Unit Propagation with Stable Watches

Markus Iser
Karlsruhe Institute of Technology (KIT), Germany

Tomáš Balyo
CAS Software AG, Karlsruhe, Germany

Abstract

Unit propagation is the hottest path in CDCL SAT solvers, therefore the related data-structures, algorithms and implementation details are well studied and highly optimized. State-of-the-art implementations are based on reduced occurrence tracking with two watched literals per clause and one blocking literal per watcher in order to further reduce the number of clause accesses. In this paper, we show that using runtime statistics for watched literal selection can improve the performance of state-of-the-art SAT solvers. We present a method for efficiently keeping track of spans during which literals are satisfied and using this statistic to improve watcher selection. An implementation of our method in the SAT solver CaDiCaL can solve more instances of the SAT Competition 2019 and 2020 benchmark sets and is specifically strong on satisfiable cryptographic instances.

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

Keywords and phrases Unit Propagation, Two-Watched Literals, Literal Stability

Digital Object Identifier 10.4230/LIPIcs.CP.2021.6

Category Short Paper

Supplementary Material *Software*: `https://github.com/sat-clique/cadical_stability`

1 Introduction

Boolean satisfiability (SAT) solvers are used in a large variety of applications, e.g., software and hardware verification [2], automated planning [11], or cryptography [10]. Complete state-of-the-art SAT solvers are based on the Conflict-Driven Clause Learning (CDCL) algorithm [8, 9]. CDCL conducts a series of decisions with subsequent unit propagation and conflict resolution. The runtime of CDCL is dominated by the runtime of unit propagation [5]. Unit propagation is the process of inferring a new assignment from a current partial assignment and the given set of clauses. That requires to map literals which are not satisfied by the current partial assignment to the clauses in which they occur. One can effectively reduce the number of clause accesses for unit-clause and conflict detection by watching only two literals per clause [9, 3].

In this paper we propose a new method for selecting the two literals to be watched for each clause. We introduce the notion of stable literals, i.e., literals that tend to be satisfied for long time periods during the CDCL search. In our method, such stable literals are preferred when selecting new watched and blocking literals.

We implemented our method by modifying the well-known state-of-the-art SAT solver CaDiCaL [1]. Compared to the original CaDiCaL, our modified version can solve more benchmark instances of the Main tracks of SAT Competitions 2019 and 2020 and performs specifically well on a set of satisfiable cryptographic instances. Additionally, the two versions are rather orthogonal in the sense that they perform well on different subsets of the benchmark instances.

2 Preliminaries and Related Work

A *Boolean variable* can take on two possible values: *True* and *False*. A *literal* is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is a disjunction ($\vee$) of literals and a formula is a conjunction of clauses. A clause with only one literal is called a *unit clause*. A positive (resp. negative) literal is satisfied if the corresponding variable is assigned the value *True* (resp. *False*). A clause is satisfied, if at least one of its literals is satisfied and a formula is satisfied, if all its clauses are satisfied.

The satisfiability (SAT) problem is to determine whether a given formula has a satisfying assignment, and if so, also find it. A key component of the CDCL algorithm is unit propagation, which is the following process. Given a unit clause $C = \{l\}$, l has to be satisfied in each possible model of the formula, therefore we can immediately assign l's variable such that l is satisfied. Next we remove each clause that contains l from the formula (since these clauses are already satisfied) and remove $\bar{l}$ from each clause that contains $\bar{l}$ (since these clauses cannot be satisfied by this literal anymore). Removing literals from clauses may produce new unit clauses or an empty clause. The process is repeated until no more unit clauses are found or until an empty clause is generated. In the latter case, the algorithm has uncovered a conflict between the partial assignment and the formula.

Unit propagation dominates the runtime of the CDCL algorithm [5], therefore it is of paramount importance to implement this procedure as efficiently as possible. Currently the best known implementations of unit propagation are based on the idea of two-watched literals [9, 3]. As long as both watched literals are unassigned or satisfied, the clause can be ignored. When a watched literal is falsified, we must check if some of the clauses where it is watched became unit or empty, otherwise we find a new literal to watch for those clauses.

Competitive implementations of literal watch lists store a so-called *blocking literal* next to each clause pointer. The blocking literal is an arbitrary literal from the clause. If the blocking literal is satisfied, the clause access can be skipped. Another optimization which is used in state-of-the-art implementations is that the search for a new literal to watch does not start at the beginning of the clause. Instead, the position of the last found watching literal is stored in the clause and search starts from there (cycling through the beginning when we pass by the end) [4].

Epochs in CDCL SAT solvers can be measured in several ways. The naive approach is to measure an epoch in terms of real time, which is usually not a good idea as it hurts the reproducibility of runtime results [7]. Epochs are better measured in terms of assignment phases, i.e., the number of decisions, conflicts, or propagations. Recently, Biere came up with the dedicated epoch *ticks* which approximates the number of accessed cache lines during propagations [1]. In the following, an epoch n denotes the solver state in which we process the n-th decision – the *number of decisions*.

3 Selecting Stable Literals to Watch

The *stability* of a literal l is the number of epochs in which l has been satisfied. In Section 3.1, we show how to efficiently maintain and calculate literal stability. Our modified unit propagation periodically prioritizes literals of high stability as watched and blocking literals. We call this concept *stable watches* and explain it in Section 3.2.

Procedure limp(Literal l).

Data: Stability $S : \mathit{Literals} \to \mathbb{N}$
Data: Number of Decisions N
$S[l] \leftarrow N - S[l]$

3.1 Literal Stability

The *stability* of a literal l denotes the total number of epochs (as specified by the total number of decisions) in which l has been satisfied. Those epochs are given by a set of tuples $\{(T_i(l), U_i(l)) \mid 0 < i \leq n\}$ of start- and stop-epochs, in which $T_i(l)$ denotes the i-th epoch in which l becomes True (satisfied) and $U_i(l)$ denotes the i-th epoch in which l becomes unassigned or unsatisfied. It is easy to see that $U_i(l) \geq T_i(l) \geq U_{i-1}(l) \geq \cdots \geq T_1(l)$. The stability $S_n(l)$ of a literal l is given by Definition 1.

▶ **Definition 1** (Literal Stability). *Given a literal l which is satisfied in epochs $\{(T_1(l), U_1(l)), \ldots, (T_n(l), U_n(l))\}$, its stability $S_n(l)$ is defined as follows.*

$$S_n(l) := \sum_{i=1}^{n} \big(U_i(l) - T_i(l)\big)$$

We can incrementally update literal stability during backtracking in an epoch $U_i(l)$ by using the recursive form $S_n(l) = \big(U_i(l) - T_i(l)\big) + S_{n-1}(l)$. However, this requires to additionally keep track of epochs $T_i(l)$ in which l gets assigned to true. By reformulating the recursive form like in Equation 1, we can store $T_i(l)$ as an intermediate state of $S_n(l)$ in order to save some cache on a hot path in the solver.

$$S_n(l) := U_i(l) - \big(T_i(l) - S_{n-1}(l)\big) \tag{1}$$

In our implementation, we update $S_n(l)$ with the dirty intermediate value $S_n^d(l)$ (Equation 2) when it is assigned to true and then use $S_n^d(l)$ to calculate the new literal stability $S_n(l)$ when l is backtracked (Equation 3).

$$S_n^d(l) = T_i(l) - S_{n-1}(l) \tag{2}$$
$$S_n(l) = U_i(l) - S_n^d(l) \tag{3}$$

Our method for interval accumulation is specified by Procedure limp. Procedure limp is called twice per interval, first when a literal l becomes satisfied by a taken assignment in epoch T and again when that assignment is undone by backtracking in epoch $U \geq T$. Between T and U, literal stability $S[l]$ is in its dirty state. After each second call to $\mathsf{limp}(l, U)$, $S[l]$ is a valid sum of intervals.

3.2 Selecting Stable Watches

Traditional implementations of the two watched literal algorithm start by watching the first two literals in the clause and also blocking literals are initialized accordingly. During search new watchers are found according to the order in which literals are stored. Also the blocking literal is updated lazily during propagation.

Our approach exploits the fact that watched and blocking literal initialization starts with the first literals in the clause and then progresses along the order of literals in the clause. Watcher (re-)initialization takes place regularly in *cleanup phases* for learned clause

Procedure StableWatches(Assignment A, Clauses C).

Data: Stability $S : \mathit{Literals} \to \mathbb{N}$
Data: Value $v : \mathit{Literals} \to \{0, 1, 2\}$

```
// Cleanup Stability Values
1 for Literal l ∈ A do limp(l)
// Apply Stability-Induced Priorities
2 for Clause c ∈ C do
3     if c is not Reason Clause then
4         stable_sort(c.literals, l_0 < l_1 ⟺ v(l_0)S(l_0) > v(l_1)S(l_1))
// Revert to Dirty State
5 for Literal l ∈ A do limp(l)
```

forgetting and memory defragmentation. By reordering the literals in each clause, we control the order in which literals are considered as watching and blocking literals. For each clause, we sort literals in a descending order according to their *literal stability*.

Procedure StableWatches outlines our method. It is called after cleanup and before reattaching clauses to the watcher data-structure. Before we can use the accumulated literal stabilities, we have to fix those values which are currently in their dirty state, and revert their values after sorting (Lines 1 and 5). In order to protect the relative order of literals with the same sorting value, we use stable sorting.

Since there exists a partial assignment that can falsify even the most stable literal of a clause, we must also be careful not to watch a falsified literal. Therefore, we multiply the stability of currently false literals by zero. We place additional weight on the stability of a literal that is also currently satisfied by multiplying its stability by two. This was very easy to implement based on the value function already available in CaDiCaL. The factors are given by the value function $v(l)$, which is as follows.

$$v(l) = \begin{cases} 0 & \text{if } l \text{ is false} \\ 1 & \text{if } l \text{ is unassigned} \\ 2 & \text{if } l \text{ is true} \end{cases}$$

Then, for each clause (Line 2) which is not a reason clause (Line 3)[1], we (stable-) sort its literals in a descending order according to the value of the product of their stability and the value function (Line 4).

4 Evaluation

We experimentally investigated the effectiveness and efficiency of our methods. Our experiments were executed on a cluster of 20 compute nodes, each equipped with 32 GiB RAM and 2×2.66 GHz *Intel Xeon E5430* CPU. The operating system is *Ubuntu 18.04.4 LTS, Linux Kernel 5.4.0-66*. We ran 2 processes per node and used a time limit of 5000 seconds and a memory limit of 16 GiB per benchmark instance.

[1] A clause can be reason for propagation in the current partial assignment, in which case literal order carries additional semantics.

Table 1 PAR-2 score and number of solved instances for several variants of watched literal prioritization in Candy.

Method	PAR-2 Score	Solved Instances
Literal Stability	6747	152
Literal Constrainedness (Desc.)	6844	151
Variable Constrainedness (Desc.)	6920	149
Default Performance	6952	145
Variable Constrainedness (Asc.)	6954	143
Literal Constrainedness (Asc.)	7201	132

We experimented with three sets of instances. The instance sets Main-2020 (400 instances) and Main-2019 (399 instances) correspond to the benchmark sets which were used in the Main tracks of the respective SAT Competitions. By projecting on the instance families represented in Main-2020, we found that our method seems specifically well suited for cryptographic instances. Our third benchmark set Crypto is a collection of 409 cryptographic instances of previous SAT competitions.[2] Both the number of instances solved and the average runtime with a penalty factor of two (PAR-2 score) are used to compare performance.

We also ran initial experiments with our SAT solver Candy on the instances in Main-2020. We report on those in Section 4.1. Later, we were able to reproduce and further analyze our results with the well-known state-of-the-art SAT solver CaDiCaL (Version 1.1.4) by Armin Biere. We report on results for CaDiCaL with all instances in Section 4.2.

4.1 Initial Results

Literal Stability emerged as a possible explanation for what happens in our initial experiments with (trivial) constrainedness-based watcher-priorities. Table 1 displays the preliminary results for several types of literal priorities, which we used for establishing watched literal priorities through recurrent watcher reinitialization in the clause forgetting intervals of our solver Candy. In our initial experiments, we sorted clauses by variable and literal constrainedness, both in ascending and descending order. To calculate constrainedness, we use the well-known Jeroslow-Wang score [6].

Our experimental data shows that prioritizing watched literals by low constrainedness leads to fundamentally worse performance than prioritizing those of high constrainedness. Prioritizing by high variable and literal constrainedness both outperform the original implementation. Prioritizing by literal stability however, shows the best performance, solving seven more instances than the default approach.

4.2 Experimental Results

A summary of the results of our experiments with CaDiCaL and our modified version CaDiCaL Stability is displayed in Table 2. We also included a combination of both versions that takes the best result for each benchmark – a virtual best solver (VBS). CaDiCaL Stability solves 6 instances more in Main-2019 and 4 instances more in Main-2020. Figure 1 shows that our approach is stronger in long solver runs.

In Main-2020, our approach is particularly strong on the station-repacking and cryptographic families of instances. Of the 12 station-repacking instances, CaDiCaL solves 4 instances, while CaDiCaL Stability solves 10 instances. Of the 35 cryptographic instances, CaDiCaL solves

[2] Query for `family = cryptography` at `https://gbd.iti.kit.edu/`

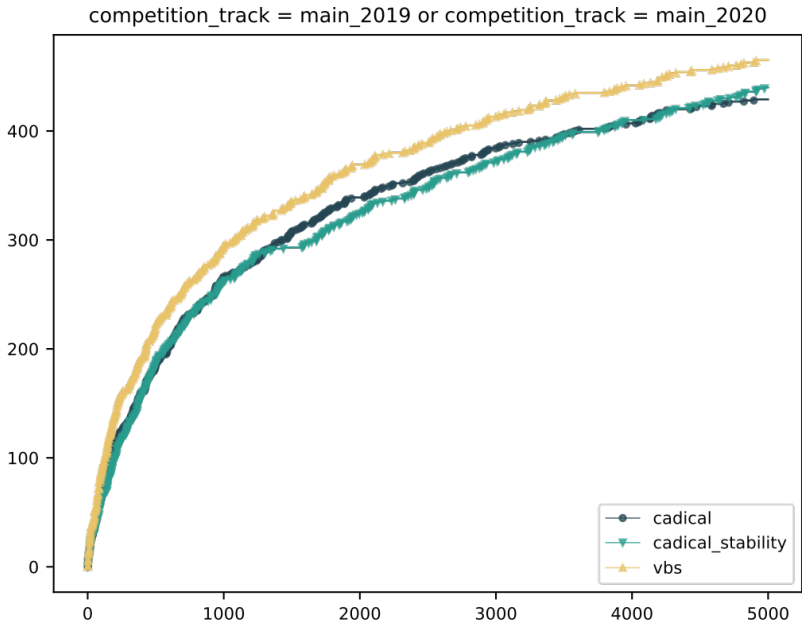

Figure 1 Cumulated runtimes of CaDiCaL with and without Stable Watches, and their VBS on the benchmarks of SAT Competitions 2019 and 2020.

16 instances, while CaDiCaL Stability solves 20. Table 2 and Figure 2 show, that our approach performs significantly better on the 409 instances in Crypto. CaDiCaL Stability solves 13 instances more in Crypto with a significantly better PAR-2 score.

The clear winner is however VBS, a theoretical solver combining both approaches with a perfect oracle that selects for each instance the fastest approach. This suggests that the two CaDiCaL versions are orthogonal and would work well together in a portfolio.

Since our approach comes with the overhead of maintaining literal stability statistics, which is done once per value assignment and again during backtracking, we measured whether we actually speed-up unit propagation, i.e., watcher iteration. The average number of propagations per second (PPS) over all instances in Main-2019, Main-2020 and Crypto goes down from 1.06 million PPS for CaDiCaL to 0.99 million PPS for CaDiCaL Stability. Also the total number of propagations goes down from 2.23 billion propagations for CaDiCaL to 2.16 billion propagations for CaDiCaL Stability. So on average, our approach does less propagations per second, but it also needs a lower total number of propagations to solve the benchmark instances.

Table 2 PAR-2 score and number of solved instances of CaDiCaL, CaDiCaL Stability and their VBS on several benchmarks.

		CaDiCaL	CaDiCaL Stability	VBS
Main-2019	**Solved**	229	235	243
	PAR-2	4937.9	4890.3	4581.0
Main-2020	**Solved**	215	219	237
	PAR-2	5212.2	5221.2	4732.5
Crypto	**Solved**	222	235	259
	PAR-2	5343.5	5146.7	4593.9

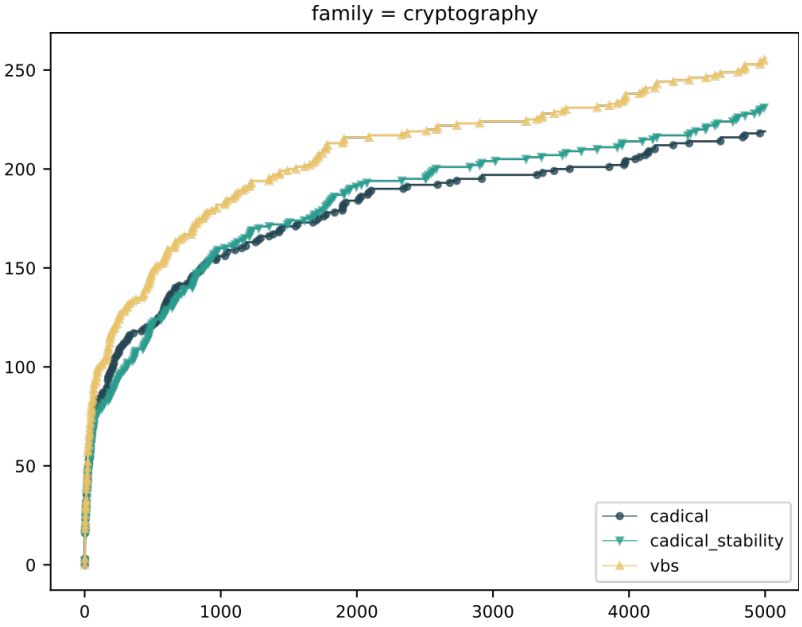

Figure 2 Cumulated Runtimes of CaDiCaL with and without Stable Watches, and their VBS on a set of cryptographic instances aggregated from several SAT Competition benchmarks.

5 Conclusion

We showed that we can we afford the overhead of maintaining literal stability values on the assignment level (which is a hot path). Using stability values to establish priorities for watched literals leads to improved SAT solver performance, particularly on *satisfiable cryptographic* instances. We also showed that the observed performance gain is *not* due to an increased number of propagations per second but by requiring less total propagations to solve the benchmark instances.

The internal state of the watcher data-structure determines *propagation order*. A partial assignment can be conflicting for several reasons. With stable watches we break ties differently such that we analyze *different conflicts*. In the presented approach, propagation-ties are resolved in favor of clauses which are less stable (or more rarely satisfied). We could empirically show that this helps finding solutions for hard satisfiable instances more quickly.

In the future, we expect other effective tie-breakers to be discovered and analyzed. Future work should focus on how exactly resolution space navigation is affected by propagation order for several types of instances. In the recent SAT Competition 2021, Kaiser and Clausecker won a special price with their solver CaDiCaL_PriPro, which performs a different kind of prioritized propagation.[3] This is an additional indication that propagation order is important.

Our modified CaDiCaL is kind-of orthogonal to the original CaDiCaL in the sense that it performs well on a different subset of the benchmark instances. This suggests, that combining our approach with the standard approach to select literals to watch could be a promising topic for future work. That might include research on hybrid heuristics or instance-specific heuristic selection.

[3] `https://satcompetition.github.io/2021/downloads.html`

References

1. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2020*, pages 50–53. Department of Computer Science, University of Helsinki, 2020.
2. Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *J. Satisf. Boolean Model. Comput.*, 6(1-3):165–201, 2009.
3. Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artificial Intelligence*, Accepted for Publication, 2021.
4. Ian P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013.
5. Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010.
6. Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.
7. Stepan Kochemazov. F2TRC: deterministic modifications of SC2018-SR2019 winners. In Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, pages 21–22. Department of Computer Science, University of Helsinki, 2020.
8. Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535, 2001.
10. Saeed Nejati, Jan Horácek, Catherine H. Gebotys, and Vijay Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 737–754, 2018.
11. Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomás Balyo. Efficient SAT encodings for hierarchical planning. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019*, pages 531–538. SciTePress, 2019.