# Increasing The Effectiveness of Deduction in Propositional SAT Solvers

by

**Hyojung Han**

B.S., Kangnung National University, Korea, 1999

M.S., University of Colorado at Boulder, 2010

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computerm, and Energy Engineering

2010

This thesis entitled:
Increasing The Effectiveness of Deduction in Propositional SAT Solvers
written by Hyojung Han
has been approved for the Department of Electrical, Computerm, and Energy Engineering

_____

Fabio Somenzi

_____

Aaron Bradley

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Han, Hyojung (Ph.D., Electrical and Computer Engineering)

Increasing The Effectiveness of Deduction in Propositional SAT Solvers

Thesis directed by Professor Fabio Somenzi

The satisfiability (SAT) of a propositional formula is the decision problem to determine whether there is a satisfying assignment that can make the formula `true` or not. In the past few years, many successful SAT solvers based on the David-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62, MS99, MMZ+01, ES03] for formulae in conjunctive normal form (CNF) have been developed. Since the deduction procedure of DPLL is sound but not complete, its effects depend on which formula is selected to represent the input function. CNF transformations are among the most effective techniques to improve quality of the input formula by either simplifying clauses [ES03, EB05, SE05, ZKKSV06, HS07, HS09] or learning new ones [MS99]. Specifically, effective CNF transformations can help SAT solvers to be sped up by allowing them to do more deductions and less enumerations.

In my dissertation, I characterize existing transformations in terms of their impact on the **deductive power** of the formula and their effects on the **proof conciseness**, that is, the sizes of the implication graphs. I also present two new techniques that try to increase deductive power. The first is a check performed during the computation of resolvents. The second is a new preprocessing algorithm based on **distillation** that combines simplification and increase of deductive power. Most current 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. I show how subsumption of the operands by the resolvent can be inexpensively detected during resolution; I then discuss 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 strong conflict analysis and the generation of unsatisfiability proofs. Experiments show the effectiveness of the new techniques.

SAT solvers also benefit from clauses learned by the DPLL procedure, even though they are by definition redundant. In addition to those derived from conflicts, the clauses learned by **dominator analysis**

during the deduction procedure tend to produce smaller implication graphs and sometimes increase the deductive power of the input CNF formula. I extend dominator analysis with an efficient self-subsumption check. I also show how the information collected by dominator analysis can be used to detect redundancies in the satisfied clauses and, more importantly, how it can be used to produce supplemental conflict clauses. I characterize these transformations in terms of deductive power and proof conciseness. Experiments show that the main advantage of dominator analysis and its extensions lies in improving proof conciseness.

This thesis is dedicated to my grandmother ”OkJib Choi”. Thank you for your strength, your kindness and your love.

## Acknowledgements

First and foremost I would like to thank my adviser Professor Fabio Somenzi for his guidance and support throughout my graduate studies. He has been my inspiration as I hurdle all the obstacles in the completion this research work. I also thank him for giving me the great opportunity to enjoy hiking in the Rocky Mountains. Especially, I will never forget the moment when we were mountaineering the Mountain Evans at 2005. For this dissertation, I would like to make a special reference to Professor Armin Biere, Professor Aaron Bradley, Professor Michael Lightner, and Professor Sriram Sankaranarayanan for kindly serving on my thesis committee.

The members of the VLSI CAD group have contributed immensely to my personal and professional time at University of Colorado. I especially want to acknowledge honorary group member Hoonsang Jin who is now working for Cadence Design Systems. We have worked together on our Satisfiability Solver, CirCUs. I appreciate his enthusiasm and intensity for our work. Besides Hoonsang, I am grateful for time spent with Hyondeuk Kim and Saqib Sohail, who shared with me not only knowledge in our study but also such a great experience in Boulder, Colorado.

My internships at Intel allow me to have a valuable experience of solving real-world problems from industry. I would like to thank Dr. Murali Talupur and Dr. James Grundy for mentoring and supervising me during my internship in Intel Strategic CAD Laboratories.

Lastly, I would like to thank my family for all their love and encouragement. For my parents who supported me in all my pursuits. And most of all for my loving, supportive and encouraging husband Hyoungsuk and my lovely daughter Binna, I heartily appreciate for their faithful support and patient during this Ph.D.

## Contents

**Chapter**


**1** Introduction 1

1.1 Background . . . . . 1

1.2 Thesis Contributions . . . . . 3

1.3 Thesis Organization . . . . . 5

1.4 Related Work . . . . . 6

**2** Propositional Satisfiability Solvers 8

2.1 Propositional Satisfiability Problems . . . . . 9

2.2 Representations . . . . . 9

2.3 CNF Formulae . . . . . 11

2.4 CNF SAT Solvers . . . . . 12

2.5 The DPLL Procedure . . . . . 13

2.6 Conflict Analysis . . . . . 15

2.7 Proof of Unsatisfiability . . . . . 19

2.8 Preprocessing . . . . . 20

**3** Increasing the Efficiency of the Deduction Procedure 21

3.1 Deductive Power of a CNF Formula . . . . . 23

3.2 On-The-Fly Self-Subsumption . . . . . 29

3.3 Clause Distillation . . . . . 37

3.4 Variable Elimination . . . . . 44
3.5 Experimental Results . . . . . 47
3.6 Summary and Discussion . . . . . 59

**4** Clause Simplification through Dominator Analysis 61
4.1 Dominators . . . . . 62
4.2 Simplifying Clauses During Deduction . . . . . 67
4.3 Dominator Clauses and Redundancy . . . . . 71
4.4 Garbage Collection . . . . . 72
4.5 Dominator-Based Conflict Clauses . . . . . 73
4.6 Experimental Results . . . . . 76
4.7 Summary and Discussion . . . . . 84

**5** Conclusions 86
5.1 Thesis Conclusions . . . . . 86
5.2 Future Work . . . . . 87

**Bibliography** 88

**Appendix**

**A** Tables for Comparison 94

# Tables

**Table**

A.1 Comparison of CirCUs with and without the proposed techniques (1) . . . . . 95
A.2 Comparison of CirCUs with and without the proposed techniques (2) . . . . . 96
A.3 Comparison of CirCUs with and without the proposed techniques (3) . . . . . 97
A.4 Comparison of CirCUs with and without the proposed techniques (4) . . . . . 98
A.5 Comparison of CirCUs with and without the proposed techniques (5) . . . . . 99
A.6 Comparison of CirCUs with and without the proposed techniques (6) . . . . . 100
A.7 Comparison of CirCUs with and without the proposed techniques (7) . . . . . 101
A.8 Comparison of CirCUs with and without the proposed techniques (8) . . . . . 102
A.9 Comparison of CirCUs with and without the proposed techniques (9) . . . . . 103
A.10 Comparison of CirCUs with and without the proposed techniques (10) . . . . . 104
A.11 Comparison of CirCUs with and without the proposed techniques (11) . . . . . 105
A.12 Comparison of CirCUs with and without the proposed techniques (12) . . . . . 106
A.13 Comparison of CirCUs with and without the proposed techniques (13) . . . . . 107
A.14 Comparison of CirCUs with and without the proposed techniques (14) . . . . . 108
A.15 Comparison of CirCUs with and without the proposed techniques (15) . . . . . 109

# Figures

**Figure**


2.1 GRASP_DPLL algorithm. . . . . 14

2.2 Implication hypergraph for Example 2.5. . . . . 15

2.3 Implication graph for the first conflict of Example 2.6. . . . . 16

2.4 Resolution graph of conflict analysis for Example 2.6. . . . . 17

2.5 Conflict clause computed on an implication graph. . . . . 18

3.1 Implication graph of Example 3.2. . . . . 24

3.2 Implication graph of Example 3.7. . . . . 27

3.3 Implication graph of Example 3.10. . . . . 28

3.4 Implication graph for the first conflict of Example 3.11. . . . . 29

3.5 Implication graph without additional conflict clause. . . . . 30

3.6 Implication graph with additional conflict clause. . . . . 30

3.7 Implication graph of Example 3.13. . . . . 32

3.8 Resolution tree of conflict analysis for Fig. 3.7 . . . . . 32

3.9 Implication graph for the second conflict of Example 3.13. . . . . 33

3.10 Resolution tree of conflict analysis for Fig. 3.9. . . . . 33

3.11 Implication graph shrunk from Fig. 3.7 with a new conflict node. . . . . 34

3.12 Algorithm for conflict analysis with on-the-fly simplification. . . . . 36

3.13 The implication graph of Example 3.17. . . . . 37

3.14 Implication graph of Example 3.18. . . . . 39

3.15 Resolution graph of conflict analysis for Example 3.18. . . . . 39

3.16 Algorithm for clause distillation. . . . . 43

3.17 A trie of the CNF clauses of $F$. . . . . 45

3.18 The implication graph generated under $A_1 = \{\{\neg b\}\}$. . . . . 45

3.19 The implication graph generated under $A_2 = \{\{b\}, \{\neg c\}\}$. . . . . 45

3.20 The resolution graph of the conflict analysis on Fig. 3.20. . . . . 45

3.21 The implication graph under $A_1 = \{\{\neg c\}\}$. . . . . 46

3.22 The resolution graph of conflict analysis on Fig. 3.21. . . . . 46

3.23 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 . . . . . 49

3.24 Comparison of the performance improvements between (a)SAT and (b)UNSAT instances of Fig. 3.23. . . . . 50

3.25 Effect of CirCUs with and without EVAL+OCI on (a) CPU time: GEOMETRIC MEAN = $0.56$, $p$-value = $2.2 \cdot 10^{-16}$; (b) number of decisions: GEOMETRIC MEAN = $0.51$, $p$-value = $2.2 \cdot 10^{-16}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = $0.57$, $p$-value = $2.2 \cdot 10^{-16}$; (d) number of literals per conflict clause: GEOMETRIC MEAN = $0.82$, $p$-value = $8.25 \cdot 10^{-8}$. . . . . 52

3.26 Effect of CirCUs with and without EV+OCI on (a) CPU time: GEOMETRIC MEAN = $0.63$, $p$-value = $5.7 \cdot 10^{-16}$; (b) number of decisions: GEOMETRIC MEAN = $0.59$, $p$-value = $2.2 \cdot 10^{-16}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = $0.68$, $p$-value = $2.29 \cdot 10^{-15}$; (d) number of literals per conflict clause: GEOMETRIC MEAN = $0.87$, $p$-value = $7.8 \cdot 10^{-5}$. . . . . 53

3.27 Effect of CirCUs with and without AL+OCI on (a) CPU time: GEOMETRIC MEAN = $0.86$, $p$-value = $0.003$; (b) number of decisions: GEOMETRIC MEAN = $0.77$, $p$-value = $1.71 \cdot 10^{-6}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = $1.01$, $p$-value = $0.76$; (d) number of literals per conflict clause: GEOMETRIC MEAN = $0.92$, $p$-value = $0.03$. . . . . 54

3.28 The number of resolution steps per conflict. . . . . 55

3.29 The effect on memory consumption. . . . . 55

3.30 Number of OCI applications per resolution step with and without preprocessing: (a) both elimination and distillation: GEOMETRIC MEAN = $1.9$, $p$-value = $2.96 \cdot 10^{-9}$; (b) only elimination: GEOMETRIC MEAN = $1.69$, $p$-value = $6.65 \cdot 10^{-12}$; (c) only distillation: GEOMETRIC MEAN = $0.92$, $p$-value = $0.17$. . . . . 56

3.31 Implication graph of Example 3.21 without EV. . . . . 57

3.32 Implication graph of Example 3.21 with EV. . . . . 58

3.33 Number of instances simplified by various preprocessors versus CPU time. SatELite times out on one instance after 3600 s. . . . . 58

3.34 Ratio of simplification made by various preprocessors on (a) variables, (b) clauses, and (c) literals. . . . . 60

4.1 Implication graph of Example 4.2 . . . . . 63

4.2 Implication graph of Example 4.3 . . . . . 64

4.3 Dominator analysis in PrecoSAT . . . . . 66

4.4 Implication graph of Example 4.4 . . . . . 66

4.5 Implication graph of Example 4.6 . . . . . 68

4.6 Implication graph of Example 4.7 . . . . . 69

4.7 Dominator analysis with simplifying asserting clauses . . . . . 70

4.8 Implication graph of Example 4.11 . . . . . 74

4.9 Implication graphs of Example 4.13 . . . . . 75

4.10 Implication graphs of Example 4.13 . . . . . 75

4.11 Algorithm for generating a new conflict clause based on recursive dominators. . . . . 77

4.12 CPU time by PrecoSAT and CirCUs with and without proposed techniques . . . . . 78

4.13 The contribution of proposed technique DOM . . . . . 79

4.14 The contribution of proposed technique DOMSUB . . . . . 80

4.15 The contribution of proposed technique DSSCL . . . . . 81

4.16 The contribution of proposed technique DCCL . . . . . 82

4.17 Effect of proposed techniques on (a)the number of subsuming dominator clauses per dominator computation: GEOMETRIC MEAN = $1.11$, $p$-value = $0.001$; (b)the number of literals per conflict clause: GEOMETRIC MEAN = $0.6$, $p$-value = $2.2 \cdot 10^{-16}$ . . . . . 83

# Chapter 1

# Introduction

## 1.1 Background

The last two decades have seen great advances in the performance of satisfiability solvers for propositional logic, in particular those based on the David-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62, MS99, MMZ+01, ES03]. These solvers have evolved in symbiotic relationship with many Electronic Design Automation (EDA) applications including model checking [BCCZ99, McM02, McM03, LS06, Li06], logic synthesis [MB89], testing [SBV96], and timing analysis.

Progress has been made both in the pruning of the search space [MS99] and in the efficient implementation of the basic operations like deductions [MMZ+01]. Here we are concerned with techniques that transform a Conjunctive Normal Form (CNF) formula, either as a preprocessing step [EB05, SE05, ZKKSV06] or during the DPLL procedure. These transformations should be relatively inexpensive and produce formulae on which the DPLL procedure runs faster than on the original ones.

Reducing the size of the formula is a common objective of transformations. For instance, a set of clauses is **redundant** if a proper subset represents the same function. A subsumed clause (i.e., a clause implied by another) is redundant, and the cost of many SAT solver operations decreases with a smaller formula. Hence, removing subsumed clauses is usually beneficial. However, not all redundant clauses can be removed without negative effect on the speed of the solver.

In this thesis, I introduce two notions that help in the design and evaluation of formula transformations. The first is **deductive power** of a CNF formula. The higher this power, the more consequences the DPLL procedure can deduce from each of its decisions; hence, the more effective is the pruning of the

search space. The second notion is **proof conciseness**. It reflects the fact that the DPLL procedure progresses through the search space by proving that parts of that space contain no satisfying assignment and recording such findings in the form of new clauses and their derivations. More concise proofs are faster to build and usually more effective at pruning further search.

To see how deductive power may help in the analysis of SAT solvers, consider clause recording, which 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. The conflict clauses learned by SAT solvers are by definition redundant, but they always improve the deductive power of a CNF formula.

Clauses that are subsumed by other clauses slow down the implication process, but do not help the solver in pruning the search space. I show that they never improve deductive power. Therefore, preprocessing often removes them to accelerate implications. On the other hand, removing literals from clauses may increase the deductive power of a formula. I study in detail several approaches to such elimination, both as preprocessing and during DPLL.

Literal removal procedures are often based on resolution. In addition, resolution may be applied to eliminate variables from the formula. Since the elimination of variables may increase the number of clauses, it is usually applied with restraint [SP04, EB05]. Deductive power is not guaranteed to improve either. Instead, the main benefit of variable elimination is the decrease in the average number of decisions and implications required to produce a conflicting assignment. Not only conflicts occur sooner, but their analysis is faster, and the learned clauses tend to prune larger portions of the search space.

In this thesis I analyze existing techniques that increase deductive power or generate more concise implication graphs and I propose two new ones. I show how to detect subsumptions during resolution during both preprocessing and conflict analysis with minimal overhead. The proposed on-the-fly subsumption check can be applied to both regular and strong [JS06] conflict analysis. I show how this inexpensive check is used to improve deductive power at three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis. I then describe a **distillation** algorithm that asserts the negations of clauses to remove

redundant literals and possibly derive new clauses. Unlike previous approaches, this distillation procedure may replace a clause with the resolvent of two or more existing clauses without explicitly deriving any such resolvents in advance. I show that distillation increases deductive power and shortens implication graphs. Experiments show that the presented techniques speed up our SAT solver. Variable elimination works primarily by shortening the implication graphs, while other transformations mainly improve deductive power.

Despite recent progress in DPLL-based SAT solvers, more improvements can be achieved with several extensions of existing formula transformation techniques. One example concerns conflict clauses learned by SAT solvers. They are redundant definition, but I have shown that they always improve the deductive power of a CNF formula. In previous work [Nad09, SB09, Pre], different approaches to produce learned clauses from one based on UIP have been proposed to more efficiently prune the search space. **Assignment shrinking** [Nad09] applies the assignments again in the newly found conflict clause until a new conflict occurs. This may produce a new smaller conflict clause. In [SB09, Pre], a clause is learned from the analysis for a single dominator during the implication process. Since the clause contains only two literals of which one is for the dominator and the other is for the implied literal, its addition is effective in shortening the implication graph. Even though those schemes have empirically proved to help a SAT solver prune more of the search space, a formal analysis of their effectiveness has not been attempted. In this thesis, I investigate them for the improvement of either deductive power or proof conciseness. In particular, I also study how efficiently extend the learning scheme based on dominators for further improvement of deduction in SAT solvers.

## 1.2 Thesis Contributions

This thesis deals with the effectiveness of deduction procedure in propositional satisfiability problems. To this aim, I followed three research directions.

In DPLL-based SAT solvers, deduction based on **modus ponens** plays a key role in boosting efficiency by finding what literals are implied by the current partial assignment. Since this deduction procedure is sound but not complete, its effects depend on how the CNF input formula is presented to it. This motivates

techniques that transform a CNF formula, either as preprocessing step [EB05, SE05, ZKKSV06, HS07] or during the DPLL procedure [HS09, HSJ10, SB09, Pre]. These transformations should be relatively inexpensive and produce formulae on which the DPLL procedure runs faster than on the original ones.

To achieve this goal several important research items are identified. They are briefly summarized as follows.

- I have introduced two notions that help in the design and evaluation of formula transformations. The first is **deductive power** of a CNF formula. It is motivated by the observation that the more consequences the DPLL procedure can deduce from each of its decisions, the more effective the pruning of the search space. The second notion is **proof conciseness**. It reflects the fact that the DPLL procedure progresses through the search space by proving that parts of that space contain no satisfying assignment and recording such findings in the form of new clauses.

- Modern DPLL-based SAT solvers heavily rely on various CNF transformation techniques to enhance the effectiveness in pruning the search space. These transformations include simplifying clause data base and clause recording. I have formally characterized these transformation techniques in terms of deductive power and proof conciseness. In addition, I have proved their effectiveness in speeding up the SAT solver by in-depth analysis of experimental results.

- I have developed efficient transformations that aim at increasing the deductive power of a CNF formula and generating more compact implication graphs. The procedure of **clause distillation** at the preprocessing stage and **on-the-fly simplifications** based on self-subsumption during DPLL considerably speed up the SAT solver by increasing deductive power. On the other hand, the transformation based on variable elimination works mainly by reducing the number of resolution steps required in conflict analysis, that is, by producing more concise proofs.

- Despite recent progress in DPLL-based SAT solvers, more improvements can be achieved with several extensions of existing formula transformation techniques. One example concerns a clause learned from the analysis for a single dominator during the implication process. Since the clause

contains only two literals of which one is for the dominator and the other is for the implied literal, its addition is effective in shortening the implication graph. Even though those schemes have empirically proved to help a SAT solver prune more of the search space, a formal analysis of its effectiveness had not been attempted. I have investigated it for the improvement of either deductive power or proof conciseness. In particular, I also proposed how to efficiently extend the learning scheme based on dominators for the generation of even more compact implication graphs.

## 1.3 Thesis Organization

The organization of this thesis is as follows.

Chapter 2 covers background and definitions related to the satisfiability problems for propositional formulae that are pertinent to my work.

Chapter 3 presents several approachs to make the deduction procedure more efficient. In DPLL-based SAT solvers, deduction based on **modus ponens** plays a key role in boosting efficiency by finding what literals are implied by the current partial assignment. Since this deduction procedure is sound but not complete, its effects depend on how the CNF input formula is presented to it. This motivates techniques that transform a CNF formula, either as preprocessing step or during the DPLL procedure. These transformations should be relatively inexpensive and produce formulae on which the DPLL procedure runs faster than on the original ones. In this chapter, I have introduced two notions that help in the design and evaluation of formula transformations. The first is **deductive power** of a CNF formula. It is motivated by the observation that the more consequences the DPLL procedure can deduce from each of its decisions, the more effective the pruning of the search space. The second notion is **proof conciseness**.

In Chapter 4, the **dominator-based CNF simplification** techniques are presented. A clause with two literals may be derived during the deduction process. Since such a clause tends to shorten the implication graph, it can be characterized in terms of the notions defined in Chapter 3. I have extended dominator analysis with an efficient self-subsumption check. I also show how the information collected by dominator analysis can be used to detect redundancies in the satisfied clauses and, more importantly, how it can be used to produce supplemental conflict clauses. I have characterized these transformations in terms of deductive

power and proof conciseness. My experiments show that the main advantage of dominator analysis and its extensions lies in improving proof conciseness.

Chapter 5 includes the conclusions of this thesis and some future research directions.

## 1.4 Related Work

Most powerful modern SAT solvers [zCh, Jer, Satb, Rsa, Pic, Pre, Bar, SATc] employ variants of the DPLL procedure, and recently they have achieved great improvement in several ways other than CNF transformation techniques like efficient implementations based on two-watched literal schemes [Zha97, MMZ$^+$01, Bie08b] for faster implication process, heuristics to select the decision variables [Lib00, GN02, HB03, JS04a], and restart techniques [GSK97, Bie08a, PD09].

More recently, there has been considerable interest in efficient translation techniques from the original problem to CNF formula, which are called **SAT encoding problems** [Vel04, ES06, EMS07, MV07]. In particular, [EMS07] explores the preprocessing stage of SAT for circuit problems using recent logic synthesis techniques. In contrast with preprocessing steps of the DPLL-based SAT solver, SAT encoding techniques are applied to generate simpler CNF formulae to be processed by the SAT solver.

The notion of deductive power that is defined in this thesis is related to, but distinct from the **deducibility** of [VH05], which counts the number of implications due to assignment to a variable of a CNF formula.

A problem related to preprocessing of a CNF formula is the preprocessing of conflict clauses in an incremental SAT solver. An incremental solver is given a sequence of SAT instances and tries to use clauses learned in earlier instances to expedite the solution of later instances. If each instance is obtained from the previous by addition of new clauses, all clauses learned by the solver can be **forwarded** to the new instance. However, in the general case, clauses must be validated before they can be forwarded. In [JS04b], a process called **distillation** was proposed, which forwards a clause derived from a previously learned clause $\gamma$ only if asserting the negation of $\gamma$ causes a conflict in the new instance. In [HS07] and [HSJ10] I apply distillation to preprocessing the original clauses of a CNF formula and we characterize this approach from the point of view of deductive power.

**Assignment shrinking** [Nad09] can also be seen as on-the-fly distillation of selected conflict clauses. At the end of conflict analysis, the algorithm of [Nad09] backtracks to a level preceding the backtracking level to undo some assignments in the conflict clause. It then applies those assignments again in a different order until a new conflict occurs. This may produce a new smaller conflict clause. Since this is a potentially expensive technique, its invocation is controlled by a heuristic.

Previous work besides [Nad09] has addressed the quality of conflict clauses [ZMMM01, ES03, SE05, JS06, SB09]. In particular, the clause minimization algorithm of [SE05, SB09] traverses the implication graph beyond the 1-UIP to remove literals in the conflict clause that are implied by other literals. The strong conflict analysis proposed in [JS06] generates a second conflict clause that is often more effective than a regular conflict clause of [ZMMM01] in escaping regions of the search space where the solver would otherwise linger for a 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 earlier. Unlike the on-the-fly subsumption to be discussed in Section 3.2, these earlier techniques focus on simplification of the conflict-learned clauses, instead of looking at all clauses appearing in the resolution graph.

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. The on-the-fly simplification algorithm used in [Zha05] can detect the subsumed clause with a **one watched literal** scheme, when a new clause is generated by conflict analysis. While the one watched literal scheme is efficient, the removal of subsumed clauses does not improve deductive power and does not produce more concise proofs. The practical ability of this technique to speed up SAT solvers was not the focus of [Zha05] and remains to be established.

# Chapter 2

## Propositional Satisfiability Solvers

The propositional satisfiability (SAT) problem is of central importance in various areas of computer science, including artificial intelligence, hardware design, electronic design automation, and verification. The last two decades have seen great advances in the performance of satisfiability (SAT) solvers for propositional logic, in particular those based on the David-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62, MS99, MMZ$^+$01, ES03]. These solvers have found many applications in electronic design automation (EDA) including model checking, logic synthesis, testing, and timing analysis. Especially in the formal verification area, the SAT solving algorithms have helped make Bounded Model Checking (BMC [BCCZ99]) a widely used alternative to BDD-based model checking. Progress has been made in the pruning of the search space [MS99] and in the efficient implementation of the basic operations like deductions [MMZ$^+$01] during the DPLL procedure; for instance, non-chronological backtracking and conflict analysis based on unique implication points (UIPs), and efficient implication based on two-watched literal scheme [Zha97, MMZ$^+$01], decision variable heuristics, e.g., Variable State Independent Decaying Sum (VSIDS) heuristic [MMZ$^+$01] and conflict cluase based heuristic in BerkMin [GN02], and effective constraints database management.

This chapter covers backgrounds and definitions related to the satisfiability problems for propositional formulae that are pertinent to my work.

## 2.1 Propositional Satisfiability Problems

Variables that can take truth values true and false are called **Boolean variables**. Letters $a$, $b$, $c$, ... will be used for Boolean variables. Also, **Boolean connectives** are conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). The **propositional formulae** in the standard Boolean connectives are inductively defined as follows.

- false and true are propositional formulae.
- Every Boolean variable is a propositional formula.
- If $F$ is a propositional formula, then $\neg(F)$ is a propositional formula.
- If $F_1$ and $F_2$ are propositional formulae, then $(F_1)\vee(F_2)$ and $(F_1)\wedge(F_2)$ are propositional formulae.

We drop outer parentheses, "(" and ")", when no ambiguity arises. Other standard Boolean connectivities can be defined as abbreviations. For instance, exclusive-OR of variables $a$ and $b$ is defined by $(\neg a \wedge b) \vee (a \wedge \neg b)$, and function "$a$ implies $b$" ($a \rightarrow b$) is defined by $\neg a \vee b$. An **assignment** to the set of variables $V$ of CNF formula $F$ 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. Formula $F$ is said to be **satisfiable** if there is any satisfying assignment for $F$. Otherwise, it is said to be **unsatisfiable**. The **satisfiability problem** (SAT) is the decision problem to determine whether a propositional formula is satisfiable or not.

**Example 2.1.** *Considering the following propositional formula:*

$$F = (\neg a \wedge b) \vee (a \wedge \neg c).$$

*Formula $F$ is satisfiable because the assignment $a = 0$, $b = 1$, and $c = 0$ makes $F$ become* true. □

## 2.2 Representations

We consider several ways of representing a propositional formula. Conjunctive Normal Form (CNF) is often used because it can be manipulated efficiently and because constraints of various provenance are

easily translated into it.

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. The CNF formula

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

therefore corresponds to the following propositional formula:

$$(\neg a \vee c)_1 \wedge (\neg b \vee c)_2 \wedge (\neg a \vee \neg c \vee d)_3 \wedge (\neg b \vee \neg c \vee \neg d)_4,$$

where subscripts indicate clause numbers for ease of reference; in this thesis $c_i$ represents the clause that is numbered by $i$.

SAT is a central problem in complexity theory, and several special cases have been studied. The problem called "3-SAT" in which each clause in CNF formula has exactly three literals was the first problem proved to be **NP-complete** [Coo71].[1] The general CNF SAT problem is as hard as the 3-SAT problem. On the other hand, the "2-SAT" problem, in which each clause is restricted to have at most two literals, can be solved in polynomial time.

SAT problems can also become easier if the formulae are restricted to Disjunctive Normal Form (DNF), that is, disjunctions of terms; each term is a conjuction of literals. This is because such a formula is satisfiable if and only if some term is satisfiable, and a conjunctive term is satisfiable if and only if it does not contain both $a$ and $\neg a$ for variable $a$. This can be checked in polynomial time.

Propositional formulae can be represented in Boolean circuit forms. One example is the **And-Inverter Graph** (AIG) [KGP01], where each internal node $\nu$ has exactly two predecessors; if the predecessor variables are $a$ and $b$, its function $\varphi(\nu)$ is one of $a \wedge b$, $a \wedge \neg b$, $\neg a \wedge b$, and $\neg a \wedge \neg b$. Even if the AIG is not a **canonical representation**, that is, it does not provide a unique representation of a given function, it is often used because it allows a variety of simplification techniques that may significantly speed

[1] Stephen Cook and Leonid Levin discovered certain problems in **NP**, the class of languages decidible in nondeterministic polynomial time, whose complexity is related to that of the whole class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete[Sip96].

up subsequent analyses. The SAT problem of the input formula can be written in AIG formats [GAG+02], and this is also NP-complete [WCC09].

Canonical circuit representations, like Binary Decision Diagrams (BDDs), are useful to manipulate large propositional formulae. A BDD representing function $F$ consists of two types of nodes: **terminal nodes** that are labeled by either true or false, and **internal nodes** that are labeled with variable names. Following a path from the root to a terminal node evaluates $F$ for a given assignment to the input variables. That is, the label of the terminal node is the result of the evaluation. Each internal node represents the function $(a \wedge f_a) \vee (\neg a \wedge f_{\neg a})$ (with $f_a \neq f_{\neg a}$) [2] , where $a$ is the **control variable** of the internal node, and $f_a$ and $f_{\neg a}$ are the functions of the successors of the internal node. In popular usage, BDDs refer to reduced and ordered BDDs.

Different representations of propositional formulae have peculiar advantages regarding SAT problems. For the representations like DNF and BDDs, the hurdle lies in converting the SAT problem into the required form; if this can be accomplished, satisfiability is then trivial. In particular, with BDDs, determining whether a function is satisfiable requires constant time, while a satisfying assignment, if it exists, can be found in $O(n)$ time, where $n$ is the number of variables. Since converting a Boolean circuit into a BDD may incur an exponential blow-up, naive application of BDDs to SAT lacks robustness. On the other hand, there exist numerous cases in which a proper mix of canonical (e.g., BDDs) and non-canonical representations (e.g., CNF or AIG) is very beneficial [KK97, BS98]. This is true, in particular, of SAT solvers based on search, and applied to instances for which compact search trees do not exist or are hard to find.

## 2.3 CNF Formulae

In this thesis I assume that the input to the SAT solver is a formula in CNF defined in Sec.2.2. 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, $a \wedge b \wedge \neg d$. Given CNF formulae $F_1$ and $F_2$ over variable set $V$, $F_1$ **implies** $F_2$, written $F_1 \rightarrow F_2$, if all the assignments to $V$ that satisfy $F_1$ satisfy $F_2$; $F_1$ and $F_2$ are **equivalent** if $F_1 \rightarrow F_2$ and $F_2 \rightarrow F_1$. A clause

[2] This is known as the **expansion theorem** of $f$ with respect to $a$.

$\gamma$ 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, and also say that the antecedent implies its asserted literal. Clause $\gamma_1$ subsumes clause $\gamma_2$ if $\gamma_1 \subseteq \gamma_2$.

**Example 2.2.** *Given two clauses* $(a \vee b \vee c)_1$ *and* $(a \vee c)_2$, $c_2$ *subsumes* $c_1$. □

Given $\gamma_1 = \gamma_1' \cup \{l\}$ and $\gamma_2 = \gamma_2' \cup \{\neg l\}$, the **resolution** of the two clauses over $l$ produces the **resolvent** $\gamma_1' \cup \gamma_2'$, which is implied by $\{\gamma_1, \gamma_2\}$.

**Example 2.3.** *Given the following propositional formula:*

$$(a \vee \neg b \vee c)_1 \wedge (\neg a \vee d)_2,$$

*resolving* $c_1$ *and* $c_2$ *over* $a$ *yields the resolvent* $(\neg b \vee c \vee d)$. □

Clauses $\gamma_1$ and $\gamma_2$ are in **self-subsumption** relation if their resolvent subsumes $\gamma_1$. If $F$ contains clauses $\gamma_1$ and $\gamma_2$ such that $\gamma_1$ is in self-subsumption relation with $\gamma_2$, the CNF $F'$ obtained by replacing $\gamma_1$ with the resolvent of $\gamma_1$ and $\gamma_2$ is equivalent to $F$.

**Example 2.4.** *Given the following formula:*

$$F = (a \vee \neg b \vee c)_1 \wedge (a \vee b)_2 \wedge (c \vee d)_3,$$

*resolution of* $c_1$ *and* $c_2$ *gives the resolvent* $\gamma = (a \vee c)$ *that subsumes* $c_1$, *that is* $c_1$ *is in self-subsumption relation with* $c_2$. *Then, formula*

$$F' = (a \vee c)_4 \wedge (a \vee b)_2 \wedge (c \vee d)_3$$

*that is obtained by replacing* $c_1$ *in* $F$ *with* $\gamma$ *is equivalent to* $F$. □

## 2.4 CNF SAT Solvers

SAT algorithms for CNF formulae can be categorized as **incomplete** or **complete** algorithms. Incomplete algorithms do not guarantee that they will eventually either report a satisfying assignment or prove the given formula unsatisfiable. Incomplete methods are usually based on stochastic local search

[GW93, SKC93], while the complete algorithms are based on an exhaustive **branching** and **backtracking search**. GSAT [SLM92] and Walksat [SKC95] played a key role in the success of local search in SAT. GSAT is based on a randomized local search technique [LK73, Pap94]. They start by assigning a random value to each variable. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, a variable is flipped and the above is then repeated until all the clauses are satisfied. Such SAT solvers based on stochastic local search perform better on random SAT instances rather than on structured instances like the ones obtained from real verification problems.

Given an input formula $F$, complete algorithms either produce a satisfying assignment for $F$ or prove that $F$ is unsatisfiable. Most complete methods remain variants of a procedure introduced several decades ago: the DPLL procedure. The DPLL procedure performs a backtrack search in the space of partial truth assignments. The key feature of DPLL is the efficient pruning of the search space. My work only concerns DPLL-based SAT solvers, and the following section is devoted to reviewing these complete SAT solvers. Another compete approach is Staålmarck's method [SS98], which is based on the **dilemma rule**. This rule opens two branches and assumes a formula to be true in one branch and false in the other. The branches are eventually merged and the intersection of the two branches is kept: for variable $x$ of the formula, consequences that are gained both from $x$ and $\neg x$ must be true independently of $x$. This proof procedure has been successfully used in industrial verification problems [Bor97, Bor98, CG05].

## 2.5 The DPLL Procedure

Resolution can be used to **eliminate** variable $l$ from a CNF formula. One replaces the clauses that contain either $l$ or $\neg l$ with all their resolvents. If, for example, variable $b$ is to be eliminated from $F$ and $a \vee b$, $\neg b \vee c$ and $\neg b \vee d$ are the only clauses of $F$ containing $b$, then they are replaced by all the resolvents over $b$, namely $a \vee c$ and $a \vee d$. The resulting CNF formula is **equisatisfiable** to $F$; that is, it is satisfiable if and only if $F$ is. Therefore, repeated application of variable elimination results in a decision procedure for CNF satisfiability that is known as Davis-Putnam (DP) procedure [DP60]. If in some iteration, one resolves $\{l_i\}$ and $\{\neg l_i\}$, then the empty clause is produced and the CNF formula is unsatisfiable. Otherwise, once all variables are eliminated, no clauses are left and the formula is satisfiable. The DP procedure often produces

```
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   }
```

Figure 2.1: GRASP_DPLL algorithm.

too many resolvents; in applications, it has been mostly replaced by the Davis-Putnam-Loveland-Logemann (DPLL) procedure [DLL62] that is a search algorithm based on branching and backtracking.

Many successful SAT solvers are based on the DPLL procedure, whose modern incarnations are described by the pseudocode of Fig. 2.1. The solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. While extending the partial assignment, the DEDUCE procedure tries to detect as many implications as possible by using asserting clauses.

A derivation $F \cup A \vdash l$ ($l$ is implied by CNF formula $F$ together with partial assignment $A$) is conveniently represented by its **implication hypergraph**. An implication hypergraph has a vertex for each literal in $A$ and each asserted literal; it has a directed hyperedge (i.e., a set of directed edges) for each asserting clause with more than one literal that is involved in the derivation. The implication hypergraph may also have a special conflict node, named $\kappa$, to be described later.

**Example 2.5.** *Consider the following CNF formula:*

$$F = (a \vee b)_1 \wedge (a \vee c)_2 \wedge (a \vee d)_3 \wedge (\neg b \vee \neg c \vee e)_4 \wedge (\neg c \vee \neg d \vee e)_5.$$

*Under partial assignment $\{\{\neg a\}\}$, literals $b$, $c$, and $d$ are implied by $c_1$, $c_2$, and $c_3$ of $F$ and $e$ is then implied by either the fourth or the fifth clause. The implication hypergraph for $F \cup \{\neg A\} \vdash e$ is shown in Fig. 2.2.*

*Hyperedges are labeled with antecedent clause numbers. The number of edges in a hyperedge equals the number of literals in the corresponding clause minus one. In the context of DPLL, each node is annotated*

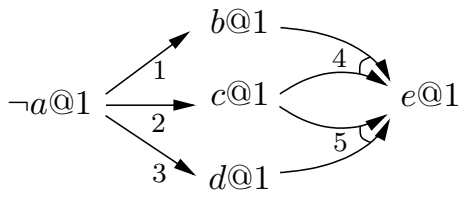



Figure 2.2: Implication hypergraph for Example 2.5.

*with a **decision level** (the number following the @ sign in the figure). For the literals in the assignment, the level is the order in which they are asserted. For a literal asserted by a clause, it is the highest level of its predecessors. Literals asserted by unit clauses have decision level equal to 0. The hypergraph of Fig. 2.2 shows that $e$ can be implied in two different ways.*

SAT solvers usually keep track of just one way to assert a literal. Hence, they use an **implication graph** rather than a hypergraph. The implication graph corresponds to a subgraph of the implication hypergraph in which every vertex has at most one incoming hyperedge.

If extension of the assignment produces a conflict—that is, a clause, which is said to be **conflicting**, has all its literals assigned to false—the solver analyzes the conflict and **backtracks** accordingly.

A conflict results in the presence of the conflict node $\kappa$ in the implication hypergraph, with a hyperedge joining it to the negations of the literals of the conflicting clause. Multiple conflicts may be derived from the same partial assignment. Hence, the conflict node may have multiple incoming hyperedges. SAT solvers, once again, usually work with a subset of the hypergraph that contains only one hyperedge into each node.

## 2.6 Conflict Analysis

Conflict analysis [MS96] leads to learning a **conflict learned clause** (in short, conflict clause), that is, a clause $C$ with the following properties: given CNF formula $F$ and assignment $A$,

- $F \rightarrow \{C\}$,
- $C \notin F$, and

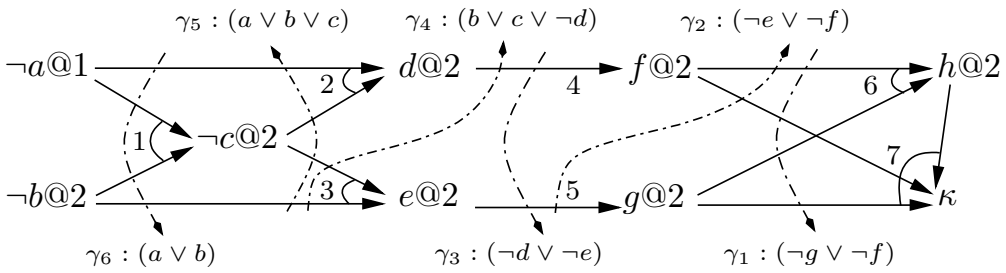



Figure 2.3: Implication graph for the first conflict of Example 2.6.

- $C$ is conflicting under $A$.

A conflict clause is computed by resolving the conflicting clause with the antecedents of literals that appear in it. The antecedents are processed in reverse order in which the literals they assert were implied. The conflict clause can be added to the given SAT instance to prevent the examination of regions of the search space that contain no solutions.

**Example 2.6.** *Consider the following formula:*

$$\begin{aligned} F \;\; = \;\; & (a \vee b \vee \neg c)_1 \wedge (a \vee c \vee d)_2 \wedge (b \vee c \vee e)_3 \wedge (\neg d \vee f)_4 \wedge (\neg e \vee g)_5 \wedge \\ & (\neg f \vee \neg g \vee h)_6 \wedge (\neg f \vee \neg g \vee \neg h)_7 \;\; . \end{aligned}$$

*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 2.3 shows the implication graph that is derived when the following rule is applied: the earliest asserting clauses adds a new implied literal into the graph. The implication graph shows the literals implied up to the current decision level. The implications make clause $c_7$ conflicting as shown by the conflict node $\kappa$. Conflict analysis is illustrated in Fig. 2.4. The implication graph of Fig. 2.3 also shows each resolvent $\gamma_i$ of the resolution graph of Fig. 2.4 that the conflict analysis generates while traversing backward the implication graph from the conflicting clause $c_7$. Every resolvent and, hence, every conflict clause corresponds to a* ***cut*** *in the implication graph. The literals having outgoing edges that cross the cut comprise a sufficient reason for the conflict.*

Most conflict analysis algorithms terminate as soon as they find a clause containing a **Unique Implication Point** (UIP), that is, a single literal asserted at the current level. There may be more than one

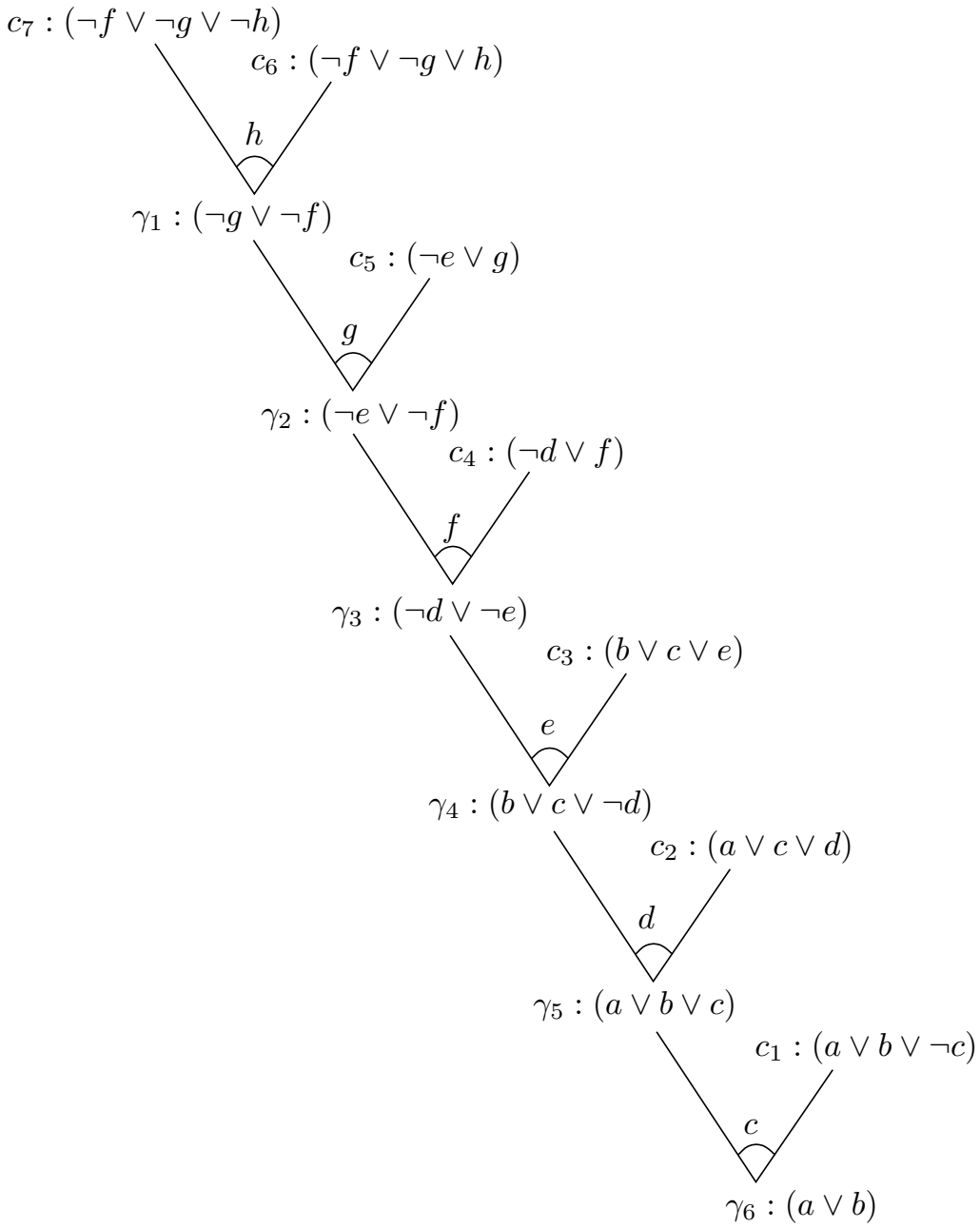



Figure 2.4: Resolution graph of conflict analysis for Example 2.6.

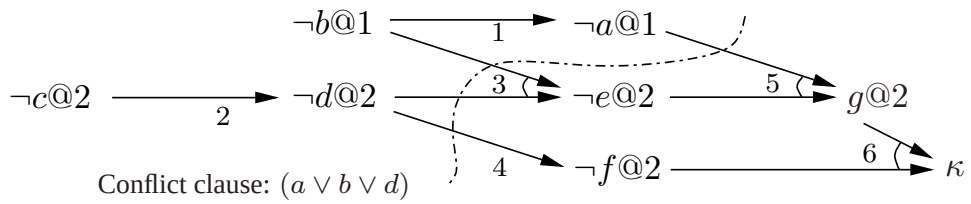



Figure 2.5: Conflict clause computed on an implication graph.

cut containing a UIP in the implication graph. Specifically, the cut closest one to the conflicting clause in the implication graph contains the **first** UIP (in short, 1-UIP). In [ZMMM01], conflict clauses based on the 1-UIP have been empirically shown effective in pruning the search space. In Example 2.6, since $\gamma_6$ contains the only UIP, that is literal $b$, it is chosen as conflict clause.

When the 1-UIP is far from the conflict in the implication graph, the conflict clause may not be effective in preventing the SAT solver from producing many conflicts involving the same clause. **Strong conflict analysis** [JS06] 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 1-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 the 1-UIP.

A SAT solver can simplify a conflict clause by dropping the literals implied at decision level $0$ from the conflict clause. In [EMS07, SB09], this **conflict clause minimization** method has been extended to remove a literal that is implied at a decision level higher than $0$ as long as it is implied by other literal in the conflict clause. This procedure can be applied to the results of both standard and strong conflict analysis. It applies resolution to the conflict clause and the antecedent clauses.

**Example 2.7.** *In Fig. 2.5, the literal $a$ is lifted from the conflict clause, since the conflict clause is subsumed by $(b \vee d)$, which is the resolvent of the antecedent of $a$ and the conflict clause itself. In other words, $a$ is removed because $\neg a$ is implied by $\neg b$ in $(\neg a \vee b)$.*

## 2.7 Proof of Unsatisfiability

Once conflict analysis adds a new conflict clause, it computes the **backtrack** level, named $blevel$, that is the highest decision level of the literals in the conflict clause except for the UIP. After conflict analysis, the DPLL procedure backtracks to $blevel$, where the newly recorded conflict clauses is asserting. If the conflict occurs while propagating an assignment at decision level $0$, then the DPLL procedure computes -1 as $blevel$. This means that there is no way to resolve the conflict, and the procedure declares the formula unsatisfiable. When a CNF formula is unsatisfiable, a DPLL-based SAT solver can generate **a proof of unsatisfiability** [GN03, ZM03] in the form of a **resolution graph**. A resolution graph is a directed acyclic graph like the one of Fig.2.4. Each node in the graph represents a clause; the sources represent original clauses, and the inner nodes represent the resolvents of their immediate predecessors. In a proof of unsatisfiability, there is a sink node associated with the empty clause. The sources identify a subformula often referred to as an **unsatisfiable core** [LMS04, OMA$^+$04]. To generate a proof of unsatisfiability, the SAT solver keeps track of the derivations of conflict clauses. When the empty clause is learned as the result of a level-0 conflict, the solver recursively replaces each conflict clause with its derivation. The process starts from the empty clause and terminates when only clauses of the original formula are left. In particular, an unsatisfiabile clause set $F' = \{c_1, \ldots, c_n\} \subseteq F$ is **minimally unsatisfiable** if any proper subset of $F'$ is satisfiable. Both problems of finding a unsatisfiable core and proof of unsatisfiability have been researched in last few years due to its increasing importance in formal verification [AKMM03, KOSS04, GLST05, McM03, LS06, Li06]. Hence, a new technique added to a SAT solver should not interfere with its ability to produce either.

To apply CNF transformations without interrupting proofs of unsatisfiability, the SAT solver, like CirCUs [JAS04, VIS] can move every modified clause to a separate database during DPLL. For instance, if a clause is removed by variable elimination or simplification, it is stored as a reason to the derivation of the resolvents or of the simplified clause. In the context of a **dominator clause** to be discussed in Chapter 4, the solver keeps track of the antecedents involved in the dominator computation, as it does for a conflict clause.

## 2.8 Preprocessing

The GRASP_DPLL procedure is often applied after a **preprocessing** phase, which attempts to remove redundant clauses and literals from the given formula. SatELite [EB05, Satb] simplifies a CNF formula by removing clauses subsumed by other clauses, by simplifying clauses that are in self-subsumption relation with other clauses, and by eliminating variables. By contrast, [HS07] proposed a new prepocessing algorithm where the CNF formula is distilled by analyzing the implication graphs to generate the improved clauses.

**Equivalent variable substitution** [Bra01] is another method to simplify the input formula. If formula $F$ contains two clauses $c_1 = (\neg p \vee q)$ and $c_2 = (p \vee \neg q)$, literals $p$ and $q$ are equivalent in $F$, that is, $F \cup \{p\} \vdash_{\mathcal{D}} q$ and $F \cup \{\neg p\} \vdash_{\mathcal{D}} \neg q$. In the standard deduction procedure of DPLL-based SAT solvers, this can be found by checking cycles of implications, but this may spend considerable time while searching and comparing these two-literal clauses. Variables in equivalence relation belong to the same **equivalence class**. In an equivalence class, a representative is selected and it substitutes for all other variables in the clause database. This yields fewer variables, and allows the SAT solver to explore a reduced search space.

Preprocessing may reduce the workload of a SAT solver. However, there exists a trade-off between effect and cost of the preprocessing techniques, because it is, in most cases, too time consuming to remove all the redundancies in the given SAT instance or eliminate all variables.

# Chapter 3

## Increasing the Efficiency of the Deduction Procedure

The Progress of DPLL-based SAT solvers of Sect. 2.5 has been made both in the pruning of the search space [MS99] and in the efficient implementation of the basic operations like deductions [MMZ+01]. Here we are concerned with techniques that transform a CNF formula, either as a preprocessing step [EB05, SE05, ZKKSV06] or during the DPLL procedure. These transformations should be relatively inexpensive and produce formulae on which the DPLL procedure runs faster than on the original ones.

Reducing the size of the formula is a common objective of transformations. For instance, a set of clauses is **redundant** if a proper subset represents the same function. A subsumed clause (i.e., a clause implied by another) is redundant, and the cost of many SAT solver operations decreases with a smaller formula. Hence, removing subsumed clauses is usually beneficial. However, not all redundant clauses can be removed without negative effect on the speed of the solver.

We introduce two notions that help in the design and evaluation of formula transformations. The first is the **deductive power** of a CNF formula. The higher this power, the more consequences the DPLL procedure can deduce from each of its decisions; hence, the more effective is the pruning of the search space. The second notion is **proof conciseness**. It reflects the fact that the DPLL procedure progresses through the search space by proving that parts of that space contain no satisfying assignment and recording such findings in the form of new clauses and their derivations. More concise proofs are faster to build and usually more effective at pruning further search.

To see how deductive power may help in the analysis of SAT solvers, consider clause recording, which 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. The conflict clauses learned by SAT solvers are by definition redundant, but they always improve the deductive power of a CNF formula.

Clauses that are subsumed by other clauses slow down the implication process, but do not help the solver in pruning the search space. We show that they never improve deductive power. Therefore, preprocessing often removes them to accelerate implications. On the other hand, removing literals from clauses may increase the deductive power of a formula. We study in detail several approaches to such elimination, both as preprocessing and during DPLL.

Literal removal procedures are often based on resolution. In addition, resolution may be applied to eliminate variables from the formula. Since the elimination of variables may increase the number of clauses, it is usually applied with restraint [SP04, EB05]. Deductive power is not guaranteed to improve either. Instead, the main benefit of variable elimination is the decrease in the average number of decisions and implications required to produce a conflicting assignment. Not only conflicts occur sooner, but their analysis is faster, and the learned clauses tend to prune larger portions of the search space.

In this work we analyze existing techniques that increase deductive power or generate more concise implication graphs and we propose two new ones. We show how to detect subsumptions during resolution during both preprocessing and conflict analysis with minimal overhead. Our on-the-fly subsumption check can be applied to both regular and strong [JS06] conflict analysis. We show how this inexpensive check is used to improve deductive power at three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis. We then describe a **distillation** algorithm that asserts the negations of clauses to remove redundant literals and possibly derive new clauses. Unlike previous approaches, this distillation procedure may replace a clause with the resolvent of two or more existing clauses without explicitly deriving any such resolvents in advance. We show that distillation increases deductive power and shortens implication graphs.

Experiments show that the presented techniques speed up our SAT solver. Variable elimination works primarily by shortening the implication graphs, while other transformations mainly improve deduc-

tive power.

## 3.1 Deductive Power of a CNF Formula

Among the operations performed by a DPLL-based SAT solver, deduction (i.e., the DEDUCE procedure of Fig. 2.1) plays a key role in boosting efficiency by finding what literals are implied by the current partial assignment. Deduction is usually based on **modus ponens**:

$$\frac{P, \neg P \vee Q}{Q}, \tag{3.1}$$

where $P$ and $Q$ are formulae. The rule of modus ponens used in the DPLL procedure of Sect. 2.5 is a specialized form, where $\neg P \vee Q$ is an asserting clause and $Q$ is the asserted literal. In other words, given a clause $\{l_1, \ldots, l_n\}$ and a partial assignment $\{\{\neg l_1\}, \ldots, \{\neg l_{n-1}\}\}$, modus ponens deduces $l_n$. The deduction procedure in a DPLL-based SAT solver repeatedly applies modus ponens to asserting clauses in the given formula until either no new literal is implied, or a clause becomes conflicting. We denote this deduction procedure, which employs modus ponens as the only inference rule, by $\mathcal{D}$. We write $F \vdash_{\mathcal{D}} l$ if the truth of $l$ can be established by repeatedly applying modus ponens to asserting clauses in $F$. We write $F \vdash_{\mathcal{D}} \mathsf{false}$ if procedure $\mathcal{D}$ applied to $F$ finds a conflicting clause. Procedure $\mathcal{D}$ is sound ($F \vdash_{\mathcal{D}} l$ implies $F \vdash l$ and $F \vdash_{\mathcal{D}} \mathsf{false}$ implies $F \vdash \mathsf{false}$) but not complete.

**Example 3.1.** *$\mathcal{D}$ is not sufficient to deduce that*

$$F = (a \vee b)_1 \wedge (a \vee \neg b)_2 \wedge (\neg a \vee c)_3 \wedge (\neg a \vee \neg c)_4$$

*is unsatisfiable; that is, $F \vdash \mathsf{false}$, but $F \nvdash_{\mathcal{D}} \mathsf{false}$. In contrast, if $(\neg a \vee c)$ and $(\neg a \vee \neg c)$ are replaced by $(\neg a)$, then $\mathcal{D}$ would deduce unsatisfiability of*

$$F' = (a \vee b)_1 \wedge (a \vee \neg b)_2 \wedge (\neg a)_5;$$

*that is, that $F' \vdash_{\mathcal{D}} \mathsf{false}$.* □

While $F \cup \{\{p\}\} \vdash q$ is equivalent to $F \cup \{\{\neg q\}\} \vdash \neg p$, $F \cup \{\{p\}\} \vdash_{\mathcal{D}} q$ does not imply $F \cup \{\{\neg q\}\} \vdash_{\mathcal{D}} \neg p$, as illustrated by the next example.

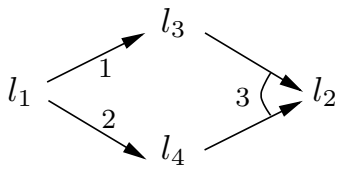



Figure 3.1: Implication graph of Example 3.2.

**Example 3.2.** *Consider the following formula:*

$$F = (\neg l_1 \vee l_3)_1 \wedge (\neg l_1 \vee l_4)_2 \wedge (\neg l_3 \vee \neg l_4 \vee l_2)_3 \ .$$

*Procedure $\mathcal{D}$ deduces literal $l_2$ from $F \cup \{\{l_1\}\}$ as shown in Fig. 3.1 . However, it does not deduce $\neg l_1$ from $F \cup \{\{\neg l_2\}\}$.* □

The rule of modus ponens is a special case of resolution:

$$\frac{P \vee Q, \neg P \vee R}{Q \vee R} \ , \tag{3.2}$$

where $P$, $Q$, and $R$ are formulae. In the DP procedure of Sect. 2.5, $P \vee Q$ and $\neg P \vee R$ are clauses. The deduction procedure that repeatedly applies that inference rule is a sound and complete proof system for CNF. However, as mentioned in Sect. 2.5, this procedure is inefficient in practice: DPLL usually achieves better results by combining an incomplete deduction procedure and search. Since $\mathcal{D}$ is incomplete, its effects depend on how the input is presented to it. In particular, the strengthening of clauses as in Example 3.1 or the addition of new clauses may help. Though the DPLL procedure only needs to be able to detect conflicting assignments to be complete, it is clearly advantageous for a SAT solver based on it to work on a CNF formula that allows more to be done through deduction and less through enumeration. This motivates the following definition. It is convenient to assume that when assignment $A$ is conflicting in $F$, for every literal $l$, $F \cup A \vdash_{\mathcal{S}} l$.

**Definition 3.3.** *For a given sound (but possibly incomplete) deduction procedure $\mathcal{S}$ and two equivalent sets of clauses $F_1$ and $F_2$, let $A$ denote a partial assignment to the variables in $F_1 \cup F_2$. We say that $F_1$ has* ***deductive power*** *greater than or equal to $F_2$ (relative to $\mathcal{S}$) if and only if for every $A$ and any literal $l$ such that $F_2 \cup A \vdash_{\mathcal{S}} l$, $F_1 \cup A \vdash_{\mathcal{S}} l$.*

*If $F_1$ has deductive power greater than or equal to $F_2$ (relative to $\mathcal{S}$) we write $F_2 \preceq_{\mathcal{S}} F_1$. If $F_2 \preceq_{\mathcal{S}} F_1$ and $F_1 \not\preceq_{\mathcal{S}} F_2$, we write $F_2 \prec_{\mathcal{S}} F_1$. If $F_2 \preceq_{\mathcal{S}} F_1$ and $F_1 \preceq_{\mathcal{S}} F_2$, then $F_1$ and $F_2$ have the same deductive power (relative to $\mathcal{S}$), written $F_1 \simeq_{\mathcal{S}} F_2$.*

Note that if $F_2 \preceq_{\mathcal{S}} F_1$ and $A$ is conflicting in $F_2$, then $A$ is also conflicting in $F_1$. In Example 3.1, $F \preceq_{\mathcal{D}} F'$. Since it is reflexive and transitive, $\preceq_{\mathcal{S}}$ is a preorder. In the following, unless otherwise stated, the deduction procedure is assumed to be $\mathcal{D}$ and we write $\preceq$ for $\preceq_{\mathcal{D}}$. We are interested in transformations of a CNF formula that increase, or at least preserve, its deductive power. The following fact proves useful.

**Lemma 3.4.** *Let $F_1$ be a CNF formula and let $\gamma$ be an **implicate** of $F_1$ (that is, a clause implied by $F_1$). Let $F_2$ be the CNF formula obtained from $F_1$ by adding $\gamma$ and optionally removing clauses that are subsumed by $\gamma$. Then, $F_1 \preceq F_2$.*

*Proof.* $F_1$ and $F_2$ are obviously equivalent. Also, adding an implicate to a CNF formula cannot decrease its deductive power. For the removal of subsumed clauses, we need to consider two cases. Let $A$ be an assignment and $l$ be a literal such that $F_1 \cup A \vdash_{\mathcal{D}} l$. Suppose $\gamma'$ is a clause subsumed by $\gamma$ that is used in the derivation of $l$. If $\gamma'$ asserts a literal that is also in $\gamma$, then $\gamma'$ can be replaced by $\gamma$ in the derivation. If $\gamma'$ asserts a literal not in $\gamma$, then all literals of $\gamma$ are false in the derivation and adding $\gamma$ to it leads to a conflict. In both cases, the conditions of Definition 3.3 are met. □

We can use Lemma 3.4 to characterize the change in deductive power of a CNF formula when it is simplified by either subsumption or self-subsumption. As a special case, since any clause of $F$ is an implicate of $F$, one obtains the intuitive result that clauses subsumed by other clauses can be removed from a CNF formula without negatively affecting its deductive power. Adding a clause to $F$ that is subsumed by other clauses does not decrease the deductive power either. Hence, a CNF formula $F_1$ and the formula $F_2$ obtained by removing subsumed clauses from $F_1$ have the same deductive power, i.e., $F_1 \simeq F_2$. Similarly, since every resolvent of clauses of $F$ is implied by $F$, augmenting a CNF with a resolvent may increase deductive power. Therefore, simplification based on self-subsumption may increase the deductive power of a CNF formula, while simplification based on subsumption can only speed up the deduction procedure by reducing the number of clauses to be examined.

**Example 3.5.** *The CNF formula $F_1 = (a \vee b \vee c)_1 \wedge (a \vee \neg b)_2$ can be simplified to $F_2 = (a \vee c)_1 \wedge (a \vee \neg b)_2$ by self-subsumption. Since $F_1 \cup \{\{\neg c\}\} \nvdash_{\mathcal{D}} a$, while $F_2 \cup \{\{\neg c\}\} \vdash_{\mathcal{D}} a$, we have $F_1 \prec F_2$. On the other hand, simplifying $F_3 = F_1 \wedge (a \vee \neg d) \wedge (d \vee c)$ by self-subsumption does not increase deductive power, as one can show by applying Lemma 3.6 below.* □

Since resolution is broadly used in DPLL-based SAT solvers, simplification based on self-subsumption can be applied to various stages of the procedure; in particular, to conflict analysis. This will be dealt with in the next section.

The clause $\{\neg l_0, l_n\}$, where $n \geq 2$, is a **transitive closure clause** of $F$, if there exist literals $l_1, \ldots, l_{n-1}$ such that $\{\neg l_0, l_1\}, \{\neg l_1, l_2\}, \ldots, \{\neg l_{n-1}, l_n\}$ are clauses of $F$. Adding a transitive closure clause to $F$ does not change its deductive power as stated in the following lemma.

**Lemma 3.6.** *Let $F$ be a CNF formula and $\gamma = \{\neg l_0, l_n\}$ be a transitive closure clause of $F$. Let $F' = F \cup \{\gamma\}$. Then, $F' \simeq F$.*

*Proof.* If $\gamma \in F$, there is nothing to prove. Suppose not. By Lemma 3.4, since $\gamma$ is an implicate of $F$, $F \preceq F'$. If $l_n$ ($\neg l_0$) is asserted by $\gamma$ in $F'$, then $l_0$ ($\neg l_n$) must be true; therefore $l_n$ ($\neg l_0$) is also implied by the sequence of clauses $\{\neg l_0, l_1\}, \ldots, \{\neg l_{n-1}, l_n\}$ ($\{\neg l_{n-1}, l_n\}, \ldots, \{\neg l_0, l_1\}$) in $F$. Therefore, $F' \preceq F$. □

While adding a transitive closure clause of the implications does not affect deductive power, it may help the solver by shortening the implication graph. A more concise implication graph may benefit the procedures that work on it. For instance, the deduction procedure may identify a conflicting clause more quickly, and conflict analysis may resolve fewer antecedents. On the other hand, adding clauses to the CNF database indiscriminately may substantially slow down the deduction procedure. To prevent this, a supplemental clause should be generated only when its usefulness is established by an effective criterion (i.e., strong conflict analysis).

Adding a clause that is the resolvent of other clauses may either increase deductive power or shorten the implication graph. Adding a transitive closure clause may lead to a more concise implication graph. On the other hand, some clauses may never become asserting and therefore never appear in an implication hypergraph, as shown in the next example.

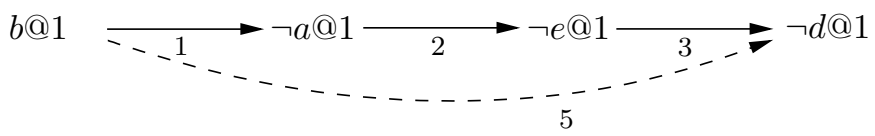



Figure 3.2: Implication graph of Example 3.7.

**Example 3.7.** *Consider the following CNF formula:*

$$F = (\neg a \vee \neg b)_1 \wedge (a \vee \neg e)_2 \wedge (e \vee \neg d)_3 \wedge (\neg b \vee c \vee \neg d)_4 \ .$$

*The formula $F$ is not simplified by either subsumption or self-subsumption. Assigning any literal of $c_4$ to false causes another literal of the same clause to be implied to true; for example, $\neg d$ is implied by $b$ in the implication graph of Fig. 3.2. Hence, $c_4$ can be removed from $F$ without affecting its deductive power or the size of the implication graphs. On the other hand, since $(\neg b \vee \neg d)$ is a transitive closure clause of $F$ that subsumes $(\neg b \vee c \vee \neg d)$, its addition may shorten an implication graph, e.g., the dashed edge in Fig. 3.2, even though it does not improve deductive power.* □

Even though adding an implicate to a CNF formula may not affect its deductive power, the situation is different when a conflict clause containing a UIP is learned by a DPLL-based solver. After recalling a known result (Lemma 3.8, [MS99]), we show that the addition of a conflict learned clause containing a UIP always increases the deductive power.

**Lemma 3.8.** *Let $F$ be a CNF formula and let $\gamma$ be a conflict clause containing a UIP. Then $\gamma$ is an implicate of $F$ not subsumed by any clause of $F$.*

*Proof.* Since $\gamma$ is obtained by resolution of clauses in $F$, it is implied by their conjunction, and hence by $F$. Since $\gamma$ evaluates to false at the last decision level, any clause that subsumes it should evaluate to false as well. However, all clauses that are false at the last decision level contain at least two literals assigned at the last decision level. (Otherwise they would have been asserting at some previous level.) On the other hand, $\gamma$ contains exactly one literal assigned at the last decision level, namely the UIP. Therefore, it cannot be subsumed by any conflicting clause. □

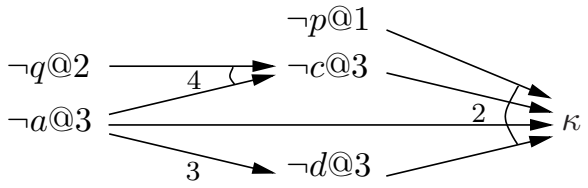



Figure 3.3: Implication graph of Example 3.10.

**Lemma 3.9.** *Let $F_1$ be a CNF formula and let $\gamma$ be a conflict clause containing a UIP. Let $F_2$ be the CNF formula obtained from $F_1$ by adding $\gamma$ and optionally removing clauses that are subsumed by $\gamma$. Then, $F_1 \prec F_2$.*

*Proof.* By Lemma 3.4, $F_1 \preceq F_2$. Let $\gamma = \{l_1, \ldots, l_n, u\}$ be the conflict clause and let $u$ be its UIP. Consider the assignment $A = \{\{\neg l_1\}, \ldots, \{\neg l_n\}\}$. We have $F_2 \cup A \vdash_{\mathcal{D}} u$, but $F_1 \cup A \nvdash_{\mathcal{D}} u$, for otherwise $u$ would have not had a higher decision level than the other literals. Hence, $F_2 \not\preceq F_1$. □

Lemma 3.9 does not apply to strong conflict analysis. The following example illustrates a clause that is not new may be generated.

**Example 3.10.** *Consider the following clauses:*

$$(a \vee \neg d)_1 \wedge (a \vee \neg c \vee p)_2 \wedge (a \vee c \vee d)_3 \wedge (a \vee d \vee p)_4 \ .$$

*Suppose that the SAT solver makes decisions $\neg p@1$ and $\neg a@2$, and, at level 2, examines $c_1$, $c_2$, and $c_3$ in order. It then identifies $c_3$ as a conflicting clause. As shown in Fig. 3.3, clause $c_4$ does not appear in the implication graph, even though it is also conflicting under the current assignment. The resolution of $c_2$ and $c_3$ on $c$ produces $(a \vee d \vee p)$, which is $c_4$. Since this clause contains two literals assigned at level 2, it may be chosen by strong conflict analysis.* □

Adding duplicate clauses clearly does not improve deductive power. Even when a clause added by strong conflict analysis is new, it may not improve it. However, it may still contribute to generating more compact implication graphs.

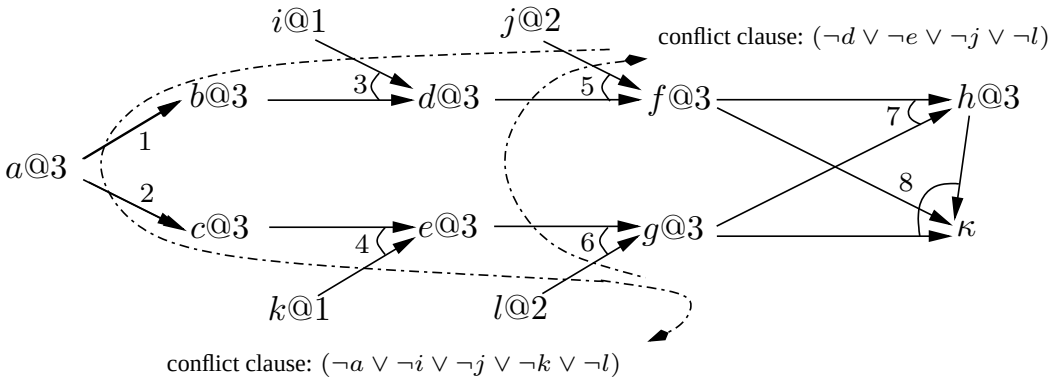



Figure 3.4: Implication graph for the first conflict of Example 3.11.

**Example 3.11.** *Consider the implication graph of Fig. 3.4. For that conflict, the SAT solver computes a 1-UIP based clause* $(\neg a \vee \neg i \vee \neg j \vee \neg k \vee \neg l)$*, which becomes asserting at level 2. Now suppose that* $m$ *is assigned to* true *by decision making and that the deduction procedure creates the implication graph shown in Fig. 3.5. This graph is similar to the one of Fig. 3.4; in particular, the same clause is conflicting. However, if strong conflict analysis adds* $(\neg d \vee \neg e \vee \neg j \vee \neg l)$*, then, under the same decisions, the additional clause will cause a conflict after fewer implications as shown in Fig. 3.6. A simpler implication graph is analyzed more quickly. Moreover, the additional conflict clause may increase deductive power. For instance, if later in the search* $d$*,* $e$*, and* $j$ *are the only assigned literals, the additional conflict clause is asserting, while the 1-UIP based conflict clause is not.* □

## 3.2 On-The-Fly Self-Subsumption

Lemma 3.4 implies that simplification based on self-subsumption may improve the deductive power of a CNF formula. Since detecting whether the resolvent of two clauses subsumes either operand is easy and inexpensive, 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 the application of the on-the-fly subsumption check to conflict analysis. Then, we discuss on-the-fly subsumption in preprocessing.

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

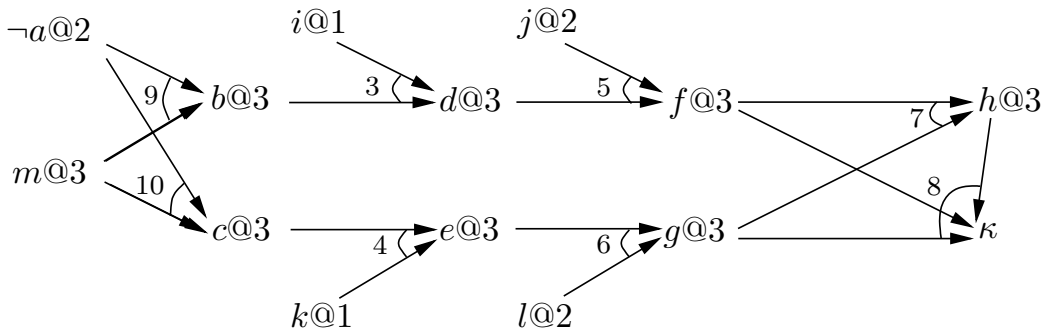



Figure 3.5: Implication graph without additional conflict clause.

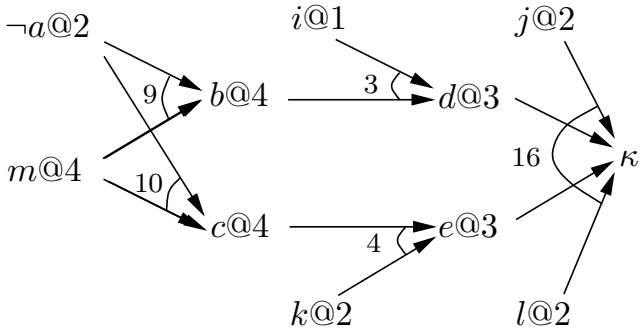



Figure 3.6: Implication graph with additional conflict clause.

fact.

**Lemma 3.12.** *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 $c_1$ ($c_2$) if and only if $|c'_1 \cup c'_2| = |c_1| - 1$ ($|c'_1 \cup c'_2| = |c_2| - 1$).*

*Proof.* Subsumption of $c_1$ occurs if and only if $c'_1 \cup c'_2 = c'_1$, which is equivalent to $|c'_1 \cup c'_2| = |c'_1| = |c_1| - 1$. Likewise for subsumption of $c_2$. □

Thanks to Lemma 3.12, existing clauses that are subsumed by resolvents can be detected and replaced by 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 the first position in the clause [ES03, Bie08b], it is easily accessed. In variable elimination, the literal to be removed corresponds to the variable to be eliminated. Therefore, it is enough to save its position in the clause being scanned. 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 3.13.** *Consider the following set of clauses:*

$$(a \vee b \vee \neg c)_1 \wedge (a \vee b \vee \neg d)_2 \wedge (c \vee d \vee \neg e)_3 \wedge (c \vee e \vee f)_4$$
$$\wedge (d \vee e \vee \neg f)_5 \wedge (\neg b \vee \neg d \vee e)_6 \wedge (\neg d \vee \neg e)_7 \ .$$

*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 $f$ through $c_4$, at which point $c_5$ is conflicting. Analysis of this conflict proceeds on the implication graph shown in Fig. 3.7. Conflict analysis goes back through the implication graph building the resolution graph shown in Fig. 3.8. The resolution graph shows that $\gamma_2$ subsumes $c_3$, and that $\gamma_4$ subsumes $c_1$. The subsumed clauses can be strengthened by eliminating the pivot variable on which they were resolved. In addition to the simplifications, $\gamma_4$, containing the 1-UIP, subsumes $c_1$; it is not required to add it to the clause database. Rather, $c_1$ is strengthened to $a \vee b$. The simplified CNF is therefore*

$$(a \vee b)_1 \wedge (a \vee b \vee \neg d)_2 \wedge (c \vee d)_3 \wedge (c \vee e \vee f)_4 \wedge$$
$$(d \vee e \vee \neg f)_5 \wedge (\neg b \vee \neg d \vee e)_6 \wedge (\neg d \vee \neg e)_7 \ .$$

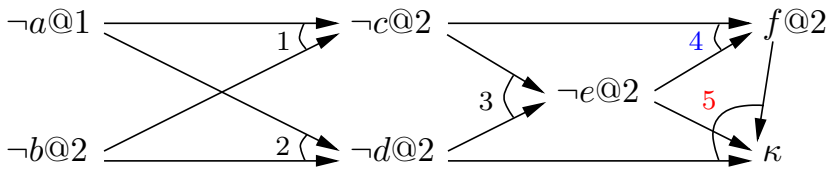



Figure 3.7: Implication graph of Example 3.13.

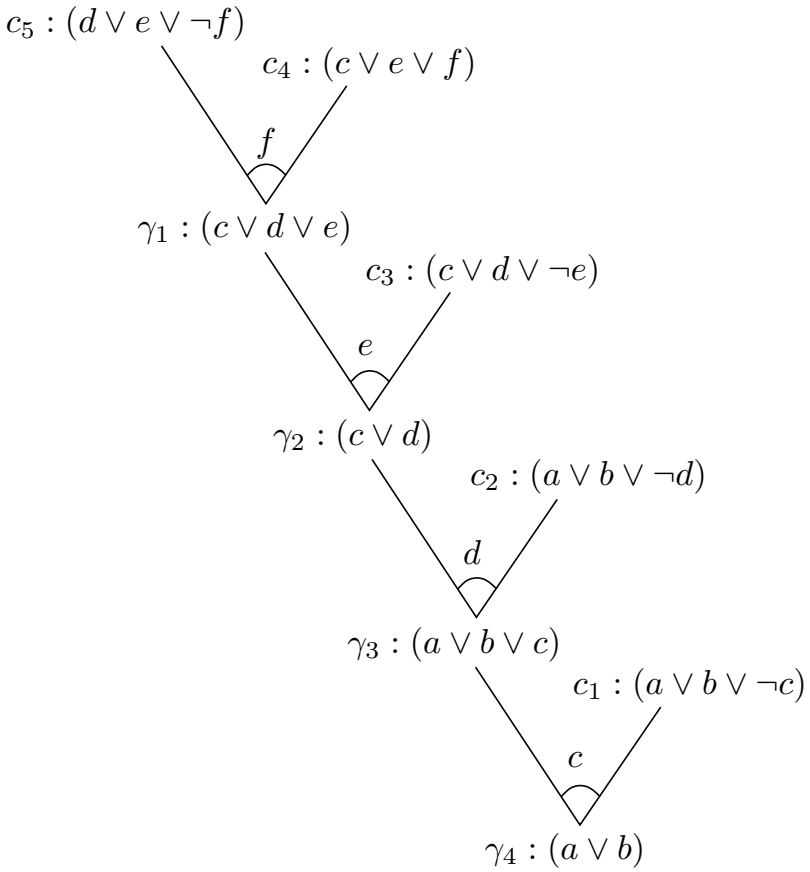



Figure 3.8: Resolution tree of conflict analysis for Fig. 3.7

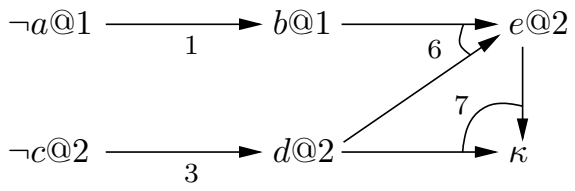



Figure 3.9: Implication graph for the second conflict of Example 3.13.

*After conflict analysis, the solver backtracks to level 1, which is the highest decision level in the strengthened* $c_1$ *when the UIP* $b$ *is ignored. After backtracking,* $b@1$ *is asserted by* $c_1$.

*Suppose that* $\neg c$ *is decided at level 2. From this decision, literal* $d$ *is implied. This partial assignment, in turn, implies* $\neg e$ *through* $c_7$. *The chain of implications leads to another conflict at* $c_6$, *as shown in Fig. 3.9. Conflict analysis, illustrated in Fig. 3.10 yields the conflict clause* $\gamma_5$. *Then, since* $c_6$ *is subsumed by* $\gamma_5$, $c_6$ *is strengthened. Hence the CNF formula is simplified as follows:*

$$(a \vee b)_1 \wedge (a \vee b \vee \neg d)_2 \wedge (c \vee d)_3 \wedge (c \vee e \vee f)_4 \wedge$$
$$(d \vee e \vee \neg f)_5 \wedge (\neg b \vee \neg d)_6 \wedge (\neg d \vee \neg e)_7 \ .$$

□

Example 3.13 shows how the CNF database can be simplified by checking subsumption on-the-fly. A clause can be shortened when it is resolved during conflict analysis if it is subsumed by the resolvent. The resolvent may contain a UIP; then, the clause that is strengthened can serve as conflict-learned clause.

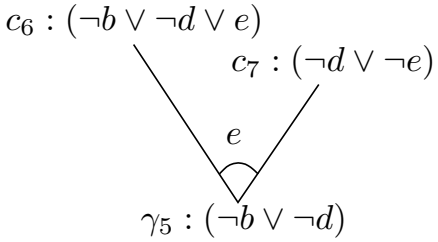



Figure 3.10: Resolution tree of conflict analysis for Fig. 3.9.

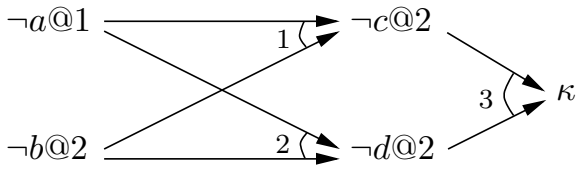



Figure 3.11: Implication graph shrunk from Fig. 3.7 with a new conflict node.

When this happens, an increase in deductive power is achieved even without adding a conflict clause.

Conflict analysis based on 1-UIP may be followed by strong conflict analysis. Therefore, we consider the on-the-fly subsumption check in the context of strong conflict analysis.

**Lemma 3.14.** *If clause $\gamma$ has been simplified by self-subsumption during conflict analysis, it is conflicting at the current level.*

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

In Example 3.13, once $c_3$ is strengthened by $\gamma_2$, it becomes conflicting. Then, the implication graph of Fig. 3.7 is shrunk as shown in Fig. 3.11 by establishing $c_3$ as the antecedent of a new $\kappa$ node.

**Lemma 3.15.** *Let $\gamma$ 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 3.14, $\gamma$ is conflicting at the current decision level. If $\gamma$ contains the 1-UIP, the subgraph consists of a single vertex and strong conflict analysis is not invoked. Otherwise, since the residual clauses beyond $\gamma$ on the graph were not touched, they form a valid graph for strong conflict analysis. □

Lemmas 3.14 and 3.15 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 2.6, in Fig. 2.3, $\gamma_1$, $\gamma_2$, and $\gamma_3$ all have a chance to be chosen as additional conflict clauses by strong conflict analysis, since they have only two literals, both of which are assigned at the current level. Strong conflict analysis dismisses $\gamma_1$ as too close to the conflicting clause. However, on-the-fly subsumption achieves the same effect on deductive power that the addition of $\gamma_1$ would have by strengthening $c_6$ and dropping $c_7$. On-the-fly subsumption therefore complements strong conflict analysis. If no subsumption is found during conflict analysis, additional clauses generated by strong conflict analysis may still be helpful, even though they are not guaranteed to increase deductive power.

One may be tempted to apply on-the-fly subsumption to conflict clause minimization [SE05, SB09]. However, the antecedent clauses involved in the minimization are never subsumed by their resolvents since they do not contain any literal assigned at the current level, while the resolvents contain the UIP.

Figure 3.12 shows the pseudocode of the algorithm that detects and simplifies the subsumed clauses during conflict analysis. The algorithm AnalyzeConflictWithSimplification() checks the subsumption condition whenever RESOLVE() produces a new resolvent as long as FOUNDUIP() is false (line 4). By Lemma 3.12, if one of the operands exists in the clause database—either the old resolvent with in_CNF_resolvent = TRUE or the antecedent of the pivot variable (line 9)—and the new resolvent contains fewer literals than one of its operands (lines 10 and 12), the operand is strengthened by removing the pivot variable (line 11). When both operands are subsumed, one of them survives and the other is deleted (line 13). If a clause is replaced with the resolvent, the flag in_CNF_resolvent is set to TRUE (line 14); otherwise, it is set to FALSE (line 16), since the new resolvent is not yet in the clause database. At the end of the resolution step, if the final resolvent containing the UIP strengthens an existing clause, that is, if in_CNF_resolvent is true, the conflict analysis algorithm refrains from adding a new conflict clause to the clause database. Otherwise the clause is added at line 20. Whether a conflict clause is added or not, the DPLL procedure backtracks to the level returned by conflict analysis (line 21), and asserts the clause finally learned from the conflict.

The pseudocode of Fig. 3.12 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.

```
1   AnalyzeConflictWithSimplification(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           oprnd = in_CNF_resolvent ? resolvent : ante;
10          if (SIZE(resolvent′) < SIZE(oprnd)) {
11              STRENGTHENCLAUSE(oprnd, var);
12              if (in_CNF_resolvent & textscSize(resolvent′) < SIZE(ante))
13                  DELETECLAUSE(ante);
14              in_CNF_resolvent = TRUE;
15          }
16          else in_CNF_resolvent = FALSE;
17          resolvent = resolvent′ ;
18      }
19      if (!in_CNF_resolvent)
20          ADDCONFLICTCLAUSE(resolvent);
21      blevel = COMPUTEHIGHESTLEVEL(resolvent);
22      return (blevel);
23  }
```

Figure 3.12: Algorithm for conflict analysis with on-the-fly simplification.

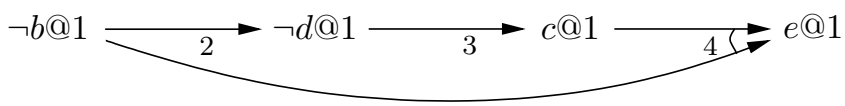



Figure 3.13: The implication graph of Example 3.17.

## 3.3 Clause Distillation

In this section, we present an extension of simplification based on self-subsumption, which is called **clause distillation**. Given a CNF formula, clause distillation removes clauses subsumed by implicates that may not be explicitly found in the formula, and optionally adds new conflict clauses. Like simplification based on self-subsumption, the distillation procedure often increases deductive power.

**Lemma 3.16.** *If $A = \{\{\neg l_1\}, \ldots, \{\neg l_{n-1}\}\}$ is a partial assignment to the variables of CNF formula $F$ and $F \cup A \vdash l_n$, then $\{l_1, \ldots, l_n\}$ is an implicate of $F$.*

*Proof.* Suppose that a satisfying assignment for $F$ included $\{\{\neg l_1\}, \ldots, \{\neg l_n\}\}$. Such an assignment would contradict $F \cup A \vdash l_n$. Therefore, any complete assignment that satisfies $F$ must contain some literal $l_i$, $1 \leq i \leq n$. Hence, such assignment satisfies clause $\{l_1, \ldots, l_n\}$. □

The next example shows how Lemmas 3.16 and 3.4 combine to improve a CNF formula that cannot be simplified by either subsumption or self subsumption.

**Example 3.17.** *Consider the following CNF formula:*

$$F = (a \vee b \vee c)_1 \wedge (b \vee \neg d)_2 \wedge (c \vee d)_3 \wedge (b \vee \neg c \vee e)_4 \ .$$

*Under partial assignment $\{\{\neg b\}\}$, $\neg d$ is implied by $c_2$ of $F$ and $c$ is implied by $c_3$. The clause $c_1$ is then satisfied. Finally, $e$ is implied by $c_4$. The implication hypergraph depicted in Fig. 3.13 shows that $(b \vee c)$ is an implicate of $F$ that subsumes the first clause, and that $(b \vee e)$ is another implicate of $F$ that subsumes $c_4$. The simplified CNF is therefore*

$$F' = (b \vee c)_1 \wedge (b \vee \neg d)_2 \wedge (c \vee d)_3 \wedge (b \vee e)_4.$$

*Note that $F' \succ F$ because $F' \cup \{\neg e\} \vdash_{\mathcal{D}} b$, but $F \cup \{\neg e\} \nvdash_{\mathcal{D}} b$. If, from $F'$, $F''$ is obtained by deleting $c_1$, which is the transitive closure clause of $c_2$ and $c_3$, then $F''$ is equivalent to $F'$ and $F' \simeq F''$. On the other hand, $F''$ is not as effective as $F'$ in shortening the implication graph.* □

Example 3.17, Lemmas 3.16 and 3.9 suggest a systematic approach to improving the deductive power of a CNF formula $F$. Suppose $F$ contains no unit clauses. (If it does, simplify $F$ in the obvious way.) Let $\gamma = \{l_1, \ldots, l_n\}$ be a clause of $F$. Consider the sequence of assignments $A_i = \{\{\neg l_1\}, \ldots, \{\neg l_i\}\}$ for $1 \leq i < n$ (the **assignment sequence** of $\gamma$). There exists a least $i$ such that either $F \cup A_i \vdash_{\mathcal{D}} \mathsf{false}$, or $F \cup A_i \vdash_{\mathcal{D}} l_j$, for $i < j \leq n$. In either case, we extract from the implication graph an implicate of $F$ that subsumes $\gamma$. This implicate may be $\gamma$ itself, another clause of $F$ that subsumes $\gamma$, or the resolvent of several clauses of $F$.

If $F \cup A_i \vdash_{\mathcal{D}} \mathsf{false}$, the learned conflict clause is added to $F$ to increase its deductive power (thanks to Lemma 3.9). The conflict analysis used in distillation differs from that of Sec. 2.6 in that it stops when it computes a resolvent that is the negation of a subset of $A_i$. Such a resolvent always exists, contains a UIP, and subsumes $\gamma$. If intermediate resolvents contain a UIP, but are not in subsumption relation with $\gamma$, then conflict analysis produces two clauses: the 1-UIP clause and the one that subsumes $\gamma$.

**Example 3.18.** *Given a CNF formula $F$:*

$$\begin{aligned} F \;\; = \;\; & (a \vee b \vee c \vee d)_1 \wedge (a \vee e)_2 \wedge (b \vee f)_3 \wedge \\ & (\neg e \vee \neg f \vee g)_4 \wedge (\neg e \vee \neg f \vee \neg h)_5 \wedge (\neg g \vee h)_6 \;\; , \end{aligned}$$

*suppose $\gamma$ is $c_1$, and $A_2$ is $\{\neg a@1, \neg b@2\}$; then $c_6$ is conflicting as shown in Fig. 3.14. Analysis of this conflict leads to the resolution graph shown in Fig. 3.15. Since $\gamma_2$, containing the 1-UIP, does not subsume $\gamma$, the analysis returns two clauses: $\gamma_2$ and $\gamma_4$. Adding either $\gamma_2$ or $\gamma_4$ increases the deductive power of $F$ thanks to Lemma 3.9. Adding both $\gamma_2$ and $\gamma_4$ does not, however, further increase deductive power in this case, because $\gamma_4$ is the transitive closure clause of $c_2$, $\gamma_2$, and $c_3$.* □

If $F \cup A_i \vdash_{\mathcal{D}} l_j$, we use Lemma 3.16 to extract an implicate of $F$ that subsumes $\gamma$ from the implication hypergraph. If $\gamma$ is subsumed by another clause $\gamma'$ in $F$, the implication hypergraph contains a hyperedge

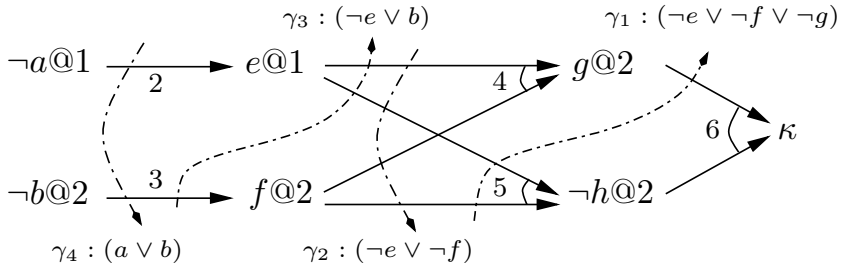



Figure 3.14: Implication graph of Example 3.18.

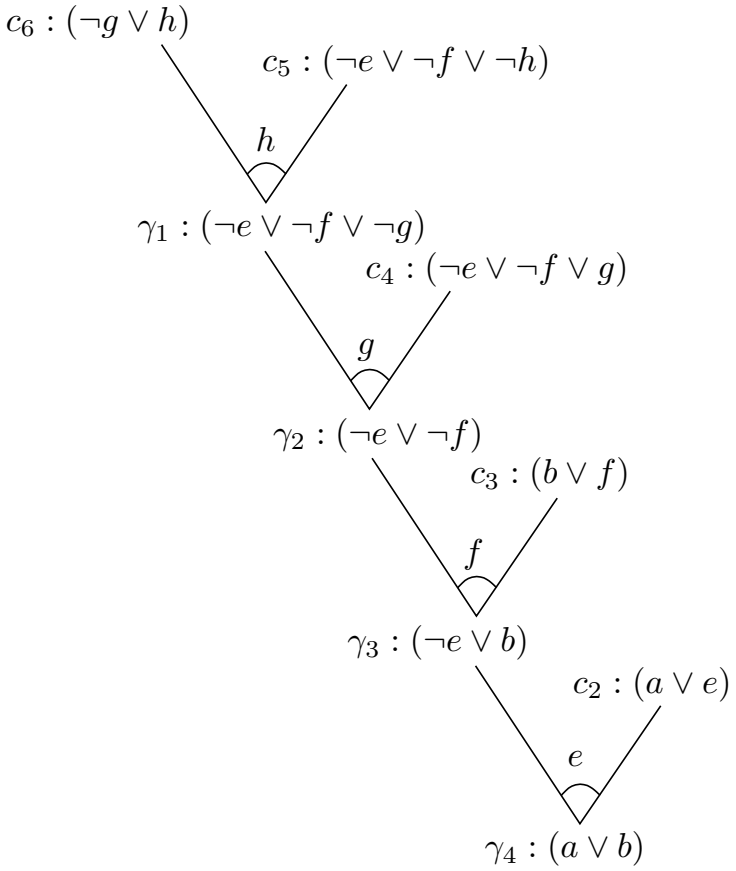



Figure 3.15: Resolution graph of conflict analysis for Example 3.18.

that asserts $l_j$ and such that all the directed edges originate from literals in $A_i$. Therefore, even if finding a minimal implicate of $F$ that subsumes $\gamma$ is hard, removing from $F$ a clause that is subsumed by another clause only requires inspecting the hypergraph induced by $A_i$. However, if an implication graph is used instead of the full hypergraph, the computed implicate may be subsumed by another clause. This problem is solved by integrating subsumption-based simplification with the distillation procedure.

**Example 3.19.** *Given clauses* $(a \vee b \vee c)_1$ *and* $(b \vee c)_2$ *of* $F$*, suppose* $\gamma = c_1$ *is distilled with* $A_2 = \{\{\neg a\}, \{\neg b\}\}$*. If* $c$ *is implied by* $c_1$ *rather than* $c_2$*, then the implicate* $(a \vee b \vee c)$ *is computed, which does not strictly subsume* $\gamma$*.* □

If $\gamma$ is strictly subsumed by the implicate obtained through Lemma 3.16, then replacing $\gamma$ with the implicate in $F$ may or may not increase deductive power, as shown in Example 3.17, in which adding $(b \vee e)$ to $F$ improves its deductive power, while adding $(b \vee c)$ does not.

The distillation procedure outlined above may be used to detect some cases of self subsumption. For instance, if $\gamma = \{l_1, l_2, l_3, l_4\}$ and $F = \{\gamma, \gamma'\}$, with $\gamma' = \{l_1, \neg l_2\}$, then $F \cup A_1 \vdash_{\mathcal{D}} \neg l_2$. From that, it is concluded that $\{l_1, l_3, l_4\}$ is the desired implicate of $F$ that subsumes $\gamma$. Another example is given by $\gamma = \{l_1, l_2\}$ and $\gamma' = \{l_1, \neg l_2\}$. Asserting $\neg l_1$ leads to a conflict, and the learned clause, $\{l_1\}$, subsumes $\gamma$ (and $\gamma'$). Self-subsumption, however, may go undetected. Consider $F = \{\gamma, \gamma'\}$, with $\gamma = \{l_1, l_2, l_3, l_4\}$ and $\gamma' = \{\neg l_1, l_2\}$. In this case, $A_1$ will cause no implications, and the simplified clause $\{l_2, l_3, l_4\}$ will not be discovered. Such limited ability should not surprise. In general, for a clause $\gamma$ of $F$ with $n$ literals, $n$ attempts are sufficient to find an implicate of $F$ that subsumes $\gamma$ and cannot be further simplified by self subsumption. This is comparable to what the procedure of [EB05] does.

More simplifications can be achieved if the on-the-fly simplification discussed in Sect. 3.2 is applied to conflict analysis in the distillation procedure. For instance, in Fig. 3.14 of Example 3.18, $c_4$ is simplified by on-the-fly subsumption check because it is subsumed by $\gamma_2$. In addition, as we shall see in the detailed discussion of the algorithm, on-the-fly simplification improves the efficiency of the distillation procedure.

A preprocessing algorithm can be based on distilling each clause $\gamma$ of a CNF formula by trying its assignment sequence until either a conflict occurs or a literal $l_j$ of $\gamma$ is asserted. Clause $\gamma$ is replaced by

either a conflict clause or an implicate containing $l_j$. We have seen in Example 3.19 that after distillation a clause may still be subsumed by other clauses of the CNF formula. In addition, as distillation proceeds and shorter clauses are added to $F$, a clause that is initially not subsumed may lose this property. Therefore, subsumption-based simplification is applied after distillation.

Figure 3.16 describes an algorithm based on the distillation approach outlined in this section. The clauses are initially stored in a **trie** [AHU83] so that common prefixes may be identified. Each trie node has two sets of children corresponding to the two literals of each variable. The $0$-child is for the positive literal, and the $1$-child is for the negative literal. Every path in the trie represents a clause; a leaf node in a path stores the index of the clause associated with it. Hence, clauses that have a common prefix will share nodes in the trie. While building the trie, clauses that are subsumed by others may be detected and removed. However, both building a compact trie and detecting subsumed clauses depend on the variable order. The frequency of variables in the clause database usually provides a good order.

Since the trie supplements instead of replacing the clause database, it takes extra memory. Its use is justified by the speed-up that the sharing among clauses affords. However, if we distill the clauses based on the trie, instead of enumerating them one by one, we need to reach a leaf node to locate the corresponding clause in the database. Applying on-the-fly simplification to conflict analysis may help the distillation procedure dispense with that search. If $\gamma$ participates in conflict analysis, then it may be identified by on-the-fly subsumption check and replaced with the conflict clause that subsumes $\gamma$.

In Fig. 3.16, variable Trie is the set of roots of the trie that is built on the given formula $F$. Distillation consists of a depth-first traversal of the trie by TRIEBASEDIMPLICATION(). If the value of the node is assigned, i.e., value != UNKNOWN (line 4), ANALYZEIMPLICATE() analyzes the implication graph to find an implicate that strictly subsumes the clause being distilled (line 5). If the implicate exists, it replaces the clause being distilled (lines 7–8). At each node whose value is not yet asserted (lines 15 and 16), CHOOSENEXTASSIGNMENTONTRIE assigns values $0$ and $1$ to the children only if they have siblings (line 21). Procedure DEDUCE() propagates the decision over the clauses of $F$ (line 22). If a conflict occurs during DEDUCE(), ANALYZECONFLICTFORDISTILLATION() generates two clauses: cl1 is the one that subsumes the clause being distilled, and cl2 is the 1-UIP clause (line 24). The conflict clauses can be NULL: cl1

is NULL when a clause is simplified by the conflict clause and is marked during conflict analysis; cl2 is NULL when the conflict analysis produces only one conflict clause. If cl1 is not NULL, the clause being distilled is found by FINDDISTILLEDCLAUSEONTRIE() along the path from the current node (line 26), is simplified by the generated conflict clause cl1 (line 27), and marked (line 28). If cl2 is not NULL, it is added to $F$ as a conflict clause (line 31) and also marked (line 32). If a conflict does not occur during DEDUCE(), TRIEBASEDIMPLICATION() is invoked to test the next sibling at the current decision level (line 35). The procedure backtracks to the previous decision level when it has traversed all the children (line 36). Once the traversal on the trie is complete, each clause in $F$ is added to $F'$ only if it is marked. After distillation formula $F'$ can be further simplified by subsumption check.

**Example 3.20.** *Consider the following CNF formula:*

$$\begin{aligned} F &= (a \vee b \vee c)_1 \wedge (b \vee \neg d)_2 \wedge (c \vee d)_3 \wedge (b \vee c \vee e)_4 \wedge \\ &\quad (\neg b \vee c \vee f)_5 \wedge (\neg d \vee \neg f)_6 \ . \end{aligned}$$

*The trie shown in Fig. 3.17 is built with the CNF clauses of $F$ according to the variable order $b < c < d < f < a < e$. (The variables are sorted by their number of occurrences in $F$.) The procedure starts traversal from the first $0$-child of the $b$ root (line 15 of Fig. 3.16), that is, by considering $A_1 = \{\{\neg b\}\}$. The assignment $A_1$ is applied to all clauses that contain $b$. Propagation of $\neg b@1$ over $F$ leads to $\neg d@1$ and $c@1$ as shown in Fig. 3.18. The distillation procedure then reaches node $c$. Finding that $c$ is already implied, it computes the implicate $(b \vee c)$ by resolving $c_3$ and $c_2$ (line 5). Then, a depth-first search is performed on the $0$-children of $c$ to find one of the clauses that share the traversed path as a prefix (line 7). If $c_1$ is found, it is simplified to $(b \vee c)$ (line 8) and is marked (line 9) to be retained at the end of distillation; $c_4$ is not marked and can be deleted because it is subsumed by the simplified $c_1$. Since node $c$ does not have $1$-children, the procedure goes back to root $b$ and then on to $d$. The implication graph is not changed; hence, $d$ is still assigned. However, since the implicate $(b \vee \neg d)$ is the antecedent $c_2$ of $d$, $c_2$ is just marked with no simplification. Then, the procedure backtracks to level $0$, and it continues traversing the $1$-child of $b$ with $A_1 = \{\{b\}\}$ (line 16). Variable $c$ is assigned to* false *because it is not yet assigned. From $A_2 = \{\{b\}, \{\neg c\}\}$, $f$ and $d$ are implied through $c_5$ and $c_3$, respectively, and then $c_6$ becomes conflicting. Analysis of this conflict*

```
1   TRIEBASEDIMPLICATION(Trie) {
2       for each (node ∈ Trie) {
3           value = VALUE(node);
4           if (value != UNKNOWN) {
5               implicate = ANALYZEIMPLICATE()
6               if (implicate) {
7                   distilled = FINDDISTILLEDCLAUSEONTRIE(node);
8                   SIMPLIFYCLAUSE(distilled, implicate);
9                   GETCLAUSEMARKED(distilled);
10              }
11              if (node.child[value])
12                  TRIEBASEDIMPLICATION(node.child[value]);
13              continue ;
14          }
15          TRIEBASEDIMPLICATIONAUX(node, 0);
16          TRIEBASEDIMPLICATIONAUX(node, 1);
17      }
18  }

19  TRIEBASEDIMPLICATIONAUX(node, value) {
20      child = node.child[value];
21      if (child) {
22          level = CHOOSENEXTASSIGNMENTONTRIE(node, value);
23          if (DEDUCE() == CONFLICT) {
24              (cl1, cl2) = ANALYZECONFLICTFORDISTILLATION();
25              if (cl1 != NULL) {
26                  distilled = FINDDISTILLEDCLAUSEONTRIE(child);
27                  SIMPLIFYCLAUSE(distilled, cl1);
28                  GETCLAUSEMARKED(distilled);
29              }
30              if (cl2 != NULL)
31                  ADDCONFLICTCLAUSE(cl2);
32                  GETCLAUSEMARKED(cl2);
33          }
34          else
35              TRIEBASEDIMPLICATION(child);
36          BACKTRACK(level-1);
37      }
38  }
```

Figure 3.16: Algorithm for clause distillation.

*(line 24) proceeds on the implication graph shown in Fig. 3.20, building the resolution graph of Fig. 3.20. The resolution graph shows that $\gamma_2$ containing the 1-UIP subsumes $c_5$, which appears at the leaf node of the current path. Therefore, $c_5$ is strengthened by $\gamma_2$ and marked; this eliminates the work required to get the clause index stored in a leaf node (i.e., cl1 and cl2 are NULL in line 24). Moreover, this increases the deductive power of $F$. The procedure backtracks to level $0$, and it moves to the second root, $c$, of the trie. The $0$-child of $c$ is traversed with $A_1 = \{\{\neg c\}\}$. This assignment causes a conflict at $c_2$, as shown in Fig. 3.21 (left): the antecedent of $\neg b$, $c_5$, was simplified by on-the-fly subsumption in the previous conflict analysis. Conflict analysis, illustrated in Fig. 3.21 (right), produces $\gamma_2$ and simplifies $c_3$ because it is subsumed by $\gamma_2$. Since $c_3 = (c)$ is a unit clause, it must be propagated at level $0$ after backtracking; if a conflict occurs during this propagation, then the formula $F$ is declared unsatisfiable. After level $0$ propagation, the root $c$ is revisited only if it has $1$-children to be traversed. Finally, the procedure traverses the $1$-child of the last root $d$ with $A_1 = \{\{d\}\}$, under which $\neg f$ is asserted by $c_6$. Then, $c_6$ is marked because it is the implicate itself. Once the trie has been traversed, the marked clauses of $F$ are forwarded to $F'$. Therefore,*

$$F' = (b \vee c)_1 \wedge (b \vee \neg d)_2 \wedge (c)_3 \wedge (\neg b \vee c)_5 \wedge (\neg d \vee \neg f)_6 \ .$$

*The formula $F'$ is processed by subsumption-based simplification. In this case, $c_1$ are $c_5$ deleted from $F'$ because they are subsumed by $c_3$. At last, the formula is simplified to*

$$F' = (b \vee \neg d)_2 \wedge (c)_3 \wedge (\neg d \vee \neg f)_6 \ .$$

*Addition of the unit clause $(c)$ to $F$ definitely increases its deductive power, that is, $F \prec F'$.* □

## 3.4 Variable Elimination

In this section, we review the preprocessor for variable elimination that can be integrated with the distillation procedure of Sec. 3.3.

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 $\text{clauses}_v$ stands for an occurrence list of variable $v$, and $|\text{clauses}|$ represents the length of the list. $\delta$ stresses the fact that the less symmetric occurrence lists are, the

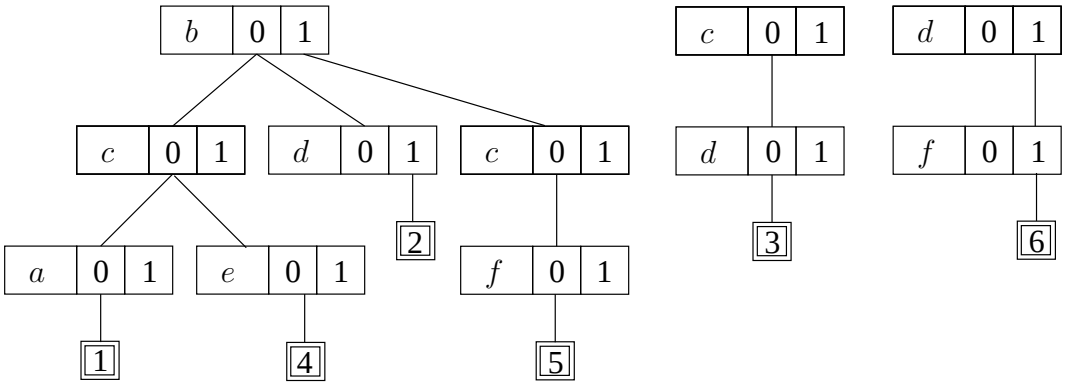



Figure 3.17: A trie of the CNF clauses of $F$.

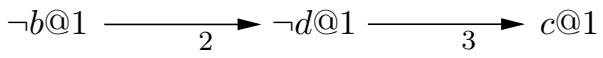



Figure 3.18: The implication graph generated under $A_1 = \{\{\neg b\}\}$.

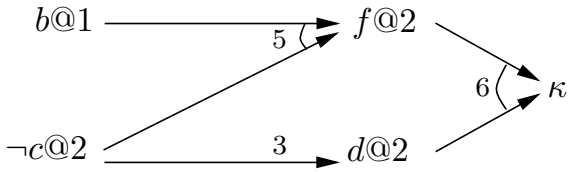



Figure 3.19: The implication graph generated under $A_2 = \{\{b\}, \{\neg c\}\}$.

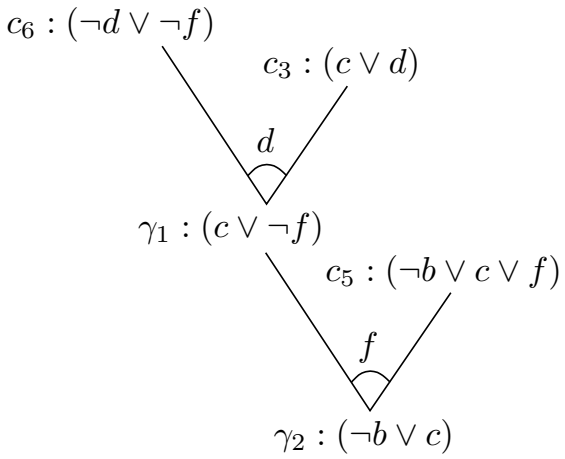



Figure 3.20: The resolution graph of the conflict analysis on Fig. 3.20.

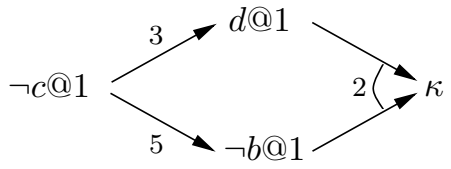



Figure 3.21: The implication graph under $A_1 = \{\{\neg c\}\}$.

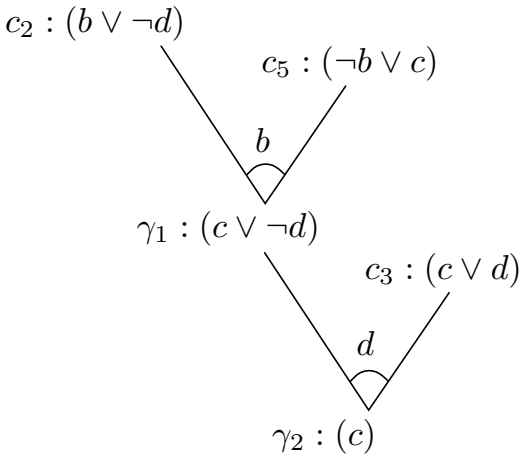



Figure 3.22: The resolution graph of conflict analysis on Fig. 3.21.

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 [CLR90]. 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. 3.2. 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.

## 3.5 Experimental Results

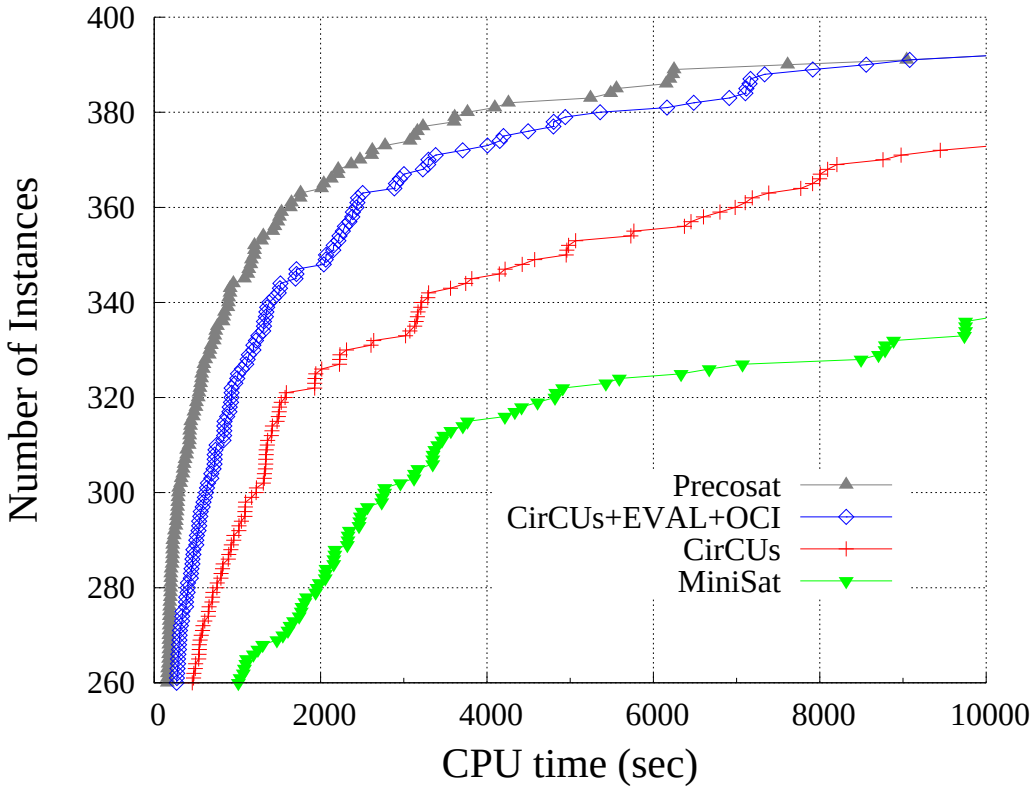
We have presented techniques that aim at increasing the deductive power of a CNF formula and promoting more concise implication graphs. In order to evaluate them, we have implemented a preprocessor on top of the CNF SAT solver **CirCUs** 2.0 [HJKS09, VIS], which applies variable elimination, the distillation procedure of Sect. 3.3, named **Alembic**, and simplification based on subsumption and self-subsumption as in [EB05]. We have also implemented the three applications of on-the-fly clause simplification discussed in this paper, namely, to variable elimination and conflict analysis in Alembic as well as to conflict analysis in CirCUs. In variable elimination, an increase in the average length of the clauses is detrimental for deductive power. Hence, in our implementation, only variables whose elimination does not cause such an increase are eliminated.

Since SAT solvers often need to provide either a satisfying assignment or a **proof of unsatisfiability**, clauses that are either removed or simplified are set aside just as the derivations of conflict clauses [GN03, ZM03]. The SAT solver CirCUs only needs these clauses to recover a complete solution (for a satisfiable instance), or to produce a proof of unsatisfiability in terms of the original clauses. This scheme requires extra memory, but its effect on speed is negligible.

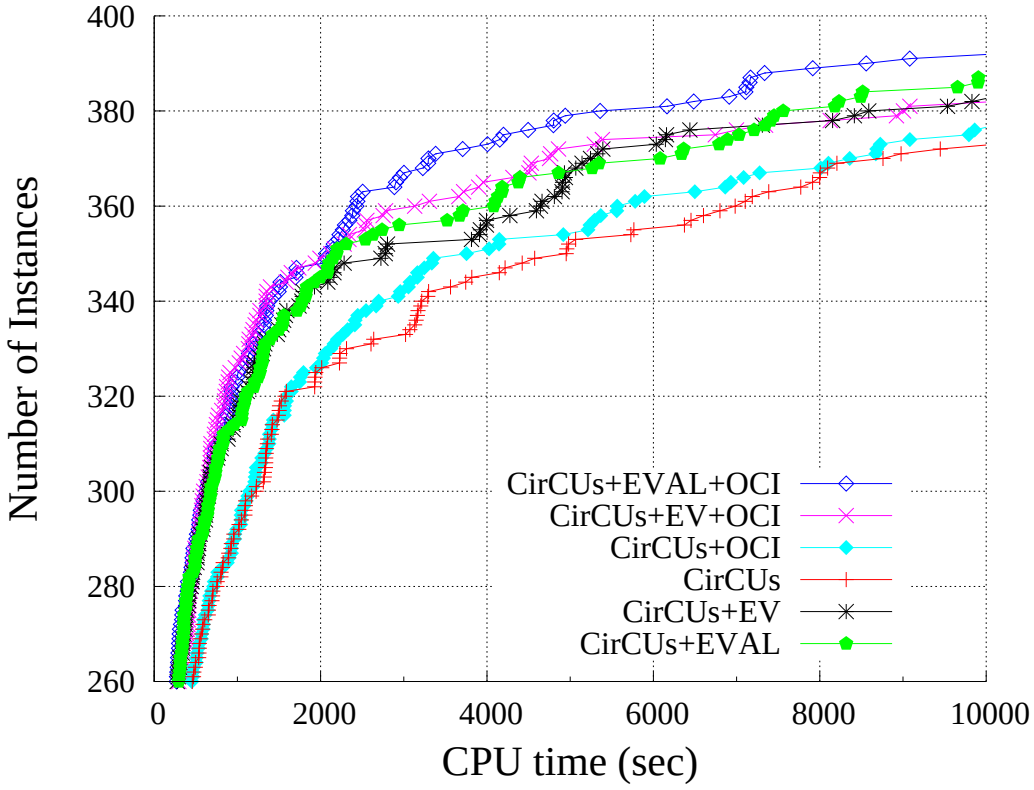
The benchmark suite is composed of all the CNF instances (with no duplicates) from the industrial category of the SAT Races of 2006 and 2008, and the SAT Competitions of 2007 and 2009 [SATa]. We conducted the experiments on a 2.4 GHz Intel Core2 Quad processor with 4GB of memory. We used 10000 seconds as timeout, and 2GB as memory bound. We tested MiniSat 2.0 [Satb] and PrecoSAT 236 [Pre] along with CirCUs 2.0 to provide reference points.

The plot of Fig. 3.23 shows how many instances are solved by selected solvers within a given time bound. Our variable elimination algorithm is named EV; Alembic is abbreviated AL, EVAL stands for EV+AL, and OCI denotes the on-the-fly clause improvement described in Sect. 3.2. Figure 3.23 shows the CPU time taken by CirCUs (with various subsets of the proposed approaches), MiniSat, and PrecoSAT. Both MiniSat and PrecoSAT use their own preprocessors [EB05]. Figure 3.23 confirms that CirCUs is comparable to state-of-the-art SAT solvers, and that its performance is significantly improved by applying all the proposed approaches (i.e., EVAL+OCI). Among the instances of Fig. 3.23, Figure 3.5 shows that unsatisfiable instances fare a bit better, but not much, than satisfiable ones in terms of performance improvements.

The scatterplots of Fig. 3.25, 3.26, and 3.27 examine the effects of the proposed techniques on deductive power and size of implication graphs, by showing the changes in CPU time, numbers of decisions, average numbers of resolution steps per conflict analysis, and average length of conflict-learned clauses. For each of these quantities the geometric mean of the new/old ratios is reported (excluding cases in which one of the values is $0$). Single-sample $t$-tests were performed to confirm the statistical significance of the data. The null hypothesis was that the mean of the logarithms of the ratios is $0$. The alternative hypothesis is two-sided. Since the data that are compared span several orders of magnitudes, differences and ratios may paint very different pictures of the experiments. Analyzing the ratios puts equal emphasis on short and long-

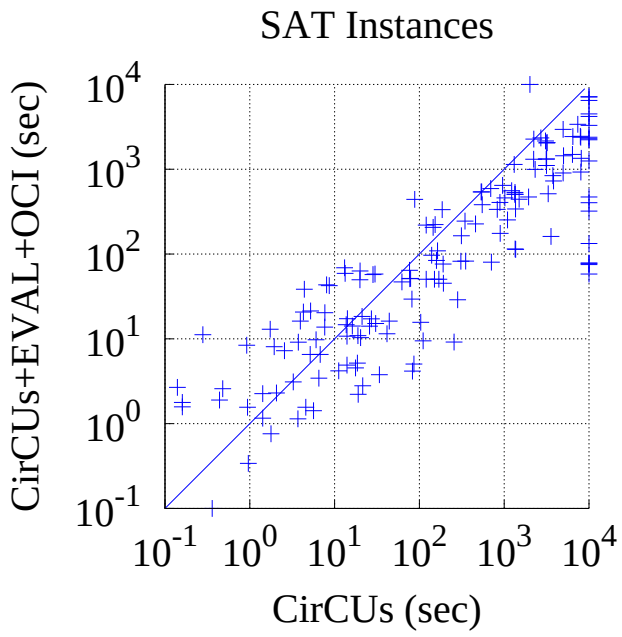



(a)

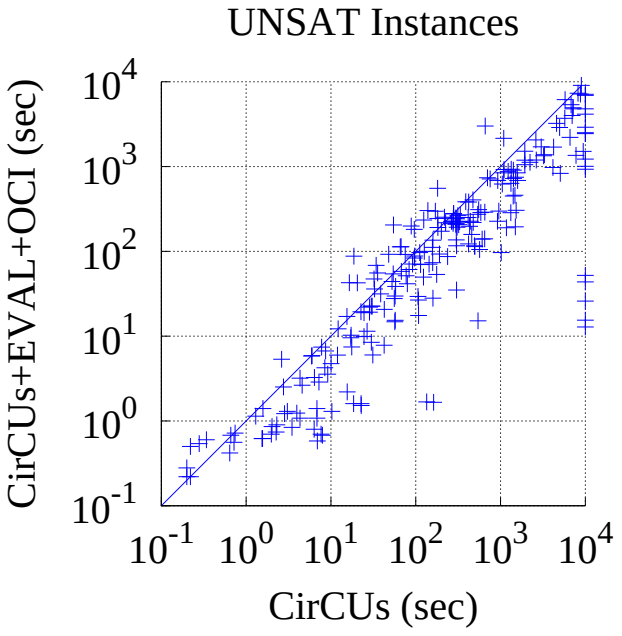



(b)

Figure 3.23: 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




(a)




(b)

Figure 3.24: Comparison of the performance improvements between (a)SAT and (b)UNSAT instances of Fig. 3.23.

running instances. This is partly compensated by the scatterplots and the views in Fig. 3.23, which highlight the ability of the improved procedure to complete more instances in the allotted time. Specifically, one can find the relative size of the resolution graphs for conflict analysis from the data of Fig. 3.25(c), as shown in Fig. 3.28.

A marked decrease in the numbers of decisions confirms that the proposed techniques allow the SAT solver to rely more on deduction and less on search. The reduction in resolution steps confirms that the implication graphs are, on average, significantly smaller. As a result, shorter clauses are learned. For lack of space, we omit scatterplots illustrating the effects of individual techniques. They would show that variable elimination is the main cause for the smaller implication graphs, and that it also tends to reduce the number of decisions and shorten the learned clauses. Distillation alone decreases the numbers of decisions (as one would expect of a technique addressing deductive power) and shortens learned clauses, but has limited effect on the sizes of the implication graphs. Its effect on memory consumption proves negligible. This is shown in Fig. 3.29.

Variable elimination interacts in an interesting way with OCI. This is shown in Fig. 3.30, where the numbers of on-the-fly subsumptions per resolution step during DPLL are seen to increase significantly when EV is applied. The following example sheds light on this phenomenon.

**Example 3.21.** *Consider the following clauses:*

$$(\neg a \vee \neg p)_1 \wedge (b \vee \neg p)_2 \wedge (a \vee \neg b \vee p)_3 \wedge (a \vee \neg q)_4 \wedge$$

$$(\neg b \vee \neg q)_5 \wedge (\neg a \vee b \vee q)_6 \wedge (\neg p \vee r)_7 \wedge (\neg q \vee r)_8 \wedge$$

$$(p \vee q \vee \neg r)_9 \wedge (a \vee \neg s)_{10} \wedge (b \vee \neg s)_{11} \wedge (\neg a \vee \neg b \vee s)_{12}$$

$$\wedge (\neg a \vee \neg t)_{13} \wedge (\neg b \vee \neg t)_{14} \wedge (a \vee b \vee t)_{15} \wedge (\neg s \vee \neg u)_{16} \wedge$$

$$(\neg t \vee \neg u)_{17} \wedge (s \vee t \vee u)_{18} \wedge (r \vee u)_{19} \wedge (\neg r \vee \neg u)_{20} \ .$$

*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. 3.31. There are no instances of on-the-fly subsumption during conflict analysis, even though* $\gamma_5$ *that is derived by minimizing the conflict clause* $\gamma_4$ *subsumes* $c_{15}$*:* $\gamma_5$ *directly*

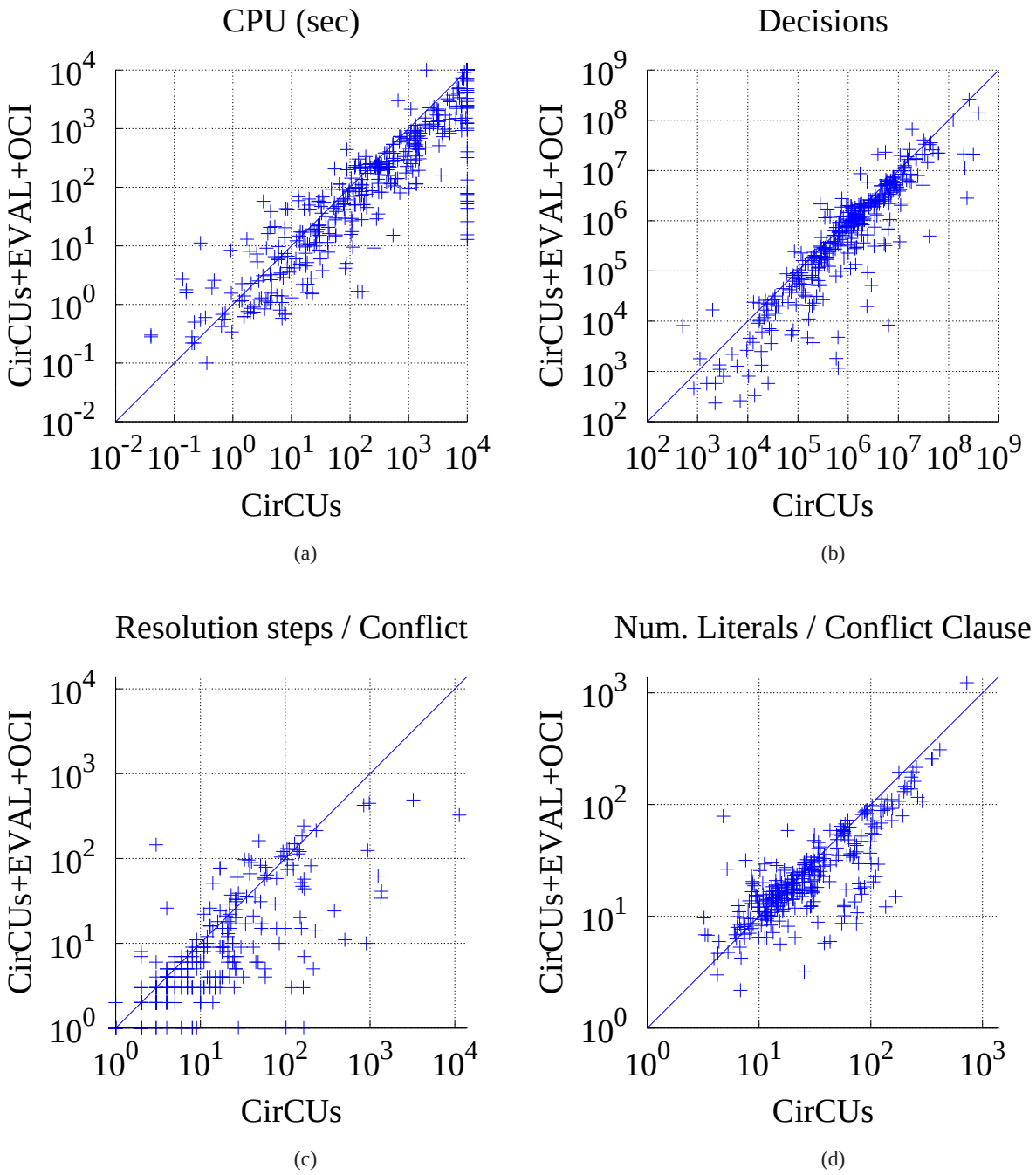



Figure 3.25: Effect of CirCUs with and without EVAL+OCI on (a) CPU time: GEOMETRIC MEAN = 0.56, $p$-value = $2.2 \cdot 10^{-16}$; (b) number of decisions: GEOMETRIC MEAN = 0.51, $p$-value = $2.2 \cdot 10^{-16}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = 0.57, $p$-value = $2.2 \cdot 10^{-16}$; (d) number of literals per conflict clause: GEOMETRIC MEAN = 0.82, $p$-value = $8.25 \cdot 10^{-8}$.

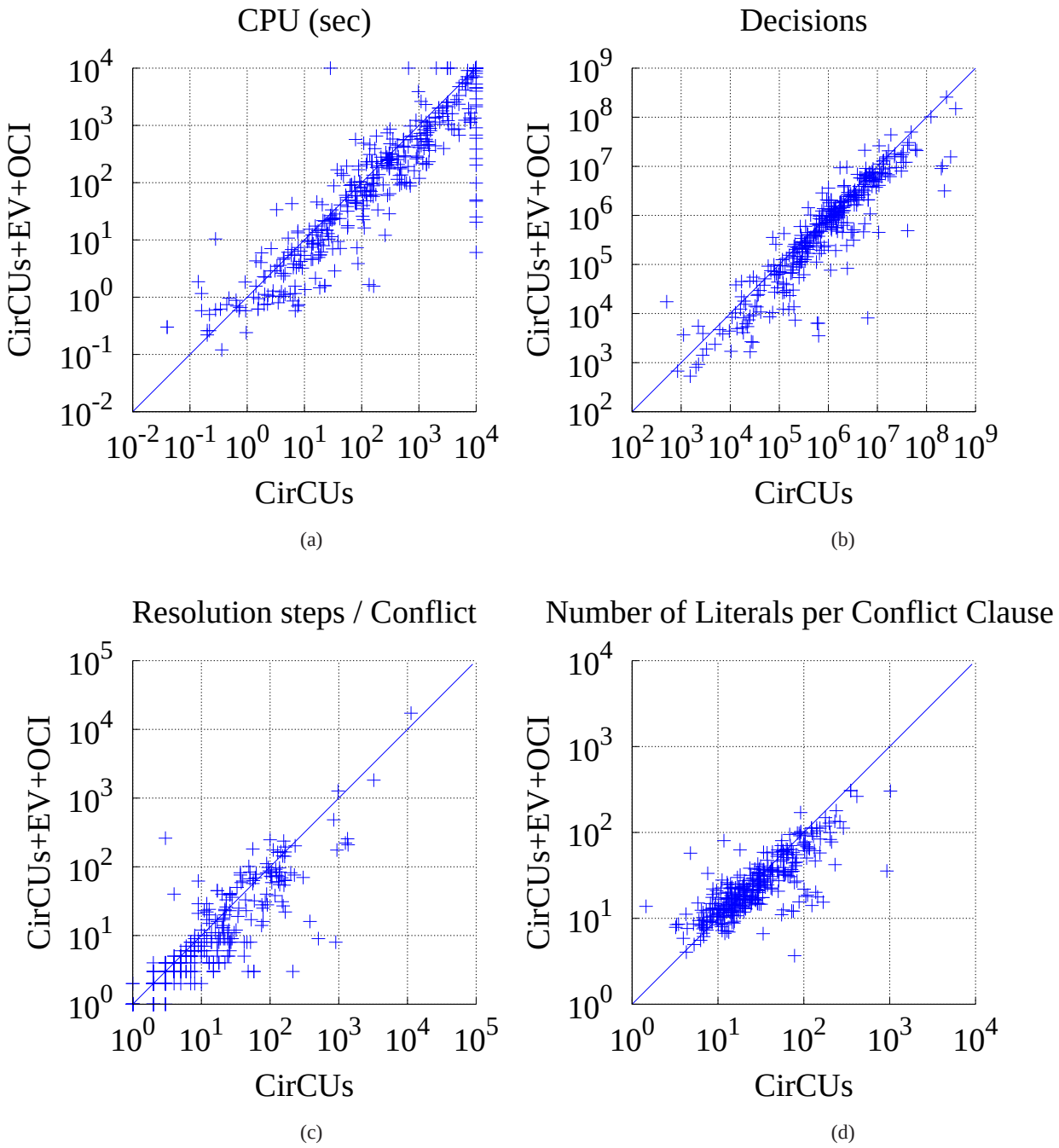



Figure 3.26: Effect of CirCUs with and without EV+OCI on (a) CPU time: GEOMETRIC MEAN = 0.63, $p$-value = $5.7 \cdot 10^{-16}$; (b) number of decisions: GEOMETRIC MEAN = 0.59, $p$-value = $2.2 \cdot 10^{-16}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = 0.68, $p$-value = $2.29 \cdot 10^{-15}$; (d) number of literals per conflict clause: GEOMETRIC MEAN = 0.87, $p$-value = $7.8 \cdot 10^{-5}$.

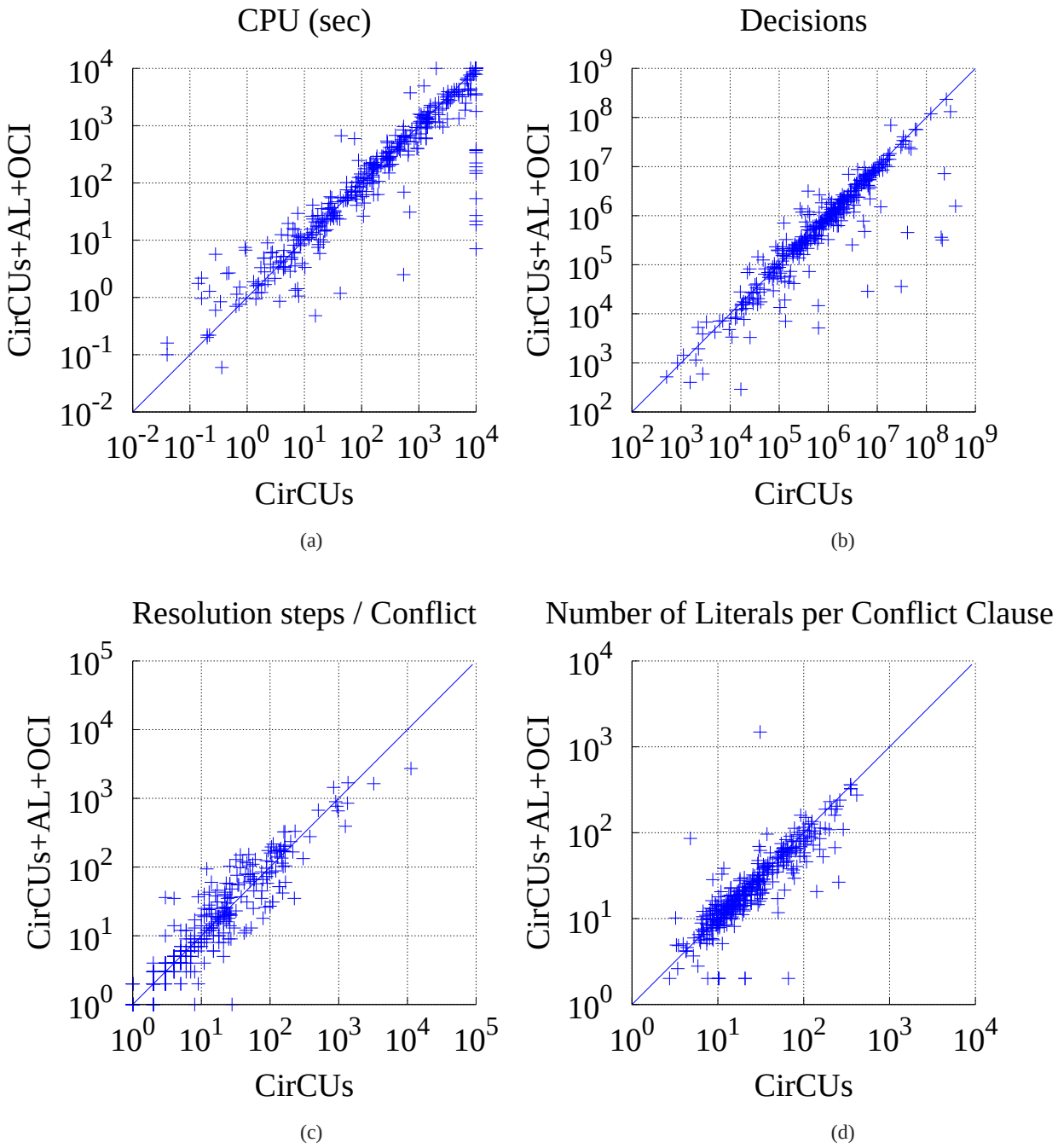



Figure 3.27: Effect of CirCUs with and without AL+OCI on (a) CPU time: GEOMETRIC MEAN = 0.86, $p$-value = 0.003; (b) number of decisions: GEOMETRIC MEAN = 0.77, $p$-value = $1.71 \cdot 10^{-6}$; (c) number of resolution steps per conflict: GEOMETRIC MEAN = 1.01, $p$-value = 0.76; (d) number of literals per conflict clause: GEOMETRIC MEAN = 0.92, $p$-value = 0.03.

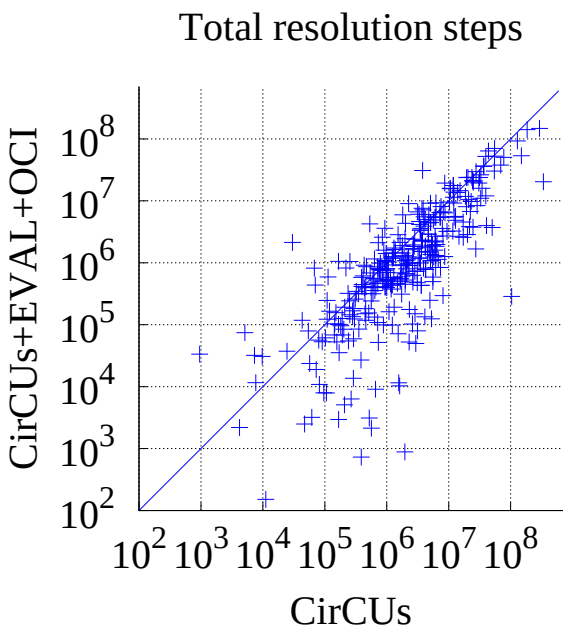



Figure 3.28: The number of resolution steps per conflict.

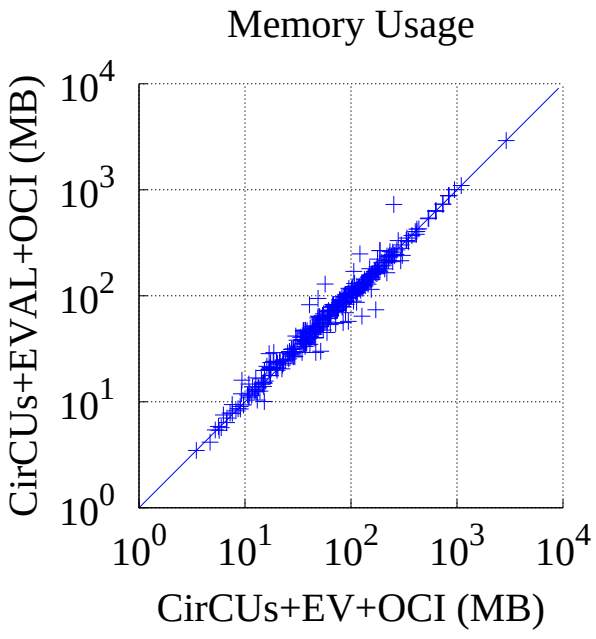



Figure 3.29: The effect on memory consumption.

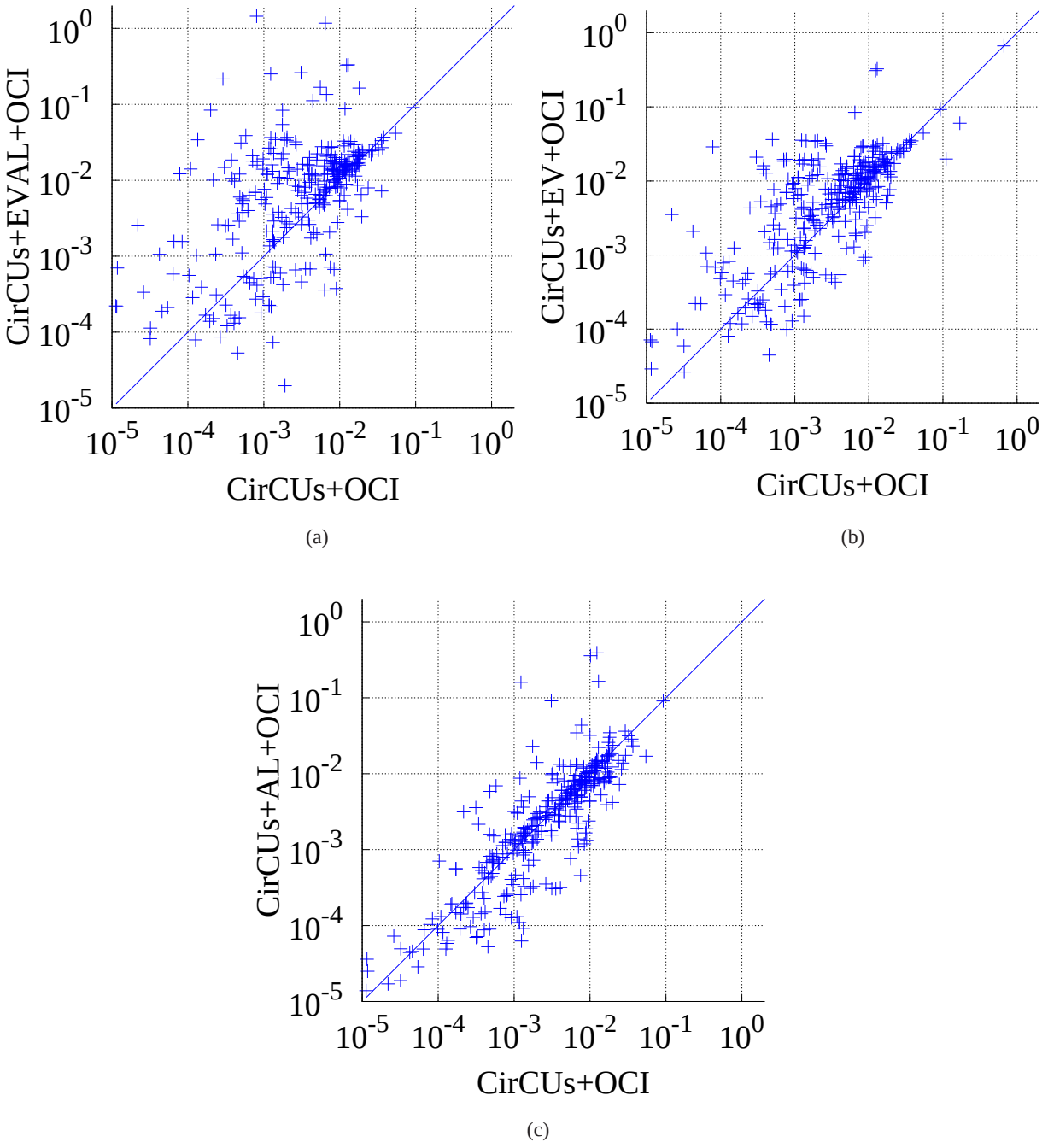



Figure 3.30: Number of OCI applications per resolution step with and without preprocessing: (a) both elimination and distillation: GEOMETRIC MEAN = 1.9, $p$-value = $2.96 \cdot 10^{-9}$; (b) only elimination: GEOMETRIC MEAN = 1.69, $p$-value = $6.65 \cdot 10^{-12}$; (c) only distillation: GEOMETRIC MEAN = 0.92, $p$-value = 0.17.

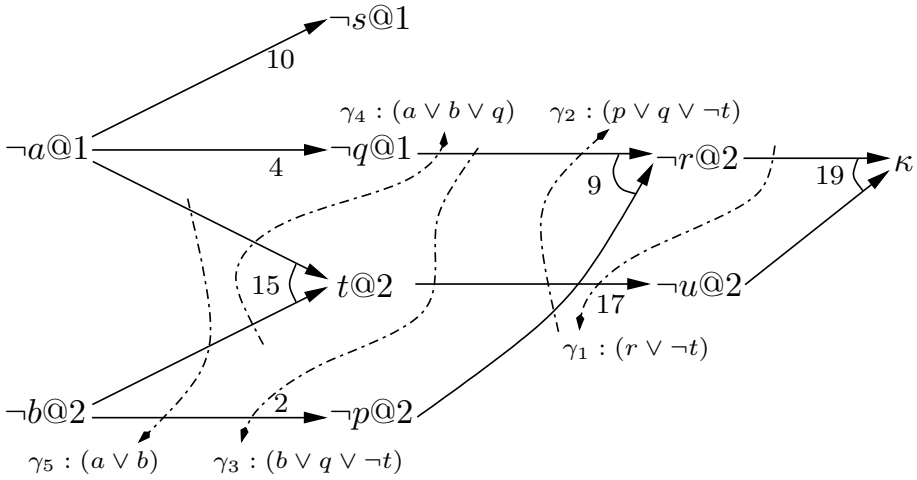



Figure 3.31: Implication graph of Example 3.21 without EV.

*subsumes other resolvents rather than $c_{15}$. 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 3.32 shows that the conflict clause subsumes $c_2$. (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.* □

We now report statistics on the performance of the preprocessors. Figure 3.33 compares the speed of various versions of EVAL to SatELite. (In these plots, SatELite is run on all CNF formulae, while, in Fig. 3.23, the solver may disable SatELite depending on the size of CNF formula.) OCI contributes to the improved preprocessor speed. This is clear in the case of EVAL vs. EVAL+OCI. It is true also without distillation, because EV+OCI removes significantly more clauses and literals than plain EV in about the same time.

It is also interesting to compare the reductions achieved by different preprocessors. In Fig. 3.34, we

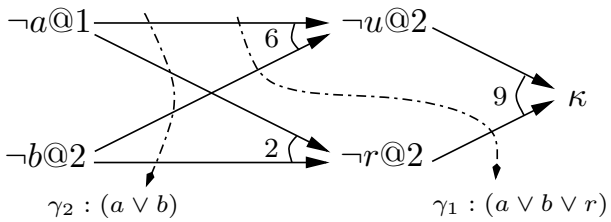



Figure 3.32: Implication graph of Example 3.21 with EV.

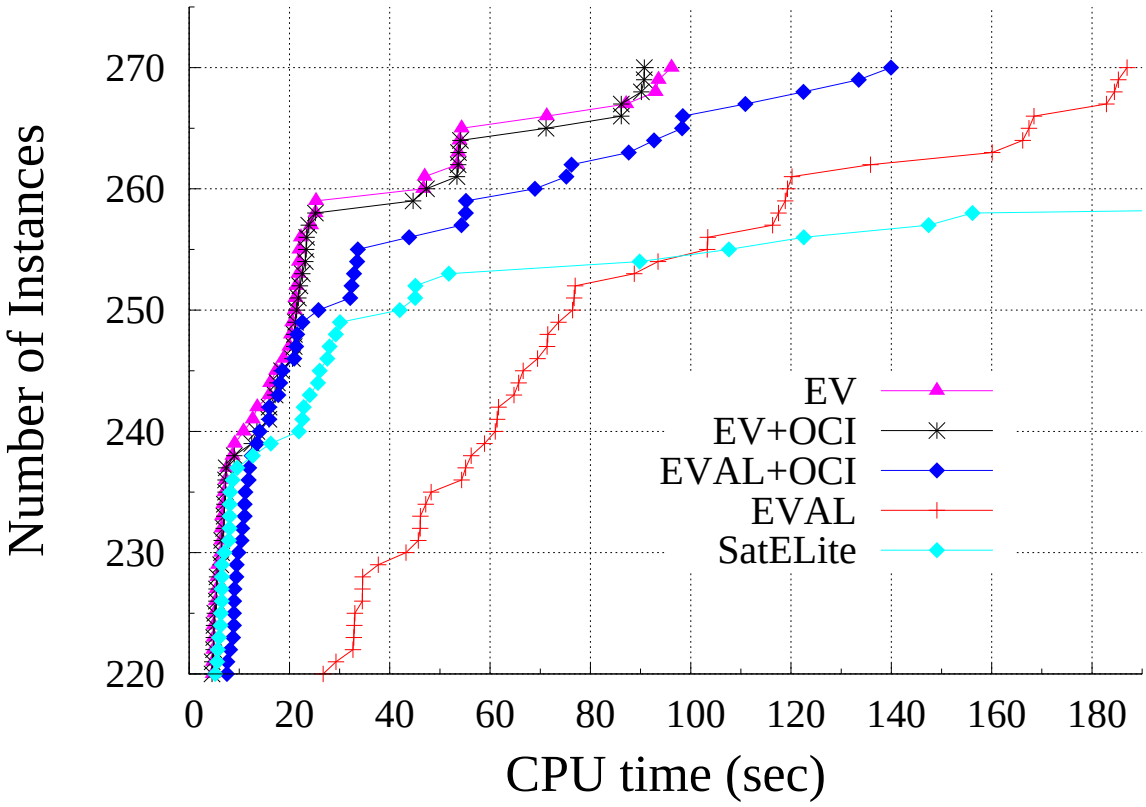



Figure 3.33: Number of instances simplified by various preprocessors versus CPU time. SatELite times out on one instance after 3600 s.

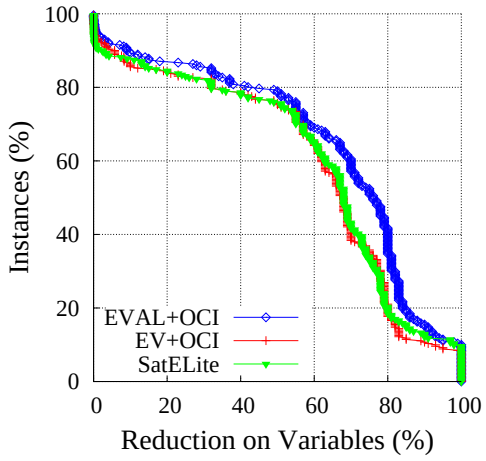
report the fractions of instances that achieve certain reductions in terms of variables, clauses, and literals. About 10% of the instances achieve close to 100% reduction. This means that preprocessing reduces the CNF formulae to either the empty clause or the empty set of clauses. CirCUs's variable elimination is less aggressive than SatELite's: 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. While conflict analysis during distillation may produce additional conflict clauses, the number of added clauses is on average $0.1\%$ of the total. Alembic often achieves more simplifications thanks to the on-the-fly subsumption check. The mean number of clauses simplified per conflict is $0.7$. Moreover, on average, in $51\%$ of the conflicts the 1-UIP clauses subsumes one of the clauses used to resolve it; in those cases, rather than the 1-UIP clause being added to the database, the operand is simplified.

## 3.6 Summary and Discussion

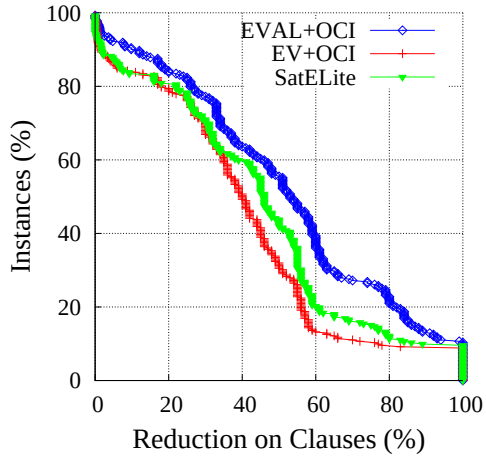
We have presented efficient transformations of a CNF formula that aim at either improving its deductive power or shortening implication graphs. We have shown that the transformations help a DPLL-based SAT solver to run faster by deducing more literals from its decisions and by reducing the depth of the implication graphs used in conflict analysis.

On-the-fly simplification based on self-subsumption can be applied to any stage that uses resolution, e.g., conflict analysis and variable elimination, with minimal overhead. Its application is compatible with advanced conflict analysis techniques and with the generation of unsatisfiability proofs. Another benefit is the reduction of the number of added conflict clauses without detriment for the deductive power.

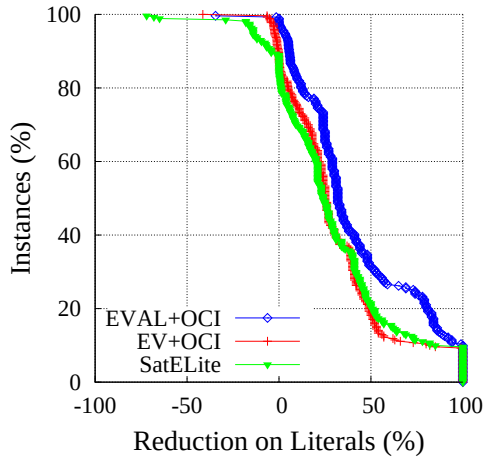
The distillation procedure applied to preprocessing of the CNF formula also considerably speeds up the SAT solver by increasing deductive power. In contrast, we have shown that variable elimination works mainly by reducing the number of resolution steps required in conflict analysis. This results in earlier conflicts, cheaper analyses and better conflict clauses.




(a)




(b)




(c)

Figure 3.34: Ratio of simplification made by various preprocessors on (a) variables, (b) clauses, and (c) literals.

# Chapter 4

## Clause Simplification through Dominator Analysis

In the previous chapter, two notions that help in the design and evaluation of formula transformations have been discussed. The first is deductive power of a CNF formula. It is motivated by the observation that the more consequences the DPLL procedure can deduce from each of its decisions, the more effective the pruning of the search space. The second notion is proof conciseness. It reflects the fact that the DPLL procedure progresses through the search space by proving that parts of that space contain no satisfying assignment and recording such findings in the form of new clauses.

These notions are at work in several techniques that are adopted by state-of-the-art SAT solvers to improve the quality of the CNF clauses. In PrecoSAT [Bie09, Pre], where many features are shared with PicoSAT [Bie08b], a clause with two literals may be derived based on **dominator** analysis during the deduction process. Such a clause variables, but it also tends to shorten the implication graph. In turn, a concise implication graph often benefits the recursive approach of [Bie09] to minimize conflict clauses. This chapter describes the effect of this type of clauses on deductive power and proof conciseness, and propose two main extensions of dominator-based analysis:

- A subsumption check concurrent with dominator computation, and
- the addition of dominator-based supplemental conflict clauses.

This chapter also reports results from the implementation of the proposed approach.

## 4.1 Dominators

While adding a transitive closure clause of the implications does not affect deductive power, it may shorten the implication graph. A more concise implication graph may benefit the procedures that work on it. For instance, the deduction procedure may identify a conflicting clause more quickly, and conflict analysis may resolve fewer antecedents. On the other hand, adding clauses to the CNF database indiscriminately may substantially slow down the deduction procedure. To prevent this, a supplemental clause should be generated only when its usefulness is established by an effective criterion.

In this section we give an overview of the approach to learning new dominator-based clauses presented in [Bie09] with the name of Lazy Hyper Binary Resolution (LHBR). The notion of **dominance** was introduced in [Pro59] for the analysis of flow diagrams. This notion is readily adapted to implication graphs, as the following definition shows.

**Definition 4.1.** *Given an implication graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of directed edges, a node $d \in V$ dominates $v \in V$ if all paths from source nodes of $G$ (decisions) to $v$ go through $d$. Node $d$ is the **earliest dominator** of $v$ if it dominates $v$ and has no other dominator than itself. Also, $d$ **immediately dominates** $v$ if it is the last dominator of $v$ distinct from $v$. Finally, $v$ is the **trivial** dominator of itself.* □

For each node $v$ in $G$ its **dominator set**, denoted by $\text{DOM}(v)$, contains every dominator of $v$. Under Definition 4.1, the dominators of a node are totally ordered and $v \in \text{DOM}(v)$.

A literal $q$ is dominated by $p$ in an implication graph for $F$ if and only if $F \cup \{\{p\}\} \vdash_{\mathcal{D}} q$. Therefore, if $q$ is dominated by $p$ the clause $(\neg p \vee q)$ is an implicate of $F$ by Lemma 3.16. We reserve the name **dominator clause** for the case in which $p \neq q$.

**Example 4.2.** *Consider the following CNF formula $F$:*

$$F = (\neg a \vee b)_1 \wedge (\neg a \vee c)_2 \wedge (\neg b \vee \neg c \vee d)_3 \ .$$

*Suppose that the SAT solver makes decision $a@1$. This leads to the implication graph shown in Fig. 4.1, where literal $a$ dominates literal $d$, i.e., $F \cup \{\{a\}\} \vdash_{\mathcal{D}} d$. Then, $\gamma = (\neg a \vee d)$ is an implicate of $F$. A new*

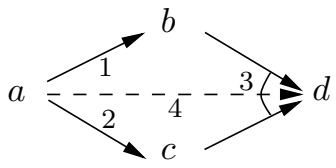



Figure 4.1: Implication graph of Example 4.2

*formula $F'$ obtained by adding $\gamma$ to $F$ may shorten the implication graph as shown by the dashed edge in the figure. Moreover, by Definition 3.3, $F \prec F'$, because $F' \cup \{\{\neg d\}\} \vdash_{\mathcal{D}} \neg a$ but $F \cup \{\{\neg d\}\} \nvdash_{\mathcal{D}} \neg a$.* □

**Example 4.3.** *Consider the following CNF formula $F$:*

$$F = (\neg a \vee b)_1 \wedge (\neg b \vee c)_2 \wedge (\neg b \vee d)_3 \wedge (\neg b \vee \neg c \vee \neg d \vee e)_4 \ .$$

*Suppose that the SAT solver makes decision $a@1$, which yields the implication graph shown in Fig. 4.2. In the implication graph, $c$, $d$, and $e$ share two non-trivial dominators: the earliest dominator $a$ and the immediate dominator $b$. Since $F \cup \{\{a\}\} \vdash_{\mathcal{D}} c$ and $F \cup \{\{a\}\} \vdash_{\mathcal{D}} d$, $\gamma_1 = (\neg a \vee c)$ and $\gamma_2 = (\neg a \vee d)$ are dominator clauses, and are transitive closure clauses. By Lemma 3.6, adding $\gamma_1$ and $\gamma_2$ does not change the deductive power of $F$. On the other hand, $\gamma_3 = (\neg a \vee e)$ and $\gamma_4 = (\neg b \vee e)$ are dominator clauses that increase deductive power, since $F \cup \{\gamma_3\} \cup \{\neg e\} \vdash_{\mathcal{D}} \neg a$ but $F \cup \{\neg e\} \nvdash_{\mathcal{D}} \neg a$. (Similarly for $\gamma_4$.) In particular, since the asserting clause $c_4$ is subsumed by $\gamma_4$, it is removed if $\gamma_4$ is added. This leads to the shorter implication graph where the asserting clause $c_4$ is replaced with $\gamma_4$. Besides, $F \prec F \cup \{\gamma_3\} \prec F \cup \{\gamma_4\} \simeq F \cup \{\gamma_3, \gamma_4\}$. In this case, deducing the negation of the immediate dominator allows one to deduce the negation of all other non-trivial dominators. When the immediate dominator is distinct from the earliest dominator, the dominator clause involving the former is the one that usually gives the greatest boost to deductive power.*

*Dominator clauses do not always increase deductive power. If we consider*

$$F' = F \wedge (e \vee \neg f)_5 \wedge (e \vee \neg g)_6 \wedge (f \vee g \vee \neg c)_7 \ ,$$

*then $\gamma_4$ is still a dominator clause, but its addition does not affect deductive power.* □

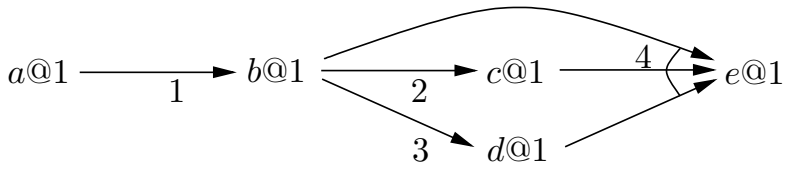



Figure 4.2: Implication graph of Example 4.3

Example 4.3 motivates the CNF transformation implemented in [Pre] by adding dominator clauses, which are derived during the deduction procedure. Let $c$ be a clause of CNF formula $F$. When literal $l$ is deduced from $c$ under a partial assignment, it is annotated with one of its dominators, $d$, which is then used to compute dominators for further implied literals.

The earliest dominator of $l$ is easily computed. The earliest dominator of a decision is the literal itself. For an implied literal $l$, if all predecessors of $l$ share the same earliest dominator $d$, then $d$ is the dominator of $l$ too; otherwise, $l$ is the earliest dominator of itself.

In [Pre, Bie09] a variation of this scheme is used. First, a dominator $d$ is computed for vertex $l$ with the above recursion in terms of the dominators chosen for the predecessors of $l$. These may not be earliest dominators; hence, $d$ may not be the earliest dominator of $l$ either. We call it the **recursive** dominator of $l$. Second, if the clause asserting $l$ has two literals or $d$ is trivial, $l$ is annotated with $d$. Otherwise, the immediate dominator $i$ of $l$ is computed. If the negation of $d$ appears in the asserting clause, then $d$ is $i$. Otherwise, $i$ is found by a search linear in the size of the subgraph of the implication graph between $d$ and $l$. The search is made easy by enforcing the invariant that every predecessor of $l$ is connected to $d$ by exactly one path. (Alternatively, that the asserting clause of every literal $l$ in the implication graph that is not its own dominator has exactly two literals: $l$ and the negation of a dominator of $l$.)

Figure 4.3 describes this procedure. The procedure is performed if the asserting clause $\gamma$ has more than two literals and does not contain the negation of the recursive dominator, “rdom”. For the first antecedent literal alit in $\gamma$, for which idom == 0 (line 12), the procedure sets alit as the first candidate of the immediate dominator, i.e. idom == alit (line 13), and traces back up to rdom as marking the literals between alit and rdom with MARK(lit) (lines 14–18). For the search of the remaining antecedent literals (line 19),

the procedure traverses back from the current antecedent literal alit, until it hits a literal on the marked path (line 20), or it meets either idom or rdom (lines 23 and 24). If a new endpoint of the marked path is met, it becomes a new idom and the literals between the new and old idom are unmarked (lines 26–27).

**Example 4.4.** *Consider the implication graph shown in Fig. 4.4. In the implication graph, $e$ is asserted from $(\neg b \vee \neg c \vee \neg d \vee e)_3$ and its predecessors have the common recursive dominator $a$. Since the negation of the recursive dominator is not contained in the asserting clause $c_3$, the procedure* COMPUTEDOMINATORINPRECOSAT *of Fig. 4.3 is invoked with arguments $c_3$ and $a$ to search the immediate dominator of $e$. Supposed that in $c_3$, the procedure examines $c$, $b$, and $d$ in that order (lines 3 and 4). Initially idom = 0. Then the search from $c$ sets $c$ as a new idom (line 13), and the procedure marks all the literals ($b$ and $c$) along the path between $e$ and $a$ (lines 14–18). In the search from the next antecedent literal $b$, since $b$ is marked, it becomes a new idom (line 26) and the current idom $c$ is unmarked (line 27). For the last search, the procedure meets the current idom $b$ while traversing the implication graph from $d$ (lines 20–25). Therefore, the procedure stops searching from $d$. Since $d$ was the last antecedent, all the marks between $b$ and $a$ are cleared (lines 5 and 6), and $b$ is returned as the immediate dominator of $e$.*

When $i$ is computed, $l$ is annotated with it and the dominator clause $\gamma = (\neg i \vee l)$ is added to $F$. The implication graph is modified accordingly by making $\gamma$ the antecedent of $l$. This simplifies the graph and guarantees that only one path connects $i$ to $l$. If the negation of $i$ is contained in the original antecedent clause $c$, $c$ is subsumed by $\gamma$, as shown in Example 4.3. In the example, $\gamma_4$ computed from immediate dominator $b$ simplifies $c_4$ while $\gamma_3$ based on the earliest dominator $a$ does not.

The use of immediate dominators is motivated by the fact that, if there is a dominator clause that subsumes the asserting clause, then it contains the negation of the immediate dominator. PrecoSAT gives up the chance of finding some non-trivial dominators in return for the ability to simplify clauses using immediate dominators. Besides, Example 4.3 shows that dominator clauses including immediate dominators are also best for deductive power.

**Lemma 4.5.** *If immediate dominator clauses are added for all implied literals with non-trivial dominators, then asserting $\neg l$ causes $\mathcal{D}$ to deduce the negation of all literals in* $\text{DOM}(l) \setminus \{l\}$.

```
1   ComputeDominatorInPrecoSAT(γ,rdom) {
2       idom = 0;
3       for each (antecedent literal lit in γ)
4           idom = SearchDom(¬lit, rdom, idom);
5       for (lit = idom; lit != rdom; lit = Pred(lit))
6           Mark(lit) = false;
7       return idom;
8   }

9   SearchDom(alit, rdom, idom) {
10      assert(rdom != alit);
11      lit = alit;
12      if (idom == 0) {
13          idom = alit;
14          do {
15              assert(size(GetAntecedentClause(lit) == 2)); {
16              Mark(lit) = true;
17              lit = Pred(lit);
18          } while (lit != rdom);
19      } else {
20          while (!Mark(lit)) {
21              assert(size(GetAntecedentClause(lit) == 2)); {
22              lit = Pred(lit);
23              if (lit == rdom) break ;
24              if (lit == idom) break ;
25          }
26          for ( ;idom != lit; idom = Pred(idom))
27              Mark(idom) = false ;
28      }
29      return idom;
30  }
```

Figure 4.3: Dominator analysis in PrecoSAT

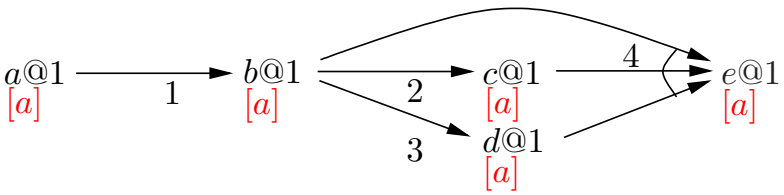



Figure 4.4: Implication graph of Example 4.4

*Proof.* If for any literal $l$ in the implication graph that has a non-trivial dominator its dominator clause is added to $F$, then $l$ is connected to all its non-trivial dominators by a chain of two-literal clauses because $\text{DOM}(l)$ is totally ordered. Let $\text{DOM}(l) = \{d_1, \ldots, d_n\}$ with $d_i < d_j$ for $i < j$. It is then sufficient to observe that $d_i \rightarrow d_{i+1}$ is equivalent to $\neg d_{i+1} \rightarrow \neg d_i$. □

## 4.2 Simplifying Clauses During Deduction

The use of immediate dominators increases the chances of subsumption of the asserting clause by the dominator clause. However, it may lead to missing non-trivial dominators.

**Example 4.6.** *Given the following clauses:*

$$(\neg a \vee b)_1 \wedge (\neg a \vee f)_2 \wedge (\neg b \vee c)_3 \wedge (\neg b \vee d)_4 \wedge$$

$$(\neg b \vee \neg c \vee \neg d \vee e)_5 \wedge (\neg a \vee \neg e \vee \neg f \vee g)_6 \ .$$

*Suppose that $a@1$ is assigned as a decision. Propagating this assignment results in the implications shown in Fig. 4.5, where literals in square brackets are the dominators computed by the algorithm of [Pre]. Since $b$, $c$, $d$, and $f$ are asserted by two-literal clauses, they are annotated with their earliest dominator $a$. For $e$ asserted by $c_5$, its immediate dominator $b$ is computed because $c_5$ does not contain the negation of the earliest dominator $a$. Literal $b$ is used for the dominator computation when $g$ is implied through $c_6$. However, since the other predecessor $f$ has a different dominator from $e$, $g$ is computed as its own dominator. This leads to missing the opportunity to simplify $c_6$ to $\gamma = (\neg a \vee g)$, which would be derived with the earliest dominator $a$.* □

Example 4.6 shows that some simplification opportunities may be missed if vertices are labeled with their immediate dominators because fewer non-trivial dominators may be found. On the other hand, Example 4.3 shows that the exclusive use of earliest dominators may prevent other simplifications, when the immediate dominator is distinct from the earliest one. The simplifications of both approaches can be obtained within the same complexity bound by labeling each vertex with its earliest dominator, but computing the immediate dominator as well. The next example shows that even the combined approach misses some

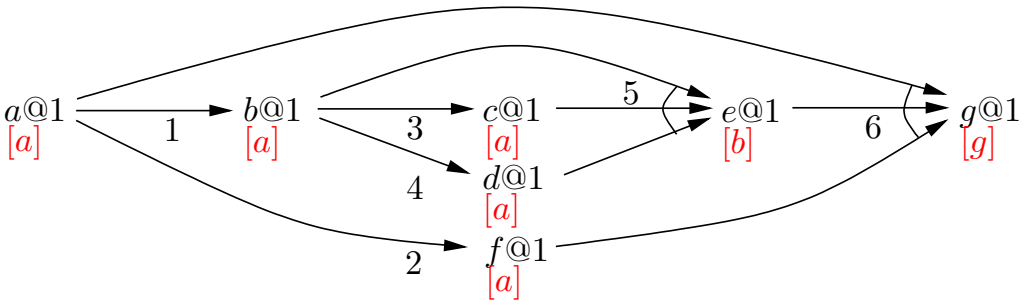



Figure 4.5: Implication graph of Example 4.6

opportunities for simplification. However, we can directly check for self-subsumption between the asserting clause and other implicates of $F$ that may or may not be present in the database. We now demonstrate how this multistep resolution can be integrated with the search for the immediate dominator.

**Example 4.7.** *Given the following clauses:*

$$(\neg a \vee b)_1 \wedge (\neg b \vee c)_2 \wedge (\neg b \vee d)_3 \wedge (\neg c \vee e)_4 \wedge (\neg d \vee f)_5 \wedge$$

$$(\neg c \vee \neg d \vee \neg e \vee \neg f \vee g)_6 \ .$$

*Suppose that propagating $a@1$ results in the implications shown in Fig. 4.6. Literal $b$ is found as the immediate dominator of $g$ in the graph. Since the asserting clause $c_6$ does not contain $b$, it cannot be simplified by the immediate dominator clause. However, $c_6$ can be simplified to $(\neg c \vee \neg d \vee g)$ because $c$ and $d$ imply $e$ and $f$, respectively.* □

An antecedent literal $a$ of a clause $\gamma$ asserting $l$ can be removed by self-subsumption with another implicate of $F$ if each path between the immediate dominator of $l$ and $a$ goes through some other literal in $\gamma$. If only one path connects a literal to its dominator, the check is simple and efficient. Self-subsumption is possible even if the immediate dominator is not among the literals in $\gamma$.

The pseudocode for the check is shown in Figure 4.7. As in [Pre], the procedure is performed if the asserting clause $\gamma$ has more than two literals and does not contain the negation of the recursive dominator. The immediate dominator is known to be on all paths connecting the recursive dominator "rdom" to the antecedent literals in $\gamma$. Therefore, it is known to be somewhere on the unique path between "rdom" and

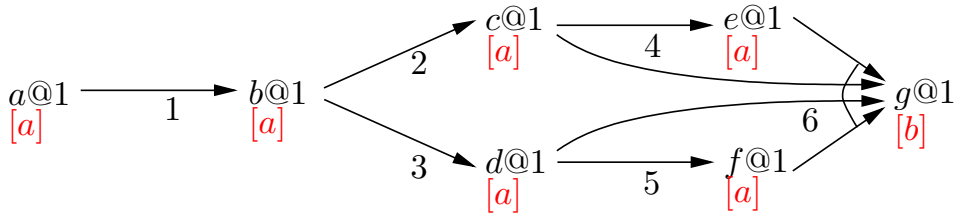



Figure 4.6: Implication graph of Example 4.7

the first such literal, for which idom == 0 (line 18). All the vertices of the implication graph on that path are marked as candidate immediate dominators with MARK(lit) $=$ true (lines 20–28). Tracing back (with PRED($l$)) from the remaining literals, i.e., idom != 0 (line 29), until a marked literal is hit eliminates more candidates until the position of the immediate dominator is known (lines 30–39). Further, if a literal $l$ in $\gamma$ is found while tracing back from another literal $a$ of $\gamma$, i.e., ANTE(lit) $==$ true (lines 23 and 33), then $a$ is marked as redundant with MARK(idom) $=$ false (line 24) and ANTE(alit) $=$ false (line 34). When this occurs during the initial path marking phase, $l$ becomes the new endpoint of the marked path (lines 25–26). Otherwise, the trace back is terminated because the remaining work either was done or will be done when starting from $l$.

While this procedure is based on multistep resolution like on-the-fly simplification during conflict analysis [HS09] and conflict clause minimization [SB09], the use of the earliest dominator to limit the search makes it suitable for frequent use during deduction. The three procedures are complementary: dominator-based simplification resolves an asserting clause with clauses that precede it in the implication graph; on-the-fly simplification resolves an asserting clause with clauses that follow it in the implication graph, while conflict clause minimization does not modify clauses in the implication graph. Moreover, the simplification of the implication graph that results from replacing asserting clauses with dominator clauses speeds up the other two procedures.

```
1   COMPUTEDOMINATORANDSIMPLIFY(γ,rdom) {
2       idom = 0;
3       for each (antecedent literal lit in γ)
4           ANTE(¬lit) = true;
5       for each (antecedent literal lit in γ)
6           idom = SEARCHDOMANDSUBSUME(¬lit, rdom, idom);
7       for (lit = idom; lit != rdom; lit = PRED(lit))
8           MARK(lit) = false;
9       for each (antecedent literal lit in γ)
10          if (ANTE(¬lit) == false)
11              REMOVE(γ, lit);
12          else ANTE(¬lit) = false
13      return idom;
14  }

15  SEARCHDOMANDSUBSUME(alit, rdom, idom) {
16      ASSERT(rdom != alit);
17      lit = alit;
18      if (idom == 0) {
19          idom = alit;
20          do {
21              MARK(lit) = true;
22              lit = PRED(lit);
23              if (ANTE(lit)) {
24                  ANTE(idom) = false;
25                  for ( ;idom != lit; idom = PRED(idom))
26                      MARK(idom) = false ;
27              }
28          } while (lit != rdom);
29      } else {
30          while (!MARK(lit)) {
31              lit = PRED(lit);
32              if (lit == rdom) break ;
33              if (ANTE(lit)) {
34                  ANTE(alit) = false;
35                  return idom;
36              }
37          }
38          for ( ;idom != lit; idom = PRED(idom))
39              MARK(idom) = false ;
40      }
41      return idom;
42  }
```

Figure 4.7: Dominator analysis with simplifying asserting clauses

## 4.3 Dominator Clauses and Redundancy

In this section we study when dominator clauses may duplicate existing clauses and how, on the other hand, dominator analysis may help a SAT solver remove redundant literals from clauses other than the asserting clauses and remove subsumed clauses from the database.

To minimize overhead, the SAT solver should not add dominator clauses that duplicate clauses already in $F$.

**Example 4.8.** *Consider the following formula $F$:*

$$F = (\neg a \vee b)_1 \wedge (\neg a \vee \neg b \vee c)_2 \wedge (\neg a \vee c)_3 \ .$$

*Suppose that the SAT solver makes decision $a@1$, and examines $c_1$ and $c_2$ in order. Then, $b$ and $c$ are asserted by $c_1$ and $c_2$, respectively. Since $c_2$ has more than two literals and $a$ is found as the dominator of $c$, the dominator clause $\gamma = (\neg a \vee c)$ is generated. However, $\gamma$ subsumes both $c_2$ and $c_3$. In particular, $c_3$ is a duplicate of $\gamma$ and is satisfied by $c$. Therefore, $F \simeq F \cup \{\gamma\}$.* □

Notice that in Example 4.8 the dominator clause is not in the implication graph and subsumes the asserting clause.

While duplication is possible, if the SAT solver processes implications in the order in which it discovers them—which is the usual way—rather strong conditions must be met. These conditions for duplication are described in the following lemma.

**Lemma 4.9.** *Suppose implications are processed in first-in, first-out manner. Let $\gamma = \{\neg d, l\}$ be a dominator clause, where $d$ is the dominator of $l$. If it is already present in formula $F$, $\gamma$ is not an asserting clause in the implication graph and it subsumes the asserting clause for $l$.*

*Proof.* Let $\gamma = (\neg d \vee l)$ be a dominator clause computed for $l$ from asserting clause $c$. Assume first that $\gamma$ is asserting in the implication graph. Then $c$ contains $l$ and at least another literal $l'$ that is deduced from $d$. Asserting $d$ makes $\gamma$ a unit clause so that $l$ is implied through it before the implications of $l'$ are examined. That prevents $\gamma$ from being found as dominator clause, resulting in a contradiction. Suppose now $\gamma$ is not

in the implication graph, but is already in $F$. Suppose $d$ is not in $c$. Then, $l$ is implied from $\gamma$ before it is implied from $c$. This prevents duplication. Therefore, $d$ appears in $c$, and $c$ is subsumed by $\gamma$. □

Lemma 4.9 suggests that duplication is not a frequent occurrence in solvers that preprocess their input and possibly remove more redundancies during DPLL. Also notice that, if in Example 4.8 $c_3$ is processed before $c_2$, no duplicate is generated. This is the case of a SAT solver, like PrecoSAT, that handles the clauses with two literals before other clauses in the deduction procedure. This approach is not adopted by many other DPLL-based SAT solvers. Besides, one can detect clauses subsumed by dominator clauses on-the-fly during the deduction procedure. That is, during the deduction procedure, if a clause $c$ that is found to be satisfied by literal $l$ contains the negation of dominator $d$ of $l$, then $c$ is subsumed by the dominator clause $(\neg d \vee l)$. The check for containment of the dominator of $l$ may be expensive for any dominator and for clauses with many literals. Hence, this approach should be applied with restraint: for example, only checking whether the recursive dominator of $l$ (and possibly of a few more true literals in the clause) is the false literal that caused the clause to be examined. The annotation of each literal in the implication graph with its recursive dominator allows this test to be carried out even when the corresponding dominator clause is not added to the database. If not all satisfied literals are checked, subsumption may not be detected. On the other hand, subsumption may be found when the examined clause is subsumed by the dominator clause, even though it is not subsumed by the clause asserting $l$.

## 4.4 Garbage Collection

According to the algorithm of Sect. 4.2, a dominator clause $\gamma$ for literal $l$ is obtained through simplification of the asserting clause of $l$. Hence, $\gamma$ may contain the negation of the immediate dominator $d$ which is different from the earliest one $i$. For this case (a dominator clause is based on $i$), the clause subsumption check for non-asserting clauses during the deduction procedure requires to save $i$ with $d$, and it uses both information to simplify the clauses. If $d \neq i$ and $\neg d$ is a literal of clause $c$ that is satisfied by $l$, then there must be a dominator clause based on $i$ that subsumes $c$, and hence $c$ is removable. Otherwise, checking the containment of $\neg d$ in $c$ can also simplify $c$ to a new dominator clause $(\neg d \vee l)$.

Applying on-the-fly simplification to the deduction procedure in addition to conflict analysis may lead to the deletion of clauses—for instance, clauses subsumed by dominator clauses. Deriving unit clauses from dominator clauses also increases the number of clauses which are satisfied by them at level $0$. Therefore, these simplification techniques should be coupled to an efficient scheme for **garbage collection**. Note that subsumed clauses can be simply deleted, because they will never be involved in a proof of unsatisfiability.

Deleting a clause $c$ is relatively expensive; $c$ is deleted after finding its position in the clause database; the clauses after that point are moved up. Similarly, the watched literals list must be updated. It makes sense to amortize the cost of deletion over multiple clauses by resorting to garbage collection. Let $c = \{l_0, \ldots, l_{n-1}\}$ be a clause and $l_n$ be a dummy literal that appears in no clause and is assigned true at level $0$. Clause $c$ is turned into a satisfied clause by replacing $l_0$ with $l_n$; the solver delays deleting the satisfied $c$ until it gets rid of all the clauses satisfied at decision level $0$. This clause deletion tends to be often invoked due to level $0$ assignments implied by unit clauses derived from dominator clauses.

## 4.5 Dominator-Based Conflict Clauses

In its original formulation, dominator analysis cannot produce implicates with more than two literals and with literals assigned at different decision levels. The following elementary fact and example suggest one way to extend the approach to generate clauses when not all antecedent literals have a common dominator.

**Lemma 4.10.** *No literal $l$ in $F$ implied at level $k$ may have a dominator at a level different from $k$.*

*Proof.* Let $d$ be a dominator of $l$ that is assigned at level $k' \neq k$. Then, by definition $F \cup \{\{\neg d\}\} \vdash_{\mathcal{D}} l$ at level $k'$. This is in contradiction with the assumption. □

**Example 4.11.** *Given the following formula $F$:*

$$F = (\neg a \vee b)_1 \wedge (\neg b \vee c)_2 \wedge (\neg d \vee e)_3 \wedge (\neg b \vee \neg d \vee \neg e \vee f)_4 \wedge (\neg b \vee \neg c \vee \neg f \vee g)_5 \ .$$

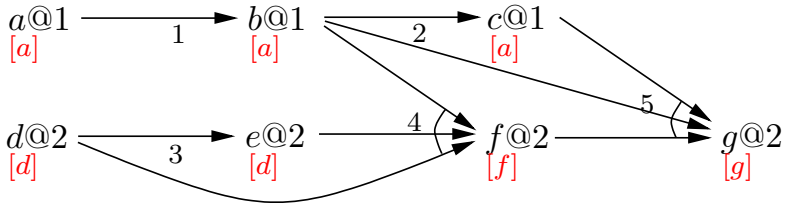



Figure 4.8: Implication graph of Example 4.11

*Suppose that decisions $a@1$ and $d@2$ are made. They result in the implications shown in Fig. 4.8; for each literal the earliest dominator is computed. In this case, no clause is generated if standard dominator analysis is applied. However, two implicates, which subsume existing clauses, can be derived if the algorithm is modified in order to check the dominators of literals at the same level. For instance, the literals of $c_4$, which asserts $f$, can be divided into $d_1 = \{\neg b\}$ and $d_2 = \{\neg d, \neg e, f\}$ according to the decision level of each literal. Application of dominator analysis to $d_2$ results in a dominator clause $\gamma = \{\neg d, f\}$ because $d$ is the earliest dominator of $f$. Clause $c_6 = \gamma \cup d_1$ becomes a new implicate of $F$, and it subsumes the antecedent clause $c_4$. Similarly, $c_7$ is another implicate that is obtained by removing $\neg c$ from $c_5$. Suppose $F$ is transformed to $F' = F \cup \{c_6\}$ and $F'' = F' \cup \{c_7\}$. Then, $F \prec F'$ because $F' \cup \{\{b\}, \{\neg f\}\} \vdash_{\mathcal{D}} \neg d$ but $F \cup \{\{b\}, \{\neg f\}\} \nvdash_{\mathcal{D}} \neg d$, and $F' \prec F''$ because $F'' \cup \{\{b\}, \{\neg g\}\} \vdash_{\mathcal{D}} \neg f$ and $\neg d$ but $F' \cup \{\{b\}, \{\neg g\}\} \nvdash_{\mathcal{D}} \neg f$ nor $\neg d$.* □

In this section we discuss the application of dominator analysis to the derivation of conflict clauses. Once a conflict clause is generated and possibly simplified, the dominator information collected for its antecedent literals can be used thanks to the following lemma.

**Lemma 4.12.** *Let $c = \{l_0, \ldots, l_n\}$ be the asserting clause of literal $l_n$. Let $d_0$ be a dominator of literal $\neg l_0$. Then, $\{\neg d_0\} \cup (c \setminus \{l_0\})$ is an implicate of $F$.*

*Proof.* It is also true that $F \cup \{\{d_0\}, \{\neg l_1\}, \ldots, \{\neg l_{n-1}\}\} \vdash_{\mathcal{D}} l_n$, because $F \cup \{\{\neg l_0\}, \ldots, \{\neg l_{n-1}\}\} \vdash_{\mathcal{D}} l_n$ and $F \cup \{\{d_0\}\} \vdash_{\mathcal{D}} \neg l_0$. By Lemma 3.16, $\{\neg d_0\} \cup (c \setminus \{l_0\})$ is an implicate of $F$. □

Replacing antecedents with their non-trivial dominators therefore produces a new clause that can be used in conjunction with, or as replacement of, the clause computed by conflict analysis. Even though

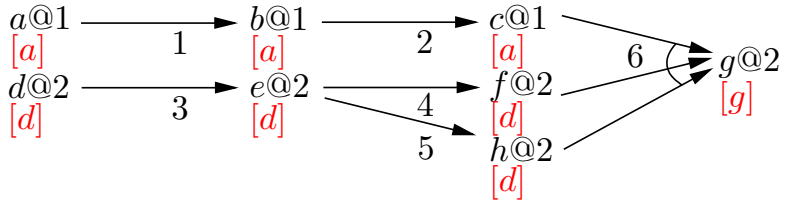



Figure 4.9: Implication graphs of Example 4.13

adding a dominator clause to a CNF formula may not affect its deductive power, when such a clause is derived from a conflict clause based on a UIP, it is at least guaranteed not to be a duplicate and it often improves deductive power.

**Example 4.13.** *Suppose conflict clause* $c_6 = (\neg c \vee \neg f \vee \neg h \vee g)$*, where* $g$ *is the UIP, is added and when after backtracking it becomes asserting, the implication graph is the one shown in Fig. 4.9. Suppose clause* $c_7 = (\neg a \vee \neg d \vee g)$ *is generated from* $c_6$ *by replacing the antecedent literals of* $g$ *with their recursive dominators. Since* $c_7$ *contains a UIP (*$g$*), it can substitute* $c_6$ *as a conflict clause. When* $g$ *is asserted by* $c_7$ *and the implication graph is shortened. To generate more compact implication graphs, we could always substitute a standard conflict clause like* $c_6$ *with a dominator-based conflict clause like* $c_7$*. However, this unlimited replacement may lead the SAT solver to miss some implications that it would have found with the standard conflict clauses. For instance, if* $c_7$ *replaces* $c_6$*, and later in the search,* $d@1$ *and* $\neg g@2$ *are assigned as decisions, literals* $e@1$*,* $f@1$*,* $h@1$*, and* $\neg a@2$ *are implied. However, if* $c_6$ *exists in the database,* $\neg c$ *and* $\neg b$ *are also implied as shown in the implication graph of Fig. 4.10.* □

One may add both conflict clauses (e.g., $c_6$ and $c_7$ of Example 4.13), but the overhead is not negligible. Hence, supplemental conflict clauses based on dominators should be carefully generated and added to

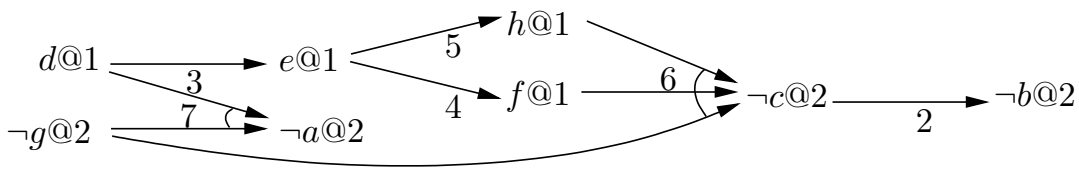



Figure 4.10: Implication graphs of Example 4.13

standard conflict clauses rather than replacing them.

Our approach is to produce a dominator-based conflict clause $\gamma'$ only when a newly-found conflict clause $\gamma$ is obtained by on-the-fly simplification during conflict analysis [HSJ10]. In this case, since $\gamma$ already exists in the database, adding $\gamma'$ has an acceptable cost.

The pseudocode of Fig. 4.11 shows the procedure that is run whenever a clause is asserting. A dominator clause is learned only when a newly found conflict clause becomes asserting and this invokes procedure ANALYZEDOMINATOR with learned $=$ true (lines 7–9). Otherwise, simplification based on either single or multiple dominators is applied to the asserting clause (lines 3–4).

The replacement based on recursive dominators is straightforward and inexpensive: while computing the earliest dominator of the asserted literal, it is sufficient to check whether such dominator is the negation of one of the antecedent literals of the clause. This can be done by a single scan of the literals.

## 4.6 Experimental Results

We have implemented the algorithms for clause simplification during deduction and addition of dominator clauses in the CNF SAT solver **CirCUs** 2.0 [HJKS09, VIS]. The benchmark suite is composed of all the CNF instances (with no duplicates) from the industrial category of the SAT Races of 2006, 2008, and the SAT Competitions of 2009 [SATa]. We conducted the experiments on a 2.4 GHz Intel Core2 Quad processor with 4GB of memory. We used 10000 seconds as timeout, and 2GB as memory bound. We tested PrecoSAT 236 [Pre] along with CirCUs 2.1 to provide a reference point. We denote the extensions by DOM (Dominator analysis), DOMSUB (Dominator analysis with Subsumption check on asserting clauses), DSSCL (Dominator-based Simplification on Satisfied Clauses), and DCCL (Dominator-based Conflict Clause generation).

The results are summarized in the graph of Fig. 4.12, which shows the number of instances completed in a given CPU time. The graphs of Fig 4.13, 4.14, 4.15, and 4.16 give the contributions of the proposed techniques in detail. From the graph it appears that the proposed techniques help CirCUs complete more instances within 10000 s, but provide limited benefits for simpler SAT problems. In fact, for the easier formulae, PrecoSAT is faster, but the improved CirCUs has performance close to that of PrecoSAT. Moreover,

```
1   REPLACECONFLICTLITSWITHDOMS(γ) {
2       γ′ = {UIP(γ)};
3       for each (antecedent literal lit in γ)
4           ANTE(¬lit) = true;
5       for each (antecedent literal lit in γ) {
6           rdom = recursive dominator of ¬lit;
7           if (!ANTE(rdom)) {
8               ANTE(¬lit) = false;
9               ANTE(rdom) = true;
10              γ′ = γ′ ∪ {¬rdom};
11          }
12          else if (rdom == lit)
13              γ′ = γ′ ∪ {lit};
14      }
15      for each (antecedent literal lit in γ′)
16          ANTE(¬lit) = false;
17      return γ′;
18  }
```

Figure 4.11: Algorithm for generating a new conflict clause based on recursive dominators.

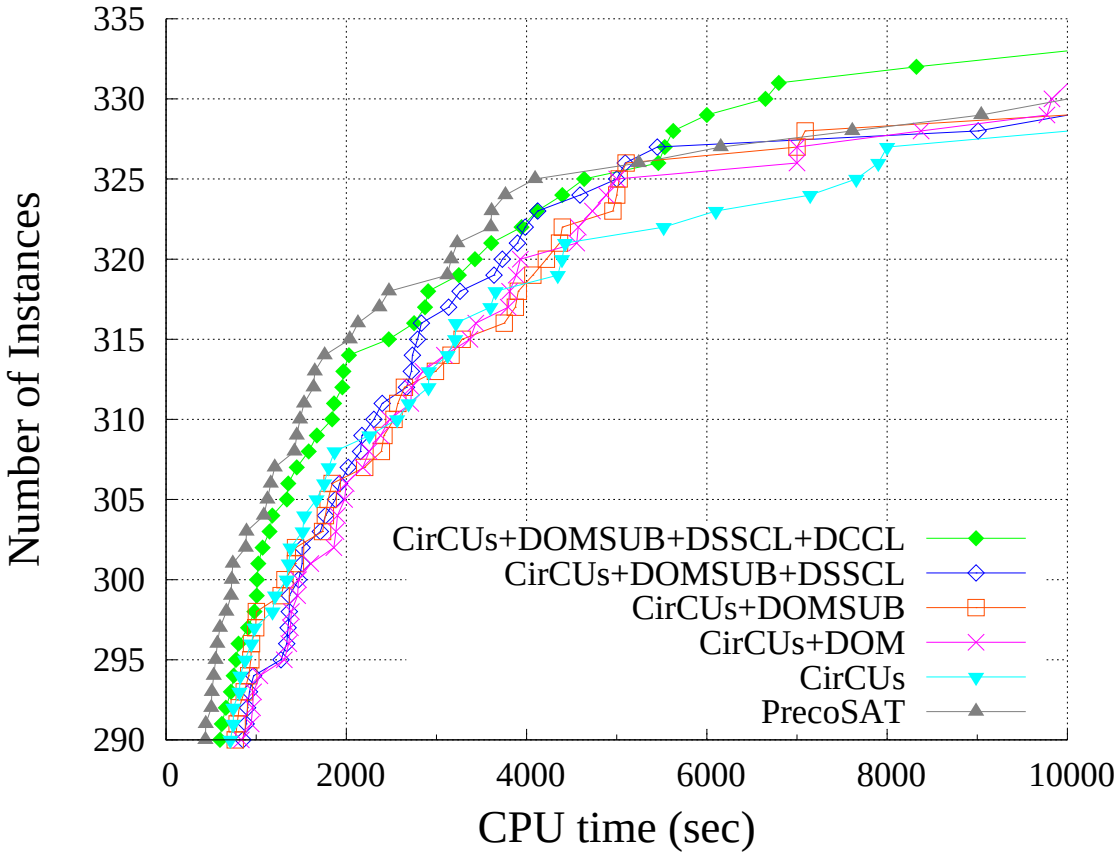



Figure 4.12: CPU time by PrecoSAT and CirCUs with and without proposed techniques

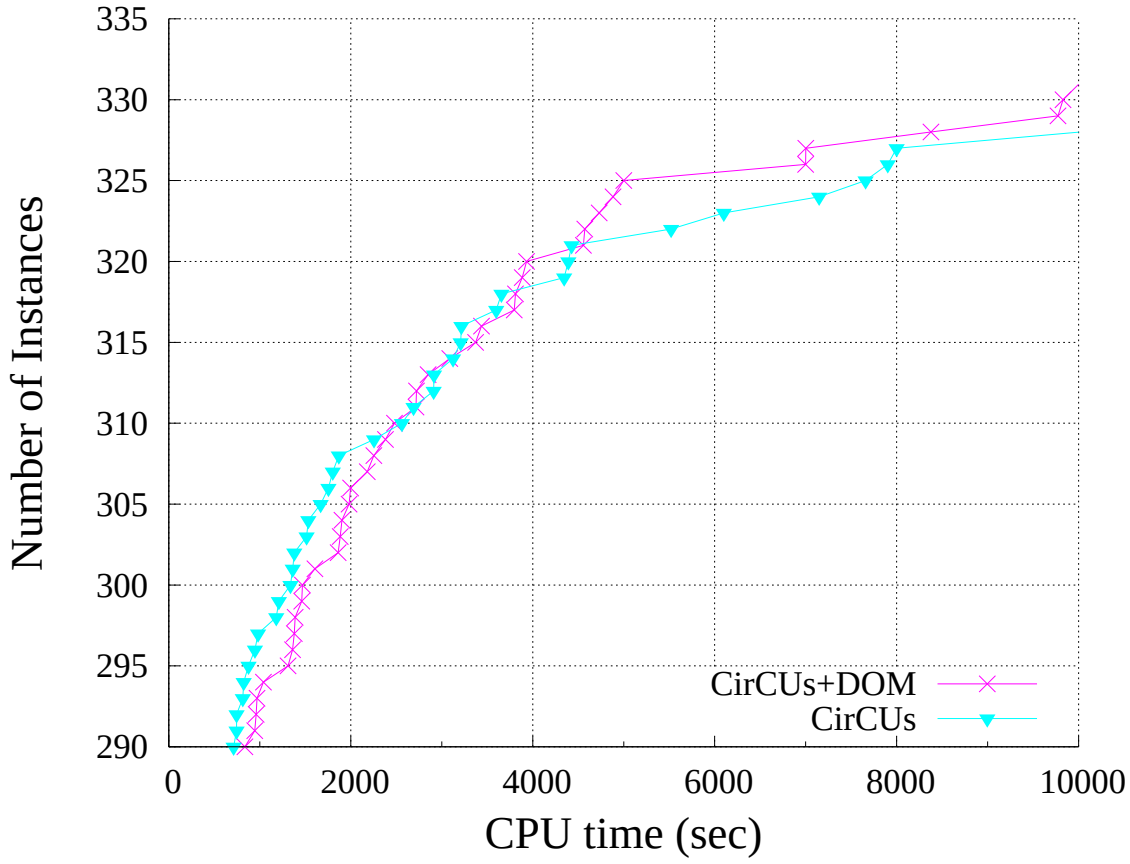



Figure 4.13: The contribution of proposed technique DOM

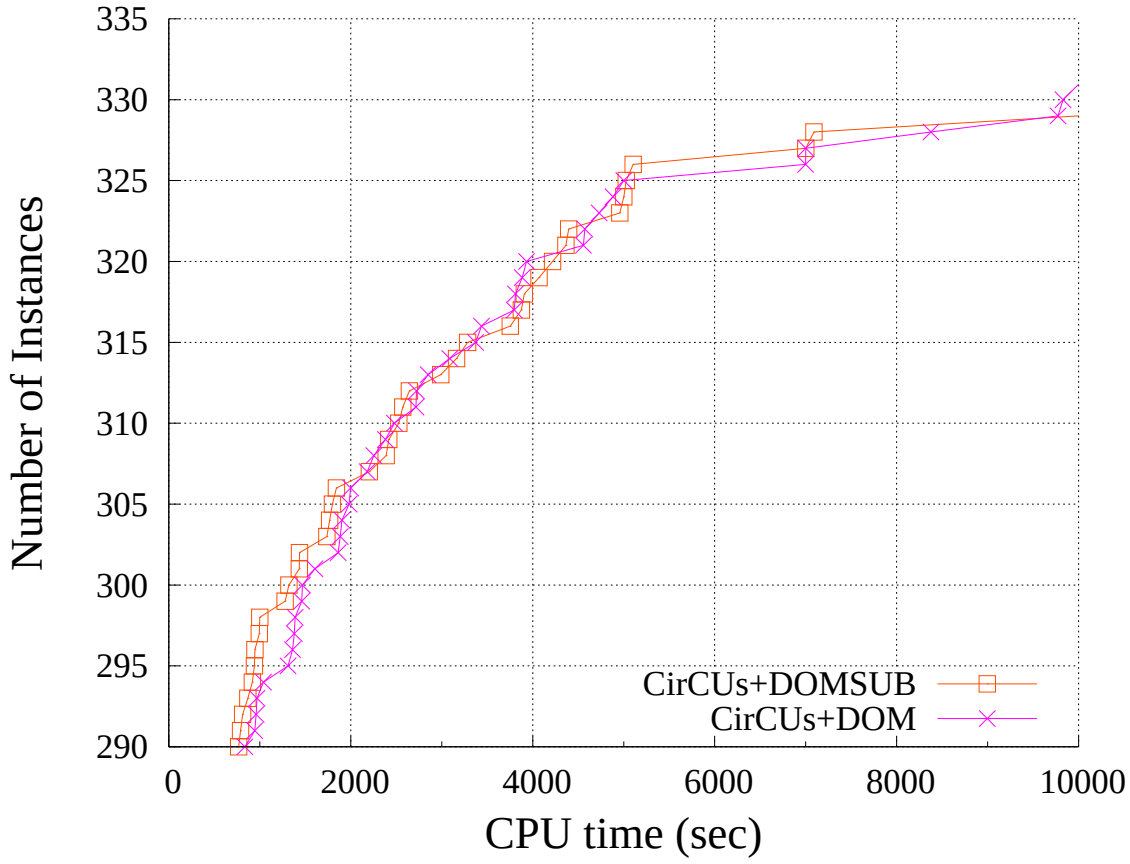



Figure 4.14: The contribution of proposed technique DOMSUB

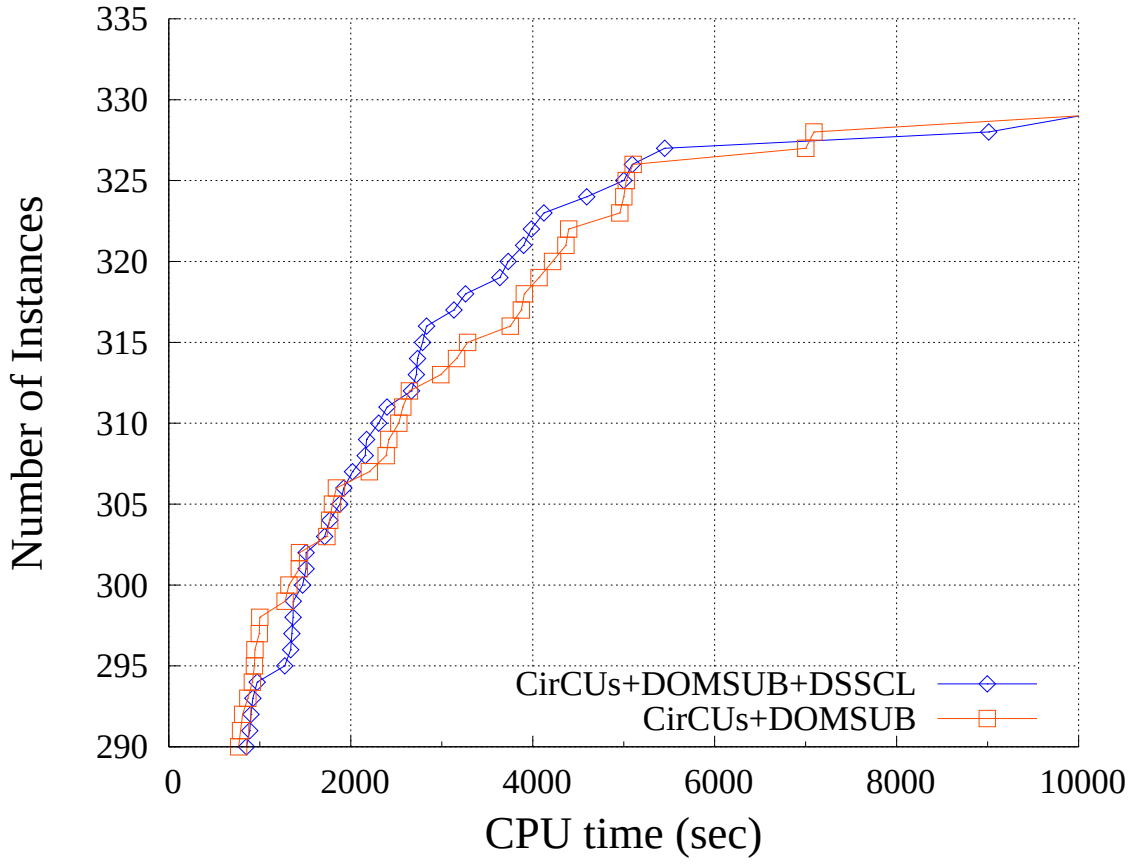



Figure 4.15: The contribution of proposed technique DSSCL

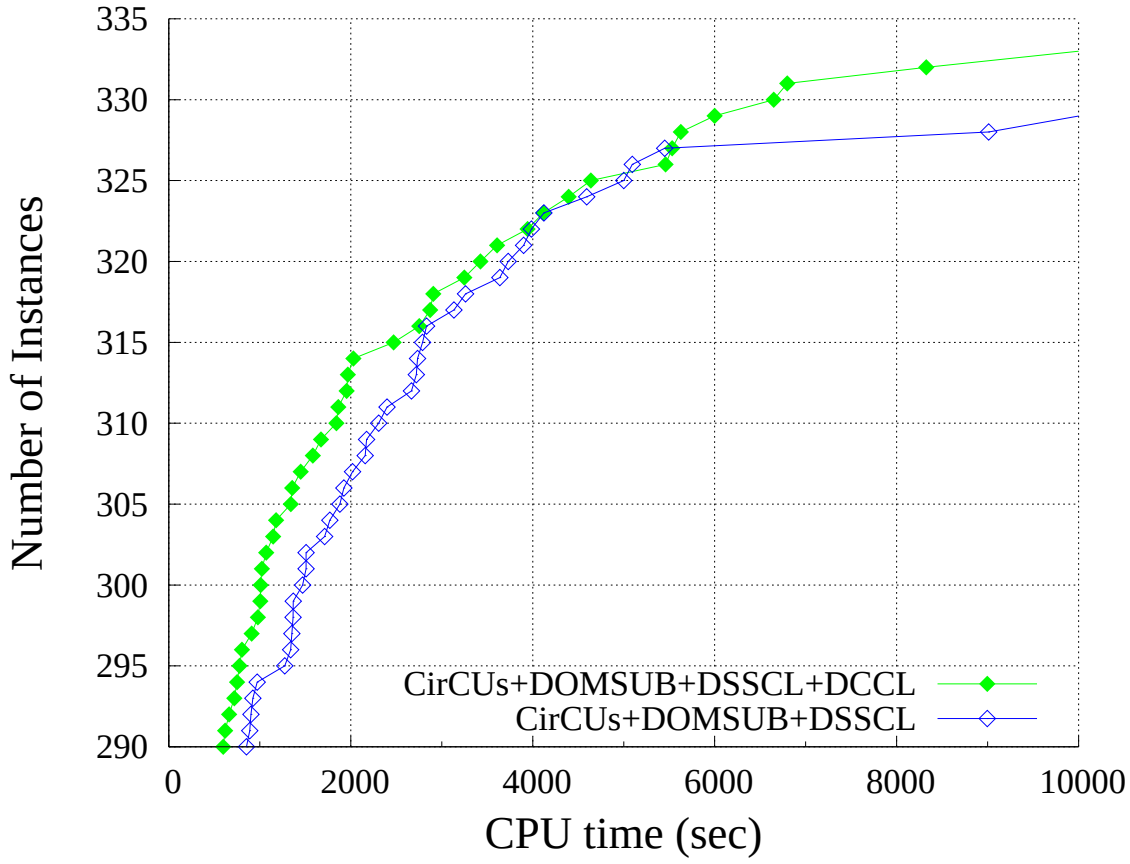
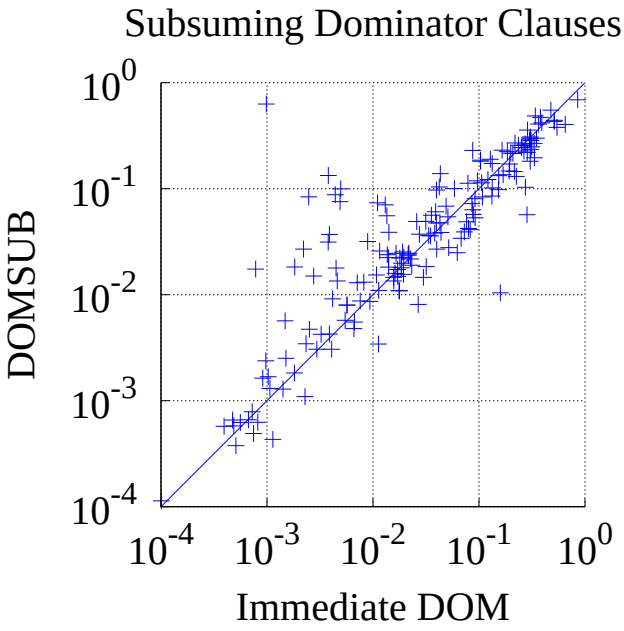
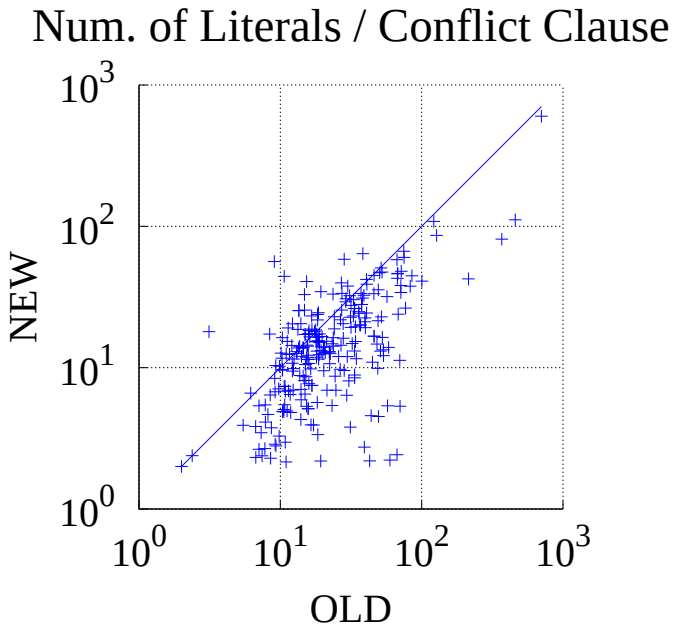



Figure 4.16: The contribution of proposed technique DCCL




(a)




(b)

Figure 4.17: Effect of proposed techniques on (a)the number of subsuming dominator clauses per dominator computation: GEOMETRIC MEAN = 1.11, $p$-value = 0.001; (b)the number of literals per conflict clause: GEOMETRIC MEAN = 0.6, $p$-value = $2.2 \cdot 10^{-16}$

the proposed techniques tends to help CirCUs to solve more hard instances than the base version of CirCUs and PrecoSAT.

Figure 4.17 examine the effects of the proposed techniques (i.e., DOMSUB+DSSCL+DCCL) on (a) the number of dominator clauses subsuming asserting clauses per dominator computation and (b) the number of literals per conflict clause. For each of these quantities the geometric mean of the new/old ratios is reported (excluding cases in which one of the values is $0$). Single-sample $t$-tests were performed to confirm the statistical significance of the data. The null hypothesis was that the geometric mean of the ratios is $1$.

In Fig. 4.17(a), our dominator analysis, i.e., DOMSUB, produces more opportunities for computed dominator clauses to subsume asserting clauses than the analysis of immediate dominators. In our experiments, on average $0.02$ literals were removed by subsuming dominator clauses for every implication.

Analysis of the CirCUs runs show that the major effect of DOMSUB+DSSCL+DCCL is in reducing the number of literals per conflict clause. (See Figure 4.17(b).) Our analysis indicates that this reduction stems from the reduction in the average number of resolution steps per conflict analysis. The reduction in resolution steps is 11% on average and the $p$-value is $1.05 \cdot 10^{-9}$. This translates in a 40% reduction in literals per conflict clause. In contrast, the indicators of increased deductive power are not changed in a decisive way. The number of conflicts per decision shows a 9% improvement on average and its $p$-value is $0.009$. This supports the conclusion that the main way in which dominator clauses improve performance is by affecting proof conciseness.

## 4.7 Summary and Discussion

Dominator analysis, introduced in PrecoSAT [Bie09], is the basis for efficient techniques that allow a SAT solver based on DPLL to simplify the given CNF formula and learn new clauses while deducing new literals. In this chapter, we have introduced two enhancements over the LHBR: a procedure to check clauses for simplification based on self-subsumption that is both more powerful and more efficient than analysis based on immediate dominators; and a low-overhead procedure to learn dominator-based conflict clauses.

In our experiments, the new techniques were especially effective on large, difficult examples. We

hope that a better understanding of the interplay between the new techniques and other components of the solver will lead to improved performance also for the easier SAT instances.

# Chapter 5

# Conclusions

## 5.1 Thesis Conclusions

The purpose of this thesis research is to devise clause transformation techniques that help a DPLL-based SAT solver to run faster by deducing more literals from its decisions and by reducing the depth of the implication graphs used in conflict analysis. Even though many transformation techniques have empirically proved to help a SAT solver prune more of the search space, a formal analysis of their effectiveness has not been attempted. In this thesis, I introduced deductive power and proof of conciseness to characterize them, and proposed the new tranformations applied at several stages in the SAT solver.

In Chapter 3, I have introduced deductive power and proof conciseness for DPLL-based SAT solvers. Then, I have presented how to evaluate the effectiveness of existing clause transformations in terms of the two notions. First, I have shown that simplifications based on self-subsumption check guarantee no deterioration in deductive power. More importantly, the addition of standard conflict clauses has been proved to improve the deductive power of the input formula always. By contrast, the empirical analysis of variable elimination showed the enhance performance of our SAT solver due to generation of more compact implication graphs. Second, I proposed two new techniques based on self-subsumption, both of which efficiently improve the deductive power of CNF formuale. On-the-fly simplification detects subsumption relation between clauses at negligible cost, and it can be applied at any stage using resolution, such as conflict analysis and variable elimination. The distillation procedure, which is implicitly extended from self-subsumption check, is applied to preprocessing of the CNF formula.

In Chapter 4, I have presented dominator-based clause learning scheme applied in PrecoSAT. First,

I presented how adding dominator clauses during implication process is effective in shortening the implication graph, and hence in deducing other literals quickly. This analysis has been extended to check self-subsumption relation over antecedent clauses with inexpensive computation. I also proposed a new scheme to generate conflict clause based on dominators.

I conducted the experiments over various benchmarks that are obtained from real SAT problems to demonstrate the effectiveness of the proposed techniques. In depth analysis of results have shown that the the proposed CNF transformations contribute in improving the performance of our SAT solver in practice. In particular, I expected that dominator-based simplification techniques may improve the deductive power of the given formula, but my experimental results showed that it primarily lead to more concise proofs.

## 5.2 Future Work

The proposed techniques have several extensions that are worth of investigation: generation of small unsatisfiable cores, application to restarts and solution enumeration, application to non-clausal reasoning, and logic synthesis and representation of sets by characteristic functions in CNF [McM02].

Generation of small unsatisfiability cores is one of the most required procedures in formal verification applications [AKMM03, KOSS04, GLST05, McM03, LS06, Li06]. An unsatifiable core is extracted from the original clauses involved in generating the empty clause. This is performed by analyzing the implication graphs generated during the DPLL procedure. Hence, proof conciseness is a meaningful criterion to evaluate a CNF transformation with respect to the generation of small unsatisfiability cores.

In this thesis, CNF transformations are only considered as a way to improve the deduction procedure of DPLL-based SAT solvers. However, there have been various approaches to make DPLL-based SAT solvers faster. One example is SAT encoding. Some optimizations of the encoding can be performed in the form of preprocessing before SAT solving. These techniques allow for significant reductions in the size of the resulting propositional formulae, and in consequent improved performance of the SAT solver. Hence, such translation techniques from a circuit to a CNF formula may be characterized in terms of deductive power and proof conciseness. For this, first, the formal definitions of two notions should be extended to non-clausal formulae and reasoning.

## Bibliography


[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. Data Structures and Algorithms. Addison-Wesley, Reading, MA, 1983.

[AKMM03] N. Amla, R. P. Kursahn, K. L. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03), pages 34–48, Warsaw, Poland, April 2003. LNCS 2619.

[Bar] URL:http://www.lsi.upc.edu/ oliveras/bclt-main.html .

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[Bie08a] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In Theory and Applications of Satisfiability Testing SAT 2008, pages 28–33. Springer-Verlag, 2008. LNCS 4996.

[Bie08b] Armin Biere. PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation, 4(2–4):75–97, 2008.

[Bie09] A. Biere. P{re,i}coSAT@sc'09. SAT Competition 2009 - Solver Description, June 2009.

[Bor97] A. Borälv. The industrial success of verification tools based on stålmarck's method. In Computer Aided Verification, 9th International Conference (CAV'97), pages 7–10, Haifa, Israel, June 1997. Springer-Verlag. LNCS 1254.

[Bor98] A. Borälv. Case study: Formal verification of a computerized railway interlocking. Formal Asp. Comput., 10(4):338–360, 1998.

[Bra01] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1, pages 515–520, San Francisco, CA, 2001. Morgan Kaufmann Publishers Inc.

[BS98] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In Proceedings of the International Conference on Computer-Aided Design, pages 570–576, San Jose, CA, November 1998.

[CG05] B. Cook and G. Gonthier. Using stålmarck's algorithm to prove inequalities. In Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods (ICFEM 2005), pages 330–344, Manchester, UK, November 2005. Springer-Verlag. LNCS 3785.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. An Introduction to Algorithms. McGraw-Hill, New York, 1990.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, pages 151–158, New York, 1971. Association for Computing Machinery.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. Communications of the ACM, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. Journal of the Association for Computing Machinery, 7(3):201–215, July 1960.

[EB05] 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), pages 61–75, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[EMS07] 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.

[ES03] 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.

[ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. JSAT, 2(1-4):1–26, 2006.

[GAG+02] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In Proceedings of the Design Automation Conference, pages 747–750, New Orleans, LA, June 2002.

[GLST05] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '05, pages 122–131, New York, NY, USA, 2005. ACM.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 142–149, Paris, France, March 2002.

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In Design, Automation and Test in Europe (DATE'03), pages 886–891, Munich, Germany, March 2003.

[GSK97] C. P. Gomes, B. Selman, and H. Kautz. Heavy-tailed distributions in combinatorial search. In Proceedings of National Conference on Artificial Intelligence, pages 431–437, 1997.

[GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. In Proceedings of 11th National Conference on Artificial Intelligence, 1993. ISBN 0262510715.

[HB03] M. Herbstritt and B. Becker. Conflict-based selection of branching rules. In Sixth International Conference on Theory and Application in Satisfiability Testing (SAT2003), pages 441–451, Portofino, Italy, May 2003. Springer. LNCS 2919.

[HJKS09] H. Han, H. Jin, H. Kim, and F. Somenzi. CirCUs 2.0 – SAT competition 2009 edition. SAT Competition 2009 - Solver Description, June 2009.

[HS07] 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.

[HS09] H. Han and F. Somenzi. On-the-fly clause improvement. In Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009), pages 209–222, Swansea, UK, June 2009. Springer-Verlag. LNCS 5584.

[HSJ10] H. Han, F. Somenzi, and H. Jin. Making deduction more effective in SAT solvers. IEEE Transactions on Computer-Aided Design, 29(8):1271–1284, August 2010.

[JAS04] 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.

[Jer] URL: http://www.cs.tau.ac.il/research/alexander.nadel/.

[JS04a] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, Canada, May 2004.

[JS04b] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. Electronic Notes in Theoretical Computer Science, 2004. Second International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[JS06] 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, March 2006.

[KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In Proceedings of the Design Automation Conference, pages 232–237, Las Vegas, NV, June 2001.

[KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In Proceedings of the Design Automation Conference, pages 263–268, Anaheim, CA, June 1997.

[KOSS04] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 308–320. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[Li06] B. Li. Satisfiability-Based Abstraction Refinement in Symbolic Model Checking. PhD thesis, University of Colorado, Department of Electrical and Computer Engineering, 2006.

[Lib00] P. Liberatore. On the complexity of choosing the branching literal in DPLL. Artificial Intelligence, 116(1–2):315–326, 2000.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic for the traveling salesman problem. Operations Research, 21:498–516, 1973.

[LMS04] I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. In International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, Canada, May 2004.

[LS06] B. Li and F. Somenzi. Efficient abstraction refinement in interpolation-based unbounded model checking. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06), pages 227–241, Vienna, Austria, March 2006. LNCS 3920.

[MB89] P. McGeer and R. Brayton. The satisfiability don't care set and invariant transformations in multi-level synthesis. In Proceedings of the IEEE International Conference on Computer Aided Design, November 1989.

[McM02] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, Fourteenth Conference on Computer Aided Verification (CAV'02), pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, Fifteenth Conference on Computer Aided Verification (CAV'03), pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[MMZ+01] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the Design Automation Conference, pages 530–535, Las Vegas, NV, June 2001.

[MS96] J. P. Marques-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, November 1996.

[MS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, 48(5):506–521, 1999.

[MV07] P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In Theory and Applications of Satisfiability Testing: SAT 2007, pages 4–9, Lisbon, Portugal, May 2007. Springer. LNCS 4501.

[Nad09] A. Nadel. Understanding and Improving a Modern SAT Solver. PhD thesis, Tel Aviv University, 2009. (Submitted).

[OMA+04] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In Proceedings of the Design Automation Conference, pages 518–523, San Diego, CA, June 2004.

[Pap94] C. H. Papadimitriou. Computational Complexity. Addison-Wesley, Reading, MA, 1994.

[PD09] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning sat solvers with restarts. In Principles and Practice of Constraint Programming - CP 2009, pages 654–668. Springer-Verlag, 2009. LNCS 5732.

[Pic] URL:http://fmv.jku.at/picosat .

[Pre] URL: http://fmv.jku.at/precosat.

[Pro59] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In IRE-AIEE-ACM '59, eastern joint IRE-AIEE-ACM computer conference, pages 133–138, New York, NY, December 1959. ACM.

[Rsa] URL: http://reasoning.cs.ucla.edu/rsat.

[SATa] URL: http://www.satcompetition.org.

[Satb] URL: http://http://minisat.se/MiniSat.html.

[SATc] URL: http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/.

[SB09] N. Sörensson and A. Biere. Minimizing learned clauses. In Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009), pages 237–243, Swansea, UK, June 2009. Springer-Verlag. LNCS 5584.

[SBV96] P. Stephan, R. K. Brayton, and A. Sangiovanni Vincentelli. Combinational test pattern generation using satisfiability. IEEE Transactions on Computer-Aided Design, 15(9):1167–1176, September 1996.

[SE05] N. Sörensson and N Eén. MiniSat v1.13 – a SAT solver with conflict-clause minimization. SAT Competition 2005 - Solver Description, June 2005.

[Sip96] Michael Sipser. Introduction to the Theory of Computation. International Thomson Publishing, 1996.

[SKC93] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, oct 1993.

[SKC95] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE, pages 521–532, 1995.

[SLM92] Bart Selman, H. J. Levesque, and D. G. Mitchell. New method for solving hard satisfiability problems. In 10th AAAI, pages 440–446, San Jose, CA, July 1992.

[SP04] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pages 276–291, Vancouver, BC, Canada, May 2004. Springer-Verlag. LNCS 3542.

[SS98] M. Sheeran and G. Stålmark. A tutorial on Stålmark's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, Formal Methods in Computer Aided Design, pages 82–99. Springer-Verlag, Palo Alto, CA, November 1998. LNCS 1522.

[Vel04] M. N. Velev. Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessor. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 10266–10271, Paris, France, February 2004.

[VH05] V. C. Vimjam and M. S. Hsiao. Increasing the deducibility in CNF instances for efficient SAT-based bounded model checking. In High-Level Design, Validation, and Test Workshop, pages 184–191, Napa, CA, 2005.

[VIS] URL: http://vlsi.colorado.edu/∼vis.

[WCC09] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Cheng. Electronic Design Automation: Synthesis, Verification, and Test (Systems on Silicon). Morgan Kaufmann Publishers, 2009.

[zCh] URL: http://www.princeton.edu/ chaff.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In Proceedings of the International Conference on Automated Deduction, pages 272–275, July 1997. LNAI 1249.

[Zha05] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), pages 482–489, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[ZKKSV06] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli. SAT sweeping with local observability don't cares. In Proceedings of the Design Automation Conference, pages 229–234, San Francisco, CA, July 2006.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In Design, Automation and Test in Europe (DATE'03), pages 880–885, Munich, Germany, March 2003.

[ZMMM01] 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, November 2001.

# Appendix A

## Tables for Comparison

In this chapter, I list the tables comparing the performance (CPU time) of CirCUs with and without the propsoed techniques (EVAL+OCI) described in Chapter 3.

Table A.1: Comparison of CirCUs with and without the proposed techniques (1)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| aloul-chnl11-13 | UNSAT | 107.54 | **26.66** |
| een-pico-prop01-75 | UNSAT | 6.84 | **1.4** |
| een-pico-prop05-50 | UNSAT | 31.12 | **6** |
| een-tip-sat-nusmv-t5.B | SAT | 5.64 | **1.42** |
| een-tip-sat-nusmv-tt5.B | SAT | 4.58 | **1.56** |
| een-tip-uns-nusmv-t5.B | UNSAT | 1.58 | **1.4** |
| goldb-heqc-alu4mul | UNSAT | **182.62** | 190.88 |
| goldb-heqc-dalumul | UNSAT | 1098.24 | **923.72** |
| goldb-heqc-desmul | UNSAT | 91.84 | **70.86** |
| goldb-heqc-frg2mul | UNSAT | 77.06 | **51.62** |
| goldb-heqc-i10mul | UNSAT | 264.18 | **225.86** |
| goldb-heqc-i8mul | UNSAT | 420.66 | **382.28** |
| goldb-heqc-term1mul | UNSAT | **96.2** | 181.44 |
| grieu-vmpc-s05-25 | SAT | **6.04** | 9.76 |
| grieu-vmpc-s05-27 | SAT | **37.3** | 203.94 |
| grieu-vmpc-s05-28 | SAT | 256.54 | **9.16** |
| grieu-vmpc-s05-34 | SAT | $>$10000 | **1252.86** |
| hoons-vbmc-lucky7 | UNSAT | 2.02 | **0.86** |
| ibm-2002-05r-k90 | SAT | 25.78 | **14.1** |
| ibm-2002-07r-k100 | UNSAT | 2.84 | **1.22** |
| ibm-2002-11r1-k45 | SAT | 190.78 | **76.32** |
| ibm-2002-19r-k100 | SAT | 3297.18 | **512.94** |
| ibm-2002-21r-k95 | SAT | 1359.72 | **340.32** |
| ibm-2002-26r-k45 | UNSAT | 541.46 | **15.16** |
| ibm-2002-27r-k95 | SAT | 103.72 | **15.68** |
| ibm-2004-03-k70 | SAT | 21.08 | **18.34** |
| ibm-2004-04-k100 | SAT | 120.26 | **50.46** |
| ibm-2004-06-k90 | SAT | 192.08 | **45.2** |
| ibm-2004-1_11-k25 | UNSAT | 9.18 | **3.56** |
| ibm-2004-1_31_2-k25 | UNSAT | 29.92 | **8.48** |
| ibm-2004-19-k90 | SAT | 1096.08 | **252.7** |
| ibm-2004-2_02_1-k100 | UNSAT | 17.6 | **7.46** |
| ibm-2004-2_14-k45 | UNSAT | 17.12 | **9.56** |

Table A.2: Comparison of CirCUs with and without the proposed techniques (2)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| ibm-2004-26-k25 | UNSAT | 42.34 | **7.82** |
| ibm-2004-3_02_1-k95 | UNSAT | 2.76 | **2.52** |
| ibm-2004-3_02_3-k95 | SAT | 18.96 | **2.22** |
| ibm-2004-3_11-k60 | UNSAT | 476.72 | **219.28** |
| ibm-2004-6_02_3-k100 | UNSAT | 11.92 | **5.98** |
| manol-pipe-c10id_s | UNSAT | 7.76 | **7.42** |
| manol-pipe-c10nidw_s | UNSAT | 302.94 | **35** |
| manol-pipe-c6nidw_i | UNSAT | 494.34 | **116.54** |
| manol-pipe-c7b | UNSAT | 56.56 | **14.8** |
| manol-pipe-c7b_i | UNSAT | 57.32 | **15.4** |
| manol-pipe-c7bidw_i | UNSAT | 1171.36 | **189.74** |
| manol-pipe-c7nidw | UNSAT | 1493.14 | **194.58** |
| manol-pipe-c9 | UNSAT | 6.38 | **3.26** |
| manol-pipe-c9nidw_s | UNSAT | 159.32 | **28.02** |
| manol-pipe-f10ni | UNSAT | 3215.34 | **1341.6** |
| manol-pipe-f6bi | UNSAT | 4.32 | **3.2** |
| manol-pipe-f7idw | UNSAT | **372.66** | 1708.78 |
| manol-pipe-f9b | UNSAT | 1572.68 | **842.14** |
| manol-pipe-f9n | UNSAT | 1589.34 | **686.14** |
| manol-pipe-g10b | UNSAT | 176.38 | **53.52** |
| manol-pipe-g10bidw | UNSAT | 1527.22 | **305.06** |
| manol-pipe-g10id | UNSAT | 158.54 | **73.52** |
| manol-pipe-g10nid | UNSAT | 954.98 | **298.14** |
| manol-pipe-g6bi | UNSAT | 1.3 | **1.14** |
| manol-pipe-g7nidw | UNSAT | **32.66** | 35.96 |
| maris-s03-gripper11 | SAT | >10000 | **402.48** |
| mizh-md5-47-3 | SAT | **2013.6** | >10000 |
| mizh-md5-47-4 | SAT | >10000 | **4494.5** |
| mizh-md5-47-5 | SAT | >10000 | **7167.48** |
| mizh-md5-48-2 | TIMEOUT | >10000 | >10000 |
| mizh-md5-48-5 | TIMEOUT | >10000 | >10000 |
| mizh-sha0-35-2 | SAT | 5005.52 | **1440.34** |

Table A.3: Comparison of CirCUs with and without the proposed techniques (3)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| mizh-sha0-35-3 | SAT | 2702.42 | **2345.72** |
| mizh-sha0-35-4 | SAT | **977.94** | 8557.52 |
| mizh-sha0-35-5 | SAT | **2407.2** | 3292.08 |
| mizh-sha0-36-2 | SAT | >10000 | **2291.66** |
| narain-vpn-clauses-6 | SAT | **534.4** | 539.56 |
| schup-l2s-guid-1-k56 | UNSAT | 432.06 | **197.48** |
| schup-l2s-motst-2-k315 | SAT | 707.44 | **80.26** |
| simon-s02b-dp11u10 | UNSAT | >10000 | **52.08** |
| simon-s02b-k2f-gr-rcs-w8 | TIMEOUT | >10000 | >10000 |
| simon-s02b-r4b1k1.1 | SAT | >10000 | **470.42** |
| simon-s02-w08-18 | SAT | >10000 | **75.02** |
| simon-s03-fifo8-300 | UNSAT | >10000 | **43.64** |
| simon-s03-fifo8-400 | UNSAT | 456.74 | **158.2** |
| vange-col-abb313GPIA-9-c | SAT | >10000 | >10000 |
| vange-col-inithx.i.1-cn-5 | SAT | >10000 | >10000 |
| velev-engi-uns-1.0-4nd | UNSAT | >10000 | **25.92** |
| velev-engi-uns-1.0-5c1 | UNSAT | >10000 | **12.84** |
| velev-fvp-sat-3.0-b18 | SAT | >10000 | **133.28** |
| velev-live-uns-2.0-ebuf | UNSAT | >10000 | **15.46** |
| velev-npe-1.0-9dlx-b71 | SAT | >10000 | **320.78** |
| velev-pipe-o-uns-1.0-7 | UNSAT | >10000 | **1008.48** |
| velev-pipe-o-uns-1.1-6 | UNSAT | **75** | 91.96 |
| velev-pipe-sat-1.0-b10 | SAT | 549.96 | **381.92** |
| velev-pipe-sat-1.0-b7 | SAT | **77.9** | 846.1 |
| velev-pipe-sat-1.0-b9 | SAT | **137.8** | 583.3 |
| velev-pipe-sat-1.1-b7 | SAT | **108.64** | 608.76 |
| velev-pipe-uns-1.0-8 | UNSAT | >10000 | **1223.2** |
| velev-pipe-uns-1.0-9 | UNSAT | 389.26 | **384** |
| velev-pipe-uns-1.1-7 | UNSAT | 278.06 | **245.42** |
| velev-vliw-sat-2.0-b6 | SAT | 344.66 | **244.9** |
| velev-vliw-sat-4.0-b1 | SAT | **40.76** | 233.62 |
| velev-vliw-sat-4.0-b3 | SAT | **49.6** | 264.86 |

Table A.4: Comparison of CirCUs with and without the proposed techniques (4)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| velev-vliw-sat-4.0-b4 | SAT | **185.58** | 334.44 |
| velev-vliw-uns-2.0-iq4 | UNSAT | >10000 | **4153.84** |
| velev-vliw-uns-4.0-9C1 | UNSAT | >10000 | **931.22** |
| AProVE07-01 | TIMEOUT | >10000 | >10000 |
| AProVE07-02 | UNSAT | 1929.42 | **1191.88** |
| AProVE07-03 | UNSAT | 4980.1 | **2885.86** |
| AProVE07-04 | UNSAT | 307.16 | **192.86** |
| AProVE07-06 | UNSAT | 96.16 | **83.58** |
| AProVE07-08 | UNSAT | 1421.6 | **725.72** |
| AProVE07-09 | UNSAT | 1018.44 | **96.8** |
| AProVE07-11 | SAT | 282.42 | **28.9** |
| AProVE07-15 | UNSAT | 29.96 | **22.58** |
| AProVE07-16 | UNSAT | 279.68 | **277.02** |
| AProVE07-20 | UNSAT | 415.52 | **251.12** |
| AProVE07-21 | UNSAT | **180.8** | 554.44 |
| AProVE07-22 | UNSAT | 79.52 | **41.54** |
| AProVE07-25 | TIMEOUT | >10000 | >10000 |
| AProVE07-26 | TIMEOUT | >10000 | >10000 |
| AProVE07-27 | UNSAT | 4218.22 | **1698.7** |
| blocks-4-ipc5-h21-unknown | UNSAT | 143.82 | **70.76** |
| blocks-4-ipc5-h22-unknown | UNSAT | 237.84 | **86.96** |
| clauses-10 | UNSAT | 96.98 | **88.24** |
| clauses-2 | SAT | **2.06** | 2.3 |
| clauses-4 | SAT | 77.1 | **51.16** |
| clauses-6 | SAT | 541.88 | **540.12** |
| clauses-8 | SAT | 3204.6 | **2061.1** |
| cube-11-h13-unsat | UNSAT | **279.04** | 7915.36 |
| cube-11-h14-sat | SAT | 351.58 | **82.4** |
| cube-9-h10-unsat | UNSAT | 38.62 | **31.52** |
| cube-9-h11-sat | SAT | **144.96** | 206.4 |
| dated-10-11-s | SAT | **4.28** | 20.7 |
| dated-10-11-u | UNSAT | 2228.76 | **1123.12** |

Table A.5: Comparison of CirCUs with and without the proposed techniques (5)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| dated-10-13-s | SAT | **8.06** | 43.32 |
| dated-10-13-u | UNSAT | 1929.4 | **1517.96** |
| dated-10-15-s | SAT | **4.4** | 38.46 |
| dated-10-15-u | UNSAT | **32.2** | 47.18 |
| dated-10-17-s | SAT | **13.32** | 59.12 |
| dated-10-17-u | TIMEOUT | >10000 | >10000 |
| dated-10-19-s | SAT | 13.98 | **10.76** |
| dated-10-19-u | TIMEOUT | >10000 | >10000 |
| dated-5-11-s | SAT | **0.92** | 8.4 |
| dated-5-11-u | UNSAT | 67.76 | **58** |
| dated-5-13-s | SAT | **1.94** | 8.08 |
| dated-5-13-u | TIMEOUT | >10000 | >10000 |
| dated-5-15-s | SAT | **2.56** | 7.24 |
| dated-5-15-u | UNSAT | 303.56 | **241.64** |
| dated-5-17-s | SAT | **3.74** | 9.14 |
| dated-5-17-u | UNSAT | 438.98 | **226.6** |
| dated-5-19-s | SAT | 6.76 | **6.54** |
| dated-5-19-u | TIMEOUT | >10000 | >10000 |
| dspam_dump_vc1080 | UNSAT | 0.74 | **0.72** |
| dspam_dump_vc1081 | UNSAT | 7.64 | **0.7** |
| dspam_dump_vc1093 | UNSAT | **0.34** | 0.6 |
| dspam_dump_vc1103 | UNSAT | 134.74 | **1.68** |
| dspam_dump_vc1104 | UNSAT | 162.18 | **1.66** |
| dspam_dump_vc949 | UNSAT | 1.54 | **0.62** |
| dspam_dump_vc950 | UNSAT | 6.92 | **0.58** |
| dspam_dump_vc962 | UNSAT | 0.72 | **0.56** |
| dspam_dump_vc972 | UNSAT | 18.5 | **1.6** |
| dspam_dump_vc973 | UNSAT | 22.74 | **1.52** |
| emptyroom-4-h21-unsat | UNSAT | 497.28 | **113.98** |
| emptyroom-4-h22-sat | SAT | 44.06 | **16.2** |
| eq.atree.braun.10.unsat | UNSAT | 753.44 | **714.2** |
| eq.atree.braun.11.unsat | UNSAT | >10000 | **7117.42** |

Table A.6: Comparison of CirCUs with and without the proposed techniques (6)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| eq.atree.braun.12.unsat | TIMEOUT | >10000 | >10000 |
| eq.atree.braun.13.unsat | TIMEOUT | >10000 | >10000 |
| eq.atree.braun.7.unsat | UNSAT | 4.58 | **2.64** |
| eq.atree.braun.8.unsat | UNSAT | 22.6 | **19.42** |
| eq.atree.braun.9.unsat | UNSAT | **67.04** | 113.48 |
| hsat_vc11773 | UNSAT | 6.84 | **1.08** |
| hsat_vc11803 | UNSAT | 3.96 | **1.24** |
| hsat_vc11813 | UNSAT | 10.24 | **1.3** |
| hsat_vc11817 | UNSAT | 2.3 | **0.9** |
| hsat_vc11935 | UNSAT | 2.26 | **0.74** |
| hsat_vc11944 | UNSAT | 7.94 | **0.68** |
| hsat_vc12016 | UNSAT | 1.98 | **0.7** |
| hsat_vc12062 | UNSAT | 3.48 | **0.84** |
| hsat_vc12070 | UNSAT | 6.34 | **0.8** |
| hsat_vc12072 | UNSAT | 4.32 | **1.08** |
| itox_vc1033 | SAT | 33.72 | **3.78** |
| itox_vc1044 | SAT | 13.98 | **4.9** |
| itox_vc1130 | SAT | 82.84 | **4.16** |
| itox_vc1138 | SAT | 86.34 | **5.04** |
| itox_vc1216 | UNSAT | **0.28** | 0.54 |
| itox_vc909 | SAT | 6.52 | **3.44** |
| itox_vc965 | UNSAT | **0.2** | 0.28 |
| itox_vc979 | UNSAT | **0.22** | 0.5 |
| partial-10-11-s | SAT | >10000 | **2380.86** |
| partial-10-11-u | TIMEOUT | >10000 | >10000 |
| partial-10-13-s | SAT | 7997.24 | **2388.14** |
| partial-10-13-u | TIMEOUT | >10000 | >10000 |
| partial-10-15-s | SAT | 3164 | **1320.1** |
| partial-10-15-u | TIMEOUT | >10000 | >10000 |
| partial-10-17-s | TIMEOUT | >10000 | >10000 |
| partial-10-17-u | TIMEOUT | >10000 | >10000 |
| partial-10-19-s | TIMEOUT | >10000 | >10000 |

Table A.7: Comparison of CirCUs with and without the proposed techniques (7)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| partial-10-19-u | TIMEOUT | >10000 | >10000 |
| partial-5-11-s | SAT | 312.7 | **164.22** |
| partial-5-11-u | TIMEOUT | >10000 | >10000 |
| partial-5-13-s | SAT | **88** | 440.44 |
| partial-5-13-u | TIMEOUT | >10000 | >10000 |
| partial-5-15-s | SAT | 1017.38 | **459.4** |
| partial-5-15-u | TIMEOUT | >10000 | >10000 |
| partial-5-17-s | TIMEOUT | >10000 | >10000 |
| partial-5-17-u | TIMEOUT | >10000 | >10000 |
| partial-5-19-s | SAT | 3817.64 | **727.3** |
| partial-5-19-u | TIMEOUT | >10000 | >10000 |
| safe-30-h29-unsat | TIMEOUT | >10000 | >10000 |
| safe-30-h30-sat | SAT | >10000 | **58.06** |
| safe-50-h49-unsat | TIMEOUT | >10000 | >10000 |
| safe-50-h50-sat | TIMEOUT | >10000 | >10000 |
| sortnet-6-ipc5-h11-unsat | UNSAT | 3289.32 | **1381.8** |
| sortnet-7-ipc5-h15-unsat | TIMEOUT | >10000 | >10000 |
| sortnet-7-ipc5-h16-sat | SAT | 1364.28 | **113.64** |
| sortnet-8-ipc5-h18-unsat | TIMEOUT | >10000 | >10000 |
| sortnet-8-ipc5-h19-sat | TIMEOUT | >10000 | >10000 |
| total-10-11-s | SAT | **8.64** | 42.42 |
| total-10-11-u | UNSAT | 111.3 | **96.88** |
| total-10-13-s | SAT | **13.14** | 68.94 |
| total-10-13-u | UNSAT | **702.22** | 736.92 |
| total-10-15-s | SAT | 110.78 | **9.48** |
| total-10-15-u | TIMEOUT | >10000 | >10000 |
| total-10-17-s | SAT | 18.84 | **10.9** |
| total-10-17-u | TIMEOUT | >10000 | >10000 |
| total-10-19-s | SAT | **13.7** | 14.7 |
| total-10-19-u | TIMEOUT | >10000 | >10000 |
| total-5-11-s | SAT | **3.94** | 16.2 |
| total-5-11-u | UNSAT | 24.56 | **18.88** |

Table A.8: Comparison of CirCUs with and without the proposed techniques (8)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| total-5-13-s | SAT | **1.74** | 13 |
| total-5-13-u | UNSAT | 42.52 | **20.7** |
| total-5-15-s | SAT | **5.16** | 6.54 |
| total-5-15-u | TIMEOUT | >10000 | >10000 |
| total-5-17-s | SAT | 16.18 | **14.14** |
| total-5-17-u | TIMEOUT | >10000 | >10000 |
| total-5-19-s | SAT | **14.18** | 17.26 |
| total-5-19-u | TIMEOUT | >10000 | >10000 |
| uts-l05-ipc5-h26-unsat | UNSAT | 54.2 | **54.14** |
| uts-l05-ipc5-h27-unknown | UNSAT | 83.56 | **61.06** |
| uts-l06-ipc5-h28-unknown | UNSAT | **54.7** | 205.16 |
| uts-l06-ipc5-h29-unknown | UNSAT | **88.42** | 198.8 |
| uts-l06-ipc5-h30-unknown | UNSAT | 225.32 | **217.8** |
| uts-l06-ipc5-h31-unknown | UNSAT | **181.86** | 244.4 |
| uts-l06-ipc5-h32-unknown | UNSAT | 335.5 | **269.94** |
| uts-l06-ipc5-h33-unknown | UNSAT | 586.14 | **286.86** |
| uts-l06-ipc5-h34-unknown | SAT | **153.36** | 222.48 |
| uts-l06-ipc5-h35-unknown | SAT | **120.3** | 219.66 |
| vmpc_24 | SAT | 3.26 | **3.1** |
| vmpc_26 | SAT | **19.9** | 49.68 |
| vmpc_29 | SAT | **706.62** | 727.32 |
| vmpc_30 | SAT | 3561.68 | **161.46** |
| vmpc_31 | SAT | >10000 | **78.54** |
| vmpc_33 | SAT | 1225.22 | **559** |
| xinetd_vc56687 | UNSAT | 0.22 | 0.22 |
| xinetd_vc56703 | UNSAT | **0.2** | 0.22 |
| anbul-dated-5-15-u | UNSAT | 295.9 | **247.32** |
| anbul-part-10-13-s | SAT | 7912.96 | **2442.8** |
| anbul-part-10-15-s | SAT | 3131 | **1318.94** |
| babic-dspam-vc1080 | UNSAT | **0.66** | 0.68 |
| babic-dspam-vc949 | UNSAT | 1.56 | **0.62** |
| babic-dspam-vc973 | UNSAT | 22.64 | **1.6** |

Table A.9: Comparison of CirCUs with and without the proposed techniques (9)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| cmu-bmc-barrel6 | UNSAT | 3.06 | **1.3** |
| cmu-bmc-longmult13 | UNSAT | 57.14 | **29.74** |
| cmu-bmc-longmult15 | UNSAT | 28.46 | **19.16** |
| een-pico-prop00-75 | UNSAT | 15.56 | **2.2** |
| een-pico-prop05-75 | UNSAT | 107.86 | **17.5** |
| een-tip-sat-texas-tp-5e | SAT | 0.36 | **0.1** |
| een-tip-sat-vis-eisen | SAT | 0.96 | **0.34** |
| fuhs-aprove-15 | UNSAT | 30.56 | **22.58** |
| fuhs-aprove-16 | UNSAT | **279.16** | 279.96 |
| goldb-heqc-x1mul | UNSAT | >10000 | **4802.72** |
| grieu-vmpc-31 | SAT | >10000 | **77.62** |
| ibm-2002-04r-k80 | SAT | 76.6 | **64.4** |
| ibm-2002-18r-k90 | SAT | 6373.16 | **1495.72** |
| ibm-2002-20r-k75 | SAT | 1501.88 | **442.06** |
| ibm-2002-22r-k60 | UNSAT | 551.62 | **310.52** |
| ibm-2002-22r-k75 | SAT | 1942.66 | **471.54** |
| ibm-2002-22r-k80 | SAT | 3743.74 | **840.5** |
| ibm-2002-23r-k90 | SAT | >10000 | **3295.52** |
| ibm-2002-24r3-k100 | UNSAT | 328.74 | **224.26** |
| ibm-2002-25r-k10 | UNSAT | 1047.24 | **622.24** |
| ibm-2002-29r-k75 | SAT | 169.78 | **50.38** |
| ibm-2002-30r-k85 | SAT | 8006.54 | **928.24** |
| ibm-2002-31_1r3-k30 | UNSAT | 606.86 | **139.38** |
| ibm-2004-1_11-k80 | SAT | 4951.88 | **904.36** |
| ibm-2004-23-k100 | SAT | >10000 | **6483.76** |
| ibm-2004-23-k80 | SAT | 8094.54 | **1347.12** |
| ibm-2004-29-k25 | UNSAT | 114.06 | **69.9** |
| ibm-2004-29-k55 | SAT | 139.72 | **96.74** |
| jarvi-eq-atree-9 | UNSAT | **65.66** | 112.92 |
| manol-pipe-c10nid_i | UNSAT | 9448.12 | **1510.68** |
| manol-pipe-c10nidw | UNSAT | >10000 | **2906.32** |
| manol-pipe-c6bidw_i | UNSAT | 565.58 | **105.16** |

Table A.10: Comparison of CirCUs with and without the proposed techniques (10)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| manol-pipe-c8nidw | UNSAT | 5065.12 | **832.02** |
| manol-pipe-c9n_i | UNSAT | 106.18 | **29.42** |
| manol-pipe-f7nidw | UNSAT | 538.14 | **273.72** |
| manol-pipe-g10bid_i | UNSAT | 7771.76 | **1357.74** |
| manol-pipe-g8nidw | UNSAT | 123.36 | **49.7** |
| marijn-philips | UNSAT | 8205.62 | **7336.88** |
| mizh-sha0-36-1 | TIMEOUT | >10000 | >10000 |
| mizh-sha0-36-3 | TIMEOUT | >10000 | >10000 |
| mizh-sha0-36-4 | TIMEOUT | >10000 | >10000 |
| narain-vpn-clauses-8 | SAT | 3171.04 | **2037.84** |
| palac-sn7-ipc5-h16 | SAT | 1347.06 | **114.74** |
| palac-uts-l06-ipc5-h34 | SAT | **149.34** | 222.42 |
| post-c32s-col400-16 | UNSAT | 1323.36 | **286.06** |
| post-c32s-gcdm16-22 | SAT | 896.52 | **175** |
| post-c32s-gcdm16-23 | UNSAT | 920.8 | **226.8** |
| post-c32s-ss-8 | UNSAT | 4147.44 | **974.2** |
| post-cbmc-aes-d-r1 | UNSAT | 5.92 | **5.84** |
| post-cbmc-aes-d-r2 | UNSAT | 1324.42 | **633.72** |
| post-cbmc-aes-ee-r2 | UNSAT | 1476.32 | **459.72** |
| post-cbmc-aes-ee-r3 | UNSAT | >10000 | **2449.82** |
| post-cbmc-aes-ele | UNSAT | **20.5** | 42.54 |
| post-cbmc-zfcp-2.8-u2 | SAT | **29.96** | 58.08 |
| schup-l2s-abp4-1-k31 | UNSAT | 28.54 | **21.98** |
| schup-l2s-bc56s-1-k391 | UNSAT | 1228.64 | **877.52** |
| simon-s02b-r4b1k1.2 | SAT | **104.5** | 142.5 |
| simon-s02-f2clk-50 | UNSAT | 418.84 | **122.52** |
| simon-s03-w08-15 | SAT | 308.6 | **82.32** |
| velev-vliw-sat-4.0-b8 | SAT | **50.76** | 132.62 |
| velev-vliw-uns-2.0-iq1 | UNSAT | 200.7 | **172.7** |
| velev-vliw-uns-2.0-iq2 | UNSAT | 926.14 | **685.8** |
| velev-vliw-uns-2.0-uq5 | UNSAT | **8976.8** | 9081.2 |
| velev-vliw-uns-4.0-9 | UNSAT | 1336.58 | **931.66** |

Table A.11: Comparison of CirCUs with and without the proposed techniques (11)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| velev-vliw-uns-4.0-9-i1 | UNSAT | 8759.8 | **7155.42** |
| ACG-10-5p0 | UNSAT | **18.62** | 87.76 |
| ACG-15-10p0 | UNSAT | 2637.58 | **1193.12** |
| ACG-15-10p1 | SAT | **2233.3** | 2266.78 |
| ACG-20-10p0 | UNSAT | 7103.24 | **4798.98** |
| ACG-20-10p1 | SAT | 7387.36 | **3382.08** |
| ACG-20-5p1 | SAT | 1312.38 | **1138.84** |
| AProVE09-01 | SAT | 1.42 | **1.16** |
| AProVE09-03 | SAT | 3.7 | **1.14** |
| AProVE09-05 | SAT | **0.94** | 1.56 |
| AProVE09-06 | SAT | 2227.66 | **1309.76** |
| AProVE09-07 | SAT | 1.78 | **0.76** |
| AProVE09-08 | SAT | **1.42** | 2.26 |
| AProVE09-10 | SAT | **3.32** | 57.52 |
| AProVE09-11 | SAT | **0.16** | 1.78 |
| AProVE09-12 | SAT | **0.44** | 1.9 |
| AProVE09-13 | SAT | **0.04** | 0.3 |
| AProVE09-15 | SAT | **5.22** | 21.3 |
| AProVE09-17 | SAT | 30.5 | **15.14** |
| AProVE09-19 | SAT | **0.48** | 2.58 |
| AProVE09-20 | SAT | 1339.74 | **531.84** |
| AProVE09-21 | SAT | 21.5 | **2.8** |
| AProVE09-22 | SAT | **0.04** | 0.28 |
| AProVE09-24 | SAT | 18.6 | **5.18** |
| AProVE09-25 | SAT | **0.16** | 1.58 |
| countbitsarray02_32 | UNSAT | **658.94** | 3002.5 |
| countbitsarray08_32 | TIMEOUT | >10000 | >10000 |
| countbitsarray32_32 | TIMEOUT | >10000 | >10000 |
| countbitsrotate016 | UNSAT | 52.58 | **44.18** |
| countbitsrotate032 | TIMEOUT | >10000 | >10000 |
| countbitsrotate128 | TIMEOUT | >10000 | >10000 |
| countbitssrl016 | UNSAT | 8.64 | **6.68** |

Table A.12: Comparison of CirCUs with and without the proposed techniques (12)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| countbitssrl032 | UNSAT | 6800.36 | **5359.78** |
| countbitssrl128 | TIMEOUT | >10000 | >10000 |
| countbitswegner064 | UNSAT | 302.34 | **116.38** |
| gss-13-s100 | SAT | 78.8 | **51.56** |
| gss-14-s100 | SAT | 150.58 | **50.36** |
| gss-15-s100 | SAT | 458.52 | **226.74** |
| gss-16-s100 | SAT | 828.14 | **336.2** |
| gss-17-s100 | SAT | 1339.48 | **505.88** |
| gss-19-s100 | SAT | >10000 | **2227.12** |
| gss-20-s100 | TIMEOUT | >10000 | >10000 |
| gss-21-s100 | TIMEOUT | >10000 | >10000 |
| gss-22-s100 | TIMEOUT | >10000 | >10000 |
| gss-23-s100 | TIMEOUT | >10000 | >10000 |
| gss-24-s100 | TIMEOUT | >10000 | >10000 |
| gss-25-s100 | TIMEOUT | >10000 | >10000 |
| gss-26-s100 | TIMEOUT | >10000 | >10000 |
| gss-27-s100 | TIMEOUT | >10000 | >10000 |
| gss-28-s100 | TIMEOUT | >10000 | >10000 |
| gss-31-s100 | TIMEOUT | >10000 | >10000 |
| gss-32-s100 | TIMEOUT | >10000 | >10000 |
| gss-33-s100 | TIMEOUT | >10000 | >10000 |
| gss-34-s100 | TIMEOUT | >10000 | >10000 |
| gus-md5-04 | UNSAT | 5.98 | **5.9** |
| gus-md5-05 | UNSAT | **15.42** | 17.08 |
| gus-md5-06 | UNSAT | 57.8 | **38.58** |
| gus-md5-07 | UNSAT | 157.06 | **111.38** |
| gus-md5-09 | UNSAT | 3020.8 | **1709.26** |
| gus-md5-10 | UNSAT | 6981.82 | **4004.86** |
| gus-md5-11 | TIMEOUT | >10000 | >10000 |
| gus-md5-14 | TIMEOUT | >10000 | >10000 |
| gus-md5-15 | TIMEOUT | >10000 | >10000 |
| gus-md5-16 | TIMEOUT | >10000 | >10000 |

Table A.13: Comparison of CirCUs with and without the proposed techniques (13)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| icbrt1_32 | UNSAT | 55.92 | **27.76** |
| maxand064 | UNSAT | 8.48 | **4.24** |
| maxor128 | UNSAT | **5766.14** | 6162.66 |
| maxxor064 | UNSAT | >10000 | **6910.84** |
| maxxor128 | TIMEOUT | >10000 | >10000 |
| maxxororand032 | UNSAT | 313.5 | **263.86** |
| maxxororand128 | TIMEOUT | >10000 | >10000 |
| minand128 | UNSAT | 26.7 | **11.44** |
| minandmaxor032 | UNSAT | 10.02 | **4.74** |
| minandmaxor128 | UNSAT | 4573.48 | **3228.66** |
| minor032 | UNSAT | 0.64 | **0.42** |
| minor064 | UNSAT | 7.22 | **2.88** |
| minxor128 | UNSAT | **138.94** | 303.16 |
| minxorminand032 | UNSAT | **2.62** | 5.34 |
| minxorminand064 | UNSAT | **47.94** | 92.36 |
| minxorminand128 | UNSAT | **1089.9** | 2152.4 |
| mulhs016 | TIMEOUT | >10000 | >10000 |
| mulhs032 | TIMEOUT | >10000 | >10000 |
| ndhf_xits_09_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_10_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_11_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_12_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_13_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_14_UNSAT | TIMEOUT | >10000 | >10000 |
| ndhf_xits_15_UNKNOWN | TIMEOUT | >10000 | >10000 |
| ndhf_xits_16_UNKNOWN | TIMEOUT | >10000 | >10000 |
| ndhf_xits_17_UNKNOWN | TIMEOUT | >10000 | >10000 |
| ndhf_xits_20_SAT | SAT | 694.26 | **592.22** |
| ndhf_xits_21_SAT | SAT | **7.66** | 13.76 |
| ndhf_xits_22_SAT | SAT | **0.28** | 11.18 |
| post-cbmc-aes-d-r2-nohole | UNSAT | 1504.94 | **749.7** |
| post-cbmc-aes-ee-r2-nohol | UNSAT | 1408.16 | **448.52** |

Table A.14: Comparison of CirCUs with and without the proposed techniques (14)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| q_query_2_L324_coli | UNSAT | 17.3 | **10.24** |
| q_query_3_L100_coli.s | UNSAT | 296.42 | **208.9** |
| q_query_3_L150_coli.s | UNSAT | 650.38 | **140.72** |
| q_query_3_L200_coli.s | UNSAT | 473.28 | **404.44** |
| q_query_3_l37_lambda | SAT | 11.2 | **4.2** |
| q_query_3_l38_lambda | SAT | 17.64 | **4.54** |
| q_query_3_l39_lambda | SAT | 20.26 | **10.34** |
| q_query_3_l40_lambda | SAT | 27.18 | **17.2** |
| q_query_3_l41_lambda | SAT | 41.52 | **11.46** |
| q_query_3_l42_lambda | SAT | 81.56 | **29.46** |
| q_query_3_l43_lambda | SAT | 62.32 | **46.98** |
| q_query_3_l44_lambda | UNSAT | 276.28 | **227.8** |
| q_query_3_l45_lambda | UNSAT | 320.38 | **206** |
| q_query_3_l46_lambda | UNSAT | 300.9 | **212.5** |
| q_query_3_l47_lambda | UNSAT | 315.02 | **220.64** |
| q_query_3_l48_lambda | UNSAT | 275.88 | **213.78** |
| q_query_3_L60_coli.sa | SAT | 158.96 | **84.18** |
| q_query_3_L70_coli.sa | SAT | 163.6 | **108.96** |
| q_query_3_L80_coli.sa | UNSAT | 190.56 | **92.86** |
| q_query_3_L90_coli.sa | UNSAT | **218.02** | 252.76 |
| rbcl_xits_06_UNSAT | UNSAT | **12.18** | 12.2 |
| rbcl_xits_07_UNSAT | UNSAT | **170.22** | 297.56 |
| rbcl_xits_08_UNSAT | TIMEOUT | >10000 | >10000 |
| rbcl_xits_09_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rbcl_xits_10_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rbcl_xits_11_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rbcl_xits_12_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rbcl_xits_13_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rbcl_xits_14_SAT | SAT | **7.68** | 20.36 |
| rpoc_xits_07_UNSAT | UNSAT | **124.2** | 233.84 |
| rpoc_xits_08_UNSAT | UNSAT | 5727.9 | **3707** |
| rpoc_xits_09_UNSAT | TIMEOUT | >10000 | >10000 |
| rpoc_xits_10_UNKNOWN | TIMEOUT | >10000 | >10000 |

Table A.15: Comparison of CirCUs with and without the proposed techniques (15)

| Design | Answer | CirCUs | CirCUs+EVAL+OCI |
|---|---|---|---|
| rpoc_xits_11_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rpoc_xits_12_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rpoc_xits_13_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rpoc_xits_14_UNKNOWN | TIMEOUT | >10000 | >10000 |
| rpoc_xits_17_SAT | SAT | **0.14** | 2.68 |
| smulo016 | UNSAT | 24.58 | **9.9** |
| smulo128 | TIMEOUT | >10000 | >10000 |
| UCG-10-5p0 | UNSAT | **35.56** | 55.6 |
| UCG-15-10p0 | UNSAT | 1415.64 | **908.14** |
| UCG-15-10p1 | SAT | 2312.28 | **998.04** |
| UCG-15-5p0 | UNSAT | 126.06 | **99.88** |
| UCG-20-10p1 | SAT | 4960.12 | **2954.5** |
| UCG-20-5p1 | SAT | 890.92 | **406.3** |
| UR-10-5p0 | UNSAT | **34.1** | 68.28 |
| UR-10-5p1 | SAT | **20** | 63.14 |
| UR-15-10p0 | UNSAT | 1933.16 | **1053.38** |
| UR-15-10p1 | SAT | 3148.88 | **1105.78** |
| UR-15-5p0 | UNSAT | 300.5 | **136.56** |
| UR-20-10p1 | SAT | >10000 | **4198.78** |
| UR-20-5p0 | UNSAT | 2605.52 | **2065.18** |
| UR-20-5p1 | SAT | 3072.84 | **2157.48** |
| UTI-10-10p0 | UNSAT | 159.8 | **135.32** |
| UTI-15-10p0 | UNSAT | 651.32 | **290.42** |
| UTI-15-10p1 | SAT | 1346.42 | **505.82** |
| UTI-15-5p0 | UNSAT | 1091.08 | **843.02** |
| UTI-15-5p1 | SAT | 957.68 | **646.26** |
| UTI-20-10p0 | UNSAT | 6601.34 | **2212.62** |
| UTI-20-10p1 | SAT | >10000 | **7104.92** |
| UTI-20-5p0 | UNSAT | 7187.76 | **4939.72** |
| UTI-20-5p1 | SAT | 6451.24 | **2431.54** |