Faster LRAT Checking than Solving with CaDiCaL

Florian Pollitt
University Freiburg, Germany

Mathias Fleury
University Freiburg, Germany

Armin Biere
University Freiburg, Germany

Abstract

DRAT is the standard proof format used in the SAT Competition. It is easy to generate but checking proofs often takes even more time than solving the problem. An alternative is to use the LRAT proof system. While LRAT is easier and way more efficient to check, it is more complex to generate directly. Due to this complexity LRAT is not supported natively by any state-of-the-art SAT solver. Therefore Carneiro and Heule proposed the mixed proof format FRAT which still suffers from costly intermediate translation. We present an extension to the state-of-the-art solver CaDiCaL which is able to generate LRAT natively for all procedures implemented in CaDiCaL. We further present Lrat-Trim, a tool which not only trims and checks LRAT proofs in both ASCII and binary format but also produces clausal cores and has been tested thoroughly. Our experiments on recent competition benchmarks show that our approach reduces time of proof generation and certification substantially compared to competing approaches using intermediate DRAT or FRAT proofs.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases SAT solving, Proof Checking, DRAT, LRAT, FRAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.20

Category Tool Paper

Supplementary Material *Dataset (Log files)*: `https://cca.informatik.uni-freiburg.de/lrat/` [20]

Acknowledgements We thank reviewers of SAT'23 and MBMV'23 for their detailed comments as well as Mario Carneiro for making FRAT-rs publicly available.

1 Introduction

Proof production became an essential part in SAT solving. For instance, unsatisfiable problems only count as solved in the SAT Competition if a certifiable proof is provided. Proofs do increase trust in solving results by providing certificates that can be checked independently. To increase trust even further proof checkers can also be entirely verified [6, 16].

In the past the only format allowed in the SAT Competition was DRAT [23], even though the SAT Competition 2023 announced to allow additional formats. However, checking DRAT proofs often takes several times the amount of solving time. The problem with DRAT is that the format is not detailed enough to avoid search during checking. Both the solver and the checker have to propagate clauses (actually using similar data structures). To reduce this overhead (and simplify verification) all verified proof checkers expect an enriched format. The DRAT proof is augmented and converted by an (untrusted) external program into such an enriched format, e.g., LRAT [6] or GRAT [16], which contains enough information to avoid search and can then be checked easily by the verified proof checker.

On top of the actual clause contents (its literals) the LRAT [6] format requires the following additional information: (*i*) clause identifiers (ids) are used to reference clauses and to make clause deletion steps more concise; (*ii*) clause antecedent ids used in the resolution

chain when deriving an added clause through reverse unit propagation (RUP) [12], i.e., as asymmetric tautology (AT) [14]; (*iii*) the ID and further resolution paths to refute the resolvent of the added clause with all clauses containing a RAT (blocking) literal in case the added clause relies on the stronger resolution asymmetric tautology (RAT) property [15].

These RAT literals would be needed to model more powerful reasoning (such as blocked clause addition or symmetry breaking etc.) but neither our SAT solver CaDiCaL [4] nor any top performing SAT solver in the SAT Competition over the last 2 years actually used such reasoning. Therefore, our efforts to extend CaDiCaL did not need to address the full power of RAT and we can focus on producing "LRUP" proofs, i.e., reverse-unit-propagation (RUP) proofs, but still need to augment these proofs with ids and resolution chains.

A similar attempt [1] by Carneiro and Heule led to a new proof format, FRAT, that sits between LRAT (because it allows for justifications) and DRAT (because it still allows steps without justification). Their aim was to fill out most "*gaps*" and leave *"harder"* to implement cases as black box to be filled in by an (untrusted) proof checker, i.e., by their FRAT-rs tool used to convert an FRAT proof to a fully justified LRAT proof. In a recent paper [18] this limitation of the FRAT producing CaDiCaL [1] forced a parallel proof-producing version of the award winning SAT solver Mallob to deactivate all steps not covered by FRAT, i.e., most inprocessing, as native LRAT proof generation is needed.

In this tool paper, we present an extension of our SAT solver CaDiCaL [4] to generate the richer LRAT format directly. Our focus is on three different aspects: (A) producing LRAT proofs for all solver configurations on all benchmarks, (B) comparable performance and, further, (C) making sure the solver behaves the same with/without proof generation.

Our goal (A) lead us to reimplement LRAT generation in the conflict analysis and all inprocessing techniques of CaDiCaL, some of which were not covered in the FRAT [1] producing implementation, such as equivalent literal subsumption (Section 3).

Like other SAT solvers, CaDiCaL generates a vast number of proof steps from which at the end, a significant fraction turns out to be unnecessary for the derivation of the empty clause. Thus most tools that process DRAT or FRAT will trim these unnecessary steps from the proof. However, we are not aware of a tool that does this for LRAT. Therefore we implemented a new tool called Lrat-Trim to trim proofs down and improve the performance of checking the proof with the verified checker Cake_Lpr [21] (Section 4).

To validate robustness of our approach we extended CaDiCaL to internally check LRAT proofs too and fuzzed the extended solver. This allowed us to use the model-based tester Mobical (which comes with CaDiCaL) to find, debug, and fix bugs much more efficiently. We further ran the extended new solver on the unsatisfiable problems from the SAT Competition 2022. We observed (almost) no slow-down without proof production (0.3%) and only a small slow-down for producing LRAT (5%). Proof checking performance was improved considerably compared to the two competing approaches DRAT and FRAT (see Section 5). Checking (and producing) our LRAT proofs has an overhead of 30% over pure solving, compared to 125% for FRAT and 180% in the SAT Competition mode (i.e., slower than producing them). Without negligible overhead over plain solving with CaDiCaL, we managed to check proofs faster than they are produced for a state-of-the-art SAT solver.

Our CaDiCaL extension is available at `https://github.com/florianpollitt/radical` and will shortly be merged into the main CaDiCaL repository. Note that a preliminary version of this paper was presented at the MBMV workshop [19] as work in progress. Compared to that shorter version, we have improved and present Lrat-Trim, give an extensive evaluation on the entire problem set of the SAT Competition 2022 (not just a single problem) and in general provide more details on the implementation.

2 Preliminaries

For an introduction to SAT solving please refer to the *Handbook of Satisfiability* [5]. In our context it is sufficient to recall that SAT solvers build a partial assignment and along the way learn new clauses preserving satisfiability until either the assignment satisfies all clauses or the empty clause is derived, meaning that the problem is unsatisfiable.

A DRAT [23] proof is the sequence of all clauses learned (or in general deduced) by the SAT solver interleaved with clause deletion steps, which are used to help the proof checker to focus on the same clauses the solver would see at this point of the proof. This design principle helps DRAT [23] to easily capture all techniques currently used by SAT solvers without the need to provide more complex justification e.g. in the form of resolution chains.

The LRAT [6] proof format has more detailed information: Each clause is associated with a clause identifier and claimed to be the result of resolving/propagating several clauses in the given order. The list of antecedent clause ids forms a justification and is part of such an addition step in LRAT. In the rest of the paper we focus on finding these justification.

3 Implementation

The LRAT extension to CaDiCaL was implemented by the first author as part of his master project and proceeded in four stages: First, the internal proof checker in CaDiCaL for DRAT clauses was extended to produce LRAT proofs, which is quite inefficient but can still be enabled through the `--lrat-external` option. Second, a separate internal LRAT checker was added to CaDiCaL to validate proofs on-the-fly while running the solver. Third, we implemented LRAT production for CaDiCaL without any inprocessing. Finally, all different inprocessing techniques were instrumented to generate LRAT proof chains directly. Thanks to the second stage, proofs could be validated on-the-fly, dramatically reducing the implementation effort (particularly for debugging). The implementation of these four stages took around two months in total but the last two stages only two weeks.

The resolution chain for justifying a new clause can be computed alongside normal CDCL search with little computational overhead but clause minimization and shrinking are a bit more involved (Section 3.1). Proof production in preprocessing and inprocessing were of varying degree of difficulty. The most interesting inprocessing technique from this point of view is equivalent literal substitution which we discuss in Section 3.2.

3.1 Conflict Analysis

Most clauses derived by a SAT solver originate from clauses learned during conflict analysis. When the solver finds a mismatch between the current partial assignment and the clauses, i.e., a conflicting clause which is falsified, then this conflict is analyzed and a clause is learned which forces the solver to adjust the partial assignment. In the standard implementation of conflict analysis the learned clause is derived by resolving individual reason clauses in reverse assignment order, starting with the conflicting clause, which in turn immediately gives the necessary justification for the (non-minimized first UIP [24]) learned clause.

We have adapted our code to generate chains for various technique relying on conflict analysis such as hyper binary resolution [13] and vivification [17]. It is crucial to distinguish between techniques that eliminate false literals (thus, necessitating an extension of the proof chain) and those that do not.

One recent addition to improve conflict analysis is the concept of "shrinking" [9,10] which can be interpreted as a more advanced version of "minimization" [8]. Minimization only

removes literals from the learned clause following resolution paths in the implication graph, but does not add any literals. The additional idea in shrinking is to continue trying to resolve literals on a particular decision level until all but one (the first UIP on that level) is left, however, without being allowed to add literals from a lower decision level.

Our approach differs from the FRAT flow [1]. Their solver performs a post-process analysis of the final learned clause $C_{mini+shrink}$ to rediscover the necessary propagation by traversing the implication graph, which repeats conflict analysis work. In contrast, we split the justification process into two parts. First, we derive the justification for the clause C_{UIP} alongside conflict analysis with little to no overhead. Then, we derive the missing resolution steps between C_{UIP} and the shrunken and minimized clause $C_{mini+shrink}$ as a post-process analysis. We identify literals that differ and add the required reason clauses. Although we still traverse parts of the implication graph, we avoid repeating the conflict analysis.

Our Algorithm 1 shows the postprocessing step only. The first step has already derived the justification $Chain_{UIP}$ for the first UIP clause $C_{original}$ from conflict analysis. Our postprocessing step calculates the justification chain in $Chain_{mini+shrink}$. For each removed literal L (in $C_{original}$ but not in $C_{shrunken}$), we extend the chain with additional justification steps (Line 3).

The function calculate_LRAT_Chain(L) (Line 5) extends the chains with the required reason and preserves the resolution order. It goes recursively over all literals of the reasons and extends the chain with the reason. If the function reaches a previously used reason (*already_added*), it can stop the analysis to avoid duplicated reasons in the chain. Our calculation stops when we reach literals that appear in $C_{shrunken}$ ($L \notin Chain_{new}$). After calculating the justification chain for minimization and shrink, we merge the two chains $Chain_{UIP}$ and $Chain_{new}$ (Line 4). Starting with an empty chain provides a valid proof when removing unit literals during both phases.

Data: currently build LRAT chain $Chain_{UIP}$
Data: the clause before $C_{original}$ and after minimization and shrinking $C_{shrunken}$
Result: resulting LRAT chain $Chain_{full}$

```
1  foreach literal L in C_original do
2      if L not in C_shrunken then
3          calculate_LRAT_Chain(L)

4  Chain_full := Chain_mini+shrink + Chain_UIP

5  calculate_LRAT_Chain (Literal K)
6      C := reason of K in the current assignment
7      foreach Literal L in C different from K do
8          already_added := reason of L in Chain_mini+shrink
9          if ¬already_added and L ∉ C_shrunken then
10             calculate_LRAT_Chain(L)
11     append C to Chain_mini+shrink
```

Algorithm 1 Recursively calculating the prefix LRAT chain for shrinking and minimizing.

Our approach can potentially lead to duplicated unit clauses: We add unit clauses to the chain during conflict analysis. We can guarantee no duplicates here, but the same unit clause might also be added during post process analysis, which means it is actually needed earlier in $Chain_{mini+shrink}$ and we could remove it from $Chain_{UIP}$. Note that this cannot happen for larger clauses since they can appear at most once as a reason for some assignment. Since removing these unit clauses afterwards would be rather costly, we actually collect unit clauses separately and put them at the start of the merged chain after the post process analysis for $C_{shrunken}$ is finished. Like this, we can avoid duplicates and still get a correct justification chain for $C_{shrunken}$.

3.2 Equivalence Literal Substitution

While the justification process for clauses derived during variable elimination and other preprocessing techniques that rely on propagation and conflict analysis is similar to normal learning, producing LRAT proof justifications for equivalent literal substitution [5] is more involved.

Equivalent literal substitution detects and replaces equivalent literals by a chosen representative. For example, if the problem includes the three clauses $(\neg A \vee B)$, $(\neg B \vee C)$ and $(\neg C \vee A)$ we know that A, B and C are equivalent and we can replace all occurrences of either literal by one of the others. As is common we use Tarjan's algorithm [22] to detect cycles in the graph spanned by the binary clauses (i.e., the binary implication graph) and fix a representative for each cycle [5]. In the DRAT proof we can simply dump all changed clauses and delete the old ones.

For LRAT we have to produce the resolution chains. After fixing representatives, proof chains have to be produced for every changed clause separately. We derive the justification for each changed or removed literal, similarly as for the shrunken clause in conflict analysis 3.1.

Fixing the representative is a rather arbitrary choice (the smallest absolute value in this implementation). We considered changing this to the first visited literal during DFS in Tarjan's Algorithm, in order to allow reusing some computation and potentially shorten proofs, but in the end decided against changing solver behavior.

4 Trimming LRAT proofs

In preliminary experiments we observed that the FRAT flow [1] produced significantly smaller proofs. FRAT-RS trims the proof during translation to LRAT, i.e., it omits clauses that are not needed to derive the empty clause, allowing for much more efficient proof checking. We concluded that we needed a tool to do such trimming on LRAT directly in order to obtain an efficient pure LRAT proof generation and checking flow.

Even though trimming is effective, it is not obvious how to cheaply achieve such reduction for DRAT proofs because dependencies between proof steps are lacking. Luckily, in LRAT these dependencies are explicit. Therefore we implemented LRAT-TRIM [2], an open-source LRAT proof trimming and checking tool. It often reduces proofs by a factor of 2 to 3, again emphasizing how many useless clauses a SAT solver actually derives during search.

Trimming LRAT proofs consists of a backward reachability analysis starting from the empty clause towards the clauses of the original CNF, marking reached clauses as needed. Clauses unmarked after this traversal are redundant and can be trimmed. This algorithm is implemented by depth first search (DFS) along antecedent clauses in justification chains.

It also determines the last usage of each clause ID and remaps original clause ids to a consecutive ID range. On completion we can dump the proofs back to a file in a forward

manner, only writing needed clauses and their antecedents and skipping redundant clauses. While doing this we can eagerly mark clauses once they are not used anymore.

Before starting to write proof lines, we check whether there are redundant original clauses and if so write a single deletion line with all unused original clause ids. This minimizes the life-span of clauses in the trimmed LRAT proof, both for added and original clauses. Note that LRAT-TRIM, in contrast to DRAT-Trim, does not require access to the original CNF nor looks at literals of clauses to trim proofs.

We also implemented a checking mode in LRAT-TRIM which, given the original CNF and an LRAT proof, checks that the resolution chains of added clauses can be resolved to produce the claimed clauses. It also checks that clauses are not used after they are deleted in a deletion step. This checking mode comes in two flavors. The default is to first trim the clauses with the trimming algorithm described above and only check needed clauses. Alternatively LRAT-TRIM supports forward checking, which checks added clauses on-the-fly during parsing and in particular allows to delete clauses in deletion steps eagerly.

On the one hand, forward checking reduces maximum memory usage to at most that of the solving process, whereas backward checking needs to keep the whole proof in memory which is usually much more than maximum usage during solving. On the other hand, forward checking substantially increases checking time, as all clauses have to be checked without trimming information, irrespective of being needed or not.

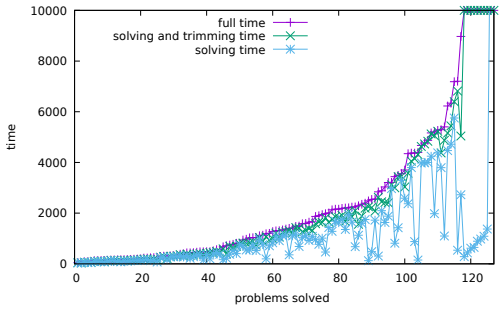
During the development of LRAT-TRIM substantial effort went into making parsing as fast and robust as possible and also provide meaningful error messages during parsing and checking. The parsing code amounts to roughly 900 lines of C code out of 2400 lines for the whole tool (including comments but formatted with CLANGFORMAT).

All three proof formats (DRAT, FRAT and LRAT) have a binary version. We implemented the binary format for LRAT (both in CADICAL and in LRAT-TRIM) which is only supported by CLRAT [7], a formally verified checker for LRAT using ACL2. We are grateful to Peter Lammich who provided us a tool that converts LRAT proofs (with some extra requirements on proofs) to GRAT [16] that his checker can check. However, GRAT is stricter as duplicate or extraneous ids are not allowed. We leave it to future work to produce stricter proofs.

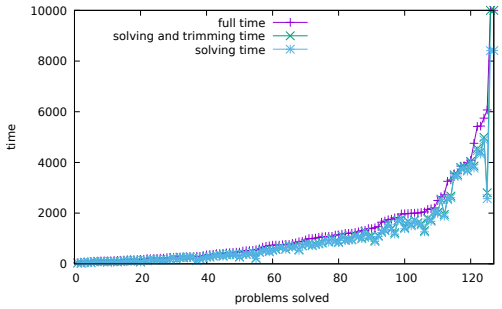
5 Experiments

While checking for our extensions not to change solver behavior with and without proof generation, i.e., validating (C), we realized that two changes to the solver became necessary. First, scheduling of garbage collection during bounded-variable elimination depends on the number of bytes allocated for clauses, which changed with LRAT proof generation, as clauses require an ID and thus became larger. Therefore, our CADICAL extension always uses clause ids, which is not expected to have major impact on performance nor memory usage. The second change is due to the way conflicts were derived in equivalent literal detection. Originally detection was aborted on such a conflict, which we now simply delay until detection finishes. Then the conflicting literal is propagated to yield a proper LRAT proof.

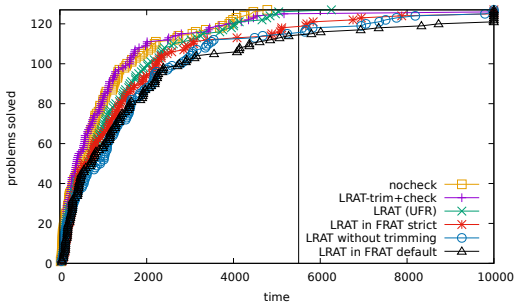
Our goal (A) of being able to always generate correct proofs was tested by intensive fuzzing of our solver, proof generation, and proof checking. We attempted to apply the same approach to the FRAT extension of CADICAL [1] but immediately experienced failing proofs, due to several reasons, particularly with respect to handling unit clauses in the input CNF. We also observed that chains often listed the same clause id multiple times. Reducing these occurrences might lead to a substantial speedup, since justifying one literal can pull in several more clauses (e.g., if some of the literals have been removed by minimization).



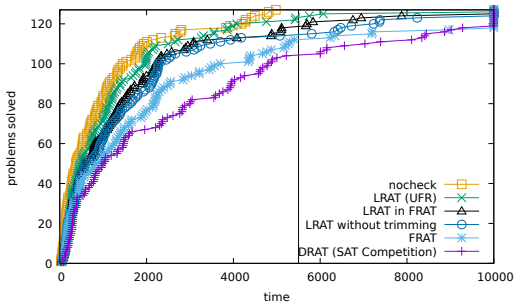
(a) Checking FRAT proofs of our new CaDiCaL version 1.5.1 from UFR but in the configuration of CaDiCaL version 1.2.1 used in [1].



(b) Checking LRAT proofs of our new CaDiCaL version 1.5.1 from UFR but in the configuration of CaDiCaL version 1.2.1 used in [1].



(c) Comparing trimmers Lrat-Trim vs. FRAT-rs (LRAT in FRAT) using CaDiCaL 1.5.1 proofs before checking with Cake_Lpr, except for the run "LRAT-trim+check" which checks with Lrat-Trim.



(d) CDF of all the solving and checking flows with the vertical black line indicating the 5,000 seconds timeout used for the solver, showing that our flow is the fastest with fewer timeouts.

Figure 1 Performance on unsatisfiable instances from the SAT Competition 2022.

After fuzzing, we ran our LRAT flow on the problems of the SAT Competition 2022 and found three issues: (*i*) Cake_Lpr did not accept some input files, because they contained trailing empty lines, which we then removed manually; (*ii*) Cake_Lpr requires a very large amount of memory (around the size of the proof file); (*iii*) one node of the cluster showed irregular behavior, when many proofs were written to the temporary disk at the same time, which lead to corrupted proof files resulting in an Lrat-Trim error. Reducing the number of jobs per node fixed this issue and we did not discover any further problem with the generated proofs, validating (A) and showing again the effectiveness of fuzzing.

To compare performance, i.e., showing that we achieved (B), of our extended version to the base version of CaDiCaL (added clause ids taking up space without being used), we let both versions write generated proofs to `/dev/null` in order to ensure that we do not introduce any bias due to file I/O limits as LRAT proofs exceed DRAT proofs in size substantially. This yielded an average overhead of 5% for our new LRAT proof production versus DRAT in base CaDiCaL.

For the remaining empirical analysis we have chosen to focus on the 127 benchmarks from the SAT Competition 2022, which were shown to be unsatisfiable during the competition. First, we tried to determine how much proofs can be reduced with our new tool Lrat-Trim. It turns out, that some proofs were reduced to *one percent*, i.e., 99% of the output is not useful for deriving the contradiction. These problems stem from the `sudoku-N30` family. In other proofs 80% and more clauses are needed – most of these problems have a short runtime

(around 200 s), contain a large amount of fixed variables and accordingly many clauses are simplified by removing these units, where each removal contributes a proof step.

In order to determine the performance of our new solving and checking flow, we compared the following three workflows: (*i*) the (competition) DRAT workflow, i.e., generating the DRAT proof, converting it to LRAT with DRAT-Trim, then checking that proof; (*ii*) the FRAT workflow, i.e., generating the FRAT proof, converting it to LRAT with FRAT-rs, then checking it; (*iii*) our new LRAT flow including generating, trimming, and checking the proof. All workflows use binary proof formats, except for feeding Cake_Lpr at the end.

We also ported the FRAT extensions [1] to the newest CaDiCaL version, but did not try to fix any issues. Nevertheless, we ran the ported version (see Figure 1a) which is now able to use the latest heuristics used in CaDiCaL, except for shrinking which had to be deactivated as it is not supported by the original FRAT code [1].

The first observation we can make is that the overhead of trimming and proof checking is quite consistent among our configurations, but wildly differs for FRAT: If many clauses without justification are used for the proof, the translation needs a lot of search – although, as expected, less than using the conversion to DRAT(see Figure 5a in appendix).

To our surprise, we observed several timeouts though. They all seem to origin from one family submitted by AWS in 2022, where solving took less than 600 s, but elaboration (translation) never finishes. In comparison, DRAT-Trim also needs a very long time (6 000 s), but stays well below the time limit. It is unclear what the problem is and thus we tested one instance `aws-c-common:aws_priority_queue_s_sift_either` on a (twice as fast) computer where it took nearly 10 h to convert the 400 MB FRAT proof to a 3.8 GB LRAT proof. We have reported the issue on GitHub,[1] but have not heard back yet.

A comparison of Lrat-Trim with FRAT-rs in both *normal mode* and *super strict mode* is shown in Figure 1c. We used the feature of our extended version of CaDiCaL to generate proofs both in LRAT and in FRAT, where in FRAT, every step is properly justified. The results show that Lrat-Trim scales much better than FRAT-rs, although there was a bug which we reported that made FRAT-rs significantly slower when not using the *super strict mode*. Furthermore, Lrat-Trim can also check proofs directly and it turns out that the additional overhead of this (untrusted) checking compared to parsing and trimming is small.

Overall, our new LRAT proof flow performs best, with reasonably small overhead on solving. To ease visual comparison, we printed all different configurations into a single graph (Figure 1d). The fastest option is (of course) "no-checking" but our new method is not too far behind. Figure 2 shows that the overhead (cost) of proof checking compared to not checking any proofs. Our approach performs best taking only 30% more time than pure solving. The existing competing approaches are much slower with DRAT incurring an overhead of 180% and FRAT still requiring 125% more time than solving, i.e., both more than doubling overall certification time, while our approach has faster checking than solving.

As a sanity check, we also tested our LRAT proof flow using the default shrinking (see Fig 3). We observed that our new approach remains faster compared to the FRAT proof flow, confirming our initial findings.

6 Conclusion

We have implemented native LRAT proof production in our SAT solver CaDiCaL. Even though direct production of LRAT proofs slows down the solver slightly this loss is by far

[1] `https://github.com/digama0/frat/issues/18`

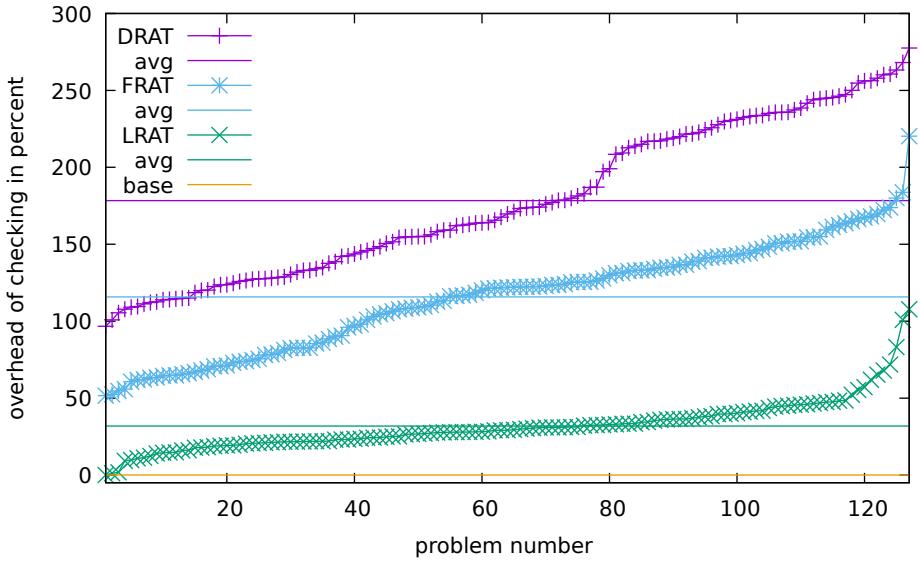


Figure 2 Overhead of the whole checking flow using FRAT, DRAT and our new LRAT flow on top of plain solving (without proof generation and checking), with averages shown as horizontal lines (LRAT 30% overhead, FRAT 125% and DRAT 180% overhead).

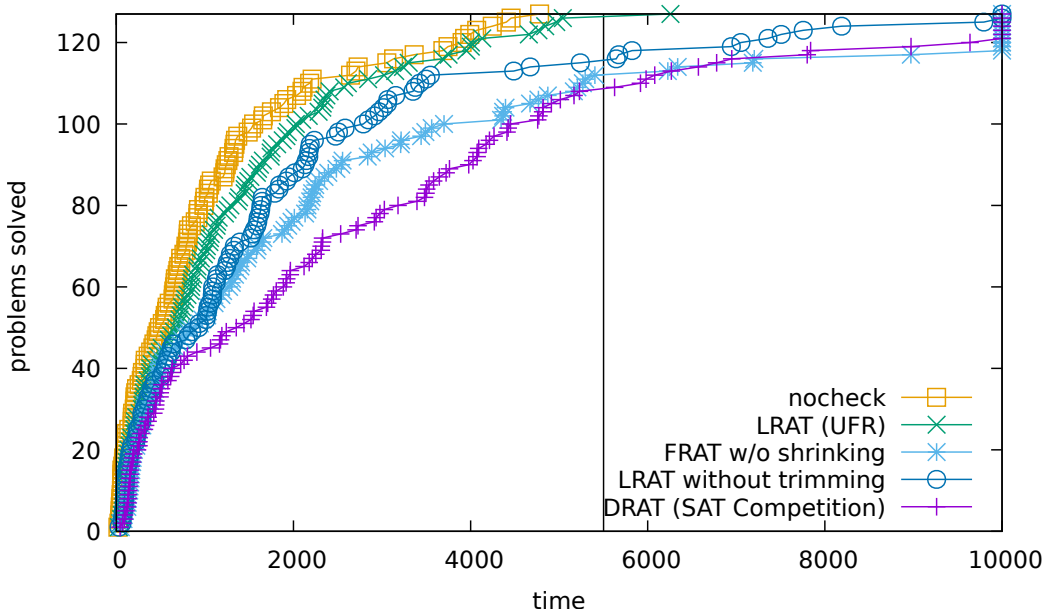


Figure 3 CDF all methods with vertical line indicating timeout of the solver with default options (i.e., with shrinking not supported by the CaDiCaL).

offset by the reduction in proof checking time, both compared to DRAT and FRAT proofs. At the end our certification flow adds only 30% overhead compared to pure solving while other approaches take more than twice the time for certification.

It might be interesting to apply this work to recent results on distributed proof generation in the context of the cloud solver MALLOB [18] as well as our multi-core solver in GIMSATUL [11]. We also see the question of how to handle clause ids for virtual binary clauses as a technical challenge. Such clauses occur in both GIMSATUL [11] and the state-of-the-art sequential solver KISSAT [3].

References

1. Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/lmcs-18(2:3)2022.
2. Armin Biere. Lrat trimmer, Last access, March 2023. Source code. URL: https://github.com/arminbiere/lrat-trim.
3. Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.
4. Armin Biere, Mathias Fleury, and Mathias Heisinger. CADICAL, KISSAT, PARACOOBA entering the SAT Competition 2021. In Marijn J. H. Heule, Matti Järvisalo, and Martin Suda, editors, *SAT Competition 2021*, 2021.
5. Armin Biere, Matti Järvisalo, and Bejamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391 – 435. IOS Press, 2nd edition edition, 2021.
6. Luís Cruz-Filipe, Marijn J. H. Heule, Jr. Hunt, Warren A., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.
7. Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.
8. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
9. Nick Feng and Fahiem Bacchus. Clause size reduction with all-UIP learning. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 28–45. Springer, 2020. doi:10.1007/978-3-030-51825-7_3.
10. Mathias Fleury and Armin Biere. Efficient All-UIP learned clause minimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2021. doi:10.1007/978-3-030-80223-3_12.
11. Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. *CoRR*, abs/2207.13577, 2022. arXiv:2207.13577, doi:10.48550/arXiv.2207.13577.

12 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.

13 Marijn Heule, Matti Järvisalo, and Armin Biere. Revisiting hyper binary resolution. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2013. doi:10.1007/978-3-642-38171-3_6.

14 Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010. doi:10.1007/978-3-642-16242-8_26.

15 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

16 Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi:10.1007/s10817-019-09525-z.

17 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020. doi:10.1016/j.artint.2019.103197.

18 Dawn Michaelson, Dominik Schreiber, Marijn J. Heule Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Unsatisfiability proofs for distributed clause-sharing SAT solvers. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2023*, Lecture Notes in Computer Science, 2023. Accepted, to appear.

19 Florian Pollitt, Mathias Fleury, and Armin Biere. Efficient proof checking with lrat in cadical (work in progress). In Armin Biere and Daniel Große, editors, *24th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*, pages 64–67. VDE, 2023. Accepted. URL: https://cca.informatik.uni-freiburg.de/papers/PolittFleuryBiere-MBMV23.pdf.

20 Florian Pollitt, Mathias Fleury, and Armin Biere. Native LRAT in CaDiCaL for faster proof checking, 2023. URL: https://cca.informatik.uni-freiburg.de/lrat.

21 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in cakeml. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2021. doi:10.1007/978-3-030-72013-1_12.

22 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

23 Nathan Wetzler, Marijn J. H. Heule, and Jr. Hunt, Warren A. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

24 Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of*

418 *the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001,*
419 *San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001.
420 doi:10.1109/ICCAD.2001.968634.

A More experiments

In this appendix we present some additional plots which did not fit into the main part of the paper, might shed more light on some of the experimental data, but we do not consider to be essential.

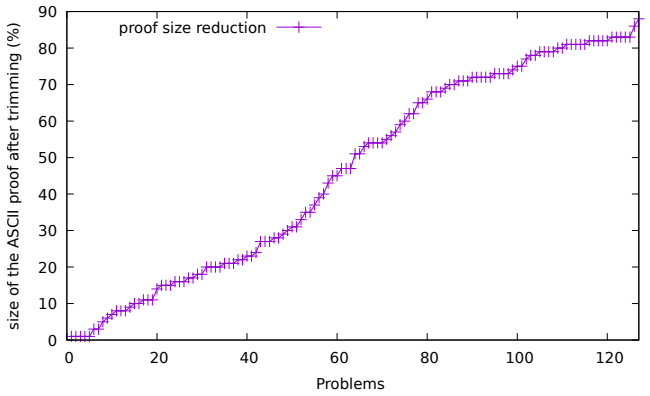
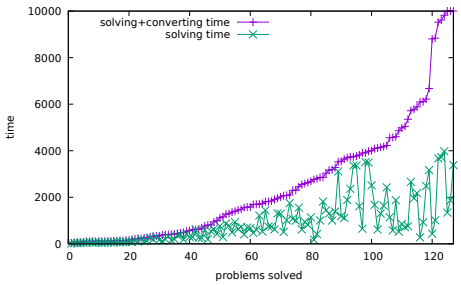
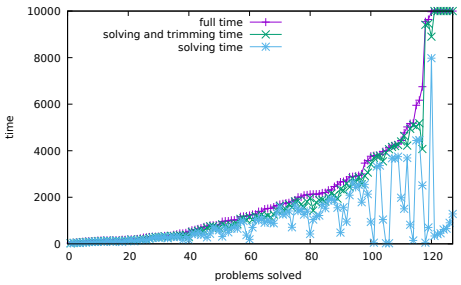


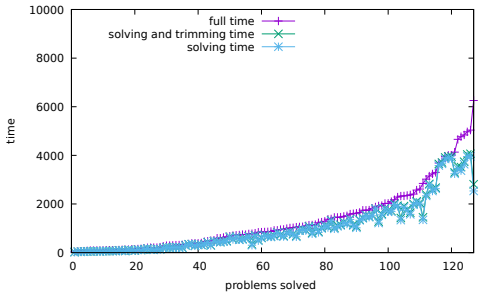
Figure 4 Trimming reduction of the ASCII proof files on solved unsatisfiable instances from the SAT Competition 2022.



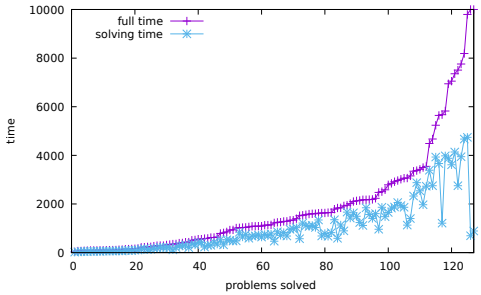
(a) DRAT conversion to LRAT (without full verification)



(b) Original FRAT version in CaDiCaL 1.2.1 (compared to our ported version in Figure 1a)



(c) Direct LRAT production with CaDiCaL 1.5.1



(d) LRAT with CaDiCaL 1.5.1 without trimming the proof with checking

Figure 5 Performance on the same set of problems of the SAT Competition 2022 (not all of them are solved), showing that the verification part is not the most critical.

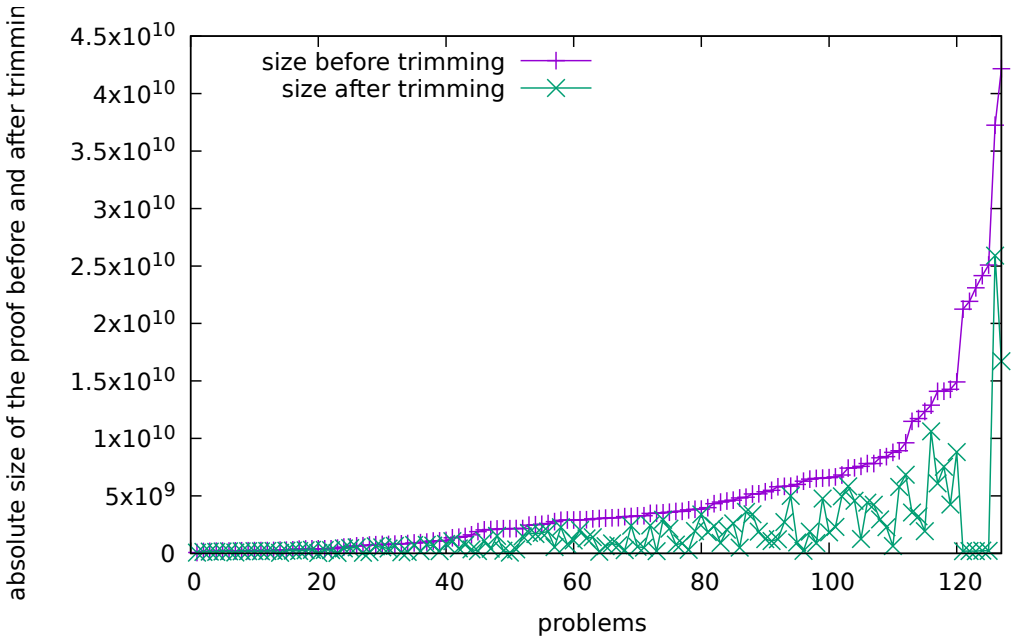


Figure 6 Size of the proofs in Bytes.

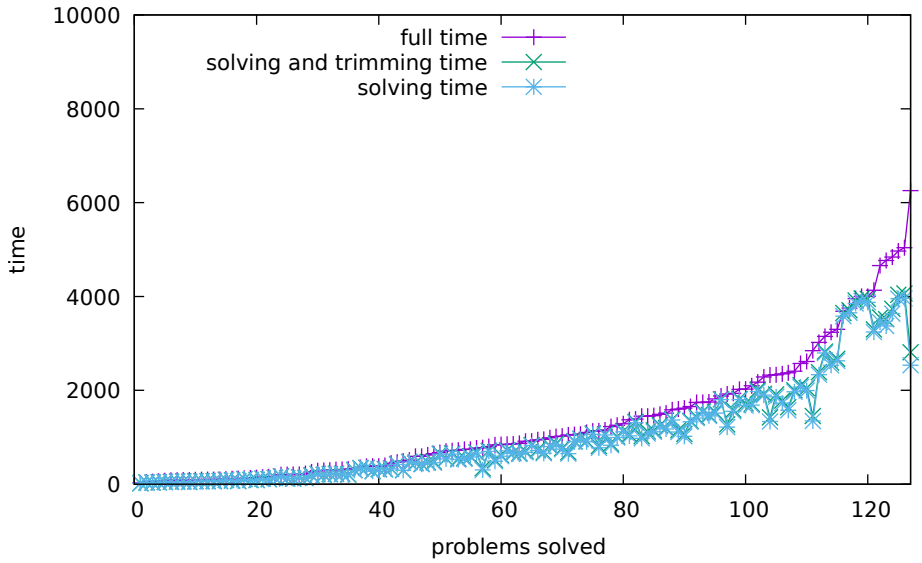


Figure 7 CaDiCaL 1.5.1 on 127 solved unsatisfiable instances of SAT Competition 2022.