Effectiveness of pre- and inprocessing for CDCL-based SAT solving

Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer

Institut für Informatik, Universität zu Köln, Pohligstr. 1, D-50969 Köln, Germany
{wotzlaw, vandergrinten, esp}@informatik.uni-koeln.de

Abstract. Applying pre- and inprocessing techniques to simplify CNF formulas both before and during search can considerably improve the performance of modern SAT solvers. These algorithms mostly aim at reducing the number of clauses, literals, and variables in the formula. However, to be worthwhile, it is necessary that their additional runtime does not exceed the runtime saved during the subsequent SAT solver execution. In this paper we investigate the efficiency and the practicability of selected simplification algorithms for CDCL-based SAT solving. We first analyze them by means of their expected impact on the CNF formula and SAT solving at all. While testing them on real-world and combinatorial SAT instances, we show which techniques and combinations of them yield a desirable speedup and which ones should be avoided.

Keywords: satisfiability, preprocessing, inprocessing, CDCL solvers.

1 Introduction

The satisfiability problem of propositional logic (SAT) has a number of important real-world applications including hardware-verification, software-verification, and combinatorial problems [3]. Despite of being NP-complete, real-world SAT instances can nowadays be solved in an acceptable time by state-of-the-art solvers.

Among all approaches for SAT solving only *conflict-driven clause-learning* (CDCL) solvers [14], an extension of the DPLL procedure, have proved their remarkable efficiency in solving real-world SAT problems, containing often more than 1 million variables and more than 10 million clauses. It has been observed that their performance can be improved if certain simplification techniques are run on the CNF formulas before the actual SAT algorithm starts or during its execution [1,5,8,9,10,11,12,15,17]. Those *preprocessing* and *inprocessing* techniques, respectively, aiming mostly at reducing the number of clauses, literals, and variables of the formula, have become an essential part of the SAT solving tool chain. However, to be worthwhile, it is necessary that their additional runtime does not exceed the runtime saved during the subsequent SAT solver execution. It is a trade-off between the amount of reduction achieved and invested time.

This paper evaluates some selection of promising pre- and inprocessing techniques and combinations of them developed in the recent years. Section 2 begins with preliminaries, describes briefly simplification techniques considered here,

and analyzes them by means of their expected impact on the CNF formula and the CDCL-based SAT solving. Section 3 gives the evaluation of those techniques and their combinations on real-world and combinatorial SAT benchmarks. In Section 4 we conclude our paper and give recommendations, based on the measured effectiveness, for the usage of simplification techniques for SAT solving.

2 Pre- and Inprocessing Techniques

For a Boolean variable $x \in V$, there are two literals, the positive literal x and the negative literal $\overline{x}$. A clause C is a disjunction of different literals over V and a CNF formula a conjunction of clauses. Let $\tau : V \to \{0, 1\}$ be a truth assignment. τ satisfies a literal l iff $\tau(l) = 1$. A clause is satisfied by τ iff τ satisfies any of its literals. τ satisfies formula F iff all clauses of F are satisfied by τ. A formula F is *satisfiable* iff there is at least one truth assignment satisfying it. Two formulas are *logically equivalent* if they are satisfied by exactly the same set of assignments. A clause is a *tautology* if it contains both x and $\overline{x}$ for some variable x.

2.1 Selected Simplification Techniques

We give now a short description of the simplification techniques whose evaluation we present in this paper. For a good overview of the recent developments of pre- and inprocessing techniques for SAT solving we refer to [2].

Subsumption (SUB) is a simple technique trying to eliminate logically redundant clauses from the CNF formula [3]. In its simplest form SUB removes a clause C iff there is a clause D that contains all literals in C as a subset. For instance, for the formula $F = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2)$, SUB will remove the second clause since the first clause is its proper subset. SatELite [5] introduces another subsumption variant. Here by resolving clauses $(\neg x_1 \vee x_2)$ and $(\neg x_1 \vee \neg x_2)$ in F, we obtain the new clause $(\neg x_1)$ subsuming both former clauses. The algorithm trying to resolve two clauses to find a resolvent subsuming both input clauses is called *resolution subsumption* (RSUB), or *self-subsuming resolution* [5], and constitutes another important simplification technique.

Bounded variable elimination (BVE) [5,17] uses the Davis-Putnam procedure [3] to remove variables from the formula. It chooses first a variable and then removes all clauses containing this variable from the formula while adding to the formula all resolvents of those clauses with respect to the variable chosen. To prevent exponential blow-up of the formula, variables are only eliminated if the number of new clauses is less than the number of removed clauses.

We call a clause C *blocked* if there is a literal l in C so that all resolvents between C and any other clause with respect to the variable inducing l are tautologies. Such a blocked clause can be removed from the formula without affecting the satisfiability properties of the formula. The procedure that removes all blocked clauses from the formula is called *blocked clause elimination* (BCE) and was introduced as a preprocessing technique in [11].

Unhiding (UH) [10] is a technique performing a depth first search on the binary implication graph [18] formed by the clauses of length two of the CNF formula. It finds all strongly connected components (SCC) of the graph as well as a subset of the failed literals [3], and transitive edges that can be removed without affecting the satisfiability of the formula. Here all literals represented by the nodes of an SCC are equivalent and can be replaced by a single one. During the search various information is extracted that can be used both for *hidden literal elimination* (HLE) [10] and *hidden tautology elimination* (HTE) [9].

Distillation (DI) is like failed literal elimination [13] a probing based technique. Consider the clause $(l_1 \vee \ldots \vee l_k)$ for a fixed order of its literals. Next assign sequentially all literals one after the other to 0 according to the order chosen and perform *unit propagation* (UP) after setting each literal. This will either lead to a conflict, or there will be a literal l_i that has already been set by UP before it is set to 0 by the procedure. In both cases we can try to shorten the clause by removing one or more of its literals. There are multiple variants of distillation and we refer to [8,15] for more details.

2.2 Properties of Simplification Techniques

We state now some criteria according to which we assess the impact of the algorithms described above on the CNF formula and the SAT solving.

1. **Preservation of unit propagation.** CDCL solvers rely on UP to fix variables while expanding the search tree. It is desirable that simplification techniques preserve UP, i.e., if a variable v can be fixed to a certain value when applying UP to the input formula and a given partial truth assignment τ, then the same variable should be fixed by UP when applied to the simplified formula and the same assignment τ.
2. **Preservation of equivalence.** If an algorithm does not preserve the logical equivalence, it is often essential to construct a satisfying assignment of the input formula from a satisfying assignment of the simplified formula.
3. **Simulation by resolution.** Some applications of SAT solvers rely on resolution refutation proofs for unsatisfiable instances, e.g., partial MAX-SAT solvers based on iterative SAT solving [6]. Hence it should be possible to construct efficiently a resolution refutation of the input formula given a resolution refutation for the formula resulting from the simplification.
4. **Confluence.** We call an algorithm *confluent* if its result does not depend on the order in which variables and clauses of the input formula are inspected. A non-confluent algorithm may need additional heuristics to improve the order in which variables or clauses are processed.
5. **Implementation.** It is desirable that the preprocessing algorithm can be implemented with data structures that are already present in CDCL solvers. Algorithms that do not rely on special data structures may also be better suitable for inprocessing in addition to preprocessing. Some techniques require data structures which for each literal list all clauses containing it. Those so called *literal occurrence lists* are usually not required for the CDCL algorithm and maintaining them would result in a performance penalty.

Table 1. Properties of the simplification techniques (*: modulo variable renaming).

Preprocessing Technique	Preserves UPs	Preserves Equivalence	Requires Occ-Lists
(Resolution-) Subsumption ((R)SUB)	Yes	Yes	Yes
Bounded Variable Elimination (BVE)	–	–	Yes
Blocked Clause Elimination (BCE)	–	–	Yes
Hidden Tautology Elimination (HTE)	–	Yes	–
Hidden Literal Elimination (HLE)	Yes	Yes	–
Unhiding without HLE/HTE (UH)	Yes	Yes*	–
Distillation (DI)	Yes	Yes	–

Table 1 lists properties of the simplification algorithms described in Section 2.1 according to the criteria given above. Additionally, all algorithms can be simulated by resolution whereas none of them, except for BCE, is confluent.

3 Comparative Evaluation

3.1 Experimental Setup

The simplification techniques presented in Section 2 were implemented in our sequential CDCL-based solver *satUZK* [7] having performance comparable with that of MiniSAT 2.2 [4]. All tests were run on a machine with two Intel Xeon E5410 2.33 GHz processors running a 64-bit Linux 2.6.32 with 32GB RAM.

There are two categories of instances that were tested: *application* and *hard combinatorial* instances. Our test suits were formed from a subset of all instances from the SAT Challenge 2012 [16]. First, instances which could be solved with MiniSAT 2.2 on our test machine in between 120 and 600 seconds were selected, resulting in 58 hard combinatorial instances and a large number of application instances, from which we took 60, selected uniformly at random. For each of the 118 CNF formulas five instances were generated at random by permuting the orders of clauses, the orders of variables inside each clause, and the polarities of the variables, resulting in 300 application and 290 hard combinatorial test instances. This was done in order to test the robustness of the simplification techniques as well as to improve the reliability of the results.

The timeout for solving each instance was set to 600 seconds. The solver was allowed to use 10% of the timeout on preprocessing and 10% of the timeout on inprocessing. These values have been determined empirically through testing. Inprocessing was run each time the ratio of the inprocessing time-limit consumed so far and the current solver runtime was less than 0.1. In addition to testing each simplification technique separately, we tested combinations of them, e.g., the combination of BCE and BVE, denoted in the following by BCE+BVE. This notion specifies also the order in which the techniques of the combination were applied. To prevent the solver from wasting time on instances where preprocessing had little effect, we used the following, empirically determined, heuristic:

1. Perform a single round of preprocessing.
2. If more than 1% of the remaining variables could be eliminated in 1% of the timeout available for solving the instance, then go to 1. and start another round. Otherwise, stop preprocessing.

Table 2 reports on the effects of the specified simplification techniques, called here *configurations*, in the application and hard combinatorial categories. Here only configurations giving the best results are presented whereas for inprocessing only lightweight techniques like UH and DI not requiring literal occurrence lists were tested. The reference configuration invokes the CDCL algorithm implemented in satUZK without doing any pre- or inprocessing at all. Note that in contrast to the original instances, it was now not possible to solve all permuted instances within the timeout. Moreover, the RSUB configuration also performs SUB in addition to RSUB, whereas UH does unhiding combined with HLE and HTE based on the information extracted during the unhiding phase. Finally, DI is applied only to the 100 most active clauses during inprocessing. Here, the variables of a clause are assigned in order of decreasing VSIDS [14] activity.

Table 2. Statistics on the application and combinatorial results. The average percentage of variables (Δvars) and clauses (Δcls) eliminated per instance by pre- and inprocessing are given in columns 4 and 5, and 7 and 8, respectively. Note that BVE was allowed to add additional but valuable binary clauses. (*: RSUB and BCE do not remove or fix variables, **: not computed for inprocessing).

		Application			Combinatorial		
Type	Configuration	solved	Δvars[%]	Δcls[%]	solved	Δvars[%]	Δcls[%]
Reference	–	184	–	–	185	–	–
Preprocessing	BVE	224	−39.59	+19.11	178	−16.67	+0.53
	RSUB	164	*	−0.18	182	*	−1.75
	BCE	192	*	−0.31	**190**	*	−0.78
	UH	200	−10.36	−2.80	187	−4.27	−2.70
	BCE+UH	203	−10.37	−3.37	189	−4.27	−3.45
	BCE+BVE	217	−40.08	+17.99	184	−18.14	−0.36
	BCE+BVE+UH	**233**	−49.63	+24.78	182	−19.12	−1.40
Inprocessing	UH	191	**	**	183	**	**
	DI	188	**	**	178	**	**
	UH+DI	**202**	**	**	183	**	**

3.2 Results for Application Instances

Figure 1 shows a cactus plot of different preprocessing configurations used for the permuted application instances. The x- and y-axis report on the instance number and the CPU time (in seconds) required to solve that instance, respectively.

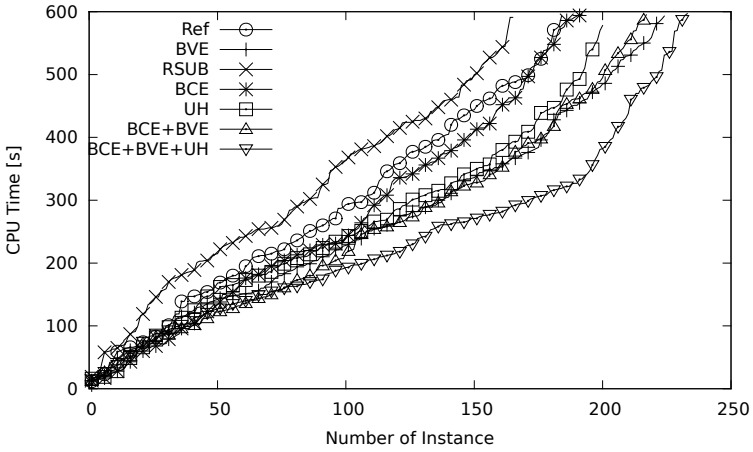

Fig. 1. Effects of preprocessing for solving application instances.

Instances are ordered by increasing runtime and counted with 600 seconds if not solved. All preprocessing configurations but RSUB perform better than the reference configuration. Configuration BVE leads to the highest performance improvement executing a single technique only. Applying BVE improved both the runtime and the number of solved instances. UH produces good results as well. The results could be further improved by combining BCE, BVE, and UH. Here, BCE alone seems to have a slightly bad effect on the solver's performance probably because BCE does not preserve UP.

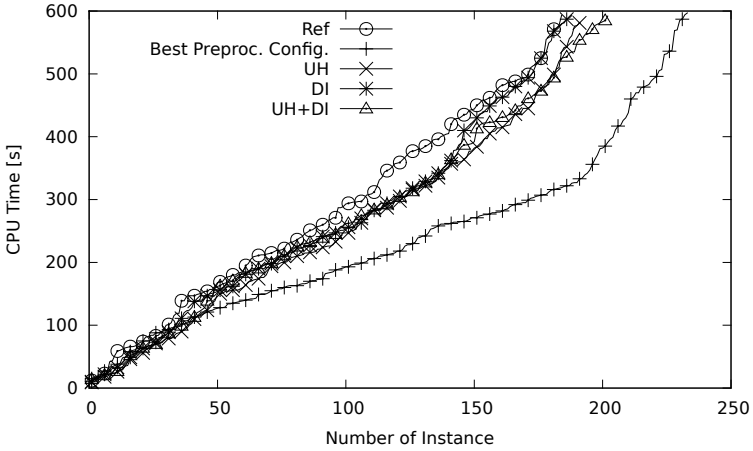

Fig. 2. Effects of inprocessing for solving application instances.

Furthermore, the bad performance of RSUB for application instances, may be explained by many cache-misses caused by subsumption when iterating through literal occurrence lists of the CDCL Solver. Interestingly, subsumption performs much better when applied to instances that have not been permuted, possibly because the variables are not randomly distributed across those instances. To assure that the results obtained did not originate from our subsumption implementation, we rerun the tests using MiniSAT 2.2 with similar results.

According to Figure 2 the performance improvements achieved by inprocessing alone for solving application instances are smaller than those obtained with preprocessing. Here UH appears to be the best configuration executing a single technique only, followed by DI solving only four more instances than the reference configuration. The best effects on SAT solving shows here UH+DI.

3.3 Results for Hard Combinatorial Instances

Figure 3 shows the effects of the different preprocessing techniques on the permuted combinatorial instances. Here only BCE and UH, both single-technique categories, outperform the reference configuration. We suspect that the reason for that is that UH improves UP by removing hidden literals. Moreover, opposite to the application category, the combination of multiple simplification algorithms does not yield better results than BCE alone. Other techniques like BVE or BCE+BVE can even prevent the solver from doing useful UP by removing clauses required for propagations. As hard combinatorial instances are small compared to application instances, the probability that important clauses are deleted is higher. Additionally, the results from Table 2 show that both BVE and UH remove fewer variables on hard combinatorial instances than they do on application instances. This, together with the inprocessing results shown in

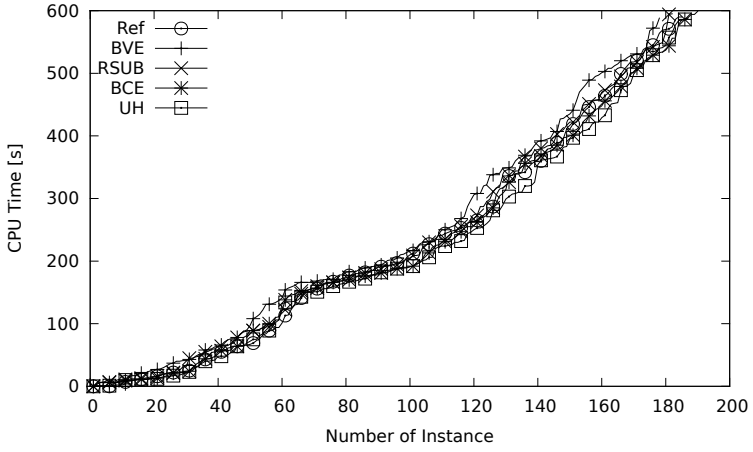

Fig. 3. Effects of preprocessing for solving hard combinatorial instances.

Figure 4, indicates that simplification techniques are not as beneficial for solving hard combinatorial instances as they are for real-world application instances.

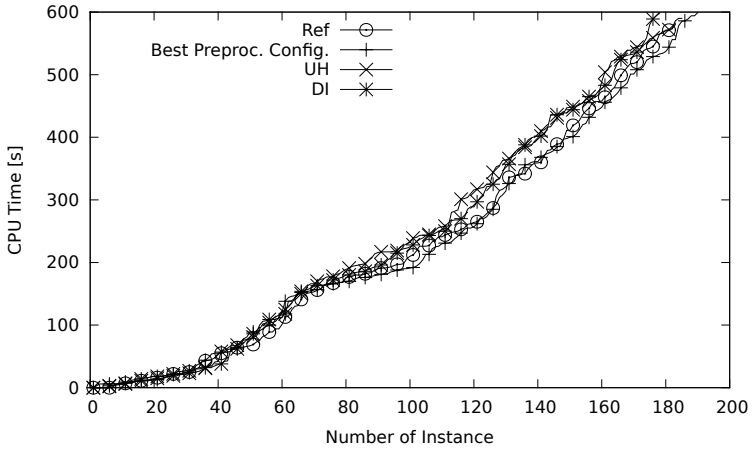

Fig. 4. Effects of inprocessing for solving hard combinatorial instances.

4 Conclusion

In this paper the effectiveness of selected simplification techniques and combinations of them for modern CDCL-based SAT solving has been examined. The techniques selected were first assessed on the basis of their theoretical characteristics important for SAT solving like preservation of UP and logical equivalence, simulatability of resolution, and implementation-related issues like practicable running times, or requirement of additional expensive data structures.

Applying preprocessing techniques to simplify CNF formulas coding real-world problems has proved to be extremely beneficial for the performance of the SAT solver. Here the combination of BCE, BVE, and UH was the most effective preprocessing configuration, followed by BVE and BCE+BVE. With proper preprocessing techniques it was possible to solve up to 27% more instances.

The inprocessing was generally not as effective as preprocessing. For solving application instances, UH+DI was the most effective configuration, improving acceptably over the reference solution. However, for the combinatorial category both UH and DI were not so successful. This indicates that developing effective inprocessing techniques is non-trivial and their success depends much on the instance type. It requires in-depth knowledge about how different techniques can be combined and integrated efficiently into the solvers' search procedure.

Finally, simplification techniques have much more effect for solving real-world SAT problems than for hard combinatorial instances, where their application could even be counterproductive.

References

1. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: SAT 2003. pp. 341–355. LNCS (2003)
2. Biere, A.: Preprocessing and inprocessing techniques in SAT. In: The 3rd Workshop on Kernelization (WorKer) (2011)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
4. Eén, N., Sörensson, N.: Minisat 2.2. `http://minisat.se/downloads/minisat-2.2.0.tar.gz`
5. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: SAT 2005. pp. 61–75. LNCS (2005)
6. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: SAT 2006. pp. 252–265. LNCS (2006)
7. van der Grinten, A., Wotzlaw, A., Speckenmeyer, E., Porschen, S.: satUZK: Solver description. In: Proc. of SAT Challenge 2012; Solver and Benchmark Descriptions. pp. 54–55. University of Helsinki (2012)
8. Han, H., Somenzi, F.: Alembic: an efficient algorithm for CNF preprocessing. In: DAC 2007. pp. 582–587 (2007)
9. Heule, M., Järvisalo, M., Biere, A.: Clause elimination procedures for CNF formulas. In: LPAR 2010. pp. 357–371 (2010)
10. Heule, M., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: SAT 2011. pp. 201–215. LNCS (2011)
11. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: TACAS 2010. pp. 129–144. LNCS (2010)
12. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: IJCAR'12. pp. 355–370. LNCS (2012)
13. Le Berre, D.: Exploiting the real power of unit propagation lookahead. Electronic Notes in Discrete Mathematics 9, 59–80 (2001)
14. Marques-Silva, J., Lynce, I., Malik, S.: CDCL Solvers, chap. 4, pp. 131–154. Vol. 185 of Biere et al. [3] (2009)
15. Piette, C., Hamadi, Y., Saïs, L.: Vivifying propositional clausal formulae. In: ECAI 2008. pp. 525–529 (2008)
16. SAT Challenge 2012. `http://baldur.iti.kit.edu/SAT-Challenge-2012/`
17. Subbarayan, S., Pradhan, D.K.: Niver: Non increasing variable elimination resolution for preprocessing sat instances. In: SAT 2004. pp. 276–291. LNCS (2004)
18. Van Gelder, A.: Toward leaner binary-clause reasoning in a satisfiability solver. Annals of Mathematics and Artificial Intelligence 43(1-4), 239–253 (2005)