Master's Thesis

Reimplications for Parallel Clause Sharing

Florian Pollitt

Examiners: Prof. Dr. Armin Biere,
Prof. Dr. Christoph Scholl
Co-Adviser: Dr. Mathias Fleury

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Computer Architecture

March 26, 2024

Writing Period

02. 10. 2023 – 02. 04. 2024

First Examiner

Prof. Dr. Armin Biere

Second Examiner

Prof. Dr. Christoph Scholl

Co-Adviser

Dr. Mathias Fleury

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 26.03.24 — Place, Date

F. Pollitt — Signature

Abstract

Sharing clauses in a parallel SAT solver can disrupt the search by forcing the solver to backtrack. Chronological backtracking counteracts this by making backtracking optional. However, it also breaks some invariants of classic CDCL which can have negative impact on the solver. Reimplication restores these invariants without backtracking by finding and fixing missed implications. The technique is complex and requires changing the watch invariants of Boolean constraint propagation and the management of the trail of assigned literals among other things.

In this work we provide an analysis of three different implementations of reimplication where two of them are implemented by us, one for this thesis and one in a previous project. Our implementations are based on a new intuition on reimplication which simplifies understanding and leads to a different approach for trail management. In an extensive experimental section we focus on the impact of reimplication on our parallel SAT solver Gimsatul and complement the previously lacking analysis with detailed statistics and inconclusive run-time results.

Contents

List of Figures

List of Algorithms

1 Introduction

Propositional satisfiability (SAT) solving is used for many applications in industry, e.g., in formal verification of hardware [1]. Big companies like Intel have adopted formal verification approaches, especially in safety critical applications [2]. Amazon Web Services provide formal reasoning based on SAT solving for billions of users [3]. Therefore, achieving progress in this field continues to be of big interest.

Conflict driven clause learning (CDCL) took off with the introduction of GRASP [4] in sovers such as CHAFF [5] in the late 1990s and early 2000s. Since then, it has been the main paradime used in SAT solving and has shown great success. There is a steady stream of novel ideas for extending and improving CDCL based SAT solvers. Recent developments include the addition of chronological backtracking [6] which breaks CDCL invariants previously considered crucial. Fortunately, small adjustments to the solver can restore correctness and completness without these invariants [7]. However, there are unwanted side-effects to applying chronological backtracking, like repeating previously performed propagations for correctness.

The reimplication procedure proposed by Nadel [8] is a novel method to fix implications on lower decision levels. This way, repropagation is avoided and other invariants are restored. In this work we provide a detailed analysis of two new implementations of reimplication and compare them to the original implementation by Nadel. Furthermore, we focus on its impact on parallel SAT solving, where clause sharing leads to increased numbers of chronological backtracking.

2 Related Work

2.1 Chronological Backtracking

CDCL evolved from the decision procedure of Davis, Logemann and Loveland (DLL) [9]. Originally, the DLL algorithm did not perform conflict analysis. When encountering a conflict it was forced to backtrack chronologically, level by level. With conflict analysis in modern solvers, this has been improved by backjumping, i.e., undoing unused decision levels below the conflict level. However, if there are many such decision levels, backjumping might erase work which is repeated later. This observation lead to the introduction of chronological backtracking in CDCL solvers by Nadel and Ryvchin in 2018 [6]. It can be applied with a simple heuristic and small changes to the solver. Intuitively, chronological backtracking improves performance when the decision levels which would be undone by backjumping are repeated later in the search. It has been formalized and proven to be correct [7]. The reimplication technique [8] which is the main focus of this thesis serves to improve chronological backtracking. However, there are contexts exceeding pure SAT solving in which reimplication can be triggered independently of chronological backtracking.

2.2 Incremental SAT Solving

Incremental SAT solvers like MiniSAT [10] allow the user to pose several similar problems in sequence and can reuse information derived from previous problems to

solve the current one. Incremental SAT has various applications, from using MiniSAT to verify ciruits with bounded model checking [11] to solving optimisation problems with maximal satisfiability solvers such as Pacose [12]. One of the main features of incremental SAT solving allows adding clauses in between each solve call. When applying incremental lazy backtracking [8] this can trigger reimplication without chronological backtracking.

2.3 User Propagator

The IPASIR-UP interface [13] allows adding clauses and propagating literals during the search phase of the SAT solver. This is similar to importing clauses in parallel SAT or adding clauses in incremental SAT and can lead to out-of-order assignments and missed implications which will be introduced later in this thesis.

2.4 Trail Saving

There is an alternative to chronological backtracking called trail saving by Hickney and Bacchus [14]. Trail saving can be applied after backtracking to speed up the reassignment of the variables on the old trail. It serves a similar goal as chronological backtracking without disrupting other parts of the solver which can be negatively impacted by chronological backtracking.

2.5 SAT Modulo Theories

There has been work on reimplication and chronological backtracking in the context of SMT solving [15]. In his thesis, Coutelier implemented chronological backtracking and reimplication in the SMT solver veriT.

3 Background

In this chapter we introduce the CDCL framework step by step. We start by defining SAT solving as well as relevant concepts, then proceed with the core CDCL algorithms. In the second part of this chapter we introduce parallel SAT solving which extends CDCL with clause sharing. Finally, we refine all previous algorithms for chronological backtracking. For more details on SAT solving and CDCL we refer to the Handbook of Satisfiability [16].

3.1 SAT solving

Definition 1 (Boolean Variables)**.** *A* (Boolean) variable v *can take values in* $\{1, 0, -1\}$*. Variables with value* 0 *are called unassigned, otherwise they are called assigned.*

Definition 2 (Literals)**.** *A* literal ℓ *is either a variable* v *or its negation* $-v$*.* $-\ell$ *is the negation of a literal and* $|\ell|$ *refers to the variable.*

Definition 3 (Clauses)**.** *A* clause C *is the conjunction of literals* $C = \ell_1 \vee \cdots \vee \ell_n$*. We typically use set notation for clauses, i.e.,* $C = \{\ell_1, \ldots, \ell_n\}$*.*

Definition 4 (Formula)**.** *A* Boolean formula $\mathcal{F}$ *in conjunctive normal form (CNF) is a disjunction of clauses* $\mathcal{F} = C_1 \wedge \cdots \wedge C_k$*. As for clauses, formulas can be expressed with set notation* $\mathcal{F} = \{C_1, \ldots, C_k\}$*.*

A formula $\mathcal{F}$ implicitly defines a set of variables by the occuring literals. We identify this set with $\mathcal{V}$ and further define $\mathcal{L} := \{v \mid v \in \mathcal{V}\} \cup \{-v \mid v \in \mathcal{V}\}$ as the corresponding set of literals.

Definition 5 (Assignments)**.** *A function $\sigma : \mathcal{V} \to \{1, 0, -1\}$ is called* partial assignment*. If σ assigns all variables in $\mathcal{V}$ it is called a* full assignment*.*

We abuse function notation by extending assignments to literals in the obvious way:

$$\sigma(\ell) := \begin{cases} \sigma(v), & \text{if } \ell = v \\ -\sigma(v), & \text{if } \ell = -v \end{cases}$$

Definition 6 (Satisfying Formulas)**.** *The (partial) assignment σ* satisfies *a literal ℓ if $\sigma(\ell) = 1$. A clause C is satisfied if* at least one *literal in C is satisfied. A formula $\mathcal{F}$ is satisfied if* all *the clauses in $\mathcal{F}$ are satisfied.*

Definition 7 (Falsifying Formulas)**.** *Correspondingly, σ* falsifies *a literal ℓ if $\sigma(\ell) = -1$. A clause C is falsified if* all *the literals in C are falsified. A formula $\mathcal{F}$ is falsified if* at least one *clause in $\mathcal{F}$ is falsified.*

For any set of literals (or variables) L we define $\mathsf{satisfied}_\sigma(L) := \{\ell \in L \mid \sigma(\ell) = 1\}$ and $\mathsf{falsified}_\sigma(L) := \{\ell \in L \mid \sigma(\ell) = -1\}$ as the subset of satisfied and falsifed literals respectively. Furthermore, we define $\mathsf{assigned}_\sigma(L) := \{\ell \in L \mid \sigma(\ell) \neq 0\}$ and $\mathsf{unassigned}_\sigma(L) := \{\ell \in L \mid \sigma(\ell) = 0\}$ as the set of assigned and unassigned literals in L. Note that both definitions apply to clauses.

Definition 8 (Model)**.** *A full assignment M which satisfies $\mathcal{F}$ is called a* model*, also written $M \models \mathcal{F}$.*

Definition 9 (Solution)**.** *Any Model M of the Formula $\mathcal{F}$ together with the claim* SATISFIABLE (*SAT*) *is a solution to the corresponding SAT problem. If no model exists then the solution is the claim* UNSATISFIABLE (*UNSAT*)*.*

Definition 10 (Consistent Assignments)**.** *Two (partial) assignments σ, σ' are* consistent *if they agree on the value for all variables that are assigned by both σ and σ'. That is, if $\sigma(v) = 1$ for any variable v, then $\sigma'(v) \neq -1$ and vice versa.*

We say that σ' extends *σ if they are consistent and the set of unassigned variables of σ' is a subset of the unassigned varibles of σ: $\mathsf{unassigned}'_\sigma(\mathcal{V}) \subsetneq \mathsf{unassigned}_\sigma(\mathcal{V})$*

Definition 11 (Unit Clauses and Conflicts)**.** *Given a partial assignment σ, a clause C is called* unit*, if σ falsifies all the literal of C except one, which is unassigned. The unassigned literal is called* unit literal*. If the clause is completely falsified instead, it is called conflicting or* conflict*.*

The only way to consistently extend σ and not falsify some unit clause is to assign the unit literal to 1.

Definition 12 (Resolution)**.** *Given two Clauses C, C', where $\ell \in C$, $-\ell \in C'$, $-\ell \notin C$ and $\ell \notin C'$. Then the resolvent of C and C' on ℓ is defined as $C \otimes_\ell C' := (C \setminus \{\ell\}) \cup (C' \setminus \{-\ell\})$. For a Formula $\mathcal{F}$ with $C, C' \in \mathcal{F}$ any model of $\mathcal{F}$ satisfies $C \otimes_\ell C'$. Therefore the solver can add any clauses derived by resolution to the formula without losing correctness or completeness.*

3.2 Conflict Driven Clause Learning

As part of its global state, the solver builds a partial assignment σ. It can assign values in the following two ways:

Definition 13 (BCP (Alg. 1))**.** *The act of finding conflicts and unit clauses, and assigning unit literals is called* Boolean constraint propagation (BCP)*. If a conflict exists it is returned, otherwise* BCP *returns $\bot$ when no more unit clauses exist.*

Definition 14 (DECIDE (Alg. 2))**.** *If there are no unit clauses and no conflicts, then the solver can make a* decision*, assigning an unassigned variable to* 1 *or* -1*.*

The order of assignments is stored on the trail. Classically, the trail is defined as a sequence of literals. This works well because newly assigned literals are always added to the top, while unassigning literals through backtracking removes blocks of literals from the top. However, chronological backtracking breaks the latter invariant and reimplication also breaks the former. We give a more abstract defintion of the trail which can easily be extended to chronological backtracking and reimplication.

Definition 15 (Trail)**.** $\tau : \mathbb{N} \to \mathsf{satsified}_\sigma(\mathcal{L}) \cup \{\bot\}$ *is called* trail*. It is used to store the order in which literals are assigned, that is, whenever a literal* ℓ *is assigned to true the solver sets* $\tau(n) = \ell$ *for the smallest number* n *with* $\tau(n) = \bot$*. It keeps the invariant that the position of a satisfied literal* ℓ *on the trail is uniquely determined. To access the position of literals we define* $\tau^{-1}(\ell) := n$ *such that* $\tau(n) = \ell$*. As variables cannot be assigned to multiple values at once we can extend this notation:*

$$\tau^{-1}(v) := \begin{cases} \tau^{-1}(v), & \text{if } \sigma(v) = 1 \\ \tau^{-1}(-v), & \text{if } \sigma(v) = -1 \end{cases}$$

The size of the trail $|\tau|$ *is defined as the smallest* $n \in \mathbb{N}$ *where* $\tau(n) = \bot$*.*

Definition 16 (Decision Level)**.** *Each assigned variable is given a* decision level *by* $\delta : \mathsf{assigned}_\sigma(\mathcal{V}) \to \mathbb{N}$*. The decision level of an assigned variable* v *is always equal to the number of decisions on the trail before and including* $\tau^{-1}(v)$*. We extend* δ *to literals* $\delta(\ell) := \delta(|\ell|)$ *and sets of literals* $\delta(L) := \max(0, \{\delta(\ell) \mid \ell \in \mathsf{assigned}_\sigma(L)\})$*.*

A conflict on decision level 0, also global conflict, implies that the formula is unsatisfiable. To simplify handling of decision levels, the solver has a global value *level*, which is always equal to current the number of decisions.

Definition 17 (Reasons)**.** *The solver has a* reason function $\rho : \mathsf{assigned}_\sigma(\mathcal{V}) \to \mathcal{F} \cup \{\bot\}$ *which allocates a clause to each literal when it is assigned. If it is a decision we mark it with the decision reason* $\bot$*. Otherwise it assigned during* BCP*, and the allocated reason is the responsible unit clause* C*.*

Definition 18 (Assign (Alg. 3))**.** *Whenever a literal is assigned, its decision level and reason are updated and the trail is extended as described above.*

Definition 19 (Backtracking (Alg. 4))**.** *Undoing assigments is always done in blocks from the top of the trail up to a certain decision level.*

Conflict analysis is performed whenever a conflict is found by BCP.

Definition 20 (Analyze (Alg. 5))**.** *The solver learns a new clause which is derived from the conflict by successive resolution steps. For every assignment which appears in the conflict with opposite sign, it is resolved with the corresponding reason clause. This is done in the order given by trail, until the new clause only has one literal left assigned on the current level.*

There are several invariants for this process. Firstly, conflicts which are found always have at least two literals assigned on the current level. This means the solver needs to resolve at least once in order to satisfy the termination criterium. Secondly, also the reason clauses which are used in the resolution steps have at least two literals assigned on the current level, where one of them is the unit literal. This means we can never remove all literals on the current level from the resulting clause. Thirdly, for each resolution step all the literals which are added to the clause are ordered on the trail below the literal which is removed. At some point the resulting clause must have exactly one literal on the current level, which might be the decision. Literals on the trail below the decision do not have to be considered for resolution.

Algorithm 1 Overview of Boolean Constraint Propagation, refined by Alg. 7

```
1: function BCP() → F ∪ {⊥}
2:     while there is a unit clause or conflict in F do
3:         if there is a conflict C ∈ F then
4:             return C
5:         end if
6:         if there is a unit clause C ∈ F with unit literal ℓ then
7:             Assign(C, ℓ)
8:         end if
9:     end while
10:    return ⊥
11: end function
```

Algorithm 2 Making Decisions

```
1: function Decide()
2:     ℓ ← any literal with σ(ℓ) = 0
3:     Assign(⊥, ℓ)
4: end function
```

Algorithm 3 Assigning Literals, refined by Alg. 13

```
1: function Assign(reason clause C, literal ℓ)
2:     if C = ⊥ then                          ▷ increase level for decisions
3:         level ← level + 1
4:     end if
5:     δ(ℓ) ← level                ▷ assign decision level, value, reason and trail
6:     σ(ℓ) ← 1
7:     ρ(ℓ) ← C
8:     n ← |τ|
9:     τ(n) ← ℓ
10: end function
```

Algorithm 4 Backtracking to a lower Decision Level, refined by Alg. 8

```
1: function Backtrack(decision level dl)
2:     n ← |τ| − 1
3:     while n > 0 and δ(τ(n)) > dl do
4:         ℓ ← τ(n)                          ▷ unassign top most literal on trail
5:         τ(n) ← ⊥
6:         n ← n − 1
7:         σ(ℓ) ← 0          ▷ no need to change ρ and δ, since ℓ is unassigned
8:     end while
9:     level ← dl                                        ▷ update decision level
10: end function
```

Definition 21 (Solve (Alg. 6))**.** *The main CDCL loop runs until a global conflict is found or the partial assignment σ is extended to a full assignment without conflicting clauses in the formula. Whenever a new assignment is made, the solver applies Boolean constraint propagation until either a conflict is found or there are no more unit clauses. If a conflict is found it is analyzed, which triggers backtracking and learns a new clause. Otherwise the solver makes a new decision. Either way a new assignment is made and the solver proceeds with BCP.*

Note that if the size of the trail $|\tau|$ equals the number of variables $|\mathcal{V}|$, all variables have been assigned. If BCP terminates without conflicting clause the solver has found a model of the formula.

3.3 Boolean Constraint Propagation with Watch Lists

The main driver of a CDCL solver is Boolean constraint propagation. It is where the solver spends most of its time [17]. This is the reason why BCP has been highly optimized in modern SAT solvers. One of these optimizations which is implemented in all modern solvers is the addition of watch lists [18] in order to efficiently find new unit clauses and conflicts.

Definition 22 (Watch Lists)**.** Watch lists $\omega : \mathcal{L} \to 2^{\mathcal{F}}$ *map each literal to a set of clauses. Every clause is always assigned to two of its literals such that it occurs in exactly two watch lists. We can access the watched literals (or watches) of a clause with* $\omega^{-1} : \mathcal{F} \to \mathcal{L} \times \mathcal{L}$.

Clauses of size one are not watched. They can be assigned at decision level 0 where they will never be unassigned. The watch lists are changed during BCP or when adding new clauses. When a literal ℓ is falsified, every clause in its watch list $\omega(\ell)$ is moved to one of its non-falsified literals if possible. It is made sure that every clause

Algorithm 5 Learning new Clauses through Conflict Analysis, refined by Alg. 9

```
1: function Analyze(conflict clause C)
2:     C' ← C
3:     n ← |τ| − 1
4:     while |{ℓ ∈ C' | δ(ℓ) = level}| > 1 do        ▷ repeat until only one literal
                                                       on the current level remains
5:         ℓ ← τ(n)
6:         if −ℓ ∈ C' then
7:             C' ← ρ(ℓ) ⊗_ℓ C'
8:         end if
9:         n ← n − 1
10:    end while
11:    ℓ ← the literal in C' with δ(ℓ) = level
12:    Backtrack(δ(C' \ {ℓ}))          ▷ unassigns ℓ but no other literal in C'
13:    F ← F ∪ C'
14:    Assign(C', ℓ)                                ▷ C' is now unit on ℓ
15: end function
```

Algorithm 6 Overview of the CDCL Loop

```
1: function Solve(CNF F) → {UNSAT, (SAT, M ⊨ F)}
2:     if ∅ ∈ F then                          ▷ formula trivially unsatisfiable
3:         return UNSAT
4:     end if
5:     while true do
6:         C ← BCP()                          ▷ returns a conflict or ⊥
7:         if C ≠ ⊥ then
8:             if δ(C) = 0 then               ▷ global conflict
9:                 return UNSAT
10:            end if
11:            Analyze(C)
12:            continue
13:        end if
14:        if |τ| = |V| then                  ▷ σ is a model
15:            return (SAT, σ)
16:        end if
17:        Decide()                           ▷ make a decision
18:    end while
19: end function
```

appears in exactly two watch lists. Moving the clause is not necessary if the other watched literal is already satisfied.

This ensures that all unit clauses and conflicts can be found by just looking at the watch lists of newly falsified literals. Literals are processed in order of their assignment. To identify which watch lists have already been processed, the solver keeps a global value *propagated* which points to the first literal on the trail that has not been processed yet.

We refine BCP to make use of watch lists (Alg. 7). This requires two more changes: The value of *propagated* is updated at the end of Backtrack (Alg. 8, Line 10) and the newly learned clauses are watched in Analyze (Alg. 9, Lines 14-18).

There are further optimizations like blocking literals [19, 20] which we will not discuss here as they extend to reimplication in an obvious way.

3.4 Parallel SAT Solving

There are many attempts to leverage the power of modern CPU's for SAT solving, which are able to compute many things at once and can be combined into clusters of hundreds or thousands of nodes. Possibly the most prominent of these is the portfolio approach as pioneered by ManySAT [21]. In this apprach, many solver instances try to solve the same problem in parallel, while being able to share information with the others. Usually the solver instances are diversified in order to explore different parts of the search space and share knowledge in form of clauses. Gimsatul [22, 23] follows the portfolio approach with shared memory, which avoids copying of clauses. It relies on fast memory access and therefore does not scale beyond single CPU's.

The clause sharing scheme selects clauses for export when they are learned and may import clauses before a decision is made. We highlight the changes to the algorithm in Solve (Alg. 10, Lines 11, 17-23) and Analyze (Alg. 11, Line 19).

Algorithm 7 Boolean Constraint Propagation with Watch Lists, replacing Alg. 1

function BCP() $\to \mathcal{F} \cup \{\bot\}$
  while $\tau(\mathit{propagated}) \neq \bot$ **do**
    $\ell_{\mathit{prop}} \leftarrow \tau(\mathit{propagated})$
    $\mathit{propagated} \leftarrow \mathit{propagated} + 1$
    for $C \in \omega(-\ell_{\mathit{prop}})$ **do**
      $\langle \ell_1, \ell_2 \rangle \leftarrow \omega^{-1}(C)$ ▷ get the two watched literals for C
      $\ell_{\mathit{other}} \leftarrow \begin{cases} \ell_2, & \text{if } -\ell_{\mathit{prop}} = \ell_1 \\ \ell_1, & \mathit{otherwise} \end{cases}$
      if $\sigma(\ell_{\mathit{other}}) = 1$ **then** ▷ no update required
        continue
      end if
      if C is unit on ℓ_{other} **then**
        ASSIGN$(C, \ell_{\mathit{other}})$
        continue
      end if
      if C is not falsified **then**
        $\ell \leftarrow$ literal in $C \setminus \{\ell_{\mathit{other}}\}$ with $\sigma(\ell) \neq -1$
        $\omega(-\ell_{\mathit{prop}}) \leftarrow \omega(-\ell_{\mathit{prop}}) \setminus \{C\}$
        $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
        continue
      end if
      return C ▷ conflict
    end for
  end while
  return $\bot$
end function

Algorithm 8 Backtracking with Watch Lists, replacing Alg. 4

function BACKTRACK(decision level dl)
  $n \leftarrow |\tau| - 1$
  while $n > 0$ and $\delta(\tau(n)) > dl$ **do**
    $\ell \leftarrow \tau(n)$ ▷ unassign top most literal on trail
    $\tau(n) \leftarrow \bot$
    $n \leftarrow n - 1$
    $\sigma(\ell) \leftarrow 0$ ▷ no need to change ρ and δ, since ℓ is unassigned
  end while
  $\mathit{level} \leftarrow dl$ ▷ update decision level
  $\mathit{propagated} \leftarrow |\tau|$ ▷ update *propagated*
end function

Algorithm 9 Watching Clauses after Conflict Analysis, replacing Alg. 5

```
1: function ANALYZE(conflict clause C)
2:     C' ← C
3:     n ← |τ| − 1
4:     while |{ℓ ∈ C' | δ(ℓ) = level}| > 1 do        ▷ repeat until only one literal
                                                       on the current level remains
5:         ℓ ← τ(n)
6:         if −ℓ ∈ C' then
7:             C' ← ρ(ℓ) ⊗_ℓ C'
8:         end if
9:         n ← n − 1
10:    end while
11:    ℓ ← the literal in C' with δ(ℓ) = level
12:    BACKTRACK(δ(C' \ {ℓ}))                ▷ unassigns ℓ but no other literal in C'
13:    F ← F ∪ C'
14:    if |C'| > 1 then
15:        ℓ' ← any literal in C' with δ(ℓ') = level      ▷ backtracking changed level
16:        ω(ℓ) ← ω(ℓ) ∪ C'
17:        ω(ℓ') ← ω(ℓ') ∪ C'                            ▷ set watches
18:    end if
19:    ASSIGN(C', ℓ)                                  ▷ C' is now unit on ℓ
20: end function
```

Algorithm 10 CDCL Loop with Import and Export of Clauses, replacing Alg. 6

```
1: function Solve(CNF F) → {UNSAT, (SAT, M ⊨ F)}
2:     if ∅ ∈ F then                                   ▷ formula trivially unsatisfiable
3:         return UNSAT
4:     end if
5:     while true do
6:         C ← BCP()                                   ▷ returns a conflict or ⊥
7:         if C ≠ ⊥ then
8:             if δ(C) = 0 then                        ▷ global conflict
9:                 return UNSAT
10:            end if
11:            AnalyzeAndExport(C)
12:            continue
13:        end if
14:        if |τ| = |V| then                           ▷ σ is a model
15:            return (SAT, σ)
16:        end if
17:        C ← Import()                                ▷ returns ⊤, ∅ or ⊥
18:        if C = ∅ then                               ▷ global conflict
19:            return UNSAT
20:        end if
21:        if C = ⊤ then                               ▷ new propagation
22:            continue
23:        end if
24:        Decide()                                    ▷ otherwise new decision
25:    end while
26: end function
```

Algorithm 11 Exporting Clauses after Conflict Analysis, replacing Alg. 5

```
1: function AnalyzeAndExport(conflict clause C)
2:     C' ← C
3:     n ← |τ| − 1
4:     while |{ℓ ∈ C' | δ(ℓ) = level}| > 1 do        ▷ repeat until only one literal
                                                       on the current level remains
5:         ℓ ← τ(n)
6:         if −ℓ ∈ C' then
7:             C' ← ρ(ℓ) ⊗_ℓ C'
8:         end if
9:         n ← n − 1
10:    end while
11:    ℓ ← the literal in C' with δ(ℓ) = level
12:    Backtrack(δ(C' \ {ℓ}))          ▷ unassigns ℓ but no other literal in C'
13:    F ← F ∪ C'
14:    if |C'| > 1 then
15:        ℓ' ← any literal in C' with δ(ℓ') = level      ▷ backtracking changed level
16:        ω(ℓ) ← ω(ℓ) ∪ C'
17:        ω(ℓ') ← ω(ℓ') ∪ C'                              ▷ set watches
18:    end if
19:    Export(C')
20:    Assign(C', ℓ)                                  ▷ C' is now unit on ℓ
21: end function
```

When importing clauses, new situations can occur. To talk about these we use the definitions from Nadel [8].

Definition 23 (Fake Conflicts)**.** *A conflicting clause C is called* fake*, if only one literal in C is assigned on tevel $\delta(C)$.*

Definition 24 (Unisatisfied Clauses and Missed Implications)**.** *Given a partial assignment σ, a clause C is called* unisatisfied *or unisat, if σ falsifies all the literal of C except one, which is satisfied. If the level of the satisfied literal ℓ is higher then the level of the rest of the clause, i.e., $\delta(\ell) > \delta(C \setminus \{\ell\})$, it is called a* missed implication*.*

Fake conflicts, conflicting clauses where clause level and solver level differ, as well as missed implications could not occur previously, but they do with chronological backtracking or when importing new clauses. For now we ensure that these situations are resolved in the new function Import (Alg. 12), such that changes stay local and the previous algorithms are untouched. In the following section we will discuss the changes to the CDCL algorithms introduced by adding chronological backtracking to the solver. This also allows us to refine the Import function.

3.5 Chronological Backtracking

As introduced in Section 2.1, instead of backjumping to the lowest level where the new clause is still unit, a solver with chronological backtracking will simply backtrack one level after analyzing the conflict clause. This seemingly small change introduces missed implications, fake conflicts, and lower conflict levels further down the line.

Following the notion of Möhle and Biere [7], we uncouple the level of the solver from the level of assignments by redefining the decision level δ.

Algorithm 12 Importing Clauses

1: **function** IMPORT() $\to \{\bot, \emptyset, \top\}$
2: $C \leftarrow$ IMPORTCLAUSE() ▷ maybe get a clause from another thread
3: **if** $C = \bot$ **then** ▷ no new clause
4: **return** $\bot$
5: **end if**
6: **if** C contains satisfied literal ℓ with $\delta(\ell) = 0$ **then** ▷ skip import
7: **return** $\bot$
8: **end if**
9: **if** C is conflicting **then**
10: **if** $\delta(C) = 0$ **then**
11: **return** $\emptyset$
12: **end if**
13: **if** $|\{\ell \in C \mid \delta(\ell) = \delta(C)\}| = 1$ **then**
14: $\ell \leftarrow$ literal in C with $\delta(\ell) = \delta(C)$
15: BACKTRACK($\delta(C \setminus \{\ell\})$)
16: **end if**
17: **end if**
18: $\mathcal{F} \leftarrow \mathcal{F} \cup \{C\}$ ▷ really import the clause
19: **if** $|C| > 1$ **then**
20: $\ell \leftarrow$ literal in C with $\begin{cases} \sigma(\ell) \neq -1, & \text{if possible} \\ \delta(\ell) = \delta(C), & \text{otherwise} \end{cases}$
21: $\ell' \leftarrow$ literal in $C \setminus \{\ell\}$ with $\begin{cases} \sigma(\ell') \neq -1, & \text{if possible} \\ \delta(\ell') = \delta(C \setminus \{\ell\}), & \text{otherwise} \end{cases}$
22: $\omega(\ell) \leftarrow \omega(\ell) \cup C$
23: $\omega(\ell') \leftarrow \omega(\ell') \cup C$ ▷ set watches
24: **if** C is conflicting **then**
25: BACKTRACK($\delta(C)$)
26: $\mathit{propagated} = \tau^{-1}(\ell)$ ▷ handle conflict implicitly in BCP
27: **return** $\top$
28: **end if**
29: **if** C is unisat and $\delta(\ell) > \delta(\ell')$ **then** ▷ missed implication
30: BACKTRACK($\delta(\ell')$)
31: **end if**
32: **end if**
33: **if** $|C| = 1$ and C is satisfied **then**
34: BACKTRACK(0)
35: **end if**
36: **if** C is unit on ℓ **then**
37: ASSIGN(C, ℓ)
38: **return** $\top$
39: **end if**
40: **return** $\bot$
41: **end function**

Definition 25 (Assignment Level)**.** *The value assigned by* $\delta : \mathsf{assigned}_\sigma(\mathcal{V}) \to \mathbb{N}$ *is now called* assignment level*. The assignment level is equal to the highest level of a literal in the responsible unit clause, i.e., the level on which the assignment is implied (Alg. 13, Lines 7-9). That is, for an assignment* ℓ *with reason* C*, the assignment level* $\delta(\ell)$ *is equal* $\delta(C)$*. For decisions the assignment level is equal to the decision level as defined previously (Alg. 13, Line 5).*

Assignments on the trail are no longer ordered by level. Hence we call assignments made at a certain decision level with a lower assignment level *out-of-order* assignments. Möhle and Biere [7] propose the concept of *blocks* and *slices*. Blocks are a sequence of consecutive literals on the trail between two decisions whereas a slice refers to all literals with a certain assignment level. Without chronological backtracking, blocks and slices refer to the same thing. This means that the solver can remove entire blocks of literals from the trail. With chronological backtracking it needs to unassign slices instead, which can stretch over multiple blocks. In order to implement this efficiently we need a helper structure called *stack*.

Definition 26 (Control Stack)**.** *The control stack* $\gamma : \{1, \ldots, level\} \to \{0, \ldots, |\tau|\}$ *keeps track of the starting point on the trail of every decision level. This is done by assigning the position of the decision on the trail to each level (Alg. 13, Line 4). For simplicity we define* $\gamma(level + 1) := |\tau|$*.*

Backtracking (Alg. 14) uses the stack to unassign literals starting at the backtrack level and to re-order out-of-order literals in-place (Lines 2, 3, 5, and 8-13). To ensure that BCP does not miss conflicts or unit clauses, these are repropagated by resetting *propagate* (Line 4).

With these changes we can now apply chronological backtracking to Analyze (Alg. 15, Line 13). Additionally, it is ensured that the conflict level matches the solver level

Algorithm 13 Assigning Literals with Chronological Backtracking, replacing Alg. 3

```
1: function ASSIGN(reason clause C, literal ℓ)
2:     if C = ⊥ then
3:         level ← level + 1
4:         γ(level) ← |τ|                    ▷ add a new decision level to the stack
5:         δ(ℓ) ← level
6:     end if
7:     if C ≠ ⊥ then                         ▷ level of ℓ now depends on C
8:         δ(ℓ) ← δ(C \ {ℓ})
9:     end if
10:    σ(ℓ) ← 1
11:    ρ(ℓ) ← C
12:    n ← |τ|
13:    τ(n) ← ℓ
14: end function
```

Algorithm 14 Backtracking with Chronological Backtracking, replacing Alg. 8

```
1: function BACKTRACK(decision level dl)
2:     n, n' ← γ(dl)
3:     m ← |τ|
4:     propagated ← n                        ▷ moved assignments are repropagated
5:     while n < m do                        ▷ no longer unassign from the top
6:         ℓ ← τ(n)
7:         τ(n) ← ⊥
8:         n ← n + 1
9:         if δ(ℓ) ≤ dl then                 ▷ move ℓ to the lowest empty trail position
10:            τ(n') = ℓ
11:            n' ← n' + 1
12:            continue
13:        end if
14:        σ(ℓ) ← 0                          ▷ unassign ℓ
15:    end while
16:    level ← dl                            ▷ γ changes implicitly
17: end function
```

Algorithm 15 Conflict Analysis with Chronological Backtracking, replacing Alg. 9

```
 1: function ANALYZE(conflict clause C)
 2:     BACKTRACK(δ(C))                          ▷ backtrack to the conflict level
 3:     C′ ← C
 4:     n ← |τ| − 1
 5:     while |{ℓ ∈ C′ | δ(ℓ) = level}| > 1 do   ▷ repeat until only one literal
                                                    on the current level remains
 6:         ℓ ← τ(n)
 7:         if δ(ℓ) = level and −ℓ ∈ C′ then    ▷ skip out-of-order assignments
 8:             C′ ← ρ(ℓ) ⊗_ℓ C′
 9:         end if
10:         n ← n − 1
11:     end while
12:     ℓ ← the literal in C′ with δ(ℓ) = level
13:     BACKTRACK(δ(ℓ) − 1)                      ▷ chronological backtracking
14:     if C′ ≠ C then
15:         F ← F ∪ C′                           ▷ C was a true conflict
16:         if |C′| > 1 then
17:             ℓ′ ← any literal in C′ with δ(ℓ′) = δ(C \ ℓ)
18:             ω(ℓ) ← ω(ℓ) ∪ C′
19:             ω(ℓ′) ← ω(ℓ′) ∪ C′               ▷ set watches
20:         end if
21:     end if
22:     ASSIGN(C′, ℓ)                            ▷ C′ is now unit on ℓ
23: end function
```

(Line 2), out-of-order literals are skipped in the resultion step (Line 7), and fake conflicts do not result in a learned clause (Line 14).

This version of the solver can handle assignments on arbitrary levels, which we can use for the IMPORT function in a parallel setting (Alg. 16, Lines 14, 23, 28, and 32).

Algorithm 16 Importing Clauses with Chronological Backtracking, replacing Alg. 12

1: **function** IMPORT() $\to \{\bot, \emptyset, \top\}$
2: $C \leftarrow$ IMPORTCLAUSE() ▷ maybe get a clause from another thread
3: **if** $C = \bot$ **then** ▷ no new clause
4: **return** $\bot$
5: **end if**
6: **if** C contains satisfied literal ℓ with $\delta(\ell) = 0$ **then** ▷ skip import
7: **return** $\bot$
8: **end if**
9: **if** C is conflicting **then**
10: **if** $\delta(C) = 0$ **then**
11: **return** $\emptyset$
12: **end if**
13: **if** $|\{\ell \in C \mid \delta(\ell) = \delta(C)\}| = 1$ **then**
14: BACKTRACK($\delta(C) - 1$) ▷ chronological backtracking
15: **end if**
16: **end if**
17: $\mathcal{F} \leftarrow \mathcal{F} \cup \{C\}$ ▷ really import the clause
18: **if** $|C| > 1$ **then** ▷ find watches
19: $\ell \leftarrow$ literal in C with $\begin{cases} \sigma(\ell) \neq -1, & \text{if possible} \\ \delta(\ell) = \delta(C), & \text{otherwise} \end{cases}$
20: $\ell' \leftarrow$ literal in $C \setminus \{\ell\}$ with $\begin{cases} \sigma(\ell') \neq -1, & \text{if possible} \\ \delta(\ell') = \delta(C \setminus \{\ell\}), & \text{otherwise} \end{cases}$
21: $\omega(\ell) \leftarrow \omega(\ell) \cup C$ ▷ set watches
22: $\omega(\ell') \leftarrow \omega(\ell') \cup C$
23: **if** C is conflicting **then** ▷ no backtrack
24: $\mathit{propagated} = \tau^{-1}(\ell)$ ▷ handle conflict implicitly in BCP
25: **return** $\top$
26: **end if**
27: **if** C is unisat and $\delta(\ell) > \delta(\ell')$ **then** ▷ missed implication
28: BACKTRACK($\delta(C) - 1$) ▷ chronological backtracking
29: **end if**
30: **end if**
31: **if** $|C| = 1$ and C is satisfied **then**
32: BACKTRACK($\delta(C) - 1$) ▷ chronological backtracking
33: **end if**
34: **if** C is unit on ℓ **then**
35: ASSIGN(C, ℓ)
36: **return** $\top$
37: **end if**
38: **return** $\bot$
39: **end function**

4 Approach

In the following we describe and contrast three different implementations of reimplication, first the original implementation in IntelSAT [8], then our two new implementations, one in the SAT solver CaDiCaL [24, 23], the other in the parallel SAT solver Gimsatul [22].

4.1 Reimplication

In the previous chapter we have described chronological backtracking, which produces missed implications and fake conflicts. Additionally, the formula can contain multiple conflicts on different levels at the same time. Soundness and completeness of the algorithm is still guaranteed, however at the cost of some unintended side effects.

When backtracking, out-of-order assignments have to be repropagated even though chronological backtracking is supposed to save work. This is how missed implications are found after they have turned into unit clauses through backtracking without changing the watch scheme for BCP. Reimplication adresses this problem by fixing missed implications eagerly, thus avoiding repropagation after backtracking.

In chronological backtracking as well as with reimplication, fake conflicts trigger backtracking instead of conflict analysis. If multiple conflicts exist, chronological backtracking only guarantees that they are found eventually, while reimplication seeks

to find the conflict with the lowest level directly. This avoids potentially unnecessary iterations of BCP and conflict analysis.

To guarantee that BCP can find all missed implications the watch scheme has to be modified. Previously, when a watched literal was satisfied, the solver could ignore the clause. Now, unisatisfied clauses can not be ignored if they are a missed implication. The new scheme guarantees that skipped clauses are no missed implications by checking the level of the two watched literals if the clause is watched by a satisified and a falsified literal. If the falsified literal is assigned at the same or higher level then the satisfied literal, the clause can not be a missed implication and can savely be ignored. By updating the watched literals of conflicts and unit clauses it is ensured that a clause is never watched by a falsified and an unassigned literal after backtracking which could then be missed by BCP.

The skipping criterium is applied to BCP in all three implementations (Alg. 21, Alg. 32 and Alg. 41, Line 5). When watching new clauses or changing watched literals it is enough to always prioritize unassigned literals, then satisified literals and finally the falsified literal with the highest level.

Before continuing with the implementation in IntelSAT from the original paper [8], we introduce the following definition for fixing missed implications as it reflects our understanding of reimplication.

Definition 27 (Elevating Literals)**.** *A missed implication is fixed by changing the level and reason of the unisatisifed literal to the missed implication clause and corresponding level. We call this process* elevation*.*

Intuitively, literals can be elevated until they are assigned at the lowest possible level. This is realisation guides our two implementations of reimplication.

4.2 Implementation in IntelSAT

For conflict analysis to work, all assignments with the same level need to be ordered on the trail chronologically, i.e., by the time they were assigned. Changing the order of literals which are assigned on different levels however is possible. Expressed in terms of blocks and slices we have to ensure that the relative order of every slice is preserved. IntelSAT uses this fact to avoid dealing with out-of-order assignments in Backtrack (Alg. 17) and Analyze (Alg. 18). Apart from chronologically backtracking (Alg. 18, Line 10), these are the same as without chronological backtracking.

When assigning literals they are inserted into the trail after the last assignment on their respective level, keeping all literals with the same assignment level in blocks (Alg. 19, Lines 12-15). This is implemented efficiently using a doubly linked list and keeping track of the beginning and end of each block with the control stack γ.

This re-ordered trail can not be used for BCP anymore because there is no way to differentiate between already processed literals and newly assigned ones. Instead, literals to be propagated are pushed on the *propagation queue*.

Definition 28 (Propagation Queue)**.** $\pi \subseteq \mathsf{satsified}_\sigma(\mathcal{L})$ *is called propagation queue. It stores all literals that have to be processed by* BCP*. Whenever a literal* ℓ *is assigned to true the solver inserts* ℓ *into* π *(Alg. 19, Line 16).* BCP *removes one literal of* π *at a time and processes it (Alg. 20, Lines 3 and 4).*

The simplest implementation of the propagation queue is a stack which can only be changed by adding a literal to the top or removing the top most literal. Interestingly, this flips the order in which literals are processed by BCP when compared to using the trail, from first assigned literal first to last assigned literal first.

All conflict clauses and missed implications are saved until they are processed.

Algorithm 17 Backtracking in IntelSAT, using Alg. 4

function Backtrack(decision level dl)
  $n \leftarrow |\tau| - 1$
  while $n > 0$ and $\delta(\tau(n)) > dl$ **do**
    $\ell \leftarrow \tau(n)$ ▷ unassign top most literal on trail
    $\tau(n) \leftarrow \bot$
    $n \leftarrow n - 1$
    $\sigma(\ell) \leftarrow 0$ ▷ no need to change ρ and δ, since ℓ is unassigned
  end while
  $level \leftarrow dl$ ▷ update decision level
end function

Algorithm 18 Conflict analysis in IntelSAT, replacing Alg. 9

function Analyze(conflict clause C)
  $C' \leftarrow C$
  $n \leftarrow |\tau| - 1$
  while $|\{\ell \in C' \mid \delta(\ell) = level\}| > 1$ **do** ▷ repeat until only one literal on the current level remains
    $\ell \leftarrow \tau(n)$
    $C' \leftarrow \rho(\ell) \otimes_\ell C'$
    $n \leftarrow n - 1$
  end while
  $\ell \leftarrow$ the literal in C' with $\delta(\ell) = level$
  Backtrack($\delta(\ell) - 1$) ▷ chronological backtracking
  $\mathcal{F} \leftarrow \mathcal{F} \cup C'$
  if $|C'| > 1$ **then**
    $\ell' \leftarrow$ any literal in C' with $\delta(\ell') = \delta(C \setminus \ell)$ ▷ ℓ' on highest possible level
    $\omega(\ell) \leftarrow \omega(\ell) \cup C'$
    $\omega(\ell') \leftarrow \omega(\ell') \cup C'$ ▷ set watches
  end if
  Assign(C', ℓ) ▷ C' is now unit on ℓ
end function

Definition 29 (Reimplication Stack)**.** *$\mu \subseteq \mathcal{F}$ is called reimplication stack. It stores all clauses that have to be processed by reimplication.*

Definition 30 (Conflict Stack)**.** *$\kappa \subseteq \mathcal{F}$ is called conflict stack. It stores all conflicts that are found in BCP.*

The functionality of BCP (Alg. 20) is split into three algorithms.

1. PropagateWatch (Alg. 21) finds unit clauses, missed implications and conflicts in the watch list of a falsified literal.

2. Reimply (Alg. 22) handles missed implications iteratively by elevating literals (Alg. 23) and fixing the watch lists of elevated literals.

3. ProcessConflicts (Alg. 24) ensures that BCP terminates with the lowest possible conflict and fixes the watched literals of the conflict clauses.

Notice that BCP can not terminate early when finding a conflict, meaning the while loop (Alg. 20, Line 3) continues until the propagation queue π is empty. Additionally, literals are repropagated every time a missed implication or conflict is found (Alg. 21, Lines 23 and 36). The CDCL loop Solve stays the same as in Section 3.2 (Alg. 6).

4.3 Implementation in CaDiCaL

IntelSAT needs to perform BCP until a fixed point is reached, even if a conflict is found. Following our intuition of elevating literals, the conflict level can not change if all literals on lower levels have been processed. Therefore, if BCP would prioritize the lowest level literal first, it could stop as soon as a conflict on the level of the

Algorithm 19 Assigning literals in IntelSAT, replacing Alg. 13

function Assign(reason clause C, literal ℓ)
  if $C = \bot$ **then**
    $level \leftarrow level + 1$
    $\gamma(level) \leftarrow |\tau|$ ▷ add a new decision level to the stack
    $\delta(\ell) \leftarrow level$
  end if
  if $C \neq \bot$ **then**
    $\delta(\ell) \leftarrow \delta(C \setminus \{\ell\})$
  end if
  $\sigma(\ell) \leftarrow 1$
  $\rho(\ell) \leftarrow C$
  $n \leftarrow \gamma(\delta(\ell) + 1)$ ▷ insert ℓ into τ such that it is sorted by level
  for all $m \geq n$ do $\tau(m+1) \leftarrow \tau(m)$
  for all decisions ℓ_d do $\gamma(\delta(\ell_d)) \leftarrow \tau^{-1}(\ell_d)$ ▷ fix γ
  $\tau(n) \leftarrow \ell$ ▷ efficient implementation with doubly linked list
  $\pi(|pi|) \leftarrow \ell$ ▷ put ℓ on the propagation queue
end function

Algorithm 20 Boolean constraint propagation in IntelSAT, replacing Alg. 7

function BCP() $\rightarrow \mathcal{F} \cup \{\bot\}$
  Reimply()
  while there is a literal $l_{prop} \in \pi$ **do**
    $\pi \leftarrow \pi \setminus \{\ell_{prop}\}$
    PropagateWatch(ℓ_{prop})
  end while
  if $\kappa \neq \emptyset$ **then**
    return any $C \in \kappa$
  end if
  return $\bot$
end function

Algorithm 21 Inner loop of Boolean constraint propagation with reimplication in IntelSAT, compare with Alg. 32 and Alg. 41

```
1: function PropagateWatch(ℓ_prop)
2:     for C ∈ ω(−l_prop) do
3:         ⟨ℓ1, ℓ2⟩ ← ω^{-1}(C)                          ▷ get the two watched literals for C
4:         ℓ_other ← { ℓ2,  if −ℓ_prop = ℓ1
                     { ℓ1,  otherwise
5:         if σ(ℓ_other) = 1 and δ(ℓ_other) ≤ δ(ℓ_prop) then          ▷ watch invariant
6:             continue
7:         end if
8:         if C is unit on ℓ_other then
9:             Assign(C, ℓ_other)
10:            ℓ ← literal in C \ {ℓ_other} with δ(ℓ) = δ(C)
11:            ω(−ℓ_prop) ← ω(−ℓ_prop) \ {C}
12:            ω(ℓ) ← ω(ℓ) ∪ {C}                          ▷ swap watched literal
13:            continue
14:        end if
15:        if C is unisatisfied on ℓ_other then
16:            ℓ ← literal in C \ {ℓ_other} with δ(ℓ) = δ(C \ {ℓ_other})
17:            ω(−ℓ_prop) ← ω(−ℓ_prop) \ {C}
18:            ω(ℓ) ← ω(ℓ) ∪ {C}                          ▷ swap watched literal
19:            if δ(C \ {ℓ_other}) < δ(ℓ_other) then       ▷ missed implication
20:                μ ← μ ∪ C
21:                π ← π ∪ ℓ_prop
22:                Reimply()
23:                ProcessConflicts()
24:                return
25:            end if
26:            continue
27:        end if
28:        if C is not falsified then
29:            ℓ ← literal in C \ {ℓ_other} with σ(ℓ) ≠ −1
30:            ω(−ℓ_prop) ← ω(−ℓ_prop) \ {C}
31:            ω(ℓ) ← ω(ℓ) ∪ {C}                          ▷ swap watched literal
32:            continue
33:        end if
34:        κ ← κ ∪ C                                      ▷ conflict
35:        if δ(C) < level or |{ℓ ∈ C | δ(ℓ) = level}| = 1 then
36:            π ← π ∪ ℓ_prop
37:            ProcessConflicts()
38:            return
39:        end if
40:    end for
41: end function
```

Algorithm 22 Reimplication in IntelSAT

function Reimply()
  while there exists $C \in \mu$ **do**
    $\mu \leftarrow \mu \setminus \{C\}$
    $\ell \leftarrow$ the satisfied literal in C ▷ all the clauses in μ are unisatisfied
    if $\delta(C \setminus \{\ell\}) \geq \delta(\ell)$ **then**
      continue ▷ no missed implication
    end if
    Reassign(C, ℓ) ▷ elevate ℓ
    if $\ell \in \pi$ **then**
      continue ▷ ℓ will be propagated and fixed that way
    end if
    for $C' \in \omega(-\ell)$ **do** ▷ fix watch list
      if $C' \in \mu$ or $C' \in \kappa$ **then**
        continue ▷ C' will be fixed anyway
      end if
      $\langle \ell_1, \ell_2 \rangle \leftarrow \omega^{-1}(C')$ ▷ get the two watched literals for C'
      $\ell' \leftarrow \begin{cases} \ell_2, & \text{if } -\ell = \ell_1 \\ \ell_1, & \textit{otherwise} \end{cases}$
      if $\sigma(\ell') = 1$ and $\delta(\ell') \leq \delta(\ell)$ **then** ▷ watches already correct
        continue
      end if ▷ C is either satisfied or has multiple unassigned literals
      $\ell'_1 \leftarrow \ell'_1 \in C'$ with $\sigma(\ell'_1) \neq -1$
      $\ell'_2 \leftarrow \ell'_2 \in C' \setminus \{\ell'_1\}$ with $\begin{cases} \sigma(\ell'_2) \neq -1, & \text{if possible} \\ \delta(\ell'_2) = \delta(C' \setminus \{\ell'_1\}), & \text{otherwise} \end{cases}$
      $\omega(-\ell) \leftarrow \omega(-\ell) \setminus \{C'\}$
      $\omega(\ell') \leftarrow \omega(\ell') \setminus \{C'\}$
      $\omega(\ell'_1) \leftarrow \omega(\ell'_1) \cup \{C'\}$
      $\omega(\ell'_2) \leftarrow \omega(\ell'_2) \cup \{C'\}$
      if C' is unisat and $\delta(\ell'_2) < \delta(\ell'_1)$ **then** ▷ missed implication
        $\mu \leftarrow \mu \cup \{C'\}$
      end if
    end for
  end while
end function

Algorithm 23 Elevating literals in IntelSAT, compare with Alg. 19

function Reassign(reason clause C, literal ℓ)
  $\delta(\ell) \leftarrow \delta(C \setminus \{\ell\})$ ▷ C cannot be $\bot$
  $\rho(\ell) \leftarrow C$ ▷ value of ℓ already set
  $n' \leftarrow \tau^{-1}(\ell)$ ▷ ℓ already on the trail
  $n \leftarrow \gamma(\delta(\ell) + 1)$ ▷ new position is always smaller
  for all $n \leq m < n'$ do $\tau(m+1) \leftarrow \tau(m)$
  for all decisions ℓ_d do $\gamma(\delta(\ell_d)) \leftarrow \tau^{-1}(\ell_d)$ ▷ fix γ
  $\tau(n) \leftarrow \ell$ ▷ move ℓ to new position
end function

Algorithm 24 Conflict handling in IntelSAT

function ProcessConflicts()
  for $C \in \kappa$ **do**
    $\ell_1, \ell_2 \leftarrow \omega^{-1}(C)$ ▷ replace watches
    $\ell_1' \leftarrow$ literal in C with $\delta(\ell_1') = \delta(C)$
    $\ell_2' \leftarrow$ literal in $C \setminus \{\ell_1'\}$ with $\delta(\ell_2') = \delta(C \setminus \{\ell_1'\})$
    $\omega(\ell_1) \leftarrow \omega(\ell_1) \setminus \{C\}$
    $\omega(\ell_2) \leftarrow \omega(\ell_2) \setminus \{C\}$
    $\omega(\ell_1') \leftarrow \omega(\ell_1') \cup \{C\}$
    $\omega(\ell_2') \leftarrow \omega(\ell_2') \cup \{C\}$
  end for
  $n \leftarrow \min\{\delta(C) \mid C \in \kappa\}$ ▷ get the lowest conflict level
  Backtrack(n)
  if there exists $C \in \kappa$ conflicting with $|\{\ell \in C \mid \delta(\ell) = n\}| = 1$ **then**
    Backtrack($level - 1$)
  end if
  for $C \in \kappa$ **do**
    if C is unit on ℓ **then**
      Assign(C, ℓ)
    end if
    if C is not falsified **then**
      $\kappa \leftarrow \kappa \setminus \{C\}$
    end if
  end for
end function

propagated literal is reached. This also guarantees that all conflicts with lower level have been found already.

We use this observation to reduce unnecessary propagations in our implementation in CaDiCaL. We further adapt the trail structure to allow both sorting the literals by level as in IntelSAT and also using it for BCP.

Definition 31 (Multitrail)**.** *There is a trail τ_i for each decision level $0 \leq i \leq level$. We call the combination of these trails the* multitrail. *Each trail can have* holes*, i.e., $\tau_i(n) = \bot$ but $\tau_i(m) \neq \bot$ for some $m > n$. These holes are created by elevating literals (Alg. 25, Lines 4 and 5) and stay until the solver backtracks to an earlier level. To deal with holes, the definition of the size of a single trail is adapted slightly: $|\tau_i| := \max_n\{\tau_i^{-1} \neq \bot\} + 1$. Each trail τ_i also gets a separate value $propagated_i$.*

Backtrack (Alg. 26) is adapted to the multitrail, unassigning trails level by level (Line 2), ignoring holes (Lines 6-8) and resetting the *propagated* value (Line 13). Holes are also ignored during conflict analysis (Alg. 28, Line 7). The size of the trail no longer directly measures the number of assignments. They are counted explicitly with a new value *assigned* instead (Alg. 27, Line 13 and Alg. 26, Line 12). This replaces the termination check in the CDCL loop Solve (Alg. 30, Line 17).

BCP iterates over each level and processes the new literals on that level. For efficiency reasons, we split the procedure into BCP with reimplication which handles all levels below the solver level (Alg. 31) and BCP without reimplication for the highest level (Alg. 29). When a conflict is found early, the latter can be skipped entirely (Alg. 30, Lines 6-9). As the name suggests, no missed implications can occur when propagating the highest level. This means we can use the already existing, highly optimized propagation routine of CaDiCaL for BCP without reimplication and simply ignore holes (Alg. 29, Lines 5-7).

Algorithm 25 Elevating literals in CaDiCaL, compare with Alg. 27

```
1: function Reassign(reason clause C, literal ℓ)
2:     δ(ℓ) ← δ(C \ {ℓ})                          ▷ C cannot be ⊥
3:     ρ(ℓ) ← C                                   ▷ value of ℓ already set
4:     n ← τ_{δ(ℓ)}^{-1}(l)                       ▷ creates a hole
5:     τ_{δ(ℓ)}(n) ← ⊥
6:     n ← |τ_{δ(ℓ)}|
7:     τ_{δ(ℓ)}(n) ← ℓ
8: end function
```

Algorithm 26 Backtracking in CaDiCaL, replacing Alg. 14

```
1: function Backtrack(decision level dl)
2:     while level > dl do
3:         n ← |τ_level| − 1
4:         while n > 0 do
5:             ℓ ← τ_level(n)                   ▷ unassign top most literal on trail
6:             if ℓ = ⊥ then
7:                 continue                     ▷ hole due to an elevated literal
8:             end if
9:             τ_level(n) ← ⊥
10:            n ← n − 1
11:            σ(ℓ) ← 0          ▷ no need to change ρ and δ, since ℓ is unassigned
12:            assigned ← assigned − 1
13:        end while
14:        propagated_level ← 0
15:        level ← level − 1
16:    end while
17: end function
```

Algorithm 27 Assigning literals in CaDiCaL, replacing Alg. 13

function ASSIGN(reason clause C, literal ℓ)
  if $C = \bot$ **then**
    $level \leftarrow level + 1$ ▷ no stack
    $\delta(\ell) \leftarrow level$
  end if
  if $C \neq \bot$ **then**
    $\delta(\ell) \leftarrow \delta(C \setminus \{\ell\})$
  end if
  $\sigma(\ell) \leftarrow 1$
  $\rho(\ell) \leftarrow C$
  $n \leftarrow |\tau_{\delta(\ell)}|$
  $\tau_{\delta(\ell)}(n) \leftarrow \ell$
  $assigned \leftarrow assigned + 1$
end function

Algorithm 28 Conflict analysis in CaDiCaL, replacing Alg. 15

function ANALYZE(conflict clause C)
  BACKTRACK($\delta(C)$) ▷ backtrack to the conflict level
  $C' \leftarrow C$
  $n \leftarrow |\tau_{level}| - 1$
  while $|\{\ell \in C' \mid \delta(\ell) = level\}| > 1$ **do** ▷ repeat until only one literal on the current level remains
    $\ell \leftarrow \tau_{level}(n)$
    if $\delta(\ell) \neq \bot$ and $-\ell \in C'$ **then** ▷ skip holes
      $C' \leftarrow \rho(\ell) \otimes_\ell C'$
    end if
    $n \leftarrow n - 1$
  end while
  $\ell \leftarrow$ the literal in C' with $\delta(\ell) = level$
  BACKTRACK($\delta(\ell) - 1$) ▷ chronological backtracking
  if $C' \neq C$ **then**
    $\mathcal{F} \leftarrow \mathcal{F} \cup C'$ ▷ C was a true conflict
    if $|C'| > 1$ **then**
      $\ell' \leftarrow$ any literal in C' with $\delta(\ell') = \delta(C \setminus \ell)$
      $\omega(\ell) \leftarrow \omega(\ell) \cup C'$
      $\omega(\ell') \leftarrow \omega(\ell') \cup C'$ ▷ set watches
    end if
  end if
  ASSIGN(C', ℓ) ▷ C' is now unit on ℓ
end function

Algorithm 29 Boolean constraint propagation without reimplication in CaDiCaL, compare with Alg. 7

1: **function** BCP() $\to \mathcal{F} \cup \{\bot\}$
2: **while** $\tau_{level}(propagated_{level}) \neq \bot$ **do**
3: $\ell_{prop} \leftarrow \tau_{level}(propagated_{level})$
4: $propagated_{level} \leftarrow propagated_{level} + 1$
5: **if** $\ell_{prop} = \bot$ **then**
6: **continue** ▷ hole
7: **end if**
8: **for** $C \in \omega(-\ell_{prop})$ **do**
9: $\langle \ell_1, \ell_2 \rangle \leftarrow \omega^{-1}(C)$ ▷ get the two watched literals for C
10: $\ell_{other} \leftarrow \begin{cases} \ell_2, & \text{if } -\ell_{prop} = \ell_1 \\ \ell_1, & otherwise \end{cases}$
11: **if** $\sigma(\ell_{other}) = 1$ **then** ▷ no update required
12: **continue**
13: **end if**
14: **if** C is unit on ℓ_{other} **then**
15: Assign(C, ℓ_{other})
16: **continue**
17: **end if**
18: **if** C is not falsified **then**
19: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\sigma(\ell) \neq -1$
20: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
21: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
22: **continue**
23: **end if**
24: **return** C ▷ conflict
25: **end for**
26: **end while**
27: **return** $\bot$
28: **end function**

Algorithm 30 CDCL loop in CaDiCaL, replacing Alg. 6

```
1:  function Solve(CNF F) → {UNSAT, (SAT, M ⊨ F)}
2:      if ∅ ∈ F then                                  ▷ formula trivially unsatisfiable
3:          return UNSAT
4:      end if
5:      while true do
6:          C ← ReimplyBCP()                  ▷ BCP with Reimplication on lower levels
7:          if C = ⊥ then
8:              C ← BCP()                              ▷ normal BCP on current level
9:          end if
10:         if C ≠ ⊥ then
11:             if δ(C) = 0 then                                    ▷ global conflict
12:                 return UNSAT
13:             end if
14:             Analyze(C)
15:             continue
16:         end if
17:         if assigned = |V| then                                  ▷ σ is a model
18:             return (SAT, σ)
19:         end if
20:         Decide()                                                ▷ make a decision
21:     end while
22: end function
```

Missed implications are not handled by a reimplication routine but are elevated immediately (Alg. 32, Line 20). Similarly to IntelSAT, conflicts are pushed on the conflict stack (Alg. 32, Line 30) and the watches are fixed by a ProcessConflict procedure (Alg. 33). However, fixing the watches is not done upon finding a conflict but on the next invocation of BCP (Alg. 31, Line 2). When a conflict on the level of the currently processed literal is found or a the level of the processed literal reaches the level of a previously found conflict, BCP terminates (Alg. 32, Lines 31-33 and Alg. 29, Lines 5-7 and 18-20).

4.4 Implementation in Gimsatul

The trail is an essential part of the solver. It is used in many places that we do not describe in this thesis, like pre- and inprocessing. In order to reduce the implementational overhead, we wanted to keep the trail structure as it was, i.e., as in chronological backtracking. However, we still want to propagate level by level to avoid unnecessary propagations. The solution is using a priority queue, which adapts the propagation queue of IntelSAT (Def. 28), and always returns the literal with the lowest level first. The trail position is used as a tie breaker, however as mentioned before, this is an arbitrary choice and just mimics the behaviour of using the trail for propagation (Alg. 40, Lines 2-5).

The other change from CaDiCaL and IntelSAT is that we no longer keep track of all conflicts, and instead focus on the conflict with the lowest level. This allows to replace the conflict stack κ (Def. 30) with a single value $conflict \in \mathcal{F} \cup \{\bot\}$ and removes the necessity for a ProcessConflicts routine. However we still need to fix the watches of the conflicts that would have been on the conflict stack. We introduce a second propagation queue π' which is not used for propagation, but instead saves the watched literals of all found conflicts (Alg. 41, Lines 30-36). After conflict analysis

Algorithm 31 Boolean constraint propagation with reimplication in CaDiCaL, compare with Alg. 7

```
 1: function ReimplyBCP() → F ∪ {⊥}
 2:     ProcessConflicts()
 3:     i ← 0
 4:     while i < level do
 5:         if there is C ∈ κ with δ(C) ≤ i then
 6:             return C
 7:         end if
 8:         while τ_i(propagated_i) ≠ ⊥ do
 9:             ℓ_prop ← τ_i(propagated_i)
10:             propagated_i ← propagated_i + 1
11:             if ℓ_prop = ⊥ then
12:                 continue                                  ▷ hole
13:             end if
14:             PropagateWatch(ℓ_prop)
15:         end while
16:         i ← i + 1
17:     end while
18:     if κ ≠ ∅ then
19:         return any C ∈ κ
20:     end if
21:     return ⊥
22: end function
```

Algorithm 32 Inner loop of Boolean constraint propagation with reimplication in CaDiCaL, compare with Alg. 21 and Alg. 41

1: **function** PropagateWatch(ℓ_{prop})
2: **for** $C \in \omega(-\ell_{prop})$ **do**
3: $\langle \ell_1, \ell_2 \rangle \leftarrow \omega^{-1}(C)$ ▷ get the two watched literals for C
4: $\ell_{other} \leftarrow \begin{cases} \ell_2, & \text{if } -\ell_{prop} = \ell_1 \\ \ell_1, & otherwise \end{cases}$
5: **if** $\sigma(\ell_{other}) = 1$ and $\delta(\ell_{other}) \leq \delta(\ell_{prop})$ **then** ▷ watch invariant
6: **continue**
7: **end if**
8: **if** C is unit on ℓ_{other} **then**
9: Assign(C, ℓ_{other})
10: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\delta(\ell) = \delta(C)$
11: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
12: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
13: **continue**
14: **end if**
15: **if** C is unisatisfied on literal ℓ_{other} **then**
16: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\delta(\ell) = \delta(C \setminus \{\ell_{other}\})$
17: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
18: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
19: **if** $\delta(C \setminus \{\ell_{other}\}) < \delta(\ell_{other})$ **then** ▷ missed implication
20: Reassign(C, ℓ)
21: **end if**
22: **continue**
23: **end if**
24: **if** C is not falsified **then**
25: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\sigma(\ell) \neq -1$
26: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
27: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
28: **continue**
29: **end if**
30: $\kappa \leftarrow \kappa \cup \{C\}$ ▷ conflict
31: **if** $\delta(C) = \delta(\ell_{prop})$ **then**
32: **return** ▷ conflict on current propagated level
33: **end if**
34: **end for**
35: **end function**

and backtracking, the literals in π' which are still assigned are pushed back onto π, fixing the watches implicitly through repropagation. (Alg. 35, Lines 23-29).

Elevating literals still creates holes (Alg. 34, Lines 4 and 5), which are skipped in conflict analysis (Alg. 37, Line 7) and backtracking (Alg. 35, Lines 11-13). We also keep the value *assigned* as termination check for the CDCL loop (Alg. 36, Line 17, Alg. 35, Line 20) and Alg. 39, Line 18).

Like in CaDiCaL, Boolean constraint propagation terminates early when the propagated level reaches the conflict level (Alg. 40, Lines 6-8). It is also split into BCP with reimplication (Alg. 40) which uses the priority queue and BCP without reimplication (Alg. 38) which uses the trail and the *propagated* value. Because of the priority queue and trail structure the solver can no longer switch between these two upon reaching the top most level, and instead only uses the latter when there is no new assignment on a lower level (Alg. 39). Because of this, there can be no holes in BCP without reimplication, so we can reuse the already existing routine of Gimsatul without modifications. When assigning literals the solver has to differentiate between using the propagation queue or not based on which propagation routine it is currently running (Alg. 36, Lines 14-16). Since literals can only be elevated through BCP with reimplication we can save this additional check (Alg. 34, Line 8).

Last but not least, reimplication can be applied when importing a clause (Alg. 42).

Algorithm 33 Conflict handling in CaDiCaL

function ProcessConflicts()
  for $C \in \kappa$ **do**
    if C is unisatisfied on ℓ and $\delta(C \setminus \{\ell\}) < \delta(\ell)$ **then**
      Reassign(C, ℓ)
    end if
    if C is unit on ℓ **then**
      Assign(C, ℓ)
    end if
    $\ell_1, \ell_2 \leftarrow \omega^{-1}(C)$ ▷ replace watches
    $\ell_1' \leftarrow$ literal in C with $\sigma(\ell_1') \neq -1$
    $\ell_2' \leftarrow$ literal in $C \setminus \{\ell_1'\}$ with $\begin{cases} \sigma(\ell_2') \neq -1, & \text{if possible} \\ \delta(\ell_2') = \delta(C \setminus \{\ell_1'\}), & \text{otherwise} \end{cases}$
    $\omega(\ell_1) \leftarrow \omega(\ell_1) \setminus \{C\}$
    $\omega(\ell_2) \leftarrow \omega(\ell_2) \setminus \{C\}$
    $\omega(\ell_1') \leftarrow \omega(\ell_1') \cup \{C\}$
    $\omega(\ell_2') \leftarrow \omega(\ell_2') \cup \{C\}$
  end for
  $\kappa \leftarrow \emptyset$
end function

Algorithm 34 Elevating literals in Gimsatul, compare with Alg. 36

function Reassign(reason clause C, literal ℓ)
  $\delta(\ell) \leftarrow \delta(C \setminus \{\ell\})$ ▷ C cannot be $\perp$
  $\rho(\ell) \leftarrow C$ ▷ value of ℓ already set
  $n \leftarrow \tau^{-1}(l)$ ▷ creates a hole
  $\tau(n) \leftarrow \perp$
  $n \leftarrow |\tau|$
  $\tau(n) \leftarrow \ell$
  $\pi \leftarrow \pi \cup \{\ell\}$ ▷ always use π
end function

Algorithm 35 Backtracking in GIMSATUL, replacing Alg. 14

```
 1: function BACKTRACK(decision level dl)
 2:     if dl ≥ level then                                  ▷ nothing to do
 3:         return
 4:     end if
 5:     n, n' ← γ(dl)
 6:     m ← |τ|
 7:     while n < m do                          ▷ no longer unassign from the top
 8:         ℓ ← τ(n)
 9:         τ(n) ← ⊥
10:         n ← n + 1
11:         if ℓ = ⊥ then
12:             continue                        ▷ hole due to an elevated literal
13:         end if
14:         if δ(ℓ) ≤ dl then
15:             τ(n') = ℓ
16:             n' ← n' + 1
17:             continue
18:         end if
19:         σ(ℓ) ← 0                                          ▷ unassign ℓ
20:         assigned ← assigned − 1
21:     end while
22:     propagated ← |τ|          ▷ moved assignments are not repropagated
23:     level ← dl                                  ▷ γ changes implicitly
24:     for ℓ ∈ π' do
25:         if σ(ℓ) = −1 then
26:             π ← π ∪ {ℓ}                  ▷ repropagate literals in π'
27:         end if
28:     end for
29:     π' ← ∅
30:     conflict ← ⊥                                    ▷ reset conflict
31: end function
```

Algorithm 36 Assigning literals in GIMSATUL, replacing Alg. 13

```
1: function ASSIGN(reason clause C, literal ℓ)
2:     if C = ⊥ then
3:         level ← level + 1
4:         γ(level) ← |τ|                    ▷ add a new decision level to the stack
5:         δ(ℓ) ← level
6:     end if
7:     if C ≠ ⊥ then
8:         δ(ℓ) ← δ(C \ {ℓ})
9:     end if
10:    σ(ℓ) ← 1
11:    ρ(ℓ) ← C
12:    n ← |τ|
13:    τ(n) ← ℓ
14:    if π ≠ ∅ or δ(ℓ) ≠ level then
15:        π ← π ∪ {ℓ}
16:    end if
17:    assigned ← assigned + 1
18: end function
```

Algorithm 37 Conflict analysis in GIMSATUL, replacing Alg. 15

```
1: function ANALYZE(conflict clause C)
2:     BACKTRACK(δ(C))                          ▷ backtrack to the conflict level
3:     C′ ← C
4:     n ← |τ| − 1
5:     while |{ℓ ∈ C′ | δ(ℓ) = level}| > 1 do   ▷ repeat until only one literal
                                                   on the current level remains
6:         ℓ ← τ(n)
7:         if δ(ℓ) ≠ ⊥ and −ℓ ∈ C′ then          ▷ skip holes
8:             C′ ← ρ(ℓ) ⊗_ℓ C′
9:         end if
10:        n ← n − 1
11:    end while
12:    ℓ ← the literal in C′ with δ(ℓ) = level
13:    BACKTRACK(δ(ℓ) − 1)                      ▷ chronological backtracking
14:    if C′ ≠ C then
15:        F ← F ∪ C′                           ▷ C was a true conflict
16:        if |C′| > 1 then
17:            ℓ′ ← any literal in C′ with δ(ℓ′) = δ(C \ ℓ)
18:            ω(ℓ) ← ω(ℓ) ∪ C′
19:            ω(ℓ′) ← ω(ℓ′) ∪ C′               ▷ set watches
20:        end if
21:    end if
22:    ASSIGN(C′, ℓ)                            ▷ C′ is now unit on ℓ
23: end function
```

Algorithm 38 Boolean constraint propagation without reimplication in GIMSATUL, compare with Alg. 7

```
1: function BCP() → F ∪ {⊥}
2:     while τ(propagated) ≠ ⊥ do
3:         ℓ_prop ← τ(propagated)
4:         propagated ← propagated + 1
5:         for C ∈ ω(−ℓ_prop) do
6:             ⟨ℓ1, ℓ2⟩ ← ω^{-1}(C)                ▷ get the two watched literals for C
7:             ℓ_other ← { ℓ2, if −ℓ_prop = ℓ1 ; ℓ1, otherwise }
8:             if σ(ℓ_other) = 1 then                ▷ no update required
9:                 continue
10:            end if
11:            if C is unit on ℓ_other then
12:                ASSIGN(C, ℓ_other)
13:                continue
14:            end if
15:            if C is not falsified then
16:                ℓ ← literal in C \ {ℓ_other} with σ(ℓ) ≠ −1
17:                ω(−ℓ_prop) ← ω(−ℓ_prop) \ {C}
18:                ω(ℓ) ← ω(ℓ) ∪ {C}                ▷ swap watched literal
19:                continue
20:            end if
21:            return C                              ▷ conflict
22:        end for
23:    end while
24:    return ⊥
25: end function
```

Algorithm 39 CDCL loop in GIMSATUL, replacing Alg. 10

```
1: function SOLVE(CNF F) → {UNSAT, (SAT, M ⊨ F)}
2:     if ∅ ∈ F then                                  ▷ formula trivially unsatisfiable
3:         return UNSAT
4:     end if
5:     while true do
6:         if π ≠ ⊥ then
7:             C ← REIMPLYBCP()                       ▷ returns a conflict or ⊥
8:         else
9:             C ← BCP()                              ▷ returns a conflict or ⊥
10:        end if
11:        if C ≠ ⊥ then
12:            if δ(C) = 0 then                       ▷ global conflict
13:                return UNSAT
14:            end if
15:            ANALYZEANDEXPORT(C)
16:            continue
17:        end if
18:        if assigned = |V| then                     ▷ σ is a model
19:            return (SAT, σ)
20:        end if
21:        C ← IMPORT()                               ▷ returns ⊤, ∅ or ⊥
22:        if C = ∅ then                              ▷ global conflict
23:            return UNSAT
24:        end if
25:        if C = ⊤ then                              ▷ new propagation
26:            continue
27:        end if
28:        DECIDE()                                   ▷ otherwise new decision
29:    end while
30: end function
```

Algorithm 40 Boolean constraint propagation with reimplication in GIMSATUL, compare with Alg. 7

```
1: function ReimplyBCP() → F ∪ {⊥}
2:     while π ≠ ∅ do                                          ▷ priority queue
3:         ℓ_prop ← ℓ ∈ π with δ(ℓ) = min{δ(ℓ') | ℓ ∈ π} and
4:                          τ^{-1}(ℓ) < τ^{-1}(ℓ') for all ℓ' ∈ π with δ(ℓ') = δ(ℓ)
5:         π ← π \ {ℓ_prop}
6:         if conflict ≠ ⊥ and δ(conflict) ≤ δ(ℓ) then
7:             π ← ∅
8:             return conflict
9:         end if
10:        PropagateWatch(ℓ_prop)
11:    end while
12:    if conflict ≠ ⊥ then          ▷ if we find a conflict when π is already empty
13:        return conflict
14:    end if
15:    propagated = |τ|              ▷ set propagated for BCP without reimplication
16:    return ⊥
17: end function
```

Algorithm 41 Inner loop of Boolean constraint propagation with reimplication in GIMSATUL, compare with Alg. 21 and Alg. 32

1: **function** PROPAGATEWATCH(ℓ_{prop})
2: **for** $C \in \omega(-\ell_{prop})$ **do**
3: $\langle \ell_1, \ell_2 \rangle \leftarrow \omega^{-1}(C)$ ▷ get the two watched literals for C
4: $\ell_{other} \leftarrow \begin{cases} \ell_2, & \text{if } -\ell_{prop} = \ell_1 \\ \ell_1, & otherwise \end{cases}$
5: **if** $\sigma(\ell_{other}) = 1$ and $\delta(\ell_{other}) \leq \delta(\ell_{prop})$ **then** ▷ watch invariant
6: **continue**
7: **end if**
8: **if** C is unit on ℓ_{other} **then**
9: ASSIGN(C, ℓ_{other})
10: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\delta(\ell) = \delta(C)$
11: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
12: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
13: **continue**
14: **end if**
15: **if** C is unisatisfied on literal ℓ_{other} **then**
16: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\delta(\ell) = \delta(C \setminus \{\ell_{other}\})$
17: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
18: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
19: **if** $\delta(C \setminus \{\ell_{other}\}) < \delta(\ell_{other})$ **then** ▷ missed implication
20: REASSIGN(C, ℓ)
21: **end if**
22: **continue**
23: **end if**
24: **if** C is not falsified **then**
25: $\ell \leftarrow$ literal in $C \setminus \{\ell_{other}\}$ with $\sigma(\ell) \neq -1$
26: $\omega(-\ell_{prop}) \leftarrow \omega(-\ell_{prop}) \setminus \{C\}$
27: $\omega(\ell) \leftarrow \omega(\ell) \cup \{C\}$ ▷ swap watched literal
28: **continue**
29: **end if**
30: **if** $conflict = \bot$ or $\delta(C) < \delta(conflict)$ **then**
31: $conflict = C$
32: **end if**
33: $\pi' \leftarrow \pi' \cup \{\ell\}$ ▷ propagate ℓ later to fix watches
34: **if** $\delta(C) = \delta(\ell_{prop})$ **then**
35: **return** ▷ conflict on currently propagated level
36: **end if**
37: **end for**
38: **end function**

Algorithm 42 Importing clauses with reimplication in GIMSATUL, replacing Alg. 16

1: **function** IMPORT() $\to \{\bot, \emptyset, \top\}$
2: $C \leftarrow$ IMPORTCLAUSE() ▷ maybe get a clause from another thread
3: **if** $C = \bot$ **then** ▷ no new clause
4: **return** $\bot$
5: **end if**
6: **if** C contains satisfied literal ℓ with $\delta(\ell) = 0$ **then** ▷ skip import
7: **return** $\bot$
8: **end if**
9: **if** C is conflicting **then**
10: **if** $\delta(C) = 0$ **then**
11: **return** $\emptyset$
12: **end if**
13: **if** $|\{\ell \in C \mid \delta(\ell) = \delta(C)\}| = 1$ **then**
14: BACKTRACK($\delta(C) - 1$) ▷ chronological backtracking
15: **end if**
16: **end if**
17: $\mathcal{F} \leftarrow \mathcal{F} \cup \{C\}$ ▷ really import the clause
18: **if** $|C| > 1$ **then** ▷ find watches
19: $\ell \leftarrow$ literal in C with $\begin{cases} \sigma(\ell) \neq -1, & \text{if possible} \\ \delta(\ell) = \delta(C), & \text{otherwise} \end{cases}$
20: $\ell' \leftarrow$ literal in $C \setminus \{\ell\}$ with $\begin{cases} \sigma(\ell') \neq -1, & \text{if possible} \\ \delta(\ell') = \delta(C \setminus \{\ell\}), & \text{otherwise} \end{cases}$
21: $\omega(\ell) \leftarrow \omega(\ell) \cup C$ ▷ set watches
22: $\omega(\ell') \leftarrow \omega(\ell') \cup C$
23: **if** C is conflicting **then** ▷ no backtrack
24: $\mathit{propagated} = \tau^{-1}(\ell)$ ▷ handle conflict implicitly in BCP
25: **return** $\top$
26: **end if**
27: **if** C is unisat and $\delta(\ell) > \delta(\ell')$ **then** ▷ missed implication
28: REASSIGN(C, ℓ)
29: **return** $\top$
30: **end if**
31: **end if**
32: **if** $|C| = 1$ and C is satisfied **then**
33: REASSIGN(C, ℓ)
34: **return** $\top$
35: **end if**
36: **if** C is unit on ℓ **then**
37: ASSIGN(C, ℓ)
38: **return** $\top$
39: **end if**
40: **return** $\bot$
41: **end function**

5 Experiments

We have performed experiments to evaluate the impact of reimplication. Thus we have measured run-times for the solvers with reimplication and compare this to their default by disabling reimplication with an option. However, this was not possible for IntelSAT, as the solver has been designed with reimplication and does not include an option to switch it off. Initially we used IntelSAT version sat22 which matches with the one used in the reimplication paper [8]. After identifying a bug in this version[1] we switched to the newest release of IntelSAT.

5.1 Setup

The experiments have been performed on the bwForCluster Helix with the following technical specification for each compute node:

- Processors: 2 x AMD Milan EPYC 7513
- Processor Frequency: 2.6 GHz
- Number of Cores per Node: 64
- RAM: 236 GB

[1] the bug did not occur on the benchmark set of the SAT competition 2022 which were used for the final experiments

All experiments were run with the benchmark set of the SAT competition of 2022 [25] with a timeout of 5000 seconds and a memory limit of 7000 MB per core. For technical reasons we had to rely on the internal timeout of IntelSAT but this had no impact on the results.

The jobs were scheduled such that as few nodes as possible were used. Since 64 jobs can run on the same node with only 236 GB, the memory was overcommited, i.e., if all jobs would have used all of their memory, they would hit the hard limit of the node. However, all memouts we observed were due to exceeding the 7000 MB limit. For the parallel experiments the memory limit was scaled with the number of cores, e.g., for two cores it was 14000 MB.

5.2 Results

To compare run-times we plot the solved instances on the y-axis and the time on the x-axis. The result for each solver is sorted separatly. Intuitively, this translates to a higher line for a faster solver. However, no statement can be made about individual instances and two seemingly similar lines could have wildly different solver behaviour.

Figure 1 shows a comparison of all three solvers with their default settings where CaDiCaL and Gimsatul are run with and without reimplication. The old version of IntelSAT is included for completeness. Figure 2 shows Gimsatul with 1, 2, 4, and 8 threads, each with and without reimplication. The results are somewhat disappointing as it is not possible to say wether reimplication has a positive or negative impact for this benchmark set.

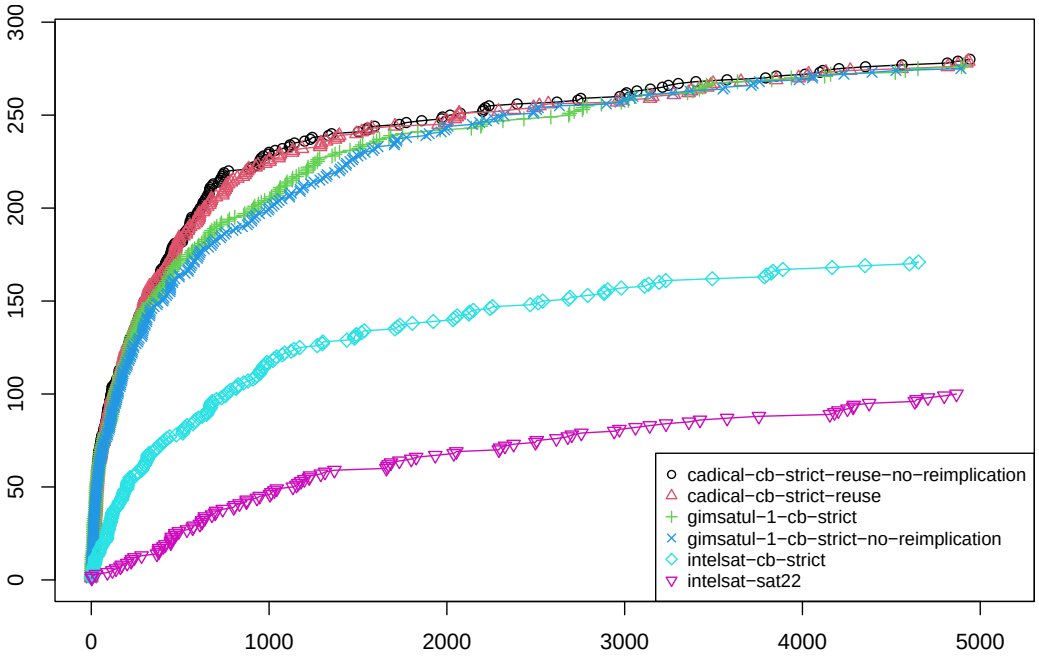

Figure 1: Solved instances for all solvers with and without reimplication

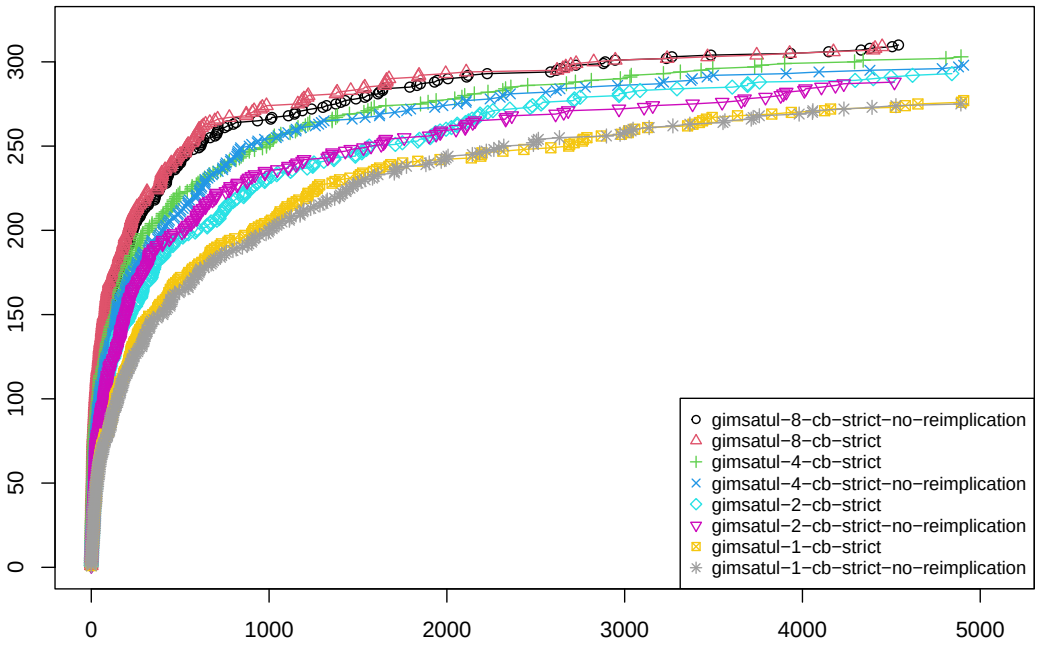

Figure 2: Solved instances for GIMSATUL with and without reimplication

5.3 Statistics

Because of the inconclusive run-time results, we decided to collect statistics in order to get a better understanding of the impact of reimplication. For the statistics we plot the instances on the x-axis instead of the y-axis. This preserves the intuition that a higher line corresponds to bigger numbers. When the range of the numbers is too big the y-axis is scaled logarithmically. Each line is sorted independently as before, therefore no statement can be made about individual instances.

While running these experiments and collecting the statistics we realised that the solvers have different heuristics for chronological backracking:

cb-strict This simple heuristic was proposed in the original chronological backtrack paper of Nadel and Ryvchin [6]. When the backjump level is at least 100 levels below the conflict level, chronological backtracking is applied instead. We call this heuristic cb-strict and it is the default heuristic in Gimsatul.

cb-reuse We call the default heuristic of IntelSAT cb-reuse. When the backjump level is at least 100 levels below the conflict level, i.e., when cb-strict would do chronological backtracking, the solver searches for the best level between backjump and conflict level. It is determined by variable scores which are also used in the decision heuristic of the solver. The solver will then backtrack to the determined level. We count all conflicts where the backjump level is smaller than the chosen backtrack level as chronological backtracking even though strictly speaking the terminology implies backtracking exactly one level.

cb-strict-reuse The default heuristic in CaDiCaL is similar to the one in IntelSAT in that it also searches for a better backtrack level by using variable scores. However, when the condition for cb-strict is met it does strict chronological backtracking, i.e.,

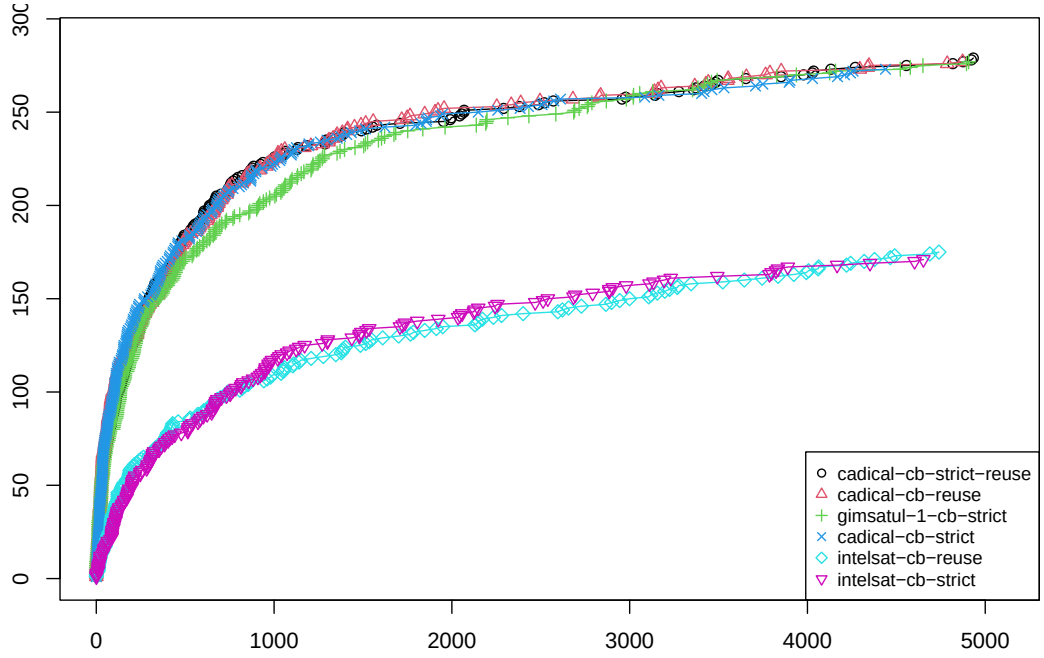

Figure 3: Solved instances for different chronological backtracking heuristics

backtracks exactly one level. Only when the distance to the backjump level is smaller it applies the search for a better level. We call this heuristic cb-strict-reuse.

Additionally to their default, cb-strict was already implemented in all three solvers and cb-reuse could be added to CaDiCaL with little implementational overhead. Figure 3 shows the results when running the different configurations with reimplication. The heuristic does not seem to impact the efficiency of the solver too much. For the statistics below we always compare all available configurations.

Figure 4 shows the total number of conflicts and chronological backtracks which reveals the irregularity of cb-strict-reuse. While the number of chronological backtracks is similar between the other heuristics, it is much closer to the total number of conflicts for cb-strict-reuse. This is emphasised again by the ratio of chronological backtracks to total number of backtracks (Fig. 5).

Other than the number of chronological backtracks the heuristic also impacts the

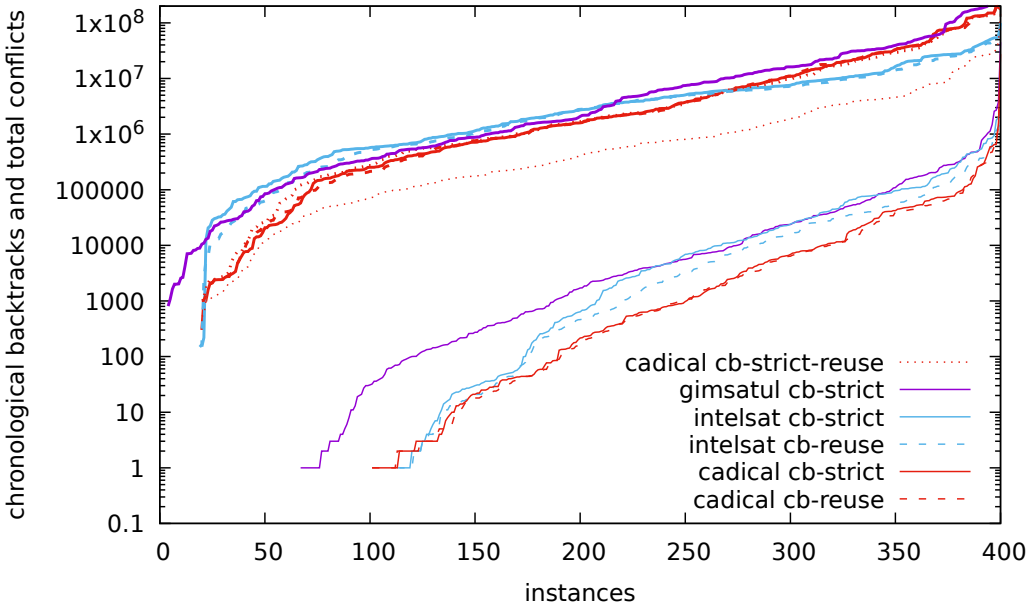

Figure 4: Total number of chronological backtracks and conflicts for all solvers on a logarithmic scale

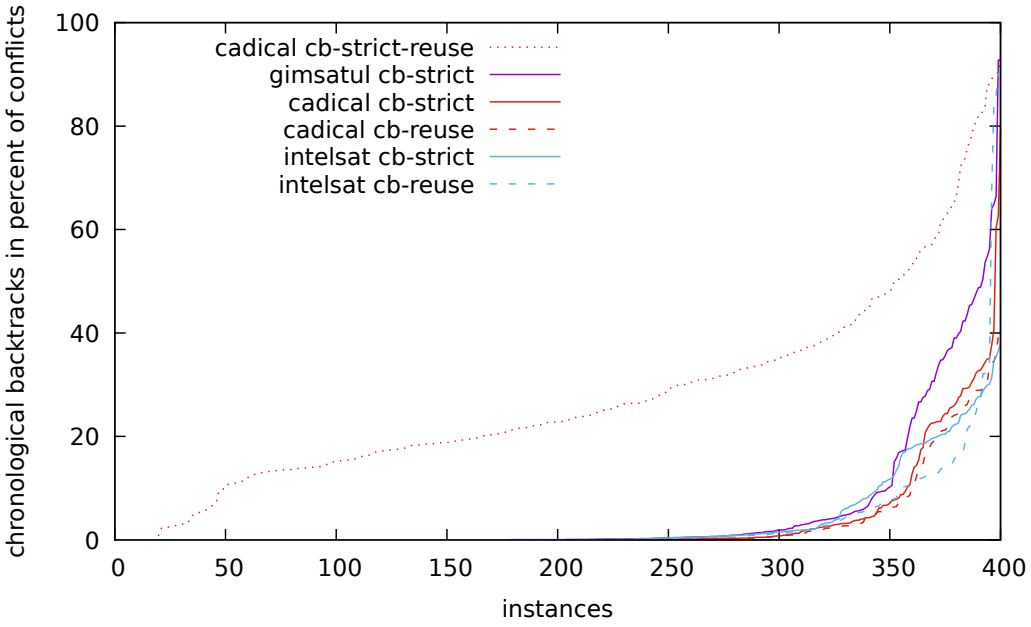

Figure 5: Chronological backtracks in percent of conflicts for all solvers

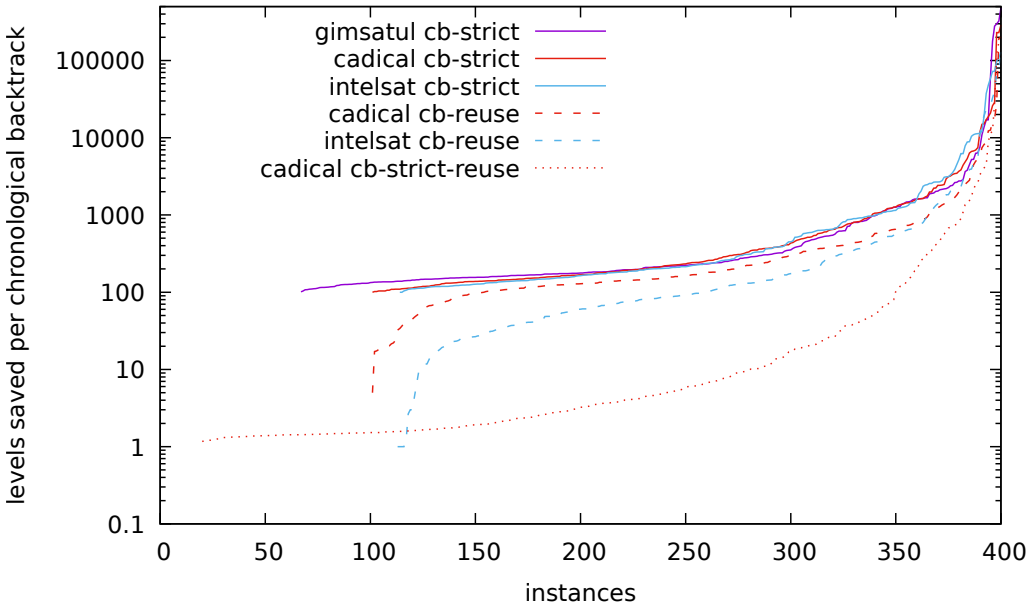

Figure 6: Average number of levels saved per chronological backtrack for all solvers on a logarithmic scale

number of levels saved, i.e., the difference between backjump and backtrack level. Figure 6 shows the ratio of levels saved per chronological backtrack. For cb-strict this starts at 100 levels. With cb-reuse it is slightly lower due to the search for a better level. There are benchmarks where chronological backtracking never happens for cb-strict and cb-reuse. From the previous plot we already saw that cb-strict-reuse triggers much more chronological backtracking. Therefore the number of levels saved is diluted and starts at one which is the minimum to be counted as chronological backtracking.

As a measure of efficiency we plot propagations per second (Fig. 7). This measure counts the number of literals which are processed by Boolean constraint propagation. Since BCP is the hotspot of the solver the number of propagations per second correlates with solving speed, but is of course not the only factor. IntelSAT for example shows a similar number of propagations per second and still solves much less problems, possibly because it does not do pre- and inprocessing. The old version

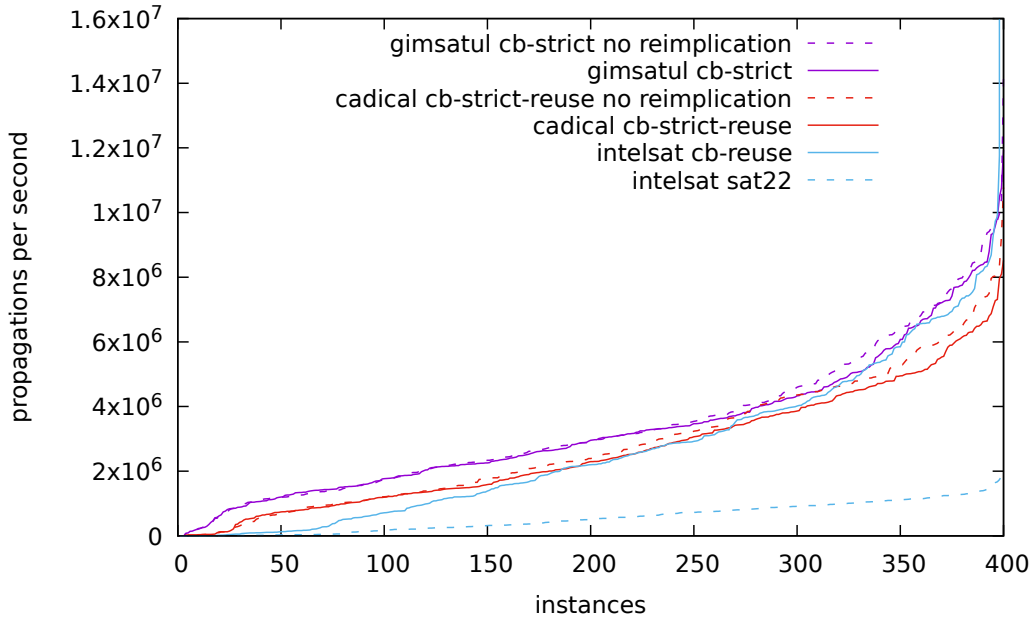

Figure 7: Average number of propagations per second for all solvers

of IntelSAT on the other hand likely was much less optimized for fast propagation. We can observe a slight decrease of propagations per second with reimplication, potentially a result of a slightly slower routine for BCP with reimplication.

For CaDiCaL and Gimsatul we can compare the number of propagations with reimplication (or reimplication propagations) to the total number of propagations. We do this with the absolute values (Fig. 8) and the ratio (Fig. 9). For IntelSAT we plot the number of elevated literals instead of reimplication propagations which approximates the additional work done in the reimplication routine. We can observe that CaDiCaL cb-strict has much less reimplication propagations compared to Gimsatul cb-strict which is a result of the different switching techniques in the two solvers. On the other hand, using the cb-strict-reuse heuristic greatly increases the number of reimplication propagations. This is not surprising as we expect this measure to correlate with the number of chronological backtracks. However, Gimsatul still caps out at a much higher percentage for reimplication propagations.

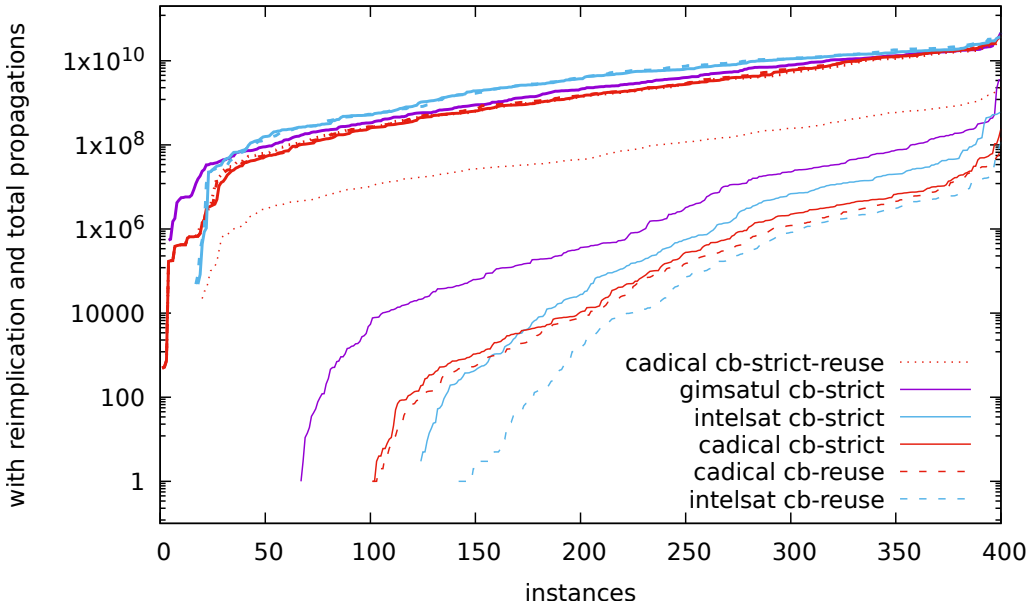

Figure 8: Total number of all propagations and reimplication propagations for all solvers on a logarithmic scale

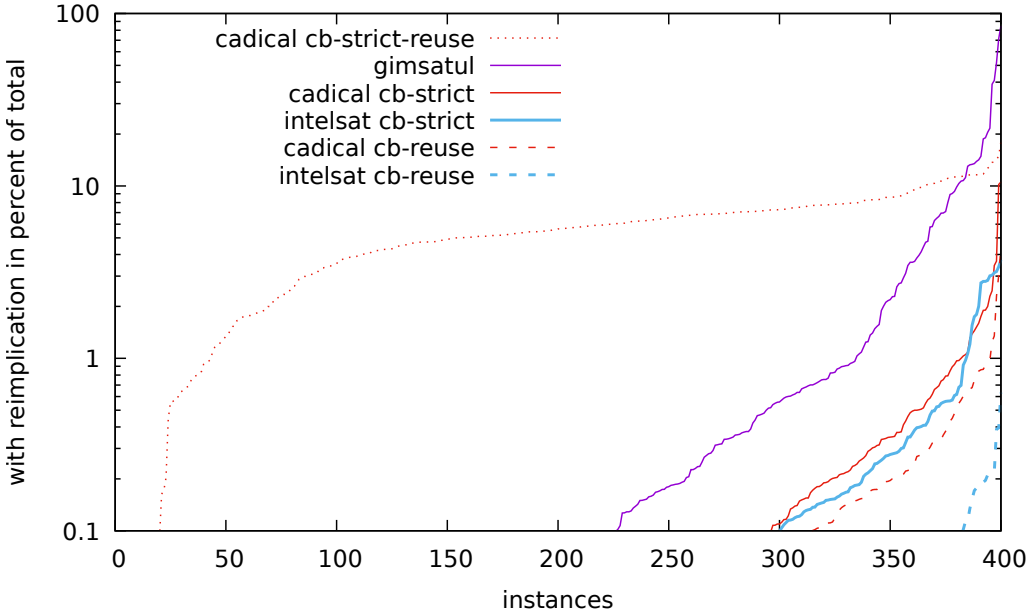

Figure 9: Reimplication propagations in percent of all propagations for all solvers on a logarithmic scale

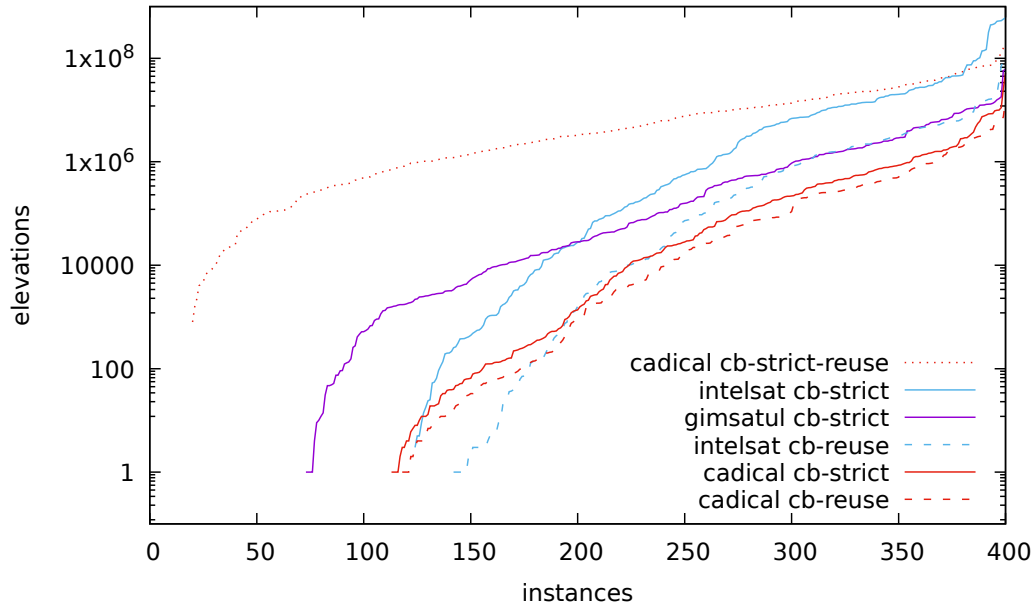

Figure 10: Total number of elevations for all solvers on a logarithmic scale

A similar trend can be observed in the total number of elevations (Fig. 10), except that cadical cb-strict-reuse is much closer to the other solvers. Apart from cb-strict-reuse, IntelSAT has the next highest number of elevations. This might result from the main difference of the implementation. IntelSAT always performs BCP and reimplication until a fixed point is reached on the lowest conflict level it can find. Gimsatul and CaDiCaL on the other hand can abort propagation as soon as the propagation level reaches the conflict level which avoids propagating all the literals on the conflict level and all higher levels, if the conflict is found early.

Usually, each elevated literal will trigger a reimplication propagation such that the ratio of these two will always be below one (Fig. 11). We can interpret this as the efficiency of switching propagation routines, which is lower for Gimsatul and even lower for cb-strict-reuse implying that the number of reimplication propagations grows more with chronological backtracks than the number of elevations. This is confirmed by Figure 12. The trend flips again when looking at the elevations per level saved by chronological backtracking (Fig. 13). By producing way more chronological

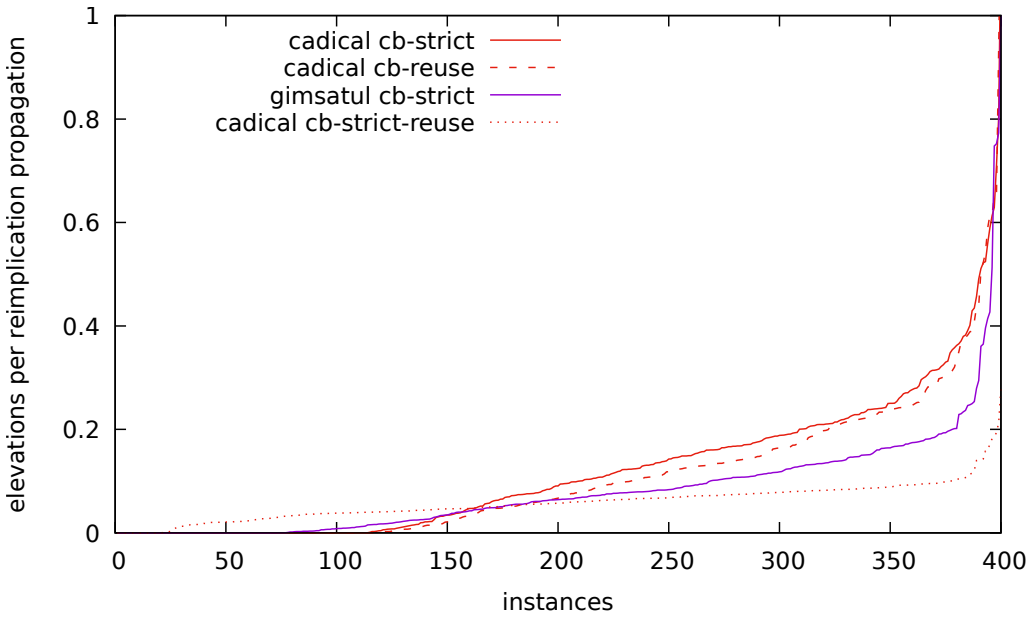

Figure 11: Average number of elevations per reimplication propagation for all solvers

backtracks but saving way less levels, cb-strict-reuse inevitably gets more elevations per level saved than the other solvers.

5.4 Statistics for Parallel Clause Sharing

All the solvers are inherently unstable, i.e., when faced with two similar problems they can produce wildy different results, or conversely, slightly changing a heuristic detail can drastically change the run-time for the same instance. However, when running the same version of the solver with the same problem instance, the result is repeatable. This changes with multithreaded GIMSATUL as importing and exporting clauses is timing dependent and up chance. Therefore, the parallel results and statistics show more fluctuations are much less repeatable.

We have collected statistics for GIMSATUL using 2, 4 and 8 threads. For each of these there are two ways of counting, sum and single. The former sums over all threads,

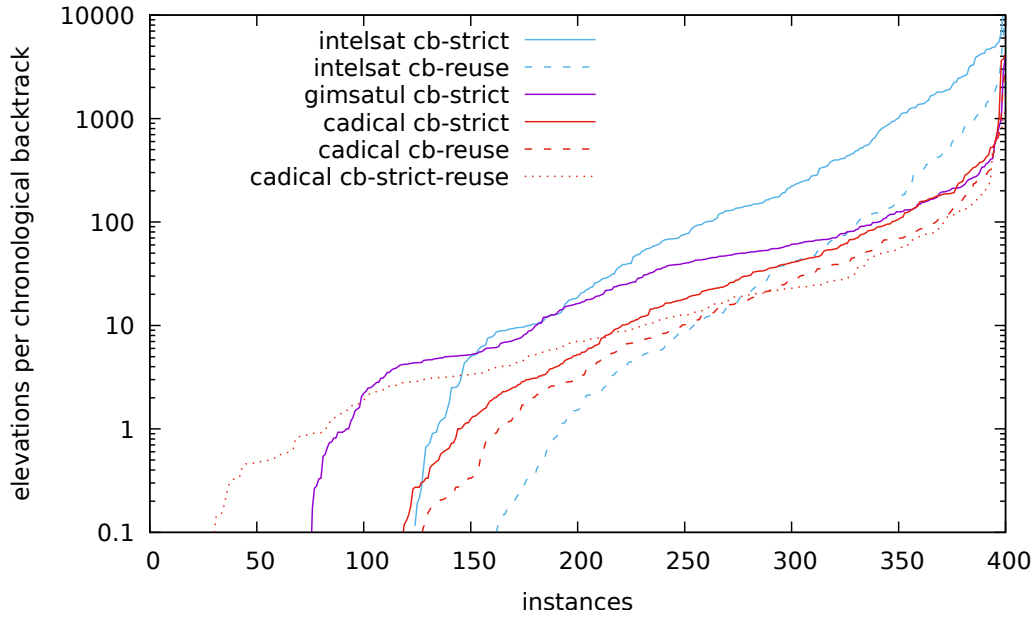

Figure 12: Average number of elevations per chronological backtrack for all solvers on a logarithmic scale

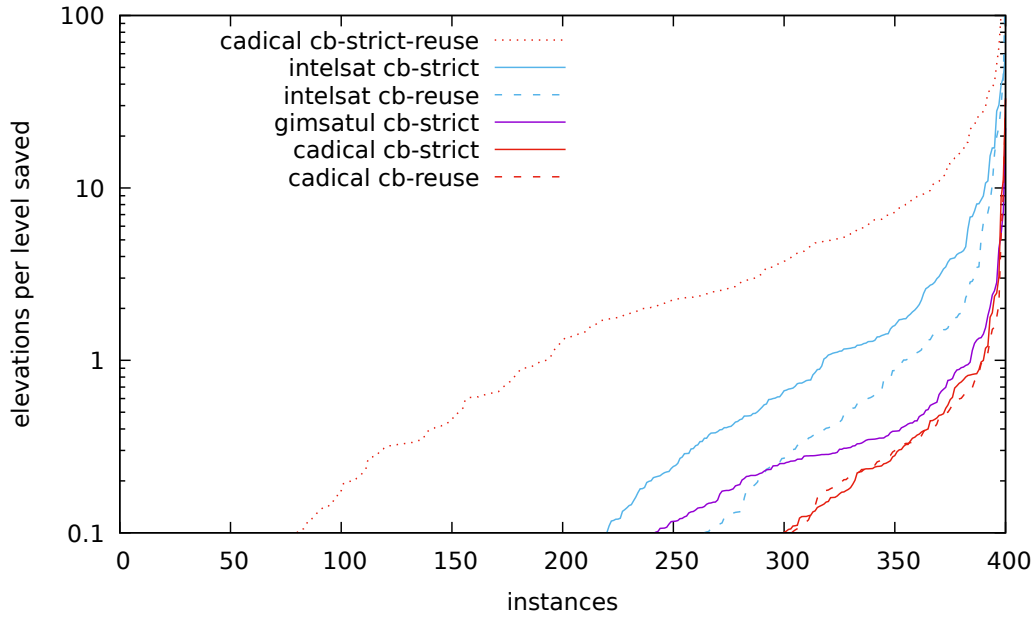

Figure 13: Average number of elevations per level saved with chronological backtracking for all solvers on a logarithmic scale

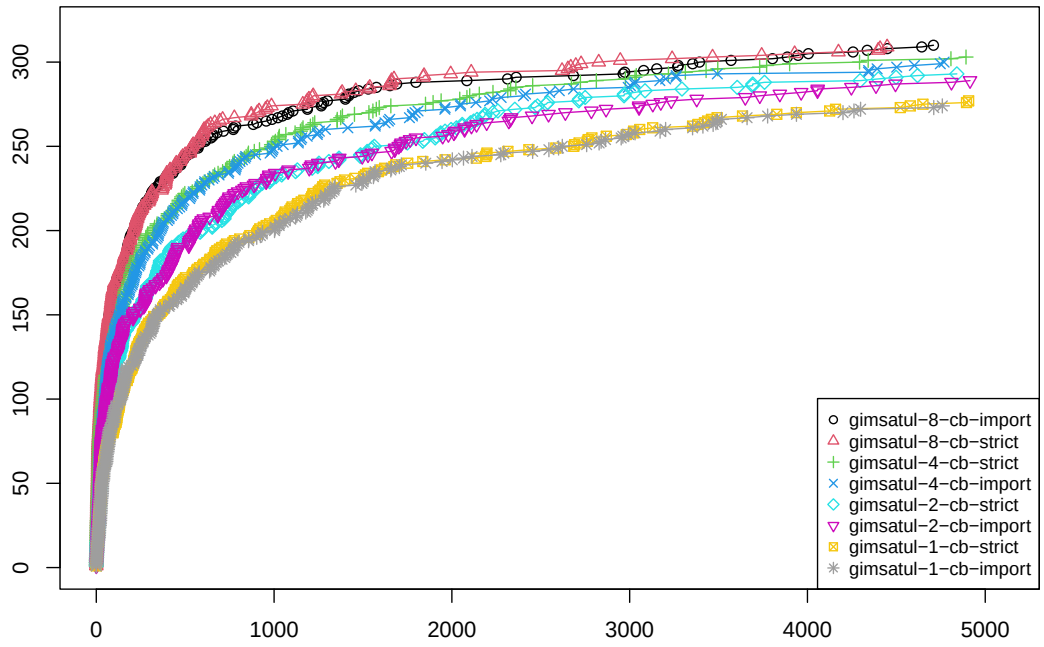

Figure 14: Solved instances for GIMSATUL with and without chronological backtracking

collecting the global solver statistics. The latter looks at an individual thread to see how it is impacted by having other threads run in parallel.

cb-import Reimplication can now happen when importing clauses. We can single out the impact of this by switching off the chronological backtrack heuristic. This means the solver will always backtrack to the backjump level when it finds a conflict but it still elevates literals from missed implications that are imported. We call this setting cb-import. As the solver can only import clauses when running in parallel, we omit this option for GIMSATUL with a single thread.

Again, run-time results for GIMSATUL cb-stric and cb-import are inconclusive (Fig. 14) so we focus on statistics instead.

When looking at the total number of conflicts and chronological backtracks (Fig. 15) we can observe a pattern which appears again and again. Summing over all threads

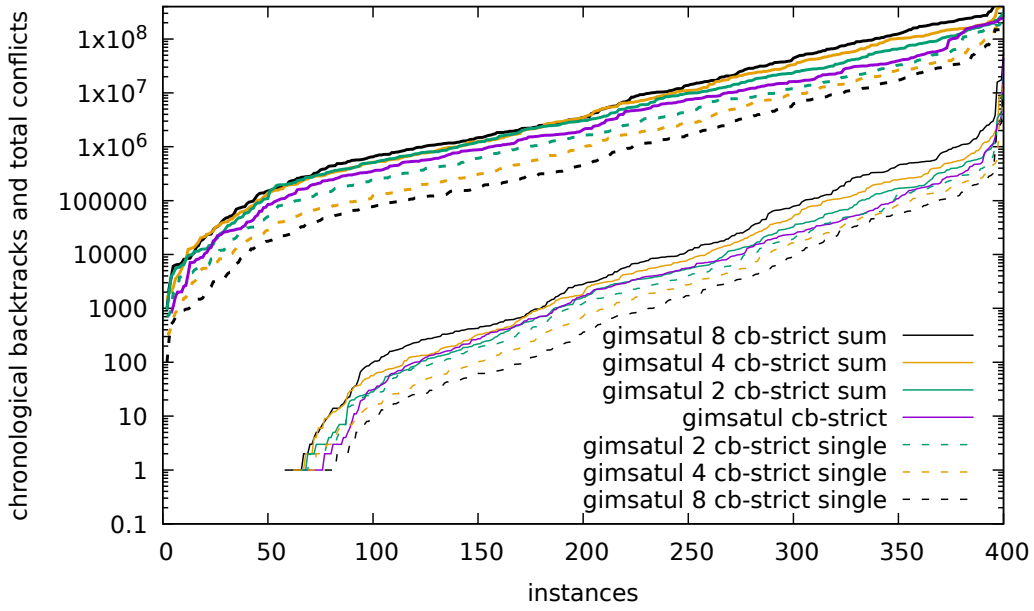

Figure 15: Total number of conflicts and chronological backtracks for GIMSATUL on a logarithmic scale

increases the total, so the lines are sorted with 8 on top, then 4, 2 and finally single threaded. When we focus on a single thread instead, this pattern is inverted, we get single thread on top, then 2, 4 and lowest is with 8 threads. This is because the additional threads add some overhead to the solver which has to make sure they are synchronised. We omit this figure for cb-import since it cannot produce any chronological backtracks.

The above mentioned pattern vanishes when we look at ratios. When we sum two measures over multiple threads and then divide, this is essentially like the average of these threads. So it is not surprising that the lines in Figure 16 and Figure 17 match, which show the percentage of chronological backtracks per conflict and ratio of levels saved per chronological backtrack respectively.

It gets interesting again when we look at a ratio where one of the measures is independent of the number of threads like propagations per second (Fig. 18 and 19). Interestingly, adding threads does not seem to increase the number of propagations

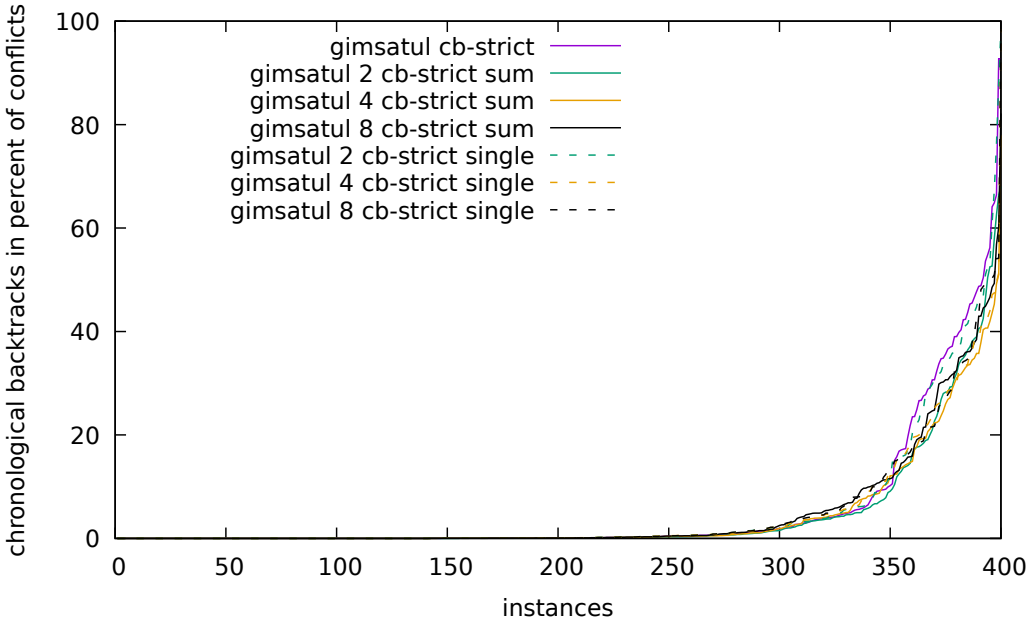

Figure 16: Chronological backtracks in percent of conflicts for GIMSATUL

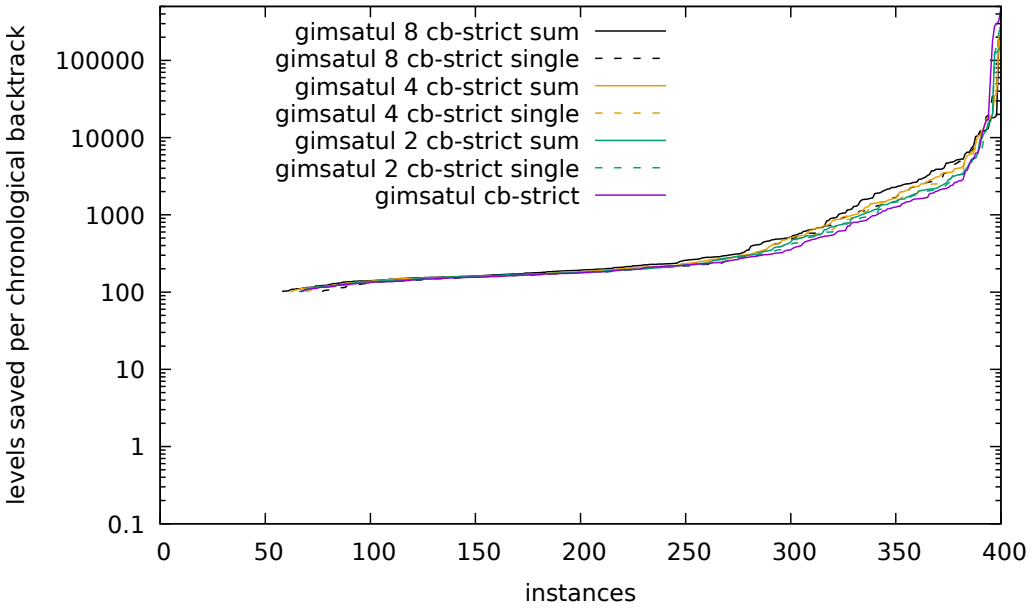

Figure 17: Average number of levels saved per chronological backtrack for GIMSATUL on a logarithmic scale

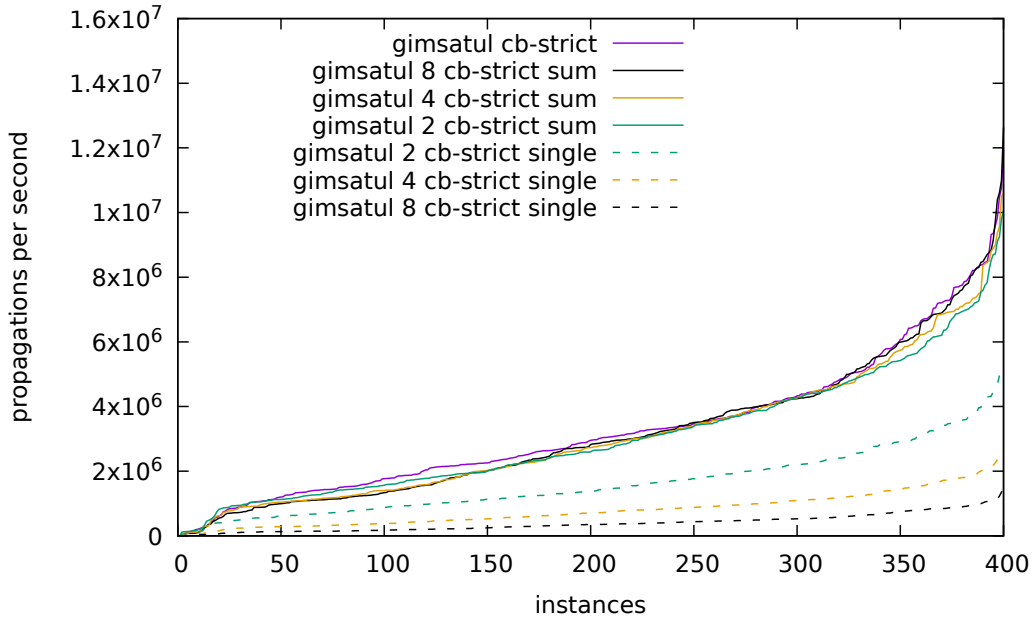

Figure 18: Average number of propagations per second for GIMSATUL

per second. Again this implies that this is not the only measure for efficiency of the solver and that clause sharing and exploring different parts of the search space with different solver threads can really increase effectiveness.

For the next few plots we see an interesting new pattern emerge. When looking at cb-import, the lines when focusing at a single thread are not sorted in reverse order like in the pattern mentioned above, but instead always favour more theads. This is because the statistic we measure can only be introduced by chronological backtracking or parallel clause sharing. If we switch of the former we see the impact of the latter, which only correlates with the number of threads (Fig. 20, 21, 22 and 23). When looking carefully at the plots for cb-strict we can see the same pattern forming at the beginning of the line, for the benchmarks where no chronological backtracking happens anyways. When we get to the benchmarks where chronological backtracking starts to happen, i.e., the beginning fo the purple line, we see this pattern flipping to the original one, where the lines are sorted in reverse order, indicating that the impact of chronological backtracking within a single thread is usually much bigger

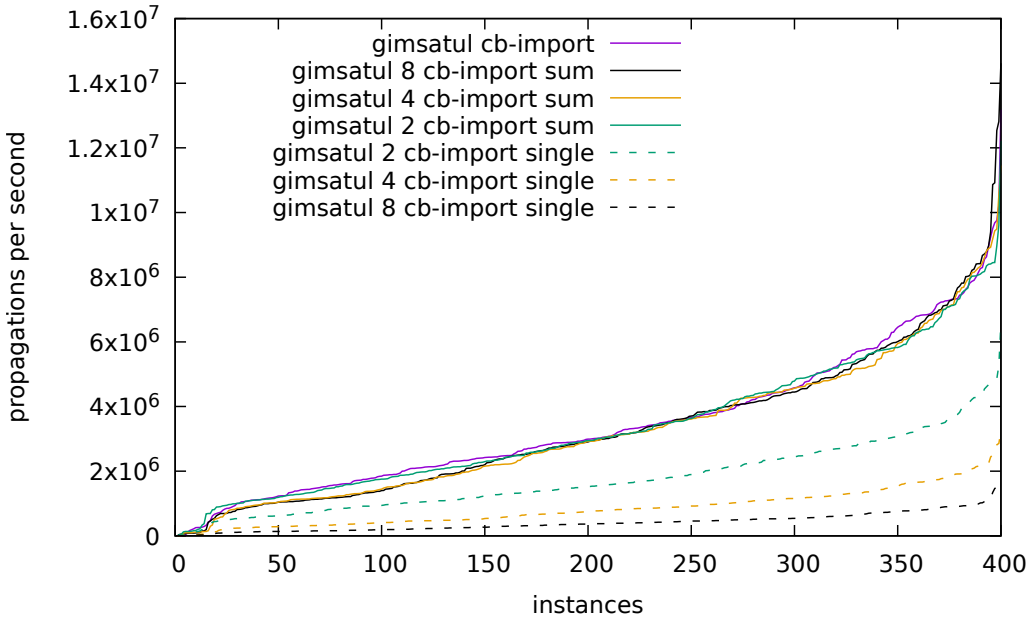

Figure 19: Average number of propagations per second for GIMSATUL without chronological backtracking

than that of importing clauses.

This also changes the pattern which is observed in the ratio of reimplication propagations to all propagations (Fig. 24). For cb-strict, all the lines are on top of each other because building the ratio cancels the pattern. For cb-import however this is different. The number of all propagations uses the first pattern, the number of propagations with reimplication the second. Building the ratio therefore always prefers more threads first, independently of whether we focus on a single thread or all together.

The efficiency of elevations per reimplication propagation goes up with more threads (Fig. 25). It is not immedieatly obvious why, but a reasonable explanation would be that reimplication propagation is triggered after a chronological backtrack which does not necessarily lead to an elevation. With clause import, reimplication propagation can be triggered with an elevation without conflict. The same can be observed for elevations per chronological backtrack (Fig. 26) and elevations per level saved

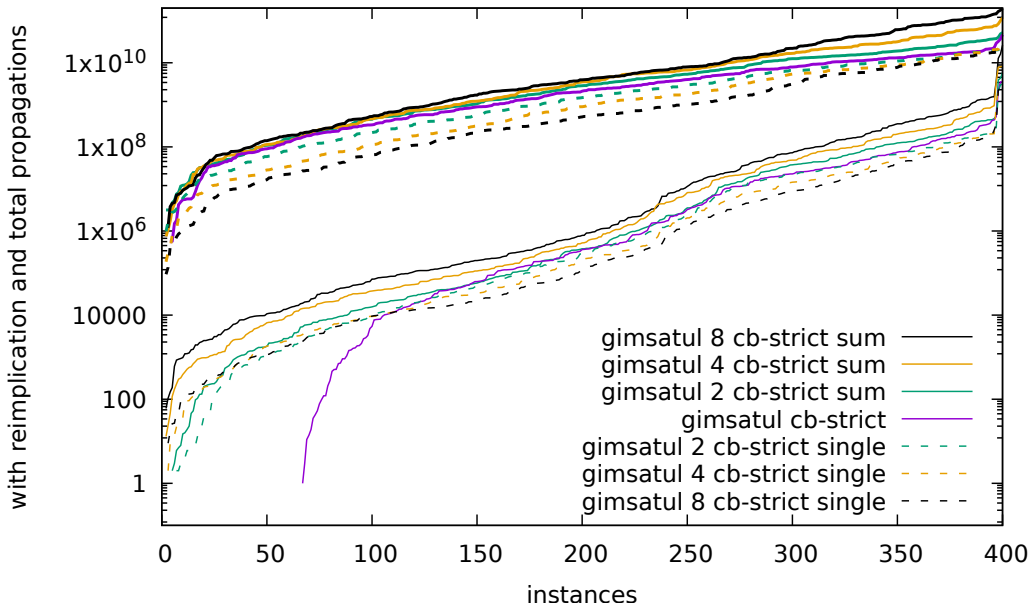

Figure 20: Total number of all propagations and reimplication propagations for GIMSATUL on a logarithmic scale

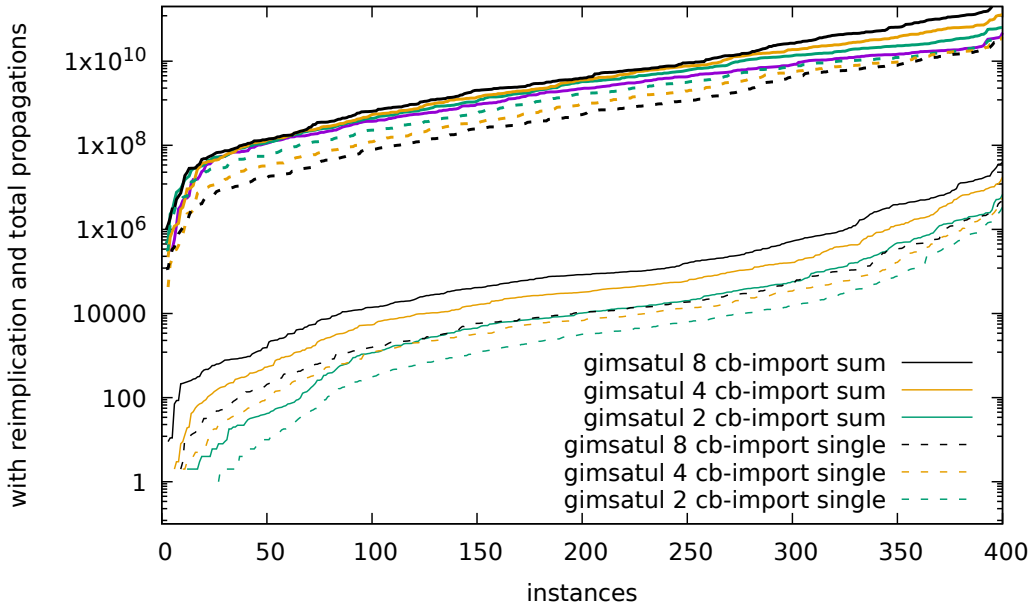

Figure 21: Total number of all propagations and reimplication propagations for GIMSATUL without chronological backtracking on a logarithmic scale

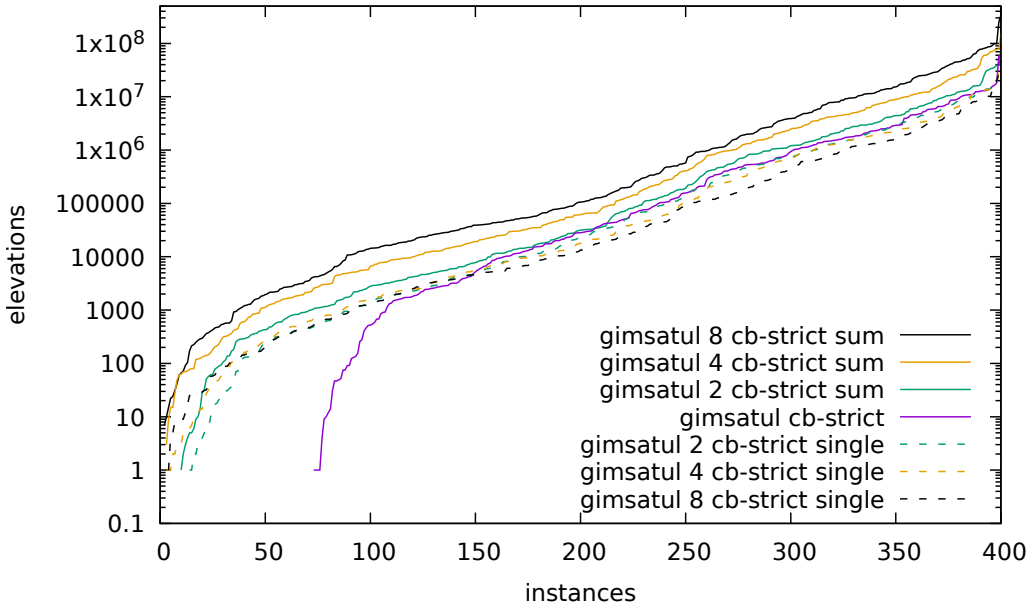

Figure 22: Total number of elevations for GIMSATUL on a logarithmic scale

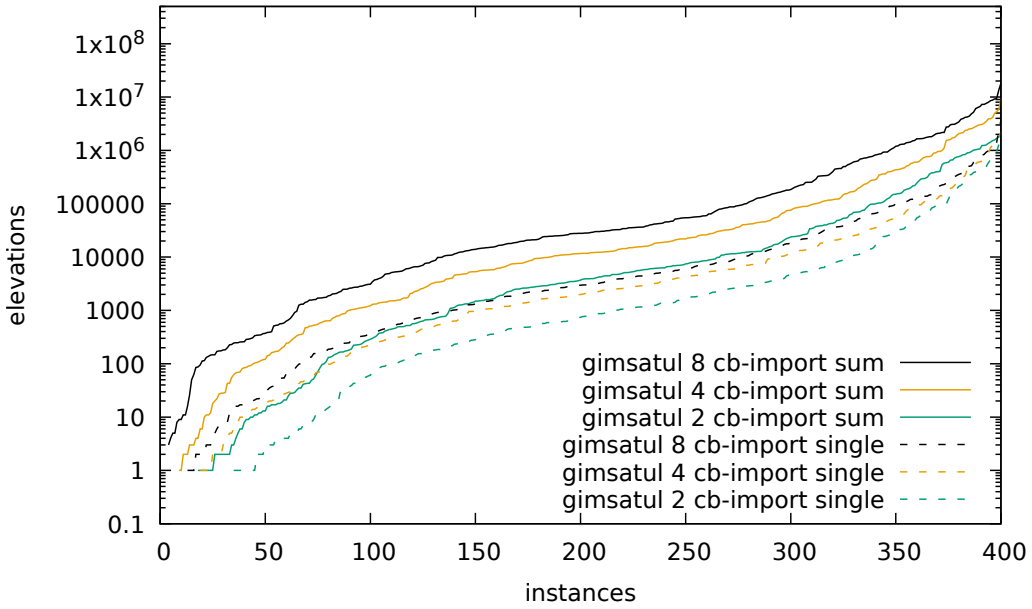

Figure 23: Total number of elevations for GIMSATUL without chronological backtracking on a logarithmic scale

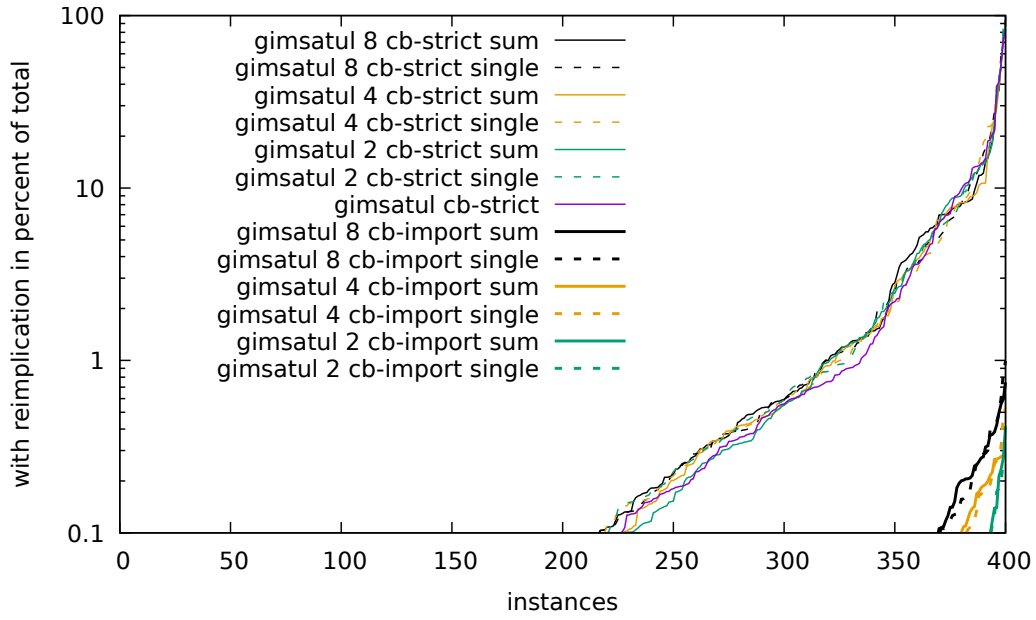

Figure 24: Reimplication propagations in percent of all propagations for GIMSATUL on a logarithmic scale

(Fig. 27). Again this is likely just because the number of elevations goes up with more threads, while the number of conflicts stays similar due to the heuristics of the solver (fig 28).

For cb-import we can see no clear pattern in the number of elevations per reimplication propagation. (Fig. 29). This suggests that it is mostly up to chance wether an imported clause triggers reimplication propagation through an elevation or simply with an out-or-order assignment.

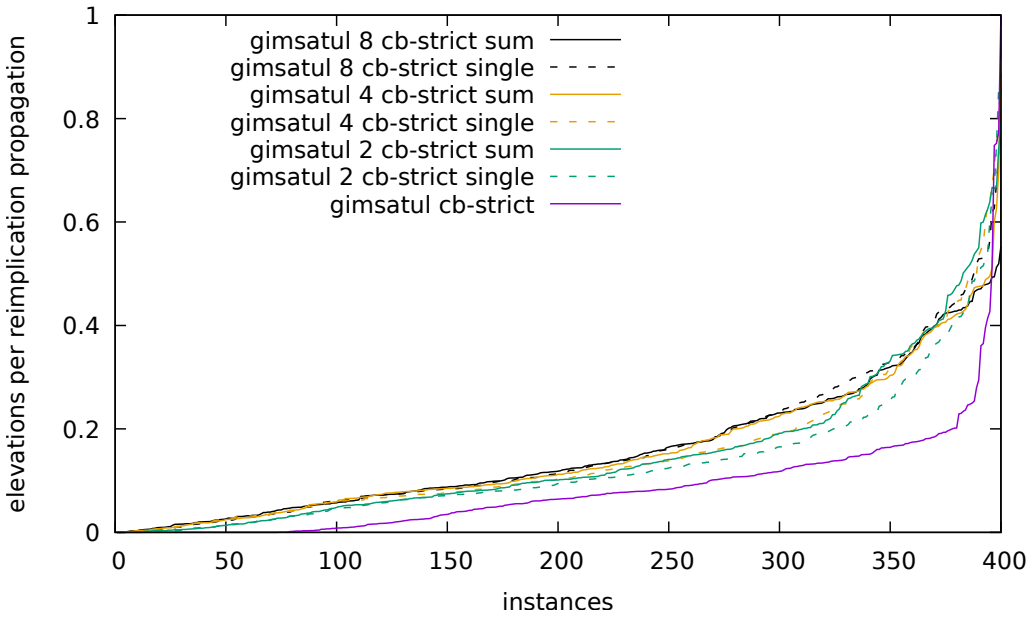

Figure 25: Average number of elevations per reimplication propagation for GIMSATUL

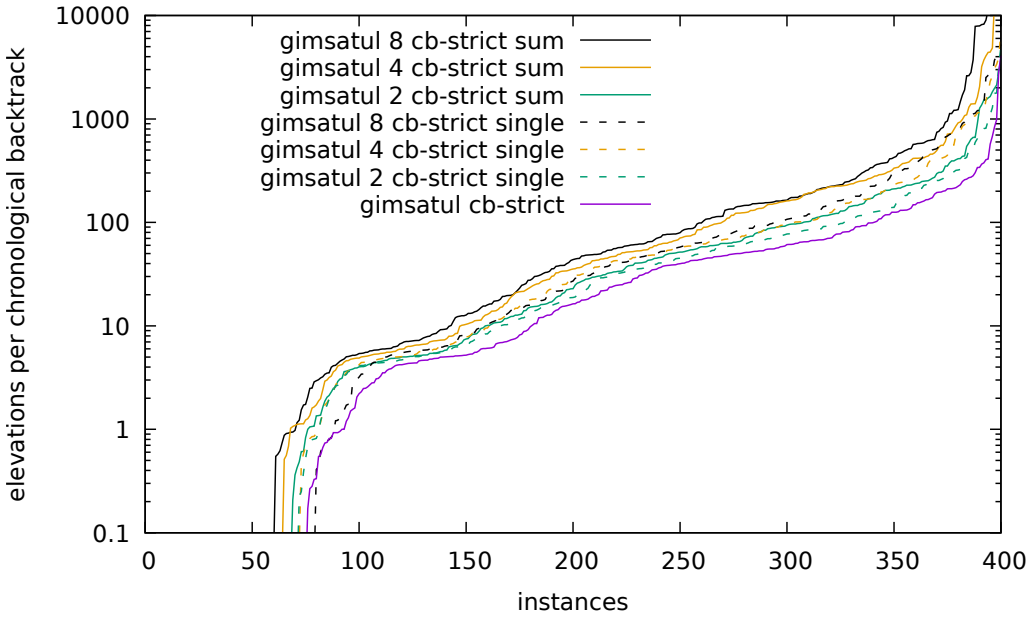

Figure 26: Average number of elevations per chronological backtrack for GIMSATUL on a logarithmic scale

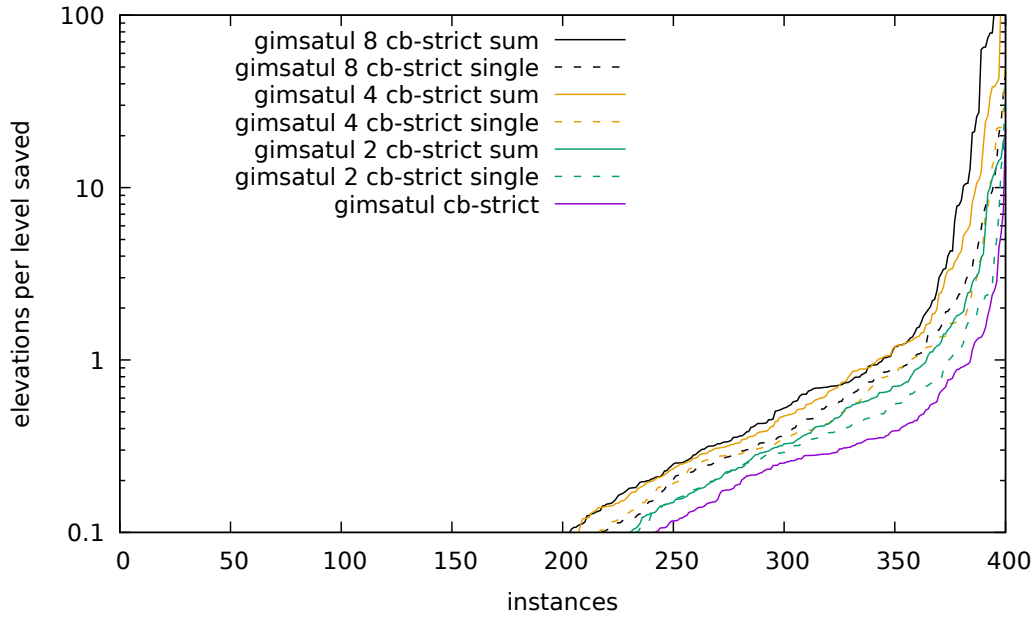

Figure 27: Average number of elevations per level saved with chronological backtracking for GIMSATUL on a logarithmic scale

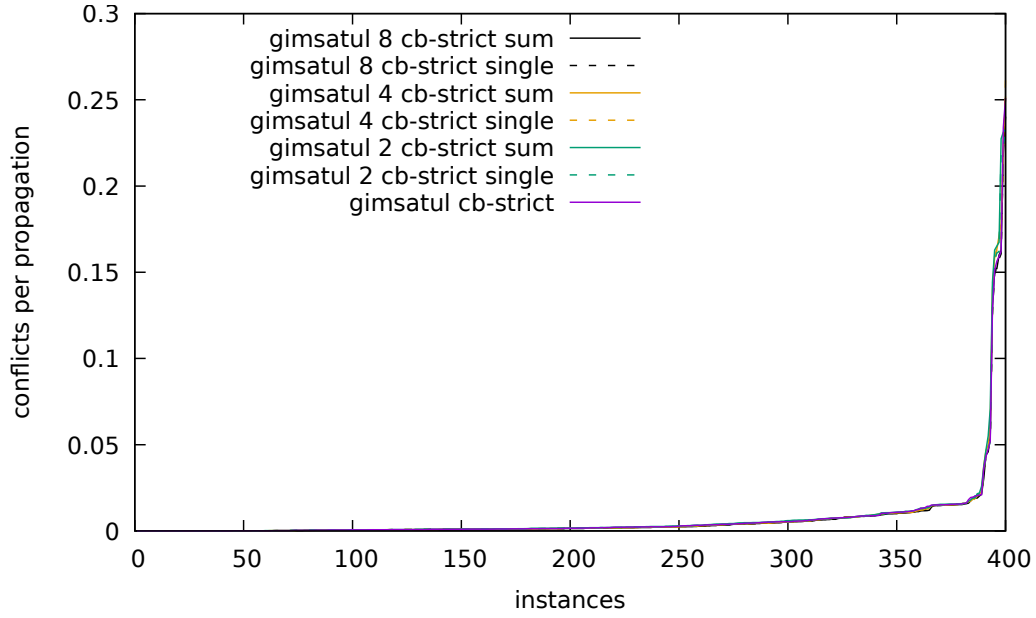

Figure 28: Average number of conflicts per propagation for GIMSATUL

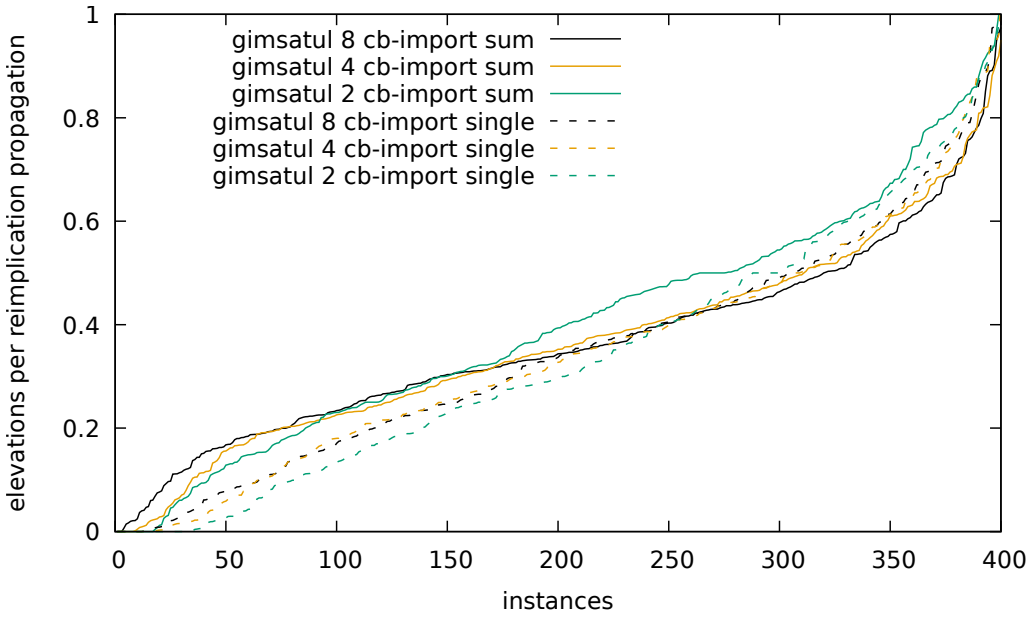

Figure 29: Average number of elevations per reimplication propagation for GIMSATUL without chronological backtracking

6 Conclusion

In this thesis we have compared three different implementations of reimplication in the three SAT solvers IntelSAT, CaDiCaL and Gimsatul. We have added reimplication to Gimsatul for this thesis and to CaDiCaL in previous project. After revisiting standard CDCL algorithms, we have elaborated on how reimplication builds upon chronological backtracking and how this idea can be applied to importing clauses in parallel SAT solving. We have provided a new intuition on reimplication, discussed the implementational overhead and have given different approaches to solve the challenges of integrating reimplication into a modern SAT solver. This has furthered our own understanding for the subtleties and challenges of implementing such a complex an invasive technique. We have provided a thourough experimental section with a focus on interpreting statistics extracted from the solvers. Our initial goal of improving the parallel SAT solver Gimsatul with reimplication has only partially succeeded as our run-time results are inconclusive at best.

Future Work It would be of interest to expand the experimental analysis to applications of incremental SAT solving and the user propagator, both of which can be sources of missed implications and out-of-order assignments. Beyond that, further optimisations should be considered to decide wether reimplication itself gives no improvement over chronological backtracking or the observed results are the product of the implementations.

7 Acknowledgments

- I would like to thank Armin and Mathias as great advisers and proofreaders

- The authors acknowledge the service of the bwForCluster Helix supported by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG.

Bibliography

[1] A. Gupta, M. K. Ganai, and C. Wang, "SAT-based verification methods and applications in hardware verification," in *Formal Methods for Hardware Verification* (M. Bernardo and A. Cimatti, eds.), (Berlin, Heidelberg), pp. 108–143, Springer Berlin Heidelberg, 2006.

[2] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in IntelCoreTM i7 processor execution engine validation," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), (Berlin, Heidelberg), pp. 414–429, Springer Berlin Heidelberg, 2009.

[3] N. Rungta, "A billion SMT queries a day (invited paper)," in *Computer Aided Verification* (S. Shoham and Y. Vizel, eds.), (Cham), pp. 3–18, Springer International Publishing, 2022.

[4] J. Marques-Silva and K. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[5] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 530–535, 2001.

[6] A. Nadel and V. Ryvchin, "Chronological backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings* (O. Beyersdorff and C. M. Wintersteiger, eds.), vol. 10929 of *Lecture Notes in Computer Science*, pp. 111–121, Springer, 2018.

[7] S. Möhle and A. Biere, "Backing backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings* (M. Janota and I. Lynce, eds.), vol. 11628 of *Lecture Notes in Computer Science*, pp. 250–266, Springer, 2019.

[8] A. Nadel, "Introducing Intel(R) SAT Solver," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)* (K. S. Meel and O. Strichman, eds.), vol. 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:23, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[9] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, p. 394–397, jul 1962.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[11] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental SAT formulation of proof- and counterexample-based abstraction," 2010.

[12] T. Paxian, S. Reimer, and B. Becker, "Dynamic polynomial watchdog encoding for solving weighted MaxSAT," in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018,*

Proceedings (O. Beyersdorff and C. M. Wintersteiger, eds.), vol. 10929 of *Lecture Notes in Computer Science*, pp. 37–53, Springer, 2018.

[13] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere, "IPASIR-UP: User propagators for CDCL," in *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, Alghero, Italy* (M. Mahajan and F. Slivovsky, eds.), vol. 271 of *LIPIcs*, pp. 8:1–8:13, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[14] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing – SAT 2020* (L. Pulina and M. Seidl, eds.), (Cham), pp. 46–61, Springer International Publishing, 2020.

[15] R. Coutelier, "Chronological vs. non-chronological backtracking in satisfiability modulo theories (Unpublished master's thesis)," Université de Liège, Liège, Belgique, 2023.

[16] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[17] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware SAT solvers," in *Logic for Programming, Artificial Intelligence, and Reasoning* (C. G. Fermüller and A. Voronkov, eds.), (Berlin, Heidelberg), pp. 519–534, Springer Berlin Heidelberg, 2010.

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver.," in *DAC*, pp. 530–535, ACM, 2001.

[19] L. Ryan, "Efficient algorithms for clause-learning SAT solvers," Master's thesis, Simon Fraser University, 2004.

[20] G. Chu, A. Harwood, and P. J. Stuckey, "Cache conscious data structures for Boolean satisfiability solvers," *JSAT*, vol. 6, no. 1-3, pp. 99–120, 2009.

[21] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais, "Diversification and intensification in parallel SAT solving," in *Principles and Practice of Constraint Programming – CP 2010* (D. Cohen, ed.), (Berlin, Heidelberg), pp. 252–265, Springer Berlin Heidelberg, 2010.

[22] M. Fleury and A. Biere, "Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses," *CoRR*, vol. abs/2207.13577, 2022.

[23] A. Biere, M. Fleury, and F. Pollitt, "CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023," in *Proc. of SAT Competition 2023 – Solver and Benchmark Descriptions* (T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2023-1 of *Department of Computer Science Report Series B*, pp. 14–15, University of Helsinki, 2023.

[24] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2020-1 of *Department of Computer Science Report Series B*, pp. 51–53, University of Helsinki, 2020.

[25] T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds., *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B, Finland: Department of Computer Science, University of Helsinki, 2022.