Translating Pseudo-Boolean Proofs into Boolean Clausal Proofs

Karthik V. Nukala, Soumyaditya Choudhuri, Randal E. Bryant, Marijn J. H. Heule
Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: {kvn, soumyadc}@andrew.cmu.edu, {rebryant, marijn}@cmu.edu

***Abstract*—Clausal proofs, particularly those based on the deletion resolution asymmetric tautology (DRAT) proof system, are widely used by Boolean satisfiability solvers for expressing proofs of unsatisfiability. Their success stems from their simplicity and scalability. When solvers go beyond pure propositional reasoning, however, generating clausal proofs becomes more difficult. Solvers that employ pseudo-Boolean reasoning, including cutting-planes operations, can express proofs in the VeriPB proof system, but its adoption is not widespread.**

We introduce PBIP (Pseudo-Boolean Implication Proof), a framework that provides an intermediate representation between VeriPB and clausal proofs. We also introduce a toolchain comprising 1) a VeriPB-to-PBIP translator that performs proof trimming and optimization, and 2) a PBIP-to-LRAT translator that makes use of proof-generating operations on ordered binary decision diagrams (BDDs) to generate clausal proofs in LRAT format, a variant of the DRAT that allows efficient checking.

We demonstrate the viability of our approach, the effectiveness of our trimming, and the performance of our clausal proof generator on a set of native PB benchmarks and compare our approach to direct checking of VeriPB proofs.

I. Introduction

Boolean satisfiability (SAT) solvers underlie a large portion of automated reasoning tools such as theorem provers, satisfiability modulo theory (SMT) solvers, and model checkers. Given the safety-critical application domains of these tools, correctness of the underlying solver is of utmost importance. Creating a formally verified solver (using an interactive theorem prover, for example) would severely compromise the ability to optimize and rapidly evolve the program. Most satisfiability solvers take as input formulas expressed in conjunctive normal form (CNF). These formulas consist of a conjunction of *clauses*, each of which is a disjunction of *literals*, where each literal is a Boolean variable or its complement.

An alternative to a formally verified solver is to have the solver generate a proof certificate for each execution. When this certificate is successfully checked by a verified proof checker, the result is guaranteed to be correct. The deletion resolution asymmetric tautology (DRAT) proof system [1] has become the standard for modern SAT solvers and is widely used by entries in the annual SAT competition [2]. DRAT is notably a clausal proof system: a proof consists of a sequence of clauses where each clause preserves the satisfiability of the preceding clauses. A proof of unsatisfiability terminates with the addition of the empty clause. Clausal proofs are of particular interest because they are simple (resulting in successful efforts to write verified proof checkers in interactive theorem provers such as ACL2 [3], Coq [4], and CakeML [5]) as well as scalable (being able to check proofs two petabytes in size [6]).

We consider pseudo-Boolean (PB) reasoning, chosen for its status as a bridge between propositional satisfiability and higher-level "beyond Boolean" reasoning. Also known as 0/1 integer linear programming, PB reasoning has been a fertile area of research since the 1950s. It has been one of longstanding multidisciplinary interest, with problems in operations research [7], combinatorics [8], economics [9], and VLSI design [10] (among others) benefiting from expressive encodings as pseudo-Boolean constraints. By virtue of these encodings, PB solvers can exploit richer structure and reason in a way that would be difficult for native SAT solvers to do. Notable PB solvers include PBS [11], Galena [12], Pueblo [13], and RoundingSAT [14]. Having a way to express and check PB proofs of unsatisfiability would enhance the level of trust users could place in these solvers.

The VeriPB proof framework [15]–[17] supports both the cutting planes (CP) proof system, viewing pseudo-Boolean constraints as linear constraints over 0/1-valued variables, and implication-based reasoning, viewing the constraints as Boolean formulas. With cutting planes, new constraints can be generated by summing two constraints or by scaling a single constraint by either multiplication or division. With implication-based reasoning, a new constraint can be added when it is shown to be implied by previous constraints via reverse unit propagation (RUP). Although a RUP-based implication can be translated into a sequence of cutting planes steps, RUP more directly captures the logical inferences made by some tools.

This paper describes a series of tools that can transform a VeriPB proof into a clausal proof in extended resolution [18], a proof system that lies within the DRAT proof framework. The generated proof is expressed in LRAT format, a variant of DRAT for which a variety of proof checkers have been developed, including ones that have been formally verified.

The key to our method is to represent pseudo-Boolean constraints as ordered binary decision diagrams (BDDs) [19], and to use a proof-generating BDD package to generate clausal proof steps justifying each of its operations [20]. The BDD representation of a pseudo-Boolean constraint over n variables and with maximum coefficient a will have at most $a \cdot n$ nodes,

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_23

and so we can say that the generated clausal proofs will be of *pseudo-polynomial* complexity relative to the VeriPB proofs. That is, the proofs will be polynomial in the values of the coefficients, but this can be exponential in the number of bits required to represent these values. In practice, many PB proofs involve only small coefficients, and so the expansion will be polynomial.

Some contributions of this work include:

- The ability to translate proofs in the relatively new and unfamiliar VeriPB framework into a more established clausal framework.
- The ability to directly compare the sizes of proofs generated using different approaches to logical reasoning.
- Methods to optimize VeriPB proof sizes and checking times by adapting some of the trimming and hint generation methods used in clausal proofs.
- Experimental results relating the sizes of the clausal proofs generated by several tools (including ours) to PB proofs. These serve to quantify the advantage of proof frameworks based on this higher level of reasoning. We show that the clausal proofs scale polynomially, relative to VeriPB, but their larger sizes pose challenges for more difficult benchmark problems.

We note several limitations to our work:

- The soundness of our toolchain relies on a program that generates CNF representations of the constraints comprising the input pseudo-Boolean formula. Although the current program is simple and has been thoroughly tested, it would be preferable to have one that has been formally verified.
- Our toolchain does not support the full suite of VeriPB proof rules. Most significantly, it cannot handle the two strengthening rules that enable symmetry reductions in VeriPB proofs [21].

We discuss these in Section VII.

II. Related Work

Recently, the CakePB [22] proof checker has been developed to enable formally verified checking of VeriPB proofs. It operates by first converting the original VeriPB proof into one in the VeriPB kernel format, where application of the RUP proof rule is expanded into a sequence of cutting-planes steps [23]. CakePB has the advantage that it can reason about operations on pseudo-Boolean constraints directly, rather than on their BDD representations. We compare the performance of our toolchain to one based on CakePB in Section VI. Our results show the effectiveness of proof trimming and motivate its addition to future versions of CakePB.

At first glance, having a translation from cutting-planes proofs into clausal proofs that only achieves pseudo-polynomial performance (in terms of the proof size) could seem to fall short of the theoretical optimum. In particular, W. J. Cook, et al. [24] sketch an algorithm for converting any cutting planes proof of unsatisfiability for CNF formula F into an extended resolution proof, such the number of steps in the extended resolution proof would be bounded by a polynomial function $p(n, m)$, with n equal to the total number of literals in F and m equal to the number of steps in the CP proof. That result is not directly comparable to ours, however:

- It assumes that the problem consists entirely of PB constraints encoding clauses, i.e., having only unit coefficients and a unit constant.
- The scale of the polynomial is not given in the paper, but it appears to be large. The presented translation requires converting the CP steps into many arithmetic operations on an encoded representation of the coefficients similar to a binary representation. [25].
- To our knowledge, the proposed algorithm has never actually been implemented and doing so would require a substantial effort.

By contrast, VeriPB allows the input formula to contain constraints with coefficients of arbitrary size. In addition, even when given a formula with small coefficients, the constraints in the VeriPB proof can have coefficients of arbitrary size. Cook's method would require encoding these with low-level arithmetic operations. This theoretical result is unlikely to translate into a practical method for proof generation.

III. Preliminaries

A. Pseudo-Boolean Formulas

We recommend the PhD thesis by Stephen Gocht [17] as a helpful introduction to pseudo-Boolean reasoning. A pseudo-Boolean constraint is a linear expression, viewing Boolean variables as ranging over integer values 0 and 1. That is, a constraint c has the form $a_1\ell_1 + a_2\ell_2 + \cdots + a_n\ell_n \mathbin{\#} b$ where the coefficients a_i and the constant b are integers, and each *literal* ℓ_i equals either input variable x_i or its complement $\overline{x}_i$. For an *ordering* constraint, the relational operator $\#$ is $<$, $\leq$, $\geq$, or $>$. For an *equational* constraint, the relational operator is $=$. An equational constraint can also be represented as the conjunction of two ordering constraints having the same coefficients but one with relation $\leq$ and the other with $\geq$. We will mostly refer to *coefficient-normalized constraints* (CNCs) of the form

$$a_1\ell_1 + a_2\ell_2 + \cdots a_n\ell_n \quad \geq \quad b \tag{1}$$

where the coefficients and the constant are nonnegative integers and the relation is $\geq$.

An *assignment* ρ is a mapping from some subset of the variables in X to truth values 1 (true) and 0 (false). We can view an assignment as a set of literals $\rho = \{x_i \mid \rho(x_i) = 1\} \cup \{\overline{x}_i \mid \rho(x_i) = 0\}$. Assignment ρ is *total* when it assigns a value to every variable.

Constraint c denotes a Boolean function, written $[\![c]\!]$, mapping total assignments to truth values. Constraints c_1 and c_2 are said to be *equivalent* when $[\![c_1]\!] = [\![c_2]\!]$. Constraint c is said to be *infeasible* when $[\![c]\!] = \bot$, i.e., it always evaluates to 0. This occurs if and only if $\sum_{1 \leq i \leq n} a_i < b$. Constraint c is said to be *trivial* when $[\![c]\!] = \top$, i.e., it always evaluates to 1. This occurs if and only if $b = 0$.

As described in [17], the following are some properties of pseudo-Boolean constraints:

- A relational constraint with comparisons $<$, $\leq$, and $>$ can be converted to an equivalent CNC.
- An equational constraint can be converted into two CNCs.
- The logical negation of CNC c, written $\bar{c}$, can also be expressed as a CNC.
- Any coefficient a_i with $a_i > b$ in a CNC can be replaced with the coefficient b without changing the underlying Boolean function.

Some nomenclature regarding CNCs will prove useful. The *constraint literals* are those literals ℓ_i such that $a_i \neq 0$. A *cardinality constraint* has $a_i \in \{0, 1\}$ for $1 \leq i \leq n$. A cardinality constraint with $b = 1$ is referred to as a *clausal constraint*: at least one of the constraint literals must be assigned 1 to satisfy a constraint. It is logically equivalent to a clause in a conjunctive normal form (CNF) formula. A cardinality constraint with $b = \sum_{1 \leq i \leq n} a_i$ is referred to as a *conjunction*: all of the constraint literals must be assigned 1 to satisfy the constraint. A conjunction for which $a_i = 1$ for just a single value of i is referred to as a *unit* constraint: it is satisfied if and only if literal ℓ_i is assigned 1.

A pseudo-Boolean *formula* F is a set of pseudo-Boolean constraints. We say that F is *satisfiable* when there is some assignment ρ that satisfies all of the constraints in F, and *unsatisfiable* otherwise.

Although feasibility can readily be tested for individual CNCs, determining whether a set of constraints (even for set size 2) is satisfiable is intractable, unless $P = NP$. For example, the subset sum problem [26] can readily be translated into an equational constraint, and this can then be expressed as the conjunction of two CNCs.

Pseudo-Boolean optimization problems can be converted to decision problem by imposing a bound on the metric being optimized. For example, two runs of a PB solver suffice to prove that a graph has maximum clique size k. First, the solver is run with a cardinality constraint requiring clique size k. The generated solution can then be checked to make sure it is indeed a clique. Then the solver is run with proof generation enabled and with a cardinality constraint requiring clique size $k + 1$. The certificate of unsatisfiability completes the proof. Similar approaches can be used for other optimization problems [15].

B. (Reverse) Unit Propagation

We let $c|_\rho$ denote the CNC resulting when c is simplified according to partial assignment ρ. That is, assume c has the form of (1) and partition the indices i for $1 \leq i \leq n$ into three sets: I^+, consisting of those indices i such that $\ell_i \in \rho$, I^-, consisting of those indices i such that $\overline{\ell}_i \in \rho$, and I^X consisting of those indices i such that neither ℓ_i nor $\overline{\ell}_i$ is in ρ. With this, $c|_\rho$ can be written as $\sum_{1 \leq i \leq n} a'_i \geq b'$ with a'_i equal to a_i for $i \in I^X$ and equal to 0 otherwise, and with $b' = b - \sum_{i \in I^+} a_i$.

Literal ℓ_i is *unit propagated* by CNC c when the assignment $\rho = \{\overline{\ell}_i\}$ causes the constraint $c|_\rho$ to become infeasible. As the name implies, a unit-propagated literal ℓ_i then becomes a unit constraint. Observe that a single constraint can unit propagate multiple literals. For example, $4x_1 + 3\overline{x}_2 + x_3 \geq 6$ unit propagates both x_1 and $\overline{x}_2$. For CNC c, we let $\mathit{Unit}(c)$ denote the set of literals it unit propagates

Rather than simplifying a constraint c according to partial assignment ρ and then detecting unit propagations, we can combine these to detect the set of unit propagations for a constraint with respect to a partial assignment. That is, we define $\mathit{Unit}_\rho(c)$ to be $\mathit{Unit}(c|_\rho)$. These propagations can readily be detected by computing the *slack*, defined as $\mathit{Slack}_\rho(c) = \sum_{i \in I^X} a_i + \sum_{i \in I^+} a_i - b$, where I^X and I^+ are the sets of indices defined previously. $\mathit{Unit}_\rho(c)$ is then defined as $\{\ell_i \mid a_i > \mathit{Slack}_\rho(c)\}$. For example, the constraint $c \doteq 4x_1 + 3\overline{x}_2 + x_3 \geq 6$ has slack $4+3+1-6 = 2$ with respect to $\rho = \emptyset$. We can therefore compute $\mathit{Unit}_\rho(c) = \{x_1, \overline{x}_2\}$. Furthermore c will be infeasible for partial assignment ρ when $\mathit{Slack}_\rho(c) < 0$.

Given a set of constraints F, we can build up a partial assignment ρ by repeatedly performing unit propagation. That is, define the operation *Uprop* as $\mathit{Uprop}(\rho, c) = \rho \cup \mathit{Unit}_\rho(c)$. For initial assignment ρ, *unit propagation* on formula F is then the process of extending ρ by repeatedly computing $\rho \leftarrow \mathit{Uprop}(\rho, c)$ to all of the constraints $c \in F$ until no more propagations are possible.

Consider a formula F consisting a set of constraints $c_1, c_2, \ldots, c_m$. The *reverse unit propagation* (RUP) proof rule [15], [17] uses unit propagation to prove that *target constraint* c can be added to a formula while preserving its set of satisfying assignments. That is, any assignment that satisfies F also satisfies $F \wedge c$. A RUP addition justifies c by assuming $\bar{c}$ holds and showing, via a sequence of *RUP steps*, that this leads to a contradiction. It accumulates a partial assignment ρ based on unit propagations starting with the empty set. Each RUP step accumulates more assigned literals by performing a unit propagation of the form $\rho \leftarrow \mathit{Uprop}(\rho, d)$, where d is either c_j, a prior constraint, or $\bar{c}$, the negation of the target constraint. The final step causes a contradiction, where $d|_\rho$ is infeasible. Unlike with clauses, a single constraint, including the negated target, can be used for unit propagation on multiple RUP steps within a single RUP addition.

C. Trusted Binary Decision Diagrams

Trusted binary decision diagrams (TBDDs) [20] provide a method for generating clausal proofs when performing sequences of operations on Boolean functions represented as ordered binary decision diagrams (BDDs) [19]. TBDDs have been used to generate proofs of unsatisfiability for SAT solvers [27], proofs of satisfaction and refutation in QBF solvers [28], and for proofs of unsatisfiability for pseudo-Boolean constraints [29]. Proofs are generated directly in the LRAT format, making use of the support for extended resolution provided by the RAT proof system.

In the following, we write a clause consisting of literals $\ell_1, \ell_2, \ldots, \ell_k$ as $[\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k]$. A unit clause with literal ℓ is written as $[\ell]$.

The key idea is to introduce an extension variable u every time a BDD node $\mathbf{u}$ is created, with proof clauses defining the semantic relation between $\mathbf{u}$, the node variable x, and child nodes $\mathbf{u}_1$ and $\mathbf{u}_0$ [27], [30], [31]. Each step in the recursive algorithms to generate new BDDs generates a sequence of proof clauses justifying an inductive invariant about the operation being performed. For example, suppose the Apply algorithm [19] computes the conjunction of BDDs with root nodes $\mathbf{u}$ and $\mathbf{v}$ to derive a BDD with root node $\mathbf{w}$. Each recursive step of the operation performs the conjunction of argument nodes $\mathbf{u}'$ and $\mathbf{v}'$ to derive a node $\mathbf{w}'$. With TBDDs, this step also generates a sequence of proof steps concluding with the addition of clause $[\overline{u}' \vee \overline{v}' \vee w']$, justifying that $u' \wedge v' \Rightarrow w'$. The final step of the recursion then generates the clause justifying $u \wedge v \Rightarrow w$.

A *trusted* BDD $\dot{\mathbf{u}}$ is a BDD having root node $\mathbf{u}$ for which the unit clause $[u]$ has been added to the proof. That is, the BDD will evaluate to 1 for any assignment that satisfies the input formula. A proof of unsatisfiability concludes with the addition of the TBDD consisting of the leaf node L_0, representing $\bot$. This is encoded in the proof by the empty clause.

D. Cutting Planes

The cutting planes proof system defines rules to derive new constraints from existing ones, as is shown in Figure 1.

$$\text{DIV}\ \frac{\sum_i a_i x_i \geq b}{\sum_i \frac{a_i}{k} x_i \geq \left\lceil \frac{b}{k} \right\rceil} \qquad \text{SAT}\ \frac{\sum_i a_i x_i \geq b}{\sum_i \min(a_i, b) x_i \geq b}$$

$$\text{MUL}\ \frac{\sum_i a_i x_i \geq b}{\sum_i k a_i x_i \geq k b} \qquad \text{ADD}\ \frac{\sum_i a_i x_i \geq b \qquad \sum_i c_i x_i \geq d}{\sum_i (a_i + c_i) x_i \geq b + d}$$

Fig. 1. Cutting-Planes Proof Rules. For the division rule, each coefficient a_i must be divisible by k.

Notably, rules DIV, SAT, and MUL do not change the underlying Boolean function of the constraint. That is, for any constraint c, and $k \in \mathbb{N}^+$ (where each coefficient a_i in the division rule must be divisible by k):

$$[\![c]\!] = [\![\mathsf{DIV}(c, k)]\!] = [\![\mathsf{SAT}(c)]\!] = [\![\mathsf{MUL}(c, k)]\!]$$

Generating a constraint via the ADD rule, on the other hand, creates a constraint with a new underlying Boolean function, but it is implied by the conjunction of the Boolean functions for the arguments:

$$[\![c_1]\!] \wedge [\![c_2]\!] \Rightarrow [\![c_1 + c_2]\!]$$

IV. PBIP: Pseudo-Boolean Implication Proof

A Pseudo-Boolean Implication Proof (PBIP) provides a systematic way to prove that a PB formula F is unsatisfiable. The PBIP file format is described in Section IV-B. It is not intended to be a useful format on its own, but rather a bridge between a PB proof and its translations into a clausal proof.

A. PBIP Proof Structure

A PBIP proof is given as a sequence of constraints, referred to as the *proof sequence*:

$$c_1, c_2, \ldots, c_m, c_{m+1}, \ldots, c_t$$

such that the first m constraints are those of formula F, while each *added* constraint c_i (referred to as a *lemma*) for $i > m$ follows by implication from the preceding constraints. That is,

$$\bigwedge_{1 \leq j < i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \tag{2}$$

The proof completes with the addition of an infeasible constraint for c_t. By the transitivity of implication, we have therefore proved that F is not satisfiable.

Constraints c_i with $i > m$, can be added in two different ways, corresponding to two different reasoning modes.

1) In *implication mode*, constraint c_i follows by implication from at most two prior constraints in the proof sequence. That is, some $H_i \subseteq \{c_1, c_2, \ldots, c_{i-1}\}$ with $|H_i| \leq 2$ satisfies
$$\bigwedge_{c_j \in H_i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \tag{3}$$
Set H_i is referred to as the *hint* for proof step i.
To simplify the generation of PBIP proofs, the checker supports a *summation* rule of the form $\sum_{1 \leq i \leq k} d_i \Rightarrow c$, where each d_i is a constraint from a previous step. Checking this is performed by computing intermediate constraints as pairwise sums and proving that they satisfy implication.
2) In *RUP mode*, constraint c_i is justified by RUP addition. The hint specifies the RUP steps as a sequence $[d_1, m_1], [d_2, m_2], \ldots, [d_{k-1}, m_{k-1}], [d_k]$. Each d_j indicates either a previous constraint $c_{i'}$ for $i' < i$, or the negated target constraint $\overline{c}_i$. Each m_j indicates a unit-propagated literal. The final constraint, indicated by d_k should conflict with the accumulated assignment $\rho = \{m_1, m_2, \ldots, m_{k-1}\}$.

Unless $P = NP$, we cannot guarantee that a proof checker can validate even a single implication step of a PBIP proof in polynomial time. In particular, consider an equational constraint c encoding an instance of the subset sum problem, and let $c_{\leq}$ and $c_{\geq}$ denote its conversion into a pair of ordering constraints such that $[\![c]\!] = [\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!]$. Consider a PBIP proof to add the constraint $\overline{c}_{\leq}$ having the $c_{\geq}$ as the only hint. Proving that $[\![c_{\geq}]\!] \Rightarrow [\![\overline{c}_{\leq}]\!]$, requires proving that $[\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!] = \bot$, i.e., that c is unsatisfiable.

On the other hand, checking the correctness of a PBIP proof can be performed in *pseudo-polynomial* time using

BDDs, meaning that the complexity will be bounded by a polynomially sized formula over the numeric values of the integer parameters. In particular, a CNC over n variables and having a as the maximum of its coefficients a_i and the constant b will have a BDD representation with at most $a \cdot n$ nodes [32]. For an implication proof step where the added constraints and the hints all have coefficients and constants less than or equal to a, the number of BDD operations to validate the step will be $O(a^2 \cdot n)$ when there is a single hint and $O(a^3 \cdot n)$ when there are two hints. This complexity is polynomial in a, but it could be exponential in the size of a binary representation of a. The number of BDD operations for each unit propagation step in a RUP proof will be linear in the size of the BDD and therefore $O(a \cdot n)$.

B. PBIP File Format

A PBIP file describes a sequence of transformations on a set of pseudo-Boolean input constraints leading to an infeasible constraint. The file therefore describes an unsatisfiability proof for a PB constraint problem. The format assumes that each input constraint is encoded as a set of clauses in conjunctive normal form (CNF). The clauses for all of the constraints are provided as a file in the standard DIMACS format. The generation of this file is performed by a separate program PB-CNF, described in Section VI.

When in implication mode, each derived constraint must follow by implication from either one or two preceding constraints, referred to as the "antecedents". That is, for target constraint c and prior constraint c_1, and possibly c_2, we must have either $[\![c_1]\!] \Rightarrow [\![c]\!]$ or $[\![c_1]\!] \wedge [\![c_2]\!] \Rightarrow [\![c]\!]$. When in RUP mode, the constraint to be derived is set as a target, and a set of unit constraints is accumulated. Each RUP step then derives an additional unit constraint based on the previously derived unit constraints, as well as either the complement of the target constraint or some preceding constraint. The final step should then cause a contradiction, with the set of accumulated unit constraints falsifying the final constraint.

PBIP files build on the OPB format for describing PB constraints, as documented in [33].

There are five line types. The first four types define constraints that can be referenced by later lines. Constraints are numbered from 1, starting with the input constraints. File lines beginning with "`*`" are treated as comments.

1) Input lines begin with "`i`". This is followed by a constraint, expressed in OPB format, and terminated by "`;`". Then, a sequence of clause numbers from the CNF file is listed, separated by spaces and terminated with end-of-line. Forming the conjunction of these clauses and existentially quantifying any variables that are not listed in the PB formula should yield a Boolean function that is implied by the PB constraint.
2) Implication-mode assertion lines begin with "`a`". This is followed by a constraint, expressed in OPB format and terminated by "`;`". Then, either one or two constraint numbers is listed, separated by spaces and terminated with end-of-line.
3) RUP lines begin with "`u`". This is followed by a constraint, expressed in OPB format and terminated by "`;`". Then, a sequence of lists is given, where each list is of the form $[I\,\ell_1\,\ldots\,\ell_k]$, indicating that constraint number I will propagate additional units $\ell_1, \ldots, \ell_k$. I can either be the number of a previous constraint, or it can be that of the current constraint. The latter case is known as a "self reference", and its unit propagations should be based on the negation of the target constraint. The final list is of the form $[I]$, and the indicated constraint must be falsified by the accumulated set of literals. A list of the form $[I\,\ell_1\,\ldots\,\ell_k]$ with $k > 1$ indicates that multiple literals will be unit propagated. This notation is equivalent to listing the literals individually with the sequence $[I\,\ell_1]\,[I\,\ell_2]\cdots[I\,\ell_k]$.
4) Summation implication lines begin with "`s`". This is followed by a constraint c, expressed in OPB format and terminated by "`;`". Then, a set of constraint numbers is listed, separated by spaces and terminated with end-of-line. These numbers identify a set of prior constraints $c_1, c_2, \ldots, c_k$ satisfying:
$$\sum_{i=1}^{k} [\![c_i]\!] \quad \Longrightarrow \quad [\![c]\!]$$
This line avoids the need to expand a summation of k constraints into $k-1$ implication lines. Instead, the checker performs the summations and tests the final implication, using heuristics to optimize the order in which the argument constraints are summed.
5) Deletion lines begin with "`d`". This is followed by a list of constraint numbers. These constraints cannot be used as hints for the remainder of the proof.

For an unsatisfiability proof, the final constraint should be infeasible, e.g. $0 \geq 1$.

V. PBIP Trimming and Checking

Given a VeriPB proof, we must transform it into a PBIP proof. In doing so, we trim the proof to eliminate those steps that are not required for the final unsatisfiability result.

A. Hinting and Trimming Cutting-Planes and RUP Proofs

We present here a set of procedures that take a VeriPB proof of unsatisfiability (generated by a proof-logging solver such as RoundingSAT [34] or the Glasgow Subgraph/Clique Solvers [15], [35]) and perform a translation into PBIP. In addition, the resulting proof is *trimmed*, removing proof steps that do not lead to the final unsatisfiability result.

Our procedures support the following VeriPB commands:

- `f`/`l` - OPB input constraint loading
- `p`/`pol` - Justification via reverse Polish notation (RPN) arithmetic/cutting planes reasoning
- `u`/`rup` - Justification via reverse unit propagation (RUP)
- `o`/`soli` - Optimal value witness

In addition, we also support auxiliary VeriPB commands such as `a` and `j` (variants of implication that show up in proofs generated by the Glasgow solvers).

Our procedure performs a backwards reachability analysis similar to DRAT-trim [1]: first, we identify a minimal set of constraints required to justify the empty constraint $\bot = 0 \geq 1$, expressing unsatisfiability of the formula. We continue with our reachability analysis from this minimal set. For each lemma in the minimal set, we identify the lemmas needed to prove it, mark them as necessary for the proof, and add them to the set. All unmarked lemmas are discarded (trimmed).

The hinting/trimming algorithms can be partitioned into two sub-procedures: (1) RUP lemma hinting and (2) arithmetic/cutting planes reasoning.

1) Lemma Justification via RUP: The RUP procedure serves to simultaneously

- trim the formula by computing a minimal *necessary* set of constraints S' required to justify all lemmas in S.
- construct a set of hints mapping each constraint $c_i \in S'$ to a list of hints H_i, where each list element h_j is of the form $[d_j \ m_j]$, indicating that constraint d_j propagates unit m_j to falsify $\overline{c}_i$.

We make two notable optimizations here:

1) We implement a saturation procedure that aims to minimize the necessary set at each step—that is, we only consider new lemmas if we cannot justify our target with the current set.
2) Our data structures support near-instant unit discovery by utilizing properties of a constraint's slack: by maintaining a sorted order over our constraint database (terms sorted in decreasing-coefficient order, constraints sorted in decreasing-slack order), unit discovery amounts to queries to the first element under this ordering, which can be done efficiently.

2) Arithmetic/Cutting Planes Reasoning: The goal of the arithmetic procedures is to unroll sequences of reverse Polish notation (RPN) arithmetic into hinted chains of clausal reasoning. Notable optimizations include:

- General trie-based arithmetic simplification: we avoid recomputation of cutting-planes sequences that share common prefixes by maintaining a trie.
- Heuristic arithmetic trimming: the solvers we have tested tend to build up chains of cutting-planes reasoning where only the last element of the chain is useful for the final result, and hence the rest of the chain is unnecessary. We proceed with this assumption, but in event of propagation failure, we revert to a more cautious step-by-step processing.

B. BDD-Based PBIP Checking and LRAT Generation

Our goal is to create a TBDD representation $\dot{\mathbf{u}}_i$ for each constraint c_i in the proof sequence. Our implementation augments the existing TBDD operations in the TBUDDY package [20] to support PB constraints and to provide special operations to support RUP proof justification. The final step of adding infeasible constraint c_t will cause the empty clause to be added to the proof. Here we provide a high-level description of how the different PBIP steps translate into TBDD operations.

When adding constraint c_i, we invoke operation $\mathsf{BDD}(c_i)$ to construct the BDD representation $\mathbf{u}_i$ of c_i according to the algorithm described by Abío, et al. [32]. Upgrading this to the trusted BDD $\dot{\mathbf{u}}_i$ requires generating the unit clause $[u_i]$. We assume that every prior proof constraint $c_{i'}$, with $i' < i$, has a TBDD representation $\dot{\mathbf{u}}_{i'}$ with an associated unit clause $[u_{i'}]$.

When c_i is added by implication mode, generating its unit clause is based on the constraints given as the hint. If the hint consists of the single constraint $c_{i'}$, we can use the BDD_IMPLY operation to add proof clause $[\overline{u}_{i'} \vee u_i]$. Resolving this with the unit clause $[u_{i'}]$ then gives the unit clause $[u_i]$. When the hint consists of two constraints $c_{i'}$ and $c_{i''}$, we first use the BDD_AND operation on BDDs $\mathbf{u}_{i'}$ and $\mathbf{u}_{i''}$ to generate their conjunction $\mathbf{w}$, along with proof clause $[\overline{u}_{i'} \vee \overline{u}_{i''} \vee w]$. We then use the BDD_IMPLY operation to generate the clause $[\overline{w} \vee u_i]$. Resolving these clauses with the unit clauses for TBDDs $\dot{\mathbf{u}}_{i'}$ and $\dot{\mathbf{u}}_{i''}$ yields the unit clause $[u_i]$.

Adding constraint c_i via a RUP addition involves two phases. The first performs a series of clause generations to justify the unit propagations. The second uses a single clausal RUP addition to add the target clause. During the first phase, each step $j < k$ in the sequence $[d_1, m_1], [d_2, m_2], \ldots, [d_{k-1}, m_{k-1}], [d_k]$, requires generating a clause of the form $[u_i \vee \overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j]$. Step k requires generating the clause $[u_i \vee \overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{k-1}]$. Implementing these justifications is complicated by the negations in the RUP steps, since negation is not directly supported in clausal reasoning. Instead, we make extensive use of DeMorgan's Laws. The final clausal RUP addition has unit clause $[u_i]$ as its target and will have as hints the unit-propagating clauses generated for the RUP steps. RUP addition will start with unit literal $\overline{u}_i$ and accumulate the propagated literals $m_1, m_2, \ldots, m_{k-1}$. The final clause will cause a conflict. For the special cases where either the previous constraint $c_{i'}$ or the target constraint c_i can be represented as a single clause, we can use this clause directly to justify unit propagation, reducing the number of BDD operations.

VI. Implementation and Results

The overall toolchain, illustrated in Figure 2, consists of the following steps

- IPBIP-HINTS: Translates from VeriPB to PBIP while simultaneously trimming the VeriPB proof, as described in Section V-A.
- PB-CNF: Generates a CNF representation of the input constraints by first constructing their BDD representations [32], and then encoding these with clauses, using at most two clauses per BDD node.
- PBIP-CHECK: Generates an LRAT file from the PBIP proof as described in Section V-B
- LRAT-CHECK: Checks an LRAT proof

As the thick lines in the figure indicate, steps PB-CNF and PBIP-CHECK can cause an exponential growth in the proof size, when input or intermediate constraints have large

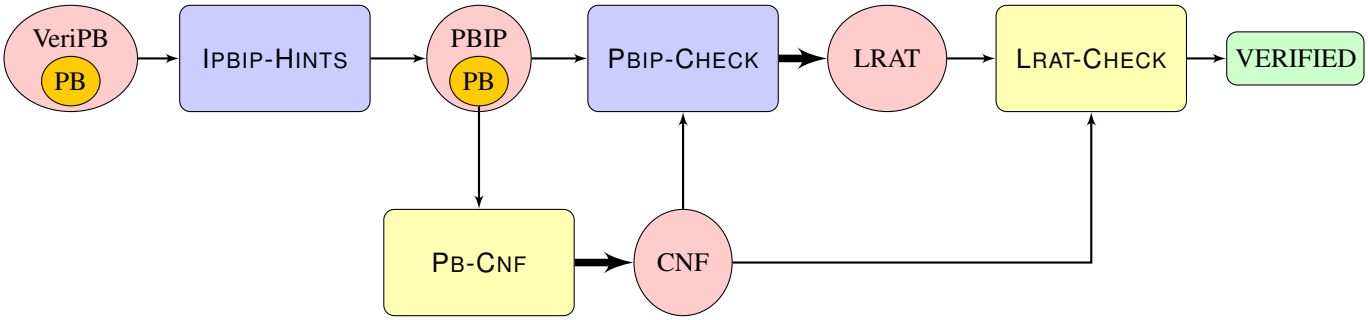

Fig. 2. PBIP toolchain: A VeriPB proof is first trimmed and translated into PBIP by IPBIP-HINTS. The VeriPB and the PBIP proofs contain a copy of the original PB problem, denoted by the orange PB subnodes. PB-CNF generates a CNF file from the original PB problem. The PBIP proof and CNF file are inputs to PBIP-CHECK, which translates the PBIP proof to LRAT using a a proof-generating BDD package [20]. The generated LRAT file can then be checked by LRAT-CHECK. The two thick lines indicate cases where there can be an exponential size increase. The yellow blocks indicate steps that must be correct to ensure soundness of the toolchain.

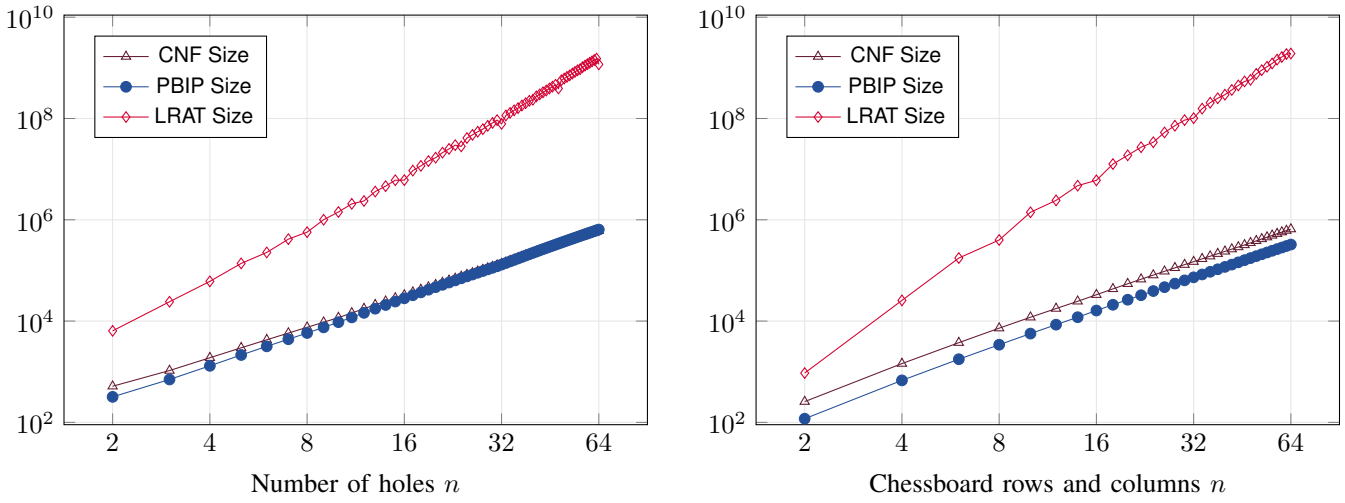

Fig. 3. File sizes in bytes for (left) pigeonhole and (right) mutilated chessboard

coefficients. The figure also indicates that steps PB-CNF and LRAT-CHECK form the trusted code base for the toolchain—they must be correct for the overall verification to be sound. In the case of LRAT-CHECK, one option would be to use a formally verified checker. In the case of PB-CNF, the tool is very simple, but it would be good to have a formally verified version, as is discussed in Section VII.

A. Benchmarks

We demonstrate the effectiveness of our tools (trimming procedure, clausal translation) and analyze our contributions by evaluating them on the following benchmark problems:

1) Pigeonhole (PHP) Formulas - PBIP proofs generated by summing the constraints across all pigeons and holes.
2) Mutilated Chessboard (MCB) Formulas - PBIP proofs generated by summing the constraints for every square.
3) DIMACS Clique (CLQ) Benchmarks (23 instances) - OPB/VeriPB proofs generated by Glasgow Clique Solver
4) Subgraph Isomorphism (SIP) Benchmarks (30 instances) - OPB/VeriPB proofs generated by Glasgow Subgraph Solver

For the CLQ benchmarks, the CakePB checker [22] was evaluated on a 55-benchmark subset of the Second DIMACS Implementation Challenge [36], out of which it was able to verify 50 graphs. From this subset of 55 benchmarks, we were able to translate 23 instances to LRAT and fully verify 21 of them with LRAT-CHECK.

For the SIP benchmarks, we selected 30 instances from subgraph isomorphism problems hosted by Christien Solnon [37] and translate all of them down to LRAT and verify them with LRAT-CHECK.

We ran our tests on the Jetstream2 cluster hosting a system with a 16-core AMD EPYC-Milan Processor, 60GB RAM, and 1TB disk space running Ubuntu 22.04.3 LTS.

Our results can be broken down into three sets of experiments: (1) a comparison between PBIP/LRAT proof sizes and how our pipeline performs against competing tools, (2) an evaluation of the effectiveness of our trimming procedure, and (3) an analysis of the runtimes of our toolchain.

B. Proof Sizes

Pigeonhole (PHP)/Mutilated Chessboard (MCB) Benchmarks: For the pigeonhole and mutilated chessboard prob-

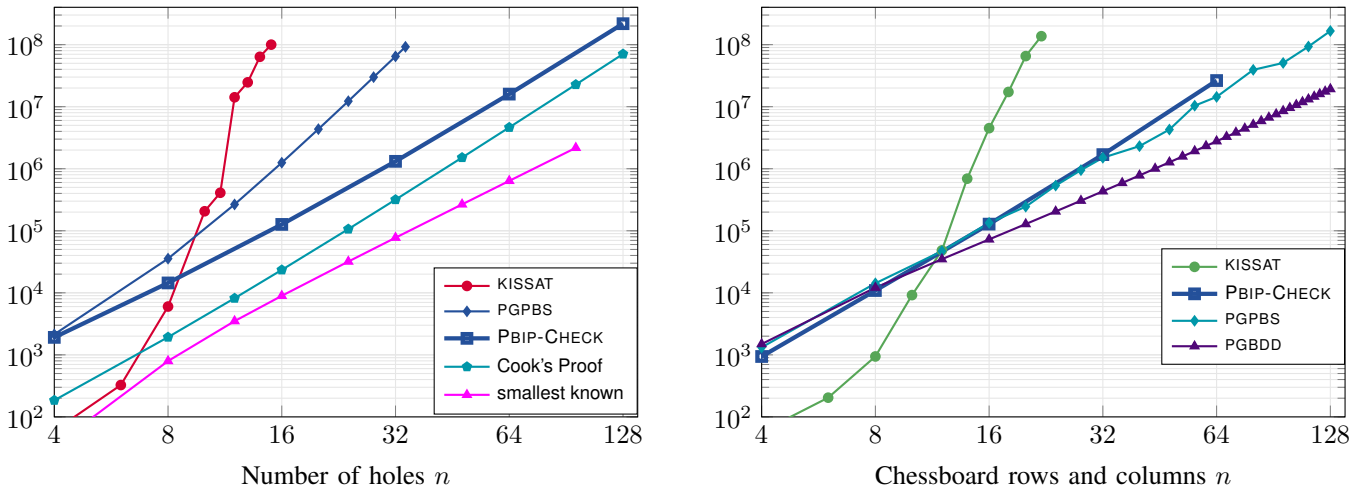

Fig. 4. Total number of clauses in proofs for (left) pigeonhole and (right) mutilated chessboard

lems, Figure 3 shows the different file sizes (in bytes) as a function of problem parameter n (the number of holes in PHP and the number of rows and columns in MCB). The graphs show a close correspondence between the CNF and the PBIP proof sizes, and a polynomial separation between the PBIP and LRAT proof sizes.

Figure 4 shows the number of clauses in proofs generated by our toolchain compared to those for proofs generated by competing tools. We consider here the proof-generating SAT solver KISSAT [38], the proof-generating pseudo-Boolean solver PGPBS [29], S. A. Cook's manually constructed $O(n^4)$ extended resolution proofs [39], and the smallest known proof ($O(n^3)$) [40]. We see that our PHP proofs asymptotically match the $O(n^4)$ scaling of Cook's proof. The scaling of the MCB proofs matches that of PGPBS but is bested by running the proof-generating BDD package PGBDD with a carefully devised variable ordering and sequencing of BDD operations [41]. In both instances, we greatly improve on the exponential performance of KISSAT.

Maximum Clique/Subgraph Isomorphism Benchmarks: We evaluate our pipeline on native pseudo-Boolean benchmarks (CLQ/SIP) starting from VeriPB cutting planes proofs. From the VeriPB proofs, we run the full pipeline presented in Figure 2 and obtain PBIP and LRAT proofs. Figure 5 compares the various proof sizes.

The general trend confirms the polynomial separation between the PBIP and LRAT proof sizes, with some irregularities due to extreme cases of proof trimming.

Figure 6 summarizes the average proof size increase (per benchmark suite) across the various proof formats in relation to the original VeriPB proof (in its non-kernel format).

C. Trimming Effectiveness

Here, we outline the effectiveness of the VeriPB trimming procedures described in Section V-A and implemented in IPBIP-HINTS.

Clique Benchmarks: Figure 7 demonstrates the effectiveness of our trimming on a set of DIMACS clique problems. On average, over our test suite, 45% of the input VeriPB lemmas are deemed unnecessary and are therefore trimmed from the proof. On 3 examples (`hamming8-2`, `c-fat500-10`, and `hamming10-2`), our trimmed (clausal) PBIP proofs are in fact shorter than the corresponding (non-clausal) VeriPB proofs. Notably, these cases are trimmed very aggressively (averaging 97.5% of lemmas trimmed) and this can be seen in Figure 5, where the VeriPB line rises above the PBIP line. The resulting LRAT proof for `hamming10-2` is also only 3.7 times larger than the corresponding VeriPB proof.

Subgraph Isomorphism Benchmarks: The proofs generated by the Glasgow Subgraph Solver [35] for the (unsatisfiable) subgraph isomorphism problems were succinct—most required only a single RUP justification amounting to the final unsatisfiability result. However, four benchmarks required more than one lemma. Notably, `g3-g12` underwent 91% trimming, with the corresponding LRAT file being comparable in size with the corresponding VeriPB file (as depicted in Figure 5, where the file sizes only differ by a factor of 1.3×).

D. Tool Runtimes

Clique Benchmarks: Our approach incurs a significant cost (in comparison with CakePB) in both the source trimming and the clausal translation, as shown in Figure 8. On easy instances, the trimming (IPBIP-HINTS, in blue) and checking (PBIP-CHECK in red and LRAT-CHECK in yellow) all perform moderately well whereas on hard instances, they become quite slow. Average ratios (in relation to CakePB's performance) over the benchmark sets are seen in Figure 9.

Subgraph Isomorphism Benchmarks: Similar to the clique benchmarks (as seen in Figure 8), our toolchain on subgraph isomorphism problems generally incurs a large runtime overhead versus CakePB. However, the trimming procedure does take less time than CakePB, amounting to 80% of CakePB's total runtime. This can be attributed to the succinctness of the

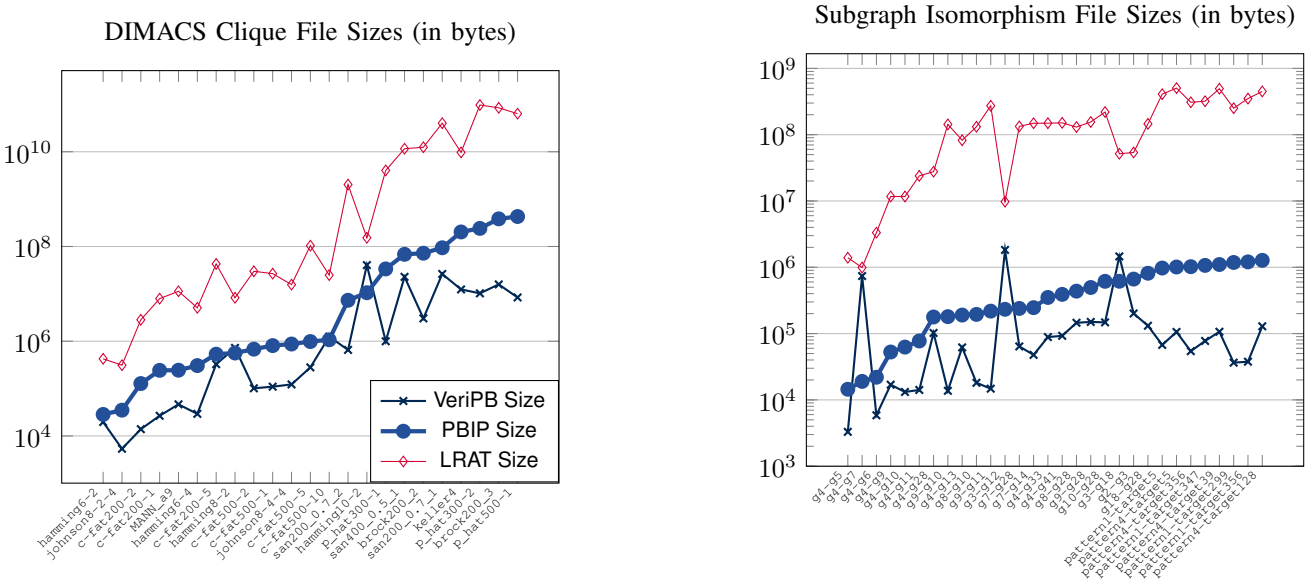

Fig. 5. (L) DIMACS MAX-Clique proof file sizes in bytes (R) Subgraph Isomorphism proof file sizes in bytes

Benchmark	VeriPB Kernel	PBIP	LRAT
CLQ	1.9×	11.3×	1682.0×
SIP	2.8×	8.1×	3342.8×

Fig. 6. Average proof size increase between VeriPB (non-kernel) and Kernel/PBIP/LRAT

generated VeriPB proofs by the subgraph solver, requiring less effort from our trimming/propagation procedures.

VII. Conclusion and Future Work

We have presented a pipeline capable of translating native pseudo-Boolean proofs (in the VeriPB format) to extended resolution proofs (in the LRAT format). This involved introducing the intermediate PBIP (Pseudo-Boolean Implication Proof) framework, from which a proof-generating BDD package can generate the LRAT proof.

The work reported here suggests several avenues for future research.

Verified PB Encodings. As indicated in Figure 2, we use the unverified program PB-CNF to generate a CNF representation of the input constraints. Although generating a clausal representation of pseudo-Boolean constraints is straightforward, this still represents a weak link in terms of the trustworthiness of our toolchain. Based on recent work on formalized CNF encodings in the Lean proof framework [42], we could produce verified CNF encodings of PB constraints expressed in the OPB format and achieve end-to-end verification (verified encodings of the PB source, verified translation by PBIP-CHECK, and verified checking via LRAT-CHECK) of our pipeline.

Supporting a Larger Subset of VeriPB. VeriPB is capable of even richer modes of reasoning, supporting rules such as redundancy-based strengthening and dominance-based strengthening [21]. Converting these to BDD-based proofs

benchmark	total u	done u	trimmed (%)
brock200_2	3758	3388	9.85
brock200_3	14251	14210	0.29
c-fat200-1	17	6	64.71
c-fat200-2	3	1	66.67
c-fat200-5	86	29	66.28
c-fat500-1	9	2	77.78
c-fat500-2	15	2	86.67
c-fat500-5	34	2	94.12
c-fat500-10	65	1	98.46
hamming6-2	17	1	94.12
hamming6-4	82	82	0.0
hamming8-2	65	2	96.92
hamming10-2	257	3	98.83
johnson8-2-4	24	24	0.0
johnson8-4-4	120	115	4.17
keller4	13542	13495	0.35
MANN_a9	71	53	25.35
p_hat300-1	1473	1434	2.65
p_hat300-2	4078	3367	17.44
p_hat500-1	9708	9677	0.32
san200_0.7_1	13396	2604	80.56
san200_0.7_2	450	246	45.33
san400_0.5_1	2276	1554	31.72
g2-g3	701	701	0.0
g3-g12	411	37	91.0
g3-g18	321	69	78.5
g4-g7	21	20	4.76

Fig. 7. VeriPB lemmas trimmed on CLQ/SIP benchmarks - The column marked "total u" represents the number of lemmas present in the source VeriPB proof (RUP lemmas logged by the Glasgow solvers in their derivations of ⊥) and "done u" represents the number of RUP lemmas *actually* deemed necessary by our IPBIP-HINTS trimming procedure. The top 23 benchmarks are max-clique benchmarks while the bottom 4 are (selected) subgraph isomorphism benchmarks.

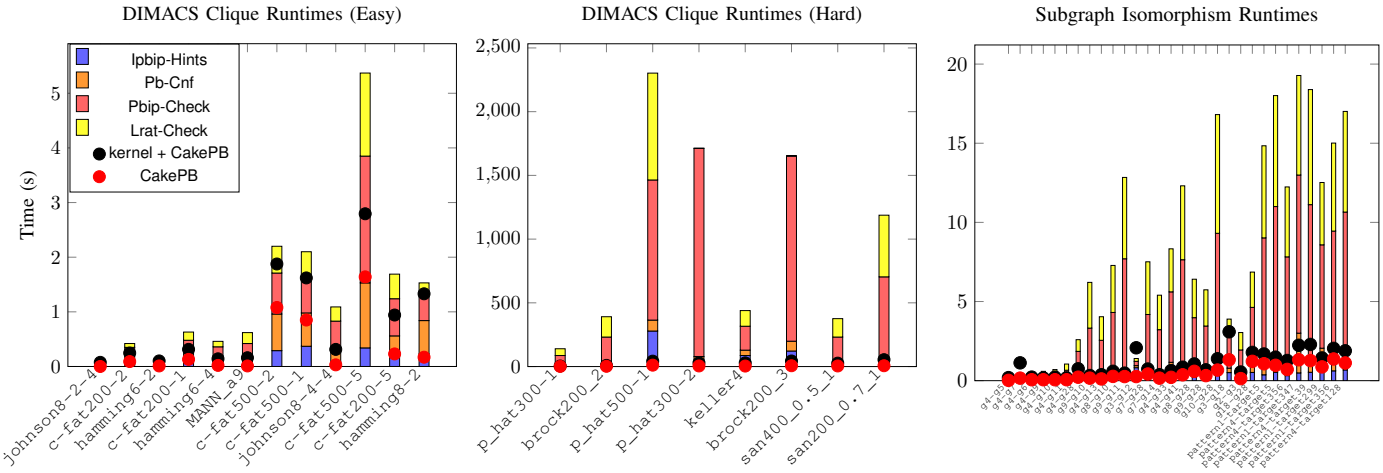

Fig. 8. (L) Toolchain performance on (easy) DIMACS Max-Clique benchmarks (M) Toolchain Performance on (hard) DIMACS Max-Clique benchmarks (R) Toolchain performance on (selected) Subgraph Isomorphism benchmarks
The red dot (labelled CakePB) corresponds to running solely the CakePB checker on an already-generated kernel format while the black dot (labelled kernel + CakePB) incorporates the time taken to generate the kernel format as well.
Note: `lrat_check` was unable to verify `p_hat300-2` and `brock200_3` from the middle (hard cliques) graph.

Set	IPBIP-HINTS	PBIP-CHECK	LRAT-CHECK
sip	0.8×	9.5×	6.9×
clq (all)	4.4×	42.5×	10.7×
clq (easy)	0.9×	4.2×	2.6×
clq (hard)	10.9×	109.5×	35.3×

Fig. 9. Average runtime overhead (ratio) of each phase in comparison with CakePB checking.

would require going beyond the implication-based proofs supported by current proof-generating BDD packages. It requires having proof rules that allow adding clauses that preserve satisfiability but exclude possible solutions to a formula, such as propagation redundancy [5], [43]. Translating the PB strengthening rules into clausal proofs remains an unsolved problem.

Fine tuning Performance/Tool Heuristics. Our tools make use of various heuristics (from proof-specific optimizations for trimming to BDD variable orderings). Fine-tuning these and optimizing relevant parts of the toolchain (more efficient structures for trimming, cache optimization) is definitely of interest and could see improvements in the tool runtimes described in Section VI-D.

Improvements to CakePB. Several of the optimizations we made in our toolchain could be applied to CakePB:

- *Direct support for RUP*. The CakePB toolchain requires converting a VeriPB proof into kernel format, replacing each RUP addition with a sequence of cutting-planes operations. Our experimental results show that this generally causes a small expansion in the proof size and a small time overhead, but it is awkward, and it prompted the authors to introduce special provisions to help users debug failed proofs [22]. We have shown that methods similar to those used by DRAT-trim [1] can be used to identify the unit propagation steps required to justify a RUP addition. These steps could be checked directly by CakePB.
- *Proof trimming*. Our experimental results show that many of the steps in VeriPB proofs are not relevant for a proof of unsatisfiability. Trimming these can reduce the checking time. It can also enable generating an "unsat core" identifying the key properties of the problem that cause it to be unsatisfiable. This capability has many applications beyond proof generation [44].

ACKNOWLEDGMENTS

The authors thank the MIAO group and their collaborators—in particular Ciaran McCreesh, Andy Oertel, and Yong Kiam Tan–for support with their tools, benchmarks, and general advice regarding pseudo-Boolean solving. Special thanks to Ciaran McCreesh for numerous in-depth clarifications on these matters. In addition, the authors acknowledge Ruben Martins and Joseph Reeves of Carnegie Mellon for helpful advice and encouraging discussions over the course of the project. Finally, the authors thank Stephen Deems of the Pittsburgh Supercomputing Center for providing the computing resources to run our experiments.

This work was supported by the U. S. National Science Foundation under grant CCF-2108521.

References

[1] M. J. H. Heule, "The DRAT format and DRAT-trim checker," *arXiv preprint arXiv:1610.06229*, 2016.

[2] A. Balint, M. J. H. Heule, A. Belov, and M. Järvisalo, "The application and the hard combinatorial benchmarks in SAT competition 2013," *Proceedings of SAT Competition*, pp. 99–100, 2013.

[3] M. J. H. Heule, W. Hunt, M. Kaufmann, and N. Wetzler, "Efficient, verified checking of propositional proofs," in *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8*, pp. 269–284, Springer, 2017.

[4] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pp. 220–236, Springer, 2017.

[5] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "Verified propagation redundancy and compositional UNSAT checking in CakeML," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 2, pp. 167–184, 2023.

[6] M. J. H. Heule, "Schur number five," *CoRR*, vol. abs/1711.08076, 2017.

[7] P. M. Dearing, P. L. Hammer, and B. Simeone, "Boolean and graph theoretic formulations of the simple plant location problem," *Transportation Science*, vol. 26, no. 2, pp. 138–148, 1992.

[8] Y. Crama and P. L. Hammer, "Recognition of quadratic graphs and adjoints of bidirected graphs," in *Proceedings of the third international conference on Combinatorial mathematics*, pp. 140–149, 1989.

[9] P. L. Hammer and E. Shlifer, "Applications of pseudo-Boolean methods to economic problems," *Theory and decision*, vol. 1, pp. 296–308, 1971.

[10] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt, "An application of combinatorial optimization to statistical physics and circuit layout design," *Operations Research*, vol. 36, no. 3, pp. 493–513, 1988.

[11] F. A. Aloul, A. Ramani, I. Markov, and K. Sakallah, "PBS: a backtrack-search pseudo-Boolean solver and optimizer," in *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pp. 346–353, 2002.

[12] D. Chai and A. Kuehlmann, "A fast pseudo-Boolean constraint solver," in *Proceedings of the 40th annual Design Automation Conference*, pp. 830–835, 2003.

[13] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-Boolean SAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1-4, pp. 165–189, 2006.

[14] J. Elffers and J. Nordström, "Divide and conquer: Towards faster pseudo-Boolean solving.," in *IJCAI*, vol. 18, pp. 1291–1299, 2018.

[15] S. Gocht, R. McBride, C. McCreesh, J. Nordström, P. Prosser, and J. Trimble, "Certifying solvers for clique and maximum common (connected) subgraph problems," in *Principles and Practice of Constraint Programming (CP)*, 2020.

[16] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint program solver," in *Principles and Practice of Constraint Programming (CP)*, 2022.

[17] S. Gocht, *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, 2022.

[18] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pp. 466–483, Springer, 1983.

[19] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[20] R. E. Bryant, "TBUDDY: A proof-generating BDD package," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 49–58, IEEE, 2022.

[21] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified dominance and symmetry breaking for combinatorial optimisation," *Journal of Artificial Intelligence Research*, 2023.

[22] S. Gocht, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "End-to-end verification for subgraph solving," in *AAAI Conference on Artificial Intelligence*, 2024.

[23] B. Bogaerts, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "Documentation of VeriPB and CakePB for the SAT competition 2023 (Mar 2023)." https://satcompetition.github.io/2023/downloads/proposals/veripb.pdf.

[24] W. J. Cook, C. R. Coullard, and G. X. R. Turán, "On the complexity of cutting-plane proofs," *Discrete Applied Mathematics*, vol. 18, pp. 25–38, 1987.

[25] S. A. Cook, "Feasibly constructive proofs and the propositional calculus," in *ACM Symposium on the Theory of Computing (STOC)*, pp. 83–97, 1975.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.

[27] R. E. Bryant and M. J. H. Heule, "Generating extended resolution proofs with a BDD-based SAT solver," *ACM Transactions on Computational Logic*, vol. 24, no. 4, pp. 1–28, 2023.

[28] R. E. Bryant and M. J. H. Heule, "Dual proof generation for quantified Boolean formulas with a BDD-based solver," in *Conference on Automated Deduction (CADE)*, vol. 12699 of *LNAI*, pp. 433–449, 2021.

[29] R. E. Bryant, A. Biere, and M. J. H. Heule, "Clausal proofs for pseudo-Boolean reasoning," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 443–461, Springer, 2022.

[30] C. Sinz and A. Biere, "Extended resolution proofs for conjoining BDDs," in *Computer Science Symposium in Russia (CSR)*, vol. 3967 of *LNCS*, pp. 600–611, 2006.

[31] T. Jussila, C. Sinz, and A. Biere, "Extended resolution proofs for symbolic SAT solving with quantification," in *Theory and Applications of Satisfiability Testing (SAT)*, vol. 4121 of *LNCS*, pp. 54–60, 2006.

[32] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "A new look at BDDs for pseudo-Boolean constraints," *Journal of Artificial Intelligence Research*, vol. 45, pp. 443–480, 2012.

[33] O. Roussel and V. Manquinho, "Input/output format and solver requirements for the competitions of pseudo-Boolean solvers." https://www.cril.univ-artois.fr/PB12/format.pdf, 2012.

[34] J. Elffers and J. Nordström, "Divide and conquer: Towards faster pseudo-boolean solving.," in *IJCAI*, vol. 18, pp. 1291–1299, 2018.

[35] C. McCreesh, P. Prosser, and J. Trimble, "The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants," in *International Conference on Graph Transformation*, pp. 316–324, Springer, 2020.

[36] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993*, vol. 26. American Mathematical Soc., 1996.

[37] C. Solnon, "Benchmarks for the subgraph isomorphism problem." http://liris.cnrs.fr/csolnon/SIP.html, 2016. visited on May 11th, 2024.

[38] A. Biere and M. Fleury, "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022," in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions* (T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2022-1 of *Department of Computer Science Series of Publications B*, pp. 10–11, University of Helsinki, 2022.

[39] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *Acm Sigact News*, vol. 8, no. 4, pp. 28–32, 1976.

[40] I. Grosof, N. Zhang, and M. J. H. Heule, "Towards the shortest DRAT proof of the pigeonhole principle," 2022.

[41] R. E. Bryant and M. J. H. Heule, "Generating extended resolution proofs with a BDD-based SAT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, vol. 12651 of *LNCS*, pp. 76–93, 2021.

[42] C. R. Codel, J. Avigad, and M. J. H. Heule, "Verified encodings for SAT solvers," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 141–151, IEEE, 2023.

[43] M. J. H. Heule, B. Kiesl, and A. Biere, "Strong extension-free proof systems," *Journal of Automated Reasoning*, 2019.

[44] J. P. Marques-Silva, "Minimal unsatisfiability: Models, algorithms, and applications," in *IEEE Symposium on Multi-Valued Logic*, 2010.