{X}}$. By definition, the prefix $P_1 = P_{(u,v,l)}$ of P is admissible. Also, by definition $T_n[P_1]$ does not depend on x_i. Thus, by Lemma 7.2.13 there are $i_0, j_0 \in [n]$ such that $\mathrm{Core}(P_1)$ does not set any variable in $\mathrm{Col}(j_0)$ and P_1^* does not set any variable in $\mathrm{Cell}(i_0, j_0)$: in particular, P_1^* does not set y_{j_0}. Now since P_1 does not set x_i but $T_n[P_1]$ does not depend on x_i, the path P_1 must set $t_{i,j_0} = 0$ to eliminate the $x_i \vee \overline{t_{i,j_0}}$ clause from T_n (otherwise, setting $x_i = 0$ forces $T_n[P_1] = 0$, so it would depend on x_i). Hence $(\mathrm{Core}(P))^*$ sets $t_{i,j_0} = 0$. Applying Proposition 7.2.8 to $\mathrm{Core}(P)$, we obtain that P_1^* sets $x_i = 0$, and hence the value of x_i is forced in P, as required. The proof for the case where the dummy node tests y_j is symmetric. □

Proof of Lemma 7.2.10. Let P be an admissible path in $\mathcal{B}^{\mathcal{X}}$ and let v be the node of $\mathcal{B}$ at which P ends. By Lemma 7.2.13, v is not an output node and hence it tests a variable z. If z is set to the value b' in P^*, then we extend P with the forced assignment $z = b'$ while maintaining that T_n is nonconstant. We repeat this procedure, until reaching a node w of $\mathcal{B}$ that is not forced and does not test a variable assigned in P^*. The resulting path to w is admissible, and by Lemma 7.2.14, has the same core as the original path P. By

Lemma 7.2.13, w cannot be a leaf node of $\mathcal{B}$ so it must test a variable x that was not set in P^*. The admissible path P' extends the path from w using the 0-edge, and P'' will extend it using the 1-edge. By definition, both P' and P'' are consistent with assignments in $\mathcal{A}$. To make P' and P'' admissible, we follow P' and P'' through any forced assignments at dummy nodes or forced nodes of $\mathcal{B}^{\mathcal{X}}$ until they next reach an unforced node of $\mathcal{B}$; Lemma 7.2.14 ensures that such a node will be reached on both paths. Observe that $\text{Core}(P') = \text{Core}(P) \cup \{x = 0\}$ and $\text{Core}(P'') = \text{Core}(P) \cup \{x = 1\}$ so that both core sizes are precisely 1 more than $|\text{Core}(P)|$ and hence at most $n - 1$ as required. □

In the rest of this section, we prove Lemma 7.2.12.

Proof of Lemma 7.2.12. Since the admissible paths P_0 and P_1 in $\mathcal{B}^{\mathcal{X}}$ end at the same node v of $\mathcal{B}$ computing T_n, they have $T_n[P_0] = T_n[P_1]$. Assume that $P_0 \neq P_1$. Then P_0 and P_1 must diverge at some node of $\mathcal{B}^{\mathcal{X}}$, so both paths must differ on some variable z that they both assign. There are three cases: $z = x_i$, $z = y_j$ and $z = t_{ij}$.

In the case where $z = x_i$, assume that P_0 sets $x_i = 0$ and P_1 sets $x_i = 1$. Since P_1 is admissible, by Lemma 7.2.13 there exist $i_0, j_0 \in [n]$ such that $\text{Core}(P_1)$ does not set any variable in $\text{Col}(j_0)$ and P_1^* does not set y_{j_0}. By Proposition 7.2.8 applied to $\text{Core}(P_1)$, the path P_1 also cannot set t_{i,j_0}, since it sets $x_i = 1$ but does not set any variable from $\text{Col}(j_0)$. Therefore $T_n[P_1]$ has the clause $\overline{t_{i,j_0}} \vee y_{j_0}$. However, since P_0 was admissible and sets $x_i = 0$, it must also set $t_{i,j_0} = 0$ by the forcing rule, so $\overline{t_{i,j_0}} \vee y_{j_0}$ cannot be a clause of $T_n[P_0]$. But this contradicts our assumption that $T_n[P_0] = T_n[P_1]$.

We can handle the case where $z = y_j$ symmetrically to the $z = x_i$ case.

Finally, in the case where $z = t_{ij}$, assume that P_0 sets $t_{ij} = 0$ and P_1 sets $t_{ij} = 1$. The assignments $x_i = 1, y_j = 1$ appear in the path P_1 since they are forced by the assignment $t_{ij} = 1$. Therefore $T_n[P_1] = T_n[P_0]$ has no dependence on on the variables t_{ij}, x_i, y_j. Since $\overline{x_i} \vee \overline{y_j}$ is a clause in $T_n[t_{ij} = 0]$ but not a clause in $T_n[P_0]$, the path P_0 eliminates the clause

by assigning the variables x_i, y_j so that at least one of x_i or y_j is assigned 0. Therefore P_0 and P_1 fall into one of the two previous subcases. □

Finally, combining Lemma 7.2.5 with Lemma 7.2.1 completes the proof of Theorem 7.1.1.

Chapter 8

CONCLUSION AND FUTURE DIRECTIONS

In this thesis, we used the perspective of proof complexity to pave a new path towards verifying nonlinear integer arithmetic. In Chapter 3, we refuted the widely believed Multiplication Barrier Conjecture by constructing polynomial size proofs for any degree two ring identity, including commutativity and distributivity. This conjecture was previously believed to be the main obstacle preventing SAT solvers from effectively reasoning with multipliers. In Chapter 4, our SAT solver experiments indicated that although we could construct polynomial size resolution proofs, the degree of these polynomial size bounds was too large for practical applications. We needed smaller proofs. In Chapter 5, we constructed optimal, $O(n^2)$ length cutting planes proofs for the class of 2-colorable ring identities. Finally, in Chapter 6, we used cutting planes based solvers to efficiently verify the equivalence of different multiplier designs, and then gave some examples of bit-vector formulas where cutting planes solvers significantly outperformed other methods.

This work is only the starting point for this cutting planes approach to verifying nonlinear arithmetic. Pseudo-Boolean solvers based on cutting planes are still in a relatively early stage of development. Despite this, for certain problems such as bit-extraction and verifying multiplier equivalence, the stronger inference rules used by current pseudo-Boolean solvers can make them much more effective than their SAT counterparts. However, for many problems where small cutting planes proofs exist, pseudo-Boolean solvers struggle to actually find these proofs [53]. The last two decades have brought enormous, unforseen advances in SAT solving technology. Could the next decade take pseudo-Boolean solving along a similar trajectory of improvement?

8.1 Better Solvers for Cutting Planes and Bit-Vector Formulas

We give two experimental directions towards realizing the potential of cutting planes solvers for practical multiplication problems.

Problem 1. *Modify a bit-vector solver to bit-blast to a pseudo-Boolean formula instead of a SAT formula, and replace the CDCL SAT solver with a pseudo-Boolean solver.*

Bit-vector solvers perform preprocessing steps that are crucial for simplifying problems so that they are tractable for a SAT solver. What is the impact on performance if we retain this preprocessing, but replace the SAT-solver by a pseudo-Boolean solver? Which multiplier description should we bit-blast to? How does this modified bit-vector solver perform on crafted multiplier problems and standard bit-vector benchmarks?

Problem 2. *Improve cutting planes proof search in pseudo-Boolean solvers, either in general or specifically for multiplier problems.*

This direction is much more open-ended. In general, finding a small cutting planes proof to within a polynomial factor is NP-hard, as was shown by Göös, Koroth, Mertz and Pitassi in [67]. Although we were able to use pseudo-Boolean solvers to efficiently show the equivalence of different multiplier circuits in Chapter 6, we were not able to escape exponentially growing run-times for more complicated properties such as distributivity. Which improvements could allow a solver to find small proofs for the 2-colorable identities from Chapter 5? To what extent do improvements in pseudo-Boolean solvers help with solving bit-vector formulas containing multiplication?

8.2 Open Proof Complexity Problems

We constructed small proofs for many properties of multiplier circuits in resolution and cutting planes. However, we were unable to determine the proof complexity in several important cases. We state these cases in increasing order of estimated difficulty.

Problem 3. *Is there a length $O(n)$ cutting planes derivation for conservation of weight of an n-bit carry-lookahead adder? What about for other non-adder-based implementations of $+$?*

In Chapter 5, we gave cutting planes proofs that were compatible with any multiplier circuit composed of full and half-adders. However, there are many multiplier designs containing non-adder components. For instance, Wallace tree multipliers contain a carry-lookahead adder (CLA). The only obstacle preventing us from generalizing our proofs to these multipliers is that we need to derive the conservation of weight equation for this CLA. If there is a length $O(n)$, or even length $O(n^2)$ derivation, then the proofs in Chapter 5 immediately generalize to multipliers containing CLAs. On the other hand, it is possible that cutting planes does not have short proofs for non-adder based designs.

Problem 4. *Can we extend our $O(n^2)$-length cutting planes proofs to Booth-encoded multipliers?*

Booth encoding is a way to use a standard unsigned multiplier circuit to implement signed multiplication in two's complement notation. With this encoding, the tableau entries $t_{i,j}$ no longer represent partial products $x_i \wedge y_j$. There is also typically some additional logic that lets the circuit "skip" over consecutive strings of 1 or 0 digits in an input. An explicit example of a Booth encoded multiplier is given in [73].

It may be possible to extend our proofs from Chapter 5 to Booth encoded multipliers by solving the following two sub-problems. First, we need to efficiently derive some analogue to the local equations $\rho(i, j)$ that related the tableau entries $t_{i,j}$ to each other. Second, we need to efficiently drive the conservation of weight equation for multipliers augmented with the additional "skip" logic described above.

Problem 5. *Does multiplier associativity have polynomial size proofs in resolution, polynomial calculus, or cutting planes?*

In Chapter 7, we conjectured that associativity requires exponentially large regular resolution

proofs. Because of the limited expressiveness of clauses, we find it likely that an exponential lower bound also holds in general resolution.

Whether polynomial size proofs of (bit-level) associativity exist in polynomial calculus or cutting planes is less clear. While we were not able to find small proofs in either system, they are both able to succinctly derive properties of multiplier circuits that resolution cannot even write down succinctly.

BIBLIOGRAPHY

[1] Michael Alekhnovich and Alexander A. Razborov. Satisfiability, branch-width and Tseitin tautologies. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), Proceedings*, pages 593–603, Vancouver, BC, Canada, November 2002.

[2] Sean Anderson. Bit twiddling hacks. `https://graphics.stanford.edu/~seander/bithacks.html`. Accessed: 2020-05-15.

[3] Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In *Proceedings of the 39th Design Automation Conference, DAC 2002*, pages 725–730, New Orleans, LA, USA, June 2002.

[4] Fabrício Vivas Andrade, Márcia C. M. Oliveira, Antônio Otávio Fernandes, and Claudionor José Nunes Coelho Jr. SAT-based equivalence checking based on circuit partitioning and special approaches for conflict clause reuse. In Girard et al. [64], pages 397–402.

[5] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 114–127, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

[8] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.

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

[10] Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Exact model counting of query expressions: Limitations of propositional methods. *ACM Trans. Database Syst.*, 42(1), February 2017.

[11] Paul Beame and Vincent Liew. Towards verifying nonlinear integer arithmetic. In *Computer Aided Verification*, pages 238–258. Springer International Publishing, 2017.

[12] Paul Beame and Vincent Liew. Towards verifying nonlinear integer arithmetic. *J. ACM*, 66(3):22:1–30, June 2019.

[13] Armin Biere. Challenges in bit-precise reasoning. In *Formal Methods in Computer-Aided Design, FMCAD 2014*, page 3, Lausanne, Switzerland, October 2014.

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

[15] Armin Biere. Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. University of Helsinki, 2016.

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

[17] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[18] Armin Biere, Manuel Kauers, and Daniela Ritirc. Challenges in verifying arithmetic circuits using computer algebra. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'17)*, 2017.

[19] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.

[20] Beate Bollig. Larger lower bounds on the OBDD complexity of integer multiplication. *Inf. Comput.*, 209(3):333–343, 2011.

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

[22] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear

programming. In *Proceedings of the ASPDAC 2002 / VLSI Design 2002*, pages 741–746, Bangalore, India, January 2002.

[23] Robert Brummayer and Armin Biere. 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: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT($\{\mathcal{BV}\}$) solver for hard industrial verification problems. In *Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007*, pages 547–560, Berlin, Germany, July 2007.

[25] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4SMT solver. In *Proceedings, Computer Aided Verification, 20th International Conference, CAV 2008*, pages 299–303, 2008.

[26] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[27] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[28] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, 1994.

[29] Sam Buss, Dima Grigoriev, Russell Impagliazzo, and Toniann Pitassi. Linear gaps between degrees for the polynomial calculus modulo distinct primes. *Journal of Computer and System Sciences*, 62(2):267 – 289, 2001.

[30] Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2020. Chapter to appear in the 2nd edition. Draft version available at `http://www.csc.kth.se/~jakobn/research/ProofComplexityChapter.pdf`.

[31] Samuel R. Buss and Maria Luisa Bonet. An improved separation of regular resolution from pool resolution and clause learning. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7313, pages 244–57, Trento, Italy, June 2012.

[32] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas:

Resolution refinements that characterize DLL algorithms with clause learning. *Logical Methods in Computer Science*, 4(4), 2008.

[33] Samuel R. Buss and Leszek Kolodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10(2), 2014.

[34] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, March 2005.

[35] Supratik Chakraborty, Ashutosh Gupta, and Rahul Jain. Matching multiplications in bit-vector formulas. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 131–150, Cham, 2017. Springer International Publishing.

[36] M. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2019.

[37] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi. Verification of gate-level arithmetic circuits by function extraction. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[38] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 174–183, New York, NY, USA, 1996. ACM.

[39] Christopher Condrat and Priyank Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 618–631, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[40] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

[41] W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25 – 38, 1987.

[42] Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the data encryption standard. In Steven D. Galbraith, editor, *Cryptography and Coding*, pages 152–169, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[43] Cybersecurity and Infrastructure Security Agency. Ics alert (ics-alert-20-063-01): Sweyntooth vulnerabilities. `https://www.us-cert.gov/ics/alerts/ics-alert-20-063-01`. Accessed: 2020-05-28.

[44] S. Dantchev and S. Riis. Planar tautologies hard for resolution. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 220–229, Oct 2001.

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

[46] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.

[47] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[48] Leonardo Mendonça de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 2005.

[49] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In Horvitz and Jensen [74], pages 211–219.

[50] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02)*, pages 635–640, July 2002.

[51] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[52] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *J. Satisf. Boolean Model. Comput.*, 2:1–26, 2006.

[53] Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 75–93, Cham, 2018. Springer International Publishing.

[54] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1291–1299. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[55] Luca Fanucci and Jürgen Teich, editors. *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*. IEEE, 2016.

[56] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical

study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 328–343, New York, NY, USA, 2017. Association for Computing Machinery.

[57] U.S. Food and Drug Administration. FDA informs patients, providers and manufacturers about potential cybersecurity vulnerabilities in certain medical devices with bluetooth low energy. `https://www.fda.gov/news-events/press-announcements/fda-informs-patients-providers-and-manufacturers-about-potential-cybersecurity-vulnerabilities`. Accessed: 2020-05-28.

[58] Anders Franzen. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

[59] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 121–128, Austin, TX, 2010. FMCAD Inc.

[60] Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *AAAI*, pages 1136–1143. AAAI Press, 2015.

[61] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007*, pages 519–531, Berlin, Germany, July 2007.

[62] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[63] David Gelles. Boeing expects 737 Max costs will surpass $18 billion. `https://www.nytimes.com/2020/01/29/business/boeing-737-max-costs.html`. Accessed: 2020-05-28.

[64] Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak, and Tomasz Garbolino, editors. *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007), Kraków, Poland, April 11-13, 2007*. IEEE Computer Society, 2007.

[65] Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-Boolean solving. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1711–1718. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[66] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[67] Mika Göös, Sajin Koroth, Ian Mertz, and Toniann Pitassi. Automating cutting planes is NP-hard. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 68–77, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 680–695, Cham, 2014. Springer International Publishing.

[69] Liana Hadarean, Clark Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. Fine-grained SMT proofs for the theory of fixed-width bit-vectors. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '15)*, volume 9450 of *Lecture Notes in Computer Science*, pages 340–355. Springer, November 2015. Suva, Fiji.

[70] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[71] Trevor Alexander Hansen. *A Constraint Solver and its Application to Machine Code Test Generation*. PhD thesis, The University of Melbourne, 2012.

[72] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 283–290, 2008.

[73] Edward Hirsch, Dmitry Itsykson, Arist Kojevnikov, Alexander Kulikov, and Sergey Nikolenko. Report on the mixed boolean-algebraic solver. *Technical report, Laboratory of Mathematical Logic of St. Petersburg Department of Steklov Institute of Mathematics*, 2005.

[74] Eric Horvitz and Finn Verner Jensen, editors. *UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*. Morgan Kaufmann, 1996.

[75] Russell Impagliazzo, Pavel Pudlák, and Jiří Sgall. Lower bounds for the polynomial calculus and the Gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.

[76] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 8, USA, 2012. USENIX Association.

[77] Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.

[78] Priyank Kalla. Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation. In *Proceedings, Formal Methods in Computer-Aided Design, FMCAD*, page 2, Austin, TX, September 2015.

[79] Michael Katelman and José Meseguer. vlogsl: A strategy language for simulation-based verification of hardware. In Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing*, pages 129–145, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[80] Daniela Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra*. PhD thesis, Johannes Kepler University Linz, 2020.

[81] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In Clark Barrett and Jin Yang, editors, *Formal Methods in Computer-Aided Design, FMCAD 2019, San Jose, California USA, October 23-25, 2019.*, pages 28–36. IEEE, 2019.

[82] Daniela Kaufmann, Armin Biere, and Manuel Kauers. SAT, computer algebra, multipliers. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, volume 71 of *EPiC Series in Computing*, pages 1–18. EasyChair, 2020.

[83] Daniela Kaufmann, Manuel Kauers, Armin Biere, and David Cok. Arithmetic verification problems submitted to the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, page 49. University of Helsinki, 2019.

[84] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[85] Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, pages 292–298, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[86] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, 59(2):323–376, Aug 2016.

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

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

[89] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 7:59–6, 01 2010.

[90] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery.

[91] Harry R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317 – 353, 1980.

[92] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning sat solvers. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 225–241, Cham, 2015. Springer International Publishing.

[93] Vincent Liew. Ring Benchmarks. `https://doi.org/10.5281/zenodo.3987457`, August 2020.

[94] Rhishikesh Shrikant Limaye and Sanjit A. Seshia. Beaver: An SMT solver for quantifier-free bit-vector logic. Master's thesis, EECS Department, University of California, Berkeley, May 2010.

[95] László Lovász, Moni Naor, Ilan Newman, and Avi Wigderson. Search problems in the decision tree model. In *SIAM Journal on Discrete Mathematics*, volume 107, pages 119–132, 1995.

[96] J. Lv, P. Kalla, and F. Enescu. Efficient Grobner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(9):1409–1420, Sept 2013.

[97] A. Mahzoon, D. Große, and R. Drechsler. Polycleaner: Clean your polynomials before backward rewriting to verify million-gate multipliers. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2018.

[98] A. Mahzoon, D. Große, and R. Drechsler. Revsca: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2019.

[99] Filip Marić and Predrag Janičić. Urbiva: Uniform reduction to bit-vector arithmetic.

In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 346–352, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[100] J. P. Marques-Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, Nov 1996.

[101] João P. Marques-Silva, Ines Lynce, and Sharad Malik. CDCL solvers. In Biere et al. [17], chapter 4, pages 131–154.

[102] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[103] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 438–445, Cham, 2014. Springer International Publishing.

[104] Raphael Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. An SMT-based approach to automated configuration. In Pascal Fontaine and Amit Goel, editors, *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories*, volume 20 of *EPiC Series in Computing*, pages 109–119. EasyChair, 2013.

[105] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, June 2001.

[106] Alexander Nadel. Bit-vector rewriting with automatic rule generation. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 663–679, Cham, 2014. Springer International Publishing.

[107] Juan Antonio Navarro and Andrei Voronkov. Proof systems for effectively propositional logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, pages 426–440, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[108] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. In *Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2*, IAAI'06, pages 1720–1727. AAAI Press, 2006.

[109] Wolfgang Nebel and Ahmed Jerraya, editors. *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*. IEEE Computer Society, 2001.

[110] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with

serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[111] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[112] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[113] Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design*, 51(3):608–636, 2017.

[114] Aina Niemetz, Mathias Preiner, Armin Biere, and Andreas Fröhlich. Improving local search for bit-vector logics in SMT with path propagation. In *Proc. 4th Intl. Work. on Design and Implementation of Formal Tools and Systems (DIFTS'15)*, page 10 pages, 2015.

[115] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

[116] Jan Nordholz. Design of a symbolically executable embedded hypervisor. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[117] Openssl.org. Openssl bug cve-2016-7055, 2016.

[118] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. An efficient finite-domain constraint solver for circuits. In *Proceedings of the 41th Design Automation Conference, DAC*, pages 212–217, 2004.

[119] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. M. Greuel. STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[120] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[121] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[122] Stephen Ponzio. A lower bound for integer multiplication with read-once branching programs. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 130–139, Las Vegas, NV, May 1995.

[123] Nathaniel Popper. Knight capital says trading glitch cost it $440 million. `https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/`. Accessed: 2020-05-28.

[124] T. Pruss, P. Kalla, and F. Enescu. Equivalence verification of large Galois field arithmetic circuits using word-level abstraction via Grobner bases. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[125] Pavel Pudlák. Proofs as games. *The American Mathematical Monthly*, 107(6):541–550, 2000.

[126] Sherief Reda and A. Salem. Combinational equivalence checking using Boolean satisfiability and binary decision diagrams. In Nebel and Jerraya [109], pages 122–126.

[127] Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic LFSC. In *Proceedings of the 8th international workshop on satisfiability modulo theories, July 2010, Endinburgh, Scotland, (SMT 2010)*, 2010.

[128] Daniela Ritirc, Armin Biere, and Manuel Kauers. Column-wise verification of multipliers using computer algebra. In Daryl Stewart and Georg Weissenbacher, editors, *Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017.*, pages 23–30. IEEE, 2017.

[129] Daniela Ritirc, Armin Biere, and Manuel Kauers. Improving and extending the algebraic approach for verifying gate-level multiplies. In *Design, Automation and Test in Europe (DATE'18)*, 2018.

[130] Daniela Ritirc, Armin Biere, and Manuel Kauers. A practical polynomial calculus for arithmetic circuit verification. In Anna M. Bigatti and Martin Brain, editors, *3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2'18)*, pages 61–76. CEUR-WS, 2018.

[131] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[132] Anthony Romano and Dawson Engler. Expression reduction from programs in a symbolic binary executor. In Ezio Bartocci and C. R. Ramakrishnan, editors, *Model Checking Software*, pages 301–319, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[133] Masahiko SAKAI and Hidetomo NABESHIMA. Construction of an robdd for a pb-constraint in band form and related techniques for pb-solvers. *IEICE Transactions on Information and Systems*, E98.D(6):1121–1127, 2015.

[134] Martin Sauerhoff and Philipp Woelfel. Time-space tradeoff lower bounds for integer multiplication and graphs of arithmetic functions. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 186–195, San Diega, CA, June 2003.

[135] Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016*, pages 1048–1053, Dresden, Germany, March 2016.

[136] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):165–189, March 2006. Preliminary version in *DATE '05*.

[137] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Form. Methods Syst. Des.*, 42(1):91–118, February 2013.

[138] X. Sun, P. Kalla, T. Pruss, and F. Enescu. Formal verification of sequential Galois field arithmetic circuits using algebraic geometry. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1623–1628, March 2015.

[139] Robert Faturechi T. Christian Miller, Megan Rose and Agnes Chang. The navy installed touch-screen steering systems to save money. ten sailors paid with their lives. `https://features.propublica.org/navy-uss-mccain-crash/navy-installed-touch-screen-steering-ten-sailors-paid-with-their-lives`. Accessed: 2020-05-28.

[140] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, January 1987.

[141] Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, and Jakob Nordström. In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 292–310, Cham, 2018. Springer International Publishing.

[142] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi. Application of symbolic computer algebra to arithmetic circuit verification. In *2007 25th International Conference on Computer Design*, pages 25–32, Oct 2007.

[143] Oliver Wienand, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Gert-Martin Greuel. An algebraic approach for proving data correctness in arithmetic data paths. In *Computer Aided Verification*, pages 473–486, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[144] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, June 2015.

[145] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[146] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[147] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):2131–2142, Dec 2016.

[148] C. Yu, M. Ciesielski, and A. Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1907–1911, Sep. 2018.

[149] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.