

# Improving Local Search and Adding DDFW to CaDiCaL

Florian Pollitt<sup>1</sup>, Mathias Fleury<sup>1,\*</sup>, André Schidler<sup>1</sup>, Armin Biere<sup>1</sup>, Johannes Gröber<sup>1</sup>, Jakob Peterson<sup>1</sup>, Rico Andris<sup>1</sup> and Lea Hohl<sup>1</sup>

<sup>1</sup>University of Freiburg, Germany

## Abstract

State-of-the-art SAT solvers from the SAT Competition are all using a local search component. Experimentally at the beginning of the search, many better assignment are found; so we developed a new method to not store those each time, based on caching flips. This method was already part of Kissat but was not described so far. We also describe the implementation in CaDiCaL which reuses watch lists instead of using full-occurrence lists. Finally, we ported the new TASSAT algorithm to CaDiCaL improving the performance of CaDiCaL.

## Keywords

Local Search, DDFW, SAT solver

## 1. Introduction

Since 2019, all winning solvers in the SAT Competition have integrated stochastic local search (SLS) or *walk* as a complementary component to conflict-driven clause learning (CDCL) [1]. This hybrid approach is supposed to combine both strength as they exchange information [2]. While most local search solvers and component (including KISSAT's local search component) use full-occurrence lists to index clauses, CADICAL's component has historically relied on single-watched lists—a technique never formally described in prior work. To our knowledge we present the first empirical comparison of these two approaches in a unified framework. Unexpectedly, our experiments reveal that single-watched lists are slower at flipping, although this does not lead to better performance (Section 2).

Limiting SLS runtime deterministically (without measuring time) is a challenge. While ticks (a cache-miss approximation metric) have been used in KISSAT and CADICAL for the scheduling of inprocessing, CADICAL does not use it for walk. We found issues on few benchmarks and adapted walk to ticks, which is non-trivial due to the high cache locality of repeatedly accessed clauses. Our approach resolves the issue of SLS dominating solver runtime on certain benchmarks, while maintaining hardware independence and determinism for easier debugging (Section 3).

Additionally we introduce a flip cache mechanism to reduce the amount of best assignment storage during SLS. By tracking recent flips and avoiding storage of intermediate assignments when no better model is found we significantly reduce the number of stored best saved assignments. On the SAT Competition benchmarks (2002–2024) we show that the number of best assignments found decreases over time—a trend not previously quantified in the literature (Section 4).

The divide and distribute fixed weights (DDFW) algorithm [3], recently modernized in TASSAT [4, 5], is currently one of the best-performing local search approaches. However, its integration into CDCL solvers has been limited. We provide the first implementation of DDFW in a state-of-the-art solver, here

---

17th Pragmatics of SAT International Workshop 2026 (PoS'26)

\*Corresponding author.

†The base of this paper were the Master Project by Lea Hohl (tickings CADICAL's walk described in Sect 3) and the Master Lab by Johannes Gröber and Jakob Peterson (who reimplemented and cleaned up TASSAT) and Rico Andris (who incorporated their implementation in a simple SAT solver from the SAT lecture). Both projects were supervised by Florian Pollitt and Mathias Fleury and Armin Biere. Finally Mathias Fleury implemented the full-occurrence and TASSAT in CADICAL.

✉ pollittf@informatik.uni-freiburg.de (F. Pollitt); fleury@cs.uni-freiburg.de (M. Fleury); schidler@cs.uni-freiburg.de (A. Schidler); biere@cs.uni-freiburg.de (A. Biere); jg515@cs.uni-freiburg.de (J. Gröber); jp351@email.uni-freiburg.de (J. Peterson); andrisrico@gmail.com (R. Andris); hohll@cs.uni-freiburg.de (L. Hohl)

🆔 0009-0001-4337-6919 (F. Pollitt); 0000-0002-1705-3083 (M. Fleury); 0000-0001-6790-7158 (A. Schidler); 0000-0001-7170-9242 (A. Biere)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CADICAL<sup>1</sup>. Our experiments show that CADICAL as a local search solver with DDFW outperforms the original TASSAT implementation, but also that normal CADICAL becomes better (Sect. 5).

## 2. Stochastic Local Search

In this section we describe first the historical overview of local search (Section 2.1) before describing our two different data structures, one based on watch lists and the standard one based on full occurrence lists (Section 2.2).

### 2.1. Preliminaries

Stochastic local search (SLS) is an incomplete method for solving the Boolean Satisfiability Problem (SAT), designed to find satisfying assignments for satisfiable instances. Unlike conflict-driven clause learning (CDCL), which systematically explores the search space and can prove unsatisfiability, SLS operates by iteratively refining a full assignment of variables through local modifications. The goal is to minimize the number of *broken clauses*, which are clauses unsatisfied under the current assignment, until a satisfying model is found (i.e., no broken clauses remain).

The core operation in SLS is the flip, which changes the truth value of a single variable. While termination guarantees exist for some variants of local search under specific heuristics, most practical implementations rely on restarts or time limits to avoid infinite loops.

Three dominant SLS approaches are currently used in modern SAT solvers. PROBSAT is a purely probabilistic algorithm that selects a broken clause uniformly at random and flips one of its literals with a probability proportional to a predefined score function. It is simple to implement and achieves high flip rates due to minimal computational overhead. This is the version that is used in KISSAT and CADICAL, but also in CryptoMiniSAT [6] via YalSAT [7] since 2019 [8].

CCANR [9] extends PROBSAT by introducing configuration checking: a variable is only flipped if its neighborhood (variables appearing in the same clauses) has changed since its last flip. This reduces cycles by avoiding redundant flips but increases computational cost per flip due to neighborhood tracking. CryptoMiniSAT [6] can use it since 5.8.

The divide-and-distribute fixed weights (DDFW) algorithm [3] and its successors [4, 5] prioritizes harder-to-satisfy clauses by assigning higher weights to them. More precisely, when a clause is satisfied, its weight is redistributed to neighboring broken clauses. It is very efficient on some graph problems.

The key difference between these three algorithms lies in their flip selection strategies. PROBSAT invests minimal effort in choosing flips, leading to high flip rates but potentially inefficient search. CCANR and DDFW, by contrast, spend more time selecting flips, resulting in fewer flips per second but often better performance on challenging instances. However, their approach is different: DDFW tries to satisfy as many clauses as possible according to their weight, while CCANR looks at variables and their neighborhood. For a broader historical overview, we refer to the Handbook of Satisfiability [1] and Balint’s PhD thesis [10].

All the SLS algorithm looks like Algorithm 1: a heuristic is used to flip literals, then some broken clauses become satisfied and some satisfied ones containing the flipped literal become unsatisfied.

---

<sup>1</sup>The code is not included in CADICAL 3.0 but should be part of the version running at the SAT Competition 2026

---

```

while there is clause not satisfied do
     $L \leftarrow$  select literal to flip according to heuristic;
    Flip  $L$  to its opposite value;
    Mark all broken clauses containing  $L$  as satisfied;
    Check all clauses containing  $L$  and update their broken status;

```

---

**Algorithm 1:** Basic Stochastic Local Search Algorithm

In recent years SLS has been integrated into CDCL solvers as a complementary component [2]. Our intuition is that the assignment exported to CDCL helps the SAT solver by finding models but also that CDCL focuses on the hard part of the search space. For example, if the solver is composed of two independent components, one satisfiable and one unsatisfiable, when the SLS component solves the satisfiable part, the CDCL component will only find conflicts in the unsatisfiable part and focus on it. We believe that the SLS integration into CDCL is responsible for the performance jump that can be observed in 2019 on the historical data from the SAT Museum [11]. Most solvers use warm-up to initialize values (an idea from Knuth): the solver first propagates ignoring conflicts to produce a total assignment for the SLS part to get propagation chains set correctly.

Most pure local search solver feature restarts (by generating new random starting assignment each time). This is usually not done in the CDCL component, because the start point of the SLS component is generated by CDCL and changes every time.

## 2.2. Data Structure

The standard implementation (including the one from KISSAT [12]) is based on full-occurrence lists [10], but the implementation in CADICAL reuses watch lists to avoid allocating more memory.

**Full Occurrence List.** SLS solvers maintain an array `trueLit` to track the number of satisfied literals per clause and an occurrence list `occurrenceList[1]` for each literal  $l$ , storing all clauses where  $l$  appears. When a variable  $v$  is flipped, the solver traverses the occurrence lists of its satisfying and falsifying literals to update clause states. The standard caching approach stores the *critical variable* for each 1-satisfied clause. To avoid iterating over the clause, the XOR scheme [13] replaces the critical variable storage with a compact representation: the XOR of all satisfying variables in a clause, stored in `trueVarX[C]`. For a clause  $C$  with two satisfying variables  $x_i$  and  $x_j$ , this is computed as:

$$\text{trueVarX}[C] = x_i \oplus x_j.$$

PROBSAT uses the XOR trick to avoid storing the clauses at all. However, this makes it expensive to find the literal to flip. Instead KISSAT stores a reference to broken clauses because the CDCL solver needs to iterate over the clauses so we have to store the clauses. In that case the XOR scheme is not necessary, although we use it for DDFW to find the *critical* literal.

In Algorithm 2, we show the code that counts how many clauses would be broken when flipping a literal. This code is used to decide which literal to flip in PROBSAT.

**Single Watched.** The watch list implementation in CADICAL’s stochastic local search (SLS) solver adapts a single-watched literals technique from CDCL solvers to efficiently track clause satisfaction states during variable flips. This approach minimizes computational overhead by maintaining dynamic pointers to one literals per clause and incrementally updating only affected clauses.

For each literal  $l$ , CADICAL maintains a `watches(1)` list containing all clauses where  $l$  appears as the only watched literal (and always as first literal of the clause). Each watch entry stores a pointer to the clause and the second watched literal (`blit`). The blocking literal is cheap to access, making it a fast

---

**Data:** Current literal  $lit$  to check

**Result:** Returns the number of new clauses broken

$broken \leftarrow 0;$

**foreach** clause  $t$  in  $occurrences(L)$  **do**

$count \leftarrow \text{clause\_info}[t].count;$

**if**  $count == 1$  **then**

$broken \leftarrow broken + 1;$

---

**Algorithm 2:** Check how many clauses become broken in the full-occurrence scheme

check if the clause is already satisfied. This representation is compatible with the normal watch scheme (where 2 literals are watched) and we can reuse the same data structure instead of allocating new ones.

When flipping a variable  $v$  (toggling its truth value), the solver performs two sequential steps. First, the solver processes clauses in  $\text{watches}(\neg v)$  (clauses where  $\neg v$  was watched). For each clause  $C$  in  $\text{watches}(\neg v)$ , if  $C$  has no satisfied literals, the clause is added to the broken clauses. Otherwise, a new literal is watched and the clause is moved to that watch list. Second, broken clauses that are now satisfied are moved to the watch list of  $v$ .

The scheme employs a **move-to-front** strategy: when a literal satisfies a clause, it is moved to the front of the clause (as the first watched literal) to improve cache locality. This reduces the cost of future checks by avoiding recently checked unsatisfied literals. To find the next literal to flip, the code needs to find the number of broken clauses. In essence the algorithm is written in Algorithm 3.

---

**Data:** Current literal  $lit$  to check

**Result:** Returns the number of new clauses broken

$broken \leftarrow 0;$

**foreach** watch  $w$  in  $\text{watches}(lit)$  **do**

**if**  $value(w.blit) > 0$  **then**

**continue;**

**if**  $w$  is binary **then**

$broken \leftarrow broken + 1;$

**continue;**

    /\* The clause is not binary: expensive memory access \*/

**if** there exists another true literal  $other \neq lit$  **then**

$w.blit \leftarrow other;$

        Apply literal-move-to-front;

**else**

$broken \leftarrow broken + 1;$

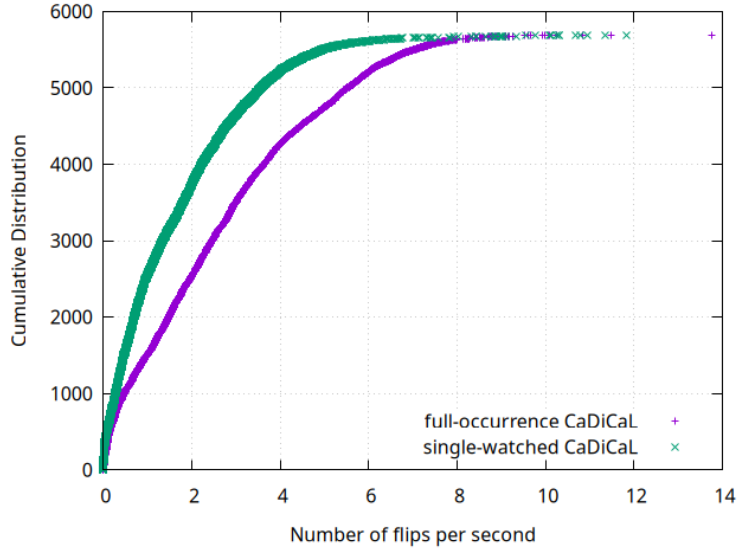
---

**Algorithm 3:** Check how many clauses become broken in the single-watched scheme

Accessing the blocking literal  $w.blit$  is cheap and avoids accessing the clauses. The move-to-front scheme avoid going quadratically over the clauses [14]. For CDCL, in CADICAL, we already remember where we searched last time and start from this point. We implemented it and observed a slight performance decrease and did not further looked into this. We attribute this to the fact that saving search does not reorder the clauses.

**Comparison.** The choice between full-occurrence lists and watch lists in stochastic local search (SLS) solvers presents a fundamental trade-off between memory consumption and computational efficiency during variable flips. Full-occurrence lists maintain explicit references to all clauses but make checking if the clause is satisfied cheap (as shown in Algorithm 2). In contrast, watch lists adopt a more memory-efficient strategy by tracking only one literal per clause. While this introduces additional memory accesses (if checking the blocking literal is not enough), the overhead is mitigated by the smaller size of watch lists compared to full-occurrence lists. Furthermore, the watch list scheme can exploit cache locality more effectively for many clauses, as it avoids the need to store or traverse large occurrence lists. Memory locality in the full-occurrence scheme can be improved by having the clauses sorted by the order in the occurrence lists. It would be possible to store an extra literal in the full occurrence list too, but this would double the already worse memory usage.

The performance gap between the two approaches can be narrowed through targeted optimizations. For instance, binary clauses—which are prevalent in many SAT instances—can be handled more efficiently in watch lists by reusing the blocking literal field to store the second literal, thereby avoiding additional memory indirection. This optimization is not directly applicable to full-occurrence lists, as their design inherently requires storing all clause references. However, a similar principle can be



**Figure 1:** Number of millions of flips per second in CaDiCaL, as CDF. CaDiCaL was run for 10 min and the average number of flips was taken (if any flip happened). The point with the highest number of flips per second for the full occurrence is a rather small instance *battleship-24-57-sat* (4680 clauses, 1767 variables) that is solved in the first walk call after 7875 flips with the single-watched version and 121005 flips with full occurrences.

applied to broken clauses (those not currently watched) by employing a tagged union to distinguish between clause pointers and literal pairs. The simplest approach is to use one bit and to store either the pointer or 2 literals. However, this requires 65 bits. Therefore, we actually store one literal on 31 bits (instead of 32 bits). If none of the literals fit into 31 bits (which is expected to happen very rarely, since CaDiCaL rennumbers literals to avoid holes), we use the pointer even if the clause is binary.

In Figure 1, we show in a CDF the number of flips per second. We can see that the number of flips per second is higher for the full-occurrence version and they seem to reach the same number of points. However, this did not improve performance (neither more solved instances nor a better PAR2 score). The total memory consumption was also the same (but the memory peaks are unlikely to be reached during local search).

### 3. Better Limits with Ticks

Since the local search is integrated into a CDCL solver we do not want to run it indefinitely or until completion. We want to limit the execution to a certain amount of time, relative to the current total running time of the solver. Since the approach should be hardware independent and deterministic, we introduce a counter which is explicitly increased as approximation to running time. Different local search algorithms and CNF instances have different characteristics, so limiting, e.g., number of flips is not consistent across implementations or even across CNF instances. KISSAT introduced ticks in 2020 [12], a more precise metric for running time that approximates cache-line accesses. CaDiCaL 3.0 [15] is also using ticks for internal scheduling of inprocessing and local search.

Introducing ticks for local search was much harder than in other parts of the SAT solver. Usually, when accessing a clause within CDCL, this clause is not in cache and, therefore, should count one tick. However, during local search, it is often the same clauses that are considered over-and-over again. Those remain in cache and should not count a ticks. Interestingly, not counting ticks in various parts of the search, always lead to instances where the ticks limit did not work. In the end, we decided to count *too many* ticks (but we count ticks everywhere), but consider that 1 tick in the rest of the solver is  $10^3$  ticks in local search.

## 4. Caching flips

We observed experimentally that SLS often finds many models (especially at the beginning of the search), and then many fewer models. Therefore, we introduce a *flip cache*: We remember the literals that have been flipped until the cache is full. As we know what the current assignment is, we can always go back by flipping backwards (going from the end of the trail).

**Example 1.** *Our flip cache contains  $A, B, \neg A, C$  and our current assignment is  $\neg ABC$ . If the best assignment was found after pushing  $B$  in our flip cache, to find it, we need to flip  $C$  to  $\neg C$ , and  $\neg A$  to  $A$ , yielding the best assignment  $A\neg B\neg C$ . We never need to flip further than the last best assignment.*

The flip cache operates as a lightweight trail of variable assignments, recording the sequence of literals flipped during the search process. When the cache reaches its capacity (set to one-fourth of the total number of variables), the solver must either *flush* the cache by storing the best model discovered so far by reversing the flip sequence or discard the cache if no improved model was discovered. This approach leverages two key insights: First, the current assignment is always maintained, eliminating the need to reconstruct it from scratch. Second, only flips occurring after the last best model need to be considered when restoring an improved assignment. Otherwise, the cache is simply emptied.

The implementation follows a straightforward workflow (Algorithm 4): for each flip, the literal is appended to the cache; when an improved assignment is found, its position in the cache is recorded; upon cache exhaustion, if a better assignment exists, the solver reverses flips from the end of the cache until reaching the recorded position, otherwise the cache is cleared. This mechanism is particularly efficient because it avoids storing complete assignments, instead maintaining only the sequence of flips, with the reversal operation being linear in the number of flips since the last improvement.

In order to test if how often a model is found, we run CADICAL for 10s on 5416 benchmarks (out of 7330) from the SAT competition between 2002 to 2024 (we remove those problems that cannot be solved by either CADICAL or KISSAT in 5000 s)—remark that not all benchmarks reach local search nor do all do the same number of walks. We printed the number of best assignments found every time the flip cache is full. The result is in Figure 2. We use transparency to give an idea of the density to be able to print all those problems: each curve is a green point with high transparency. The more points are printed, the less transparent the curve becomes. The graph therefore shows an area giving an idea how the best assignment for all problems changes. The graph clearly shows that the number of found assignment decreases over time (even if there are exceptions, e.g., the shades at the end of the 5th walk around 540).

---

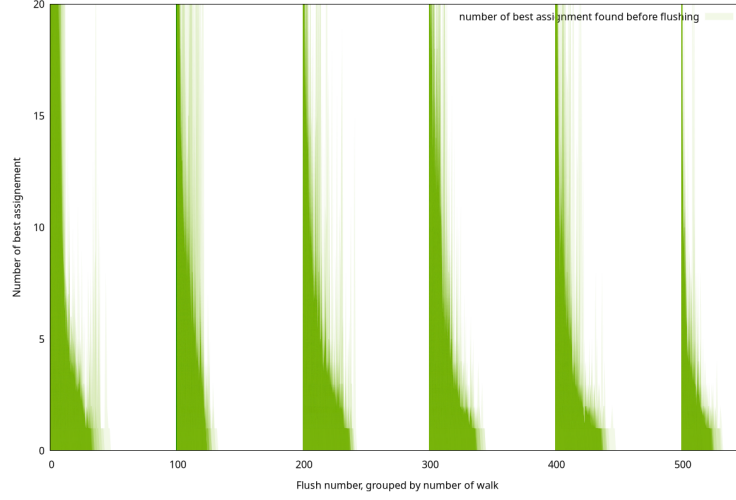
```
while instance is unsatisfied do
   $L \leftarrow$  literal to flip according to some heuristic;
  Flip  $L$  and update data structures;
  cache.append( $L$ );
  if better assignment found then
    | Remember current number of flips
  if cache is full then
    if best assignment is stored then
      | Save it by going backwards over the flips until the position of the assignment
    else
      | cache.clear()
```

---

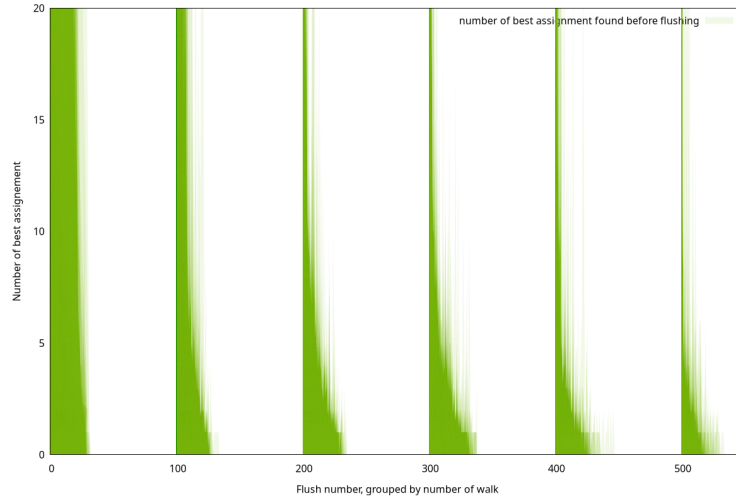
**Algorithm 4:** Stochastic Local Search with Flip Caching

## 5. DDFW in CADICAL

In order to include TASSAT’s DDFW, we reimplemented it in CADICAL, which allows us to use CADICAL as local search solver (Section 5.1). Then, we run CADICAL with DDFW limited by ticks (Section 5.2).



(a) CADICAL using PROBSAT



(b) CADICAL using DDFW

**Figure 2:** Number of best assignments found each time the flip cache is full (called a ‘flush’), which is the same as the number of assignments that do not need to be stored ( $y$ -axis). On the  $x$ -axis, the first digit is the zero-indexed walk and the remaining digits give the zero-indexed flush number, e.g., 203 is the third walk’s fourth flush. The graph is cut, since it reaches 1000 for DDFW and 100 for PROBSAT.

## 5.1. Reimplementation of DDFW

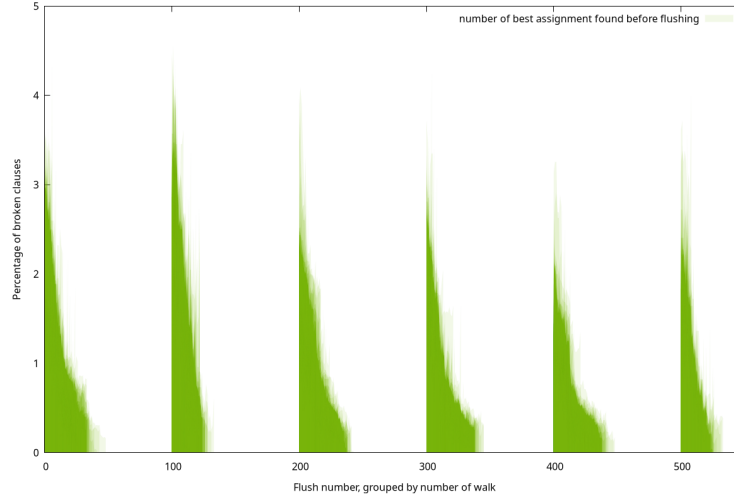
DDFW was extended to YAL-LIN and TASSAT recently. The authors implemented all their changes in the local search solver YalSAT and named those two tools YAL-LIN [4] and TASSAT [5] (the latter being mostly a (partially<sup>2</sup>) cleaned-up version of the former).

In Figure 3 we actually see how TASSAT performs compared to PROBSAT: it finds better assignments faster. This also matches the aim of the cooperation between CDCL and SAT: satisfy as many clauses as possible in order to let CDCL focus on the hard part.

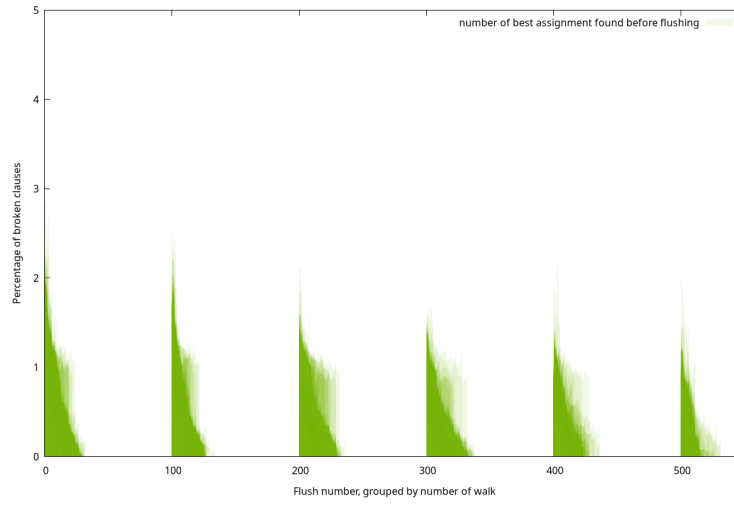
We have reimplemented the algorithms behind, DDFW, YAL-LIN, and TASSAT code in CADICAL. The first difference is that we only run those for a limited amount of ticks. The second difference is that we put all the information (weight clause pointer, xor of literals, or count of true literals) together in a single array. In our experience all information is used every time anyway (except for the weight transfer where only the weight is relevant). Therefore, we believe that our implementation is more efficient.

One (undocumented) change between YAL-LIN and TASSAT is the change of the base weight from 8 to

<sup>2</sup>most statistics are unused, the array for side jumps is still filled, although the array is never read.



(a) CADICAL using PROBSAT



(b) CADICAL using DDFW

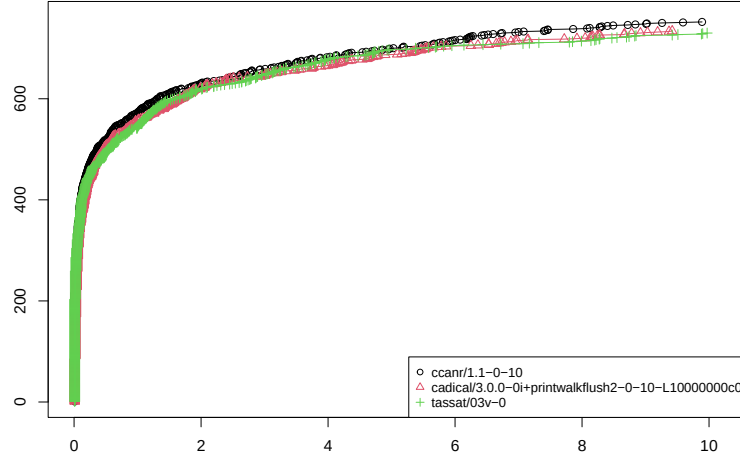
**Figure 3:** Number of broken clauses in percent of the current irredundant of clauses. The closer to 0 the number is, the closer the solver is to finding an model.

100. This change was done without changing the weight transfer when TASSAT is used in YAL-LIN mode. In CADICAL we scale the constants of the weight transfer according to the base weight. Empirically, we observed that TASSAT is faster at transferring weights.

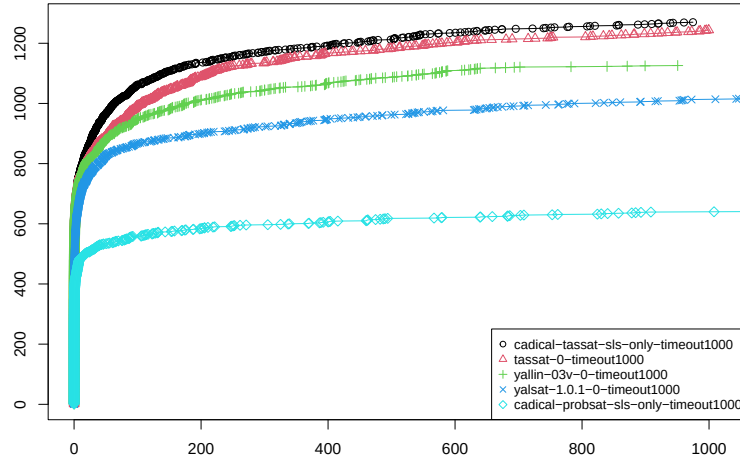
One interesting issue we had was the accumulation of floating-point errors. While debugging performance, we observed an infinite loop: CADICAL was flipping a literal back-and-forth because its weight was always exactly 0.06. This was a floating-point error (the real weight should have been zero), which led to CADICAL being stuck, until the ticks budget is finished. We fixed this by not flipping literals of weight  $\leq 0.1$ . We did not observe this effect in TASSAT on that example, but did not check all examples.

One change we made in the function to find the first weight reducing variable is that we remember how far we went and restart the search from there: we handle the vector as a circular buffer. This should make the selection of literals more uniform for ties, since we don't always start at the beginning.

We compared the performance of all three implementations (CADICAL as SLS only, TASSAT and CCANR). We can see that the performance of CADICAL and TASSAT is similar but CCANR performs better. We use a short 10 s timeout as this is closer to the usage in a SAT solver (unlike an instance of the random track of the SAT competition 2018 that used a 5000 s timeout).



**Figure 4:** CDF of the performance of CADIcAL (734 solved instances), TASSAT (730 solved instances), and CCANR (752 solved instances) with 10 s timeout.



**Figure 5:** CDF of the performance of CADIcAL using the local search only with 1000s timeout on all 7330 instances from the SAT Competitions 2002 to 2024.

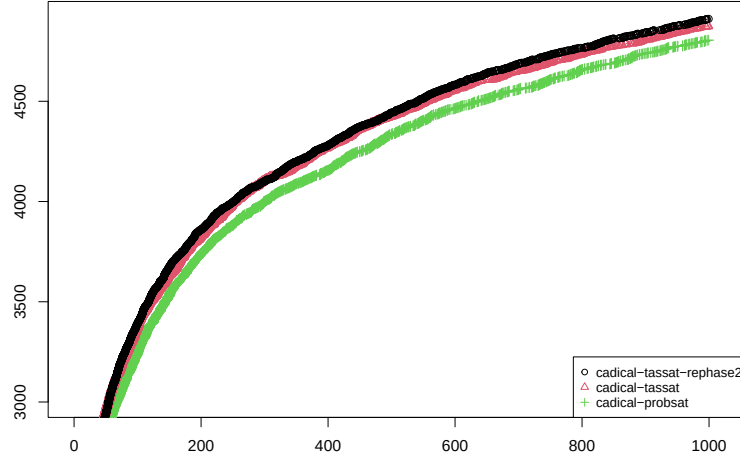
## 5.2. Using DDFW as Local Search

Our experiments were done on 96-core AMD EPYC 9654 CPUs running at 2.40 GHz and we used all problems from the SAT Competition 2002 to 2024 [16]. For each run, we enforced a time limit of 1000 s.

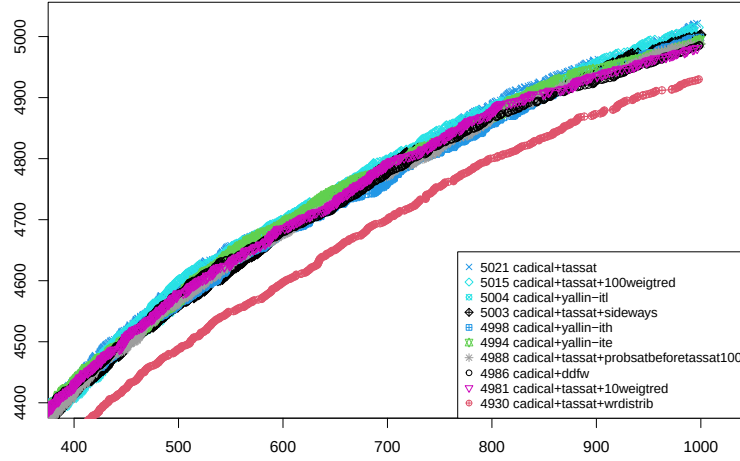
We use ticks to limit the amount of time spent in local search. Similarly to PROBSAT one thousand ticks in local search is one tick in the rest of the SAT solver. In Figure 6 we can see the performance of CADIcAL with PROBSAT and with DDFW: the latter solves nearly one hundred more instances.

Because of the different strengths of the algorithms, we tried to alternate between PROBSAT and DDFW but the performance ended up between the version using only PROBSAT and the version using only DDFW. We also alternated 3 DDFW rounds with 1 PROBSAT round but this did not improve performance either. We had slightly better performance with the option `rephase=2` which rephases only in stable mode like Kissat but we do not know whether this is due to the longer walk phases or not.

Interestingly we found a non-termination issue in CADIcAL: When the solver has assumptions, those set literals cannot be flipped. In our implementation we forgot to propagate before doing local search. This lead to non-termination of the function searching for a random satisfied clause (because no replacement clause was found).



**Figure 6:** CDF of the performance of CADIcAL with 1000s timeout. We compare here 3 strategies: ProBSAT (4805 solved instances), DDFW as drop-in replacement for ProBSAT (4878 solved instances), and DDFW with rephasing like in KISSAT (4911 solved instances). The last strategies run walk fewer times but longer, actually solving more satisfiable instances than the two other strategies.

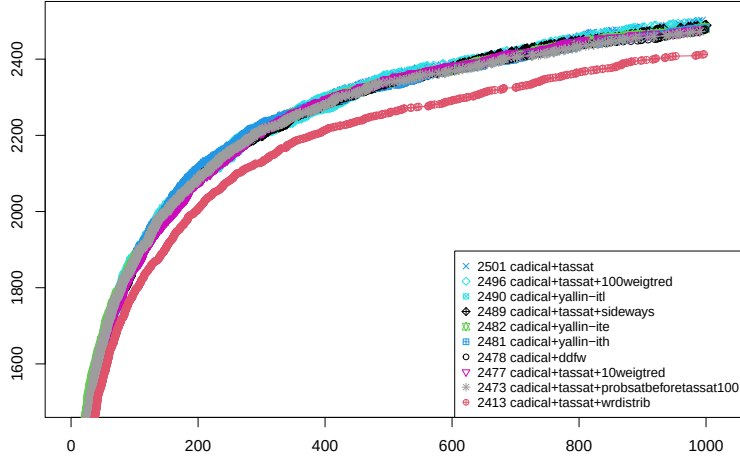


**Figure 7:** Performance of various configuration of local search. Exploring only a subset of the weight reducing variable ('weightred') is not helpful, neither are sideways flips, or adding a weight distribution over weight reducing variables ('wrdistrib'). However, CADIcAL with TASSAT's local search performs best.

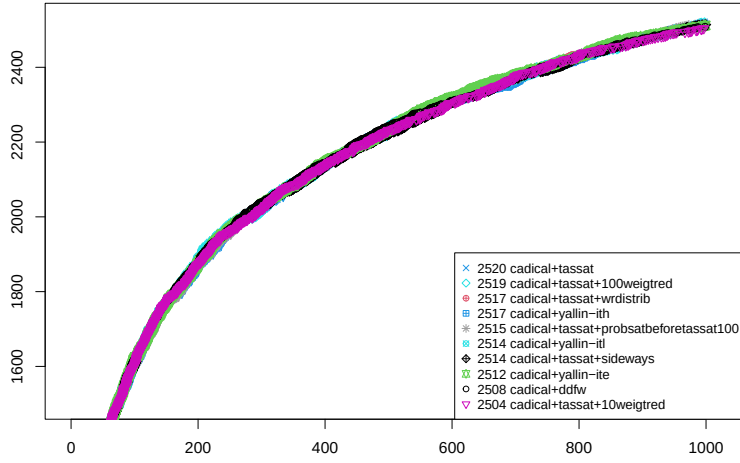
### 5.3. Variants of the DDFW Algorithm

In the YAL-LIN and TASSAT various different options have been tried but have not been kept in the final version of the code because performance was not improved. The CDF can be found in Figure 7 and the variants are described below. The main difference comes from the satisfiable instances (Figure 8) and not the unsatisfiable instances (Figure 9).

The algorithm works by eagerly flipping the literal that increases the most the weight of unsatisfied literals. The first improvement that can be found in YAL-LIN consists in using a probability function over the literals  $P(x_i) = \frac{\text{improvement}(x_i)}{\text{sum of all improvements}}$ . We reimplemented this, but we could not reproduce the improvements observed [4]. Following the ideas from MaxSAT or ProBSAT, we also tried  $P(x_i) = 2^{\text{sum of all improvements} - \text{improvement}(x_i)}$  to increase the focus on variable that improve weight. The performance was much better than the linear version but still slightly worse than the eager version. Probing only the first 100 or 1000 literals did not help either (whether with or without probability distribution over the candidates). Since the probability distribution has a similar effect for ties as our circular buffer, we also deactivated it. Our intuition was that hard-to-satisfy literals would slowly move to the beginning of the candidates. However, we did not see any performance change.



**Figure 8:** Performance of various configuration of local search, satisfiable-only instances.



**Figure 9:** Performance of various configuration of local search, unsatisfiable-only instances.

Another option which is in YAL-LIN but not in TASSAT are sideways jump: When there are no weight reducing variables, a variable that does not change the weight can be flipped (with a certain probability). This was removed from TASSAT<sup>3</sup> due to the observation that those flips become very unlikely after a few million flips. However, in CaDiCaL, we do only a limited number of flips. Therefore, we activated sideways flips, but observed a slight performance reduction.

In YAL-LIN, three main configurations of the parameters are tested, depending whether a lot or very little weight is taken from clauses initially. For YAL-LIN, the best configuration 'ith' is to take more weight at the beginning – one intuition here is that we want to move away as fast as possible from the initial, instead of take less ('itl') or equal ('ite') and exploring the initial state and its neighborhood more to know what clauses are hard. For CADICAL, the 'itl' version performs a little better, but the difference is very small.

Finally, the YAL-LIN mentions an optional mode where PROBSAT is run first to reduce the number of unsatisfied clauses, before going to DDFW, because DDFW is very expansive when many clauses are unsatisfied. We run this option too (until the number of broken clauses reaches the minimum of 100 and  $\#clauses / 100$ ), but did not see any improvement. We believe that this is due to the fact that we don't generate random assignments and instead use CDCL and warmup. Most flips (78.2%) where done by CADICAL's DDFW component – although 638 instances out of 7330 do only PROBSAT flips.

<sup>3</sup>although the array is still filled in the source code

## 6. Conclusion

We store flips, limiting the number of times the best assignment is saved, limiting memory writing. This technique was already present in KISSAT and is present in CADICAL too now. The difference in the data structure (full-occurrence vs single-watch lists) does not make a difference for the performance on average, although full-occurrence flips variables faster. We have implemented DDFW into CADICAL improving its performance and we expect it to become the default configuration in the next release.

There are two intriguing future works: first, CCANR could improve performance further and, second, a better understanding of the interaction between CDCL, local search and warm-up (can we export more information than just the assignment? what does warm-up do in the interaction?).

## Acknowledgment

This work was supported by the state of Baden-Württemberg through bwHPC, German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, and by a gift from Intel Corporation. We would also like to thank Mara Besemer for discussions (for a university seminar), which triggered all experiments from Section 5.3, and the reviewers for their comments.

## Declaration on Generative AI

During the preparation of this work, the authors used Mistral Large in order to: Grammar and spelling check, Paraphrase, and reword. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

## References

- [1] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2021.
- [2] S. Cai, X. Zhang, M. Fleury, A. Biere, Better decision heuristics in CDCL through local search and target phases, *J. Artif. Intell. Res.* 74 (2022) 1515–1563. URL: <https://doi.org/10.1613/jair.1.13666>. doi:10.1613/JAIR.1.13666.
- [3] A. Ishtaiwi, J. Thornton, A. Sattar, D. N. Pham, Neighbourhood clause weight redistribution in local search for SAT, in: P. van Beek (Ed.), *Principles and Practice of Constraint Programming - CP 2005*, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, *Proceedings, Lecture Notes in Computer Science*, Springer, 2005, pp. 772–776. URL: [https://doi.org/10.1007/11564751\\_62](https://doi.org/10.1007/11564751_62). doi:10.1007/11564751\_62.
- [4] M. S. Chowdhury, C. R. Codel, M. J. H. Heule, A linear weight transfer rule for local search, in: K. Y. Rozier, S. Chaudhuri (Eds.), *NASA Formal Methods - 15th International Symposium, NFM 2023*, Houston, TX, USA, May 16-18, 2023, *Proceedings, Lecture Notes in Computer Science*, Springer, 2023, pp. 447–463. URL: [https://doi.org/10.1007/978-3-031-33170-1\\_27](https://doi.org/10.1007/978-3-031-33170-1_27). doi:10.1007/978-3-031-33170-1\_27.
- [5] M. S. Chowdhury, C. R. Codel, M. J. H. Heule, TaSSAT: Transfer and share SAT, in: B. Finkbeiner, L. Kovács (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, *Proceedings, Part I, Lecture Notes in Computer Science*, Springer, 2024, pp. 34–42. URL: [https://doi.org/10.1007/978-3-031-57246-3\\_3](https://doi.org/10.1007/978-3-031-57246-3_3). doi:10.1007/978-3-031-57246-3\_3.
- [6] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic problems, in: O. Kullmann (Ed.), *Theory and Applications of Satisfiability Testing - SAT 2009*, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. *Proceedings, Lecture Notes in Com-*

- puter Science, Springer, 2009, pp. 244–257. URL: [https://doi.org/10.1007/978-3-642-02777-2\\_24](https://doi.org/10.1007/978-3-642-02777-2_24). doi:10.1007/978-3-642-02777-2\_24.
- [7] A. Biere, CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017, in: T. Balyo, M. Heule, M. Järvisalo (Eds.), Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions, volume B-2017-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2017, pp. 14–15.
  - [8] M. Soos, A. Biere, CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019, in: M. Heule, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Race 2019 – Solver and Benchmark Descriptions, volume B-2019-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2019, pp. 14–15.
  - [9] S. Cai, C. Luo, K. Su, CCAnr: A configuration checking based local search solver for non-random satisfiability, in: M. Heule, S. A. Weaver (Eds.), Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24–27, 2015, Proceedings, Lecture Notes in Computer Science, Springer, 2015, pp. 1–8. URL: [https://doi.org/10.1007/978-3-319-24318-4\\_1](https://doi.org/10.1007/978-3-319-24318-4_1). doi:10.1007/978-3-319-24318-4\_1.
  - [10] A. Balint, Engineering stochastic local search for the satisfiability problem, Ph.D. thesis, University of Ulm, 2014. URL: [http://vts.uni-ulm.de/docs/2014/8852/vts\\_8852\\_13227.pdf](http://vts.uni-ulm.de/docs/2014/8852/vts_8852_13227.pdf).
  - [11] A. Biere, M. Fleury, N. Froleys, M. J. H. Heule, The SAT museum, in: M. Järvisalo, D. L. Berre (Eds.), Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023, CEUR Workshop Proceedings, CEUR-WS.org, 2023, pp. 72–87. URL: <https://ceur-ws.org/Vol-3545/paper6.pdf>.
  - [12] A. Biere, K. Fazekas, M. Fleury, M. Heisinger, CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020, in: T. Balyo, N. Froleys, M. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions, volume B-2020-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2020, pp. 51–53.
  - [13] A. Balint, A. Biere, A. Fröhlich, U. Schöning, Improving implementation of SLS solvers for SAT and new heuristics for k-sat with long clauses, in: C. Sinz, U. Egly (Eds.), Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings, Lecture Notes in Computer Science, Springer, 2014, pp. 302–316. URL: [https://doi.org/10.1007/978-3-319-09284-3\\_23](https://doi.org/10.1007/978-3-319-09284-3_23). doi:10.1007/978-3-319-09284-3\_23.
  - [14] I. P. Gent, Optimal implementation of watched literals and more general techniques, J. Artif. Intell. Res. 48 (2013) 231–251. URL: <https://doi.org/10.1613/jair.4016>. doi:10.1613/JAIR.4016.
  - [15] F. Pollitt, M. Fleury, K. Fazekas, N. Froleys, A. Schidler, D. Schreiber, A. Biere, CaDiCaL 3.0, in: 29th International Conference on Theory and Applications of Satisfiability Testing, SAT 2026, 2026. To appear.
  - [16] A. Biere, SAT competition benchmarks 2002 - 2024 (version 1), <https://doi.org/10.5281/zenodo.15125952>, 2025. URL: <https://doi.org/10.5281/zenodo.15125952>. doi:10.5281/ZENODO.15125952, accessed on 2026-05-11.