Real-time Proof Checking for Distributed Incremental SAT Solving

Dominik Schreiber[1], Mathias Fleury[2], Katalin Fazekas[3], and Armin Biere[2]

[1] Karlsruhe Institute of Technology, Germany
dominik.schreiber@kit.edu
[2] University of Freiburg, Germany
[3] TU Wien, Austria

Abstract. Distributed clause-sharing SAT solvers are powerful automated reasoning tools capable of rapidly solving many difficult instances. Users of SAT solving often rely on incremental SAT solving, i.e., interactive solve calls over an evolving formula. We present the first approach to distributed incremental SAT solving that grants full confidence in the obtained result. Specifically, we extend a recent distributed real-time proof checking approach with an incremental proof interface. Our approach offers great flexibility in that it supports dynamic re-scheduling of computational resources and enables safely sharing clauses across tasks that operate on deviating assumptions and formula increments. We further add on-the-fly clause compression to checkers in order to reduce memory consumption. Experiments with the distributed solver MallobSat on up to 1216 cores show that our trusted solving approach checks incremental SAT tasks with small mean overhead ($< 33\%$) over unchecked solving.

Keywords: Propositional satisfiability · Distributed computing · Proofs.

1 Introduction

Propositional satisfiability (SAT) solving is an essential tool at the core of automated reasoning and symbolic AI [13]. In recent years, parallel (multi-core) and distributed (multi-node) approaches to SAT solving have gained traction and are now capable of solving many challenging instances hundreds of times faster than the best available sequential (single-core) solvers [37,38]. Moreover, recent works have brought the efficient generation and validation of *proofs of unsatisfiability* to large-scale SAT solvers [23,24,31]. Proofs of unsatisfiability [17,18] are artifacts that allow to validate a solver's claim of a formula's unsatisfiability with simple, independent, and even formally verified checker programs [8,22].

Modern use cases of SAT solving often require a mode of operation named *incremental SAT solving*. Rather than encoding and solving a single propositional formula, incremental SAT solving allows for interactive and interrelated solving calls on an evolving formula while preserving the SAT solver's knowledge base gathered thus far. While such interaction schemes are essential for many important application domains such as verification, electronic design automation, or

scheduling, a first proof format for (sequential) incremental SAT solving has been proposed only recently [11]. Parallel and distributed SAT solvers that operate incrementally [32] do not support proof production and checking yet. As such, challenging incremental problems for which proof checking is desired have to be solved sequentially or must be broken down into sequences of non-incremental SAT instances, both of which can drastically impact performance.

In this work, we propose the first approach to proof checking for (parallel and) distributed incremental SAT solving. We build upon a recently proposed, bottleneck-free approach that checks proofs *in real time* during solving and validates results of (un)satisfiability as soon as a solver emits it [31]. This immediate proof checking is highly natural for incremental SAT solving (see also [11]), more so than post-mortem checking of a monolithic proof produced during solving, since solving already has an interactive on-demand nature and a user may want to have full confidence in a result before proceeding with the next solving call. The downsides of real-time proof checking are that no persistent proof is generated – i.e., trust in a result cannot be easily transferred to other parties – and that distributed real-time checking relies on hash-based *fingerprints* to ensure the integrity of messages across trusted checker programs. While these particularities reduce confidence compared to conventional proof checking to some degree, real-time checking can still be a highly appealing method, e.g., for software verification in continuous integration workflows [7], for debugging purposes [26], or whenever persistent proof logging is infeasible due to time or space restrictions.

Our contributions are as follows: We significantly expand on the prior real-time checking framework ImpCheck [30] by transferring aspects of the incremental (sequential) LIDRUP format [11]. Our framework allows to check a distributed *incremental* clause-sharing solver's reasoning even if its solver units operate on deviating assumptions and/or increments of the problem. This allows our approach to support flexible (re-)scheduling of workers [27] and checked clause sharing *across* distinct tasks that operate on related problems (cf. [34]).

We formalize our approach and implemented it in a small (< 2000 effective lines of code) toolchain, connected to the distributed platform Mallob [30] as a first major use case. Evaluations at up to 1216 cores confirm that our real-time checking approach incurs little mean overhead ($< 33\%$) over un-checked solving, less than prior results for non-incremental real-time proof checking [31]. Lastly, we extend our checkers by simple and effective clause compression techniques that reduce the main memory overhead of proof checking from 45% to 32%.

2 Incremental SAT Solving

We consider *incremental SAT solving under assumptions* [9] – adjusting the formalism by Fazekas et al. [11] to allow flexible incremental interactions.

An incremental SAT task $P = (F, Q)$ consists of a *sequence of clauses* $F = \langle c_1, c_2, \ldots \rangle$ and a sequence of *queries* $Q = \langle q_1, q_2, \ldots \rangle$, where each $q_i = (e_i, A_i)$ features a *clause index* e_i and a set of *assumption literals* A_i. We define $F_e := \bigcup_{i=1}^{e} \{c_i\}$ as the *prefix* of formula F that includes its first e clauses.

A query $q = (e, A)$ is *satisfiable* if and only if the clause set $F_e \cup \bigcup_{a \in A} \{a\}$ (which consists of $e + |A|$ clauses) is satisfiable. We assume queries to be ordered w.r.t. the formula prefix, i.e., for any q_i, q_j with $i < j$, we have $e_i \leq e_j$.

Consider $F = \langle\, a \vee b, a \vee \neg b, \neg b \,\rangle$ and $Q = \langle\, (2, \{a, b\}), (2, \{\neg a\}), (3, \emptyset) \,\rangle$ as an example: q_1 features clause set $F_2 = \{a \vee b, a \vee \neg b\}$ and assumes $a = b = 1$, which satisfies F_2; q_2 also features F_2, assumes $a = 0$ and is unsatisfiable; q_3 features $F_3 = F_2 \cup \{\neg b\}$, makes no assumptions and is satisfiable ($a = 1$, $b = 0$).

The incremental LIDRUP proof format [11] allows to log and validate incremental reasoning. It extends the non-incremental LRUP proof format [26]. An LRUP proof is a sequence of clause derivations and deletions, where each clause (original or learned) has a unique identifier (ID) and each derivation of a learned clause c involves a list of prior clause IDs. These *dependencies*, or *prerequisites*, indicate how to efficiently check that c is entailed, using the so-called *Reverse Unit Propagation* (RUP) criterion [40]. An LRUP proof of unsatisfiability ends with the derivation of the *empty clause*, which implies unsatisfiability.

In contrast to LRUP proofs, a LIDRUP proof begins with explicitly listing the (initial) input clauses. It also features statements that mark the beginning of a solving query with according assumption literals. Subsequent learned and deleted clauses are stated as in LRUP. The end of each query is marked by a result statement (SAT, UNSAT, or UNKNOWN), potentially with the found satisfying assignment or the *core* clause (corresponding to *failed* assumptions) that renders the query unsatisfiable. After the result line(s), input clauses can be added once again and another query can begin. LIDRUP also supports advanced features that we do not consider in this work, such as assumption-like *constraint clauses* [14] and interactions with IPASIR-UP style *user propagators* [10].

3 Proof-Producing Clause-Sharing Solving

In parallel *clause-sharing SAT solving* [2,38], many sequential SAT solver threads are run in parallel, all on the original input formula, and occasionally exchange *learned conflict clauses* among each another. Careful clause sharing grants significant scalability up to thousands of cores [38], even if individual solver threads run (almost) the same solver program [36]. The distributed SAT solver MallobSat [38] is the first such system that supports incremental SAT solving [32]. MallobSat is integrated in the distributed framework Mallob, which allows for *flexible multi-tasking*, i.e., to process SAT tasks on-demand while continuously balancing the available workers among all active tasks. Mallob's flexible multi-tasking of incremental SAT tasks is being exploited for distributed MaxSAT solving [34] and for massively parallel bit-precise verification (SMT) [35].

As of now, there are two viable proof checking methods in parallel and distributed clause-sharing solving. Both approaches are limited to RUP proof steps and to sequential solver backends that support LRUP proof logging.[4]

[4] Authors often simplifyingly state that they use/support the *LRAT* [8] format while they focus exclusively on its *LRUP* feature subset [23,24,26,31].

First, Michaelson et al. [23,24] proposed to (re-)construct a global proof by logging all solver threads' proof information during solving and then performing a post-solving parallel backwards traversal over the transitive prerequisites of the found empty clause [23]. While this approach outputs a plain LRUP proof file like a sequential solver, funnelling all required proof steps into a single file constitutes an I/O bottleneck, and the proof's potentially huge set of active clauses can cause main memory shortage during checking [24].

Second, we introduced *ImpCheck* [31] (Immediate Massively Parallel Propositional Proof Checking) – a framework where the parallel / distributed solver's reasoning is checked *in real time* during solving. ImpCheck consists of three small sequential executables that are run as a part of the solving procedure and whose output ensures correctness. First, a *parser* process parses the input formula and returns the read clauses. Second, for each solver thread during solving, a dedicated *checker* process is executed and receives the parsed formula. Each solver thread uses inter-process communication to send its proof steps to its checker, which checks them in real-time. After checking, the learned clauses can be shared between solver threads. If a solver thread imports such a shared clause c, it also forwards c to its checker, which accepts c *without* checking – note that the dependencies of c are unknown and might not be present locally.

This approach alone would not yet offer particularly high confidence since the information transfer between trusted executables is untrusted and may corrupt or fabricate clauses. For this reason, we introduced formula and clause *fingerprints* (based on keyed pseudo-random functions [1]) that only the trusted executables are able to compute. Specifically, each ImpCheck process outputs a fingerprint together with each formula / clause it outputs and, in turn, expects a valid fingerprint for each incoming piece of information from another process. This scheme establishes trustworthy communication channels across the trusted executables that do not depend on the employed technology (shared-memory synchronization, distributed message passing, parallel file systems, etc.).

Lastly, a *confirmer* executable can be used to validate a fingerprint that a checker has output when it concluded the problem's (un)satisfiability.

While the ImpCheck approach does not yield a persistent proof, it is substantially more scalable than monolithic proof production and checking, only incurring a mean overhead of $\leq 42\%$ over a distributed solver's running time [31].

4 Distributed Incremental Real-time Checking

We extend the distributed real-time checking method ImpCheck [31] (Section 3) to support a significant subset of the incremental LIDRUP proof format [11], including *incremental addition of input clauses* with corresponding fingerprints, *incremental queries* with *assumption literals*, and the validation of (un)satisfiability results against (failed) assumption literals, with the following main challenges:

– We need to ensure that all p interacting checker instances in a setup operate on a *consistent set of input clauses*, i.e., that, given some $e \geq 1$, F_e involves exactly the same clauses across all checkers.

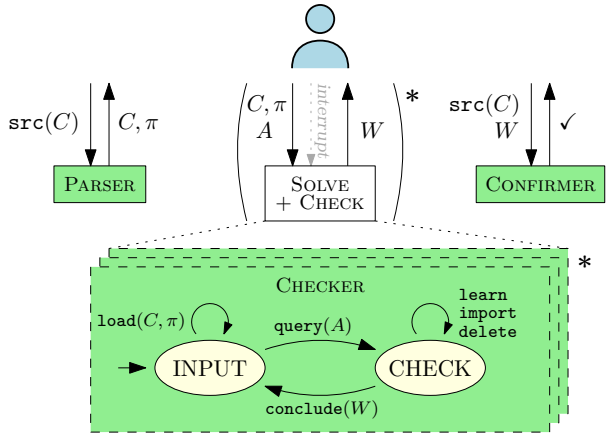


Fig. 1. Top: Possible user interactions with our system, on clauses C, fingerprints π, assumptions A and witnesses W. Bottom: Internal state machine of a checker unit. Trusted modules of the procedure are colored in green. All interactions are repeatable loops, e.g., the user can obtain growing prefixes of F by querying the parser incrementally many times over with an updated clause source. The parser and confirmer modules are considered singletons whereas multiple distinct "Solve + Check" (S+C) interactions can co-exist. In turn, each S+C procedure can entail many incarnations of the displayed checker module, each of which is controlled by the surrounding solver program. Lastly, all checker modules across *all* of the user's concurrent S+C procedures can interact with each another by sharing learned fingerprinted clauses.

- Since checker instances are not necessarily strictly synchronized, we need to ensure that learned clauses can *never travel back in time*, i.e., a clause derived from formula F_e must never be shared with a checker that currently operates on some $F_{e'}$ with $e' < e$.
- We require a *globally consistent allocation* of IDs for original clauses as well as learned clauses from all increments and across all solver units.

Our approach, illustrated in Fig. 1, offers a generic and flexible interface for trusted distributed incremental SAT. We first define the required types of fingerprints, then describe the interface of the three types of trusted modules (parser, checker, confirmer), and finally touch on strategies to assign clause IDs.

4.1 Fingerprints

In our setting, a number of individually trusted checker instances run in parallel, potentially distributed across different compute nodes. These checkers will need to exchange information such as, e.g., a certain clause being entailed by the problem. In order to trust the overall procedure, we have to either let the checkers share the full reasoning behind every result they share (e.g., dependencies of learned clauses) or establish a means of *trusted communication* between checkers.

Given that the former option is prohibitive in practice, the real-time checking approach ImpCheck established *fingerprints*[5] as an efficient, environment-independent, and highly trustworthy mechanism of establishing such trusted communication. We adopt and extend this method to our incremental setting.

As in original ImpCheck, we define fingerprinting function f using an underlying checksum or hashing algorithm $H_K : \{0,1\}^* \to \{0,1\}^\ell$, where ℓ is the bitlength of each fingerprint (e.g., 64 or 128 bits) and the output is influenced by a global *key* K. This key K is private to all trusted modules.[6] The outer function $f(\cdot) := f_K(\cdot)$ is parametrized with K and evaluated in three different contexts: on a formula F_e, on a single clause c relative to a certain increment, and to certify a query's unsatisfiability or satisfiability. Let "$\circ$" denote concatenation.

$$f_K(F_e) := H_K(F_e) \tag{1}$$
$$f_K^{F_e}(c) := H_K(id(c) \circ c \circ f_K(F_e)) \tag{2}$$
$$f_K^{F_e}(\top, A) := H_K(10 \circ f_K(F_e) \circ A) \tag{3}$$
$$f_K^{F_e}(\bot, \phi) := H_K(20 \circ f_K(F_e) \circ \phi) \tag{4}$$

Defining each clause signature $f_K^{F_e}(c)$ relative to F_e (2) ensures that receiving checkers can confirm the *no back-in-time* property of shared clauses. A fingerprint representing satisfiability (3) is based on all of the current query's assumptions A, thus ensuring that all assumptions have been checked to be satisfied by the found model. Similarly, a fingerprint representing unsatisfiability (4) must be relative to the set of *failed assumptions* $\phi \subseteq A$ that render the query unsatisfiable – otherwise, a fingerprint representing unsatisfiability w.r.t. some non-empty ϕ could be misinterpreted to represent the formula's overall unsatisfiability.

We ensure that all definitions have disjoint inputs to prevent ambiguities (cf. [31]) by prepending 10 or 20 to $f_K(F_e)$ as a single byte and by padding each clause ID with two zero-bytes, while F_e consists of a multiple of four bytes.

4.2 Trusted Modules

We now detail our changes and extensions to each of the three trusted modules of ImpCheck (see Section 3), as shown in Fig. 1.

We extend ImpCheck's **parser** to successively parse F and output a corresponding sequence $\langle (F_e, f_K(F_e)), (F_{e'} \setminus F_e, f_K(F_{e'})), (F_{e''} \setminus F_{e'}, f_K(F_{e''})), \ldots \rangle$ (where $e \leq e' \leq e'' \leq \ldots$) via a series of incremental calls. To match the incremental nature of our approach, our parser also works incrementally; a single parser execution spans an entire incremental SAT task P, where formula increments can be provided via a static or a successively updated input source (see

[5] ImpCheck's fingerprints are called *signatures* [31], which we renamed following a remark by David Basin that *signatures* have a deviating meaning in cryptography.

[6] In ImpCheck's current implementation, each trusted module computes K based on (i) some internal "salt" in the source code, (ii) compile-time system randomness compiled into the executables, and (iii) a seed provided as command-line argument.

```
state == INPUT
load(clauses: ClauseList, f: Fingerprint) → bool
oldstate == INPUT && state == CHECK
query_sat(asmpt: LiteralList) → bool
oldstate == CHECK && state == CHECK
learn(id: ID, lits: LiteralList, hints: IDList, share: bool)
  → (bool, Fingerprint?, u32?)
oldstate == CHECK && state == CHECK
import(id: ID, lits: LiteralList, fp: Fingerprint, cidx: u32) → bool
oldstate == CHECK && state == CHECK
delete(ids: IDList) → bool
oldstate == CHECK && state == INPUT
conclude_sat(M: Model) → (bool, Fingerprint?)
conclude_unsat(id: ID, failed: LiteralList) → (bool, Fingerprint?)
conclude_unknown() → bool
Terminates checker module from any state, returns acknowledgment
terminate() → bool
```

Fig. 2. Abstract checker interface with basic state invariants. Each method, if used from the wrong state or with invalid or unsound arguments, can bring the checker into an INVALID state in which all subsequent calls will return `false`.

Section 7 for details). Assumptions can either be provided with the input, in which case the parser forwards them as well, or can be introduced by the user at a later point, since assumptions are not subject to fingerprinting at this point.

The **confirmer** executable operates in an analogous fashion to the parser, the only difference being that the confirmer additionally takes a *witness* $W = (e, s, \pi, w)$ for each incremental call, where $s \in \{0, 10, 20\}$ is a result code, π is a fingerprint, and w is a (possibly empty) sequence of assumption literals. The confirmer then explicitly checks the validity of each witness W, i.e., that $\pi = f_K^{F_e}(\top, w)$ if $s = 10$ ("satisfiable") and that $\pi = f_K^{F_e}(\bot, w)$ if $s = 20$ ("unsatisfiable"). For the case $s = 0$ ("unknown"), π and w can be empty.

ImpCheck's **checker** features an initial *input* stage of loading the formula and then a subsequent *checking* stage until, at some point, the checker is terminated. We replace this linear program flow by a loop, alternating between the *input* and *check* states. Transitioning from *input* to *check* requires a `query_sat` call with a set of assumptions A. Transitioning from *check* to *input* requires a `conclude` call with a SAT, UNSAT, or no (UNKNOWN) result. Fig. 1 (bottom) shows an according state transition graph whereas Fig. 2 shows the full checker interface.

When a `learn`ed clause is checked, its fingerprint is output together with the current e. When `import`ing an external clause c, the according e must be provided. The receiving checker then uses $f_K(F_e)$ to recompute and thus validate the fingerprint $f_K^{F_e}(c)$. In other words, the receiver must check that its current F_e entails the clauses entailing c. In particular, this ensures that the message did not travel back in time from a "future" formula $F_{e'}$, $e' > e$. On a practical

note, supplying each clause with its index of origin increases the metadata to be carried with during clause sharing, e.g., from 24 to 28 bytes for 32-bit indices. Additionally, each checker must remember $f_K(F_e)$ for all e.

For any set A of assumptions, validating a satisfiability result (`conclude_sat`) entails checking the supplied model against each assumption in A and the original problem clauses (which must be kept by the checker). Conversely, validating an unsatisfiability result (`conclude_unsat`) involves a set of failed assumptions ϕ, which the checker makes sure to be part of the queried assumptions A, and a reference to a previously added *conclusion clause* $\hat{c}$, which must be (a subset of) the negation of ϕ. Thus $\hat{c}$ confirms that the provided failed assumptions in conjunction are unsatisfiable. It is important to note that the solver learns clauses that are entailed by the formula and independent of the assumptions; as such, sharing clauses is independent of the sender's and the receiver's assumptions.

The LIDRUP format tracks clause *weakening* and *restoring* in a solver [11]. For our checkers, this would mean to store active and inactive clauses separately or to supply each clause with a flag indicating whether it is active. This added effort comes at no benefit because our checker never needs to iterate over all active clauses (exploiting ID hints instead). As such, we decided to ignore weakening and restore operations, with the consequence that our checker tolerates if a solver uses a clause that it internally considers "inactive". Note that the correctness of an unsatisfiability result solely depends on each individual clause being entailed by the formula, and for satisfiable problems, the witness assignment is a model for the entire original formula. As such, (not) tracking weakened clauses is inconsequential for correctness. Lifting this restriction and extending our approach to stronger forms of learning [12] is left for future work.

4.3 Clause ID Domains

In single-threaded proof production, clause (LRAT/LRUP) IDs are commonly assigned based on a single integer counter. To obtain globally unique IDs in distributed settings, non-incremental ImpCheck requires that the first o IDs are reserved for the o original problem clauses and that producing solver thread i (out of p) should only assign new clause IDs of the shape $i + k \cdot p$ ($k \in \mathbb{N}$).[7]

In our distributed and incremental setting, we have no prior knowledge of how many clauses will be learned and shared in the scope of a single increment. Moreover, we want to support flexible modes of parallelism, where SAT solving processes can be added or removed *during* an ongoing solving procedure [27,38]. New solver-checking units joining an existing solve procedure thus also need to operate on "fresh" ID domains without interfering with existing ones.

At an abstract level, we consider each clause ID not as a plain number but as a pair (i, x), where i indicates the clause's *origin* ($i = 0$ for original input clauses and the globally unique index $i > 0$ of the producing solver-checker unit

[7] In the original ImpCheck [31], assigned clause IDs have the shape $o + 1 + i + k \cdot p$ ($k \in \mathbb{N}_0$) if there are o original input clauses (cf. [23]). Later versions use $i + k \cdot p$, which simplifies arithmetic on clause IDs and makes them independent of o.

otherwise) and x is the unit's local clause counter. There are many possibilities to encode this information into a single unsigned integer, including algebras with which we can uniquely express arbitrarily large i and x given sufficiently large IDs [4,6]. As a more pragmatic alternative, we can reserve the first O clause IDs for *all* original clauses across all increments, for some conservatively chosen O (e.g., $O = 2^{32} \approx 4.3$ billion), and then employ modulus arithmetic as in earlier works [23,31] to encode i. Specifically, each ID assigned by the i-th solver-checker unit satisfies $\text{ID} \equiv i \mod p^*$, where p^* is an upper bound for the number of solver-checker units that may be initialized in the scope of the entire procedure.

In all instances where our approach expects clause IDs to be restricted to certain domains, any violation will be noticed by a checker unit *if it results in an unsound proof*. Each checker, at any point in time, internally maps each known ID to exactly one clause. Upon receiving a clause with an already known ID, the checker raises an unrecoverable error. Similarly, an error is raised if a derivation uses a clause ID in a context that does not fit the local clause known under this ID. If a checker raises no such error, then each of its clauses has (had) a unique ID in the scope of its life time. Even if a clause ID is reused after its original clause was deleted, there is a straightforward way of renaming conflicting clause IDs that results in a sound global proof.

5 Correctness

We now prove the correctness of our approach. We first introduce an abstract solving model, determine that it is sound and then show how our real-time distributed incremental proof checking approach implements this model.

At an intuitive level, our abstract model captures two kinds of clause sets. First, there is a *local* clause database C_i for each solver/checker unit i, which contains the problem clauses of i as well as its learned and imported clauses. Second, for each increment index e there is a *global* clause database B_e that contains all clauses exported by units that operated on F_e at that point in time.

Formally, we establish clause sets $B := \langle B_1, B_2, \ldots \rangle$ and $C := \langle C_1, C_2, \ldots \rangle$ and perform a sequence of *actions*: (i) $\texttt{read}(i)$ reads a single input clause from some external source and adds it to C_i; (ii) $\texttt{learn}(i)$ learns a RUP clause c from clauses in C_i and inserts c to C_i; (iii) $\texttt{push}(i, e)$ exports a clause from C_i to B_e; (iv) $\texttt{fetch}(e, i)$ imports a clause from B_e to C_i.

For any i, we define η_i as the number of $\texttt{read}(i)$ actions performed thus far. Given formula F, we define the following *conditions* for the above procedure:

C1 All clause sets are initially empty and modified only by the above actions.
C2 For any e and i, the first e $\texttt{read}(i)$ actions together yield a prefix F_e of F.
C3 For any i, any action $\texttt{push}(i, e)$ satisfies $e = \eta_i$ and any action $\texttt{fetch}(e, i)$ satisfies $e \leq \eta_i$ at the point of their respective execution.

For example, consider $F = \{a \vee b, a \vee \neg b, \neg a\}$ and actions $\mathcal{A} := \langle\, \texttt{read}(0), \texttt{read}(0), \texttt{learn}(0), \texttt{push}(0, 2), \texttt{read}(1), \texttt{read}(1), \texttt{read}(1), \texttt{fetch}(2, 1), \texttt{learn}(1) \,\rangle$. Instance $i = 0$ reads the first two clauses of F, learns a clause (unit clause a),

and writes this clause to B_2 (since $\eta_0 = 2$ at that point). Instance $i = 1$ reads all three clauses of F, fetches clause a from B_2 (which is valid since $2 \leq \eta_1 = 3$), and can now learn the empty clause. In the end, we have $C_0 = \{a \vee b, a \vee \neg b, a\}$ and $C_1 = \{a \vee b, a \vee \neg b, \neg a, a, \bot\}$. Note that the actions of $i = 0$ and of $i = 1$ could be interleaved arbitrarily in $\mathcal{A}$ as long as $\texttt{push}(0, 2) \prec \texttt{fetch}(2, 1)$.

Theorem 1 (Soundness). *Assume a sequence of n actions that satisfies* C1–C3. *For any $e \geq 1$, each $c \in B_e \cup \{C_i : \eta_i \leq e\}$ is entailed by F_e.*

Proof. We show the claim via induction over n. After $n = 0$ actions, all clause sets are empty (C1). Next, given that the claim holds after n actions, action $n+1$ preserves it: Action $\texttt{read}(i)$ increments η_i and subsequently adds the η_i-th clause of F (C2), which is trivially entailed by F_{η_i}, to C_i. Action $\texttt{learn}(i)$ learns a clause c from clauses inductively entailed by F_{η_i} and adds it to C_i. Action $\texttt{push}(i, e)$, for $e = \eta_i$ (C3), adds a clause inductively entailed by F_{η_i} to B_{η_i}. Action $\texttt{fetch}(e, i)$, for $e \leq \eta_i$ (C3), adds a clause inductively entailed by F_{η_i} to C_i. In all cases, the action preserves the claimed property. □

Note how this model, *by design*, rules out cyclic clause dependencies. The induction over the linear action sequence shows that the distributed proof remains a DAG at all times and that no action can ever induce a dependency loop.

As the next step, we correspond the above abstract model to our checking approach. For this means, consider assumption **A1**: Fingerprints provide **perfect authenticity**, i.e., each string s that constitutes a valid fingerprint for object x originates from trusted code explicitly computing s as the fingerprint of x.

Note that (A1) rules out fingerprints $f(x)$ with a *second preimage* $x' \neq x$ such that $f(x) = f(x')$: Any such $f(x')$ is a valid fingerprint for x but was not computed as the fingerprint of x. We discuss the validity of (A1) after the proof.

Theorem 2. *Consider our distributed incremental checking under assumption* (A1) *with a parser that emits fingerprints for formula F. This approach implements the above abstract procedure for F while satisfying conditions* C1–C3.

Proof. We first establish a mapping between the abstract model and our concrete checking approach. Each C_i corresponds to the i-th checker's clause database in our approach. Each $\texttt{load}(i)$ corresponds to checker i loading its next input clause and validating it against a provided fingerprint, and each $\texttt{learn}(i)$ corresponds to checker i adding a *confirmed* RUP clause to its clause database. We define the clause sets B virtually: We consider clause c to be part of B_e if and only if fingerprint $f_K^{F_e}(c)$ has been emitted by some checker. As such, each $\texttt{push}(i, e)$ corresponds to checker i emitting a fingerprint for a clause entailed by F_e, and each $\texttt{fetch}(e, i)$ corresponds to checker i importing a clause with a valid fingerprint for an index $e \leq \eta_i$. In practice, some actions can occur in parallel and thus do not constitute a single linear sequence, which is required for the clause derivations' acyclicity and soundness (Theorem 1). However, any actions applicable in parallel are functionally independent and can thus always be ordered into such a sequence. Lastly, if a checker i receives an invalid or erroneous input,

it terminates – leaving C_i in its last, valid state and not modifying B any further. As such, we can equivalently assume that no such errors occur.

Under the described correspondence, our setup satisfies C1 because a clause can only enter a checker's database via `read`, `learn`, or `fetch`; no clause fingerprints exist initially (A1); and only `push` actions emit a clause fingerprint (A1). Condition C2 is met because fingerprints ensure that any participating checker i with $\eta_i = e$ operates on the intended formula F_e (A1). If we consider a checker x that operates not on F but on formula $F' \neq F$ (with accordingly different fingerprints), then all outputs of x are invalid inputs to checkers operating on F; therefore x is, functionally, not part of the procedure. Lastly, a checker only emits $f_K^{F_e}(c)$ if $\eta_i = e$ and it only imports incoming clauses c with fingerprint $f_K^{F_e}(c)$ if $e \leq \eta_i$, which satisfies C3. □

Note that we abstracted away clause deletions in the above proof. Deleted clauses (for LIDRUP) can be modelled as *inactive* parts of the C_i that are no longer used in derivations and exports. Original clauses from F_e are never deleted by checkers that are configured to check satisfying assignments.

Finally, we obtain an end-to-end correctness result, also when considering assumptions of individual queries and including the confirmer module.

Theorem 3. *Assume* (A1). *Consider a parser executable that successively emits pairs* $(F_e, f_K(F_e))$ *of an incremental SAT task* $P = (F, Q)$. *If a confirmer executed on* F *accepts a witness* $W = (e, s, \pi, w)$, *then for any query* $q = (e, A) \in Q$, q *is satisfiable if* $s = 10 \wedge w = A$ *and unsatisfiable if* $s = 20 \wedge w \subseteq A$.

Proof. Since the confirmer accepts W, π represents the (un)satisfiability of the corresponding F_e with (failed) assumptions w. Due to (A1), such a fingerprint can only originate from a checker i with $\eta_i = e$ producing this result. For $s = 10$ (SAT), we have $\pi = f_K^{F_e}(\top, w)$ with $w = A$, which signifies that the checker checked a model against F_e and, additionally, against each literal in A. As such, q is satisfiable. For $s = 20$ (UNSAT), $\pi = f_K^{F_e}(\bot, w)$ signifies that the checker confirmed that the entailed clause $\neg w$ is part of its database. Due to Theorem 2, $F_e \wedge w$ is unsatisfiable. Due to $w \subseteq A$, the query $q = (e, A)$ is unsatisfiable. □

Note that our argument leaves the number of actors open and (largely) how they interact. A direct consequence is that any set of incremental SAT tasks can *freely share clauses* derived at F_e if the receiver operates on a later $F_{e'}$. For singular queries without assumptions, our proof provides a more formal argument of correctness for non-incremental ImpCheck than originally [31].

We ensure (A1) in practice by choosing a high-quality 128-bit keyed hash function [1] that rules out spurious valid fingerprints and coincidental second preimages beyond any reasonable doubt. Assuming a non-adversarial environment,[8] this hashing in fingerprints is part of our approach's *trusted core*, which,

[8] A non-adversarial environment is an implicit assumption in all widespread propositional proof checking approaches, including formally verified ones. As an example, an adversary to a verified sequential proof checker (e.g., [8]) could manipulate the formula file (while it is being parsed) or the underlying filesystem.

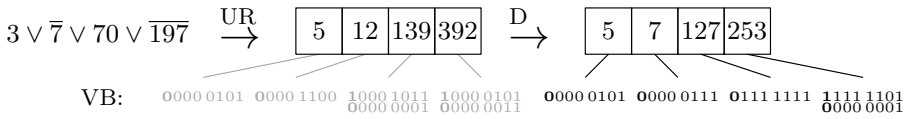


Fig. 3. Compression of a clause whose unsigned representation (UR) would, naïvely, take $4 \times 4 = 16$ bytes to store. Immediately applying variable-bytelength (VB) results in six bytes whereas first employing a differential encoding (D) results in five bytes.

as usual in proof checking, already includes the trusted executables and other aspects like file I/O (for parsing F) and integrity of main memory.

6 Reducing Memory Requirements

Real-time checking requires significant amounts of main memory since each clause is present once in the solver thread and once in the checker process [31]. As a (partial) mitigation, we introduce clause compression to the checker units.

Firstly, we compress clauses based on a *differential coding* (Fig. 3). Given a clause $c = l_1 \vee \cdots \vee l_k$, where $l_1 < \cdots < l_k$ in terms of the literals' unsigned integer representation, we store c as sequence $S_c = \langle l_1, l_2 - l_1, \ldots, l_k - l_{k-1} \rangle$. Due to the literals being sorted, all elements of S_c are positive. We then encode S_c with *variable bytelength*, where one bit per byte signifies whether a number ends at this byte (cf. [16]). Applying this encoding to differences between neighbored literals, we expect most individual values to only take a small number of bytes.

Secondly, we employ *bit-stuffing* for small clauses to avoid some memory (de)allocations. Whenever a clause's literals, after compression, take up ≤ 7 bytes, we store them directly in the 64-bit pointer field rather than pointing to a separate chunk of memory. The eighth byte of the repurposed pointer field is masked in a way that unambiguously marks the pointer as repurposed.

With our compression, fetching a prerequisite clause for checking a derivation incurs added work: The clause's literals are decompressed on-the-fly by reversing the compression steps. We refrained from compressing clauses at bit rather than byte granularity since this would introduce further complexity and overhead.

7 Implementation

We implemented our real-time checking applications (available online [33]) in C99, building upon the ImpCheck codebase [31]. Our project is now comprised of roughly 1800 effective lines of code (including some non-essential logging and debugging functionality), uses plain file I/O on *named pipes* for inter-process communication, and relies only on the C standard library.

The provided *source* from which the trusted parser reads F is a single file path across all incremental calls. If a set of input clauses resides in the user's main memory (e.g., after an internal encoding procedure), the clauses can be written

to a named pipe that is treated as the parser's input. With each call, the parser continues to read the source from where it last left off up until a termination character. This allows us to input clauses reactively based on the last solve call's outcome. As a slight deviation from our formalism, our framework also allows the solver to parse and load *batches* of $x > 1$ input clauses with a single fingerprint – skipping $x - 1$ intermediate fingerprints if they are never needed (after loading).

As a first major use case, we integrated our approach in Mallob [27] and its distributed incremental SAT solver MallobSat [30,32] (see Section 3). We use CaDiCaL [5] as MallobSat's currently only LIDRUP producing solver backend and modified its internal LIDRUP interface analogous to the original ImpCheck approach [31]: each proof line is forwarded to MallobSat, where dedicated threads write proof lines to checker processes and read their responses. Since solver threads wait for a checker confirmation before reporting a result, the user does not need to screen for potential checker errors but can consider each returned result as checked (and run the confirmer on returned fingerprints for full confidence). Our incremental checking already proved valuable during development, revealing several logical errors and bugs in early versions of our integration.

As sequential baseline approach, we run a stand-alone (C++) executable that reads and solves an incremental SAT input using CaDiCaL [5] as a library. CaDiCaL writes its LIDRUP reasoning to a named pipe while a separate process running `lidrup-check` [11] reads and validates this information in real time.

8 Experimental Setup

Throughout all experiments, we use the HPC cluster HoreKa [21] with 76-core nodes (two sockets with 38 cores each) connected by fast InfiniBand interconnect. We further use Mallob's CaDiCaL [5] backend for all experiments.

Tab. 1 lists all benchmark sets used. In terms of non-incremental SAT solving, we include the 400 instances from the 2024 International SAT Competition [20]. For incremental solving, benchmark problems are more scattered and less accessible, and consequently we hand-crafted a number of benchmarks from different application domains. First, we use incremental SAT tasks emitted by SMT solver

Table 1. Instance sets used in our evaluation. For each set we indicate whether it features incremental SAT solving ("incr.") and/or flexible scheduling of multiple, related sub-tasks with clause sharing across them ("multi"), as well as the general domain, the employed system, the origin of instances, and the number of "top level" tasks.

incr.	multi	Domain	System	Instances	#
×	×	Generic SAT	N/A	SAT Comp. 2024	400
✓	×	SMT	Bitwuzla	SMT-LIB: non-inc. QF_BV	389
✓	×	BMC	2LS	SV-Comp 2024	415
✓	×	Planning	Lilotane	IPC 2023	537
✓	✓	EDA	Mallob C&C	28- to 32-bit `cruxmiter`	4

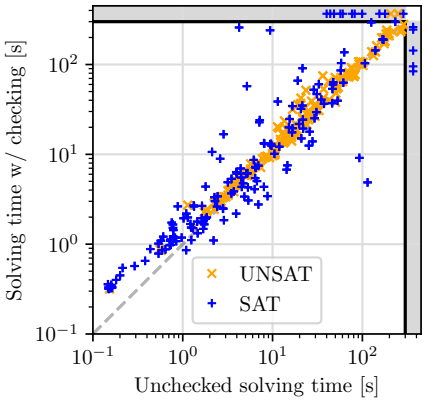
Bitwuzla [25] when run on SMT-LIB's `QF_BV` instances (quantifier-free bit vectors) without *incremental SMT* calls. Mathias Preiner provided this selection of instances, sampled by difficulty as in ref. [35]. Our second benchmark set was provided by Oguz Mutlu and is based on problems from the C program verification track of SV-Comp 2024, processed with the model checker 2LS [28]. The set features an incremental SAT instance for each checked property in each problem taking 2LS between 60 s and 900 s. Thirdly, we include incremental SAT instances extracted from the SAT-based hierarchical planner Lilotane [29], using problems from the International Planning Competition [3]. These were already used for Schreiber's initial study on distributed incremental SAT solving [32].

Lastly, as a proof of concept for settings with multiple solving tasks on deviating assumptions, we run a simple Cube & Conquer (C&C) setup [19] in Mallob ($\approx$ 300 lines of code) on a few *miter* instances from electronic design that were used before to assess distributed C&C [15]. The formula is split up into a fixed number of sub-problems (*cubes*) using CaDiCaL's built-in, look-ahead based cubing method. Each cube is represented as a set of assumption literals, which allows for clause sharing across solving tasks that solve different cubes.

The developed software and gathered data is available online [33].

9 Results

First, we examine the slowdown for *non-incremental* solving incurred by our proof checking, i.e., when used just like non-incremental ImpCheck [31]. Fig. 4 shows results at 16 nodes (1216 cores). Here and in the following, we compare our trusted setup with a run of MallobSat where proof logging is disabled entirely, which we refer to as *unchecked solving*. Remarkably, we measured a mean slowdown of only 25.3% over unchecked solving – much less than the 37.7% reported



	all		sat.		unsat.	
	#	PAR	#	PAR	#	PAR
Unchecked	339	92.3	163	129.4	176	110.8
Checked	330	105.8	157	151.4	173	119.8

Fig. 4. Performance with vs. without checking at 16 nodes for non-incremental (SAT Comp. 2024) benchmarks – by-instance comparison (left) and summary (right), showing solved instances (#) and a Penalized Average Runtime (PAR, in seconds) that attributes a running time of twice the time limit ($2 \times 300\,\text{s} = 600\,\text{s}$) to each timeout.

Table 2. Basic performance metrics for incremental benchmarks: Number of solved instances and Penalized Average Runtime (PAR-2, in wallclock seconds), which attributes twice the time limit ($2 \times 300\,\text{s} = 600\,\text{s}$) to each unsolved instance.

	SMT (#389)		BMC (#415)		Plan (#537)	
Approach	#	PAR	#	PAR	#	PAR
CaDiCaL+lc	383	11.2	285	374.2	516	44.0
Unchecked 1x76	388	4.5	409	93.5	528	18.3
Unchecked 4x76	388	4.3	408	85.5	528	17.7
Unchecked 16x76	388	4.0	397	98.6	529	15.8
Checked 1x76	387	5.6	362	172.2	526	20.9
Checked 4x76	387	5.7	382	137.3	527	19.9
Checked 16x76	388	4.2	371	147.6	530	15.8

for original ImpCheck at the same scale [31]. Possible reasons are (a) deviating benchmarks, (b) MallobSat's latest configuration sharing fewer clauses [36], and (c) low-level optimizations we introduced to proof checking. Ignoring trivial solving times (below 1 s), our checking incurs an overhead of only 21.3% (29.2% for satisfiable, 15.5% for unsatisfiable inputs). As such, the experiment confirms that our checking approach comes with little added cost in non-incremental settings.

Next, we proceed with the results obtained on incremental instances. Tab. 2 shows performance results for the sequential baseline as well as checked and unchecked MallobSat at 1, 4, and 16 nodes (76, 304, and 1216 cores). While the scalability of incremental SAT solving with MallobSat is not the focus of this work, we note that not all of the tested incremental benchmarks were able to profit from large-scale parallelism. While the planning and SMT benchmarks yield mostly consistent scaling behavior, the BMC benchmark actually reveals deteriorating performance when increasing the scale of computing from 4 to 16 nodes. This regression is caused by the very large number of incremental SAT calls per instance (on average 2619 calls per instance compared to 11.5 and 3.2 calls p.i. for Planning and SMT respectively), almost all of which are trivial.

The relative slowdown incurred by our checking approach is shown in Fig. 5. The boxplots confirm that the slowdown remains constant regardless of the scale of solving – as for non-incremental ImpCheck [31] – and stay below 33% for the median input. As Fig. 5 (right) shows, overheads are more erratic and often higher than average for BMC inputs, which, again, we attribute to the high number of trivial increments in this benchmark. In particular, a solver thread that solved a certain increment needs to block and wait for its checker's confirmation of the obtained result before it can report the result, which occasionally takes an additional few milliseconds compared to an unchecked setup.

Next, we assess our approach's memory usage. To discern meaningful differences in RAM usage, we measured each execution's global RAM usage after 30 s of elapsed wallclock time. Given a pair of runs, we compute a memory ratio for each input where both runs reached 30 s. At 48 cores, our checking *without* com-

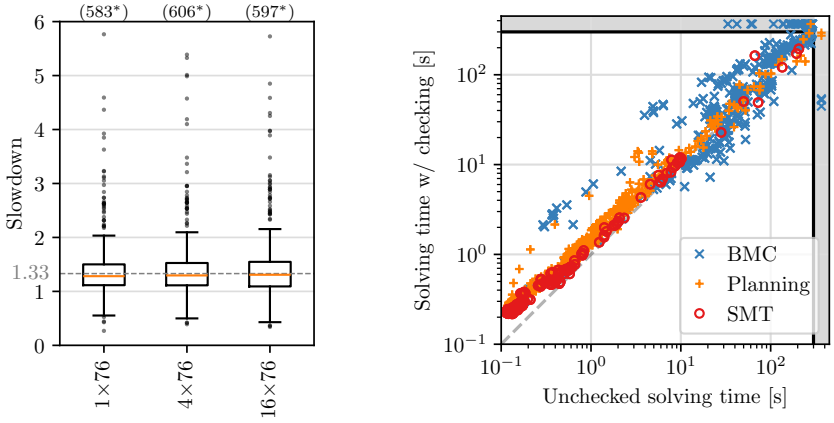


Fig. 5. Results on incremental benchmarks (planning, BMC, SMT). Left: Distribution over the relative overhead incurred by real-time proof checking compared to unchecked solving, at 1/4/16 compute nodes and over instances that took sequential CaDiCaL at least 1 s to solve. The numbers at the top denote the total number of data points used for each box plot, of which some outliers at the top have been cut off (*). Right: Per-instance comparison of running times with vs. without proof checking at 16 nodes.

pression increases memory usage by 45.2% (geom. mean) over unchecked MallobSat. Our compression reduces this overhead to 31.7%. Overall, distributed solving with compressing checkers uses 9.5% less memory than with naïve checkers. The compression's impact on performance is negligible (<1% mean deviation).

Lastly, we consider a more complex use case, using a setup with 16×4 processes à 19 cores. A single-threaded Cube&Conquer controller generates 1024 cubes and then runs 19 distributed incremental SAT solving engines, initially

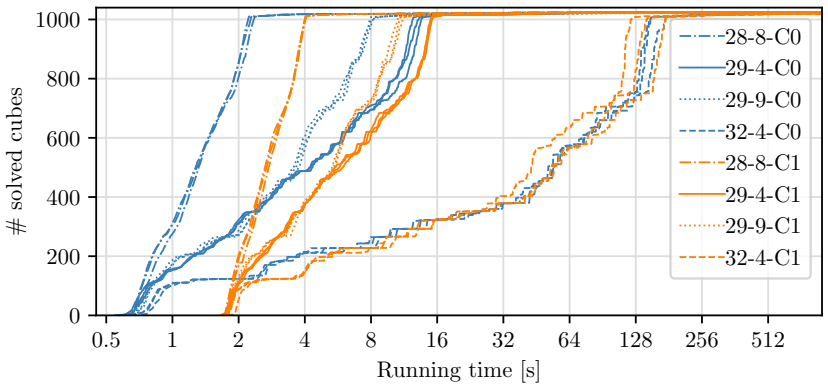


Fig. 6. Cube & Conquer experiment with (C1) and without checking (C0) on four different SAT instances `cruxmiter`(*bit-width*)`seed`(*seed*)`.cnf` with several runs each.

queried on the first 19 cubes. Whenever a SAT query returns, the controller sends back the next unsolved cube as the next SAT query, until all cubes are processed or the time limit of 900 s is reached. Clauses are exchanged within and across the co-existing distributed SAT solvers, checked by our checking approach.

Fig. 6 shows the number of solved cubes over solving time for our checked vs. unchecked setup. Initially, the checked setup incurs more than one second of additional upstart overhead due to initializing many trusted checker processes. Despite this initial bottleneck, the checked runs' relative slowdowns quickly diminish, e.g., down to about 22% for `29-4`, where all cubes are solved after 343 vs. 281 seconds (on average). After about eight seconds, the checked runs on the hardest instance (`32-4`) become difficult to discern from the unchecked runs: Significant random deviations in cube solving times cause the overall system states to increasingly diverge over time, thus causing high variances.

To summarize, our evaluations show that reliably checking the reasoning of a distributed incremental clause-sharing SAT solver in real-time is not only possible and viable but also incurs little overhead in terms of running times and memory usage. Moreover, this checking can be employed even in complex setups featuring multiple deviating incremental SAT solving tasks running in parallel, operating on different assumptions, and sharing clauses among each another.

10 Conclusion

The work at hand presents the first proof checking approach for distributed incremental SAT solving. Our real-time proof checking model covers advanced aspects like flexible re-scheduling, asynchronous and diverging solver units, and deviating assumptions. Experiments with a state-of-the-art distributed incremental solver show that a small and self-contained implementation of our checking incurs running time overhead over unchecked solving of about 33%. Optional compression techniques in our checkers reduce the main memory overhead from 45% to 32%.

Concerning future work, we aim to devise a formally verified implementation of our approach (e.g., via CakeML [39]). In the longer term, we are interested to further extend our solving and checking model towards settings where not only assumptions but also permanent clauses may deviate and clauses can still be shared under certain conditions, e.g., when all solver units operate on subsets of a common base formula (as done for example by Schreiber at al. [34]).

Acknowledgments. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research (Germany). We further acknowledge support by an Amazon Research Award (Fall 2023). The authors wish to thank Yong Kiam Tan for his idea of introducing a "virtual" clause database that is defined to contain all fingerprinted clauses. The authors also thank Florian Pollitt for helpful insights on CaDiCaL's LIDRUP backend, Mathias Preiner and Oguz Mutlu for providing SMT and BMC benchmarks, and David Basin for helpful remarks on ImpCheck.

References

1. Aumasson, J.P., Bernstein, D.J.: SipHash: a fast short-input PRF. In: International Conference on Cryptology in India. pp. 489–508. Springer (2012). https://doi.org/10.1007/978-3-642-34931-7_28
2. Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_1
3. Behnke, G., Höller, D., Bercher, P. (eds.): Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020) (2021), https://ipc2020.hierarchical-task.net/publications/IPC2020Booklet.pdf
4. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifier (uri): Generic syntax. Tech. rep. (2005). https://doi.org/10.17487/rfc3986
5. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0. In: International Conference on Computer Aided Verification. pp. 133–152. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_7
6. Cantor, G.: Ein Beitrag zur Mannigfaltigkeitslehre. Journal für die reine und angewandte Mathematik (Crelles Journal) **1878**(84), 242–258 (1878). https://doi.org/10.1515/crelle-1878-18788413
7. Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. p. 11–20. ICSE-SEIP '20 (2020). https://doi.org/10.1145/3377813.3381347
8. Cruz-Filipe, L., Heule, M.J.H., Hunt, Warren A., J., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Proc. CADE. Lecture Notes in Computer Science, vol. 10395, pp. 220–236 (2017). https://doi.org/10.1007/978-3-319-63046-5_14
9. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/10.1016/s1571-0661(05)82542-3
10. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.: Satisfiability modulo user propagators. Journal of Artificial Intelligence Research **81**, 989–1017 (2024). https://doi.org/10.1613/jair.1.16163
11. Fazekas, K., Pollitt, F., Fleury, M., Biere, A.: Certifying incremental SAT solving. In: Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR). vol. 100, pp. 321–340 (2024). https://doi.org/10.29007/pdcc
12. Fazekas, K., Pollitt, F., Fleury, M., Biere, A.: Incremental inprocessing rules beyond resolution. In: Hoenicke, J., Janota, M., Niemetz, A., Tourret, S. (eds.) Proceedings 16'th Pragmatics of SAT International Workshop (POS'25). pp. 190–200. No. 4008 in CEUR Workshop Proceedings (2025), https://ceur-ws.org/Vol-4008
13. Fichte, J.K., Le Berre, D., Hecher, M., Szeider, S.: The silent (r)evolution of SAT. Comm. ACM **66**(6), 64–72 (2023). https://doi.org/10.1145/3560469
14. Froleyks, N., Biere, A.: Single clause assumption without activation literals to speed-up IC3. In: Formal Methods in Computer Aided Design (FMCAD). pp. 72–76. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_15
15. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with Paracooba. In: Theory and Applications of Satisfiability Testing (SAT). pp. 114–122. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_9

16. Heule, M.J.H.: The DRAT format and DRAT-trim checker. CoRR **abs/1610.06229** (2016)
17. Heule, M.J.H.: Proofs of unsatisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 635–668. IOS Press (2021). https://doi.org/10.3233/FAIA200998
18. Heule, M.J.H., Biere, A.: Proofs for satisfiability problems. In: All about Proofs, Proofs for All (APPA), Math. Logic and Foundations, vol. 55. College Pub. (2015)
19. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
20. Heule, M.J., Iser, M., Järvisalo, M., Suda, M.: Proceedings of SAT Competition 2024: Solver, benchmark and proof checker descriptions (2024), https://researchportal.helsinki.fi/files/324666039/sc2024-proceedings.pdf
21. Hardware overview of HoreKa (2026), https://www.nhr.kit.edu/userdocs/horeka/hardware/
22. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
23. Michaelson, D., Schreiber, D., Heule, M.J.H., Kiesl-Reiter, B., Whalen, M.W.: Unsatisfiability proofs for distributed clause-sharing SAT solvers. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 348–366. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_18
24. Michaelson, D., Schreiber, D., Heule, M.J., Kiesl-Reiter, B., Whalen, M.W.: Producing proofs of unsatisfiability with distributed clause-sharing SAT solvers. Journal of Automated Reasoning **69**(2), 12 (2025). https://doi.org/10.1007/s10817-025-09725-w
25. Niemetz, A., Preiner, M.: Bitwuzla. In: International Conference on Computer Aided Verification (CAV). vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1
26. Pollitt, F., Fleury, M., Biere, A.: Faster LRAT checking than solving with CaDiCaL. In: Theory and Applications of Satisfiability Testing (SAT). Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.SAT.2023.21
27. Sanders, P., Schreiber, D.: Decentralized online scheduling of malleable NP-hard jobs. In: Euro-Par 2022: Parallel Processing. pp. 119–135. Springer (2022). https://doi.org/10.1007/978-3-031-12597-3_8
28. Schrammel, P., Kroening, D.: 2LS for program analysis - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). vol. 9636, pp. 905–907. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_56
29. Schreiber, D.: Lilotane: A lifted SAT-based approach to hierarchical planning. JAIR **70**, 1117–1181 (2021). https://doi.org/10.1613/jair.1.12520
30. Schreiber, D.: MallobSat and MallobSat-ImpCheck in the SAT Competition 2024. In: SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions. pp. 21–22 (2024)
31. Schreiber, D.: Trusted scalable SAT solving with on-the-fly LRAT checking. In: Theory and Applications of Satisfiability Testing (SAT). pp. 25:1–25:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPIcs.SAT.2024.25
32. Schreiber, D.: Distributed incremental SAT solving with Mallob: Report and case study with hierarchical planning. https://arxiv.org/abs/2505.18836 (2025)

33. Schreiber, D., Fleury, M., Fazekas, K., Biere, A.: Experimental data of TACAS 2026 publication "Real-time Proof Checking for Distributed Incremental SAT Solving". Zenodo Dataset (2026). https://doi.org/10.5281/zenodo.18330440
34. Schreiber, D., Jabs, C., Berg, J.: From scalable SAT to MaxSAT: Massively parallel solution improving search. In: Symposium on Combinatorial Search (SoCS) (2025). https://doi.org/10.1609/socs.v18i1.35984
35. Schreiber, D., Niemetz, A., Preiner, M.: Massively parallel bit-precise verification with Bitwuzla and Mallob. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2026), to appear
36. Schreiber, D., Rigi-Luperti, N., Biere, A.: Streamlining distributed SAT solver design. Theory and Applications of Satisfiability Testing (SAT) **23**, 1–23 (2025). https://doi.org/10.4230/LIPIcs.SAT.2025.27
37. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: Theory and Applications of Satisfiability Testing (SAT). pp. 518–534. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_35
38. Schreiber, D., Sanders, P.: MallobSat: Scalable SAT solving by clause sharing. Journal of Artificial Intelligence Research **80**, 1437–1495 (2024). https://doi.org/10.1613/jair.1.15827
39. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 223–241. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_12
40. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics (ISAIM) (2008), http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf