EPiC Series in Computing

Volume 59, 2019, Pages 34–49

Proceedings of Pragmatics of SAT 2015 and 2018

Checking Unsatisfiability Proofs in Parallel

Norbert Manthey* and Tobias Philipp

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany

Abstract

Contemporary SAT solvers emit unsatisfiability proofs in the DRAT format to increase confidence in their answers. The state-of-the-art proof verifier drat-trim uses backward-checking and deletion information to efficiently check unsatisfiability results. Checking large proofs still takes as long as solving a formula, and therefore parallelization seems to be a promising approach. However, Heule et al. stated that parallelization of the backward-checking procedure is difficult since clausal proofs do not include dependency information. In this paper, we present a parallelization approach and a prototypical implementation that scales reasonably compared to its sequential version.

1 Introduction

Satisfiability solvers can be buggy: Three solvers that participated in the SAT competition 2009, and five solvers that participated in the SAT competition 2007 were buggy, and incorrect results were given [4]. The critical case is when formulas are incorrectly reported to be unsatisfiable, since the answer is hard to verify. Subtle bugs in different components of satisfiability solvers were reported in [8,11]. Moreover, some bugs cannot be found with intensive blackbox-testing, known as *fuzzing* [4], since the configuration space of contemporary satisfiability solvers is too large. For improving the reliability of SAT solvers, one can mechanically verify them, as done in [12,13]. However, none of them can compete with state-of-the-art solvers, written in C or C++, such as `CryptoMiniSAT`, `Lingeling`, `Glucose`, and `Riss`. Alternatively, the solvers above can compute witnesses of unsatisfiability in the DRAT [16] or TraceCheck [3] format. Afterwards, the witness can be checked by `drat-trim` [16] (`TraceCheck` [3], resp.), a relatively small and widely used proof-checker, or by a mechanically verified proof checker [17]. Unsatisfiability proofs are also useful in extracting minimal unsatisfiable subformulas [2].

Proofs in the DRAT format are traces of the learned and deleted clauses in a CDCL-style SAT solver. Checking DRAT proofs can be done by checking that every learned clause in the proof is a resolution asymmetric tautology [8]. Backward checking [6] was proposed to improve the efficiency by checking only clauses that participate in the derivation of the empty clause. However, Heule et al. stated in [6] that parallelization of the backward-checking procedure is difficult since clausal proofs do not include dependency information.

*This work is supported by the DFG grant HO 1294/11-1.

In this paper, we present a parallelization approach of the backward checking algorithm, and a prototypical, lock free implementation for the omnipresent multicore architecture. In particular, we describe the forward and backward checking procedure as well as the details of the implementation. Experiments show that our implementation scales reasonably when using 8 cores compared to the sequential version.

The rest of this paper is structured as follows: We begin in Sect. 2 with propositional logic, and a presentation of unsatisfiability proofs in satisfiability testing (SAT) solving with description of the forward and backward checking algorithm. Afterwards, in Sect. 3 we describe our parallelization approach and describe it using an example. In Sect. 5, we evaluate our prototypical implementation and discuss implementation details. Section 6 concludes the paper.

2 Background

2.1 Propositional Logic

We assume a fixed infinite set $\mathcal{V}$ of Boolean *variables*. A *literal* is a variable v (*positive literal*) or a negated variable $\neg v$ (*negative literal*). The *complement* $\overline{x}$ of a positive (negative, resp.) literal x is the negative (positive, resp.) literal with the same variable as x. In SAT, we deal with finite multisets, i.e. finite sets in which elements can occur several times, called *formulas.* Each clause C is a finite set of literals. We write a clause $\{x_1, \dots, x_n\}$ also as disjunction $(x_1 \vee \dots \vee x_n)$ and a formula $\{C_1, \dots, C_n\}$ as a conjunction $(C_1 \wedge \dots \wedge C_n)$. The empty clause is denoted by $\bot$.

2.2 Unsatisfiability Proofs in SAT Solvers

CDCL-based SAT solvers are systematic backtrack-search procedures improved with *clause learning* [14]. Whenever a conflicting assignment is detected, a clause is added to the formula that prevents similar conflicts in the future. For unsatisfiable formulas, the SAT solver repeats this process until the empty clause is learned. Learned clauses are trivial resolution derivations, that can be efficiently checked by reverse unit propagation [1]. This observation led to *clausal proofs* [5], that are sequences of learned clauses, where antecedents are omitted. Later, Järvisalo et al. generalized the concept of trivial resolution derivation to *resolution asymmetric tautologies (RAT)*, thus allowing proof generation for most known formula simplification techniques [8]. The DRAT (Deletion Resolution Asymmetric Tautology) format [16] is based on the notion of asymmetric literal addition [8]. Given a CNF formula F and a clause C, the set of asymmetric literals $\mathrm{al}(F, C)$ contains all literals L such that, for some literals $L_1, \dots, L_n$ occurring in C, the clause $(L_1 \vee \dots \vee L_n \vee \overline{L})$ belongs to F. We define the *asymmetric literal addition* function ala_F from the set of clauses to the set of clauses defined as $\mathrm{ala}_F(C) = C \vee \bigvee_{L \in \mathrm{al}(F,C)} L$. We consider the repeated application of asymmetric literal addition, i.e.

$$\mathrm{ala}_F(C) \uparrow 0 = C \qquad\qquad \mathrm{ala}_F(C) \uparrow n+1 = \mathrm{ala}_F(\mathrm{ala}_F(C) \uparrow n)$$

A clause C is an *asymmetric tautology* (AT) in F if, for some $n \geq 0$, $\mathrm{ala}_F(C) \uparrow n$ is a tautology. C is said to be a *resolution asymmetric tautology* (RAT) [8] in F if, C is an AT in F, or for some literal $L \in C$, all clauses $D \in F$ with $\overline{L} \in D$ verify that $C \otimes_L D$ is an AT in F.

Example 1. *Consider the following formula F*

$$(a \vee b) \wedge (\overline{d} \vee \overline{e}) \wedge (\overline{c} \vee \overline{d} \vee e) \wedge (\overline{c} \vee d) \wedge (\overline{a} \vee c \vee d) \wedge (\overline{a} \vee c \vee e) \wedge (a \vee \overline{b} \vee c)$$

The clause $(a \vee \overline{b})$ is an asymmetric tautology, since $\text{ala}_F(a \vee \overline{b}) \uparrow 4$ *is already a tautology.*

$$\begin{array}{lcll}
\text{ala}_F(a \vee \overline{b}) \uparrow 0 & = & (a \vee \overline{b}) & \\
\text{ala}_F(a \vee \overline{b}) \uparrow 1 & = & (a \vee \overline{b} \vee \overline{c}) & \textit{using } (a \vee \overline{b} \vee c) \\
\text{ala}_F(a \vee \overline{b}) \uparrow 2 & = & (a \vee \overline{b} \vee \overline{c} \vee \overline{d}) & \textit{using } (\overline{c} \vee d) \\
\text{ala}_F(a \vee \overline{b}) \uparrow 3 & = & (a \vee \overline{b} \vee \overline{c} \vee \overline{d} \vee e) & \textit{using } (\overline{d} \vee \overline{e}) \\
\text{ala}_F(a \vee \overline{b}) \uparrow 4 & = & (a \vee \overline{b} \vee \overline{c} \vee \overline{d} \vee e \vee \overline{e}) & \textit{using } (\overline{c} \vee \overline{d} \vee e)
\end{array}$$

Moreover, the clause $(a \vee \overline{b})$ is RAT with respect to literal a, since all possible resolvents on literal a are AT:

– *The clause $(a \vee \overline{b}) \otimes_a (\overline{a} \vee c \vee d) = (\overline{b} \vee c \vee d)$ is an asymmetric tautology, since* $\text{ala}_F(\overline{b} \vee c \vee d) \uparrow 2$ *is already a tautology:*

$$\begin{array}{lcll}
\text{ala}_F(\overline{b} \vee c \vee d) \uparrow 0 & = & (\overline{b} \vee c \vee d) & \\
\text{ala}_F(\overline{b} \vee c \vee d) \uparrow 1 & = & (\overline{b} \vee c \vee d \vee \overline{a}) & \textit{using } (a \vee \overline{b} \vee c) \\
\text{ala}_F(\overline{b} \vee c \vee d) \uparrow 2 & = & (\overline{b} \vee c \vee d \vee a \vee \overline{a} \vee \overline{e} \vee \overline{d}) & \textit{using } (\overline{a} \vee c \vee d), (\overline{a} \vee c \vee e)
\end{array}$$

– *The clause $(a \vee \overline{b}) \otimes_a (\overline{a} \vee c \vee e) = (\overline{b} \vee c \vee e)$ is an asymmetric tautology, since* $\text{ala}_F(\overline{b} \vee c \vee e) \uparrow 2$ *is already a tautology:*

$$\begin{array}{lcll}
\text{ala}_F(\overline{b} \vee c \vee e) \uparrow 0 & = & (\overline{b} \vee c \vee e) & \\
\text{ala}_F(\overline{b} \vee c \vee e) \uparrow 1 & = & (\overline{b} \vee c \vee e \vee \overline{d} \vee a) & \textit{using } (\overline{a} \vee c \vee e) \\
\text{ala}_F(\overline{b} \vee c \vee e) \uparrow 2 & = & (\overline{b} \vee c \vee e \vee \overline{d} \vee a \vee b) & \textit{using } (a \vee \overline{b} \vee c)
\end{array}$$

□

The addition of asymmetric tautologies to a formula preserves semantical equivalence, while addition of resolution asymmetric tautologies to a formula preserves satisfiability [8].

Example 2. *Consider the formula F from the preceding example and the clause $(a \vee \overline{b})$. The clause $(a \vee \overline{b}) \otimes_b (a \vee b) = (a)$ is an asymmetric tautology in $F \wedge (a \vee \overline{b})$.*

A *RAT proof* in a formula F is then a sequence of clauses such that every clause is a RAT with respect to the formula F together with the preceding clauses.

Heule et. al included also *deletion information* resulting the notion of a *DRAT proofs*: The idea is to cover clause forgetting of SAT solvers in the proof, and to allow a faster proof verification. A *DRAT proof* in a formula F is a sequence of labeled clauses, i.e. expressions of the form $\mathsf{a}\,C$ and $\mathsf{d}\,C$ such that every clause is a RAT with respect to the formula F and the preceding clauses, but excluding the clauses that were deleted. Formally, a *labeled clause sequence* is a finite sequence of labeled clause $(\alpha_i \mid 0 \leq i \leq n)$. The *empty* derivation is denoted by Λ. For a labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$ and a formula F, we assign *associated formulas F_i* as follows

$$F_0 = F$$

$$F_i = \begin{cases} F_{i-1} \cup C & \text{if } \alpha_i = \mathsf{a}\,C \\ F_{i-1} \setminus C & \text{if } \alpha_i = \mathsf{d}\,C \end{cases}$$

A labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$ is a *DRAT proof* in the formula F if and only if for all $1 \leq i \leq n$ with $\alpha_i = \mathsf{a}\,C$ it holds that the clause C is a RAT in F_{i-1}. A *DRAT refutation for the formula F* is a DRAT proof α in F such that we find that $\perp \in F_n$.

```
  input  : input formula $F$, and labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$
  output: true, if $\alpha$ is a DRAT refutation for $F$
1 for $i \leftarrow 1$ to $n$ do
2     $(\ell, C) \leftarrow \alpha_i$;
3     if $\ell =$ del then continue;
4     if $C$ is not a resolution asymmetric tautology in $F_{i-1}$ then
5         terminate with "$\alpha$ is no DRAT refutation for $F$";
6     end
7 end
8 terminate with "$\alpha$ is a DRAT refutation for $F$";
```

Algorithm 1: Forward checking algorithm

2.3 Efficient Proof Checking

We now discuss two approaches for proof checking: forward [5] and backward checking [6].

2.3.1 Forward Checking

Forward checking [5] (see Alg 1) is a straightforward procedure to check DRAT refutations: Given an input formula F, and a labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$, forward checking traverses the proof from the beginning to the end and checks whether each learned clause is a RAT with respect to the associated formula (line $1-7$). For each labeled clause, we extract the label ℓ and clause C in line 2. Deletion information are not explicitly handled in the algorithm (see line 3), but hidden in the expression F_i. Afterwards, we check whether the clause C is a resolution asymmetric tautology in F_{i-1} in line 4. If not, the algorithm terminates with the answer that α is not a DRAT refutation for F (line 5). Otherwise, we check the next clause.

Forward checking can easily be parallelized up to n processors, where n is the number of learned clauses in the labeled clause sequence. In this case, the processor P_i computes F_{i-1} and checks whether the ith learned clause in the labeled clause sequence is a resolution asymmetric tautology in F_{i-1}. However, the computation of the associated formula F_{i-1} is still performed sequentially on every processor. Furthermore, many redundant clauses will be checked, although a subset of the labeled sequence might be sufficient to prove unsatisfiability.

2.3.2 Backward Checking

Experiments have shown that most of the clauses learned by the SAT solver are unnecessary for proving unsatisfiability [15]. Therefore, DRAT refutations contain much more clauses than necessary to prove unsatisfiability. *Backward checking* [6] (see Alg 2) is an algorithm that tries to omit checking unnecessary clauses. This is done by checking the proof *backwards*, and checking only clauses that are antecedents of the empty clause. Given a formula F and a labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$, the procedure works as follows: In line 1, we initialize the set of *marked clauses*, $\mathcal{M}$ to the singleton set consisting only of the empty clause. Then, we traverse the proof backwards starting with α_n (lines 2–14). In line 3, we store the label and the clause of the labeled clause α_i in ℓ and C. Deletion information and unmarked clauses do not require checks, and are skipped (line 4). The following lines (6–13) check whether C is an asymmetric tautology, or a resolution asymmetric tautology, and then adjusts the set of

```
input  : input formula $F$, and a labeled clause sequence $(\alpha_i \mid 1 \leq i \leq n)$
output: answers, whether $\alpha$ is a DRAT refutation for $F$
1  $\mathcal{M} \leftarrow \{\bot\}$ ;                                   // set of marked clauses
2  for $i \leftarrow n$ to 1 do
3      $(\ell, C) \leftarrow \alpha_i$ ;
4      if $\ell = \mathsf{del}$ or $C \notin \mathcal{M}$ then continue;
5      /* start checking                                                          */
6      if $C$ is an asymmetric tautology in $F_{i-1}$ then
7          $\mathcal{M}' \leftarrow$ clauses obtained by conflict analysis ;
8      else if $C$ is a resolution asymmetric tautology in $F_{i-1}$ w.r.t. $L$ then
9          $\mathcal{M}' \leftarrow$ union of clauses obtained by conflict analysis with respect to the
            resolvent $C \otimes_L D$, where $D \in F$ and $\overline{L} \in D$ ;
10     else
11         terminate with "$\alpha$ is not a DRAT refutation for $F$"
12     end
13     $\mathcal{M} \leftarrow (\mathcal{M} \setminus \{C\}) \cup \mathcal{M}'$ ;
14 end
15 terminate with "$\alpha$ is a DRAT refutation for $F$"
```

Algorithm 2: Backward checking algorithm.

marked clauses. First, we check whether C is an asymmetric tautology (line 6). In this case, we compute a small set of clauses $\mathcal{M}'$, by conflict analysis, that shows that C is an asymmetric tautology (line 7). Otherwise, we check whether C is a resolution asymmetric tautology with respect to the literal L (line 8). In this case, we compute $\mathcal{M}'$ as follows: For each resolvent E, that needs to be checked for being an asymmetric tautology, we compute a small set of clauses that is responsible that E is an asymmetric tautology, by conflict analysis. Then, we collect all of these clauses (line 9). Otherwise, C is no RAT, and we terminate the algorithm with the answer that α is not a DRAT refutation for F. In line 13, we remove the clause C from the set of marked clauses, and add all antecedents of C to $\mathcal{M}$.

Finally, we terminate with the answer that α is a DRAT refutation for the input formula, since all marked clauses occurring in the proof are checked, including the empty clause (line 15).

Please note that backward checking terminates with the answer that α is a DRAT refutation for F, even if some clauses in α are no resolution asymmetric tautologies. This can happen, if a SAT solver is buggy and adds a clause C to its learned clause database, but the clause will not be marked during proof checking. However, in this case, such clauses can be omitted from α, resulting in a DRAT refutation for F in our sense. In such a case the unsatisfiability of a formula can still be checked.

Please also note that, in the worst case, the backward checking algorithm checks *every* clause as the forward checking algorithm.

2.4 Analysis of Backward-Checking

Our research in parallelization of proof checking was motivated by the following observation: There are many marked clauses during the proof checking that have not been checked. This distribution is illustrated in Fig. 1, for a proof generated by glucose 3.0 for the instance p01_lb_05.cnf, and checked by `drat-trim`: The x-axis is the number of checked clauses (lemmas) of the proof, whereas the y-axis is the number of marked clauses that are not in the input formula and have not been checked already. Initially, the graph starts at zero checked clauses and no marked clauses, but steeply grows to ca. 5000 marked clauses, and falls down later.

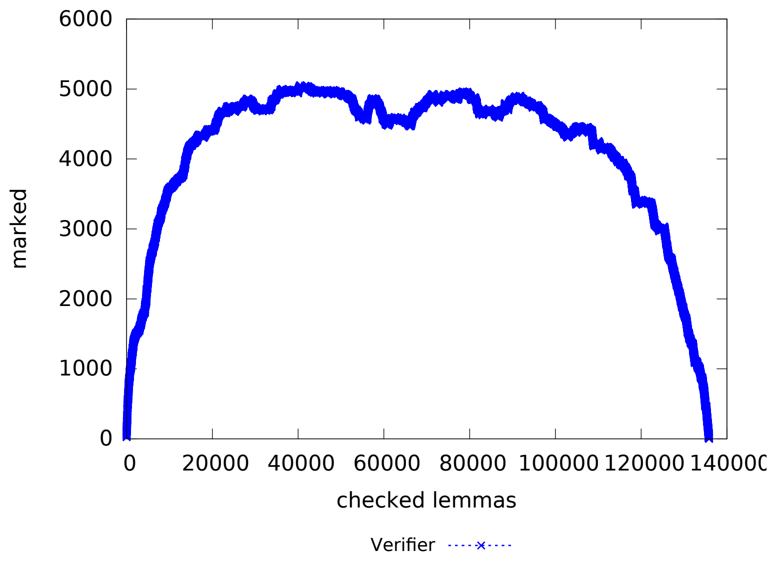


Figure 1: Number of marked clauses for the proof of p01_lb_05.cnf, obtained by `Glucose 3.0`

3 Parallel Proof Checking

The main contribution of this paper is the parallelization approach for an unsatisfiability proof. The underlying idea uses load balancing and splits the clauses to be verified in the backward scheme among multiple workers. Then, multiple workers check the proof with the usual routine, mark clauses to be checked, and check only the clause that have been marked by themselves.

During initialization, a single worker starts checking the proof, until its number of clauses to be checked reaches a predefined threshold. Then, every second marked clause is passed to a new worker. This procedure is continued recursively until all workers have received a set of clauses to be verified.

During verification, each worker iterates from the end to the beginning of the DRAT proof and maintains a private state. When there is a clause labeled to be verified by this worker, then the worker applies the usual verification routine to determine whether the clause is an AT or a RAT. During this step, the worker also marks all clauses that are necessary for this derivation, such that those clauses are verified as well. Note, that the worker marks only clauses that have not been marked already (possibly by another worker). In this way, we avoid redundant work. Then, each marked clause is verified by (at least)[1] one worker. After successful verification, the worker continues iterating backwards on the DRAT proof, until it finds a clause that has

[1]Due to race conditions, two workers might mark a clause at the same time

been labeled by itself. If a clause in the proof is found to be neither AT nor RAT, then the verification is aborted, and the unsatisfiability proof is rejected.

The algorithm terminates if all workers verified all clauses that have been marked by them to be verified. The parallel algorithm produces the same result as the sequential algorithm, since all marked clauses will be verified by some worker. Hence, no clauses can be missed. If all verification steps have been successful, the unsatisfiability proof is accepted. Otherwise, if some worker found a clause that could not be verified, the proof is rejected.

The actual implementation of this algorithm is discussed in more details, including the used data structures, in Section 5.1, such that the relation to the sequential algorithm can be followed more easily.

4 An Example Trace of the Parallel Proof Checker

In this section, we illustrate our parallelization approach with an example (see Table 1). As input formula, we consider the formula from Example 1, and consider two workers, W1 and W2. The table is distributed into four parts: The first seven columns represent the proof construction, the columns 8–11 represent the marked clauses during proof checking, the column 12 presents performed unit propagations for verification of the AT and RAT property, and columns 13 and 14 present the marked clauses for the two workers except those that are present in the input formula. Moreover, the table is horizontally split into two parts: In the upper part, information regarding the input clauses are presented, whereas in the lower part, information regarding the clauses in the proofs are presented.

The seven input clauses are listed in the upper-left corner of the table, where each clause is assigned a unique identifier (ID) from 1 to 7. In the left part of the table, below the horizontal line, a trace of the SAT solver is given, showing that the input formula is unsatisfiable: First, it performs *covered literal elimination (CLE)* [10], as specified in the *Op-column*, with respect to the variable a, resulting in the clause $(a \vee \overline{b})$. In the *Uses-column*, clause identifiers are given that are necessary for the operation. The clause is assigned the unique identifier 8, and is the first proof step. Afterwards, the SAT solver detects that the clause with identifier 8 subsumes (SUB) the clause with identifier 7, and consequently removes $(a \vee \overline{b} \vee c)$. Deletion information are shown in the *D-column* with the D-mark. Then, the solver applies resolution (RES) using the clauses with identifiers 1 and 8, resulting in the unit clause (a). In the following 12 steps, the SAT solver shows that one can reach the empty clause with a few more resolution steps, and furthermore clauses are removed by subsumption. We maintained an additional column, the *Present Until-column*, that shows the maximal i such that the clause is an element of the associated formula F_i. If a clause is not deleted among the proof, we denote this by ∞.

Proof checking is performed backwards, i.e. the first step, as given in the *verification step-column*, is to check the empty clause. The empty clause has been marked at time point 0 (compare to line 1 in Alg. 2). Please note that proof construction and checking are two different processes. In particular, backward checking can use different clauses in resolution chains. In the *unit propagation-column*, the reverse unit propagation chain is presented: One can infer d with the reason clause 16, the literal $\overline{e}$ with reason clause 20, $\overline{c}$ with reason clause 3, as well as e with reason clause 13. Consequently, we reached a conflict and the empty clause is an asymmetric tautology. The set of marked clauses is then $\{13, 16, 20\}$, and this work is distributed among both workers, as illustrated in the last two columns. Clause 16 is passed to worker W2 to split the work. Note, that clause 3 is not present in this set, as this clause is part of the input formula. In the example, the information in which verification step a clause got marked locally and globally is also presented in the *Marked-column*. In the first verification step, the clause 3,

Table 1: A trace of a SAT solver on a small example, generated proof and trace of the parallel proof verifier: On the left side, it is demonstrated how the the proof has been created step by step. On the right side, it shows how the proof is verified. We consider two workers W1 and W2, where the tasks are distributed among the two workers accordingly. The symbol ∞ denotes that a clause will not be deleted among the proof.

Proof Step	ID	D	Clause	Uses Clause	Op	Present Until	Veri. Step	Marked (global)	Marked by W1	Marked by W2	Unit Propagations	Work W1	Work W2
	1		$(a \vee b)$			4		*13*		*13*			
	2		$(\overline{d} \vee \overline{e})$			14		*3*	*3*				
	3		$(\overline{c} \vee \overline{d} \vee e)$			∞		*1*	*1*				
	4		$(\overline{c} \vee d)$			10		*7*		*7*			
	5		$(\overline{a} \vee c \vee d)$			11		*7*		*7*			
	6		$(\overline{a} \vee c \vee e)$			7		*10*	*10*	*15*			
	7		$(a \vee \overline{b} \vee c)$			2		*15*		*15*			
1	8		$(a \vee \overline{b})$	7, 5, 2, 6	CLE(a)	5	*15*			*13*	$\overline{a} \wedge b \rightarrow (c,7),(d,4),(\overline{e},2),(e,3)$	$\emptyset$	$\emptyset$
2	9	D	$(a \vee \overline{b} \vee c)$	7, 8	SUB		*14*						
3	10		(a)	1, 8	RES	∞	*13*	*7 [10]*		*7*	$\overline{a} \rightarrow (b,1),(\overline{b},8)$	$\{8\}$	$\{8\}$
4	11	D	$(a \vee b)$	1, 10	SUB		*12*						
5	12	D	$(a \vee \overline{b})$	8, 10	SUB		*11*						
6	13		$(c \vee e)$	6, 10	RES	∞	*10*	*1*	*1*		$\overline{c} \wedge \overline{e} \rightarrow (\overline{a},6),(a,10)$	$\{10\}$	$\{10\}$
7	14	D	$(\overline{a} \vee c \vee e)$	6, 13	SUB		*9*						
8	15		$(d \vee e)$	4, 13	RES	12	*8*		*3?*	*7?*		skip	
9	16		(d)	4, 5, 10	RES	∞	*7*	*1*	*1*$\rightarrow$	*1*	$\overline{d} \rightarrow [(e,15)],(\overline{c},4),(\overline{a},5),(a,10)$	$\{13\}$	$\{10\}$
10	17	D	$(\overline{c} \vee d)$	4, 16	SUB		*6*						
11	18	D	$(\overline{a} \vee c \vee d)$	5, 16	SUB		*5*						
12	19	D	$(d \vee e)$	15, 16	SUB		*4*						
13	20		$(\overline{e})$	2, 16	RES	∞	*3*	*1*	*1*		$e \rightarrow (\overline{d},2),[\overline{c},4],(d,16)$	$\{13\}$	$\{16\}$
14	21	D	$(\overline{d} \vee \overline{e})$	2, 20	SUB		*2*						
16	22		$\perp$	3,13,16,20	RES	∞	*1*	0			$\perp \rightarrow (d,16),(\overline{e},20),(\overline{c},3),(e,13)$	$\{20,13\}$	$\{16\}$

13, 16 and 20 are marked by worker W1, so that these flags also appear in the example. The arrow of clause 16 denotes that this mark is moved to another worker.

In the second verification step, deletion information are presented and no further checks are required.

Next, in step 3, worker W1 verifies clause 20 and checks whether this unit clause $(\overline{e})$ is an asymmetric tautology. In the corresponding *Unit propagation-column*, a redundant unit rule step is given in brackets, which in fact is unnecessary. Therefore, the reason clause 4 is *not* marked. To mark as few clauses as possible, not all clauses that are used during unit propagation are marked, but only the clauses that are used to derive the conflict. This set is constructed by conflict analysis, and results in marks for the clauses 2 and 16.

In step 7, worker W2 verifies the clause 16, which has been forwarded from worker W1. Again, the same redundant implication of e with clause 15 is present. A conflict is found, by using the clauses 4, 5, and 10, which are all marked to be verified.

In the next step, step 10, worker W1 verifies the clause 13, resulting in marks for the clause 6 and 10. Due to race conditions, the mark for clause 10 might be set by worker W1, instead of W2, or by both workers at the same point in time. Then, the two workers would check clause 10. This mark is given in brackets. Assume worker W2 marked clause 10 already, then this clause is also not marked locally by W1, and W1 will not check clause 10.

Next, W2 verifies clause 10 in step 13. In this step, clause 1 and clause 8 have to be marked with the current verification step for worker W2.

Finally, clause 8 is verified by worker W2 in step 15. The clause 8 is an asymmetric tautology, i.e. unit propagation will result in a conflict. In the case this check would fail, we need to check whether the clause 8 is a strict resolution asymmetric tautology, i.e. consider all resolution candidates for the literal a, namely the clauses 5 and 6. Next, for the resulting resolvents, $(\overline{b} \vee c \vee d)$ and $(\overline{b} \vee c \vee e)$, respectively, would be checked to be asymmetric tautologies: From $(b \wedge \overline{c} \wedge \overline{d})$ we obtain $(a, 7)$ and $(\overline{a}, 5)$ and from $(b \wedge \overline{c} \wedge \overline{e})$ we obtain $(a, 7)$ and $(\overline{a}, 6)$. As both resolvents are AT due to the found conflicts, clause 8 would be verified and the clauses 5, 6 and 7 would be marked with step 15 (as illustrated in Table 1).

Finally, both workers have no marked clauses outside the input formula, and therefore the proof checker accepts this proof, as all clauses have been verified successfully. By exploiting the information of the mark information, the unsatisfiable core of the formula, as well as the reduced proof can be extracted. In the core, all clauses of the formula are necessary. Note, that this core is build by combining the marks from the two workers. The reduced proof is obtained by collecting all marked clauses. In this example, only clause 15 can be dropped.

5 Evaluation

To show that our parallelization approach scales, we first implemented a backward DRAT verification procedure, and then applied the above parallelization scheme, including all necessary additional data structures for communication. The implementation of our tool `proofcheck` is described in more details below. This implementation is similar to the implementation of `drat-trim`, however, the implementation of `drat-trim` has not been discussed in the presented level of detail. We present all details, such that a discussion of the results of the parallelization can be done more comprehensible. Afterwards, we present the experimental setup and a comparison to the `drat-trim` tool, as well as a scalability analysis.

5.1 Implementation

We implemented `proofcheck` on top of `Riss`[2] in C++, and use the available data structures for clauses and watch lists, which are similar to `MiniSAT`'s data structures, except that binary clauses are stored in the watch list directly. During parsing the proof, for each element the step window of its presence is generated as well, effectively turning the parsed proof into an IODRAT proof (see the example above, as well as [7]). To improve parsing time, a huge hash-map is used to match clauses that are deleted again.

The parallel verification is implemented similar to parallel portfolio SAT solvers: all incarnations have private data structures and only a few structures are shared. In `proofcheck`, each incarnation has its private copy of the clause database and the watch list, and shares only the information about whether a clause has been marked and verified already. This way, memory consumption is increased, but fast unit propagation with private clauses accesses is possible. The information whether a clause is verified already, and whether it is marked by some incarnation already is stored in one shared array. As all write operations to this array write the same information (a clause has been marked or a clause has been verified), no locks are needed for the parallel implementation.

The implementation aims at being generic, such that more extensions can be added easily, for example extracting proof properties. The current implementation already supports extracting the proof width, length and space.

5.1.1 Checking a Single Clause

A clause is checked by an incarnation only if this clause is marked by this incarnation (private flag), and if this clause is not verified by another incarnation already (shared flag). The latter flag tries to avoid that a clause is checked twice. Checking a marked clause C is done as follows: The complements of the literals of C are added as search decisions, and then propagation is executed until a conflict is hit. During propagation, all entries in the watch lists that are not present any longer are deleted on-the-fly. Furthermore, propagation is executed first on marked clauses as long as possible, before continuing on non-marked clauses. This information is read from the global array. After propagating one unmarked clause, propagation is continued on unmarked clauses. This procedure tries to ensure that marked clauses are used as often as possible, to reduce the number of clauses that have to be checked. For this procedure, two pointers for the propagation queue are necessary, as well as an additional pointer to memorize the state of the watch list for unmarked clauses.

When unit clauses are added in the backward procedure (due to deletion information), then the top level propagation is performed w.r.t. all present clauses. This top level state is used for all verification steps, until a clause that is used as a reason on the top level is removed. Then, the top level propagation is performed again.

Conflict Analysis and Marking Clauses We could mark all clauses that are used during unit propagation and continue with the next clause. By using conflict analysis, the set of marked clauses can be reduced by spending a little more time during checking one clause. In our implementation, we have chosen the latter approach.

Conflict analysis is performed with the found conflict clause. To ensure marking all clauses that are necessary to derive the clause to be verified, analysis is performed until only decision literals are left in the learned clause, i.e. a decision clause is created. All clauses that are used

[2]The implementation is available at `http://tools.computational-logic.org/content/riss/proofcheck.tar.gz`.

in the resolution derivation are marked to be verified as well. The order to mark clauses is as follows: If the clause is marked globally for verification already, no mark is set, because some other incarnation marked this clause. Otherwise, the global mark is set by this worker. Here, a race condition might allow that two incarnations perform the exact same steps synchronously, resulting in redundant work, i.e. checking the clause twice. If the global mark was not set by some other worker, the clause is furthermore marked privately, such that this incarnation will check the marked clause.

DRAT Clauses If no conflict is encountered, the first literal l of C is considered as the literal that has been used to generate a DRAT clause.[3] Then, the procedure to check a clause is repeated for all resolvents $C \otimes D$, for all $D \in F_{\bar{l}}$, including marking additional clauses. Furthermore, all clauses D are marked to be verified, independently of whether the resolvent $C \otimes D$ becomes a tautology or not. However, D is not marked, if it is a tautology. To find all clauses D efficiently, we store an additional data structure that stores per literal all present clauses that contain this literal. This structure is also updated lazily, similarly to the watch lists.

5.1.2 Work Splitting

The parallel verification is started with a single incarnation, which verifies a few clauses, until a certain threshold of marked clauses is reached. Then, the procedure is interrupted, and the load is distributed among two incarnations: every second marked clause is moved to the new incarnation. If there are more computing resources, the above procedure is repeated for the two incarnations, and all marked clauses are spread over four incarnations. The above routine is repeated until all incarnations have been initialized with marked clauses. If the number of available resources is no power of 2, then in the last iteration only the required number of incarnations are added.

In the experiments, an incarnation is interrupted as soon as it marked two clauses that have to be verified.

5.2 Setup

The used cluster uses Intel Xeon E5-2670 CPUs with 8 cores and 20 MB level 3 cache that is shared by all cores. Each node of the cluster has two CPUs available, but we use only a single eight-core CPU. The allowed main memory varies over the experiments. The wall clock time limit for proof verification is 10000 seconds.

The evaluation of the techniques is performed on all unsatisfiable application instances of the SAT Competition 2014. As benchmark we used all proofs that could be produced by `druplig` for 150 unsatisfiable application formulas of 2014. For this solving process, we allowed 5000 seconds of runtime, and 6.5 GB main memory. This way, we received 131 proofs.

5.3 Results

We first analyze the scalability of our prototype `proofcheck`. Therefore, checking all proofs with a sequential configuration, and with a parallel configuration that uses 8 cores. The sequential configuration is allowed to use 20 GB of memory, and the parallel configuration is allowed to use 100 GB of memory.

[3]The implementation also supports checking all literals of a clause w.r.t. DRAT.

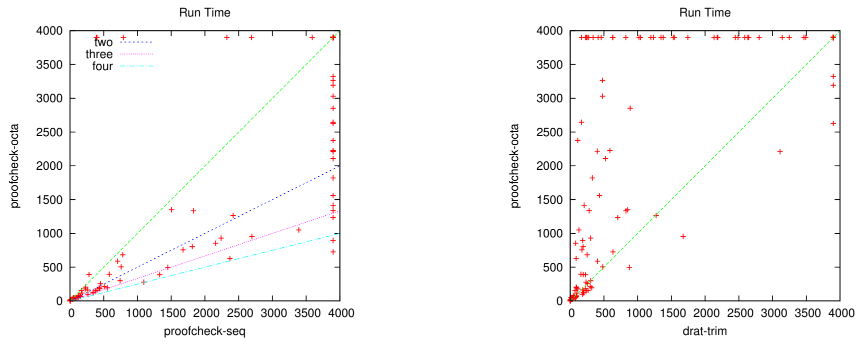


Figure 2: In the left diagram, the verification times of the sequential configuration and the parallel configuration of `proofcheck` with 8 workers are compared. In the right diagram the verification time of the parallel proofcheck configuration is compared to drat-trim.

The resulting run times are visualized in Figure 2. The plot shows, that the main goal of the parallelization is reached, the presented implementation scales reasonably well for the used number of resources. Hence, this work presents a step towards an efficient parallelization of parallel proof checking, an issues mentioned in [6].

Both the sequential, as well as the parallel algorithm can check 92 of the given proofs in the given resource limits, where 67 proofs can be solved by both algorithms, and each procedure can check 25 proofs uniquely. For these 67 proofs, Figure 3 shows, besides the relation of the unsatisfiable core and the proof length, the efficiency of the parallelization. The figure shows, that the efficiency of the approach increases with the length of the proof: in the upper half of the diagram, the color of the dots is turned into red, which means the efficiency is higher than 25 %. If the size of the core also increases, which implies that the formula has been larger as well, then the efficiency increases further: in the upper right corner, there are also red dots, that show that the efficiency of the implemented prototype reaches 50 %.

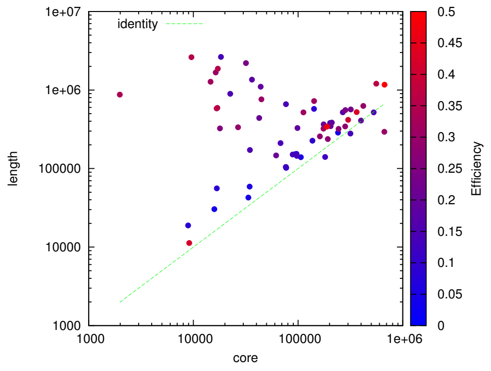


Figure 3: The diagram compares the size of the extracted unsatisfiable core (x-axis) with the length of the unsatisfiability proof (y-axis). More importantly, the color of each dot encodes the efficiency of the parallelization approach.

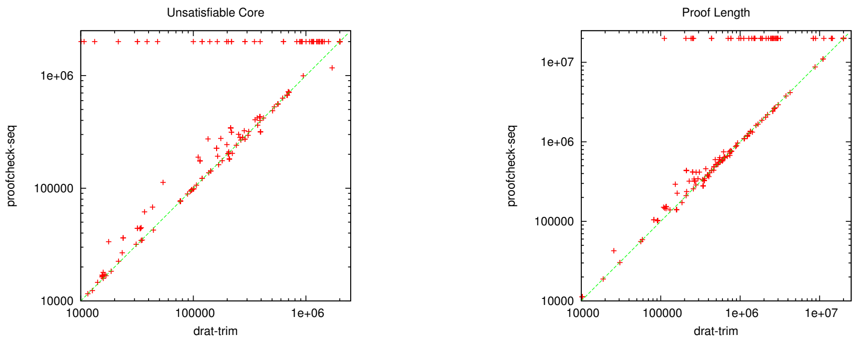


Figure 4: In the left diagram, the size of the unsatisfiable core that has been extracted from the original formula is compared between drat-trim and proofcheck. In the right diagram, we compare the number of clauses in the proof that have been used to actually verify unsatisfiability.

5.4 Comparison to drat-trim

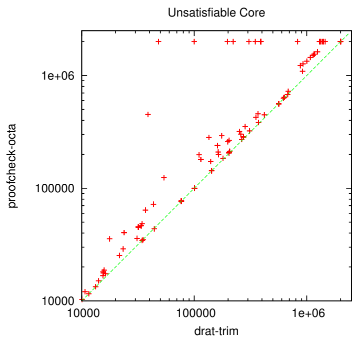
We furthermore compared our prototype implementation to `drat-trim`, the currently fastest available DRAT proof verifier. The runtime of the sequential implementation is already presented in Figure 2 (right diagram). This plot shows that the `drat-trim` is faster than our parallel prototype except for a small number of proofs – as the plot shows – even when eight cores are used. By using `drat-trim`, 129 proofs can be verified. Another reason for the huge run time difference might be the implementation of the tools. While `drat-trim` is implemented in plain C, and puts all information into a single struct, `proofcheck` is implemented in C++, and adds extra data structures that require extra memory accesses. Furthermore, the data structures are reused in `proofcheck`: the item in a watch list stores the same information as an element in the proof list. In preliminary analysis we could see that the cache miss rate of `drat-trim` is much smaller than the rate of `proofcheck`, and the memory consumption is also smaller. We account these effects to the used way of implementation and data structures. As our aim in this work is to show that proof verification can be parallelized, we did not increase the complexity on data structures to decrease the cache miss rate. Similar steps are planned as future work.

As the implemented algorithm is only published in pseudo code, we expect some important details are hidden in the source code. We already implemented numerous improvements to the pseudo-code presented in [5–7]. The results we find with `proofcheck` are still comparable to `drat-trim`.

5.5 Comparison of Proof Length and Unsatisfiable Core Size

The runtime, which is spend during verification, is mainly due to the number of clauses that have to be verified. This number represents the reduced proof, which is extracted from the proof that is omitted by the SAT solver. If the proof lengths are comparable, then the amount of clauses to be verified is also comparable. Furthermore, the algorithm to mark clauses during the verification process might be different. Hence, the size of the unsatisfiable core that is extracted might also be important.

Figure 4 and 5 compare the mentioned values for the sequential and the parallel variant of `proofcheck`. The diagrams show in both pictures that the size of the unsatisfiable core in the formula is at most ten time higher than for `drat-trim`. Note that these clauses do



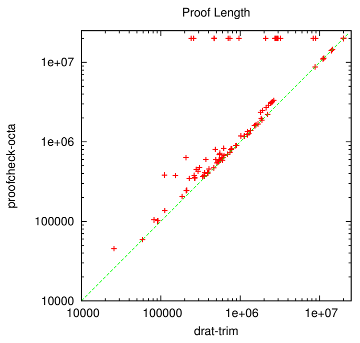


Figure 5: In the left diagram, the size of the unsatisfiable core that has been extracted from the original formula is compared between drat-trim and the parallel proofcheck configuration. In the right diagram, we compare the number of clauses in the proof that have been used to actually verify unsatisfiability.

not have to be checked, as they are part of the input formula. The difference of the length of the extracted proof is even smaller. Especially for long proofs, the length of `proofcheck` is sometimes smaller than for `drat-trim`. We assume that `proofcheck` simply follows a different propagation order, resulting in a different conflict analysis and hence different (and more) clauses might be marked.

When comparing the sequential variant to the parallel variant, we furthermore see, that the number of clauses in the unsatisfiable core are higher when multiple workers are used. Similarly, the length of the extracted proof increases when moving from one worker to eight workers. Still, as discussed above, an efficiency of 50 % can be achieved without any synchronization.

6 Conclusion

We presented an algorithm that performs backward proof checking of DRAT proofs in parallel. We furthermore implemented the prototype `proofcheck` and discussed the details of its implementation. The efficiency of `proofcheck` has been analyzed: for eight cores an efficiency of 50 % can be reached, even with shared resources and a preliminary implementation. Our new results are new steps towards efficient parallel backward checking, an issue raised in [6].

6.1 Future Work

Given the poor runtime behavior of `proofcheck` compared to the state-of-the-art verifier `drat-trim`, many future work is possible. First, we will try to extract the algorithmic details from `drat-trim`, which are not described in the corresponding publications. Furthermore, we will reduce the amount of required memory, by sharing the formula among all running incarnations. This way, the cache-miss rate of `proofcheck` should drop significantly.

We believe that smaller verification times for large proofs might be achieved if the tool is parallelized for large clusters. As the verification processes in `proofcheck` are only loosely coupled, we think that an OpenMPI implementation might offer further wall clock time reductions.

Finally, we like to add more features to the verifier, such that it is able to print more statistics of the proof. From our current point of research, the depth of a proof [9] is interesting and should be extracted automatically. In [9], the proof generation routine had to be adapted, to print this value. Once this feature is available during verification, this value can be computed for any proof that is available in the DRAT format.

Acknowledgments The author thank the ZIH of TU Dresden for providing the computational resources to produce the experimental data for the empirical evaluation.

References

[1] Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR 22(1), 319–351 (2004)

[2] Belov, A., Heule, M., Marques-Silva, J.: MUS extraction using clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 48–57. Springer (2014)

[3] Biere, A.: BooleForce and TraceCheck (2014), `http://fmv.jku.at/booleforce`

[4] Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010, LNCS, vol. 6175, pp. 44–57. Springer Berlin Heidelberg (2010)

[5] Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE 2003. pp. 10886–10891. IEEE Computer Society, Washington, DC, USA (2003)

[6] Heule, M., Hunt Jr, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Jobstman, B., Ray, S. (eds.) FMCAD 2013. pp. 181–188. IEEE (2013)

[7] Heule, M., Jr., W.A.H., Wetzler, N.: Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. Software Testing, Verification and Reliability 24(8), 593–607 (2014)

[8] Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)

[9] Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In: des Jardins, M., Littman, M.L. (eds.) Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence. AAAI Press (2013), `http://www.aaai.org/Library/AAAI/aaai13contents.php`

[10] Manthey, N., Philipp, T.: Formula simplifications as DRAT derivations. In: Lutz, C., Thielscher, M. (eds.) Annual German Conference on AI (KI 2014). LNCS, vol. 8736, pp. 111–122. Springer (2014)

[11] Manthey, N., Philipp, T., Wernhard, C.: Soundness of inprocessing in clause sharing SAT solvers. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 22–39. Springer, Heidelberg (2013)

[12] Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science 411(50), 4333 – 4356 (2010)

[13] Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 363–378. Springer (2012)

[14] Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD 1996. pp. 220–227. IEEE Computer Society, Washington (1996)

[15] Simon, L.: Post mortem analysis of sat solver proofs. In: Berre, D.L. (ed.) POS-14. EPiC Series, vol. 27, pp. 26–40. EasyChair (2014)

[16] Wetzler, N., Heule, M.J., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer (2014)

[17] Wetzler, N., Heule, M.J., Hunt Jr, W.A.: Mechanical verification of SAT refutations with extended resolution. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 229–244. Springer, Heidelberg (2013)