Revisiting Clause Vivification

Florian Pollitt[1], Mathias Fleury[1], Armin Biere[1], Marijn Heule[2], Karem Sakallah[3], Jiawei Chen[3] and Yonathan Fisseha[3]

[1]University of Freiburg
[2]Carnegie Mellon University
[3]University of Michigan

Abstract
We explain in detail how we reimplemented vivification in our award winning solvers and then, focusing on Kissat, report on experiments on an interesting scalable factoring benchmark suite which helped us to find and remove a subtle performance regression in the vivification code.

Keywords
SAT solving, inprocessing, simplification, vivification

1. Introduction

While SAT solvers spend most of their time running the conflict-driven clause learning (CDCL) procedure [1], all winners of the SAT Competition since 2014 interleave this "search" procedure with other procedures globally transforming the formula, i.e., techniques collectively referred to as *inprocessing* [2]. They aim either to simplify the formula in ways that CDCL cannot or help it to run faster.

There are different types of inprocessing techniques. Some like variable elimination [3] and variable addition [4] do not preserve models, but only *equisatisfiability* (requiring models to be fixed). Other techniques reduce duplicated information over variables, like equivalent literal substitution (ELS) [5]. Finally, many different techniques remove duplicated information within clauses, including subsumption or self-subsuming resolution. They are based on combining two clauses, either to show that one subsumes the other or to generate a shorter subsuming clause by resolving the two clauses.

The focus of this work is one of the key inprocessing techniques, referred to as *distillation* [6], which also has been independently discovered and described as *vivification* [7]. We adopt the term "vivification" for this paper, in line with more recent literature [8, 9]. This technique generalizes self-subsuming resolution by utilizing multiple clauses to produce a shorter subsuming clause, rather than just resolving two. The process leverages the optimized unit propagation component of a SAT solver. In particular it is useful to simplify and remove learned "glue clauses" [10] which otherwise are kept indefinitely.

The vivification algorithm must differentiate among various scenarios involving the detection of subsumed clauses, their deletion, or their strengthening (*aka.* shrinking). We discuss how vivification has been implemented in our solvers, Kissat [11], since 2023, and in CaDiCaL (part of version 2.2-rc2) both with respect to how exactly clauses are "vivified" and vivification is scheduled.

While working on a new scalable family of benchmarks *factoring* (Sec. 3), based on proving that no factorization of a prime number is possible, we observed an interesting performance regression in Kissat. After quite some debugging effort, we noticed that fewer clauses were deleted, which we then traced down to keeping more clauses during vivification. This lead to a solver slowdown accumulating over time, yielding much worse performance on these remarkable benchmarks.

16th Pragmatics of SAT International Workshop 2025 (POS'25)
pollittf@cs.uni-freiburg.de (F. Pollitt); fleury@cs.uni-freiburg.de (M. Fleury); fleury@cs.uni-freiburg.de (A. Biere); marijn@cmu.edu (M. Heule); karem@umich.edu (K. Sakallah); chenjw@umich.edu (J. Chen); yonathan@umich.edu (Y. Fisseha)
0009-0001-4337-6919 (F. Pollitt); 0000-0002-1705-3083 (M. Fleury); 0000-0001-7170-9242 (A. Biere); 0000-0002-5587-8801 (M. Heule); 0000-0002-5819-9089 (K. Sakallah)

In this paper, we discuss our new vivification algorithm implemented in both CaDiCaL and Kissat and how it is scheduled. A major difference to our original vivification implementation in CaDiCaL [12] in 2018 is that the old version of vivification first worked on irredundant (original) clauses before vivifying redundant ones. Further, now, overall scheduling as well as the time-budget allocated to vivification is based on ticks (as a deterministic proxy of running time – similar to Knuth's "mems" [13]), and total vivification effort is explicitly split between clauses with different clause quality, i.e., redundant tier-1, tier-2, tier-3, and irredundant clauses (Sec. 4). We further present how our new vivification algorithm detects subsumed clauses on-the-fly, but otherwise mostly focus on clause deletion (Sec. 5).

Our experiments (Sec. 6) show that considerable run-time variation can be observed for different choices of removing or keeping a redundant clause during vivification. We also show that vivification candidates are frequently subsumed. Still, the most common case of successful vivification are shorter clauses as well as removing clauses due to finding an implied literal. Finally, we show that ticks-based scheduling and limiting vivification effort is effective across different benchmarks.

2. Vivification Algorithm

For background on SAT, we refer to the *Handbook of Satisfiability* [14]. Vivification relies mostly on unit propagation of literals and on a dedicated conflict analysis similar to "analyze-final" in MiniSat [15] producing decision-only learned clauses. For related work on pre- and inprocessing (simplification before running and during running CDCL), we also refer to the corresponding chapter of the Handbook [14].

Given a candidate clause the idea of vivification [6, 7] is to iteratively assume all its literals to be *false*. Between each assumption / decision, the solver fully propagates all other clauses, ignoring the candidate clause. Otherwise, the candidate clause would propagate the last unset literal.

As in CDCL (actually as in DPLL already), if a conflict is found with the decisions $\neg\ell_1, \neg\ell_2, \ldots, \neg\ell_k$, we know that the clause $D := \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$ is entailed by the formula. Therefore, it can be used to strengthen the candidate clause C as follows:

- if the clause D is a *strict* subset of C, then D subsumes C, so we can *strengthen* the candidate clause C by simply replacing C with the shorter clause D (also called "*shrinking*" C);
- similarly, if the negation of one literal of C is propagated instead before being assigned as decision, we can remove it from C, i.e., also strengthening the candidate clause;
- otherwise, if a literal ℓ_i of C is propagated positively, then ℓ_i is implied, we assume that the propagation power of C is covered by others, it is redundant and can be removed.

Vivification had minor impact in 2007 and 2008, when it was originally described, but started to shine in 2017 [9, 8], where a solver with a new variant of vivification won the SAT Competition 2017. The important idea was that vivification should not only be applied to irredundant (equisatisfiable to the original) clauses but also to redundant (learned) clauses, i.e., during inprocessing. In contrast to our implementation and description of vivification above, Li et al. [8] propose to use a dedicated but more general conflict analysis, which potentially can learn a completely different new clause, instead of focusing on decision-only learned clauses, i.e., sub-sets of the candidate clause, as we do.

Vivification was subsequently implemented in various solvers including Glucose (in the version that entered the 2018 SAT Competition as the first and currently only inprocessing technique implemented in Glucose [16]) and CaDiCaL [12] in 2018.

3. Factoring Benchmarks

We created a new set of unsatisfiable benchmarks based on factoring prime numbers available at https://github.com/m-fleury/sat-factoring-generator. For a fixed prime number (represented by a bit vector of an appropriate bitwidth), we assert that there are two numbers (greater than one) whose product generates said prime number (Alg. 1). We then simply use Bitwuzla [17] to bit-blast this SMT-LIB bit vector problem into CNF.

```
generate-factoring-smt (prime, bitwidth)     // C++ like output stream with "<<"
    << "(set-info :smt-lib-version 2.6)"
    << "(set-logic QF_BV)"
    << "(declare-fun a () " << bv-type (bitwidth) << ")"   // SMT-LIB constants for inputs and output
    << "(declare-fun c () " << bv-type (bitwidth) << ")"
    << "(declare-fun d () " << bv-type (bitwidth) << ")"
    << "(assert (= a (bvmul c d)))"
    << "(assert (= a #b" << binary (prime, bitwidth) << "))"   // fix the output to "prime"
    << "(assert (not (= c #b" << binary (1, bitwidth) << ")))"   // avoid "one" as one of the factors
    << "(assert (not (= d #b" << binary (1, bitwidth) << ")))"
    << "(assert (not (bvumulo c d)))"   // ensure no overflow during multiplication
    << "(check-sat)"
    << "(exit)"
```

Algorithm 1: Generating SMT Formula

```
create-benchmarks (lower-bitwidth, upper-bitwidth, primes-per-bitwidth)
    for current-bitwidth from lower-bitwidth to higher-bitwidth
        low = (1 << current-bitwidth)     // "<<" = bit-shifting
        high = (1 << (current-bitwidth + 1))
        increment = (high - low) / primes-per-bitwidth     // (non-random) uniform distribution
        for k from 1 to primes-per-bitwidth
            lower-limit = (1 << current-bitwidth) + increment * (k - 1)
            upper-limit = (1 << current-bitwidth) + increment * k
            prime = find-smallest-prime-between (lower-limit, upper-limit)
            if prime generate-factoring-smt (prime, current-bitwidth)
```

Algorithm 2: Generating Factoring Family

To obtain a benchmark set scaling in size and difficulty to solve, we generate a fixed number of primes for each bitwidth (primes-per-bitwidth), uniformly distributed in the entire range which can be expressed by this number of bits, forcing the most-significant bit to true (Alg. 2). To our surprise (see Fig. 1), this actually created a remarkable family of unsatisfiable benchmarks scaling very nicely: the runtime increases slowly and predictably, and it is easy to produce more benchmarks by increasing the number of generated instances for each bitwidth, as well as harder benchmarks with larger bit-width.

We attempted to produce scalable satisfiable benchmarks too by taking as starting point the product of two prime numbers (prime numbers that are close to each other). However, the generated problems turned out to be too easy (even for large numbers). A related benchmark set of satisfiable instances was evaluated in [18] using [19] to generate CNF, while we rely on SMT-based bit blasting.

4. Scheduling and Ordering

There are three major questions when implementing vivification:

- How often to vivify?
- Which clauses should be vivified?
- In which order should the literals of a clause be assumed?

We answer these questions on an abstract level and dive into some additional details below. The next Section 5 provides concrete pseudo-code with a more detailed discussion, high-lighting specific

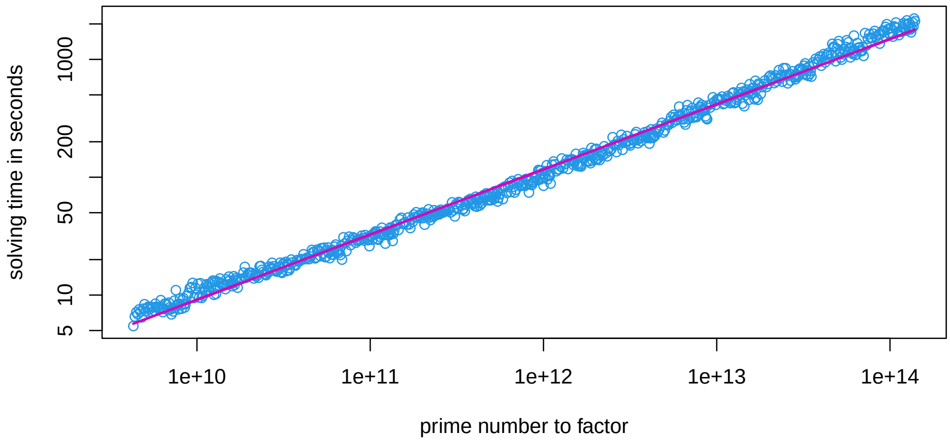

Figure 1: Scaling of the factoring benchmarks (x-axis logarithmic prime numbers, y-axis solving time in seconds).

implementation aspects. Ultimately the reader is invited to explore the actual implementation in Kissat or CaDiCaL (available for instance on GitHub[1]).

How often? How often vivification should be scheduled could be considered to be part of the "black art" of SAT solver design. In CaDiCaL and Kissat, probing (including vivification) is scheduled in increasing $\mathcal{O}(n \log n)$ conflict intervals. The duration of running one vivification round during probing inprocessing is limited by the number of "ticks" taken during propagation while vivifying in relation to the number of ticks consumed by CDCL search since the last time vivification was run. Ticks refer to an estimation of memory access and are used as a deterministic proxy to actual time. For instance visiting a clause during propagation counts as one tick. Ticks correlate well with running time, but still allow fully deterministic behavior of the solver. Limiting time spent in vivification is necessary, as unbounded vivification until completion is too costly on most instances.

Which clauses? Li et al. [8] advocate to vivify only clauses with low LBD [10], because larger LBD clauses are expected to be less relevant and more expensive to vivify. Both CaDiCaL (2.2-rc2) and Kissat work differently: they determine a fixed budget of ticks spent on each kind of clauses; the budget is split between irredundant, tier-1, tier-2, and tier-3 clauses, where the tiers are defined by fixed (or dynamically calculated) LBD thresholds. Therefore, fewer tier-3 (high LBD) clauses are vivified achieving a similar limiting effect, but without completely disregarding them.

Li et al. [8] further suggest re-vivifying a clause only once when its LBD has decreased twice, in order to avoid spending too much time in vivification. We do not explicitly limit our vivification procedure in this way, but prioritize clauses that could not be vivified in the previous inprocessing round (as the corresponding ticks budget was exhausted). As our solvers rarely manage to vivify all clauses, this achieves a similar effect of not retrying a clause before the formula changed considerably.

Initially we vivified first tier-3, followed by tier-2, tier-1 and finally irredundant clauses. The argument was that vivification might remove redundant clauses and potentially promote shrunken clauses to a smaller tier, e.g., larger clauses from tier-2 could be shrunken to size 3 and thus become part of tier-1. It seemed natural to schedule them again immediately during the same vivification round.

However, in practice there are usually fewer tier-1 than tier-2 followed by tier-3 clauses and many more irredundant clauses. If vivification completes vivification of one tier with fewer clauses it is beneficial to use the remaining vivification ticks budget for the next tier with more clauses. Otherwise the ticks budget is wasted. As consequence of this argument, our latest version of vivification first vivifies tier-1, then tier-2, followed by tier-3 clauses, and finally irredundant clauses.

[1] https://github.com/arminbiere/kissat/blob/master/src/vivify.c and https://github.com/arminbiere/cadical/blob/rel-2.2.0-rc2/src/vivify.cpp

What order? Finally, the question remains in which order the decisions should be picked, when vivifying a candidate clause. Instead of choosing a heuristic based on the best expected result, CaDiCaL and Kissat build an implicit prefix tree (or trie) of the clauses, which allows us to reuse decisions and propagations over different candidate clauses, i.e., we do not necessarily backtrack between vivification of two candidates. The order of decisions follows the prefix tree, maximizing reusability (throughput).

In fact, this simulates the trie-data-structure in distillation [6], without the need to explicitly transform the CNF into a trie. Reusing propagation effort has also been the target of other related techniques [20, 21]. To maximize the effect, the candidate clause (and prefix tree) is sorted by literal occurrences (number of positive or negative occurrences). Note, however, that CaDiCaL until version 2.1 used a weighted occurrence count similar to the Jeroslow-Wang heuristic [22] instead.

Further, sorting the candidate stack lexicographically (also with respect to literal occurrence – taking into account separately positive and negative occurrences of variables) would maximize reusability, however this can be too costly. Instead, we can only consider the first literal (or first two literals in CaDiCaL) in each clause for sorting the stack, which allows us to use the faster radix sort (faster compared to C++'s default stable `std::stable_sort`).

Our experiments show, that building a prefix tree saves roughly 30% of decisions (median for both the factoring benchmarks as well as the SAT Competition 2024 benchmarks is 33%). This also saves propagations and makes it possible to vivify many more clauses within the same ticks budget.

Further discussion. A minor subtlety is that the propagation of vivification should ignore the candidate clause but should *still* update it to make sure that the watched literals invariants are valid. Otherwise, propagations might be missed for subsequent clauses. Similarly, if the candidate clause was propagating, the solver cannot reuse the last decision level (without loosing propagations). Not doing so does not lead to issues in the subsequent CDCL loops, because at the end of vivification, the solver backtracks to decision level zero, implicitly restoring watch invariants of the ignored clauses.

To reiterate on sorting candidate clauses, it is of course important that sorting is fast. Considering only one or two literals gives big speed-ups. However, one also looses certain potential advantages compared to sorting lexicographically as was done in CaDiCaL previously. Actually the Kissat version considered in the experiments uses a mixed strategy of lexicographic sorting for redundant clauses, i.e., tier-1, tier-2, and tier-3 clauses, and sorting irredundant clauses with respect to a single most occurring literal in a clause (the considered fixed number of literals is a compile-time parameter). Storing these one or two literals separately has another advantage: we keep the clauses watched without reordering it, leading to hopefully good watches and no very long watch lists which you would need if you put literals appearing the most often first.

It is important to use a stable sorting algorithm for determinism and debugging (which is substantially slower than default non-stable algorithms). When sorting with respect to the complete prefix tree, uniquely determining the position of each clause, it is possible to use a non-stable algorithm. The only issue in this regard is if the same clause occurs multiple times. This however can be detected cheaply and solved by removing one of them (`vivifyflush` option in CaDiCaL, activated by default).

It is also possible to detect subsumed clauses during candidate sorting: when comparing clauses with a shared prefix and one clause is identical to that shared prefix the longer clause is subsumed by the shorter. This technique requires sorting each individual clause first and updating watches accordingly. We did not see any performance improvements in CaDiCaL due to employing this subsumption technique.

Other solvers use different sorting scheme: For instance Glucose 4.2.1 [16] does not sort the candidate clauses but simply takes the current order. Alternatively CryptoMiniSat [23] uses either the number of occurrences or the order induced by the VSIDS decision heuristic [24]. The ParaFROST solver [25] vivifies tier-1 and tier-2 clauses separately but within one tier sorts candidate clauses by literals.

vivify (CNF F) // CNF updated in place / passed by reference

1 ticks-budget = search-ticks-since-last-vivification$_{\text{stats}}$ $\times$ relative-vivification-effort$_{\text{option}}$
2 tier-1-budget = ticks-budget $\times$ relative-tier-1-budget$_{\text{option}}$
3 tier-2-budget = ticks-budget $\times$ relative-tier-2-budget$_{\text{option}}$
4 tier-3-budget = ticks-budget $\times$ relative-tier-3-budget$_{\text{option}}$
5 irredundant-budget = ticks-budget $\times$ relative-irredundant-budget$_{\text{option}}$
6 remaining-ticks = *vivify-tier*(F, tier-1 clauses of F, tier1-budget)
7 remaining-ticks = *vivify-tier*(F, tier-2 clauses of F, tier2-budget $+$ remaining-ticks)
8 remaining-ticks = *vivify-tier*(F, tier-3 clauses of F, tier3-budget $+$ remaining-ticks)
9 *vivify-tier*(F, irredundant clauses of F, irredundant-budget $+$ remaining-ticks)

Algorithm 3: Main vivification inprocessing algorithm scheduled from CDCL loop.

vivify-tier (CNF F, CNF G, ticks-budget) // update subset of clauses in original CNF in place

1 limit = ticks$_{\text{stats}}$ $+$ ticks-budget // global variable "ticks$_{\text{stats}}$" updated during propagation
2 sort literals in clauses $C \in G$ by number of occurrences (more occurrences first)
3 let G_1 be the sub-set of clauses of G which were *not* tried during vivification last time
4 let $G_2 = G \backslash G_1$ // new clauses or clauses already tried last time
5 sort G_1 and separately G_2 lexicographically *w.r.t.* literal occurrences (more first)
// decision level set to zero at this point
6 **for all** clauses C in the sequence G_1, G_2 sorted as in line 5 as long ticks$_{\text{stats}}$ < limit
7 **if** *vivify-clause* (F, C) **then** increment vivified$_{\text{stats}}$
8 backtrack to decision level zero
9 **if** ticks$_{\text{stats}}$ $>$ limit **return** 0 // incomplete – remember untried clauses
10 **return** limit $-$ ticks$_{\text{stats}}$ // return unused ticks budget – no untried clauses remembered

Algorithm 4: Vivifying one "tier" of clauses under a given ticks budget.

5. Revisited Algorithm

The revisited vivification algorithm *vivify* in Alg. 3 is called in Kissat from the *probing* inprocessing procedure, after clausal congruence closure [26], equivalent literal substitution (ELS) [5] and binary backbone computation [27]. It is followed by bounded clausal SAT sweeping [28], another round of ELS, transitive reduction of the binary implication graph [29], a second round of binary backbone computation and finally bounded variable addition (BVA) [4]. Probing is called in increasing conflict intervals from the main CDCL search loop, i.e., the conflict interval before it is rerun after its n-th invocation is set to "probe-interval$_{\text{option}}$ $\cdot\, n \cdot \log_{10}(n+9)$" for some base conflict interval probe-interval$_{\text{option}}$ (default is 100 conflicts, which is also the initial conflict interval before the first probing round).

The ticks-budget computed at (line 1 in Alg. 3) is then split (line 2-4) into a fixed fraction for each of the tiers (according to run-time options). We hand over any remaining-ticks of the budget to the next tier, with vivification of irredundant clauses last (line 9), as it is usually the most costly tier to run vivification until completion (at least initially, for bigger formulas). The four calls to *vivify-tier* in Alg. 3 (line 6-9) consider only specific tier clauses as candidates to vivify as second argument. We formally also give a reference to the global formula F as argument, to emphasize that Boolean constraint propagation (unit-resolution) is always performed on the whole formula (line 12 in Alg. 5).

The *vivify-tier* function is shown in Alg. 4. It first, as already discussed in the previous Sect. 4, determines a total ticks limit on propagation effort for all considered candidate clauses G, based on the given budget (line 1). Then all candidates have their literals sorted by the number of occurrences (line 2) in order to subsequently (line 5) allow sorting the clauses lexicographically w.r.t. the literal occurrences. Before the candidates G are split into two sets G_1 and G_2 (line 3-4) prioritizing left-over

```
    vivify-clause (CNF F, clause C)    // update F and C in place
1   mark C as having been tried    // puts it in G₂ next time
2   let C = ℓ₁ ∨ · · · ∨ ℓₙ sorted by number of occurrences (more occurrences first)
3   find maximal m such that ℓᵢ is assigned to false at decision level i for all i < m  // reuse trail
4   if m > 0 and decision level larger than m − 1 backtrack to decision level m − 1
5   add m − 1 to both probes_stats and reused_stats    // reused m decisions / probes
6   literal implied = ⊥,  clause conflict = ⊥    // initialize both to be undefined denoted as "⊥"
7   for i = m . . . n as long conflict = ⊥    // and implied = ⊥
8      if ℓᵢ is assigned to false continue
9      if ℓᵢ is assigned to true then implied = ℓᵢ and break
10     increase decision level and assign ℓᵢ to false,   increment probes_stats
11     // temporarily disable propagation over C, i.e., C is simply skipped during propagation
12     conflict = propagate (F, C)    // update global assignment and ticks_stats
    // now we have either implied ≠ ⊥, conflict ≠ ⊥, or C is falsified by the current assignment
13  (subsuming, learned, irredundant) = vivify-analyze (C, conflict, implied)
14  if subsuming ≠ ⊥
15     remove C from F,   increment subsumed_stats   and   return true
       // . . . and need to make "subsuming" irredundant if it was redundant but C not
16  if |learned| < |C|    // actually "learned ⊂ C" as it is a decision learned clause
17     replace C in F by learned,   increment shrunken_stats   and   return true
18  if implied ≠ ⊥ and C redundant
19     // regression version "without-implied" would only return false but the "default" version has:
20     remove C from F,   increment implied_stats   and   return true
21  conflicting  =  conflict ≠ ⊥ ∨ implied ≠ ⊥
22  if conflicting and C irredundant as well as analysis resolved only irredundant clauses
23     remove C from F,   increment asymmetric_stats   and   return true
24  if implied ≠ ⊥ and vivify-instantiate (F, C, ℓₙ)    // C falsified at decision level n
25     remove ℓₙ from C,   increment instantiated_stats   and   return true
26  return false
```

Algorithm 5: Vivifiying a single clause candidate.

candidates not tried last time vivification was run. This prioritization makes sure that all clauses of a tier are vivified in a round-robin fashion, i.e., clauses are tried at least once before being attempted to be vivified again. Note, that this scheme of vivification by tiers in combination with separate but fixed relative ticks budgets allows to complete vivification of tiers with fewer clauses earlier and more often.

Alg. 5 gives a high-level overview on our revisited clause vivification algorithm *vivify-clause* for vivifying a single candidate clause. It maintains a global assignment that is not reset between different candidates, i.e., subsequent calls to this function. Instead it reuses the trail as much as possible, in order to reduce the number of necessary propagations to vivify the candidate.

The propagation procedure called at line 12 ignores the candidate clause C given as argument and further updates the global $\text{ticks}_{\text{stats}}$ statistics counter. It approximates non-local cache line access, similar to Knuth's "mems" statistics (counting the number of pointer dereferences instead). Counting ticks is only an approximation and the result of profiling runs and inspecting the code, i.e., the programmer adds instructions which increase the ticks counter whenever the program reaches a point, where a non-local memory access is expected. Access to the same cache line (e.g., propagating many virtual binary clauses for the same literal) is counted only once. Even though less automatic to implement, ticks are more precise than mems, i.e., correlate better with actual running time.

The function *vivify-analyze* (line 13) is a standard conflict analysis routine, similar to "analyze-final" in MiniSat. It returns a decision-only learned clause, which is a subset of the negations of decisions, that lead to the given conflict, derive the implied literal or (if both are undefined) to falsify the given candidate clause. It further checks on-the-fly whether any of the resolved clauses subsumes the candidate clause. In this case it aborts the analysis and returns the subsuming clause. Otherwise it returns the learned clause and determines whether all resolved clauses in deriving the learned clause are irredundant.

The function *vivify-instantiate* (line 24) originates from a CaDiCaL hack CaDiCaL _vivinst [30] submitted to the SAT Competition 2023, taking first place in the hack track and third place in the main track on satisfiable instances. After all the literals in the candidate clause are assigned and the clause could not be subsumed nor shrunken, i.e., the candidate clause is falsified and all its literals are decisions, we backtrack one level and assign the last literal ℓ_n to *true*. If then propagation fails we know that we can remove ℓ_n. This is similar to *variable instantiation* [31], targeting to remove literals with few occurrences, such that they become pure or can be eliminated by variable elimination later.

What is missing in the high-level description of Alg. 5 is when (if at all) and to which level to backtrack to after shrinking a clause and successful instantiation. If the procedure fails but a conflict was deduced we might need additional backtracking too, as well as explicitly reestablish clause-watching invariants for the candidate clause (as it was skipped during propagation). Further note, that sorting literals within clauses during scheduling in *vivify-tier* (line 2 in Alg. 4) as well as during candidate clause vivification in *vivify-clause* (line 2 in Alg. 5) should be done on copies of the clauses in separate data-structures to keep watch-invariants intact (or as alternatives either reestablish them before propagation or use an approximate short list of literals only – *cf.* Sect. 4).

The procedure has 5 positive cases in which a candidate clause is $\text{vivified}_{\text{stats}}$ (line 7 in Alg. 4) and thus the candidate clause is shrunken (replaced by a strict subset of literals) or removed. These correspond to incrementing in Alg. 5 the statistics counters $\text{subsumed}_{\text{stats}}$ (line 15), $\text{shrunken}_{\text{stats}}$ (line 17), $\text{implied}_{\text{stats}}$ (line 20), $\text{asymmetric}_{\text{stats}}$ (line 23) and $\text{instantiated}_{\text{stats}}$ (line 25) with returning *true*. Otherwise the procedure fails returning *false* (line 26) and keeps the candidate as is. The next section evaluates how often these cases occur in practice (*cf.* Fig. 6 and 7).

6. Experiments

We ran Kissat version sc2024 as submitted to the SAT Competition 2024 on all 400 problems of the SAT Competitions 2023 and 2024 and our 750 factoring benchmarks using the bwForCluster Helix with AMD Milan EPYC 7513 CPUs and a time limit of 5000 seconds. More precisely we used *four* configurations of Kissat: the default configuration, matching the description in Fig. 5 (including removing implied literal candidates - keeping line 18-20 in Alg. 5 as is); versus the keep-implied configuration which keeps implied clauses (returning *false* at line-20 in Alg. 5 without removing the candidate); the discard-both configuration (replacing the condition "implied $\neq \bot$" with "implied $\neq \bot \vee$ conflict $\neq \bot$" at line-18 in Alg. 5 which, if triggered, not only removes clauses with an implied literal but also conflicting candidate clauses); finally the no-vivify configuration which disables vivification completely.

On the 400 problems of the SAT Competitions 2023 and 2024 (Figures 2 and 3), the performance does not differ much, but the default version performs best. Interestingly, it seems that completely deactivating vivification does not make much of a difference on these instances anyhow. However, on our *factoring* benchmarks the difference between the configurations is important (Fig. 4). For these benchmarks, the discard-both configuration performs slightly better than the default configuration.

In an intermediate version of Kissat we accidentally introduced the keep-implied variant, probably, as it might appear that implied literal candidates should be treated the same way as conflicting candidates. Note, that the difference only becomes relevant if a redundant clause candidate is not subsumed nor shrunken during vivification (we reach line 18 in Alg. 3 and a literal is implied or we found a conflict). These factoring benchmarks however revealed, that these two cases should be differentiated.

On competition benchmarks keeping conflicting but discarding implied literal candidates works best (default), while for the factoring benchmarks, discarding implied literal candidates is a must (default vs. keep-implied even though discard-both is even better). After we observed this regression empirically, finding the root cause was rather difficult. Only after realizing that the regression version would keep for long running benchmarks many more learned clauses, it became clear that we should search for code where clauses are removed in one version and kept in the other. Furthermore, it was the first time we observed such an almost linear line in a run-time scatter-plot of SAT solvers (*cf.* Fig. 5, right plot).

To give more insight into the impact of vivification and how often specific cases of Alg. 5 occur we have extracted statistics for the default configuration. Figure 6 shows the ratio and amount of time the solver spends in vivification with respect to search and overall time on the factoring benchmarks. These quantities scale very nicely on this benchmark set, as is evidenced by the fact that we can sort by just one criterion, always plotting the same benchmark in one cross-section. Looking at the vivification statistics (*cf.* end of Sect. 5), the percentages between the different cases are almost constant, only $\text{asymmetric}_{\text{stats}}$ seems to tail off for the harder instances. The $\text{implied}_{\text{stats}}$ clauses make out almost 40% of all positive vivified clauses, explaining the regression we can observe in the previous plots. These statistics on the SAT Competition 2024 benchmarks vary more but similar trends can be observed (Fig. 8, 9). A difference overall seems to be the number of subsumed $\text{clauses}_{\text{stats}}$, which is similar to $\text{shrunken}_{\text{stats}}$ and $\text{implied}_{\text{stats}}$ on these instances, compared to about half for the factoring benchmarks.

7. Conclusion

We gave a deep dive into technical details of the implementation of our latest version of vivification in our SAT solvers Kissat and CaDiCaL. Beside explaining the novel feature of on-the-fly subsumption we further reported on an interesting performance regression we observed due to a subtle difference of two versions of vivification, differing only in the choice of removing a vivification candidate with an implied literal or not. Keeping them lead to a regression. It was detected on a new set of factoring benchmark which by itself is quite remarkable as it yields very smooth run-time scaling behavior. Our experiments reveal that all the considered special cases during vivification do occur in practice, but most frequently, successfully vivified clauses are either shrunken, subsumed or have an implied literal, while successful instantiation or asymmetric literal elimination in irredundant clauses occurs rarely.

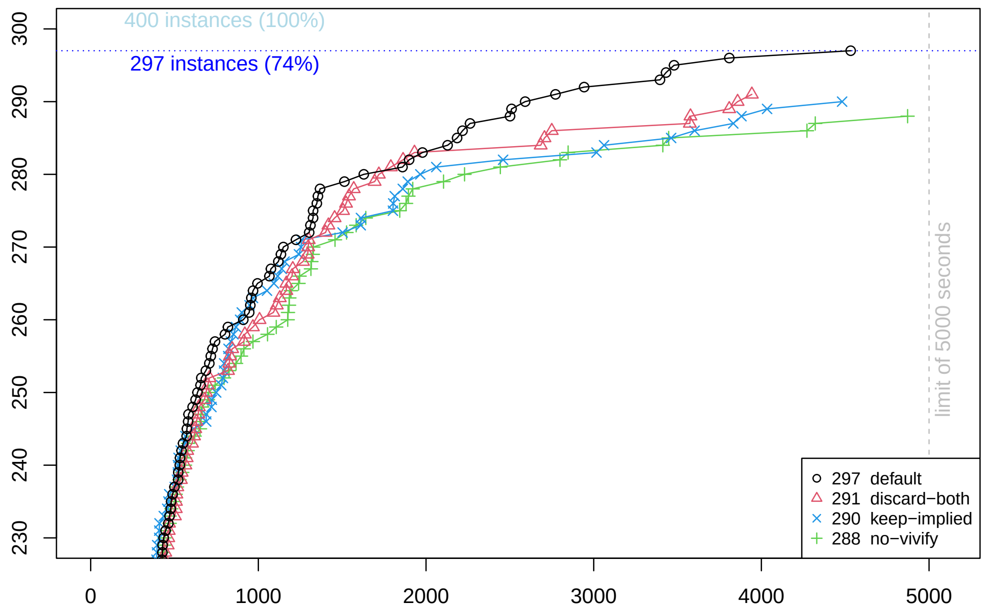

Figure 2: Performance of various Kissat configuration as CDF (number of solved instances on the y-axis versus the time-limit on the x-axis) on all 400 instances of the SAT Competition 2023. The default configuration is best.

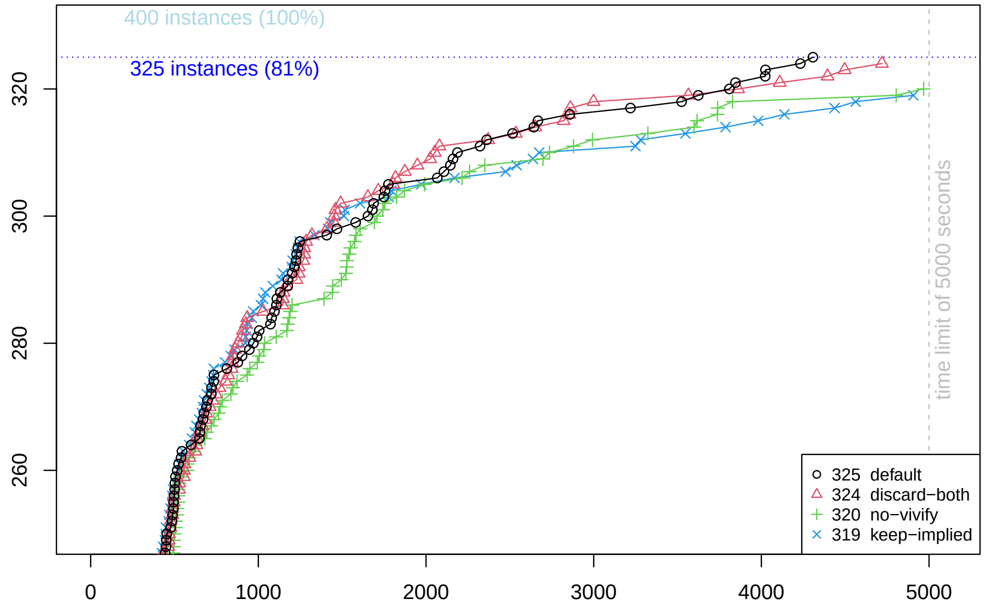

Figure 3: Performance of various KISSAT configuration on all 400 instances of the SAT Competition 2024. The performance difference between the configurations is limited, but the default configuration works best. Axis are as for the SAT competition and in Fig. 2, i.e., solved instances on the y-axis versus the time limit on the x-axis.

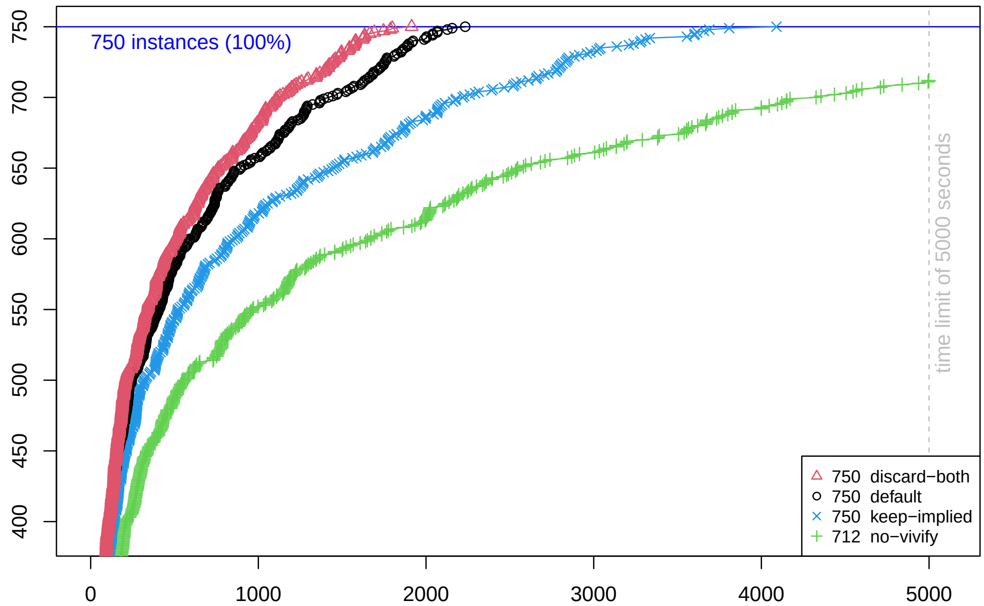

Figure 4: Performance of various KISSAT configuration on our new 750 factoring benchmarks described in Sect. 3. Again axis follow the common practice how results of the SAT competition are presented (as in Fig. 2+3 too).

Acknowledgments

This work was supported in part by the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, and by a gift from Intel Corporation.

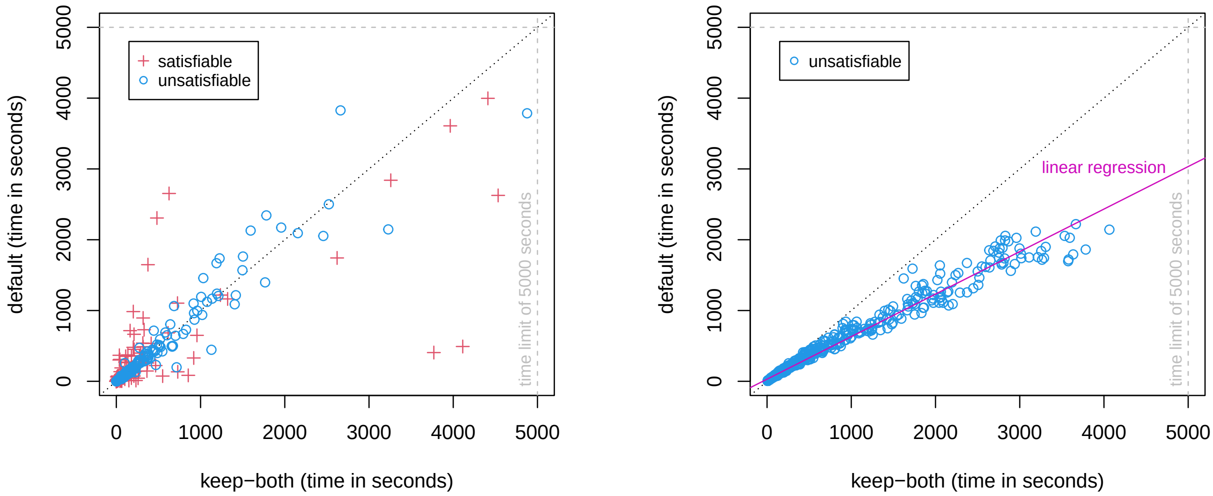

Figure 5: Comparing the default version of Kissat which keeps conflicting redundant clauses during vivification but discards those with an implied literal with the regression version which keeps both clauses (keep-both) on SAT Competition 2024 benchmarks on the left and the factoring benchmarks on the right.

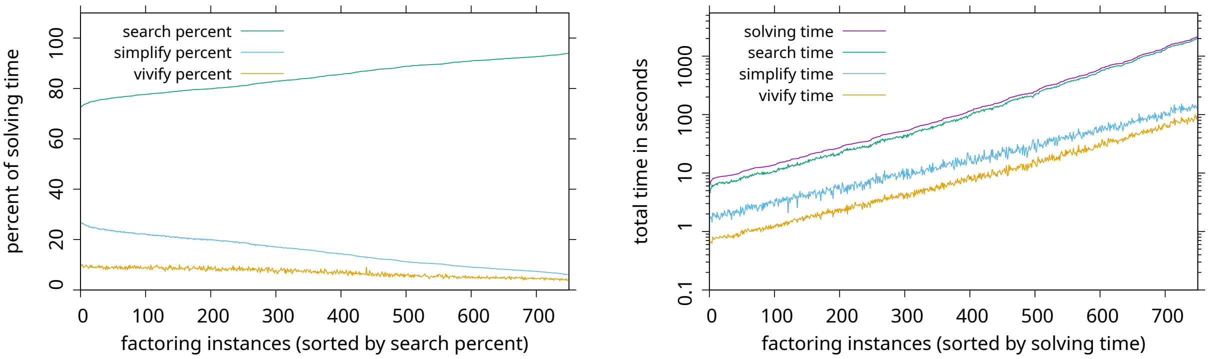

Figure 6: Solving statistics of the default version of Kissat on the factoring benchmarks (*cf.* Fig. 1).

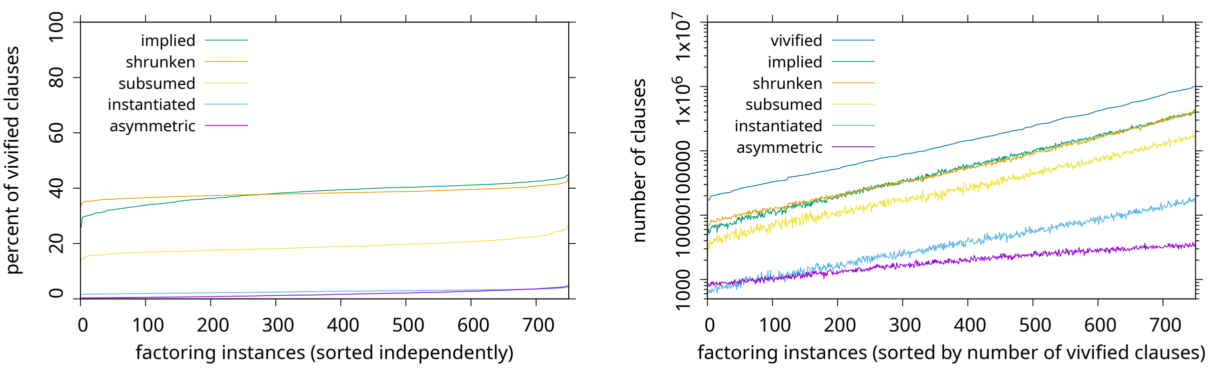

Figure 7: Vivification statistics of the default version of Kissat on the factoring benchmarks (*cf.* Sect. 5).

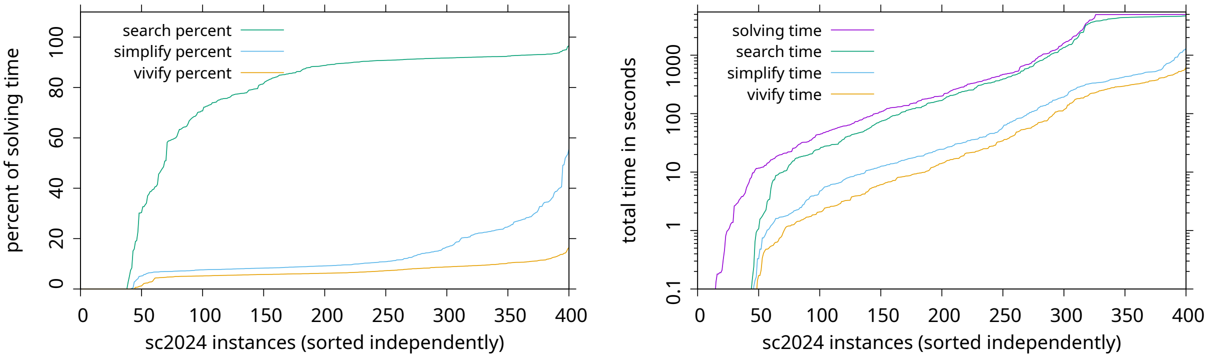

Figure 8: Solving statistics of the default version of KISSAT on the SAT Competition 2024 benchmarks. Search time starts only after the preprocessing and hence can be 0 for some instances.

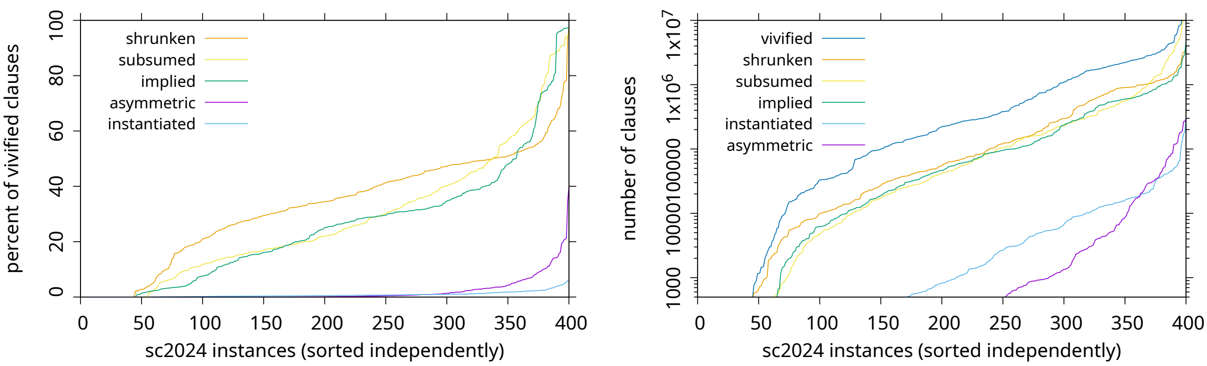

Figure 9: Vivification statistics of the default version of KISSAT on the SAT Competition 2024 benchmarks.

References

[1] J. Marques-Silva, I. Lynce, S. Malik, Conflict-driven clause learning SAT solvers, in: 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, pp. 133–182. doi:10.3233/FAIA200987.

[2] M. Järvisalo, M. Heule, A. Biere, Inprocessing rules, in: B. Gramlich, D. Miller, U. Sattler (Eds.), Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings, volume 7364 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 355–370. doi:10.1007/978-3-642-31365-3_28.

[3] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: F. Bacchus, T. Walsh (Eds.), Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings, volume 3569 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 61–75.

[4] N. Manthey, M. Heule, A. Biere, Automated reencoding of boolean formulas, in: A. Biere, A. Nahir, T. E. J. Vos (Eds.), Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers, volume 7857 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 102–117. doi:10.1007/978-3-642-39611-3_14.

[5] A. V. Gelder, Y. K. Tsuji, Satisfiability testing with more reasoning and less guessing, in: D. S. Johnson, M. A. Trick (Eds.), Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS/AMS, 1993, pp. 559–586. doi:10.1090/DIMACS/026/27.

[6] H. Han, F. Somenzi, Alembic: An efficient algorithm for CNF preprocessing, in: Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007, IEEE, 2007, pp. 582–587. doi:10.1145/1278480.1278628.

[7] C. Piette, Y. Hamadi, L. Sais, Vivifying propositional clausal formulae, in: M. Ghallab, C. D. Spyropoulos, N. Fakotakis, N. M. Avouris (Eds.), ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, volume 178 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2008, pp. 525–529. doi:10.3233/978-1-58603-891-5-525.

[8] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, Y. Li, Clause vivification by unit propagation in CDCL SAT solvers, Artif. Intell. 279 (2020). doi:10.1016/J.ARTINT.2019.103197.

[9] M. Luo, C. Li, F. Xiao, F. Manyà, Z. Lü, An effective learnt clause minimization approach for CDCL SAT solvers, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 703–711. doi:10.24963/IJCAI.2017/98.

[10] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: C. Boutilier (Ed.), IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009, pp. 399–404. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

[11] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, F. Pollitt, CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024, in: M. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions, volume B-2024-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2024, pp. 8–10.

[12] A. Biere, CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018, in: M. Heule, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions, volume B-2018-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2018, pp. 13–14.

[13] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming), Addison-Wesley Professional, 2006.

[14] A. Biere, M. Järvisalo, B. Kiesl, Preprocessing in SAT solving, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 2nd edition ed., IOS Press, 2021, pp. 391 – 435.
[15] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, volume 2919 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 502–518. doi:10.1007/978-3-540-24605-3_37.
[16] G. Audemard, L. Simon, Glucose and Syrup: Nine years in the SAT competitions, in: M. Heule, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions, volume B-2018-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2018, pp. 24–25.
[17] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II, volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17. doi:10.1007/978-3-031-37703-7_1.
[18] M. L. Ginsberg, Satsisfiability and systematicity, J. Artif. Intell. Res. 53 (2015) 497–540. doi:10.1613/JAIR.4684.
[19] P. Purdom, A. Sabry, CNF generator for factoring problems, 2005. URL: https://cgi.luddy.indiana.edu/~sabry/cnf.html.
[20] P. van der Tak, A. Ramos, M. Heule, Reusing the assignment trail in CDCL solvers, J. Satisf. Boolean Model. Comput. 7 (2011) 133–138. doi:10.3233/SAT190082.
[21] M. Heule, M. Järvisalo, A. Biere, Revisiting hyper binary resolution, in: C. P. Gomes, M. Sellmann (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings, volume 7874 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 77–93. URL: https://doi.org/10.1007/978-3-642-38171-3_6.
[22] R. G. Jeroslow, J. Wang, Solving propositional satisfiability problems, Ann. Math. Artif. Intell. 1 (1990) 167–187. doi:10.1007/BF01531077.
[23] 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, volume 5584 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 244–257. doi:10.1007/978-3-642-02777-2_24.
[24] Y. S. Mahajan, Z. Fu, S. Malik, zChaff 2004: An efficient SAT solver, in: H. H. Hoos, D. G. Mitchell (Eds.), Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers, volume 3542 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 360–375. doi:10.1007/11527695_27.
[25] M. Osama, A. Wijs, GPU acceleration of bounded model checking with parafrost, in: A. Silva, K. R. M. Leino (Eds.), Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II, volume 12760 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 447–460. doi:10.1007/978-3-030-81688-9_21.
[26] A. Biere, K. Fazekas, M. Fleury, N. Froleyks, Clausal congruence closure, in: S. Chakraborty, J. R. Jiang (Eds.), 27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India, volume 305 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 6:1–6:25. doi:10.4230/LIPICS.SAT.2024.6.
[27] N. Froleyks, E. Yu, A. Biere, BIG backbones, in: A. Nadel, K. Y. Rozier (Eds.), Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023, IEEE, 2023, pp. 162–167. doi:10.34727/2023/ISBN.978-3-85448-060-0_24.
[28] A. Biere, K. Fazekas, M. Fleury, N. Froleyks, Clausal equivalence sweeping, in: N. Narodytska, P. Rümmer (Eds.), Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024, IEEE, 2024, pp. 1–6. doi:10.34727/2024/ISBN.978-3-85448-065-5_29.
[29] M. Heule, M. Järvisalo, A. Biere, Clause elimination procedures for CNF formulas, in: C. G.

Fermüller, A. Voronkov (Eds.), 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*, Springer, 2010, pp. 357–371. doi:10.1007/978-3-642-16242-8_26.

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

[31] G. Andersson, P. Bjesse, B. Cook, Z. Hanna, A proof engine approach to solving combinational design automation problems, in: Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002, ACM, 2002, pp. 725–730. doi:10.1145/513918.514101.