On-The-Fly Clause Improvement*

Hyojung Han, Fabio Somenzi

University of Colorado at Boulder
{Hhhan,Fabio}@Colorado.EDU

Abstract. Most current propositional SAT solvers apply resolution at various stages to derive new clauses or simplify existing ones. The former happens during conflict analysis, while the latter is usually done during preprocessing. We show how subsumption of the operands by the resolvent can be inexpensively detected during resolution; we then show how this detection is used to improve three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis. The "on-the-fly" subsumption check is easily integrated in a SAT solver. In particular, it is compatible with the strong conflict analysis and the generation of unsatisfiability proofs. Experiments show the effectiveness of this technique and illustrate an interesting synergy between preprocessing and the DPLL procedure.

1 Introduction

In the last decade, advances in the satisfiability checking (SAT) of propositional formulae have been achieved through a variety of techniques to efficiently prune the search space and algorithms to improve the quality of the input formula.

Simplifying the CNF clauses speeds up Boolean Constraint Propagation (BCP) and accelerates detection of conflicts. Techniques like subsumption, variable elimination, and distillation have had significant success in practice. Preprocessing may solve some simple problems by itself, but usually does not remove all redundant clauses. That is because it has to trade off the reduction in the input formula against to the time spent.

Clause recording adds *conflict-learned clauses* or, simply, *conflict clauses* to the original SAT instance. Each conflicting assignment is analyzed to identify a subset that is sufficient to cause the current conflict. The disjunction of the literals in the subset becomes a new clause added to the original SAT instance.

Previous work has addressed the quality of conflict clauses [7, 13, 11, 6]. In particular, strong conflict analysis proposed in [6] generates a second conflict clause that is often more effective than a regular conflict clause of [13] in escaping regions of the search space where the solver would otherwise linger for long time. A common thread of most work on the subject is the search for a balance between a technique's cost and its ability of to detect implications—the *deductive power* of [4].

In this paper we propose an algorithm that detects subsumptions during resolution during both preprocessing and conflict analysis with the minimal extra effort required to compare the lengths of operands and resolvent. Our on-the-fly subsumption check can be easily applied to both strong and regular conflict analysis. We show how this

* This work was supported in part by SRC contract 2008-TJ-1859.

inexpensive check is used to improve deductive power at three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis.

Experiments show that the on-the-fly subsumption check proposed produces a great effect on run time of both the SAT solver and the preprocessing stage. Moreover, the results illustrate an interesting synergy between preprocessing techniques and on-the-fly subsumption check in the DPLL procedure.

Related to on-the-fly simplification is the problem of finding compact conflict clauses after conflict analysis. *Conflict clause minimization* [2] tests every literal in a newly-generated conflict clause. It removes any literal in the conflict clause if its negation is implied by the other literals in the clause. To do this, it also uses a resolution-based subsumption check (self subsumption) applied to the conflict clause and the antecedent clause. However, in contrast to on-the-fly simplification, this method does not simplify existing clauses. The minimization algorithm traverses only the part beyond the UIP in the implication graph, while our proposed simplification algorithm traverses between the conflicting clause and the first UIP. *Assignment shrinking* [7] is a technique to derive a new conflict clause, which tends to be more compact than the conflict clause. The algorithm of [7], after generating a conflict learned clause, backtracks to the highest level to undo all the assignments in the conflict clause. It then starts replicating the assignments to the literals involved in the previous conflict, until a new conflict occurs. This may produce a new smaller conflict clause. Since this is an expensive technique, its invocation is controlled by a criterion based on the length of the conflict clause.

An existing clause may be subsumed by a conflict clause newly found by any of the conflict analysis algorithms. Hence, one may try to simplify the newly redundant clauses. On-the-fly simplification algorithm used in [12] can detect the subsumed clause with a *one watched literal* scheme, when a new clause is generated by conflict analysis. In spite of the efficiency afforded by the one watched literal scheme, checking the subsumption relation between clauses requires significant extra work.

The rest of this paper is organized as follows. Background material is covered in Section 2. In Section 3 we describe the principles of our on-the-fly clause improvement approach during conflict analysis and present the details of the algorithm. Section 4 discusses how the subsumption check applies to the preprocessing stages, i.e., variable elimination and distillation of clauses. Section 5 reports results from the implementation of the proposed approach. We draw conclusions and outline future work in Sect. 6.

2 Preliminaries

In this paper we assume that the input to the SAT solver is a formula in Conjunctive Normal Form (CNF). A CNF formula is a set of *clauses*; each clause is a set of *literals*; each literal is either a variable or its negation. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses.

$$\{\{\neg a, c\}, \{\neg b, c\}, \{\neg a, \neg c, d\}, \{\neg b, \neg c, \neg d\}\}$$

corresponds to the following propositional formula:

$$(\neg a \vee c) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee \neg d).$$

Clause γ_1 subsumes clause γ_2 if $\gamma_1 \subseteq \gamma_2$. Given $\gamma_1 = \gamma_1' \cup \{l\}$ and $\gamma_2 = \gamma_2' \cup \{\neg l\}$, the *resolvent* of γ_1 and γ_2 is $\gamma_1' \cup \gamma_2'$ and is implied by $\{\gamma_1, \gamma_2\}$. An *assignment* for CNF formula F over the set of variables V is a mapping from V to $\{\mathsf{true}, \mathsf{false}\}$. A *partial* assignment maps a subset of V. A satisfying assignment for CNF formula F is one that causes F to evaluate to true. We represent assignments by sets of *unit* clauses, that is, clauses containing exactly one literal. For instance, the partial assignment that sets a and b to true and d to false is written $\{\{a\}, \{b\}, \{\neg d\}\}$ or, interchangeably, $a \wedge b \wedge \neg d$. A literals assigned under the partial assignment may be annotated with a *decision level*, the number of following the @ sign. For example, a assigned true at level 1 is written $a@1$. A clause γ is *asserting* under assignment A if all its literals except one (the asserted literal) are false. We say that an asserting clause is an *antecedent* of its asserted literal.

Many successful SAT solvers are based on the DPLL procedure, whose modern incarnations are described by the pseudocode of Fig. 1. The solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. In the latter case, the solver analyzes the conflict and *backtracks* accordingly. Conflict analysis [10]leads to learning a *conflict clause*, that is, a clause computed from repeated applications of resolution to the conflicting clause and the *antecedent clause* of the literal in the conflicting clause that was most recently implied.

```
1  GRASP_DPLL() {
2    while (CHOOSENEXTASSIGNMENT() == FOUND)
3      while (DEDUCE() == CONFLICT) {
4        blevel = ANALYZECONFLICT();
5        if (blevel < 0) return UNSATISFIABLE;
6        else BACKTRACK(blevel);
7      }
8    return SATISFIABLE;
9  }
```

Fig. 1. GRASP_DPLL algorithm.

The GRASP_DPLL procedure is often applied after a preprocessing phase, which attempts to remove redundant clauses and literals to speed up SAT solvers. SatELite [1, 9] simplifies a CNF formula by iteratively checking the subsumption relation between clauses and eliminates variables by resolution. For instance, $a \vee b$ and $a \vee \neg b \vee c$ give a resolvent $a \vee c$ that subsumes the latter; this is called *self subsumption*. Resolution can also be applied to eliminate variables from the formula. If, for example, $a \vee b$, $\neg b \vee c$ and $\neg b \vee d$ are the only clauses containing b, then they can be replaced by $a \vee c$ and $a \vee d$ while guaranteeing *equisatisfiability*. Since the elimination of variables may increase the number of clauses, it is used in a limited way in preprocessing. In [4], the CNF formula is *distilled* to increase its *deductive power*. The transformation is done by asserting the negation of each in a clause until either a conflict is found, or one of the

literals of the clause is implied true. In both cases, the distillation procedure analyzes the implication graph to generate the improved clause.

Example 1. Consider the following formula:

$$(a \vee b \vee \neg c) \wedge (a \vee c \vee d) \wedge (b \vee c \vee e) \wedge (\neg d \vee f) \wedge (\neg e \vee g) \wedge (\neg f \vee \neg g \vee h) \wedge (\neg f \vee \neg g \vee \neg h)$$

Suppose that the decisions $\{\neg a@1, \neg b@2\}$ are made by the SAT solver and that the implications of those decisions are computed. Figure 2 shows the implication graph that represents the implications derived up to the current decision level. Directed edges in the graph are labeled with clause numbers. The implications result in a conflict on variable h, that is, two opposite assignment to h at the same time. Conflict analysis is therefore invoked. The implication graph in Fig. 2 also shows each resolvent γ_i that the conflict analysis generates while traversing backward the implication graph from the conflicting clause c_7. □

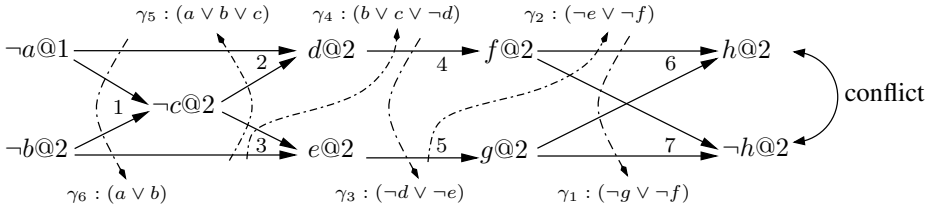

Fig. 2. Implication graph for the first conflict of Example 1.

Most conflict analysis algorithms terminate as soon as they find a clause containing a *Unique Implication Point* (UIP), that is, a single assignment made at the current level. For this case, since γ_6 contains the first UIP, that is literal b, it is chosen as conflict clause. However, the UIP is in this case the decision variable. When the first UIP is far from the conflict in the implication graph, the conflict clause may not be effective in preventing the SAT solver from repeating the same mistake. *Strong conflict analysis* [6] can be a remedy in such cases: It examines intermediate resolvents as UIP-based conflict analysis does. Contrary to UIP-based analysis, however, it generates an additional conflict clause that contains more than one literal assigned at the current decision level. This additional conflict clause must be one of the intermediate resolvents derived between the conflict and the first UIP. Usually, the closer to the conflict, the fewer literals the resolvent contains. Therefore, the additional conflict clause tends to be shorter than the conflict clause with 1-UIP.

3 On-the-fly Simplification Based on Resolution

Detecting whether the resolvent of two clauses subsumes either operand is easy and inexpensive. Therefore, checking *on-the-fly* for subsumption can be added with almost

no penalty to those operations of SAT solvers that are based on resolution. In this section we review the basic idea and detail its application to conflict analysis. Later, we discuss on-the-fly subsumption in preprocessing.

An efficient on-the-fly check for subsumption during resolution is based on the following elementary fact.

Lemma 1. *Let* $c_1 = c_1' \cup \{l\}$ *and* $c_2 = c_2' \cup \{\neg l\}$ *be two clauses. Their* resolvent $c_1' \cup c_2'$ *subsumes* c_1 *(*c_2*) iff* $|c_1' \cup c_2'| = |c_1| - 1$ *(*$|c_1' \cup c_2'| = |c_2| - 1$*).*

Thanks to Lemma 1, it is possible to detect existing clauses that are subsumed by resolvents and replace them with the resolvents themselves. Doing so during conflict analysis is easy because the eliminated literal is the one asserted by the clause itself. If that literal is kept in first position in the clause [3], it is easily accessed. In variable elimination, as we shall see, the literal to be removed corresponds to the variable that is being eliminated. Therefore, it is enough to save its position while scanning a clause. In summary, the overhead of on-the-fly subsumption check is negligible. The advantages, on the other hand, may be significant as illustrated by the following example.

Example 2. Consider the following set of clauses:

$$F = (a \vee b \vee \neg c) \wedge (a \vee b \vee \neg d) \wedge (c \vee d \vee \neg e) \wedge (c \vee e \vee f) \wedge (d \vee e \vee \neg f) \wedge (\neg b \vee \neg d \vee e) \wedge (\neg d \vee \neg e)$$

Suppose that the first decision is to set a to false, and the second decision is to set b to false. From these decisions literals $\neg c$, $\neg d$, and $\neg e$ are deduced at level 2. This partial assignment, in turn, yields f and $\neg f$ through the fourth and the fifth clause. Analysis of this conflict produces the implication graph shown in Fig. 3.

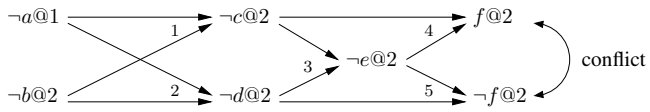

Fig. 3. Implication graph of Example 2.

Suppose that the fifth clause is the conflicting clause. Conflict analysis goes back through the implication graph building the resolution tree shown in Fig. 4. The tree is comprised of leaves, c_i, which correspond to the clauses appearing in the implication graph; intermediate nodes, γ_i, which are the resolvents, and a root node which is both a resolvent and the conflict-learned clause. The resolution tree shows that γ_2 subsumes c_3, and that γ_4 subsumes c_1. The subsumed clauses can be strengthened by eliminating the pivot variable on which they were resolved. In addition to the simplifications, γ_4 containing the first UIP subsumes c_1; it is not required to add it to the clause data base. Rather, c_1 is strengthened. The simplified CNF is therefore

$$F' = (a \vee b) \wedge (a \vee b \vee \neg d) \wedge (c \vee d) \wedge (c \vee e \vee f) \wedge (d \vee e \vee \neg f) \wedge (\neg b \vee \neg d \vee e) \wedge (\neg d \vee \neg e)$$

Then, the solver backtracks to level 1, which is the highest decision level in c_1. After backtracking, $b@1$ is asserted by c_1.

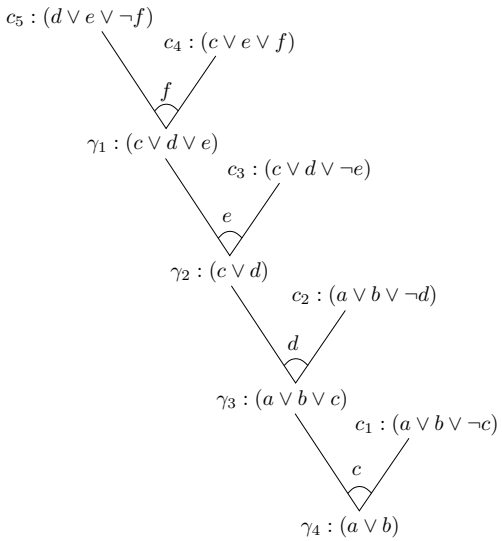

Fig. 4. Resolution tree of conflict analysis for Fig. 2

This example shows how the CNF database can be simplified by checking subsumption on-the-fly. First of all, a clause can be shortened when it is resolved during conflict analysis and it is subsumed by the resolvent. The resolvent may contain a UIP. Then, the clause that is strengthened can serve as conflict-learned clause. In this case, the SAT solver has the same deductive power, even without adding conflict clause. □

In our solver, conflict analysis based on 1-UIP may be followed by strong conflict analysis. We now consider the on-the-fly subsumption check with respect to strong conflict analysis.

Lemma 2. *Let C be a clause simplified with the resolvent produced in conflict analysis. Then C is conflicting at the current level.*

Proof. Every resolvent produced in conflict analysis is conflicting at the current decision level. Therefore, clause C, which is one such resolvent, is also conflicting. □

Lemma 3. *Let C be the clause most recently simplified by on-the-fly subsumption during conflict analysis. The subgraph of the implication graph between this clause and the 1-UIP is either a single vertex or a valid implication graph (hence, suitable for strong conflict analysis).*

Proof. The requirement for a valid implication subgraph is that the source vertex be a clause with at least two literals assigned at the current level. By Lemma 2, C is conflicting at the current decision level. If C contains the 1-UIP, the subgraph consists of a single vertex and strong conflict analysis is not invoked. Otherwise, since the residual clauses beyond C on the graph were not touched, they are also valid to be examined by strong conflict analysis. □

Lemmas 2 and 3 allow us to conclude that on-the-fly subsumption check is compatible with strong conflict analysis. As an alternative, one could postpone the strengthening of the clauses until after strong conflict analysis. Our experiments, however, indicate that it would not be as efficient.

Returning to Example 1, in Fig. 2, γ_1, γ_2, and γ_3 all have a chance to be additional conflict clauses, since all of them have only two literals, and both literals are assigned at the current level. The strong conflict analysis, however, dismisses γ_1 because it is too close to the conflicting clause. Therefore, it misses the chance to strengthen c_6 and drop c_7. In contrast, on-the-fly subsumption is not constrained to use only one clause for simplification.

Figure 5 shows the pseudocode of the algorithm that detects and simplifies the subsumed clauses during conflict analysis. The algorithm AnalyzeConflictWithDistill() keeps checking the subsumption condition whenever a new resolvent is produced as long as FoundUIP() is false (line 4). By Lemma 1, if the operand exists in the clause database, that is, the old resolvent with in_CNF_resolvent = TRUE (line 9) or the antecedent (line 13 and 17), and the new resolvent contains fewer literals than one of its operands (line 10, 13, and 17), the operand is strengthened for the pivot variable by StrengthenClause() (line 11 and 18). When both operands are subsumed, only one of them is selected to survive, and the other is deleted (line 14). If a clause is replaced with the resolvent, the flag in_CNF_resolvent is set to TRUE (line 12, and 19). Otherwise,in_CNF_resolvent is reset to FALSE (line 22), since the new resolvent does not exist in the clause database yet. At the end of the resolution step (line 25), if the final resolvent containing the UIP is identified as an existing clause, that is in_CNF_resolvent is false, the conflict analysis algorithm refrains from adding a new conflict clause into the clause database (line 26). Whether a new conflict clause is added or not, the DPLL procedure backtracks up to the level returned by the conflict analysis (line 28), and asserts the clause finally learned from the latest conflict.

The pseudocode of Fig. 5 omits some details for the sake of clarity. In the actual implementation, the implication graph is shrunk with a new conflicting clause by replacing the current conflicting clause with a newly strengthened clause, which must be a new resolvent. The modified graph then is available for strong conflict analysis.

4 Application to Preprocessors

Resolution is the main operation in preprocessing. In this section, we review its application to the preprocessors for variable elimination and clause distillation.

To select variables to be eliminated, all the variables are sorted by a metric such that $\delta = (|\text{clauses}_v| * |\text{clauses}_{\neg v}|) - (|\text{clauses}_v| + |\text{clauses}_{\neg v}|)$, where clauses_v stands for

```
1  AnalyzeConflictWithDistill(F, conflicting) {
2    resolvent = conflicting;
3    in_CNF_resolvent = TRUE;
4    while (!FoundUIP(resolvent)) {
5      lit = GetLatestAssignedLiteral(resolvent);
6      ante = GetAntecedentClause(lit);
7      var = Variable(lit);
8      resolvent' = Resolve(resolvent, ante, var);
9      if (in_CNF_resolvent) {
10       if (Size(resolvent') < Size(resolvent)) {
11         StrengthenClause(resolvent, var);
12         in_CNF_resolvent = TRUE;
13         if (Size(resolvent') < Size(ante))
14           DeleteClause(ante);
15       }
16     }
17     else if (Size(resolvent') < Size(ante)) {
18       StrengthenClause(ante, var);
19       in_CNF_resolvent = TRUE;
20     }
21     else
22       in_CNF_resolvent = FALSE;
23     resolvent = resolvent' ;
24   }
25   if (!in_CNF_resolvent)
26     AddConflictClause(resolvent);
27   blevel = ComputeHighestLevelInConflictClause(resolvent);
28   return (blevel);
29 }
```

Fig. 5. Algorithm for conflict analysis with subsumption check

an occurrence list of variable v, and $|\text{clauses}|$ represents the length of the list. δ stresses the fact that the less symmetric occurrence lists are, the earlier the variable should be selected. The length of a resolvent should also be taken into account, because clauses may also be lengthened through resolution. This can be harmful to the SAT solver. Hence, we use an additional criterion, the number of literals of the resolvents, to choose variables to be eliminated.

To eliminate a variable, resolutions are applied to all the pairs of clauses in the occurrence lists of the two literals of the variable. In our variable elimination, all the literals of each clause are sorted by variable index. Taking the union of two sorted clauses can be done in linear time by a variation of the *merge-sort* algorithm. This linear operation guarantees that all the literals are still sorted after merging. With minor modification in the algorithm, the linear operation can be also used to check subsumption relation between two clauses.

A variable is eliminated only when the produced resolvents are fewer than the occurrence clauses of the variable. At each resolution operation, we can check if one of

the operands is subsumed by the resolvent, like the on-the-fly subsumption check in conflict analysis of Sect. 3. A clause can be simplified by the on-the-fly subsumption, regardless of whether the variable is eliminated. The clause simplified by the on-the-fly subsumption is removed out of the occurrence list. In such a case, the current elimination check may benefit from the shortened occurrence list. Every simplified clause is checked for subsumption to other clauses after the variable elimination check.

During distillation of clauses, conflict analysis takes the majority of the time. Conflict analysis in clause distillation also performs resolutions steps as conflict analysis in DPLL. Therefore we can increase efficiency in conflict analysis by using on-the-fly simplification. In the original algorithm of distillation, the clauses of the SAT instance are distilled only if such conflict clauses are detected. More chances for simplifications, however, can be produced if we apply on-the-fly simplification. In addition to the on-the-fly subsumption check, the distillation procedure is combined with "regular subsumption check" used in variable elimination procedure. The clause that is being distilled may participate in conflict analysis. If a clause that is an antecedent in conflict analysis contains all the decision variables, then it is the clause being distilled. We can identify such clauses while scanning the clause for resolution. If such a clause is found, it can be replaced by the conflict clause. Otherwise, the regular subsumption check can simplify the clause after distillation procedure.

5 Experimental Results

We have implemented the three applications of on-the-fly subsumption checking proposed in this paper on top of the CirCUs SAT solver [5]. The implemented approaches are the enhanced variable elimination, the improved Alembic, and on-the-fly simplification in conflict analysis.

The benchmark suite is composed of all the instances (no duplication) from the industrial category of the SAT Races of 2006 and 2008, and the SAT competition of 2007. We conducted the experiments on a 2.4 GHz Intel Core2 Duo processor with 4GB of memory. We used 3600 seconds as timeout, and 2GB as memory bound. We tested MiniSat 2.0 along with CirCUs 2.0 to provide a reference point.

Figure 6(a) shows the CPU time taken by MiniSat and CirCUs with and without their respective preprocessors. In the plot, to distinguish from SatELite, our variable elimination algorithm only is named EV, and EV with clause distillation of Alembic is named EVAL, and OTS stands for on-the-fly simplification. The data points on the plot show how many instances are solved within the given time bound.

Figure 6(b) details the effect of adding on-the-fly subsumption check to each of the three steps discussed in this paper. It can be seen that while the overall benefit is clear, there is no advantage to the application of on-the-fly subsumption to conflict analysis alone. We investigate this apparent anomaly with the help of Fig. 7, which compares the number of on-the-fly subsumptions per conflict(Fig. 7(a)) and the average resolution depth in conflict analysis(Fig. 7(b)) with and without variable preprocessing. The scatterplot confirms that there is a strong synergy between preprocessing and on-the-fly subsumption. Further analysis shows that variable elimination is the main responsible for the marked increase of subsumptions per conflict and for the shortening of reso-

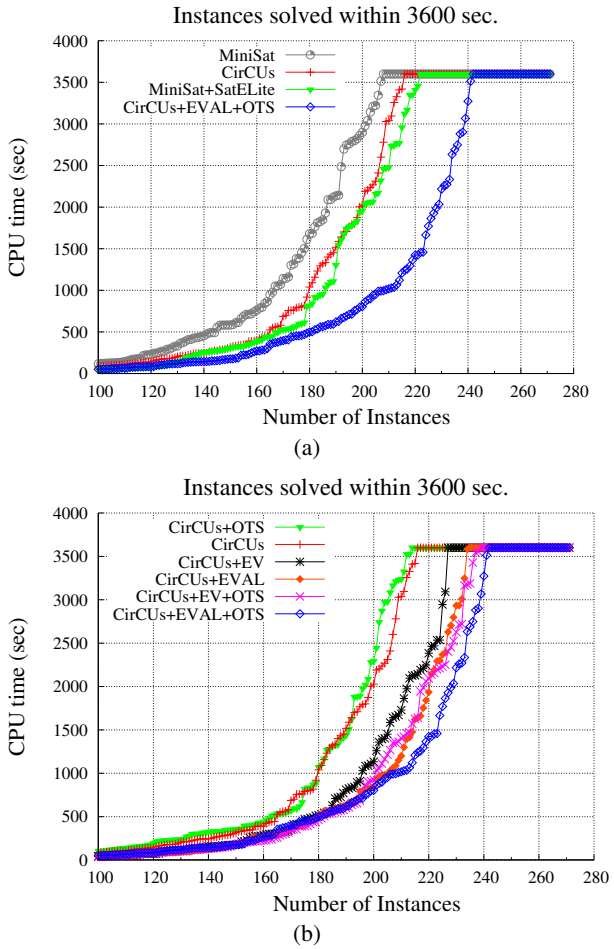

Fig. 6. Number of instances solved by various SAT solvers versus CPU time. (a) comparison of the proposed algorithm to modern SAT solvers; (b) individual contributions of simplification methods to CirCUs

lution steps computed in conflict analysis. The following example sheds light on this phenomenon.

Example 3. Consider the following clauses:

$$(\neg a \vee \neg p)_1 \wedge (b \vee \neg p)_2 \wedge (a \vee \neg b \vee p)_3 \wedge (a \vee \neg q)_4 \wedge (\neg b \vee \neg q)_5 \wedge (\neg a \vee b \vee q)_6$$
$$\wedge(\neg p \vee r)_7 \wedge (\neg q \vee r)_8 \wedge (p \vee q \vee \neg r)_9 \wedge (a \vee \neg s)_{10} \wedge (b \vee \neg s)_{11} \wedge (\neg a \vee \neg b \vee s)_{12}$$
$$\wedge(\neg a \vee \neg t)_{13} \wedge (\neg b \vee \neg t)_{14} \wedge (a \vee b \vee t)_{15} \wedge (\neg s \vee \neg u)_{16} \wedge (\neg t \vee \neg u)_{17}$$
$$\wedge(s \vee t \vee u)_{18} \wedge (r \vee u)_{19} \wedge (\neg r \vee \neg u)_{20}.$$

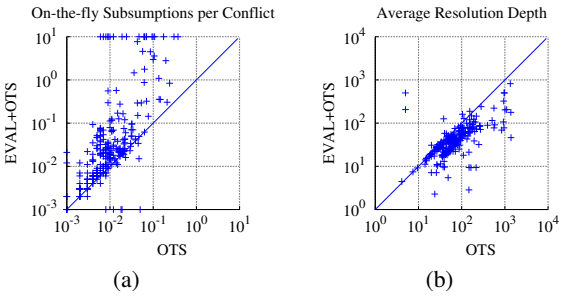

Fig. 7. (a) On-the-fly subsumptions per conflict; (b) Average depth of resolution

Suppose that the SAT solver makes decisions $\neg a@1$ and $\neg b@2$. This leads to a conflict on c_{19}, with the implication graph shown in Fig. 8. There are no instances of on-the-fly subsumption during conflict analysis, even though the learned clause, $\gamma_5 = a \vee b$, subsumes c_{15}: γ_5 directly subsumes other resolvents rather than c_{15}.

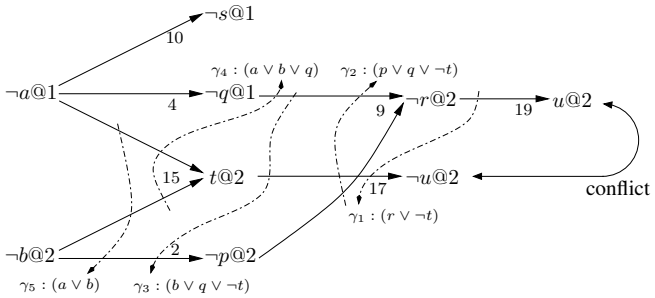

Fig. 8. Implication graph for the original clauses of Example 3

If we eliminate p, q, s, and t, we get the following clauses:

$$(a \vee \neg b \vee r)_1 \wedge (a \vee b \vee \neg r)_2 \wedge (\neg a \vee \neg b \vee \neg r)_3 \wedge (\neg a \vee b \vee r)_4 \wedge (a \vee \neg b \vee u)_5$$
$$\wedge (a \vee b \vee \neg u)_6 \wedge (\neg a \vee \neg b \vee \neg u)_7 \wedge (\neg a \vee b \vee u)_8 \wedge (r \vee u)_{14} \wedge (\neg r \vee \neg u)_{15} \ .$$

Figure 9 shows that the conflict-learned clause subsume c_2. (It also subsumes c_6, but this is not detected by the algorithm.) This time there are fewer resolution steps, and this "abridgment" of the process allows the subsumed clause to enter the analysis right before the subsuming resolvent is computed instead of several steps before. This mechanism seems to account for several cases in which variable elimination, and preprocessing in general, increase the frequency of on-the-fly subsumptions. □

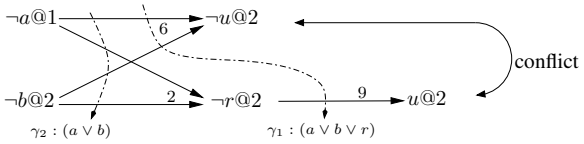

Fig. 9. Implication graph for the clauses of Example 3 after elimination

For the purpose of calibration, we also compared CirCUs+EVAL+OTS to Rsat 3.01 [8]. In the comparison, CirCUs+EVAL+OTS solved 240 instances, and Rsat 3.01 solved 256 instances within one hour.

Finally, we report statistics on the performance of the preprocessors. Figure 10(a) compares the speed of EVAL to SatELite. In this case, SatELite is run on all CNF formulae, while, in Fig. 6(a), SatELite can be disabled depending on the size of CNF formulae. This results in a few timeouts, but otherwise the two preprocessors are quite comparable. It should be noted that the current implementation of Alembic is much faster than that described in [4]. Figure 10(b), in particular, shows that on-the-fly subsumption significantly contributes to the improved preprocessor speed.

It is also interesting to compare the reductions achieved by the different preprocessors. In Fig. 11, we see that CirCUs's variable elimination is less aggressive than SatELite: it eliminates fewer clauses, but almost never increases the number of literals. Adding Alembic yields the least number of clauses without compromising the good performance in terms of literals.

6 Conclusions

We have presented a simple, efficient technique to detect subsumption of an operand by the resolvent of two clauses. This technique can be applied with minimal overhead to both preprocessing of the CNF formula and to conflict analysis. The effect is to simplify clauses in such a way that implications are obtained earlier and conflicts are sometimes avoided. Another beneficial effect is the reduction of the number of added conflict-learned clauses without detriment for the deductive power. Our experiments show that the new technique delivers a significant improvement in performance. One final advantage is its compatibility with advanced conflict analysis techniques and with the generation of unsatisfiability proofs.

References

[1] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[2] N. Eén, A. Mischchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In *Theory and Applications of Satisfiability Testing: SAT 2007*, pages 272–286, Lisbon, Portugal, May 2007. Springer. LNCS 4501.

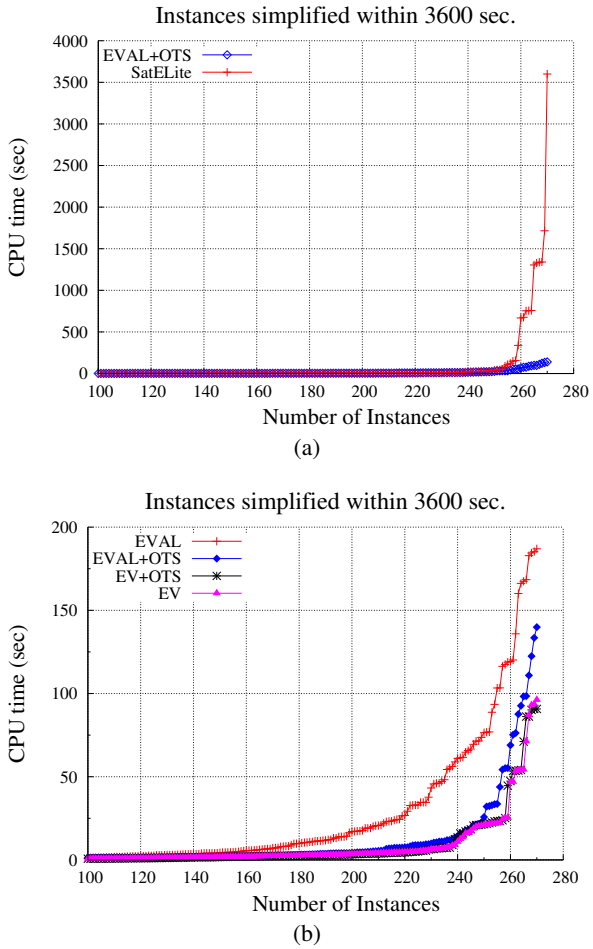

Fig. 10. Number of instances simplified by various preprocessors versus CPU time. (a) comparison of the proposed preprocessor to SatELite in MiniSat; (b) detailed comparison of the applied preprocessing techniques

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 502–518, S. Margherita Ligure, Italy, May 2003. Springer-Verlag. LNCS 2919.

[4] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the Design Automation Conference*, pages 582–587, San Diego, CA, June 2007.

[5] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 519–522. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[6] H. Jin and F. Somenzi. Strong conflict analysis for propositional satisfiability. In *Design, Automation and Test in Europe (DATE'06)*, pages 818–823, Munich, Germany, Mar. 2006.

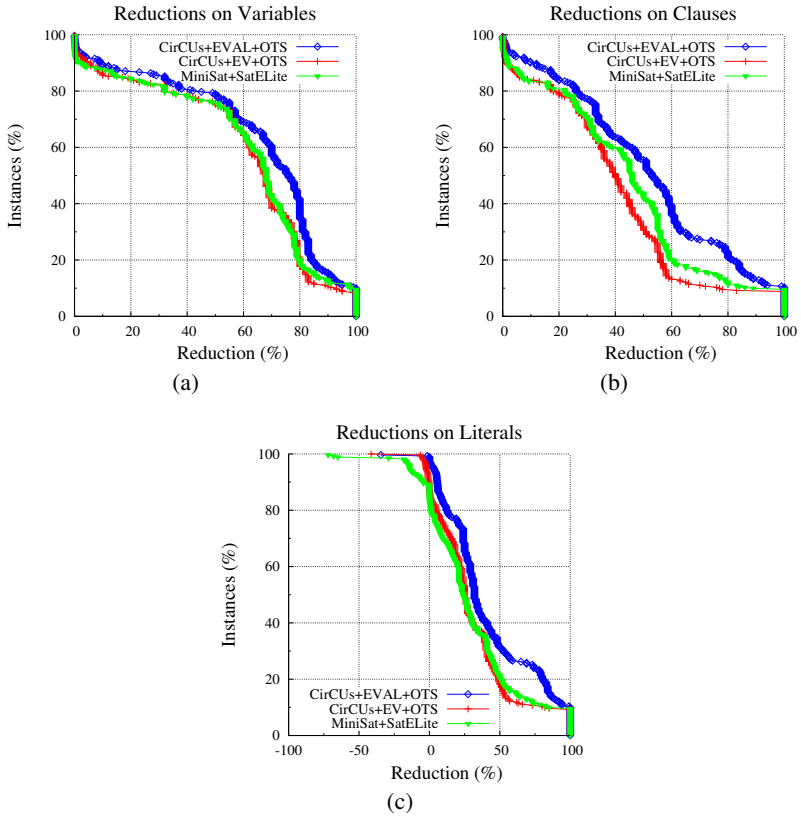

Fig. 11. Ratio of simplification made by various preprocessors on variables, clauses, and literals

[7] A. Nadel. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem, 2002.
[8] URL: http://reasoning.cs.ucla.edu/rsat/.
[9] URL: http://www.cs.chalmers.se/Cs/Research/ FormalMethods/MiniSat/MiniSat.html.
[10] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.
[11] N. Sörensson and N. Eén. MiniSat v1.13 – a SAT solver with conflict-clause minimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.
[12] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.
[13] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.