Lazy Reimplication in Chronological Backtracking

Robin Coutelier
TU Wien, Vienna, Austria

Mathias Fleury
University Freiburg, Freiburg, Germany

Laura Kovács
TU Wien, Vienna, Austria

Abstract

Chronological backtracking is an interesting SAT solving technique within CDCL reasoning, as it backtracks less aggressively upon conflicts. However, chronological backtracking is more difficult to maintain due to its weaker SAT solving invariants. This paper introduces a lazy reimplication procedure for missed lower implications in chronological backtracking. Our method saves propagations by reimplying literals on demand, rather than eagerly. Due to its modularity, our work can be replicated in other solvers, as shown by our results in the solvers CaDiCaL and Glucose.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Automated reasoning

Keywords and phrases Chronological Backtracking, CDCL, Invariants, Watcher Lists

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.24

Supplementary Material *CaDiCaL (Source Code)*: `https://github.com/arminbiere/cadical/tree/strong-backtrack`
NapSAT (Source Code GitLab): `https://gitlab.uliege.be/smt-modules/sat-library`
NapSAT (Source Code GitHub): `https://github.com/RobCoutel/NapSAT`
Glucose (Source Code): `https://github.com/m-fleury/glucose`

Acknowledgements The authors acknowledge support from the ERC Consolidator Grant ARTIST 101002685; the TU Wien Doctoral College TrustACPS; the FWF SpyCoDe SFB projects F8504; the WWTF Grant ForSmart 10.47379/ICT22007; the and the Amazon Research Award 2023 QuAT; the bwHPC of the state of Baden-Württemberg; the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG; and a gift from Intel Corporation. The first author's Master thesis [10] conducted at the University of Liège under the supervision of Pascal Fontaine laid the foundations of this work. We thank him for his insights.

1 Introduction

In the past few years, chronological backtracking in CDCL-based SAT solving attracted renewed interest as it implements less aggressive procedures when backtracking upon conflicts, particularly for undoing literal assignments stored in the assignment stack. Chronological backtracking has been proven sound and complete, while also empirically improving performance on SAT competition problems [15, 17, 18].

Without chronological backtracking in SAT solving, the truth value of each literal is set as early as possible in the solving process. With chronological backtracking, there are, however, missed lower implications (MLI), i.e., clauses that could have set a literal at a lower SAT decision level. As a remedy to MLI, IntelSAT [17] and CaDiCaL-1.9.4 [19] fix the level of the assignments. Modifying levels impacts solving performance and significantly clutters the code; for example, reimplication techniques for detecting MLI have been removed in CaDiCaL-1.9.5 [4] due to the increased code complexity.

Invariant properties for CDCL algorithms (Section 3)	
Invariant 1–*Weak watched literals*:	No conflict is missed.
Invariants on implications, native for NCB and WCB (Section 3.1)	
Invariant 2–*Implied literals*:	Literals are decisions or implied by a clause C that is made unit by the partial assignment.
Invariant 3–*Topological order*:	The partial SAT assignment follows a topological order of the implication graph.
Strong invariant, non-trivial for CDCL with CB and native in NCB (Section 3.1)	
Invariant 4–*Strong watched literals*:	No implication nor conflict can be missed.

Figure 1 Invariant properties for CDCL-based SAT solving and maintained by the different chronological backtracking (CB) strategies, particularly by non-chronological backtracking (NCB) and weak chronological backtracking (WCB).

In this paper, we introduce a *lazy reimplication procedure for resolving missed lower implications in chronological backtracking, while also ensuring efficiency in SAT solving.* Doing so, in Figure 1, we state the invariant properties to be maintained during CDCL and highlight differences between relevant backtracking approaches in SAT solving. In particular, we consider and adjust variants of *non-chronological backtracking* (NCB) [21] and strong chronological backtracking (SCB) [17]. A formal presentation of these invariants and backtracking variants is given in Section 3. Using the invariants of Figure 1, in Section 4 we introduce a lazy reimplication procedure to handle missed lower implications, with a particular focus on handling unit implications after backtracking. We also adjust and enhance the first unique implication point (UIP) algorithm [16] with the knowledge of missed lower implications. Our approach is sound (Section 5). We implemented our work in the new solver NapSAT [20] and present our empirical findings in Section 6. To demonstrate the flexibility of our lazy reimplication techniques, we also implemented the algorithms of Section 4 in CaDiCaL [5] and Glucose [1], and provide empirical comparisons using these solvers.

Related work. Within CDCL, the truth values of literals are assigned by guessing (deciding) and propagating them in a trail until a conflict is found. Upon conflict analysis, the trail is adapted by backtracking, i.e. revoking some assignments and swapping the truth value of one variable, called the unique implication point (UIP). The standard approach [21] is to fix the conflict as early as possible with *non-chronological backtracking* (NCB) and all assignments between the current point and the point where the UIP is set are deleted.

A different backtracking approach comes with *chronological backtracking* (CB) [15, 18].

Here, a less aggressive backtracking scheme is used and some propagations and decisions are kept. Chronological backtracking may backjump at any level between the UIP and the UIP falsification point minus one. As a result, chronological backtracking resets a smaller part of the trail, but it may miss propagations that could have been done earlier if the learned clause was known beforehand. In this paper, we refer by *weak chronological backtracking* (WCB) to the CDCL algorithms that use chronological backtracking mechanism and which do not detect every propagation as early as possible (see Section 3.2)

For recovering such missed propagations, we define *strong chronological backtracking* (SCB). In particular, Nadel [17] introduced a reimplication procedure that eagerly re-assigns literals detected as missed lower implications to their lowest possible level. We refer to this SCB technique as *eager strong chronological backtracking* (ESCB). Our work introduces a new SCB method, *lazy strong chronological backtracking* (LSCB). Unlike ESCB, within LSCB we reimply missed implications on demand. As such, our work is stronger than WCB, as WCB does not perform reimplications at all. In addition, our technique is shown to be easier and more flexible to implement than ESCB or WCB (Section 6).

Our contributions. This paper brings the following contributions to chronological backtracking in CDCL-based proof search.

1. We formalize invariant properties that need to be maintained during SAT solving with chronological backtracking (Section 3). Our invariants incorporate and reason over different backtracking strategies.
2. We introduce *lazy strong chronological backtracking* (LSCB) for on-demand reimplication of (conflict) (Sec. 4) and prove soundness of our approach (Section 5).
3. We implement our work in the new NapSAT [11, 20] solver (Section 6). To showcase the flexibility and efficiency of our approach, we integrate LSCB into CaDiCaL [5] and Glucose [1], and provide experimental comparisons using these solvers.

2 Preliminaries

We assume familiarity with propositional logic and CDCL [6], and use the standard logical connectives $\neg$, $\wedge$, and $\vee$. A finite set of elements (e.g. literals) is called *conjunctive* (respectively, *disjunctive*) to indicate that the set is the conjunction (respectively, disjunction) of its elements. An *ordered set* is a set $\mathcal{S}$ which defines a bijective function $p_{\mathcal{S}}$ from elements of $\mathcal{S}$ to naturals, such that $p_{\mathcal{S}}(e)$ is the position of the element e in the ordered set $\mathcal{S}$. We consider the first element of $\mathcal{S}$ to have the position 0. Ordered sets are stable under the removal of elements; that is, for the ordered sets $\mathcal{S}, \mathcal{T}, \mathcal{U}$ with $\mathcal{S} = \mathcal{T} \setminus \mathcal{U}$, we have $\forall e, e' \in \mathcal{S}.\ p_{\mathcal{S}}(e) < p_{\mathcal{S}}(e') \Leftrightarrow p_{\mathcal{T}}(e) < p_{\mathcal{T}}(e')$. We denote by $\cdot$ set concatenation; for simplicity, we use $\cdot$ to also denote appending a sequence with an element. We write $\mathcal{S}[a : b]$ to select the ordered elements e in $\mathcal{S}$ with positions $a \leq p_{\mathcal{S}}(e) \leq b$.

We denote by $\mathcal{V}$ a countable set of Boolean variables v. We consider propositional formulas F in conjunctive normal form (CNF), represented by a conjunctive set of clauses $\{C_1, C_2, \ldots, C_n\}$ over $\mathcal{V}$. Clauses are disjunctive sets of literals $C = \{c_1, c_2, \ldots, c_m\}$, where a literal c_i is either a Boolean variable v or a negation $\neg v$ of a variable v.

To efficiently identify unit propagations, SAT solvers track two literals per clause in the two-watched literal scheme [16]. We denote the watched literals of a clause C by c_1 and c_2, and write $\mathrm{WL}(c_1)$ and $\mathrm{WL}(c_2)$ for the watched lists of c_1 and c_2. We have $C \in \mathrm{WL}(c_1) \cap \mathrm{WL}(c_2)$.

During SAT solving, solvers keep track of a *partial assignment*, also called *trail* and denoted as the conjunctive ordered set $\pi = \tau \cdot \omega$, which is split into two parts: (i) τ is the set of literals that were already propagated and do not need to be inspected anymore (by checking

the watch lists); (ii) ω is the *propagation queue* containing literals that were implied and waiting to be propagated. The partial assignment π contains the set $\pi^d \subseteq \pi$ of decision literals. Decisions literals in π^d are arbitrarily chosen literals when unit propagation cannot be further used and the truth value of a (decision) literal needs to be picked and assigned. We call *unit*, a clause C containing exactly one unassigned literal ℓ and whose other literals are falsified, i.e., $\exists \ell \in C.\, C \setminus \{\ell\}, \pi \models \bot \wedge |\ell| \notin |\pi|$. For conflict analysis, the propagation reasons of literals are analyzed. Therefore, SAT solvers use a ρ function that maps literals to clauses such that $\rho(\ell)$ captures the reason for propagating ℓ. The reason for propagating ℓ is the clause C that implied ℓ under assumption π, that is, $[\ell \in \pi] \wedge [\ell \in \rho(\ell)] \wedge [\rho(\ell) \setminus \{\ell\} \wedge \pi \models \bot]$. Following [15], we use δ to represent the (decision) *level of* ℓ, i.e., the level when a truth assignment to ℓ was made. Formally, if ℓ is a decision literal, then the level $\delta(\ell) = \delta(\neg\ell)$ of ℓ is the number of decisions preceding and including ℓ, that is $\delta(\ell) = |\pi[0 : p_\pi(\ell)] \cap \pi^d|$. Further, for literals ℓ implied by $\rho(\ell)$, we have $\delta(\ell) = \max_{\ell' \in \rho(\ell) \setminus \{\ell\}} \delta(\ell')$. Finally, $\delta(\ell) = \infty$ for unassigned literals ℓ. The definition of δ is extended to clauses and trails, with $\delta(C) = \max_{\ell \in C} \delta(\ell)$; similarly for $\delta(\pi)$. The level of the empty set is $\delta(\emptyset) = 0$. We write $\delta[\ell \leftarrow d]$ to denote that the level of ℓ is updated to d. We reserve the special symbol $\blacksquare$ to denote *undefined clauses* during SAT solving, with $\delta(\blacksquare) = \infty$.

In standard CDCL with non-chronological backtracking (NCB) [21], level δ stores the number of decisions that appear before in the trail, and is always the lowest level possible. In CDCL with chronological backtracking, the history of propagations and conflicts may, however, lead to *missed lower implications* (MLI), where a MLI captures the fact that a clause C is satisfied by a unique literal ℓ at a level *strictly higher* than $\delta(C \setminus \{\ell\})$. Therefore, in a MLI, the literal ℓ could have been propagated at a lower level in the trail.

▶ **Example 1** (Missed Lower Implications – MLI). Figure 2 shows a clause set $\{C_1, \ldots, C_7\}$ and a trail $\pi = \tau \cdot \omega$ during CDCL solving with chronological backtracking. The trail diagram displays, from left to right, the order in which literals are decided and propagated, as well as the location of the propagation head (symbolized by the dashed line). The propagation level is symbolized by the height of the step. As a visual aid, literals are colored green, red or black, symbolizing respectively satisfied, falsified, and unassigned literals. The watched literals are the first two in the clause.

In the example, the clause set $F_0 = \{C_1, \ldots, C_6\}$ was given as input. The decisions v_1, v_2 and v_3 were made, then the solver found a conflict in C_2 after implying v_4 with reason $\rho(v_4) = C_1$. The clause $C_7 = \neg v_3@3 \vee \neg v_1@1$ is learned and the solver backtracks to level $\delta(C_7) - 1 = 2$, continuing its propagations until it reaches the assignment shown on Figure 2. Figure 2 shows that C_4 is a MLI. Indeed, v_2 is satisfied at level 2, while all other literals are falsified at level 1. After backtracking to level 1, the implication of v_2 by C_4 is missed since (i) $\neg v_3$ was already propagated, and (ii) C_4 is watched by v_3 and v_2.

3 Invariant Properties on CDCL Variants

To properly handle MLI similar to Example 1, in this section we revisit and formalize our invariants from Figure 1, expressing properties that need to be maintained in (variants of) CDCL with chronological backtracking.

The crux of our invariant properties is captured by watched literals [16]. They reduce the number of clauses to be checked when propagating a literal. Invariant 1 therefore expresses that, as long as CDCL does not falsify one of the watched literals c_1, c_2 of a clause C, the clause C is not a conflict. Therefore, when propagating a watched literal c_i during CDCL, only checking the clauses watched by $\neg c_i$ is sufficient to not miss any conflict.

$$
\begin{aligned}
C_1 &= \neg v_3@1 \vee \quad v_4 \\
C_2 &= \neg v_3@1 \vee \neg v_4 \quad \vee \neg v_1@1 \\
C_3 &= \quad v_5@1 \vee \quad v_3@1 \\
C_4 &= \quad v_2@2 \vee \quad v_3@1 \vee \neg v_5@1 \\
C_5 &= \quad v_6@2 \vee \neg v_5@1 \vee \quad v_3@1 \\
C_6 &= \neg v_6@2 \vee \neg v_2@2 \vee \neg v_5@2 \\
C_7 &= \neg v_3@1 \vee \neg v_1@1
\end{aligned}
$$

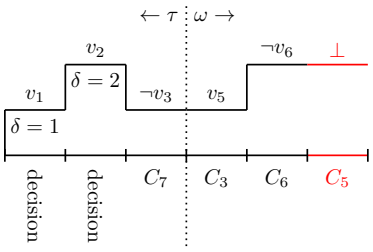


Figure 2 C_4 is a MLI while C_5 would be a MLI if v_5 was set to true. We use the notation $v@1$ to indicate that literal v is on level 1.

▶ **Invariant 1** (Weak watched literals). *Let $\pi = \tau \cdot \omega$ be the current trail. For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow \neg c_2 \notin \tau$.*

Invariant 1 ensures that conflicts are not missed during CDCL. Indeed, if there is a conflicting clause C, the conflict is found after propagating all literals of C. After propagation, no more literal has to be propagated, so $\pi = \tau$. A conflicting clause C thus violates Invariant 1, and hence the conflict of C is captured during CDCL.

3.1 CDCL Invariants on Implications

We next ensure the soundness of unit implications. Invariant 2 expresses that literals are either decisions or implied by a sound implication. Note that an implication can be performed if there is only one unassigned literal that can satisfy a clause C; hence, C is a unit clause. In addition to ensuring that the solver infers correct literals, Invariant 2 is also relevant for conflict analysis (see proof of Theorem 13).

▶ **Invariant 2** (Implied literals). *If a literal ℓ is in the trail π, then ℓ is either a decision literal or ℓ is implied by π and its reason $\rho(\ell)$. That is,*

$$\forall \ell \in \pi.\ \ell \in \pi^d \ \vee\ [\ell \in \rho(\ell) \ \wedge\ [\rho(\ell) \setminus \{\ell\} \wedge \pi] \vDash \bot]\,.$$

To perform conflict analysis with the first unique implication point (UIP) [16], CDCL solving assumes that literals are organized in a topological sort of the implication graph.

▶ **Invariant 3** (Topological order). *Trail π is a topological order of the implication graph:*

$$\forall \ell \in \pi.\ \forall \ell' \in \rho(\ell).\ p_\pi(\neg \ell') \leq p_\pi(\ell),$$

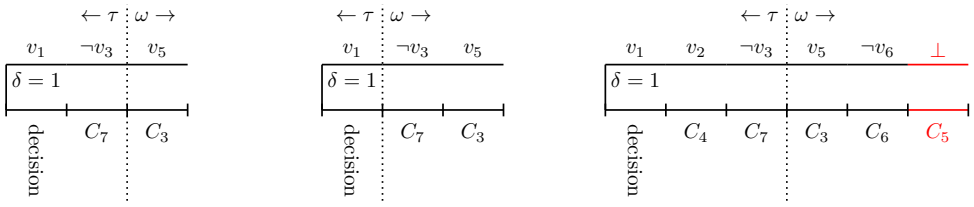
where $p_\pi(\ell)$ and $p_\pi(\neg \ell')$ are respectively the positions of ℓ and $\neg \ell'$ in π.

Invariant 3 holds by construction in CDCL with non-chronological backtracking (NCB) and chronological backtracking without reimplication. However, Invariant 3 is crucial in any setting of reimplying literals.

Finally, we impose that Boolean constraint propagation (BCP) in CDCL does not miss unit implication during proof search. Invariant 4 therefore formalizes that CDCL cannot have one propagated falsified watched literal without the clause being satisfied.

▶ **Invariant 4** (Strong watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow c_2 \in \pi$.*

Invariant 4 strengthens Invariant 1. When a conflicting clause C is detected while propagating ℓ, the literal ℓ cannot be added to τ without violating Invariant 4. As such, by imposing Invariant 4, the conflict of C is resolved and the trail is adapted.



(a) Weak CB **(b)** Restoration SCB **(c)** Eager SCB

Figure 3 Different CB ways of handling the missed lower implications of Figure 2.

3.2 Chronological Backtracking

Invariant 4 holds for CDCL with NCB, since the trail contains monotonically increasing decision levels. Therefore, within NCB, literals are unassigned in the reverse order of propagation. In particular, if a literal ℓ is satisfied in a clause C when propagating another literal ℓ', the literal ℓ remains satisfied at least until ℓ' is backtracked.

Weak chronological backtracking (WCB). When considering (variants of) chronological backtracking in CDCL, Invariant 4 becomes critical, as detailed next. The core idea is to save parts of the trail without repropagating unlike [12].

▶ **Example 5.** Let us revisit the example of Figure 2. Figure 3a shows the trail after backtracking to level 1. Literal v_3 is already propagated ($v_3 \in \tau$), and C_4 is still watched by v_2 and v_3. Therefore, the implication of v_2 is missed, even though Invariant 1 is not violated.

To circumvent the problem of missing implications similar to Example 5, we distinguish a *weak chronological backtracking (WCB)* variant of CDCL with chronological backtracking. Within WCB, Invariant 4 is not necessarily satisfied, as unit implications at lower levels can be missed. To recover Invariant 4 in variants of CDCL with CB, we adjust and label two existing solutions in SAT solving: (i) *restoration* [18], for repairing the trail p after backtracking; and (ii) *prophylaxis* [17], for forcing literals at the lowest possible level.

Restoration. We call *restoration* the approach in which the trail π is repaired by pushing back the propagation head when propagating [18]. Out-of-order literals are repropagated whenever they are moved in the trail during backtracking. For example, in Figure 3b, v_3 was the first literal that changed position during backtracking, so this is where the propagation head is set. When backtracking to level δ, the propagation head is set to $p_\pi(\pi^d[\delta])$. When v_3 is repropagated, v_2 is reimplied. We, therefore, restore Invariant 4 by repropagating the out-of-order literals. We call this approach *restoring strong chronological backtracking* (RSCB), allowing to restore the trail π by propagating more. It is also used in CaDiCaL [5].

Prophylaxis. We name *prophylaxis*[1] the approach in which missed lower implications are prevented from becoming missed unit implications [17]. Prophylaxis uses an eager reimplication procedure and imposes the validity of a compatibility invariant; we formalize this property in Invariant 6. That is, when a clause C is detected to be a missed lower implication of ℓ, then ℓ is reimplied at level $\delta(C \setminus \{\ell\})$ and its reason for propagation is updated. Prophylaxis thus enforces our *backtrack compatible* Invariant 6 by ensuring that no clause can become unit after backtracking. Furthermore, Invariant 6 guarantees that literals are always propagated at the lowest level, and conflicts are detected at the lowest level.

[1] "Prophylaxis" is a chess term referring to a move that deals with a threat before it becomes a problem.

Table 1 CB variants in CDCL, together with their invariant properties.

	Inv. 1	Inv. 4	Inv. 6	Inv. 9	Solvers
NCB	✓	✓	✓	✓	Most CDCL solvers
WCB	✓	✗	✗	✗	Our work – NapSAT
RSCB	✓	✓	✗	✗	Maple_LCM_Dist [18], CaDiCaL
ESCB	✓	✓	✓	✓	IntelSAT and CaDiCaL 1.9.4
LSCB	✓	✓	✗	✓	Our work – NapSAT, now in CaDiCaL

▶ **Invariant 6** (Backward compatible watched literals). *For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow [c_2 \in \pi \wedge \delta(c_2) \leq \delta(c_1)]$.*

▶ **Example 7.** Figure 3c shows the trail after v_2 is reimplied. In this case, v_2 was a decision, and $\neg v_6$ has to be reimplied to level 1 as well. All literals are propagated at the lowest possible level. Thus, using Invariant 6, the conflict C_5 is properly detected at level 1, instead of level 2. Figure 3c also shows that the trail π no longer follows a topological order of the implication graph. These issues have to be addressed.

Based on Invariant 6, *eager strong chronological backtracking (ESCB)* is used in [17, 19], yielding a CDCL method with chronological backtracking that satisfies Invariant 6 by eager reimplication of missed lower implications. In Table 1 we summarize backtracking strategies in CDCL, also listing our solution in this respect: *lazy reimplication in strong chronological backtracking (LSCB)*. Our LSCB approach maintains Invariant 1 and Invariant 4, while weakening Invariant 6 via Invariant 9, as described next in Section 4 and implemented in Algorithm 1.

4 Adapting CDCL with Lazy Reimplications

Embedding the prophylaxis approach of Section 3.2 in existing CDCL data structures is highly non-trivial, due to the rigid and entangled data structures [17, 19], see e.g. [10]. In addition, reimplying literals [17, 19] changes the implication graph, and hence the trail π is no longer a topological sort of the implications; as such, Invariant 3 must be restored.

While the restoration approach of Section 3.2 offers a practically simpler solution, restoration might require the re-propagation of a large part of π and thus can be computationally very expensive. For example, while in Figure 3b only one literal had to be re-propagated, re-propagation could be applied on an arbitrary number of literals.

Our solution: Lazy reimplication in CDCL. To overcome inefficiencies of restoration and pure prophylaxis, our work advocates a *lazy reimplication* technique for CDCL with strong chronological backtracking. To ensure Invariant 4, we reimply literals after backtracking. That is, *we detect missed lower implications eagerly but reimply them lazily.*

Our lazy reimplication approach for CDCL-based solving is summarized in Algorithm 1. In what follows, we describe the key ingredients of Algorithm 1 and revise the CDCL invariants of Section 3, adjusted to Algorithm 1. To this end, we introduce a lazy reimplication vector λ to store missed lower implications, where λ is a function from literals to clauses. Intuitively, the lazy reimplication vector λ stores the lowest

detected missed lower implication for each literal ℓ. The clause $\lambda(\ell) \neq \blacksquare$ is an alternative reason that would propagate ℓ in trail π, lower than the reason $\rho(\ell)$. Initially, no clause is assigned, and $\forall \ell.\ \lambda(\ell) = \blacksquare$ (that is, the undefined clause). Invariant 8 is asserted to hold during proof search.

▶ **Invariant 8** (Lazy reimplication). *If the lazy reimplication reason $\lambda(\ell)$ of literal ℓ is defined, then the clause $\lambda(\ell)$ is a missed lower implication of ℓ. That is,*

$$\begin{aligned}\lambda(\ell) \neq \blacksquare \;\Rightarrow\quad & \ell \in \pi \wedge\ \ell \in \lambda(\ell) \\ & \wedge \big(\lambda(\ell) \setminus \{\ell\} \wedge \pi\big) \vDash \bot \\ & \wedge\ \delta(\lambda(\ell) \setminus \{\ell\}) < \delta(\ell)\end{aligned}$$

When a missed lower implication for ℓ is detected, then ℓ is not reimplied directly. Rather, we store the MLI in λ until ℓ is unassigned during backtracking. For example, if a literal ℓ is assigned at level 3 and a missed lower implication C for ℓ is detected with $\delta(C \setminus \{\ell\}) = 1$, then backtracking to level 2 will reassign ℓ from level 3 to level 1 by C.

Using our lazy reimplication vector λ, we weaken Invariant 6 into Invariant 9 such that, during backtracking, we identify missed lower implications without requiring the re-propagation of out-of-order literals.

▶ **Invariant 9** (Lazy backtrack compatible watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$, if one watched literal c_1 of C is falsified by τ, then the other c_2 must be satisfied at a lower level, or a missed lower implication lower than c_1 is set in λ.*

$$\neg c_1 \in \tau \Rightarrow \Big(c_2 \in \pi \wedge \big(\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)\big)\Big)$$

Lazy reimplication for strong chronological backtracking – LSCB. Guided by the reimplication and backtracking properties of Invariant 8 and Invariant 9, Algorithm 1 shows our LSCB algorithm for CDCL with chronological backtracking, as a slight refactoring of weak chronological backtracking (WCB). In the following algorithms, particularities of LSCB are highlighted in blue.

An important detail should be noted upon Algorithm 1: in our abstract representation, it is not explicitly checked whether the learned clause D is different from the conflicting clause C; such a check, however, should be performed when implementing Algorithm 1. Indeed, as pointed out in RSCB [15], it is possible that a conflicting clause C does not require conflict analysis since C might already be a UIP. However, if the highest literal ℓ in C is a MLI, then the clause might be conflicting again after backtracking (see Algorithm 4).

▶ **Example 10.** Consider the example of Figure 2. Here, the conflicting clause C_5 only has one literal at the highest level, and, as such, it qualifies as a UIP. Therefore no conflict analysis is required, we only backtrack to level 1, and then C_5 implies v_6 at level 1. However, if $\neg v_6$ was a missed lower implication, then backtracking to level 1 would reimply $\neg v_6$, with C_5 conflicting again; this time, however, C_5 would require conflict analysis.

Propagation in LSCB. When falsifying a watched literal, Algorithm 1 might need to find a replacement candidate to become the new watched literal (line 7 of Algorithm 2). We define the property of the candidate literal with information about its level as below.

▶ **Definition 11** (Candidate literal). *Let clause C be watched by the literals c_1 and c_2. with $\neg c_1 \in \omega$. Then, SearchReplacement(C, c_1, c_2) from Algorithm 2 returns a* candidate literal *r for which one of the following holds:*

- *Invariant 6 is satisfied on C after $\neg c_1$ is added to τ, i.e.* $\neg r \in (\tau \cdot \neg c_1) \Rightarrow c_2 \in \pi \wedge \delta(c_2) \leq \delta(r)$;
- *C is conflicting, propagating, or a MLI for c_2. As such, $C \setminus \{c_2\}$ is unsatisfiable with the current assignment, and r is at the highest decision level in $C \setminus \{c_2\}$, that is* $\big(C \setminus \{c_2\} \wedge \pi\big) \vDash \bot \quad \wedge \quad \delta(r) = \delta(C \setminus \{c_2\})$

Algorithm 1 Lazy Reimplication in CDCL with CB

```
1: π = τ = ω = π^d = ∅
2: ∀ℓ. δ(ℓ) = ∞
3: ∀ℓ. WL(ℓ) = ∅
4: ∀ℓ. ρ(ℓ) = λ(ℓ) = ■
5: procedure CDCL(F)
6:     for C ∈ F do                                  ▷ Fill the watcher lists
7:         c1, c2 ← two literals in C
8:         WL ← WL[c1 ← WL(c1) ∪ {C}][c2 ← WL(c2) ∪ {C}]
9:     while ⊤ do
10:        C ← BCP()                                 ▷ Algorithm 2
11:        if C = ⊤ then
12:            if |π| = |V| then                     ▷ All variables are assigned
13:                return SAT
14:            ℓ ← DECIDE()
15:            ω ← ω · ℓ, π^d ← π^d · ℓ, δ ← δ[ℓ ← |π^d|]
16:            continue
17:        D ← ANALYZE(C)                            ▷ Algorithm 4
18:        if δ(D) = 0 then
19:            return UNSAT
20:        d ← any level between δ(D) − 1 and the second highest level of D
21:        BACKTRACK(d)                              ▷ Algorithm 3
22:        ℓ ← the unassigned literals in D
23:        c2 ← the second highest literal in D
24:        ω ← ω · ℓ, δ ← δ[ℓ ← δ(C \ {ℓ})], ρ ← ρ[ℓ ← D]
25:        F ← F ∪ {D}                               ▷ Does nothing if C = D
26:        WL ← WL[ℓ ← WL(ℓ) ∪ {D}][c2 ← WL(c2) ∪ {D}]
```

Concretely, the SEARCHREPLACEMENT(C, c_1, c_2) procedure iterates over literals of $C \setminus \{c_2\}$ and stops when it finds a literal r that would satisfy Invariant 6 if c_1 was replaced by r. In case of failure, it returns the highest literal in $C \setminus \{c_2\}$. The knowledge of the highest literal in $C \setminus \{c_2\}$ is enough to determine the nature and level of the clause.

Algorithm 2 shows our Boolean constraint propagation (BCP) algorithm adapted to support LSCB. As opposed to standard BCP, Algorithm 2 does not stop when the other watched literal is satisfied. We need the extra guarantee that either c_2 is implied at a level lower than c_1, or it is registered as a MLI before skipping the clause. Further, when a non-falsified replacement literal cannot be found, Algorithm 2 still changes the watched literal. While this is not always strictly necessary (for example, in conflicts), systematically swapping the highest literal allows checking the level of the clause in constant time and provides cheap useful properties to the clause.

Backtracking in LSCB. When backtracking, our LSCB approach has the information of whether a clause C violates Invariant 4. Therefore, Algorithm 3 can directly imply those missed lower implications (line 15 of Algorithm 3).

The order in which literals are reimplied in Algorithm 3 is not important, as shown later in Theorem 17. It is, however, unclear whether a specific order would impact performance in problems where the stability of literal position in the trail is important. In such cases, ordering the reimplications in increasing levels might be beneficial.

Algorithm 2 Boolean Constraint Propagation in LSCB

```
1: procedure PropagateLiteral(ℓ)
2:     c1 ← ¬ℓ
3:     for C ∈ WL[c1] do
4:         c2 ← the other watched literal in C
5:         if c2 ∈ π∧ [δ(c2) ≤ δ(c1) ∨ δ(λ(c2) \ {c2}) ≤ δ(c1)] then
6:             continue
7:         r ← SearchReplacement(C, c1, c2)
8:         WL ← WL[c1 ← WL(c1) \ {C}][r ← WL(r) ∪ {C}]
9:         if ¬r ∉ π then
10:            continue
11:        if ¬c2 ∈ π then                          ▷ Conflict
12:            return C
13:        if c2 ∈ π then
14:            if δ(c2) > δ(r) ∧ δ(λ(c2) \ {c2}) > δ(r) then
15:                λ ← λ[c2 ← C]                    ▷ New or improved MLI
16:            continue
17:        ω ← ω · c2, ρ ← ρ[c2 ← C], δ ← δ[c2 ← δ(r)]
18:    return ⊤
1: procedure BCP
2:     while ω ≠ ∅ do
3:         ℓ ← First(ω)
4:         C ← PropagateLiteral(ℓ)
5:         if C ≠ ⊤ then
6:             return C
7:         ω ← ω \ {ℓ}, τ ← τ · ℓ
8:     return ⊤
```

Conflict analysis with MLI. As opposed to traditional backtracking, Algorithm 1 does not guarantee that, once it backtracks to a level lower than the level of the learned clause D, the clause D will be propagating. Indeed, let the falsified learned clause $D = \{c_1, c_2, \ldots, c_m\}$ with c_1 a unique literal at level $\delta(D)$. If we backtrack to level $\delta(D)-1$, c_1 might be reimplied at a lower level, and D would still be a conflict. In response to this, we propose the following two solutions:

(Analyze-1) we analyze the conflict and backtrack again until we get a unit clause;

(Analyze-2) we perform conflict analysis with the knowledge of missed lower implications.

In Algorithm 4 we chose option 2. Option 1 will generate the same clause in the end, but might create some unnecessary ones in the process. We empirically check our intuition in Section 6 and demonstrate that option 2 indeed works better. We refer to $D \otimes_\ell C'$ as the result of binary resolution applied to the clauses C and D over the literal ℓ.

In Algorithm 4, when possible, we use the lazy reimplication reason $\lambda(\ell)$ instead of the real reason $\rho(\ell)$ during conflict analysis. The lazy reason $\lambda(\ell)$ is guaranteed to introduce literals at a level lower than $\delta(C)$, making it converge to a UIP faster. Once a UIP is obtained, Algorithm 4 does not stop if there exists a missed lower implication for the last literal at the conflict level. Furthermore, we adapted the learnt clause minimization approach [22], adjusted to Algorithm 4 so that both reasons are checked if the literal can be removed.

Algorithm 3 Backtracking and Reimplication.

```
1: procedure BACKTRACK(d)
2:     Λ ← ∅                                   ▷ Λ is the set that will be reimplied
3:     π = τ · ω
4:     for ℓ ∈ π do
5:         if δ(ℓ) > d then
6:             if δ(λ(ℓ) \ {ℓ}) ≤ d then
7:                 Λ ← Λ ∪ {λ(ℓ)}              ▷ Store the MLI for later
8:             π ← π \ {ℓ}                     ▷ Unassign ℓ
9:             δ ← δ[ℓ ← ∞], ρ ← ρ[ℓ ← ■]
10:            λ ← λ[ℓ ← ■]                    ▷ λ(ℓ) is either used, or no longer valid
11:    π^d ← π ∩ π^d                           ▷ Remove the unassigned literals
12:    τ ← π ∩ τ
13:    ω ← π \ τ
14:    for C ∈ Λ do                            ▷ Reimplying the MLI
15:        ℓ ← the unassigned literal in C
16:        ω ← ω · ℓ, ρ ← ρ[ℓ ← C], δ ← δ[ℓ ← δ(C \ {ℓ})]
```

$C_1 = \neg v_2@1 \vee \ v_1@1$
$C_2 = \neg v_5@2 \vee \ v_3@2 \vee \neg v_4@1$
$C_3 = \neg v_6@1 \vee \ v_2@1 \vee \neg v_4@1$
$C_4 = \ v_7@2 \vee \ v_5@2 \vee \ v_3@1$
$C_5 = \ v_5@2 \vee \neg v_7@2 \vee \ v_6@1$
$C_6 = \neg v_3@2 \vee \neg v_4@1 \vee \ v_2@1$
$C_7 = \ v_4@1 \vee \ v_2@1$

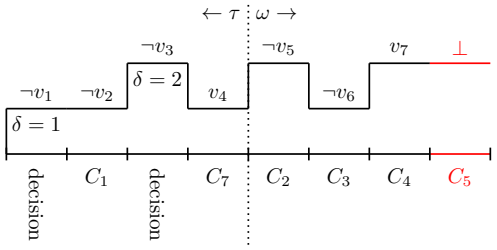


Figure 4 The clause $\neg v_3@2 \vee \neg v_4@1 \vee v_2@1$ is a missed lower implication in this example. v_2 and $\neg v_4$ are falsified at level 1, whereas $\neg v_3$ is only satisfied at level 2.

▶ **Example 12.** Figure 4 shows a conflict after Algorithm 2 detected a missed lower implication C_6. From Invariant 9, we have $\lambda(\neg v_3) = C_6$. Algorithm 1 will then trigger Algorithm 4 to analyse the conflict on C_5. During conflict analysis with Algorithm 4, we start from the conflicting clause $D = \neg v_7@2 \vee v_5@2 \vee v_6@1$ and apply the resolution $D \leftarrow D \otimes_{\neg v_7} C_4$ to obtain $D = v_5@2 \vee v_3@2 \vee v_6@1$. We once again apply resolution and have $D \leftarrow D \otimes_{v_5} C_2$, yielding the clause $D = v_3@2 \vee v_6@1 \vee \neg v_4@1$. As this D is a UIP, most CDCL approaches would stop conflict analysis here. However, in our LSCB approach we know that v_3 can be reimplied at level 1. Therefore, after backtracking to level 1 and reimplying $\neg v_3$ with Algorithm 3, the clause D would still be conflicting and conflict analysis would need to be triggered again. Instead we apply the resolution $D \leftarrow D \otimes_{v_3} C_6$ to get a clause at level 1, namely clause $D = v_6@1 \vee \neg v_4@1 \vee v_2@1$. We then continue until the procedure at level 1 and obtain the final clause $D = v_2@1$.

Algorithm 4 Conflict Analysis.

```
1: procedure ANALYZE(C)
2:     π ← τ · ω                                  ▷ Array version of the trail.
3:     D ← C                                      ▷ Current learned clause.
4:     n ← |{ℓ : ℓ ∈ D ∧ δ(ℓ) = δ(D)}|             ▷ Number of literals at the highest level.
5:     while ⊤ do
6:         ℓ ← the last literal in π falsified in D at level δ(D)
7:         if n = 1 ∧ λ(ℓ) = ■ then
8:             return D
9:         C′ ← ρ(ℓ)
10:        if λ(ℓ) ≠ ■ then
11:            C′ ← λ(ℓ)
12:        D ← D ⊗_ℓ C′
13:        n ← |{ℓ : ℓ ∈ D ∧ δ(ℓ) = δ(D)}|
```

5 Soundness of Lazy Reimplication

This section proves the soundness and completeness[2] of our LSCB approach given in Algorithm 1. We note that Algorithm 1 implements strong chronological backtracking and does not miss any implication; as such, Invariant 4 holds.

▶ **Theorem 13** (Soundness of conflict analysis)**.** *Let $C \in F$ be a conflicting clause with the partial assignment π. Then, conflict analysis in* ANALYZE(C) *from Algorithm 1 returns a conflicting clause that is implied by the clause set.*

Proof. The starting clause $D \leftarrow C$ is conflicting. At each step, D is resolved with a clause C' such that $C' = \rho(\ell)$ or $C' = \lambda(\ell)$, with $\ell \in C'$ and $\neg\ell \in D$. From the definition of ρ and λ, we have $\big(C' \setminus \{\ell\} \wedge \pi\big) \vDash \bot$. Therefore, the clause $D \leftarrow D \otimes_{\neg\ell} C'$ is conflicting, and implied by F, since $C' \in F$. ◀

▶ **Theorem 14** (No missed unit implication)**.** *Algorithm 1 satisfies Invariant 9. As such, our LSCB method in Algorithm 1 does not miss unit implications.*

Proof. We prove that Invariant 9 holds for each building block of Algorithm 1.

BCP. Invariant 9 trivially holds at the starting state, where $\pi = \emptyset$. Further, during the propagation of one literal, Algorithm 2 ensures that for each clause $C \in F$ watched by c_1 and c_2, the following Hoare triple holds

$$\{P\}\text{PROPAGATELITERALL}(\ell)\{Q\},$$

where

$$P \equiv \neg c_1 \in \tau \Rightarrow [c_2 \in \pi \wedge [\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)]]$$
$$Q \equiv \neg c_1 \in (\tau \cdot \ell) \Rightarrow [c_2 \in \pi \wedge [\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)]]$$

By structural induction over the statements of Algorithm 2, we conclude that Invariant 9 is maintained by BCP.

[2] with details also in the code base of NAPSAT

Backtracking. During backtracking in Algorithm 3, each literal c_i is inspected: c_i is either removed from the trail π or c_i is kept. Violating Invariant 9 means that a literal c_2 from the trail is removed such that $\neg c_1 \in \tau \wedge c_2 \notin \pi$ for some clause $C = \{c_1, c_2, \ldots, c_m\}$ (since the levels are not altered). However, this case is rectified, since either $\delta(c_1) \leq \delta(c_2)$ (and then $\neg c_1$ would be removed from τ), or $\delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)$ (and then c_2 would be reimplied at level $\delta(\lambda(c_2) \setminus \{c_2\})$ and $c_2 \in \pi \wedge \delta(c_2) \leq \delta(c_1)$ would be true), or $\delta(\lambda(c_2) \setminus \{c_2\}) > \delta(c_1)$ (and then $\neg c_1$ is also backtracked). As such, backtracking in Algorithm 3 preserves Invariant 9.

Analysis. Within conflict analysis in Algorithm 4, the state of CDCL is not modified, only read. Therefore, any invariant that held before Algorithm 4 also holds after Algorithm 4.

CDCL. We finally ensure that Invariant 9 is maintained by Algorithm 1 also during its decision step and while adding a clause to the formula F. First, deciding still preserves Invariant 9, since it merely adds a non-assigned literal to the propagation queue ω. Second, after backtracking in Algorithm 1, we know by construction that the learned clause will have a single literal ℓ that is unassigned. This literal ℓ is then implied at level $\delta(D \setminus \{\ell\})$, satisfying Invariant 9 since the second watched literal c_2 is falsified at level $\delta(D \setminus \{\ell\})$. ◀

▶ **Corollary 15** (No missed conflict/implication). *Our LSCB method from Algorithm 1 preserves the strong watched literal property of Invariant 4.*

Based on the results above, we conclude the soundness and completeness of LSCB.

▶ **Theorem 16** (LSCB soundness and completeness). *Lazy reimplication with strong chronological backtracking from Algorithm 1 is sound and complete.*

Proof. Theorem 13 implies that clauses added to the clause set are implied by F. By induction, if ϕ is the original CNF, then if $\phi \models F$ and $F \models C$, then $\phi \models F \cup \{C\}$. Furthermore, from Corollary 15 we conclude that Invariant 4 holds.

Algorithm 1 returns unsat iff there exists a conflict at level 0; that is, there exists a set of clauses $F' \subseteq F$ such that $F' \models \bot$. As $\phi \models F$, then $\phi \models \bot$, and thus ϕ is unsatisfiable. Otherwise, Algorithm 1 returns SAT if a model τ exists such that every variable has been assigned and propagated ($\pi = \tau$). Based on Invariant 4, no conflict is possible and $\phi \models \tau$. ◀

▶ **Theorem 17** (Topological order in LSCB). *The literals reimplied by the backtracking procedure of Algorithm 3 respect the topological order of the implication graph.*

Proof. The reimplied literals cannot depend on each other. Indeed, if they are reimplied, their implication level before backtracking was higher than d. Therefore, if a literal ℓ depends on a literal ℓ' in the implication graph, then $\delta(\ell) \geq \delta(\ell')$. If the missed lower implication $\lambda(\ell)$ has a level lower than d, then all literals in $\lambda(\ell) \setminus \{\ell\}$ are lower than d, and therefore were not backtracked. Therefore, since all literals are independent, they can be reimplied in an arbitrary order at the end of the trail, and still respect the topological order. ◀

6 Empirical Analysis

In this section, we discuss the implementation of Algorithm 1 in our new SAT solver NapSAT. We also integrated it in CaDiCaL and Glucose, and present our empirical results using NapSAT, CaDiCaL, and Glucose.

6.1 NapSAT for Lazy Reimplication in CDCL

We implemented our LSCB method from Algorithm 1 in the new SAT solver NapSAT. Our NapSAT tool is a CDCL solver using the watcher list scheme [16] with blocker literals [9]. NapSAT supports the backtracking variants of NCB, WCB, RSCB, and LSCB at runtime. In chronological backtracking, the backtracking scheme is purely chronological, that is, NapSAT always backtracks to one level before the conflict (unlike CaDiCaL). NapSAT uses the VSIDS decision heuristic [16] with the agility restart strategy [3] and root-level clause elimination [7]. NapSAT is available at `https://github.com/RobCoutel/NapSAT` and consists in a total of ∼5.800 loc, among which the core of the solver represents ∼1.500 loc.

Blocker literals in NapSAT. Blocker literals are useful to reduce the number of pointer dereferencing of the literal pointer [9]. If the blocker b is assigned at a level higher than the literal ℓ being falsified, then it might get backtracked before ℓ and a conflict might be missed. Invariant 9 can therefore be weakened, while still ensuring that no unit implication is missed.

▶ **Invariant 18** (Lazy backtrack compatible watched literals with blocker literals)**.** *For each clause $C \in F$ watched by the two distinct literals c_1, c_2 and with blocker b, we have*

$$\neg c_1 \in \tau \Rightarrow \Big(c_2 \in \pi \wedge \big(\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)\big)\Big) \vee \Big(b \in \pi \wedge \delta(b) \leq \delta(c_1)\Big)$$

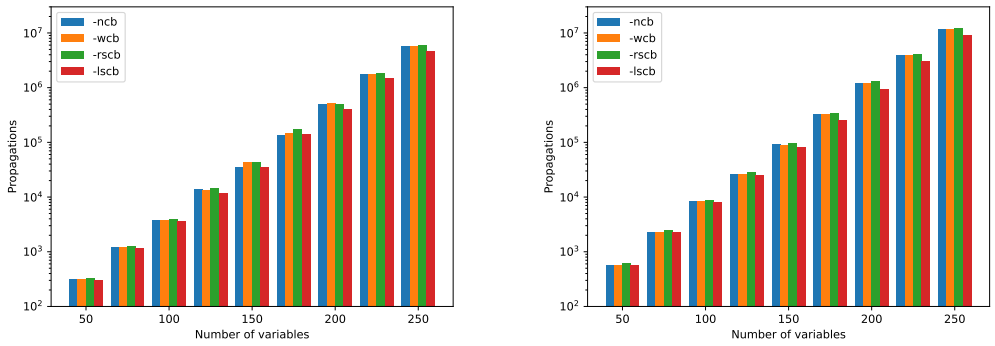
The eager update of blocking literals is in essence similar to strategies that aggressively update watched literals during BCP [14].

Experiments. Figure 5 shows the average total number of propagations of NapSAT on the 3-SAT uniform random problems from SATLIB [13]. Our LSCB method from Algorithm 1, indicated via `-lscb`, performs better than the other backtracking versions of NapSAT, both for satisfiable and unsatisfiable instances. Figure 6 shows more details. In particular, it shows the total number of propagations of each unsatisfiable problem with 250 variables. It shows that LSCB consistently has fewer propagations than NCB, WCB, and RSCB.

We acknowledge that the number of propagations alone is not always representative of the real performance of a SAT solver, since propagation in LSCB is slightly more expensive than in NCB or WCB. However, the number of propagations in NapSAT indicates the impact of missed lower implications. For example, Figure 5 shows that restoring the trail with RSCB might not be worth finding the missed lower implications in the random 3-SAT benchmarks; yet, reimplying literals lazily significantly reduces the total number of propagations.

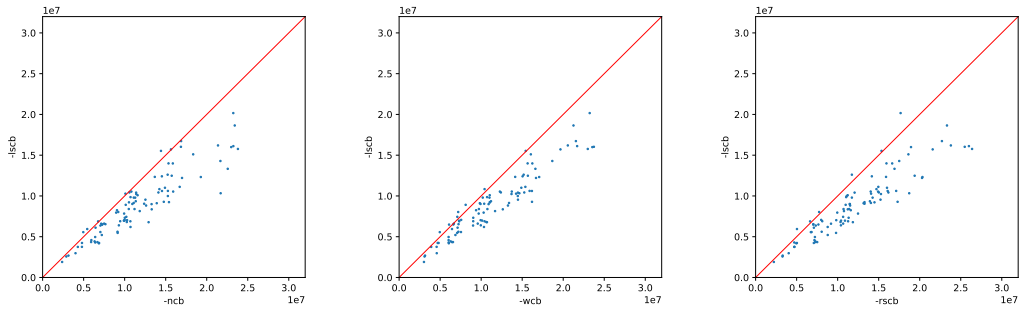
6.2 Integration of LSCB in CaDiCaL and Glucose

LSCB in CaDiCaL. We implemented our LSCB approach from Algorithm 1 in CaDiCaL [4], the baseline solver of the hack track of the SAT Competition. Thanks to the built-in model-based tester Mobical, the most effort came with ensuring that we have implemented correctly Invariant 4: CaDiCaL does not require watching the literals of two highest levels



(a) Satisfiable instances **(b)** Unsatisfiable instances

Figure 5 Average total number of propagations performed by NapSAT on the SATLIB 3-SAT random problem, clustered by the number of variables, and backtracking technique employed.



(a) LSCB vs. NCB **(b)** LSCB vs. WCB **(c)** LSCB vs. RSCB

Figure 6 Total number of propagations of NapSAT for each unsatisfiable Uniform Random 3-SAT problem from SATLIB. The red line is the equality line. Marks under the equality line favour our new approach.

when the clause is propagating. This, however, requires iterating over the clause to find the propagation level, which we do not need.

We remark that we did not change the default backjumping policy of CaDiCaL: when backjumping from more than 100 levels occurs (following the value implemented in CaDiCaL), we resort to backtracking (going one level back). Otherwise, an algorithm similar to trail reuse for restarts is used to decide how many levels should be kept. Unlike the version implemented in NapSAT, in CaDiCaL we store the missed level instead of checking the level of the MLI each time we need the level.

We tested various configurations, as summarized in Table 2, on the bwForCluster Helix with AMD Milan EPYC 7513 CPUs, using a memory limit of 16 GB RAM on the problems from the SAT Competition. Overall, we can see there is little difference between the considered configurations. In particular, the performance difference between WCB and NCB is limited, making it unclear if chronological backtracking is important. However, similar to the original CaDiCaL implementation [15], on the benchmarks from the SAT Competition 2018, there is an improvement from WCB over NCB. Our intuition is that chronological backtracking is especially useful when the decision heuristic is picking the wrong literals finding conflicts

Table 2 Number of solved instances by different variants of strong backtracking on the SC2023 competition, using a 5.000 s timeout

CaDiCaL version	solved	PAR-2 ($\times 10^3$)
base-CaDiCaL = RSCB	**248**	**4.09**
LSCB, Analyze-2 and minimization	246	4.16
ESCB	245	4.16
LSCB and Analyze-2	246	4.19
NCB	247	4.19
LSCB and Analyze-1	242	4.24

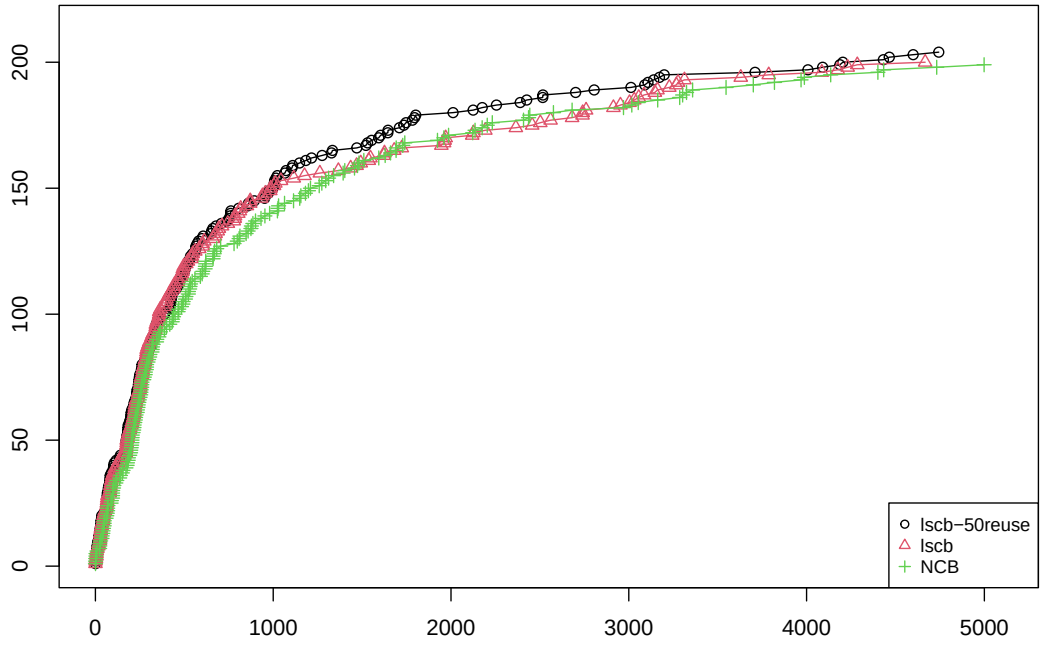
late instead of early (like finding a new unit at level 500 instead of level 1). The decision heuristics seem to perform worse on the 2018 benchmarks, while this did not seem to have happened since.

While the results in NapSAT seem to indicate a large decrease in the number of propagations, three factors mitigate this effect in CaDiCaL, as follows. (i) Propagating a literal ℓ a second time as in RSCB is cheaper than propagating it for the first time. Most clauses remaining in the watch list of ℓ will already be satisfied and are faster to check. (ii) RSCB allows to use of blocking literals more loosely. There is no need to compare the level of the blocking literal and the propagated one, making them more potent. (iii) Searching for a replacement literal is slightly more expensive in LSCB since we need to record the highest literal in the clause.

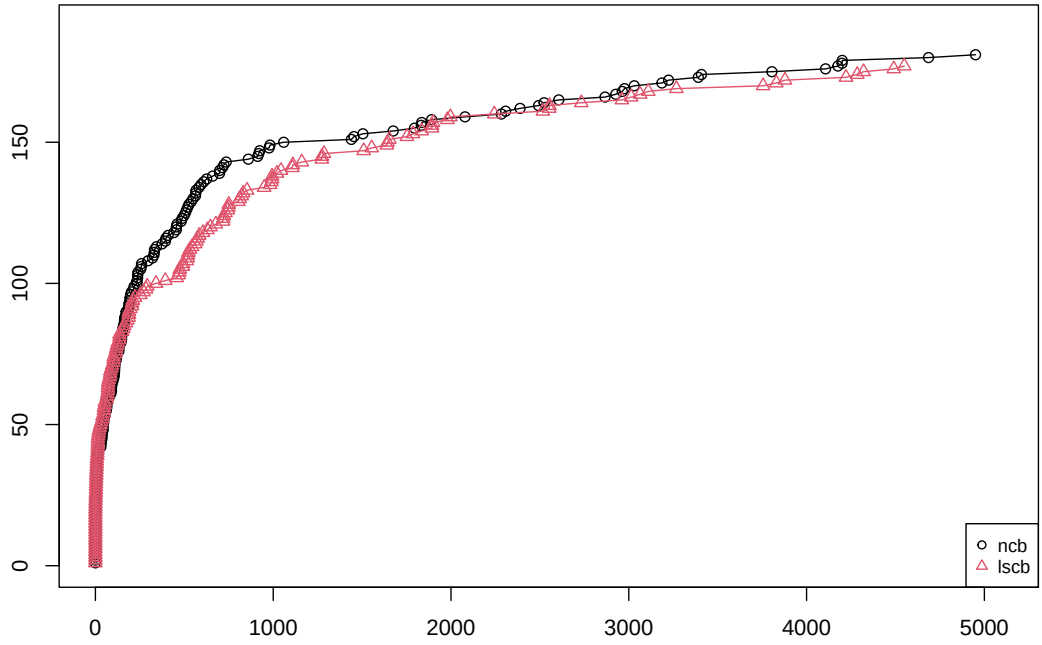
In a context where propagations are more expensive, such as SMT or user-propagators [8], these considerations might weigh less on the overall performance of the solver. We will investigate these applications for future work.

LSCB in Glucose. We also implemented our Algorithm 1 for LSCB into the latest version of Glucose [1], the SAT solver that pioneered the LBD heuristic for the usefulness of clauses (only without the minimization part). This is the only solver where we implemented the LSCB without any existing CB in the code. The entire diff (including new logging information and more assertions) is less than 1.000 lines. Our actual implementation of LSCB in Glucose is very close to our abstract Algorithm 1, because the blocker literal is always exactly the other watched literal. We use the simple heuristic to backtrack one level if jumping back more than 100, otherwise use the normal backjumping. We did not change the heuristic to block restarts [2], which is based on the trail length.

While running Glucose with LSCB on the SAT Competition 2023 (Fig. 7b), we observed worse performance. Interestingly, this is mostly due to one family of benchmarks, `SC23_Timetable`, that perform much worse with strong backtracking (but are solved eventually). On the 2018 benchmarks again (Fig. 7a), we observed a slight performance improvement when using Glucose with LSCB and it seems to be better to trigger chronological backtracking more often.



(a) GLUCOSE variants in the SAT Competition 2018



(b) GLUCOSE variants in the SAT Competition 2023

Figure 7 CDF of the different GLUCOSE (without strategy adapting) versions. The constant indicates when chronological backtracking is triggered instead of backjumping: We apply chronological backtracking when NCB would require jumping back more than 100 levels by default, like in CADICAL. In the SAT Competition 2018, the version that triggers chronological backtracking for more than 50 levels performs best.

7 Conclusion

We introduce a lazy reimplication procedure to be used in CDCL with (variants of) chronological backtracking. We particularly focus on the definitions of weak chronological backtracking (WCB), restoring strong chronological backtracking (RSCB), eager strong chronological backtracking (ESCB), and lazy strong chronological backtracking (LSCB). Our invariant properties on these backtracking variants exploit watched literals. We prove that our approach of lazy reimplication in strong chronological backtracking (LSCB) yields a sound and complete SAT solving method. Our implementation in NapSAT, and its integration with CaDiCaL and Glucose, gives practical evidence that LSCB is significantly easier to implement than ESCB, while also propagating fewer literals than RSCB, and providing better guarantees than WCB.

In the future, we intend to extend our LSCB method to reason over virtual literal levels, that is, levels of missed lower implications if such a clause is detected. We believe such an extension would allow to converge closer to the guarantees of ESCB, while mitigating both algorithmic complexities and reimplication costs. Further, we will explore the integration of chronological backtracking variants in the context of SMT, as a robust approach to handling arbitrary incremental clauses and expensive theory propagations.

References

1 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, pages 399–404, 2009. URL: `http://ijcai.org/Proceedings/09/Papers/074.pdf`.

2 Gilles Audemard and Laurent Simon. On the Glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. `doi:10.1142/S0218213018400018`.

3 Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In *SAT*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008. `doi:10.1007/978-3-540-79719-7\_4`.

4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel und Vijay Ganesh, editor, *Computer Aided Verification - 36th International Conference, CAV 2024, Paris, France, July 24-27, 2024*, LNCS. Springer, 2024. To appear.

5 Armin Biere, Mathias Fleury, and Florian Pollitt. CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023. In *SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 14–15. University of Helsinki, 2023.

6 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. `doi:10.3233/FAIA336`.

7 Armin Biere, Matti Järvisalo, and Bejamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391 – 435. IOS Press, 2nd edition edition, 2021.

8 Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. User-propagators for custom theories in SMT solving. In David Déharbe and Antti E. J. Hyvärinen, editors, *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 71–79. CEUR-WS.org, 2022. URL: `https://ceur-ws.org/Vol-3185/extended6630.pdf`.

9 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache Conscious Data Structures for Boolean Satisfiability Solvers. *JSAT*, 6(1-3):99–120, 2009. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/71`.

10 Robin Coutelier et al. Chronological vs. Non-Chronological Backtracking in Satisfiability Modulo Theories. Master's thesis, Université de Liège, Liège, Belgique, 2023.

11 Robin Coutelier Pascal Fontaine. ModularIT solver. Accessed March 2024. URL: `https://gitlab.uliege.be/smt-modules/`.

12 Randy Hickey and Fahiem Bacchus. Trail saving on backtrack. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2020. `doi:10.1007/978-3-030-51825-7\_4`.

13 Holger H Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *Sat*, 2000:283–292, 2000.

14 Norbert Manthey. Watch Sat and LTO for CaDiCaL. In *SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 10–11. University of Helsinki, 2023.

15 Sibylle Möhle and Armin Biere. Backing Backtracking. In *SAT*, volume 11628 of *LNCS*, pages 250–266. Springer, 2019. `doi:10.1007/978-3-030-24258-9\_18`.

16 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001. `doi:10.1145/378239.379017`.

17 Alexander Nadel. Introducing Intel(R) SAT Solver. In *SAT*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.SAT.2022.8`.

18 Alexander Nadel and Vadim Ryvchin. Chronological Backtracking. In *SAT*, volume 10929 of *LNCS*, pages 111–121. Springer, 2018. `doi:10.1007/978-3-319-94144-8\_7`.

19 Florian Pollitt. Cadical 1.9.4. Accessed March 2024. URL: `https://github.com/arminbiere/cadical/tree/reimply-branch`.

20 Coutelier Robin. NapSAT solver. Accessed March 2024. URL: `https://github.com/RobCoutel/NapSAT`.

21 João P. Marques Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. `doi:10.1109/12.769433`.

22 Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In *SAT*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009. `doi:10.1007/978-3-642-02777-2\_23`.