




Migrating Solver State

Armin Biere   

University of Freiburg, Germany

Md Solimul Chowdhury   

Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule   

Carnegie Mellon University, Pittsburgh, PA, USA

Amazon Web Services, Inc., Pittsburgh, PA, USA

Benjamin Kiesl   

Amazon Web Services, Inc., Munich, Germany

Michael W. Whalen   

Amazon Web Services, Inc., Minneapolis, MN, USA

The University of Minnesota, Minneapolis, MN, USA

Abstract

We present approaches to store and restore the state of a SAT solver, allowing us to migrate the state between different compute resources, or even between different solvers. This can be used in many ways, e.g., to improve the fault tolerance of solvers, to schedule SAT problems on a restricted number of cores, or to use dedicated preprocessing tools for inprocessing. We identify a minimum viable subset of the solver state to migrate such that the loss of performance is small. We then present and implement two different approaches to state migration: one approach stores the state at the end of a solver run whereas the other approach stores the state continuously as part of the proof trace. We show that our approaches enable the generation of correct models and valid unsatisfiability proofs. Experimental results confirm that the overhead is reasonable and that in several cases solver performance actually improves.

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, SMT, Cloud Computing, Serverless Computing

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.27

Supplementary Material *Software (Source Code)*: <https://github.com/amazon-research/cadical>

Funding *Md Solimul Chowdhury*: partially supported by NSF grant CCF-2015445.

Marijn J. H. Heule: partially supported by NSF grant CCF-2015445.

1 Introduction

Satisfiability solvers are powerful tools that are used in a wide range of applications, including hardware and software verification [15, 33]. When used in practice, the runtime of solvers can vary significantly: while for some applications solvers take just a few milliseconds, for others they require large amounts of time, often several hours or even days. Especially in the latter case, being able to stop a solver and resume its computation at a later point – possibly even on different hardware – opens up multiple opportunities.

For example, a user performing long-running SAT jobs on their computer can benefit from the ability to resume a job at any time (instead of having to start from scratch) in case it failed, e.g., due to power outage or other kinds of hardware failure. Similarly, in a cloud environment where software must be resilient against hardware failures, and where the runtime of jobs is often restricted (e.g., in serverless environments such as AWS Lambda [3]), being able to migrate the state of a solver to a different compute architecture enables significant flexibility in architecting and hosting SAT solvers.



© Armin Biere, Md Solimul Chowdhury, Marijn J. H. Heule, Benjamin Kiesl, and Michael W. Whalen; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022).

Editors: Kuldeep S. Meel and Ofer Strichman; Article No. 27; pp. 27:1–27:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

But better fault-tolerance and increased flexibility are not the only benefits of migrating solver state. If the state is stored in a solver-agnostic way, we can combine multiple solvers to solve a single problem, e.g., by having them take turns. Finally, another possible application is the efficient scheduling of parallel SAT jobs on restricted hardware by assigning (multiple) limited time intervals to each job: this reduces resource contention by preventing hard jobs from using up all resources while still allowing them to make progress.

The above are just a few of the many opportunities offered by state migration. In this paper, we therefore propose effective mechanisms for storing and restoring (migrating) solver state. There are a variety of possible implementation mechanisms that one can pursue to implement state migration, including OS- and VM-level techniques for freezing a solver process, extensions to solvers to store all or fragments of the solver state, and reconstruction techniques based on the proof trace produced by the solver. We examine the range of possible mechanisms and focus on two approaches.

The first approach stores a minimal representation of the solver state: the redundant and irredundant clauses as well as the reconstruction stack necessary to reconstruct a satisfying assignment. The second approach involves restoring the solver state directly from the proof log. In order to restore the state, we describe extensions to the commonly-used DRAT proof format [36], called *Dual DRAT* (DDRAT), that enable solution reconstruction and soundness for both SAT and UNSAT results. With this extended proof format, there does not need to be an explicit state serialization – the proof functions as a write-ahead log of the state, making it resilient to compute-resource failures and process terminations. It also has potential benefits for debugging solvers by pinpointing where a proof (of either satisfiability or unsatisfiability) fails. We implement the first approach on top of the solver CaDiCaL [8], and both approaches on top of MapleSat [39] (the distribution Maple_LCM_Dist_ChronoBT, which won the main track of the SAT Competition 2018). As MapleSat is an advanced descendant of MiniSat [17], our approaches can be implemented analogously on top of well-known related solvers (e.g., MiniSat and variants of MapleSat, Glucose [1], or MergeSat [41]).

We conduct an evaluation of our approaches from the perspective of efficiency, considering the time to store and restore state as well as the overall effect on solving time and number of solved problems. We also examine both approaches in terms of ease of implementation and integration with MapleSat and CaDiCaL. We find that the store/restore mechanisms perform slightly differently on each solver: For MapleSat, both approaches can lead to a degradation of performance whereas for CaDiCaL, migrating the state leads to a surprising performance *improvement*. Even though we did not concern ourselves with improving solver efficiency, we believe our experiments reveal that “hard restarts”, where a solver forgets everything except its current clause sets, can lead to a significant performance increase.

The main contributions of this paper are as follows:

- Two solver-agnostic approaches to storing and restoring solver state that allow greater flexibility in hosting, migrating, and combining SAT solvers.
- An extension of the DRAT proof format, called Dual DRAT, that allows the extraction of solver state from proofs, with only a small overhead over the standard DRAT format.
- An evaluation of both techniques against monolithic solvers on a range of benchmarks.
- Empirical evidence that hard restarts can improve solver performance significantly.

The rest of the paper is organized as follows. In Section 2, we present the background required to understand our paper and discuss related work. In Section 3, we outline the problems involved with migrating solver state and overview possible solutions. We present our concrete approaches for solver-state migration in detail in Sections 4 and 5. In Section 6,

we present an empirical evaluation of our approaches before concluding with final remarks and an outlook for future work in Section 7. Supplemental material accompanying this paper can be found at <https://github.com/amazon-research/cadical>.

2 Background and Related Work

Propositional satisfiability has been studied for many years; for a comprehensive overview, see [11]. We first present the basics and afterwards the most important related work.

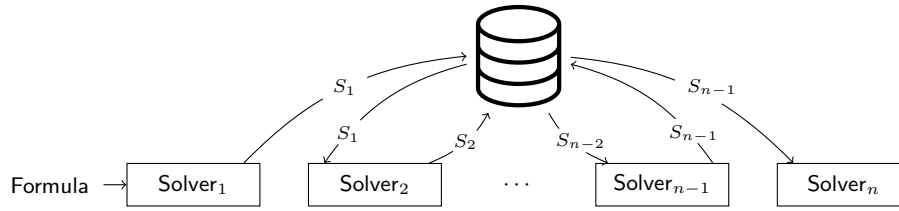
We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation \bar{x} of a variable x (a *negative literal*). The *complement* \bar{l} of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and as $\bar{l} = x$ if $l = \bar{x}$. For a literal l , we denote the variable of l by $\text{var}(l)$. A *clause* is a finite disjunction of the form $(l_1 \vee \dots \vee l_n)$ where l_1, \dots, l_n are literals. A *formula* is a finite conjunction of the form $C_1 \wedge \dots \wedge C_m$ where C_1, \dots, C_m are clauses. For example, $(x \vee \bar{y}) \wedge (z) \wedge (\bar{x} \vee \bar{z})$ is a formula consisting of the clauses $(x \vee \bar{y})$, (z) , and $(\bar{x} \vee \bar{z})$. Formulas can be viewed as sets of clauses, and clauses can be viewed as sets of literals.

A *truth assignment* (or *assignment* for short) is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). A literal l is *satisfied* by an assignment α if l is positive and $\alpha(\text{var}(l)) = 1$ or if l is negative and $\alpha(\text{var}(l)) = 0$. A literal l is *falsified* by an assignment if its complement \bar{l} is satisfied by the assignment. A clause C is satisfied by an assignment α if α satisfies at least one of C 's literals. A formula F is satisfied by an assignment α if α satisfies all of F 's clauses, in which case α is a *model* of F . Given two assignments α and ω , the composition $\alpha \circ \omega$ of α and ω is defined as $\alpha \circ \omega(x) = \omega(x)$ if ω assigns a truth value to x and $\alpha \circ \omega(x) = \alpha(x)$ otherwise. We sometimes denote assignments by the sequences of literals they satisfy, i.e., we write $x\bar{y}$ to denote the assignment that assigns 1 to x and 0 to y .

A formula is *satisfiable* if there exists an assignment that satisfies it, otherwise it is *unsatisfiable*. Two formulas are *logically equivalent* if they are satisfied by the same assignments; they are *equisatisfiable* if they are either both satisfiable or both unsatisfiable. A *SAT solver* is a computer program that takes as input a propositional formula and decides whether or not the formula is satisfiable. In case the formula is satisfiable, a solver can output a satisfying assignment; in case the formula is unsatisfiable, a solver can output a proof of unsatisfiability. We discuss proofs in more detail in Section 5.

There are two main techniques for parallelizing SAT problems: *portfolio-based* approaches and *divide-and-conquer* approaches. In portfolio-based approaches (c.f. [2, 7, 9, 20, 21, 27, 38, 45, 48]), multiple solvers or solver configurations all attempt to solve the same problem, and the portfolio completes as soon as one of the solvers finishes the problem. Some portfolio solvers use *clause sharing* to share conflict clauses between the solvers, allowing progress from one solver to be used by other solvers within the portfolio. Sharing conflict clauses involves similar engineering to the state-migration techniques described in this paper. The winner of the last two SAT-competition cloud tracks [5, 6], `mallob` [44], uses this approach. Portfolio solvers have also been used to parallelize SMT [14, 31, 47].

Divide-and-conquer approaches partition the search space into multiple – sometimes millions or even billions of – subproblems [23, 27, 30, 42]. A particularly successful approach is *cube-and-conquer*, which uses look-ahead techniques to partition the search space and then solves each of the subproblems using CDCL [27]. For various hard combinatorial problems [24, 26], cube-and-conquer can realize linear-time speedups, even when running on thousands of cores. Such scaling cannot be achieved with portfolio-based approaches. Some techniques combine divide-and-conquer with portfolio/clause sharing, e.g. [32].



■ **Figure 1** Storing and restoring solver state in external storage while solving a single SAT problem.

Recent work has examined how to use cloud resources and serverless computing to scale SAT solving. Ozdemir et al. [43] examine the use of divide-and-conquer parallel algorithms on AWS Lambda in a tool called *gg-SAT*. When a Lambda node approaches a timeout, it splits the problem to run on several more Lambda nodes (the timeout and the splitting factor are configurable). Solver state is discarded at the timeout, so the approach is orthogonal to our idea of migrating state; it may be possible to combine both in the future. This work is in turn built on *gg* [19], a tool for process management and I/O for serverless computing.

Our approaches increase the complexity of the solver. To ensure correctness, we support proof logging in an extension of the commonly-used DRAT format [29, 36, 46]. We use two extensions. The first adds information to the proof to validate the results on satisfiable formulas by producing solutions for the original formula, similar to solution reconstruction [18, 36]. This extension is similar to *dual proofs* for quantified Boolean formulas [13, 28]. The second extension adds weight to clauses in the proof. With this information it is possible to identify and extract the most performance-critical information when migrating state via proofs. Both extensions can also easily be used for stronger proof systems than DRAT [25].

3 Migrating Solver State

Our goal is to solve a given SAT problem over multiple solver runs, potentially involving multiple different solvers on multiple compute architectures. For example, we want to run a solver on a problem until a timeout is reached; at that point, we terminate the solver and start another solver – possibly on a different machine – that continues solving the problem. In case the second solver finishes before its timeout, we return a solution, otherwise we continue by starting yet another solver run, and we keep doing so until the formula is eventually solved. The main problem we need to address in order to reach our goal is to *migrate* the state of a SAT solver, i.e., to serialize it and store it such that it can be subsequently re-used to continue solving, as shown in Figure 1.

3.1 Which State to Migrate?

A naïve approach to migrating solver state would be to store the complete state of a solver process, including the stack, heap, etc. This is known as *application checkpointing*, and while there have been attempts to implement application-independent solutions (like *Berkeley Lab Checkpoint/Restart for LINUX* [22]), the approach depends strongly on the underlying compute architecture, making it a suboptimal choice on generic cloud infrastructure.

We want an infrastructure-independent solution that achieves sound and efficient solver performance across multiple runs. Consider two contrasting approaches:

1. Store virtually all internal data structures of a solver.
2. Store only a “minimum viable subset” of the data structures.

```

vector<Flags> ftab;           // variable and literal flags
vector<int64_t> btab;        // enqueue time stamps for queue
vector<int64_t> gtab;        // time stamp table to recompute glue
vector<Occc> otab;          // table of occurrences for all literals
vector<int> ptab;            // table for caching probing attempts
vector<int64_t> ntab;        // number of one-sided occurrences table
vector<Bins> big;           // binary implication graph
vector<Watches> wtab;       // table of watches for all literals

```

■ **Figure 2** An excerpt from the state declarations inside CaDiCaL’s source code.

The first approach stores not only the obvious pieces of state such as clause sets, scores of heuristics, and phases, but a multitude of other data structures maintained by a solver. This has the advantage that a subsequent solver loading this information can continue exactly where the previous solver left off. However, modern solvers are immensely complex; for example, in the case of CaDiCaL, the internal state involves more than 90 interdependent data structures (an excerpt is shown in Figure 2) that must be properly serialized. This approach has substantial disadvantages: it is solver-specific, brittle to data-structure changes, requires a maximal amount of storage and time to store and restore solver data structures, and adds solver maintenance overhead for any change to the data structures.

As an alternative, we consider approaches that store only a small fragment of the entire solver state. In the next section, we define a “minimum viable subset” of solver state (we do not use the term *minimal* in a formal sense but rather in a vague sense of “practicality”); the resulting approach is not only almost solver independent but also achieves surprisingly good performance. In addition, as this subset of state is so succinct, we do not necessarily need to store it explicitly at the end of a solver run. In Section 5, we show how a small modification of the DRAT proof format allows a solver to store it continuously in its proof trace.

4 Migrating Solver State by Explicitly Writing Solver State

In this section, we describe the fragment of solver state necessary to achieve soundness and acceptable performance over multiple solver runs, and we discuss practical considerations for how it can be stored and restored. This fragment must obviously include the current clause set maintained by the solver. However, if we are not only interested in a simple SAT/UNSAT answer but also in a satisfying assignment (in case of SAT), storing *only* the clause set is insufficient to achieve soundness. An additional piece of solver state, the so-called *reconstruction stack*, is required for *solution reconstruction* of SAT results.

4.1 Solution Reconstruction

SAT solvers routinely perform clause-set simplifications that preserve equisatisfiability but not necessarily logical equivalence. Even though these simplifications do not impact the solver’s ultimate verdict on the formula’s satisfiability, they can weaken a formula enough to admit assignments that do not satisfy the original formula. Some solvers, like MiniSat [17], perform such simplifications only at the beginning, as *preprocessing* (see [12] for details), whereas other solvers, like CaDiCaL and Kissat [9], also perform them during solving, as so-called *inprocessing*. Example 1 illustrates the impact of these simplifications on a simple formula.

► **Example 1.** Consider the formula $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{y})$. A solver might conclude that the deletion of the clause $(x \vee y)$ does not affect the satisfiability of the formula and thus remove it (e.g., via the technique of *blocked-clause elimination* [35]). The resulting formula $(\bar{x} \vee \bar{y}) \wedge (\bar{y})$ is satisfied by the assignment $\alpha = \bar{x}\bar{y}$. However, α falsifies $(x \vee y)$ and is thus not a valid model of the original formula.

To deal with this problem, solvers that perform such simplifications must maintain information that allows them to turn a solution of the simplified formula into a valid solution of the original formula. This process of restoring a solution is known as *solution reconstruction*, and the data structure maintaining the required information is called the *reconstruction stack* [34]. Hence, if we aim to solve a formula over multiple solver runs, and if we are interested in a valid satisfying assignment, we also need to store the reconstruction stack.

Formally, the reconstruction stack is a sequence of pairs $\langle C, \omega \rangle$, where C is a clause and ω (called the *witness*) is a set of literals such that $C \cap \omega \neq \emptyset$. Intuitively, the clauses on the stack represent clauses that were deleted from the formula, and the witnesses contain literals that should be made true during solution reconstruction in case their corresponding clause is not satisfied by the current assignment. Algorithm 1 formalizes this intuition [18].

■ **Algorithm 1** Solution-Reconstruction Algorithm.

```

function RECONSTRUCTSOLUTION(assignment  $\alpha$ , reconstruction stack  $\sigma$ )
  while  $\sigma$  is non-empty do
     $\langle C, \omega \rangle \leftarrow \sigma.pop()$            \\ Get the top pair from the stack.
    if  $\alpha$  falsifies  $C$  then  $\alpha \leftarrow \alpha \circ \omega$            \\ Make the literals of  $\omega$  true.

```

► **Example 2.** Consider Example 1 again. When deleting the clause $(x \vee y)$, the solver can push the pair $\langle (x \vee y), \{x\} \rangle$ on the reconstruction stack. When performing solution reconstruction with the assignment $\bar{x}\bar{y}$ and the reconstruction stack $\sigma = \langle (x \vee y), \{x\} \rangle$, it then obtains the assignment $x\bar{y}$, which does indeed satisfy the original formula $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{y})$.

In Example 2, making x true does not falsify the original formula (if it did, solution reconstruction would fail). In practice, solvers use only simplification techniques that ensure this, and there are many subtleties involved with maintaining a correct reconstruction stack. We do not discuss these subtleties here and instead refer to the existing literature [12, 34].

4.2 A Minimum Viable Subset of Solver State

We have seen that in addition to storing the clauses of a solver, storing the reconstruction stack is required for soundness. Considering soundness alone, we could ignore learned/redundant clauses. For efficiency, however, we include them too; but we must make sure that subsequent solver runs can distinguish them from the irredundant clauses. The reason is that CDCL solvers eagerly remove learned clauses, which is unproblematic because they are implied by the irredundant clauses. Removing some learned clauses improves performance by keeping the size of the clause database small. To judge the quality of learned clauses, solvers usually rely on the *literal-block distance* (LBD) [1], so we include the LBDs with the learned clauses. This leaves us with the following *minimum viable subset of solver state*: (1) irredundant clauses, (2) redundant clauses and their LBDs, and (3) the reconstruction stack.

4.3 Migrating Minimum Viable Solver State: Practical Considerations

We implemented dedicated serialization and deserialization routines on top of `CaDiCaL` and `MapleSat`.

At the end of a solver run (when a solver has finished or a time limit has been reached), the solvers write minimum viable state to separate files, one for irredundant clauses, one for redundant clauses, and one for the reconstruction stack. At the beginning of a solver run, they can also accept as input a file with redundant clauses and LBDs in addition to the irredundant clauses. Our implementations can serialize their state in a binary format and in a plain-text format. We define both formats in Appendix A.

Both solvers depend on the invariant that learned clauses do not contain falsified literals (i.e., literals whose complements have been derived as unit clauses, or equivalently, literals that are false at decision level 0). Thus, when adding learned clauses at the beginning of a solver run, we must remove these literals. In the spirit of the robustness principle (often worded as “be conservative in what you send, be liberal in what you accept”), we remove falsified literals both when storing clause sets and when restoring them.

For solution reconstruction, we use a dedicated script that takes a truth assignment (in competition format, as printed by the solvers) and a serialized reconstruction stack, and outputs the assignment obtained by applying Algorithm 1. This allows us to use the reconstruction stacks of previous solver runs to transform an assignment returned by a final solver call into an assignment of the original input formula.

Our experimental evaluation in Section 6 shows that – compared to solving a formula in a single run – this approach leads to mixed impact on the performance of `MapleSat` and to a surprising performance *increase* for `CaDiCaL`. Before presenting the results of our evaluation, however, we consider an alternative approach to storing the minimum viable state.

5 Migrating Solver State Using Proofs

The previous approach requires time to store solver state at the end of a run. In addition, it fails when a solver terminates unexpectedly. A more radical approach allows the state of the solver to be reconstructed at any time by using the proof trace as a write-ahead log that maintains the solver state. Most state-of-the-art solvers adhere to the DRAT proof format [46], so we build on DRAT in the following.

Intuitively, a DRAT proof of a formula is a sequence of clause additions and deletions that – when applied to the original formula – give rise to an “accumulated formula” maintained by the solver. More formally, a *proof* is a sequence of pairs $\langle m, C \rangle$, where $m \in \{\mathbf{a}, \mathbf{d}\}$ and C is a clause. If $m = \mathbf{a}$, the pair is an *addition*, and if $m = \mathbf{d}$, it is a *deletion*. Given a propositional formula F_0 and a DRAT proof $\langle m_1, C_1 \rangle, \dots, \langle m_n, C_n \rangle$, the *accumulated formula* at point i ($1 \leq i \leq n$) is defined as follows:

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } m_i = \mathbf{a} \\ F_{i-1} \setminus \{C_i\} & \text{if } m_i = \mathbf{d} \end{cases}$$

For a proof to qualify as a *valid* DRAT proof, it is required that for each addition $\langle a, C_i \rangle$, the clause C_i has the *RAT property* [36] with respect to the clause set F_{i-1} . The RAT property is a specific syntactic criterion ensuring that the addition of C_i preserves satisfiability. The specifics of the RAT property are irrelevant to our approach. In fact, our approach works with any other syntactic criterion that preserves satisfiability and thus applies also to proof systems like the blocked-clause proof system [37] or PR [25].

27:8 Migrating Solver State

In practical SAT solving, a DRAT proof is stored either in plain text or in a dedicated binary format. Example 3 shows a DRAT proof, its accumulated formula, and a serialization of the proof in the plain-text DRAT format.

► **Example 3.** Consider the formula $F = \{(x \vee y), (\bar{x} \vee y), (x \vee \bar{y}), (\bar{x} \vee \bar{y})\}$. The sequence $\langle \mathbf{a}, (y) \rangle, \langle \mathbf{d}, (x \vee y) \rangle, \langle \mathbf{a}, (\bar{y}) \rangle$ is a DRAT proof of F . The accumulated formula after the first addition is the formula $F \cup \{(y)\} = \{(x \vee y), (\bar{x} \vee y), (x \vee \bar{y}), (\bar{x} \vee \bar{y}), (y)\}$. The accumulated formula after the subsequent deletion is the formula $\{(\bar{x} \vee y), (x \vee \bar{y}), (\bar{x} \vee \bar{y}), (y)\}$. Assuming that the variables x and y are respectively assigned the numbers 1 and 2, the plain-text version of the formula in DIMACS and the proof in DRAT format are as follows:

DIMACS			
p	cnf	2	4
1	2	0	
-1	2	0	
1	-2	0	
-1	-2	0	

DRAT			
		2	0
d	1	2	0
		-2	0

Observe that in the plain-text DRAT format – analogous to the DIMACS format – lines end with a 0. Deletions are preceded by a **d** while additions are not preceded by a symbol. Note that we do not require proofs to end with the empty clause.

The accumulated formula of a proof trace represents the most relevant piece of state maintained by the solver, namely its current clause set. Thus, if a solver attempted to solve a formula without finishing, the proof trace can be used to reconstruct the accumulated formula, which can be passed to a new solver instance to continue.

This idea of extracting an accumulated formula from a proof is indeed the main idea behind the approach we present in the following. However, as we discussed in Section 4, extracting only a single clause set alone is not sufficient to ensure soundness and acceptable performance. To achieve these goals, we need to extend the proof format slightly to enable solution reconstruction and a separation of the accumulated formula into a redundant and an irredundant clause set.

5.1 Dual DRAT: A Simple Extension of the DRAT Format

In the following, we introduce two slight modifications of the DRAT format that equip proofs with enough information to extract the reconstruction stack and to separate the clause set into irredundant and redundant clauses. For clause deletions we add a witness that can be used for solution reconstruction (Section 4.1); for clause additions we add a non-negative integer that indicates the usefulness of the clause and that allows us to distinguish redundant clauses from irredundant ones. Although we consider both modifications of DRAT in combination, they could be applied independently too. We start with a formal definition of proofs in our format before discussing how to serialize these proofs in practice.

► **Definition 4 (Dual DRAT Proof).** A Dual DRAT (DDRAT) proof is a sequence of triples $\langle m, C, \omega \rangle$, where $m \in \{\mathbf{a}, \mathbf{d}\}$, C is a clause, and ω is either a non-negative integer (if $m = \mathbf{a}$) or a set of literals (if $m = \mathbf{d}$).

The non-negative integers added to clause-addition steps are meant to indicate the quality of a clause – the lower the value, the more important the clause. In practice, we use the value 0 for irredundant clauses and the literal-block distance for redundant clauses.

► **Example 5.** The following sequence of triples is a DDRAT proof: $\langle a, x \vee y, 1 \rangle$, $\langle d, x \vee y \vee z, \emptyset \rangle$, $\langle d, \bar{x} \vee \bar{y}, \{\bar{x}\} \rangle$, $\langle a, v \vee w, 0 \rangle$.

Given a DDRAT proof, we extract an *accumulated state* in the form of (1) an *irredundant formula* F , (2) a *redundant formula* R , and (3) a *reconstruction stack* σ .

► **Definition 6 (Accumulated Formulas).** Let F_0 and R_0 be propositional formulas, and let $\langle m_1, C_1, \omega_1 \rangle, \dots, \langle m_n, C_n, \omega_n \rangle$ be a DDRAT proof. The accumulated irredundant formula F_i and the accumulated redundant formula R_i ($1 \leq i \leq n$) are defined as follows:

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } m_i = a \text{ and } \omega_i = 0 \\ F_{i-1} & \text{if } m_i = a \text{ and } \omega_i \neq 0 \\ F_{i-1} \setminus \{C_i\} & \text{if } m_i = d \end{cases} \quad R_i = \begin{cases} R_{i-1} & \text{if } m_i = a \text{ and } \omega_i = 0 \\ R_{i-1} \cup \{C_i\} & \text{if } m_i = a \text{ and } \omega_i \neq 0 \\ R_{i-1} \setminus \{C_i\} & \text{if } m_i = d \end{cases}$$

In line with the original definition of DRAT, we define a DDRAT proof as *valid* if for every clause addition $\langle a, C_i, \omega_i \rangle$, the clause C_i is a RAT with respect to the formula $F_{i-1} \cup R_{i-1}$. As mentioned earlier, the RAT property is not relevant to our approach and so we are not going to discuss it further.

► **Definition 7 (Accumulated Reconstruction Stack).** Let $\langle m_1, C_1, \omega_1 \rangle, \dots, \langle m_n, C_n, \omega_n \rangle$ be a DDRAT proof and let σ_0 be a (possibly empty) sequence of pairs $\langle C, \omega \rangle$. The accumulated reconstruction stack σ_i is defined as follows:

$$\sigma_i = \begin{cases} \sigma_{i-1} \cdot \langle C_i, \omega_i \rangle & \text{if } m_i = d \text{ and } \omega_i \neq \emptyset \\ \sigma_{i-1} & \text{otherwise} \end{cases}$$

► **Example 8.** Let $F_0 = \{(\bar{u}), (u \vee v \vee w), (\bar{x} \vee \bar{y}), (x \vee y \vee z), (x \vee y \vee \bar{z})\}$, let $R_0 = \emptyset$, and let σ_0 be the empty sequence ϵ . Consider the proof from Example 5. After the first addition $\langle a, x \vee y, 1 \rangle$, $R_1 = \{(x \vee y)\}$ while $F_1 = F_0$ and $\sigma_1 = \sigma_0$. After the deletion $\langle d, x \vee y \vee z, \emptyset \rangle$ we get $F_2 = F_1 \setminus \{(x \vee y \vee z)\}$ while $R_2 = R_1$ and $\sigma_2 = \sigma_1$. The subsequent deletion $\langle d, \bar{x} \vee \bar{y}, \{\bar{x}\} \rangle$ gives $F_3 = F_2 \setminus \{(\bar{x} \vee \bar{y})\}$, $R_3 = R_2$ and $\sigma_3 = \langle (\bar{x} \vee \bar{y}), \{\bar{x}\} \rangle$. Finally, after the addition $\langle a, v \vee w, 0 \rangle$, we end up with $F_4 = F_3 \cup \{(v \vee w)\} = \{(\bar{u}), (u \vee v \vee w), (x \vee y \vee \bar{z}), (v \vee w)\}$, $R_4 = R_3 = \{(x \vee y)\}$, and $\sigma_4 = \sigma_3 = \langle (\bar{x} \vee \bar{y}), \{\bar{x}\} \rangle$.

In practical SAT solving, we can serialize a DDRAT proof in plain-text format by serializing irredundant-clause additions just as in DRAT. For redundant-clause additions, we start with an 1 (lower-case L, for *learned* clause) and first add the LBD as usefulness score before listing the literals. For deletions, we append the witness literals after the clause literals – as the witness must intersect with the deleted clause, we don't need an additional symbol to separate the witness literals from the other literals, but we require the first literal of the clause and the first literal of the witness to coincide (we thus know that the clause has ended and the witness has started as soon as the first literal repeats). Deletions with empty witnesses are thus serialized the same way as in DRAT.

► **Example 9 (Plain-Text Serialization of a DDRAT Proof).** The following proof is the plain-text serialization of the proof from Example 5, obtained by respectively mapping the variables v, w, x, y, z to the integers 1, 2, 3, 4, 5:

DDRAT	
1 1 3 4 0	// add (x y) as redundant clause with LBD 1
d 3 4 5 0	// delete (x y z), empty witness
d -3 -4 -3 0	// delete (-x -y), witness is {-x}
1 2 0	// add (v w) as irredundant clause

Note that a DDRAT proof serialized as above can be easily transformed into an ordinary DRAT proof. The resulting proof can then be checked with a proof checker like `drat-trim` [46]. Note also that proofs of multiple solver runs can be concatenated to obtain a proof of the original formula. If a solver produces a DDRAT proof, we can easily extract its clause sets and its reconstruction stack.

5.2 Producing Dual DRAT Proofs: Practical Considerations

As proof of concept, we extended the SAT solver `MapleSat` to enable the serialization of DDRAT proofs.

Most importantly, witnesses must be added to clause deletions that do not preserve logical equivalence. In addition, we believe that for performance it is crucial to ensure a close correspondence between the accumulated clause sets of the proof and the clause sets maintained internally by the solver – we must thus make sure that the accumulated clause sets do not contain clauses that are not actually retained by the solver (e.g., because their deletions were not logged in the proof). As discussed below, ensuring this for `MapleSat` required some effort. Designing a proof format based on FRAT [4], which forces solvers to be more careful with what they log, could thus lead to a fruitful alternative to DDRAT. The following changes to `MapleSat` were required:

- *Serializing witnesses for bounded variable elimination*: The solver performs *bounded variable elimination* [16] in a preprocessing step. This leads to clause deletions that do not preserve logical equivalence. When creating the proof, we thus must add the witnesses (the literals of the eliminated variables) for each of these clause deletions.
- *Logging clause deletions from the parsing phase*: When parsing the input formula from a DIMACS file, the solver ignores trivially satisfied clauses. To remove these clauses from the accumulated clause sets, we add proof logging for the corresponding deletions.
- *Adding unit clauses for locked-clause deletions*: The solver regularly deletes satisfied clauses. For clauses involved in top-level propagations (so-called *locked clauses*), we add the corresponding unit clauses to the proof (as done by `MergeSat`).
- *Adding clause deletions*: The solver regularly performs clause minimization as an inprocessing step [40]. In the original code, when a clause was minimized, the new clause was added to the proof but (presumably for performance reasons) the corresponding deletion of the original clause was not logged. We add proof logging of these deletions.
- *Reading redundant clause sets*: As discussed in Section 4.3, when reading redundant clause sets from input files, we must remove falsified literals from all clauses.

Finally, there are two slight details that we omitted for the sake of simplicity: In practice, accumulated clause sets of proofs are not viewed as ordinary sets but as multisets.¹ This means that for each clause deletion, we only delete a single occurrence of the clause (instead of all occurrences). Moreover, although Definition 6 does not add the LBDs to the accumulated redundant formula, we store them with the clauses when extracting state from a proof.

Apart from minor implementation details, these changes allowed us to extract clause sets and reconstruction stacks from the proofs produced by the solver, and to start subsequent solver runs with these clause sets. Applying the solution-reconstruction algorithm (Algorithm 1) led to valid solutions of satisfiable formulas, and concatenating the proofs and converting them to ordinary DRAT led to verified proofs of unsatisfiable formulas.

¹ <https://github.com/marijnheule/drat-trim>

5.3 Advantages and Disadvantages of the Proof-Based Approach

The most obvious disadvantage of the proof-based approach is that we need to parse all of a proof (of the previous run) to extract the clause sets and the reconstruction stack. While this is costly, it removes the time required to explicitly store the state at the end of a run. In our experiments (Section 6), the proof-based approach still takes significantly longer than the alternative approach from Section 4. However, because the proof is written to disk continuously as a write-ahead log, the proof-based approach has the advantage of being robust to unexpected terminations of the solver.

Our implementation work on `MapleSat` revealed that adding support for DDRAT to a solver is non-trivial and requires several careful changes. On the bright side, these format extensions can provide debugging support for incorrect SAT results (similar to how DRAT improved debugging UNSAT results), which may lead to better solver implementations.

6 Evaluation

In this section, we present an evaluation of our approaches for migrating solver state. We start by defining the experimental setup and then present and discuss the results.

6.1 Experimental Setup

Our experiment attempts to simulate the behavior of two kinds of solver migration in the cloud: in the first case, we consider an initial run on a serverless compute platform followed by a migration to a long-running dedicated compute platform. This simulates an environment in which we solve the easy cases using serverless resources and use managed computing resources to solve harder problems. In the second case, we consider repeated use of serverless resources until a problem is solved.

For all experiments, we set the timeout for migration to 500 seconds. The choice of 500 seconds is not arbitrary; it was chosen because – at the time of writing – the shortest timeout for calls of serverless functions of the three big cloud providers is 540 seconds. With this experiment we thus mimic the case where we start a solver run as part of a serverless function call but (40 seconds before the timeout) determine that the solver will likely not finish in time; we thus stop it and continue solving on another compute instance. As overall timeout, we use 5000 seconds per problem, which was the timeout in the SAT Competition 2021.

We consider three experimental settings:

1. *Single Run*: Run a solver on the benchmarks with a timeout of 5000 seconds per instance. This setting does not involve state migration and serves as a baseline for comparison.
2. *Double Run*: Run a solver until a timeout of 500 seconds, then start a second solver that reads the state from the first solver and runs until an overall timeout (including the time to store and restore state) of 5000 seconds. This setting describes migration from a serverless platform to dedicated computing resources.
3. *Multi Run*: Run a solver until a timeout of 500 seconds, then migrate state to another solver that runs again for 500 seconds. Repeat such 500 seconds runs until the formula is solved, with an overall time limit of 5000 seconds.

We perform an experiment with the full benchmark suite from the SAT Competition 2021 (400 formulas), using the following solver variants:

1. The original, unmodified `CaDiCaL` (version 1.5.2).
2. The original, unmodified `MapleSat`.
3. Our variant of `CaDiCaL` that stores state explicitly at the end of a run.
4. Our variant of `MapleSat` that stores state explicitly at the end of a run.
5. Our variant of `MapleSat` that stores state in the proof trace.

■ **Table 1** Overview of solved instances per experiment. Black numbers describe median solved instances over 5 runs, and gray ranges describe the min and max solved over 5 runs.

Solver	Variant	Experiment	Solved	SAT	UNSAT
CaDiCaL	Original	Single Run	270 [270, 270]	131 [131, 131]	139 [139, 139]
CaDiCaL	Dump	Double Run	274 [271, 277]	130 [127, 135]	144 [142, 146]
CaDiCaL	Dump	Multi Run	279 [275, 280]	134 [131, 135]	145 [144, 146]
MapleSat	Original	Single Run	245 [244, 250]	108 [108, 112]	137 [134, 138]
MapleSat	Dump	Double Run	246 [244, 255]	113 [110, 117]	133 [133, 138]
MapleSat	Dump	Multi Run	217 [215, 221]	100 [97, 103]	117 [113, 119]
MapleSat	Proof	Double Run	241 [238, 244]	108 [107, 113]	133 [129, 133]
MapleSat	Proof	Multi Run	222 [220, 225]	108 [104, 108]	114 [114, 120]
CaDiCaL+Maple	Dump	Double Run	276 [274, 277]	128 [127, 129]	148 [146, 149]
CaDiCaL+Maple	Dump	Multi Run	280 [278, 283]	132 [131, 136]	148 [146, 148]
Maple+CaDiCaL	Dump	Double Run	273 [270, 278]	128 [126, 135]	145 [143, 145]
Maple+CaDiCaL	Dump	Multi Run	278 [277, 280]	132 [130, 133]	146 [145, 148]

In addition, we consider *combinations* of solvers, in the following way:

6. Alternate CaDiCaL (var. 3) and MapleSat (var. 4) at each timeout, starting with CaDiCaL.
7. Alternate CaDiCaL (var. 3) and MapleSat (var. 4) at each timeout, starting with MapleSat.

We perform single-run experiments with unmodified versions of CaDiCaL and MapleSat (variants 1 and 2), and we perform double-run and multi-run experiments with all other variants (variants 3-7). For variants 6 and 7, where we combine both solvers, one solver starts the first run, afterwards either the other solver runs until the overall timeout (double run) or the two solvers take turns (multi run). We refer to the combination where CaDiCaL performs the first run (variant 6) as CaDiCaL+Maple and to the combination where MapleSat performs the first run (variant 7) as Maple+CaDiCaL. Each solver in the experiment produces proofs in binary format. To account for runtime variance induced by migrating state, we perform five independent runs per combination of experiment setting and solver variant/combination.

The experiments were performed on Amazon EC2 m5d.metal bare metal instances. Each instance is powered by AWS-custom Intel Xeon Scalable (Skylake) processors (96 vCPUs per instance), has 384 GiB memory and four SSDs with 900 GB of storage each. The instances run Amazon Linux 2, and for each experiment we ran 24 parallel processes per instance.

6.2 Results

Table 1 contains an overview of the numbers of solved instances per solver variant and experiment. As we performed five runs per experiment, the numbers in the table represent for each row the solved instances of the median run, i.e., the run for which there were two other runs that performed better and two other runs that performed worse in terms of overall solved instances.

In the table, *Dump* stands for the solver variants that store their state explicitly at the end of a solver run (the approach from Section 4), and *Proof* stands for the solver variant that stores its state as part of the proof trace (the approach from Section 5).

Even though the number of solved instances is a good performance indicator, it does not tell the full story. As an extreme example, consider a benchmark set where on each problem solver *A* either takes less than 2500 seconds or times out (taking more than 5000 seconds). In this setting, if solver *B* took exactly twice the time of solver *A*, it would still solve the

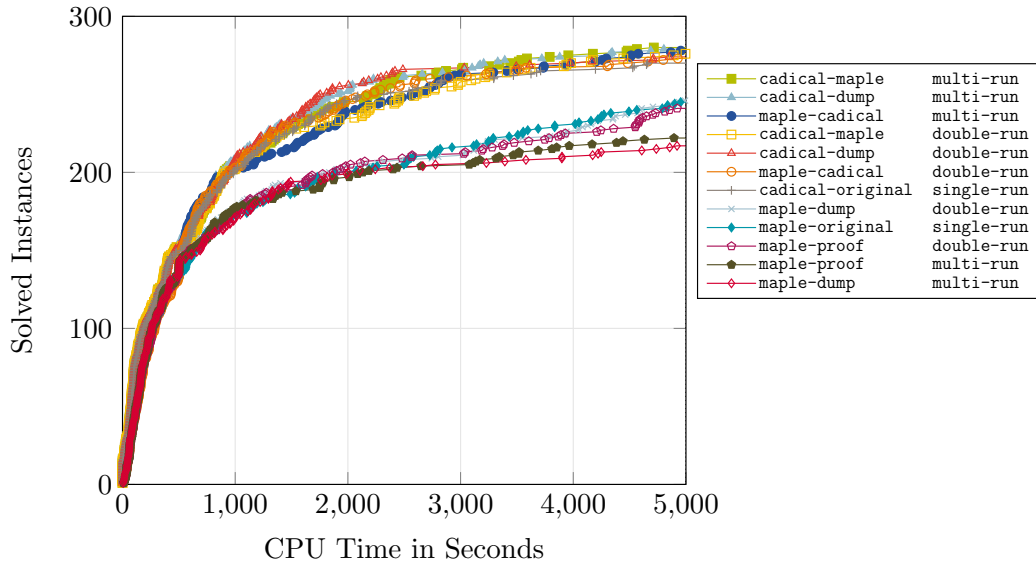


Figure 3 Cactus plot for all solver variants and experiments (median runs).

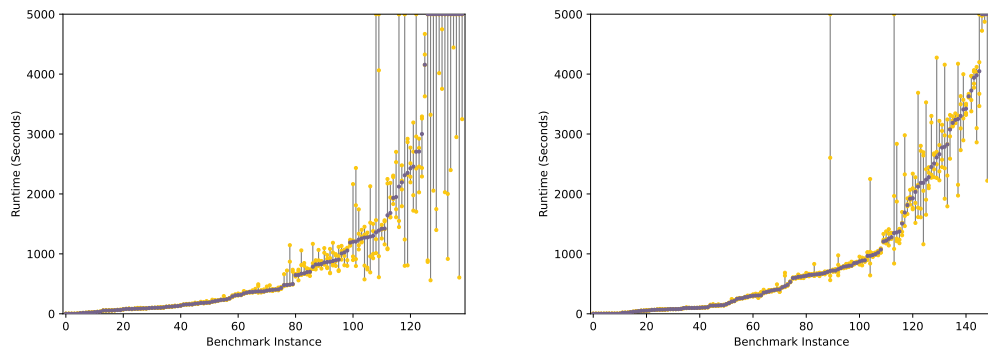
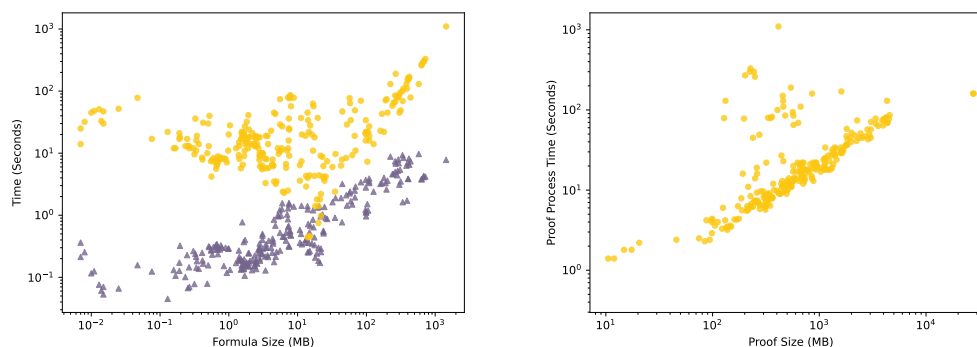


Figure 4 Distribution of runtimes for the instances solved by CaDiCaL in the double-run setting (state migration after 500 seconds). Left: satisfiable instances. Right: unsatisfiable instances.

exact same number of problems as A , giving the wrong impression that B was as efficient as A . To give a better indication of the actual solver runtime, Figure 3 shows a cactus plot for all combinations of experiments and solvers. Each curve represents the performance of a specific solver (or solver combination) in a specific experimental setting (single run, double run, or multi run). For each time (x -axis), the number on the y -axis denotes the number of problems for which the solver took at most that time to solve it. For example, a y -value of 110 at $x = 1000$ says that there were 110 problems for which the solver took at most 1000 seconds (each) to solve. Note that the solving time in Figure 3 includes the time spent on storing and reading the solver state. Appendix C contains additional cactus plots that separate the runs, making it easier to distinguish solvers.

As mentioned earlier, migrating solver state induces variance to the runtime, which is the reason why we performed multiple runs per experiment. To illustrate the run-time variance, we generated plots of the runtimes for the double-run experiment with the solver variant

27:14 Migrating Solver State



■ **Figure 5** Left: `MapleSat` state store+restore time when storing state explicitly (purple triangles) and proof process time when restoring state from proof (yellow circles) by formula size after the first 500 seconds run. Right: `MapleSat` proof process time by proof size after the first 500 seconds run.

of `CaDiCaL` that stores state explicitly at the end of a run (Figure 4). Notice that there is considerably more variance on the satisfiable instances than on the unsatisfiable instances. The median runtime for each problem is shown in purple, and the runtimes of the other four runs are shown in yellow. The diagrams look similar for `MapleSat` (see Figure 8 in Appendix D).

6.2.1 Time Spent on Storing and Restoring State

Figure 5 compares the time `MapleSat` spent on explicitly storing and restoring solver state with the time required (by a dedicated tool) for extracting solver state from a proof in the proof-based approach. We measured this time after the first run in the double-run experiments with `MapleSat` (data is from the median runs in terms of overall solved instances; notice that time and formula size are in log scale). The time required for storing and restoring state with `CaDiCaL` is given in Figure 9 in Appendix E.

Note that for the proof-based approach, the state is stored continuously as part of the proof trace, so we did not measure the storing time explicitly. For our experiments, we implemented a dedicated tool that processes a proof trace, extracts the clause sets and the solution-reconstruction stack, and writes them to separate files. The time shown in the plots is the time that our tool spent on reconstructing both the clause sets and the reconstruction stack in memory, excluding the time it spent on writing them to disk. We considered this fair for comparison because a solver could be extended to parse the proof directly (without writing to disk) if more efficiency is desired.

With `MapleSat`, explicitly storing and restoring state never takes more than about 10 seconds, and much less than that in the majority of cases. For the proof-based approach, restoring state takes less than about 100 seconds in most cases, but on some problems it takes even more than that. Notice the time spent on explicitly storing and restoring state is strongly correlated with the size of the input formula. In the proof-based approach, the restore time is strongly correlated with the size of the proof trace.

6.3 Discussion

Our experiments show that migrating solver state leads to mixed results for `MapleSat`, where in the double-run experiment our approaches perform about the same as when solving a formula in a single run with the original solver, but in the multi-run setting they perform considerably worse. The proof-based approach performs similarly to the approach that stores state explicitly, even though it spends more time on restoring state.

For `CaDiCaL`, where we implemented only the approach that stores state explicitly (and not in the proof), migrating state does not only perform on par with solving a formula in a single run but actually leads to more solved instances and better runtimes.

The combinations of the two solvers also perform well in our experiments. In fact, it is the combination of `CaDiCaL` and `MapleSat` in the multi-run setting, where both solvers take turns, that performed best in the experiment. This is especially interesting since `MapleSat` performed considerably worse than `CaDiCaL` in the experiments where solvers performed on their own. Three of the problems were only solved by a combination of the two solvers but not in any of the experiments involving only a single solver.

Our experiments indicate that hard restarts – like the ones we perform when migrating solver state, where only the clause sets and the LBDs are preserved – can improve the overall solving time significantly. Note that the usual restarts performed by CDCL solvers keep more state (e.g., phases and parts of the trail) than just the clause sets [10].

Migrating solver state generally (both for `MapleSat` and `CaDiCaL`) increases the variance in solving time, where especially on the satisfiable instances it can lead to huge differences. This could be explained by the fact that luck plays a greater role on satisfiable instances, where a good guess can make a huge difference, than on unsatisfiable instances, where a proof has to be derived laboriously by examining the entire search space.

Finally, observe that there is a strong correlation between store/restore time and formula/proof size. Storing state explicitly at the end of runs is much faster than restoring state from proofs, but for both approaches the time spent on storing and restoring state is mostly negligible compared to the overall solving time.

7 Conclusion and Future Work

We have examined mechanisms to migrate solver state across different computational resources. This allows us to stop and resume solvers at any point in time, and even to migrate from one solver to another. In a large experiment, we found that the cost in terms of time and storage necessary to migrate state was reasonable and in many cases even led to an unexpected performance improvement. This indicates that hard restarts, where a solver forgets everything except its clause sets, can lead to significant increases in solver performance.

For future work, we hope to extend our approach to support incremental solving, where we can store solver state – after solving an increment – for later retrieval. This would allow us to cache incremental problems, which may be useful for a variety of explorations of hard SAT problems. For example, one could imagine approaches such as `gg-SAT` [43] in which computations that time out are migrated using incremental queries that split the subformulas, preserving the learned clauses from the base problem. This approach could also lead to a variety of interesting portfolio approaches: for example, we could “late bind” parallelism, migrating an initial query to several solvers, permuting clauses, changing seeds, etc. We hope our paper serves as a starting point for a broader discussion on migrating solver state that will lead both to better proof formats and to fruitful alternative approaches.

References

- 1 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL: <http://ijcai.org/Proceedings/09/Papers/074.pdf>.
- 2 Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. 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 197–205. Springer, 2014. doi:10.1007/978-3-319-09284-3_15.
- 3 AWS Lambda system description. <https://aws.amazon.com/lambda/>. Accessed: 2022-02-06.
- 4 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.
- 5 Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.
- 6 Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2020.
- 7 Tomáš Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 156–172, Cham, 2015. Springer International Publishing.
- 8 Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- 9 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- 10 Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 1–17. EasyChair, 2019.
- 11 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. URL: <http://dblp.uni-trier.de/db/series/faia/faia185.html>.
- 12 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. *Handbook of Satisfiability*, 336:391–435, 2021.
- 13 Randal E. Bryant and Marijn J. H. Heule. Dual proof generation for quantified boolean formulas with a bdd-based solver. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28 – 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 433–449. Springer, 2021. doi:10.1007/978-3-030-79876-5_25.
- 14 Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jianguang Sun. Parallelizing SMT solving: Lazy decomposition and conciliation. *Artificial Intelligence*, 257:127–157, 2018. doi:10.1016/j.artint.2018.01.001.

- 15 Fady Coptý, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV*, pages 436–453. Springer, 2001. doi:10.1007/3-540-44585-4_43.
- 16 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. doi:10.1007/11499107_5.
- 17 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.
- 18 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019. doi:10.1007/978-3-030-24258-9_9.
- 19 Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 475–488, USA, 2019. USENIX Association.
- 20 Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001. URL: <http://dblp.uni-trier.de/db/journals/ai/ai126.html#GomesS01>.
- 21 Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *J. Satisf. Boolean Model. Comput.*, 6(4):245–262, 2009. doi:10.3233/sat190070.
- 22 Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 46(1):067, 2006.
- 23 Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with paracooba. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 114–122, Cham, 2020. Springer International Publishing.
- 24 Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 6598–6606. AAAI Press, 2018. URL: <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952>.
- 25 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. doi:10.1007/s10817-019-09516-0.
- 26 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.
- 27 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 28 Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning – 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014. doi:10.1007/978-3-319-08587-6_7.

- 29 Marijn J.H. Heule. Proofs of unsatisfiability. *Handbook of Satisfiability*, 336:635–668, 2021.
- 30 Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, pages 385–399, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- 31 Antti E. J. Hyvärinen and Christoph M. Wintersteiger. Parallel satisfiability modulo theories. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 141–178. Springer, 2018. doi:10.1007/978-3-319-63516-3_5.
- 32 Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010. URL: <http://dblp.uni-trier.de/db/conf/lpar/lpar2010y.html#HyvarinenJN10>.
- 33 Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008. doi:10.1016/j.tcs.2008.03.013.
- 34 Matti Järvisalo and Armin Biere. Reconstructing solutions after blocked clause elimination. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2010. doi:10.1007/978-3-642-14186-7_30.
- 35 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. doi:10.1007/978-3-642-12002-2_10.
- 36 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.
- 37 Oliver Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.
- 38 Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Painless: A framework for parallel SAT solving. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 233–250, Cham, 2017. Springer International Publishing.
- 39 Jia Hui Liang. *Machine Learning for SAT Solvers*. PhD thesis, University of Waterloo, December 2018.
- 40 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017. doi:10.24963/ijcai.2017/98.
- 41 Norbert Manthey. The mergesat solver. 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 387–398. Springer, 2021. doi:10.1007/978-3-030-80223-3_27.
- 42 Saeed Nejati, Zack Newsham, Joseph Scott, Jia Hui Liang, Catherine Gebotys, Pascal Poupart, and Vijay Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 251–260, Cham, 2017. Springer International Publishing.

- 43 Alex Ozdemir, Haoze Wu, and Clark Barrett. SAT solving in the serverless cloud. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 241–245, 2021. doi: 10.34727/2021/isbn.978-3-85448-046-4_33.
- 44 Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.
- 45 Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel SAT solving with threads and message passing. *J. Satisf. Boolean Model. Comput.*, 6(4):203–222, 2009. doi: 10.3233/sat190068.
- 46 Nathan Wetzler, Marijn J.H. Heule, and Warren A. Hunt Jr. 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.
- 47 Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. A concurrent portfolio approach to SMT solving. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09*, pages 715–720, Berlin, Heidelberg, 2009. Springer-Verlag. doi: 10.1007/978-3-642-02658-4_60.
- 48 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008. URL: <http://dblp.uni-trier.de/db/journals/jair/jair32.html#XuHHL08>.

A Formats for State Serialization

We propose both a plain-text format and a binary format for the irredundant clauses, redundant clauses (with their usefulness scores), and the reconstruction stack.

A.1 Plain-Text State Format

A.1.1 Irredundant Clauses

Irredundant clauses are serialized in the common DIMACS format, which we summarize here for the sake of completeness. A DIMACS file starts with a header line of the following form:

```
p cnf #number-of-variables #number-of-clauses
```

After the header line, all clauses are listed, literal by literal, with positive integers representing positive literals, and negative integers representing negative literals. Each clause must go to its own line ending with 0. Lines that start with the symbol `c` represent comments and are thus ignored.

► **Example 10.** Let $F = (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1) \wedge (x_1 \vee \bar{x}_4)$. The DIMACS encoding of F is as follows:

```
DIMACS
c this line is a comment
p cnf 4 3
2 -3 0
-1 0
1 -4 0
```

A.1.2 Redundant Clauses

Redundant clauses are serialized in a similar way as in DIMACS, with the only difference that files do not contain a header and that for each clause, the literals are preceded by the usefulness score.

► **Example 11.** Let $R = \{\langle(x_2 \vee \bar{x}_3), 1\rangle, \langle(\bar{x}_1), 1\rangle, \langle(x_1 \vee \bar{x}_4), 2\rangle\}$ be a redundant-clause set, i.e., a set of pairs where the first element is a clause and the second element is the clause's usefulness score. The plain-text encoding of R is as follows:

PLAIN-TEXT REDUNDANT CLAUSES				
1	2	-3	0	
1	-1	0		
2	1	-4	0	

A.1.3 Reconstruction Stack

We serialize a reconstruction stack by listing one clause-witness pair per line (we first list the literals of the clause and then the literals of the witness), ending each line with a 0. Similar to deletions in DDRAT, we require the first literal of the witness to coincide with the first literal of the clause. The stack is serialized bottom-to-top (i.e., earlier pairs in the sequence are serialized earlier in the file).

► **Example 12.** Let $\sigma = \langle(x_2 \vee \bar{x}_3), \{x_2, \bar{x}_3\}\rangle, \langle(\bar{x}_1), \{\bar{x}_1\}\rangle, \langle(x_1 \vee \bar{x}_4), \{\bar{x}_4\}\rangle$ be a reconstruction stack, i.e., a sequence of clause-witness pairs. The plain-text encoding of σ is as follows:

PLAIN-TEXT RECONSTRUCTION STACK				
2	-3	2	-3	0
-1	-1	0		
-4	1	-4	0	

A.2 Binary State Format

We propose binary formats that rely on the same integer encoding that is used for encoding literals in the binary DRAT format.²

A.2.1 Integer Encoding

Literals are identified by signed integers that are first mapped to unsigned (positive) integers and then serialized as variable-length byte strings (using an encoding known as *variable-length quantity*). The mapping of signed integers to positive integers is defined as follows:

$$u(n) = \begin{cases} 2n & \text{if } n > 0 \\ -2n + 1 & \text{if } n \leq 0 \end{cases}$$

Positive integers are then serialized as variable-length byte strings where the most-significant bit of each byte indicates whether the byte is the last byte in a sequence or whether there are more bytes to follow (to be precise, the most significant bit of a byte is 0 if and only if

² <https://www.cs.utexas.edu/~marijn/drat-trim/>

the byte is the last byte in the sequence). The remaining seven bits of each byte then encode the actual number in big-endian (i.e., earlier bytes are more significant). More precisely, given an integer n , we can split the conventional binary representation of $u(n)$ into a finite sequence $w_m \dots w_0$ of seven-bit words such that:

$$u(n) = w_0 + 2^7 w_1 + 2^{14} w_2 + \dots + 2^{7m} w_m$$

In our binary integer format, the number n is now serialized as the byte sequence

$$n_{\text{B}} = 1w_m \dots 1w_2 1w_1 0w_0$$

Note that for a given seven-bit word w , we denote by $0w$ the byte obtained by setting the most significant bit to 0 and then appending the bits of w . For example, if w is represented by the seven bits 1000000, then $0w = 01000000$, and similarly $1w = 11000000$.

► **Example 13.** Let $n = 979$. Then, $u(n) = 2 \times 979 = 1958$. The conventional binary representation of 1958 is 11110100110. We can now split this bit string into the two seven-bit words $w_1 = 0001111 = 15$ and $w_0 = 0100110 = 38$ such that $1958 = w_0 + 2^7 w_1 = 38 + 2^7 \times 15$. Thus, we obtain the two-byte encoding $979_{\text{B}} = 1w_1 0w_0 = 10001111 \ 00100110$.

► **Example 14.** Let $n = -979$. Then, $u(n) = -2 \times -979 + 1 = 1959$, which is 11110100111 in conventional binary. Thus, $w_1 = 0001111 = 15$ and $w_0 = 0100111 = 39$, resulting in the two-byte encoding $-979_{\text{B}} = 1w_1 0w_0 = 10001111 \ 00100111$.

A.2.2 Irredundant Clauses: Binary DIMACS

For the irredundant clauses, we define a binary DIMACS format. A binary DIMACS file starts with `0x00`, i.e., with a 0-byte (this initial 0-byte allows a solver/parser to identify the file as a binary DIMACS file) and is then followed by a sequence of clauses where each clause is represented by a sequence of integers (in the binary format defined in Section A.2.1) that is followed by `0x00`. Similar to DIMACS, positive literals are mapped to positive integers and negative literals are mapped to negative integers before they are encoded.

► **Example 15.** Let $F = (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1) \wedge (x_1 \vee \bar{x}_4)$. The binary DIMACS encoding of F is the byte sequence `0x00 2B -3B 0x00 -1B 0x00 1B -4B 0x00`, which boils down to:

BINARY DIMACS
0x00 0x04 0x07 0x00 0x03 0x00 0x02 0x09 0x00

Note that in contrast to the plain-text DIMACS format, we do not require a binary DIMACS file to explicitly mention the number of clauses or the number of variables.

A.2.3 Redundant Clauses

Redundant clauses are serialized in a similar way as irredundant clauses. Files start with a 0-byte and then list the clauses, with the only difference being that we precede the literals of a clause with the binary encoding U_{B} of its usefulness score.

► **Example 16.** Let $R = \{\langle (x_2 \vee \bar{x}_3), 2 \rangle, \langle (\bar{x}_1), 1 \rangle, \langle (x_1 \vee \bar{x}_4), 1 \rangle\}$ be a redundant-clause set. The binary encoding of R is the byte sequence `0x00 2B 2B -3B 0x00 1B -1B 0x00 1B 1B -4B 0x00`, which boils down to:

BINARY REDUNDANT CLAUSES
0x00 0x04 0x04 0x07 0x00 0x02 0x03 0x00 0x02 0x02 0x09 0x00

A.2.4 Reconstruction Stack

For reconstruction stacks in binary format, we do not start files with a 0-byte. This allows for straight-forward concatenation of multiple reconstruction-stack files, which is often useful. Apart from that, the binary format is analogous to the plain-text format: Each clause-witness pair is serialized by first listing the literals (now in binary format) of the clause and then the literals of the witness, whereby the first literal of the witness must also be the first literal of the clause. We also add a 0-byte after each pair.

► **Example 17.** Let $\sigma = \langle (x_2 \vee \bar{x}_3), \{x_2, \bar{x}_3\} \rangle, \langle (\bar{x}_1), \{\bar{x}_1\} \rangle, \langle (x_1 \vee \bar{x}_4), \{\bar{x}_4\} \rangle$ be a reconstruction stack, i.e., a sequence of clause-witness pairs. The binary encoding of σ is the byte sequence $2_B -3_B 2_B -3_B 0x00 -1_B -1_B 0x00 -4_B 1_B -4_B 0x00$, which boils down to:

BINARY RECONSTRUCTION STACK

```
0x04 0x07 0x04 0x07 0x00 0x03 0x03 0x00 0x09 0x02 0x09 0x00
```

B Binary DDRAT Format

In the binary DDRAT format, additions of irredundant clauses and deletions without witnesses are serialized in the exact same way as in binary DRAT. Additions of redundant clauses and deletions with witnesses are defined by introducing a separate starting symbol for both of them.

- *Additions of irredundant clauses* start with the byte $0x61$ (ASCII for the character ‘a’), followed by the binary encoding of the clause’s literals, and ending with the zero-byte $0x00$.
- *Additions of redundant clauses* start with the byte $0x72$ (ASCII for the character ‘1’), followed by the binary encoding U_B of the clause’s usefulness score U , then the binary encoding of the clause’s literals, and finally the zero-byte $0x00$.
- *Deletions* start with the byte $0x64$ (ASCII for the character ‘d’), followed by the binary encoding of the clause’s literals. If the witness is non-empty, the clause’s literals are followed by the binary-encoded literals of the witness, whereby the first literal of the clause and the first literal of the witness must be the same. Finally, deletions end with the zero-byte $0x00$.

► **Example 18.** Consider the DDRAT proof from Example 5: $\langle a, x \vee y, 1 \rangle, \langle d, x \vee y \vee z, \emptyset \rangle, \langle d, \bar{x} \vee \bar{y}, \{\bar{x}\} \rangle, \langle a, v \vee w, 0 \rangle$. Assuming that the variables v, w, x, y, z are respectively identified by the integers 1, 2, 3, 4, 5, the binary DDRAT encoding of this proof is the byte sequence $0x72 1_B 3_B 4_B 0x00 0x64 3_B 4_B 5_B 0x00 0x64 -3_B -4_B -3_B 0x00 0x61 1_B 2_B 0x00$, which boils down to:

BINARY DDRAT

```
0x72 0x02 0x06 0x08 0x00 0x64 0x06 0x08 0x0A 0x00 0x64 0x07 0x09 0x07 0x00
0x61 0x02 0x04 0x00
```

C Additional Cactus Plots

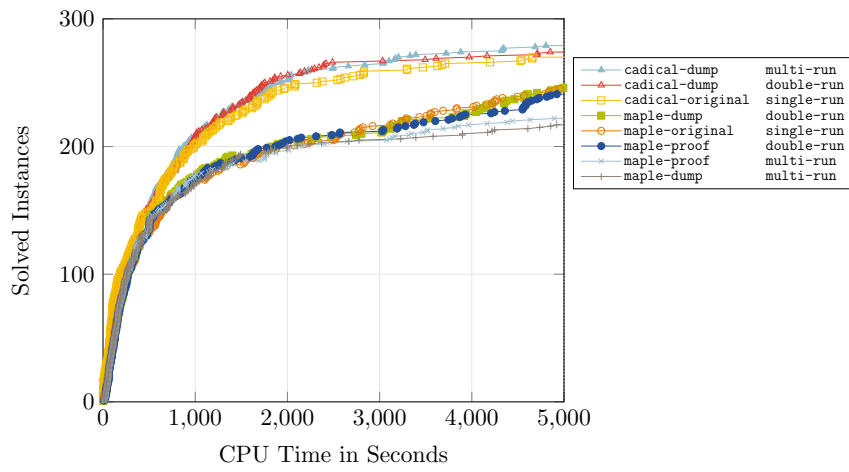


Figure 6 Cactus plot excluding the combinations of two solvers (median runs).

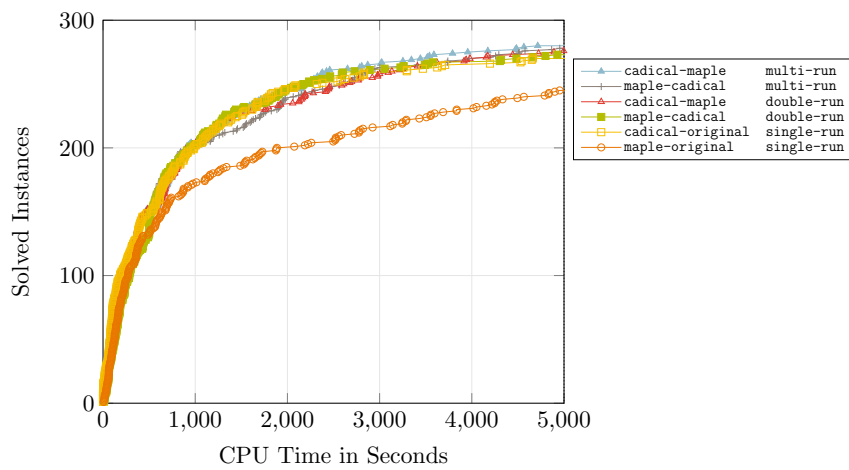
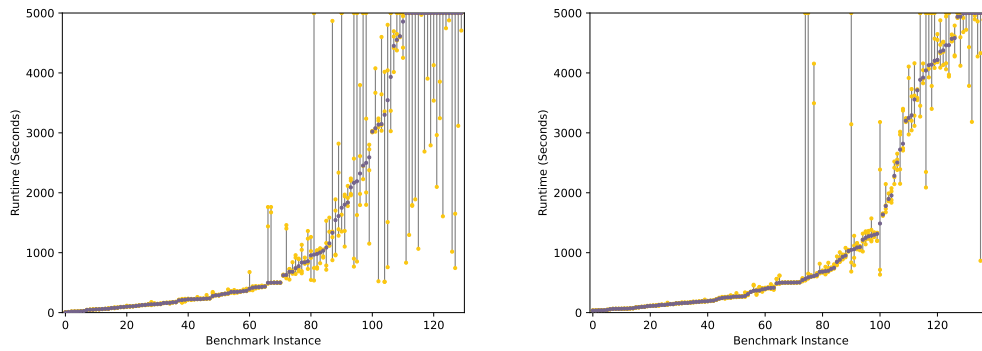


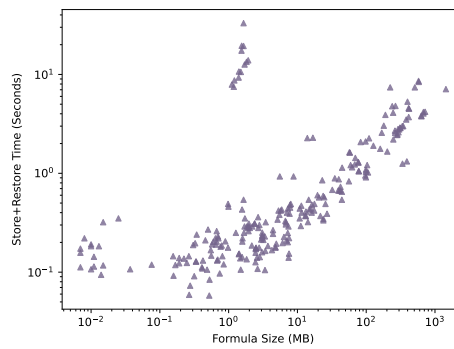
Figure 7 Cactus plot for solver combinations and single runs of original solvers (median runs).

D Runtime Plots for MapleSat



■ **Figure 8** Distribution of runtimes for the instances solved by MapleSat in two runs (state is stored explicitly between runs). Left: satisfiable instances. Right: unsatisfiable instances.

E State Store and Restore Time for CaDiCaL



■ **Figure 9** CaDiCaL state store+restore time by formula size after 500 seconds run.