Design, implementation and evaluation of a distributed CDCL framework

Inaugural-Dissertation

zur

Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Universität zu Köln

vorgelegt von

Alexander van der Grinten

aus Bergisch Gladbach

Berichterstatter:	Prof. Dr. Ewald Speckenmeyer
(Gutachter)	Prof. Dr. Henning Meyerhenke

Tag der mündlichen Prüfung: 16.01.2018

Köln, 2018

Zusammenfassung

Im Kern beschäftigt sich diese Dissertation mit dem praktischen Lösen des Erfüllbarkeitsproblems der Aussagenlogik (SAT). Die Probleminstanzen, um die es dabei geht, stammen aus industriellen Anwendungen. Die Entdeckung des Conflict-Driven Clause-Learning (CDCL) Algorithmus führte zu enormen Fortschritten in diesem Bereich. CDCL wurde durch effektive Pre- und Inprocessingtechniken erweitert, welche die Leistungsfähigkeit des Algorithmus erhöhen. Während in der Vergangenheit viel Forschungsarbeit in den Einsatz von shared-memory Parallelität zum Beschleunigen von CDCL geflossen ist, blieb das Lösen von SAT auf verteilten Rechnern weniger gut erforscht.

In dieser Arbeit entwickeln wir ein verteiltes, auf CDCL basierendes Framework, um SAT zu Lösen. Dieses Framework besteht hauptsächlich aus drei Komponenten: **1.** Einer Implementierung des CDCL-Verfahrens, die wir von Grund auf neu geschrieben haben, **2.** einem neuartigem Algorithmus um SAT mit Hilfe von Parallelität zu lösen, und **3.** einer Ansammlung von parallelen Vereinfachungstechniken für SAT-Instanzen. Das entstandene Framework nennen wir satUZK, während der parallele Lösungsalgorithmus der Distributed Divide-and-Conquer (DDC) Algorithmus ist.

DDC verwendet eine parallele Lookahead-Prozedur, um den Suchraum dynamisch zu zerteilen. Dabei wird Load Balancing genutzt um sicherzustellen, dass alle zur Verfügung stehenden Resourcen des verteilen Rechners ausgenutzt werden. Diese Prozedur erstellt einen Divide-and-Conquer Baum, der über alle Prozessoren verteilt wird. Individuelle Threads werden durch diesen Baum geleitet, bis sie ein ungelöstes Blatt erreichen. Bei Ankunft an einem Blatt wird entweder die Lookahead-Prozedur erneut aufgerufen oder das Blatt wird durch CDCL gelöst. Wir schlagen mehrere Erweiterungen für diesen Algorithmus vor. Insbesondere integrieren wir eine Strategie um Klauseln zwischen Threads austauschen und ein Schema, um den sogenannten LBD-Wert von Klauseln relativ zur lokalen Position im Suchbaum anzupassen. LBD ist ein Maß für die Nützlichkeit einer Klausel während der CDCL-Suche. Wir evaluieren unseren Algorithmus empirisch und messen ihn an den besten, verteilen SAT Algorithmen. In diesem Experiment ist unser Algorithmus schneller als andere, verteilte Solver und löst mindestens ebenso viele Instanzen.

Zusätzlich zu dem parallelen Lösungsalgorithmus betrachten wir auch parallele Vereinfachungstechniken. Dabei entwickeln wir zunächst eine theoretische Grundlage, die es uns erlaubt, die Korrektheit von parallelen Vereinfachungstechniken nachzuweisen. Auf dieser Basis untersuchen wir etablierte Vereinfachungstechniken auf ihre Parallelisierbarkeit. Es stellt sich heraus, dass einige dieser Techniken sehr gut parallelisierbar sind. Für diese Techniken stellen wir parallele Implementierungen bereit, die wir empirisch auf ihre Effektivität hin untersuchen. Als Ergebnis dieser Untersuchung identifizieren wir mehrere Techniken, die es erlauben, Instanzen zu Lösen, die der DDC Algorithmus alleine nicht lösen kann.

Abstract

The primary subject of this dissertation is practically solving instances of the Boolean satisfiability problem (SAT) that arise from industrial applications. The invention of the conflict-driven clause-learning (CDCL) algorithm led to enormous progress in this field. CDCL has been augmented with effective pre- and inprocessing techniques that boost its effectiveness. While a considerable amount of work has been done on applying shared-memory parallelism to enhance the performance of CDCL, solving SAT on distributed architectures is studied less thoroughly.

In this work, we develop a distributed, CDCL-based framework for SAT solving. This framework consists of three main components: **1.** An implementation of the CDCL algorithm that we have written from scratch, **2.** a novel, parallel SAT algorithm that builds upon this CDCL implementation and **3.** a collection of parallel simplification techniques for SAT instances. We call our resulting framework satUZK; our parallel solving algorithm is called the distributed divide-and-conquer (DDC) algorithm.

The DDC algorithm employs a parallel lookahead procedure to dynamically partition the search space. Load balancing is used to ensure that all computational resources are utilized during lookahead. This procedure results in a divide-and-conquer tree that is distributed over all processors. Individual threads are routed through this tree until they arrive at unsolved leaf vertices. Upon arrival, the lookahead procedure is invoked again or the leaf vertex is solved via CDCL. Several extensions to the DDC algorithm are proposed. These include clause sharing and a scheme to locally adjust the LBD score relative to the current search tree vertex. LBD is a measure for the usefulness of clauses that participate in a CDCL search. We evaluate our DDC algorithm empirically and benchmark it against the best distributed SAT algorithms. In this experiment, our DDC algorithm is faster than other distributed, state-of-the-art solvers and solves at least as many instances.

In addition to running a parallel algorithm for SAT solving we also consider parallel simplifcation. Here, we first develop a theoretical foundation that allows us to prove the correctness of parallel simplification techniques. Using this as a basis, we examine established simplification algorithms for their parallelizability. It turns out that several well-known simplification techniques can be parallelized efficently. We provide parallel implementation of the techniques and test their effectiveness in empirical experiments. This evaluation finds several combinations of simplification techniques that can solve instances which could not be solved by the DDC algorithm alone.

Contents

Chapter 1

Introduction

Satisfiability is far from an abstract exercise in understanding formal systems. Revolutionary methods for solving such problems emerged at the beginning of the twenty-first century, and they've led to game-changing applications in industry. These so-called "SAT solvers" can now routinely find solutions to practical problems that involve millions of variables and were thought until very recently to be hopelessly difficult.

From the back cover of *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability* by Donald Knuth [60]

Boolean satisfiability (SAT) is the problem of determining whether a given formula over Boolean variables has a solution. In addition to its importance as the canonical NP-complete problem [28], SAT solvers are often used to practically solve hard industrial problems. Those problems include verification of hardware and software via bounded model checking [21, 55], debugging via symbolic execution [26] and AI planning [74]. Other applications of SAT solving include hard combinatorial problems. For example, such problems arise from cryptography [80] or from the encoding of NP-complete problems.

In addition to finding solutions, SAT solvers are also able to prove the non-existance of solutions even for very hard problems. In the recent years, two long-standing mathematical problems have been solved by the use of SAT solvers, namely the Erdős discrepancy conjecture [61] and the Pythagorean triple problem [50]. The SAT-based proof of the latter problem has a size of over 200 terabytes and took four years of CPU time to be produced.

The practical use of SAT-based tools is made feasible by the tremendous progress [79, 69, 99, 70, 33, 81, 8] that has been made on SAT solving technologies after the invention of the so called *conflict-driven clause-learning* (CDCL) method in 1996. CDCL is an algorithm for SAT solving that performs exceptionally well on large industrial and hard combinatorial instances. Advances include specialized data structures [69], better heuristics [69, 99, 70, 8] and extensions to the basic CDCL algorithm [69, 81]. These extensions include the use of sophisticated *pre-* and *inprocessing* techniques [33, 42, 45, 47, 46] that simplify the Boolean formula before or while a solver searches for a solution.

With the ubiquitous availability of multicore processors and clusters of computers, SAT solvers that utilize parallelism have been developed. These solvers follow multiple different approaches: *Portfolio* solvers [40, 16, 11, 4, 10] use multiple cooperating SAT engines to solve the same problem while *search space partitioning* solvers [24, 51, 3, 7] try to accelerate the search by solving different parts of the problem in parallel.

The SAT Competition [78] measures the continuing progress of SAT solving technologies. It features both industrial and hard combinatorial problems and evaluates sequential as well as parallel, shared-memory solvers. While some SAT solvers for distributed architectures have been proposed [24, 11, 3, 7], most state-of-the-art solvers only support shared-memory [16, 17, 4, 10]. Furthermore, those solvers continue to perform pre- and inprocessing only sequentially, even if the solving procedure itself is parallelized. The challenge of developing parallel simplification techniques is explicitly mentioned in [41].

In this dissertation, we develop a framework that supports distributed SAT solving. We call this framework *satUZK*. satUZK consists of multiple components: A sequential CDCL implementation that has been written from scratch, a novel distributed SAT solving algorithm that builds upon this CDCL implementation and a set of parallel simplifcation techniques that can be applied as pre- and inprocessing. While distributed solving algorithms have been published in the past and parallel simplification techniques have also been proposed, this work constitutes the first framework that utilizes parallelism at all stages of the solving process.

Our contribution can be summarized as follows:

- We have written a competitive implementation of the CDCL algorithm from scratch. This implementation integrates all significant advances that have been made on the CDCL core algorithm in the last 20 years. In constrast to existing solvers our implementation is written in a modern C++ style that combines modularity and extensibility with high performance.

- We present a novel parallel SAT algorithm that runs on distributed architectures. This algorithm has also been implemented in C++. We demonstrate the scalability of our algorithm and evaluate our implementation against the state of the art in parallel SAT solvering.

In this evaluation, our algorithm compares favorably against the best distributed SAT solvers. We propose multiple extensions to our distributed algorithm that further boost its performance. For each of those extensions, we present benchmarks of their performance.

- We work towards a theory of parallel simplification techniques and present methods that can prove the correctness of such techniques. Furthermore, we describe parallelizations of many common simplification techniques which we have also implemented in C++. In particular, we present a distributed algorithm for *distillation*, which is one of the most powerful simplification techniques that is applied in SAT solvers. We evaluate the scalability and effectivness of these parallel techniques.

This dissertation is organized as follows: In chapter 2 we review basic notions related to Boolean satisfiability. This chapter prepares the required preliminaries and the basics that the CDCL algorithm builds upon. Chapter 3 presents our implementation of the CDCL algorithm. We first introduce the CDCL method and summarize the algorithms and data structures that are required to implement it. After that we describe details of our CDCL implementation in satUZK and evaluate the performance of this implementation. In chapter 4 we present our novel distributed algorithm for SAT solving. We discuss extensions and implementation details of this algorithm and assess its scalability and performance empirically. Chapter 5 deals with the theory, implementation and parallelization of simplification techniques. We evaluate the scalability and effectiveness of those techniques. Finally, we conclude by summarizing the results of this dissertation.

Chapter 2

Preliminaries

In this chapter, we will introduce basic concepts that are required to present our work on SAT solvers in later chapters. These concepts are widely-known in the literature.

2.1 Prerequisites

In this section, we state some concepts that we assume the reader is familiar with.

Complexity theory The classes P and NP are well-known complexity classes of decision problems. *co*NP is the complement of NP, or in other words, *co*NP is the class of decision problems where certificates for "no" answers can be verified in polynomial time. We further assume that the reader is accustomed to the notion of NP-hardness and NP-completeness.

Graphs We assume that the reader is familiar with graphs and with the usual terminology related to graphs. We denote the vertex set of a graph G by $V(G)$ and its set of edges by $E(G)$.

Propositional logic While we introduce most terms of propositional logic that are related to SAT solving in the next section, we expect the reader to have a basic understanding of propositional logic.

2.2 The SAT problem

In this section, we will define the SAT problem and state some basic definitions and propositions related to SAT.

2.2.1 Boolean formulas

We start by defining what a Boolean formula is.

Definition 2.1. *A* Boolean variable *is a variable that assumes a value in the set* $\{\mathsf{true}, \mathsf{false}\}$*.* Boolean formulas *are formulas that are constructed from a finite number of Boolean variables and some set of junctors. In this thesis, variables are generally denoted by* x*,* y*,* z *and* w*. Formulas are denoted by* ϕ *or* ψ*.*

We will usually restrict the set of junctors in definition 2.1 to $\wedge, \vee$ and $\neg$ (corresponding to the Boolean "and", "or" and "not" functions). Other junctors such as $\rightarrow$, $\leftrightarrow$ or $\oplus$ (corresponding to implication, equivalence and "xor") can be constructed from $\wedge, \vee$ and $\neg$. In order to conserve space, instead of writing $\neg\alpha$ for some term α, we often write $\bar{\alpha}$ instead.

Definition 2.2. *For each Boolean variable* x*, the two terms* x *and* $\bar{x}$ *are called* literals*;* x *is called* positive *and* $\bar{x}$ *is called* negative*. We often denote literals by* a*,* b *or* c*.*

An assignment *(sometimes also called* partial *assignment) is a set of literals. An assignment is called* total assignment *with respect to a Boolean formula* ϕ *if it contains at least one literal for each variable that occurs in* ϕ*. A partial assignment* τ *is called* conflicting *if there is a variable* x *so that* $\{x, \bar{x}\} \subseteq \tau$*.*

Let τ be an assignment. If τ is non-conflicting, we can think of τ as a function that assigns a value from $\{\mathsf{true}, \mathsf{false}, \mathsf{unknown}\}$ to each Boolean variable via

$$\tau(x) = \begin{cases} \mathsf{true} & x \in \tau \\ \mathsf{false} & \bar{x} \in \tau \\ \mathsf{unknown} & \text{otherwise} \end{cases}$$

This function can be naturally extended to arbitrary Boolean terms, using the usual semantics for the junctors. This enables us to state the following definition:

Definition 2.3. *Let* ϕ *be a Boolean formula. If* $\tau(\phi) = \mathsf{true}$*, then* τ *is called a* model *of* ϕ*.*

The concept of a model is central to this thesis, this will become clear in the next subsection.

2.2.2 Satisfiability

Given the definitions from the last section we can now state the SAT problem.

Definition 2.4. *The* satisfiability problem of propositional logic (SAT) *asks if there is a model for a given Boolean formula ϕ. In that case ϕ is called* satisfiable*.*

It is well known that SAT is NP-complete; specifically SAT is the first problem that was proven to be NP-complete by Cook [28]. Therefore, many NP-completeness proofs use reductions from SAT.

A *SAT solver* is an algorithm that solves the SAT problem. We usually demand that a SAT solver can output a model for "yes" answers and sometimes even demand that it can output an unsatisfiability proof for "no" answers. In this thesis, we only consider SAT solvers that are of practical relevance. We are not interested in algorithms that are used to prove lower bounds on the complexity of SAT but that are inefficient in practice.

If a SAT solver eventually terminates, we call the solver *complete*. We will only consider complete solvers here. For some classes of problems, for example random formulas, *incomplete* algorithms that use randomized methods to search for a solution are very efficient in practice. However, those methods cannot prove unsatisfiability and do not terminate on unsatisfiable formulas.

2.2.3 Notions of equivalence

SAT solvers often transform their input formulas while solving them. In order to express whether such transformations are correct, we need the notion of equivalence of two Boolean formulas.

Definition 2.5. *Two Boolean formulas are called* equivalent*, if they share exactly the same models.*

When discussing the SAT problem, equivalence of Boolean formulas is sometimes too strong. For example, equivalence does not allow us to introduce or eliminate variables or to restrict the set of models while preserving satisfiability. We are often only interested in finding a single model, so it is not necessary to keep all models invariant. We formalize a weaker notion of equivalence by introducing the concept of satisfiability-equivalence.

Definition 2.6. *Two formulas are* satisfiability-equivalent*, if they are either both satisfiable or both unsatisfiable.*

Satisfiability-equivalence is obviously much weaker than equivalence; in particular, if ϕ and ϕ' are satisfiability-equivalent, models of ϕ cannot naturally be recovered from models of ϕ' (or vice-versa). In practice we will only transform formulas ϕ into new formulas ϕ' in ways that still allow us to efficiently translate ϕ' models to ϕ models.

2.2.4 CNF formulas

Programmatically handling general Boolean formulas in a SAT solver is complicated as those formulas can form arbitrary trees of subformulas. Therefore, we will restrict ourselves to a subset of formulas that have a certain form. It turns out that this form is still general enough to encode arbitrary problems.

Definition 2.7. *Let $a_1, \ldots, a_k$ be literals. The term $(a_1 \vee \ldots \vee a_k)$ is called a* clause *(of length k) and the term $(a_1 \wedge \ldots \wedge a_k)$ is called a* cube *(of length k). Clauses of length one, two and three are respectively called unary, binary and ternary clauses. We usually denote clauses by C or D.*

Let ϕ be a formula. If $\phi = C_1 \wedge \ldots \wedge C_m$ where $C_1, \ldots, C_m$ are clauses, we say ϕ is in Conjunctive Normal Form (CNF)*. Likewise if $\phi = D_1 \vee \ldots \vee D_m$ where $D_1, \ldots, D_m$ are cubes, we say ϕ is in* Disjunctive Normal Form (DNF)*.*

If a CNF (or DNF) formula entirely consists of clauses of length less than k, it is called k-CNF (or k-DNF).

Because clauses and CNF formulas have a fixed structure, we can use the mathematical set notation to denote them. Instead of writing $(a_1 \vee \ldots \vee a_k)$ as in definition 2.7, we use the notation $\{a_1, \ldots, a_k\}$. Similarly, we replace the notation $C_1 \wedge \ldots \wedge C_m$ with the set $\{C_1, \ldots, C_m\}$. When using set notation, we understand clauses as sets, but CNF formulas as multisets. This is due to practical considerations; a program should easily be able to remove duplicate literals from clauses when reading its input and avoid generating such duplicate literals at runtime, however, it might be expensive to detect duplicate clauses.

A basic theorem of Tseitin [85] states that CNF formulas are sufficient to encode arbitrary Boolean problems while preserving satisfiability-equivalence. Later work [72] improved upon this encoding, yielding encodings that are more compact in practice.

Theorem 2.8. *An arbitrary Boolean formula ϕ can be transformed to a satisfiability-equivalent CNF formula ϕ' in polynomial time. Only a linear number of new clauses and variables is introduced in this process. Furthermore every model of ϕ' is also a model of ϕ if it is restricted to the variables of ϕ.*

The proof of this theorem is constructive: For each subformula that is not already in CNF, a new variable is introduced. The subformula is replaced by that variable and clauses that encode the equivalence between the new variable and the original subformula are added to the formula. It should be noted that the theorem does not hold for DNF formulas as equivalences between variables and subformulas cannot easily be encoded in DNF.

As a result of this theorem, when considerung the SAT problem, it suffices to work with CNF formulas. Thus, the SAT problem for arbitrary Boolean formulas is equivalent to the *CNF-SAT* problem.

Many restrictions of CNF-SAT have been studied. In particular, it is well known that the 2-CNF-SAT problem is solvable in linear time. The same holds if all clauses are Horn clauses; a *Horn clause* is a clause that contains at most one positive literal.

In contrast to that, it is also easy to see that SAT remains NP-hard if all clauses are restricted to a length of three. In fact, 3-CNF-SAT is often studied in the context of random CNF formulas and when theoretical upper bounds on the complexity of SAT are considered. We will, however, not restrict us to 3-CNF formulas and allow clauses of arbitrary length.

In the remainder of this thesis, we will use the terms SAT and CNF-SAT interchangeably.

2.2.5 Implication and incidence graphs

Sometimes, modeling relations between variables and literals as graphs is more convenient than stating them in the language of CNFs. In this subsection we will introduce two graphs that can be associated with a CNF formula.

The first of those graphs is the variable-incidence graph.

Definition 2.9. *The* variable-incidence *graph of a CNF formula ϕ is the undirected graph $VI(\phi)$ which has variables as vertices and which has an edge between two variables x and y if x and y appear in the same clause of ϕ.*

This graph encodes the connectivity of variables that are part of the CNF formula. In particular, in order to find a model of a CNF formula, it suffices to find a model of each connected component of the variable-incidence graph.

The second graph that we consider here is the binary-implication graph.

Definition 2.10. *The* binary-implication graph *$BIG(\phi)$ of a CNF formula ϕ is the digraph which has literals as vertices and which has two edges $\bar{c_1} \to c_2$ and $\bar{c_2} \to c_1$ for every binary clause $\{c_1, c_2\} \in \phi$.*

Edges in $BIG(\phi)$ correspond to implications that are encoded as clauses in ϕ: If there is an edge $b \to c$ in $BIG(\phi)$, then c must be assigned to true whenever b is true. Thus, cycles in $BIG(\phi)$ correspond to literals that have to take the same value in every model of ϕ. Pairs of such literals are called *equivalent literals.*

Many algorithms that are complex on general CNF formulas can be simplified to graph algorithms when only the 2-CNF parts of those formulas are considered. Particularly, if ϕ is a 2-CNF formula, then $BIG(\phi)$ encodes ϕ completely. In this case ϕ is satisfiable, if no cycle of $BIG(\phi)$ contains both the positive and the negative literal of the same variable. This proves our claim that 2-CNF formulas are solvable in polynomial time which we stated in the previous subsection.

2.2.6 Resolution

Several proof systems for propositional logic are studied in the literatuire. In the context of SAT solvers, proof systems allow us to verify the correctness of the solvers: If we can show that the actions that a SAT solver performs correspond to derivations in a proof system, we have proven the correctness of the solver. Regarding CNF formulas, *resolution* is the most important proof system.

Definition 2.11. *Let x be a variable and let C and C' be two clauses. We say that C and C' can be* resolved *by variable x if $x \in C$ and $\bar{x} \in C'$ or vice versa. If that is the case, the* resolvent *$C \otimes_x C'$ is defined by $C \otimes_x C' := (C \setminus \{x\}) \cup (C' \setminus \{\bar{x}\})$.*

If the choice of the variable x is obvious from context, we drop the index of the $\otimes$ sign.

It is easy to see that adding resolvents to a CNF formula does not restrict the set of models of that CNF formula: If the resolvent $C \otimes C'$ is unsatisfied under some assignment, one of the original clauses C and C' also has to be unsatisfied. Therefore, the following lemma holds:

Lemma 2.12. *Let $C, C' \in \phi$ be two clauses. If C and C' can be resolved, then ϕ and $\phi \cup \{C \otimes C'\}$ are equivalent.*

Indeed, iterating this procedure yields a complete procedure to solve the SAT problem. This is formulated in the well-known resolution theorem [75]:

Theorem 2.13. *Let ϕ be a CNF formula. ϕ is unsatisfiable iff the empty clause $\varnothing$ can be derived by successively adding resolvents to ϕ.*

It is known that there are unsatisfiable CNF formulas which require an exponential amount of resolution steps to derive the empty clause. This was first proved [39] for CNF formulas that encode the pigeonhole principle [29].

An additional difficulty when trying to use resolution to solve SAT is that there is no natural order in which clauses should be resolved. The choice of such an order will heavily affect the number of resolution steps that are required to solve a given problem. In particular, it is known that several restrictions of resolution are exponentially weaker than unrestricted resolution. For example, this is the case for regular resolution [85] and tree-like resolution [86]. Here, a resolution proof is called *regular* if every variable is only resolved once along every path through the resolution DAG; a resolution proof is called *tree-like* if the resolution DAG is a tree.

On the other hand, if resolution is augmented with an *extension* rule that allows the definition of equivalences of the form $x \leftrightarrow (c_1 \vee \ldots \vee c_k)$, the resulting proof system is exponentially stronger than resolution alone [29] and in fact equivalent to more traditional proof systems like extended Frege systems.

2.2.7 Unit propagation and DPLL

Unit propagation is a simple deduction rule that is based on the observation that if there is a clause $\{c\}$ of length one in a CNF formula ϕ, then c must be part of every model of ϕ. Unit propagation is heavily used in modern SAT solvers. This subsection will present the unit propagation rule and the DPLL algorithm for SAT solving that is based on this rule.

Definition 2.14. *Let ϕ be a CNF formula and let τ be an assignment. The set $\mathrm{UP}_\phi(\tau)$ is the smallest superset of τ so that if $a_1, \ldots, a_k \in \mathrm{UP}_\phi(\tau)$ and there is a clause $\{\bar{a}_1, \ldots, \bar{a}_k, c\} \in \phi$, then $c \in \mathrm{UP}_\phi(\tau)$. The clauses $\{\bar{a}_1, \ldots, \bar{a}_k, c\}$ are called* unit clauses[1] *and forming $\mathrm{UP}_\phi(\tau)$ is called* unit propagation *or* Boolean constraint propagation *(BCP).*

Formalizing our statement from the beginning of this subsection gives us the following lemma:

Lemma 2.15. *Let τ be an assignment of some CNF formula ϕ. If τ can be extended to a model of ϕ, then $\mathrm{UP}_\phi(\tau)$ is a subset of this model. In particular, τ cannot be extended to a model if $\mathrm{UP}_\phi(\tau)$ is conflicting.*

[1]Note that we use the two terms "unit clause" and "unary clause" with slightly different meanings. Unary clauses are always unit but the converse is not true.

In the special case of an empty τ this gives us the statement that ϕ is unsatisfiable, if $\mathrm{UP}_\phi(\varnothing)$ is conflicting. However, the converse is not true; unit propagation is not a complete procedure for SAT solving.

Computing unit propagation is one of the most performance critical parts of a modern SAT solver. Solvers use specialized data structures to be able to compute unit propagation efficiently. We will not discuss how this is done here. Instead, the discussion of how unit propagation is computed in practice will be postponed to chapter 3.

The first SAT solving algorithm that employed unit propagation was the DP algorithm [32] which was later improved to the DPLL algorithm [31]. Both algorithms are named after their inventors Davis, Putnam, Logemann and Loveland. DPLL is a backtracking algorithm that performs unit propagation at every node of the search tree. Branching heuristics are only used when unit propagation does not fix any additional variables [2]. Algorithm 2.1 depicts a recursive variant of the DPLL algorithm. The algorithm returns true if the input CNF formula ϕ is satisfiable, and false otherwise.

Algorithm 2.1 DPLL algorithm

```
 1: procedure DPLL($\tau$)
 2:     $\tau \leftarrow \mathrm{UP}_\phi(\tau)$
 3:     if $\tau$ is conflicting then
 4:         return false
 5:     else if there is a variable $x$ with $x, \bar{x} \notin \tau$ then
 6:         if DPLL($\tau \cup \{x\}$) then
 7:             return true
 8:         end if
 9:         return DPLL($\tau \cup \{\bar{x}\}$) via tail recursion
10:     else
11:         return true
12:     end if
13: end procedure
```

The process of selecting a branching variable is called a *decision*; thus, the literal that is assigned by a decision is called a *decision literal.* The recursion depth at which those decisions are made is called the corresponding *decision level.* Here, the tail recursion does not count as an own level. Per convention, we say that literals which are assigned by $\mathrm{UP}_\phi(\varnothing)$ (i.e. the first iteration of unit propagation), are assigned at decision level zero.

Note that if a literal c is assigned as a decision literal but $\tau \cup \{c\}$ cannot be extended to a model (as determined by recursive calls to the algorithm), the DPLL procedure returns to the decision level of c and then assigns $\bar{c}$ at the previous decision level. At this point, $\bar{c}$ is not a decision literal

[2]The original DP and DPLL algorithms also include a "pure literal" rule that adds a literal c to the current assignment if $\bar{c}$ does not occur in any clause that is not already satisfied. As this rule does not impact the practical performance of the algorithm, we ignore it here.

anymore: The solver has “learned” that $\bar{c}$ must be true under the current assignment τ. This can be seen as a very limited form of the Conflict-Driven Clause-Learning mechanism that we will study later in chapter 3.

The recursive structure of the algorithm makes DPLL very easy to understand. However, implementing DPLL recursively is often not desirable: As we will see later, a recursive implementation makes some extensions of DPLL difficult to state because they require backtracking more than one level. Furthermore, as industrial CNF formulas might have millions of variables, a simple recursive implementation is likely to exhaust the space that is available on call stacks of typical programming languages. Therefore, DPLL is usually implemented iteratively, as shown in algorithm 2.2.

Algorithm 2.2 Iterative DPLL algorithm

```
procedure DPLL
    loop
        PROPAGATE
        if ATCONFLICT then
            if CURRENTDECLEVEL $> 0$ then
                $a \leftarrow$ last decision literal
                BACKTRACK
                ENQUEUE($\bar{a}$)
            else
                return false
            end if
        else if there is an unassigned variable $x$ then
            NEWDECISIONLEVEL
            ENQUEUE($x$)
        else
            return true
        end if
    end loop
end procedure
```

Note that the iterative variant of the DPLL procedure does not take the assignment τ as an argument, instead, a global assignment is managed implicitly by the algorithm. The algorithm uses an internal stack to remember the decision literals. The position of those literals on this stack equals the decision level in the recursive algorithm, with CURRENTDECLEVEL denoting the size of the stack. PROPAGATE updates the global assignment via unit propagation. NEWDECISIONLEVEL adds a new decision level to the stack, while BACKTRACK removes the last decision level and undoes all literal assignments that were done on that decision level. ENQUEUE assigns a literal, with the first literal after NEWDECISIONLEVEL becoming a new decision literal.

All complete state-of-the-art solvers for SAT run some variant of this DPLL algorithm. They typically amend it either with more sophisticated decision heuristics (i.e. they replace the selection of variable x in the previously named algorithms with something less arbitrary) or add additional rules to the algorithm or both.

2.2.8 The lookahead procedure

Lookahead algorithms are a class of DPLL-based algorithms. Lookahead algorithms can directly be used to solve SAT. We are not primarily interested in this approach; however, we use lookahead procedures as subprocedures in some of the algorithms that we study in chapters 4 and 5.

Lookahead algorithms use a lookahead procedure as sophisticated decision heuristics: Before determining a branching variable, all literals (or a preselected subset of all literals) are probed iteratively. Each literal is assigned and unit propagation is performed. Then, the new assignment is evaluated and discarded again. Typically, this evaluation results in a *score* for each literal. The decision procedure then combines the score of the positive and the negative literal of each variable and picks the variable with highest combined score.

We will not discuss the evaluation heuristic here. Some discussion on this topic will be done in chapter 4. Instead, we focus on the lookahead procedure itself.

Algorithm 2.3 depicts the naive lookahead procedure which takes a set L of literals as input. The procedure returns true if it could successfully evaluate all literals of L. Otherwise false is returned. The latter case can happen if assigning one of the literals c leads to a conflict. In this case c is called a *failed literal*. The solver then learns that $\bar{c}$ must be assigned on the previous decision level. It can either repeat the entire lookahead on $L \setminus \{c\}$ and update scores that were already associated with some literals of L or it can reinvoke the lookahead procedure on only those literals which did not receive a score yet.

2.2.9 Tree-based lookahead

In the last section we have stated a naive lookahead procedure. In many applications this procedure can be improved by reducing the number of unit propagations that have to be done.

In particular, this is the case if we suppose that the lookahead evaluation function only depends on the assignment that results from applying unit propagation after c has been assigned (and not on other data like the order in which literals are assigned). In order to improve the procedure we notice that if we have an implication $c \to b$, then we can assign b first, compute the evaluation function for b, then assign c and compute the evaluation function for c, without undoing the assignment of b first. As b would have to be assigned anyway, this optimization reduces the number of literals that

Algorithm 2.3 Naive lookahead procedure

```
procedure LOOKAHEAD($L$)
    for $a \in L$ do
        if $a$ or $\bar{a}$ is already assigned then
            continue
        end if
        NEWDECISIONLEVEL
        ENQUEUE($a$)
        PROPAGATE
        if ATCONFLICT then
            BACKTRACK
            ENQUEUE($\bar{a}$)
            PROPAGATE
            return false
        else
            Evaluate the current assignment
            BACKTRACK
        end if
    end for
    return true
end procedure
```

have to be assigned during unit propagation, although it does not reduce the number of times that we have to invoke the PROPAGATE and BACKTRACK procedures.

Improving the naive procedure from the last section using this idea leads to the *tree-based* lookahead procedure. Tree-based lookahead is described in [46] but was earlier implemented in the March solver [44].

Algorithm 2.4 Tree-based lookahead procedure

```
procedure LOOKAHEAD($L$)
    $Q$ : Queue
    procedure DFS($a$)
        $Q$.PUSH($a$)
        for $a' \to a$ do
            if $a' \notin Q$ then
                DFS($a'$)
            end if
        end for
        $Q$.PUSH(nil)
    end procedure

    for $a \in L$ do
        if $a \notin Q$ then
            DFS($a$)
        end if
    end for
    $a \leftarrow Q$.POP()
    if $a \neq$ nil then
        NEWDECISIONLEVEL
        ENQUEUE($a$)
        PROPAGATE
        if ATCONFLICT then
            BACKJUMP(0)
            ENQUEUE($\bar{a}$)
            return false
        else
            Evaluate the assignment
        end if
    else
        BACKTRACK
    end if
    return true
end procedure
```

The tree-based lookahead procedure first performs a DFS through the binary-implication graph in lines 2 to 16. This DFS builds a forest of literals, so that for each literal c, c implies its parent. The resulting forest is stored as a queue. In the second part of the procedure, this queue is traversed. During the traversal the procedure assigns a new literal whenever it descends into a subtree of the DFS forest and backtracks when it finishes visiting a subtree of the DFS forest.

2.2.10 The state of the art

In this subsection we will briefly present the state of the art of SAT solvers.

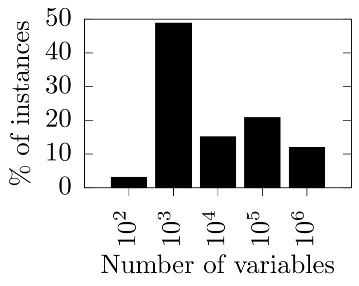
CNF formulas are commonly grouped into three categories: *Real-world* (or *industrial*) instances, *hard combinatorial* instances and *random formulas*. Real-world instances encode problems that arise from industrial applications, for example from bounded model checking [21]. Hard combinatorial instances are usually encodings of NP-hard problems. Random formulas can be generated by various mechanisms, for example k-CNF formulas can be constructed by picking literals from a uniform distribution. Random formulas and some types of hard combinatorial instances are typically quite small and only contain a few thousands of variables. Other hard combinatorial instances and most industrial instances are much larger. The best known approaches to solve these instances differ among the three categories.

The state of the art is periodically evaluated by the SAT Competition [78]. For satisfiable random formulas, the best known algorithms are incomplete *stochastic local search* (SLS) solvers that use randomized methods to find models. For unsatisfiable random formulas and for small hard combinatorial formulas, solvers based on lookahead heuristics are very successful. However, these solvers become less effective when large formulas are considered.

In this thesis, we will focus on large industrial and hard combinatorial instances. These formulas have more "structure" than randomly generated formulas: For example, in many cases, large parts of the formulas are already encoded in the binary-implication graph. For industrial and hard combinatorial formulas, the Conflict-Driven Clause-Learning (CDCL) algorithm is the most successful approach. We will discuss this algorithm in detail in chapter 3. The performance of the CDCL algorithm depends on the structure of the instances in order to quickly prune the search space. In the remainder of this subsection, we shortly discuss the peculiarities of large industrial and hard combinatorial instances.

As an example for industrial and hard-combinatorial CNF instances, we take all formulas from the Main track of SAT Competition 2017. Figure 2.1 depicts the distributions of the numbers of variables and clauses in these instances. Instances typically consist of thousands to millions of variables and up to tens of millions of clauses.

In Figure 2.2 the size of the formulas with respect to the number of



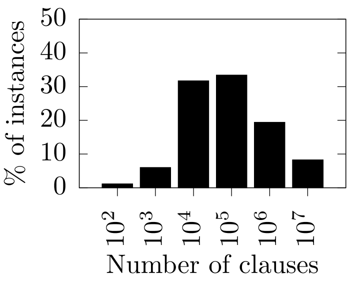
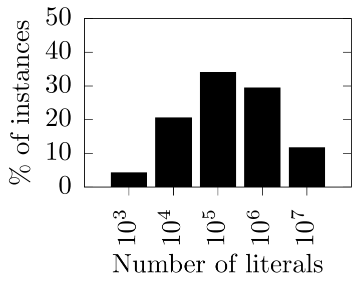


Figure 2.1: Distribution of variables and clauses in SC'17 instances
The histograms plot the numbers of variables and clauses, rounded down to the previous power of ten.

literals and their memory consumption is plotted. The figure illustrates that the number of literals in industrial and hard combinatorial formulas is of the same order of magnitude as the number of clauses. In fact, out of all clauses in this set of formulas, 45.4% are binary clauses and 51.5% are ternary clauses. Only 3.1% of all clauses have a length greater than three. Thus, in order to be successful, a SAT solver for those instances does not only need to handle large amounts of clauses but also needs to handle small clauses efficiently. Minimizing memory consumption is especially important because solvers regularly allocate new clauses during search (as we will see in chapter 3) and parallel solvers need to store a separate copy of the formula for each thread.



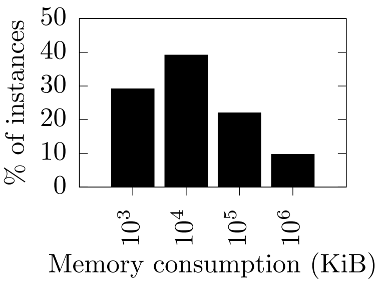


Figure 2.2: Size of SC'17 instances
Again, numbers are rounded down to the previous power of ten. Memory consumption was measured by parsing the formula using the satUZK solver.

To better understand how the afforementioned structure of CNF formulas emerges, we give an example that demonstrates how combinatorial problems are encoded into CNF.

Example 2.16. *Consider the NP-complete problem of graph 3-coloring. Let G be a graph. We want to construct a CNF formula ϕ so that each model of ϕ corresponds to a 3-coloring of G.*

This can be achieved in the following way: We introduce variables x_v^k with $k \in \{1, 2, 3\}$ denoting a color and $v \in V(G)$ iterating through the vertices of G. After the construction is completed, the literal x_v^k will be true *in a given model if and only if v receives the color k in the corresponding coloring.*

We now construct the clauses of ϕ. For each vertex v, the clause $\{x_v^1, x_v^2, x_v^3\}$ ensures that at each vertex receives at least one color. On the other hand, the clauses $\{\overline{x_v^1}, \overline{x_v^2}\}$, $\{\overline{x_v^1}, \overline{x_v^3}\}$, and $\{\overline{x_v^2}, \overline{x_v^3}\}$ prevent coloring the vertex with more than one color. Finally, for each edge $u - v \in E(G)$, the clauses $\{\overline{x_u^1}, \overline{x_v^1}\}$, $\{\overline{x_u^2}, \overline{x_v^2}\}$, and $\{\overline{x_u^3}, \overline{x_v^3}\}$ guarantee that adjacent vertices do not receive the same color. This completes the construction of ϕ.

As we can see, the construction in example 2.16 results in $3(|V(G)| + |E(G)|)$ binary clauses and $|V(G)|$ ternary clauses. While graph coloring can be seen as a synthetic problem, it is obvious that "at-least-one", "at-most-one" and similar constraints are ubiquitous in industrial applications.

2.3 Parallel computing

As we will discuss parallel algorithms in chapters 4 and 5, we need some terminology related to parallel computing. This terminology will be introduced in this section.

2.3.1 Parallel computation models

The best-known parallel computation model is the P-RAM machine. P-RAM extends the *random-access machine* (RAM) model to multiple processors that are all connected to shared memory. These processors operate in a *single instruction, multiple data* (SIMD) fashion: All processors run the same instructions but each processor is allowed to operate on a different set of data. In pseudocode, this is often expressed via **for** … **do in parallel** statements. Several specializations of the P-RAM model exist; most commonly, *exlusive read, exclusive write* (EREW) or *concurrent read, exclusive write* (CREW) models are considered.

The class of decision problems that can be solved by a P-RAM machine using a polynomial number of processors and polylogarithmic time is called NC[3]. The definition of NC is the same in the EREW and CREW models. It is an open problem whether P equals NC. Similar to P-reductions in the context of the NP complexity class, we can define NC-reductions as reductions that can be executed on a P-RAM machine with a polynomial number of processors in polylogarithmic time. A problem is called P-hard if there is a reduction from every problem in NC to that problem. If such a problem is also in P, it is called P-complete. It is well-known that the

[3]NC stands for "Nick's class" and is named after Nick Pippenger.

circuit value problem (CVP) is P-complete [56]. CVP is the problem of determining if the output of a circuit consisting of Boolean gates is `true` for a given set of input values.

In reality, shared memory systems are limited in their scalability even when running NC-algorithms. Modern processors feature a hierachy of caches; writes to memory typically need to invalidate entire cache lines and thus affect the performance of other processors. In addition to that, memory latency is affected by the distance of a processor to the memory module. Attaching an arbitrary amount of processors to a given memory module is not physically possible without increasing memory latencies. Furthermore, implementations of shared memory algorithms require the use of synchronization primitives like mutexes or carefully crafted lock-free algorithms. This leads to contention between processors and increases the complexity of the implementation.

Because of these inherent problems, large parallel computers typically use a *distributed* architecture. In such a distributed architecture, computers consist of individual *nodes* that all have isolated memory spaces. All nodes are connected by a network that we call an *interconnect*. Instead of performing implicit communication over shared memory, the nodes communicate by explicitely exchanging message via the interconnect. Communication costs on the interconnect typically depend on the physical proximity of two nodes. We do not specify the exact shape of the interconnect as we do not want to specialize our algorithms for specific interconnects.

2.3.2 Dynamic load balancing

In many applications, evenly distributing the load (i.e. the computational tasks that are to be performed in parallel) to different processors and/or different nodes is hard to achieve statically. Even if a good initial load distribution is found, some processors might finish their tasks earlier than others. If this happens the processors become idle and their computational resources are wasted. *Dynamic load balancing* is the process of adaptively moving load between processors so that idle periods of processors are minimized.

The diffusion method is one of the earliest algorithms that were considered for dynamic load balancing [30]. An example of this method is the *dimension-exchange* algorithm on a hypercube. Consider a d-dimensional hypercube. Its vertices are binary strings of length d. Suppose that each processor is associated to one of the vertices. Note that each processor has exactly one neighbor in each for the d dimensions. The dimension-exchange algorithm iteratively cycles through the d dimensions. For each dimension i it exchanges load between each pair of neighbors in dimension i so that the load on both of those neighbors is equal. Eventually, this procedure balances the total load uniformly.

Note that this diffusion method does not take communication that is required to solve the computational tasks into account. There are more sophisticated load balancing algorithms that do account for the cost of this communication (e.g. methods that are based on graph partitioning). However, in our cases, only negligible amounts of communication are required to solve each task. Thus, the dimension-exchange methods suffices in our applications.

2.3.3 Practical parallel computing

In our implementations, we use both shared memory and communication over interconnects. Each node consists of one or more *sockets*. Each socket hosts multiple *cores* and each of those cores is a full-fledged CPU. All cores of the same node (even those cores that belong to different sockets) share their memory. However, the latency of accesses to different parts of this shared memory might differ among different sockets.

Operating systems (OSs) typically abstract over these details. They allow programs to create “processes” and “threads”. A *process* consists of multiple *threads* that each represent a single execution context. All threads of the same process share the same memory. We assume that the OS maps each thread of a program to a different physical core and the number of threads we create will usually equal the number of cores on a node or socket. When we talk about threads that execute a specific algorithm (as opposed to threads that only perform book keeping, communication or other secondary tasks) we often use the term *worker*.

Chapter 3

The satUZK solver

In this chapter we review the algorithms of our CDCL solver satUZK and discuss how to implement them efficiently. We start by discussing the algorithms that make up the generic CDCL framework and then describe how they are realized in satUZK.

The C++ implementation of satUZK was written by the author of this thesis. We submitted satUZK to previous SAT Competitions in 2012, 2013, 2014 and 2017 [96, 90, 89, 87]. In 2013, the sequential satUZK-seq solver won a bronze medal in the Application, SAT track. The parallel ppfolioUZK solver [95], written by Andreas Wotzlaw, included satUZK as a subroutine and won a gold medal in 2012. satUZK is available from `https://github.com/satuzk/satUZK`.

3.1 CDCL basics

This section will review the components that constitute the core of modern CDCL solvers. Most of the algorithms that are disussed in this section are not novel, however, presenting them is necessary to discuss their specific implementation in satUZK in the next section.

The modern CDCL approach was not developed in a single step. Instead, it was refined iteratively starting from the DPLL procedure. We give a short overview of this development before we present the individual steps in detail in the following subsections. The first clause learning solver was GRASP [79][1]. Chaff [69] improved upon GRASP by implementing the two-watched-literals scheme[2], restarts, learned clause deletion and the VSIDS decision heuristic. zChaff [99] then introduced the 1-UIP learning scheme.

[1]The relsat solver [12] independently added clause learning to a DPLL algorithm but its architecture is quite different from that of modern CDCL solvers.

[2]The two-watched literal scheme is related to the head/tail scheme that was already implemented in the SATO solver [98], which is based on DPLL without clause learning.

MiniSat improved upon zChaff by incorporating learned clause minimization [81, 84, 34] and preprocessing techniques [33]. Glucose [8] introduced the LBD measure for clause quality. Lingeling [15] added inprocessing algorithms to the search procedure.

3.1.1 The CDCL algorithm

The *Conflict-Driver Clause-Learning* (CDCL) algorithm is an extension to the iterative DPLL procedure that was presented in algorithm 2.2. The CDCL algorithm itself improves upon the DPLL algorithm by preventing it from reaching the same conflict multiple times. This is done by adding new clauses to the CNF formula every time a conflict is encountered. The following example demonstrates how this mechanism works.

Example 3.1. *Consider the CNF formula* $\{\{\bar{w}, y\}, \{\bar{x}, y\}, \{\bar{y}, z\}, \{\bar{y}, \bar{z}\}\}$*. Assume that the DPLL algorithm first assigns* w*. Then,* y *and* z *are fixed by unit propagation and the propagation procedure runs into a conflict when trying to assign* $\bar{z}$*. Note that assigning* w *was not necessary in order to trigger this conflict; just assigning* y *would have led to the same conflict. However, DPLL does not recognize this fact. Instead, the algorithm backtracks (i.e. it undoes every assignment) and assigns* $\bar{w}$*. If the DPLL algorithm now assigns* x*, it runs into the same conflict again.*

If we would have used our knowledge that assigning y *already leads to a conflict, we could have assigned* $\bar{y}$ *after the solver backtracks. In fact, we can add the clause* $\{\bar{y}\}$ *to the CNF formula: Adding the clause forces the solver to perform this assignment via unit propagation; this is the idea of the CDCL algorithm. Unit propagation then also fixes* $\bar{w}$ *and* $\bar{x}$ *automatically. Thus, only a single backtracking operation is required to fix the same amount of literals that were fixed by multiple backtracking operations of the DPLL algorithm.*

The example already shows how the DPLL algorithm is to be extended to be turned into CDCL. When the solver runs into a conflict, *conflict analysis* is performed, which outputs a clause C that is logically implied by the input CNF formula but unsatisfied under the current assignment. C is called a *learned clause* or a *conflict clause* in this context. The solver then backtracks to the first decision level where C is unit and fixes a new literal by applying unit propagation to C. Algorithm 3.1 formulates this procedure in pseudocode. The only difference to algorithm 2.2 (the iterative DPLL algorithm) is the handling of conflicts in lines 6 to 10. The basic structure of this CDCL algorithm was introduced by GRASP [79].

Note that the algorithm can backtrack more than one level as result of a conflict. This mechanism is called *non-chronological* backtracking [79].

Algorithm 3.1 CDCL algorithm

```
procedure CDCL
    loop
        PROPAGATE
        if ATCONFLICT then
            if CURRENTDECLEVEL $> 0$ then
                $C \leftarrow$ ANALYZECONFLICT
                $k \leftarrow$ first decision level so that $C$ is unit
                while CURRENTDECLEVEL $> k$ do
                    BACKTRACK
                end while
            else
                return false
            end if
        else if there is an unassigned variable $x$ then
            NEWDECISIONLEVEL
            ENQUEUE($x$)
        else
            return true
        end if
    end loop
end procedure
```

Non-chronological backtracking ensures that the newly learned clause is propagated at the correct decision level. It also cleans decision literals that did not participate in the conflict from the assignment.

There are, of course, formulas which require an exponential number of conflicts to be solved. For such formulas, the CDCL algorithm generates an exponential number of conflict clauses. Nevertheless, as long as the algorithm does not produce duplicate conflict clauses, the procedure eventually terminates. We will see that learning schemes that are used in practice indeed do not generate duplicate clauses. Thus, this form of the CDCL algorithm is a complete algorithm for SAT. In reality, solvers delete learned clauses heuristically in order to prevent the solver from consuming an exponential amount of memory. If this is done, the completeness guarantee of the algorithm is lost. In practice, this is usually not considered a problem; the result from a solver that does not terminate within a reasonable time frame is indistinguishable from the result of a solver that does not terminate at all.

Although the CDCL algorithm shares an exponential upper bound with the DPLL algorithm, CDCL is much stronger than DPLL, even from a theoretical point of view. While DPLL can be simulated by regular resolution (see 2.2.7), CDCL is known to be equivalent to general resolution [71, 13] [3]. Thus, there are formulas that can be solved by CDCL in polynomial time

[3] The proof requires some assumptions that are reasonable but still not always satisfied in practice. However, the proof still gives a theoretical justification to the CDCL algorithm. For example the proof in [71] requires a restart (see subsection 3.1.6) after each conflict. Real solvers do not restart after each conflict as restarts have a non-negligible runtime cost.

but that require exponential amounts of time for traditional DPLL solvers (see 2.2.6 and [1]).

In the following subsections we will discuss how the conflict analysis procedure works and introduce a few extensions to algorithm 3.1 that modern CDCL solvers use.

3.1.2 Clause learning

The previous subsection presented an example of how clause learning can be used to prun the search space of a SAT solver. In this subsection, we will develop a systematic method to construct useful learned clauses.

In order to extract learned clauses from conflicts, the CDCL algorithm constructs a "conflict graph" that encodes which literals were used to fix new literals during unit propagation. The solver builds this graph by remembering the clause $ant(c)$ that is used to fix a literal c during unit propagation. This clause is called the *antecedent* of c. Conflict graphs are constructed from the antecedents of assigned literals; the following definition explains how this is done.

Definition 3.2. *Consider the run of DPLL algorithm on the CNF formula ϕ. At a given point of time during this run, an* antecedent graph *is a graph that has (a subset of the) literals of ϕ as vertices and that has edges $a_1 \to c$, ..., $a_k \to c$ if and only if c is true under the current assignment and $D = \{c, \bar{a_1}, \ldots, \bar{a_k}\} \in \phi$ is the antecedent of c. We label those incoming edges of c with D.*

Suppose that G is an antecedent graph and that there is a clause $C = \{\bar{a_1}, \ldots, \bar{a_k}\} \in \phi$ with $a_1, \ldots, a_k \in V(G)$ (i.e. the DPLL algorithm encountered a conflict). If we add a special vertex $\perp$ and edges $a_i \to \perp$ to G, for $i = 1, \ldots, k$, and label those edges with C, the resulting graph is called a conflict graph*.*

A conflict cut *is a cut of a conflict graph G that separates all decision literals from $\perp$. Literals on the decision side of the cut which are adjacent to the conflict side of the cut (i.e. the side containing $\perp$) are called* cut literals*.*

Conflict graphs were introduced by GRASP [79].

Let us illustrate how the conflict graph looks like in case of the example from the last subsection. We will see that the inverses of the cut literals of conflict cuts form possible learned clauses.

Example 3.3. *Consider the same situation as in example 3.1. The conflict graph from that example is depicted in figure 3.1. This figure also shows three different conflict cuts. We form learned clauses by taking the inverses*

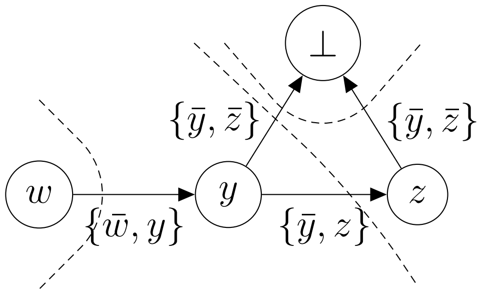


Figure 3.1: Conflict graph from example 3.1
Three cuts, corresponding to the sets $\{w\}$, $\{y\}$ and $\{y, z\}$ of cut literals (from left to right) are drawn into the graph.

of the cut literals. This results in the clauses $\{\bar{w}\}$, $\{\bar{y}\}$ and $\{\bar{y}, \bar{z}\}$ (from left to right).

The learned clause $\{\bar{y}, \bar{z}\}$ is not very useful: As it contains two literals on the current decision level, it will not be unit after backtracking. The clause $\{\bar{w}\}$ does not have this problem. However, this clause contains exactly the single decision literal that is currently assigned. Cuts that have only decision literals as cut literals are called decision cuts*. If a CDCL solver always learns decision cuts, the CDCL algorithm degenerates to the DPLL algorithm without clause learning.*

Therefore, we conclude that $\{\bar{y}\}$ is the best learned clause that we can extract from the given conflict graph. It only contains a single literal at the current decision level (i.e. it will be unit after backtracking) and is also strictly superior to $\{\bar{w}\}$ as it will fix both $\bar{y}$ and $\bar{w}$ after backtracking.

Next, we prove that learning clauses using this mechanism is indeed sound. For that, we show how the learned clause can be derived using resolution. Results from chapter 5 will later generalize this result and show that learned clauses are in fact so called "asymmetric tautologies".

Lemma 3.4. *Let G be a conflict graph for some CNF formula ϕ. For every conflict cut of G, the inverses of the cut literals form a clause that can be derived by resolution.*

Proof. Our proof explicitely constructs a resolution derivation. Per definition, G contains no cycles. Note that $\perp$ is a sink vertex. Let $a_1, \ldots, a_\ell = \perp$ be a topological ordering of G that ends in the $\perp$ vertex (i.e. there are only edges from a_i to a_j if $i < j$). Let C be the label of the incoming edges of $\perp$. Start the resolution derivation with C; this is our "current clause". We will maintain the invariant that every literal of the current clause that is not also a cut literal is on the conflict side of the cut; certainly, this is initially true.

Let i be maximal so that a_i is part of our current clause but not a cut literal. In particular, because of our invariant, a_i cannot be a decision literal.

Let C' be the label of the incoming edges of a_i. Resolve our current clause with C' on the literal a_i; the result becomes our new current clause. This is possible because the definition of an antecedent graph guarantees that $\bar{a}_i \in C'$. The resolution operation only adds literals that are predecessors of a_i and those literals have indices less than i in the topological ordering. This ensures that the procedure terminates eventually. Furthermore, as a_i is not a cut literal, the predecessors of a_i are either cut literals themselves or still on the conflict side. Thus, our invariant is maintained. □

Corollary 3.5. *Let D be the set of cut literals for an arbitrary conflict cut of G. Then ϕ and $\phi \cup \{D\}$ are equivalent.*

This lemma gives us many possible conflict clauses. The question now arises how to select the conflict cut? In our previous example 3.3 we already saw that not all conflict cuts yield useful learned clauses. In particular, cuts with more than one cut literal at the current decision level are unhelpful, as they do not fix literals after backtracking.

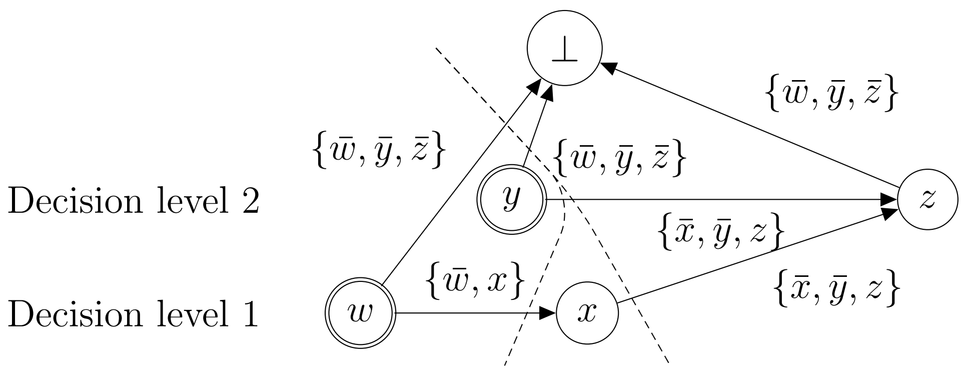
While multiple methods to find a conflict cut have been studied in the literature, virtually all modern CDCL solvers use the "1-UIP" scheme. 1-UIP was empirically proven to be superior to earlier schemes. It was introduced in zChaff [99]. The same paper also contains an evaluation of othe learning schemes.

Definition 3.6. *Let G be a conflict graph. Let c be vertex with decision level k. If all paths from the decision literal of level k to $\bot$ pass through c (i.e. c is a dominator of this subgraph), then c is called a* unique implication point *(UIP) on level k. Note that UIPs of the same level can be naturally ordered.*

We call the UIP that is closest to $\bot$ the first UIP. The 1-UIP *conflict cut is uniquely defined as the conflict cut that contains only the first UIP on the current decision level on the decision side and is as close to $\bot$ as possible on all other decision levels. Here, as close as possible means that a vertex is on the conflict side if and only if it is on a path between the UIP and $\bot$.*

In example 3.3 and figure 3.1, the 1-UIP cut is given by the set $\{y\}$ of cut literals and corresponds to the learned clause $\{\bar{y}\}$.

Note that for each decision level there is at least one UIP, namely the decision literal itself. Thus, the 1-UIP cut is always well-defined. Furthermore, as the 1-UIP cut contains exactly one literal on the current decision level, the generated learned clause will always be useful. This property also guarantees that 1-UIP clauses never duplicate already existing clauses: If an already existing clause was contained in a 1-UIP clause, that clause



$\perp$
$\{\bar{w}, \bar{y}, \bar{z}\}$
$\{\bar{w}, \bar{y}, \bar{z}\}$
$\{\bar{w}, \bar{y}, \bar{z}\}$
Decision level 2
y
z
$\{\bar{x}, \bar{y}, z\}$
$\{\bar{w}, x\}$
$\{\bar{x}, \bar{y}, z\}$
Decision level 1
w
x

Figure 3.2: Learned clause minimization in example 3.7
Displays the conflict graph from the example. Decision literals are indicated by double circles. The right cut corresponds to the usual 1-UIP clause. The left cut corresponds to the 1-UIP clause after minimization has been applied.

would have fixed the UIP on an earlier decision level via unit propagation. Therefore, as we discussed in the previous subsection, the CDCL algorithm using the 1-UIP scheme is complete. Also note that the 1-UIP cut is strictly more useful than any conflict cut that contains a different UIP on the current decision level; any UIP other than the first UIP would fix the first UIP via unit propagation and run into a conflict again.

Note that the 1-UIP cut is not the only useful cut that contains exactly one literal at the current decision level. After all, we could resolve additional literals on earlier decision levels (e.g. replace them with the first UIP on their decision levels). However, there are both practical and theoretical reasons why we prefer the 1-UIP clause. In practice, we will see that the 1-UIP cut can be computed very efficiently; the same is not necessarily true for more complex cuts. Theoretically, while the 1-UIP cut does not minimize the learned clause's length, it is easy to observe that it indeed minimizes the number of decision levels that are part of the learned clause, over all clauses that contain the first UIP on the current level. The reason for that is that the resolution process in the proof of lemma 3.4 only adds decision levels to the clause but can never remove them [4].

3.1.3 Learned clause minimization

In many cases, the 1-UIP clause still contains superfluous literals. *Learned clause minimization* is the process of removing those literals from the learned clause. Learned clause minimization was popularized by MiniSat [81, 84] [5]. The next example illustrates the idea of learned clause minimization.

[4]This property implies that the 1-UIP clauses minimizes the LBD measure over all clauses that contain the UIP. LBD will be discussed in subsection 3.1.5.

[5]It was, however, already considered in [13].

Example 3.7. *Consider the CNF formula $\phi = \{\{\bar{w}, x\}, \{\bar{x}, \bar{y}, z\}, \{\bar{x}, \bar{y}, \bar{z}\}$. Suppose that a CDCL solver assigns w on the first decision level and y on the second decision level. This situation is depicted in figure 3.2.*

The CDCL solver learns the 1-UIP clause $\{\bar{w}, \bar{x}, \bar{y}\}$; here, y is the first UIP on the second decision level. However, the literal $\bar{x}$ in this clause is superfluous: We could resolve the 1-UIP clause with the clause $\{\bar{w}, x\}$ from ϕ to obtain the smaller learned clause $\{\bar{w}, \bar{y}\}$. As this clause is a subset of the 1-UIP clause, it is strictly more useful during unit propagation.

Let us formalize the idea from the preceeding example and formulate it as an algorithm: The *recursive minimization* algorithm removes a literal c from a learned clause C if any path from a decision literal to c in the conflict graph contains a cut literal that is different from c. The correctness of this algorithm follows from the fact that it is equivalent to moving c and some predecessors of c to the conflict side of the cut. As the conflict graph is acyclic, the recursive minimization procedure has a unique result (i.e. the result does not depend on the order in which literals are considered).

It should be noted that recursive minimization is unable to remove entire decision levels from the learned clause: Let c be the only remaining literal of the learned clause on a certain decision level l. If c is the decision literal on level l, then recursive minimization cannot remove c. Otherwise, there is a path from the decision literal on level l to c that only contains literals on level l: If there was no such path, then c would have been propagated on an earlier decision level. However, no predecessor of c on this path is part of the learned clause. Thus, recursive minimization does not remove c.

As a consequence, recursive minimization cannot remove the UIP on the current decision level. In particular, it does not influence the literal that is propagated by the conflict clause after backtracking, nor does it influence the decision level the CDCL algorithm backtracks to [6].

3.1.4 Decision heuristics

Up to this point, we still left the decision heuristic of the CDCL algorithm unspecified. In this section, we will present the most commonly used decision heuristic, namely the *variable-state-independent-decaying-sum* (VSIDS) heuristic.

Definition 3.8. *For a variable x, let $p_i(x) = 1$ if either x or $\bar{x}$ are part of the conflict side of the i-th conflict graph that is encountered by the CDCL algorithm and otherwise $p_i(x) = 0$. The VSIDS score of a variable x after*

[6]It also does not influence the LBD of the clause.

the n-th conflict is defined as

$$vsids(x) = \sum_{i=1}^{n} p_i(x)h^i$$

Here, $h > 1$ is a fixed constant. MiniSat uses a number close to $h = 1.05$.

The VSIDS heuristic selects the variable with highest VSIDS score at each decision. Because of the h^i term, only the last few summands actually matter and the importance of earlier summands "decays" during the CDCL algorithm. Thus, the VSIDS heursitic selects the variable that occurred most frequently in the most recent learned clauses.

In order to quickly determine the variable with highest VSIDS score, solvers usually store all variables in a heap data structure. Most solvers use a binary heap as this implementation seems to yield the best performance in practice. However, insertion and update operations of this heap consume non-negligible amounts of runtime.

There are some alternatives to VSIDS. The *variable-move-to-front* (VMTF) heuristic [77] stores variables in a list and always moves those variables that are part of conflict sides to the front of this list. It always selects the first unassigned variable of the list. While a benefit of the VMTF strategy generally is the better runtime bound of $\mathcal{O}(1)$ (versus the $\mathcal{O}(\log n)$ bound of a VSIDS implementation using binary heaps), dedicated data structures are still necessary in order to implement VMTF efficiently [23]. Another recent alternative to VSIDS is the family of *learning-rate-branching* (LRB) heuristics [64]. Those heuristics explicitly compute the frequency in which variables occur on conflict sides. All of those heuristics are related to VSIDS in the sense that they try to select variables that most frequently occur in recent conflicts. Earlier decision heuristics that tried to pick variables from certain clauses (like the heuristic from BerkMin [37]) are not competitive with VSIDS, VMTF and LRB.

After a decision variable is determined, its polarity has to be fixed in order to determine a decision literal. In modern SAT solvers, this is done by utilizing *progress-saving* [70]: For each variable, the solver memorizes the polarity that the variable is assigned to, whenever it is assigned. If a decision needs to be made, the solver always choses exactly the memorized polarity. Here, the initial polarity is chosen arbitrarily. This policy forces the solver to focus on a small part of the search space. Variables are only flipped when flipping them is necessary; that is as a result of unit propagation. This happens, for example, if a conflict clause forces a flip. In contrast to that, if variables were flipped arbitrarily, the resulting assignment could satisfy large amounts of recently learned clauses, rendering them useless for unit propagation. In this way, the progress-saving mechanism ensures that learned clauses stay relevant to the search.

3.1.5 Clause database reduction

In sections 3.1.1 and 3.1.2 we discussed how CDCL solvers add learned clauses to the CNF formula. In general, a CDCL solver can add an exponential amount of learned clauses to the formula. As the number of clauses in the formula increases, the performance of unit propagation decreases. What is worse is that the solver might even run out of memory on instances that require many conflicts to be solved. To mitigate these problems, it becomes necessary to delete learned clauses after they are no longer useful. This procedure is called *clause database reduction*.

In order to support clause database reduction, the solver has to divide its set of clauses into redundant and irredundant clauses. All clauses that originate from the input CNF formula are *irredundant*, while clauses generated by clause learning are *redundant*. The solver has to ensure that only redundant clauses are removed.

The main problem that has to be solved when designing a clause database reduction algorithm is determining when a clause is no longer useful. While clause database reduction was already present in solvers like Chaff [69], their heuristics differ significantly from modern ones. MiniSat [34] implements a more modern heuristic: It schedules clause database reduction after an exponentially increasing number of conflicts and always deletes half of the learned clauses. MiniSat never deletes binary clauses during database reduction. To determine the clauses that are being removed, MiniSat associates a VSIDS-like score which is called *activity* with each clause. It turns out that this mechanism works well for some families of formulas.

For most industrial CNF formulas, however, a more aggressive reduction policy performs better. Glucose [8] is a successful SAT solver that is based on an aggressive database reduction strategy. Glucose does not rely on the activity score to delete clauses, instead it uses a score called literal block distance (LBD).

Definition 3.9. *Let C be the learned clause that is learned after the m-th conflict. For a literal c, let $l_i(c)$ be the decision level of c before backtracking has been performed as result of conflict i, if c is assigned at that point of time. After the n-th conflict, $lbd(C)$ is defined as*

$$lbd(C) = \min_{m \le i \le n} |\{l_i(c) : c \in C\} \setminus \{0\}|$$

The minimum is only taken over all i where $l_i(c)$ is defined for all $c \in C$.

Note that clauses with an LBD of zero are unit at decision level zero and thus equally useful as unary clauses. Like binary clauses, clauses with an LBD of one are unit after some decision literal has been assigned. In this sense, clauses with LBD one are as important as binary clauses.

Note that the LBD of a clause can potentially change after every conflict. In practice, it is too costly to recompute the LBD of all clauses (or even of all redundant clauses) after each conflict. Thus, the LBD of a clause C is usually computed at the time when C is learned. If C participates in a conflict side afterwards, the LBD of C is recomputed. This mechanism allows clauses to *improve* their LBD over time.

Glucose schedules clause database reduction after a linearly increasing number of conflicts. It always keeps clauses with an LBD smaller or equal to two; this implies that Glucose, like MiniSat, does not delete binary clauses. Half of the remaining clauses are deleted. LBD is used to determine the clauses that will be deleted; clauses with large LBD are deleted first. The activity score is only used to break ties.

3.1.6 Restarts

This subsection will discuss restarts, which are one of the core components that are universally implemented in modern SAT solvers. When a SAT solver *restarts*, it undoes all decisions and starts again from an empty assignment. Restarts were first implemented in Chaff [69].

Technically, in order to restart, it is sufficient to backtrack to decision level zero. Assignments at decision level zero do not need to be undone as they would be repeated anyway. Thus, the solver gains the ability to reconsider decisions that were made at early decision levels.

Note that the purpose of a restart is not encouraging the solver to explore a different part of the search space. Indeed, the VSIDS heuristic and the progress-saving mechanism encourage the solver to explore the same part of the search space even after a restart. Instead, solvers use restarts to clean "unimportant" decision literals from the current assignment. Such literals lead to larger conflict graphs and thus decrease the quality of learned clauses. Hence, restarts are not only important to quickly find solutions to satisfiable instances but also to quickly find unsatisfiability proofs.

Experiments show, that on many classes of formulas, SAT solvers benefit from frequent restarts [38]. In particular, the number of decisions that are required to solve problems often decreases with more frequent restarts. However, especially on large formulas, redoing all unit propagations after a restart has a non-negligible cost. If restarts are scheduled too frequently, this cost can diminish the advantage of a smaller number of decisions. There is some work [92, 73] to reuse parts of the current assignment after a restart in order to reduce the cost of restarts.

There are multiple restart strategies that are used in state of the art CDCL solvers, with the most common ones being Luby restarts and LBD-based restarts. The *Luby* restart [65] strategy (named after its inventor) is a restart strategy that was discovered during the study of Las Vegas algorithms (i.e. randomized algorithms that always return the correct result if they terminate but that have a stochastic runtime bound). For these algorithms, Luby restarts are proven to be optimal up to a logarithmic factor.

In the CDCL context, Luby restarts are usually implemented by restarting after a certain number of conflicts. This number of conflicts is given by the elements of the *Luby sequence* $\mathcal{S}$. Fix some integer $r \in \mathbb{N}$. $\mathcal{S}$ can be constructed as follows: Start with $\mathcal{S}_0 = (r)$. Construct $\mathcal{S}_i$ by appending the element $r2^i$ to $\mathcal{S}_{i-1}$. $\mathcal{S}$ is formed as the concatenation of all $\mathcal{S}_i$. For $r = 1$, this yields the sequence $\mathcal{S} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \ldots)$. SAT solvers usually use values between $r = 100$ (MiniSat) and $r = 2$ (Lingeling in [22]). Even $r = 1$ might be beneficial in some scenarios [73].

Of course, it is questionable whether the behavior of the CDCL algorithm can be modeled as a Las Vegas algorithm. In particular, usual implementations of CDCL are determinisitic. However, as demonstrated by MiniSat, Luby restarts seem to perform well on many families of formulas; this is evaluated in detail in [22].

In contrast to the static Luby strategy, *LBD-based* restarts [9] take the current progress of the CDCL solver into account. Thus, LBD-based restarts form a *dynamic* restart strategy. The LBD-based restart strategy compares the LBD of the last learned clause to a moving average of the LBD scores of recently learned clause (e.g. the last 50 learned clauses). If the LBD of the recently learned clause is significantly higher than the average LBD, a restart is performed.

3.2 satUZK: Algorithms and data structures

In this section, we will discuss the concrete implementation of the CDCL algorithm in satUZK. The resulting sequential SAT solver is called satUZK-seq.

3.2.1 Variable and clause data types

The most fundamental data types relevant to SAT solving are the data types that represent variables, literals and clauses. We will shortly discuss how these data types are implemented.

In satUZK, variables and literals are represented as unsigned 32-bit integers. Specifically, for a variable that is represented by the number k, the positive and negative literals are represented by $2k$ and $2k + 1$ respectively. Thus, the inverse of a literal can be computed by flipping its least significant bit. This representation was inspired by MiniSat [34]. Some other solvers implement positive and negative literals as positive and negative numbers respectively. Our representation has the advantage that it interacts nicely with our assignment data structure that is presented in the next section.

Clauses are represented by "handles" to their data (which includes their literals and other information). The representation of clause data is more complicated: satUZK implements multiple schemes to store clause data. We call these schemes *clause spaces* and will discuss them in subsections 3.2.6 and 3.2.7. Clause handles could be implemented as pointers into such clause space data structures. However, as the solver needs to manage a large amount of clause handles (e.g. as part of watched lists that we will discuss in subsection 3.2.4), we can reduce its memory consumption by implementing clause handles as 32-bit integers, too. The exact meaning of these integers depends on the clause space. On 64-bit systems (i.e. where pointers are 64-bit) that halves the memory footprint of clause handles. Furthermore, it improves the locality of the data structures and reduces cache pressure. On the flip side, implementing clause handles as 32-bit integers reduces the maximal number of clauses that can be addressed by the solver. The solver can be recompiled with 64-bit clause handles if 32-bit handles do not suffice to solve a given instance. However, we did not encounter such instances in practice, especially because exploiting CPU alignment constraints allows us to address 16 GiB of clauses using 32-bit handles (see subsection 3.2.6).

3.2.2 Assignment and trail data structures

During an iterative DPLL or CDCL search, a SAT solver needs to keep track of the current assignment and of the order in which literals are assigned. The latter is required to be able to backtrack after the solver encountered a conflict. The *assignment* and *trail* data structures manage this information.

Assignments are stored as an array that contains two bits per variable [7]. The more significant bit determines whether the variable is assigned (encoded as zero) or unassigned (encoded as one). The less significant bit is only valid while the variable is assigned and determines if the positive or negative literal is currently true. This encoding enables a low-overhead implementation of truth checks: Testing whether a certain literal is true amounts to comparing its least significant bit (i.e. the least significant bit

[7]Technically, these two bits are padded to a byte for performance reasons.

of its $2k/2k+1$ encoding) to the 2-bit value given by the assignment.

In addition to the truth value, the assignment data structure has to store, for each currently assigned variable, the decision level that the variable was assigned on and the antecedent of the variable. As we stated earlier, the antecedent of a variable is the clause that fixed the variable via unit propagation. However, in some algorithms, we need to manage antecedents that do not directly arise from clauses. Therefore, antecedents are managed through a dedicated data type: Formally, the ANTECEDENT type is a discriminated union that can assume values of the form DECISIONANTECEDENT, CLAUSEANTECEDENT(C) and UNARYANTECEDENT(c), where C is a clause and c is a literal. Here, CLAUSEANTECEDENT(C) is the antecedent of a variable that has been fixed by the clause C through unit propagation. UNARYANTECEDENT(c) indicates that the variable was fixed by an implication (i.e. an edge of the binary-implication graph or equivalently, a binary clause) after the literal $\bar{c}$ has been assigned. The solver uses UNARYANTECEDENT instead of CLAUSEANTECEDENT if the exact clause that propagated a variable is not known. DECISIONANTECEDENT is used to mark decision variables. If a variable is not assigned, we set its antecedent to *nil*.

The trail data structure consists of two stacks. The first stack stores all assigned literals in the order in which they are assigned. The second stack stores, for each decision level, the position of the decision literal on the first stack. We call those stacks the *assignment stack* and the *level stack* respectively.

The assignment and trail data structures implement the procedures NEWDECISIONLEVEL, ENQUEUE(c, λ) and BACKTRACK. The NEWDECISIONLEVEL operation just pushes the current size of the assignment stack onto the level stack. The ENQUEUE(c, λ) operation assigns c (while memorizing the current decision level and the antecedent λ) and pushes c onto the assignment stack. BACKTRACK pops the topmost item k from the level stack. It then pops literals from the assignment stack and unassigns them until only k items remain on the assignment stack.

3.2.3 Watch lists

In this subsection we present satUZK's "watch list" data structure that is responsible for detecting unit clauses and conflicts.

The idea behind this data structure is the *two-watched-literals* scheme: In order to detect if a clause is unit, the solver marks exactly two literals of each non-unary clause C. These literals are called the *watched literals* of C. The solver maintains the following invariants:

2-WL If C is neither unit nor unsatisfied, there are at least two literals in C which do not have value false (i.e. they can either be unassigned or have value true). Two of these literals are the watched literals and they are chosen from the maximal possible decision levels.

1-WL C is unit if there is exactly one literal in C which does not have value false. This literal is one of the watched literals and is assigned after all other literals in C. The other watched literal has value false and is chosen from the maximal possible decision level.

0-WL Otherwise C is unsatisfied and both watched literals have value false. These two literals are chosen from the current decision level.

The solver then stores a *watch list* for each literal c that stores exactly the clauses that contain c as a watched literal. The watch lists are updated during unit propagation. However, it should be noted that the data structure does not need to be updated during backtracking. This follows from the fact that all invariants of the two-watched-literals data structure remain satisfied after literals are unassigned, provided that they are unassigned in the reverse order in which they were assigned. As a result, backtracking during the CDCL algorithm is an inexpensive operation.

The two-watched-literals scheme was invented by Chaff [69]. It is lazy in the sense that assigning a literal does not immediately affect all clauses that contain this literal. One of the implications of using a lazy data structure is that it is not possible to determine the set of clauses of length k under the current assignment, for $k > 1$. Other lazy schemes to detect unit clauses have been discussed in [66], however, none of them turned out to outperform the two-watched-literals scheme in practice. Due to its performance advantage, all state-of-the-art CDCL solvers rely on this scheme to perform unit propagation. It should be noted that other SAT solvers, for example lookahead-based solvers, rely on *eager* data structures (e.g. modern versions of the one discussed in [24]) which are indeed able to determine all clauses of length k. Because of their irrelevance to CDCL, we will not discuss eager data structures here.

Binary entry
Type (0)
Blocking literal

General entry
Type (1)
Blocking literal
Clause handle

Figure 3.3: Layout of watch list entries
In the actual implementation, both the type and the blocking literal are stored in a single 32-bit word. This leaves only 31-bit (i.e. ≈ 2 billion possible values), for the blocking literal which is not a problem as instances typically do not contain more than 10 million variables.

The structure of watch list entries in satUZK is depicted in figure 3.3. The *blocking literal* of an entry is arbitrarily chosen from the clause that is referenced by the entry, with the constraint that it does not equal c for entries of the watch list of $\bar{c}$. Thus, for binary clauses $\{c, c'\}$, there is only a single choice for the blocking literal and that is the literal c' that will be propagated once $\bar{c}$ is assigned. The blocking literal mechanism of course duplicates information; after all, the literals of the clause are already accessible through the clause handle. The reason that we store the blocking literal is that it improves locality: In many cases the blocking literal allows us to determine that a clause is satisfied even without following the clause handle. In the case of binary clauses, it even allows us to perform propagation without looking at the clause.

3.2.4 Propagation

Unit propagation is the most performance critical operation in a CDCL-based SAT solver. Therefore, unit propagation is heavily optimized in state-of-the-art solvers. satUZK uses a unit propagation algorithm that takes caches and CPU branch prediction into account. In this subsection, we present the details of the algorithm.

In addition to the data structures that we already discussed so far, unit propagation has to deal with the CONFLICT data type that represents incoming edges of the $\perp$ vertex in conflict graphs (similar to how the ANTECEDENT type represents incoming edges of all vertices other than $\perp$). Similar to the ANTECEDENT type, it is a discriminated union that can assume values CLAUSECONFLICT(C) and BINARYCONFLICT(c, c'), where C is a clause and c, c' are literals [8].

CLAUSECONFLICT(C) represents a conflict that happens because the clause C is unsatisfied, while BINARYCONFLICT(c, c') represents a conflict that happens because both c and c' are false under the current assignment. Similar to the antecedent type, the solver might use BINARYCONFLICT in favor of CLAUSECONFLICT if the exact clause that caused the conflict is not known. As in the case of antecedents, we use the *nil* value to represent the fact that the current assignment is not conflicting.

Algorithm 3.2 states our unit propagation algorithm. The algorithm traverses the watch list of $\bar{c}$ via two pointers *begin* and *end*. At the same time, it rewrites the watch list through the *wit* pointer. With $*p$, we denote access to the element pointed to by p; this syntax is similar to the C

[8] This list does not contain UNARYCONFLICT(c). Such a value is not required to present our propagation algorithm as we assume that unary clauses are always assigned at decision level zero, before the propagation algorithm is called. It might still make sense to use UNARYCONFLICT(c) internally to represent unsatisfied unit clauses.

Algorithm 3.2 Optimized unit propagation algorithm

```
procedure PROPAGTELITERAL($\bar{a}$)
    $\xi \leftarrow nil$
    $begin \leftarrow$ pointer to first watch list entry of $\bar{a}$
    $end \leftarrow$ pointer to last watch list entry of $\bar{a}$
    $wit \leftarrow begin$
    for $rit$ between $begin$ and $end$ do
        $b \leftarrow (*rit)$.BLOCKING
        if likely ISTRUE($b$) then
            $*wit \leftarrow *rit$
            $wit \leftarrow wit + 1$
            continue
        end if
        if $(*rit)$.ISBINARY then
            $\lambda \leftarrow$ UNARYANTECEDENT($a$)
            $\epsilon \leftarrow$ BINARYWATCHLISTENTRY($b$)
            if unlikely ISASSIGNED($b$) then
                $\xi \leftarrow$ BINARYCONFLICT($a, b$)
                break
            else
                ENQUEUE($b, \lambda$)
                $*wit \leftarrow \epsilon$
                $wit \leftarrow wit + 1$
            end if
        else
            $C \leftarrow (*rit)$.CLAUSE
            $\lambda \leftarrow$ CLAUSEANTECEDENT($C$)
            if GETWATCH1($C$) $= a$ then
                SWAPWATCHES($C$)
            end if
            $b' \leftarrow$ GETWATCH1($C$)
            $\epsilon \leftarrow$ CLAUSEWATCHLISTENTRY($b', C$)
            if ISTRUE($b'$) then
                $*wit \leftarrow \epsilon$
                $wit \leftarrow wit + 1$
                continue
            end if
            $w \leftarrow$ first literal of $C$ that is not false
            if unlikely no such literal $w$ exists then
                $*wit \leftarrow \epsilon$
                $wit \leftarrow wit + 1$
                if unlikely ISASSIGNED($b'$) then
                    $\xi \leftarrow$ CLAUSECONFLICT($C$)
                    break
                else
                    ENQUEUE($b', \lambda$)
                end if
            else
                UPDATEWATCH2($C, w$)
                INSERTINTOWATCHLIST($\bar{w}, \epsilon$)
            end if
        end if
    end for
    $wit \leftarrow$ copy the range between $rit$ and $end$ to $wit$
    RESIZEWATCHLIST($\bar{a}, wit - begin$)
    return $\xi$
end procedure
```

programming language. Futhermore, we denote branches that are likely to be taken with **if likely** and branches that are not likely to be taken with **if unlikely**. In our C++ implementation, these annotations give hints to the branch-prediction unit of the processor and thus improve performance.

The first thing that the algorithm does is checking if the blocking literal is true; this operation allows us to ignore many watch list entries without further work. In line 13, we handle watch list entries of binary clauses. Binary clauses that are part of the watch list that is currently propagated are always unit or conflicting. If they are indeed unit, they propagate the block literal. Since we already know that the blocking literal is not true, we do not have to check if it is already true after the check in line 16.

Non-binary clauses are handled by the block starting in line 24. Here, we order the watched literal so that c is the second watched literal. This is the first memory access to the clause's data and thus pulls the clause data into cache. The first watched literal might already be true, in which case we can ignore the watch list entry. This is checked in line 32. If the watched literal is not true, we try to find a new non-false watched literal. If such a literal exists, we use it as a new watched literal in line 47. The clause might still be unit; however, it will be detected during traversal of the other watch list. If no non-false literal exists, we enter the block in line 38. Here, we check if the first watched literal is assigned. If that is the case, it has to be false, as we already checked if it is true before. Thus, we either found a conflict or propagate the first watched literal.

3.2.5 Conflict analysis

Conflict analysis can be efficiently implemented using the data structures that we already discussed. In this subsection, we will review how this is done. Most ideas in this subsection are taken from MiniSat [34].

In particular, it is not necessary to construct antecedent graphs explicitely as the trail data structure already gives us a topological ordering $a_1, \dots, a_k, a_{k+1} = \perp$ of the current antecedent graph that ends in the $\perp$ vertex (so that there can only be an edge between a_i and a_j if $i < j$). UIPs on the current decision level are exactly the vertices in this ordering that are not crossed by any edge between vertices on the current decision level. Thus, we can easily extract the first UIP by reversely iterating through the trail, while only considering vertices that are reachable by the $\perp$ vertex and stopping when no such edge exists.

Algorithm 3.3 depicts the pseudocode of this algorithm. The loop in line 19 iterates over the literals in the reverse topological ordering $a_k, \dots, a_1$. During iteration, the counter n counts the number of vertices that have smaller indices than the current vertex in the topological ordering and that

Algorithm 3.3 Conflict clause extraction

```
procedure EXTRACTLEARNEDCLAUSE
    $C \leftarrow \varnothing$
    $n \leftarrow 0$
    procedure VISIT($c$)
        if $c$ is already marked then
            return
        end if
        mark $a$
        if DECISIONLEVEL($c$) = CURRENTLEVEL then
            $n \leftarrow n + 1$
        else
            $C \leftarrow C \cup \{\bar{c}\}$
        end if
    end procedure
    $\kappa \leftarrow$ CONFLICT
    for $a \in$ REASONS($\kappa$) do
        VISIT($a$)
    end for
    for literals $a$ that are part of the trail, in reverse order do
        if $a$ is not marked then
            continue
        end if
        $n \leftarrow n - 1$
        if $n \neq 0$ then
            $\lambda \leftarrow$ ANTECEDENT($a$)
            for $c \in$ REASONS($\lambda$) do
                VISIT($c$)
            end for
        else
            $C \leftarrow C \cup \{\bar{a}\}$
            return $C$
        end if
    end for
end procedure
```

have edges crossing the current vertex. Per definition, if this number drops to zero, the currently visited vertex is a UIP. For each literal, the conflict graph is traversed by calling the VISIT procedure on the literals in its antecedent. VISIT increments n as necessary and records variables from earlier decision levels that will be part of the 1-UIP clause C. The VISIT procedure marks vertices in order to avoid counting them twice. Finally, once a UIP is detected (per construction, this will be the first UIP on the current decision level) its inverse is added to the conflict clause C. The algorithm terminates and that clause is returned.

Note that because the algorithm terminates once a UIP on the current decision level is found, it actually only iterates over literals on the current decision level and not over all literals in the conflict graph. If learning schemes other than 1-UIP were used, the conflict clause extraction algorithm would have to iterate over larger portions of the conflict graph.

After the initial learned clause has been extracted, satUZK performs

Algorithm 3.4 Conflict clause minimization

```
1:  $r(c) \leftarrow \mathsf{unknown}\ \forall$ literals $c$
2:  procedure MINIMIZELEARNEDCLAUSE($a, C$)
3:      $S$ : Stack
4:      $\lambda \leftarrow$ ANTECEDENT($a$)
5:      if $\lambda =$ DECISIONANTECEDENT then
6:          return false
7:      end if
8:      for $d \in$ REASONS($\lambda$) do
9:          $S$.PUSH(($d$, false))
10:     end for
11:     while not $S$.EMPTY do
12:         ($c, p$) $\leftarrow$ $S$.POP
13:         if not $p$ then
14:             if $r(c) =$ true then
15:                 continue
16:             end if
17:             $S$.PUSH(($c$, true))
18:             if $a \in C$ then
19:                 continue
20:             end if
21:             $\lambda \leftarrow$ ANTECEDENT($c$)
22:             if $r(c) =$ false or $\lambda =$ DECISIONANTECEDENT then
23:                 for ($c'$, true) $\in S$ do
24:                     $r(c') \leftarrow$ false
25:                 end for
26:                 return false
27:             else
28:                 for $d \in$ REASONS($\lambda$) do
29:                     $S$.PUSH(($d$, false))
30:                 end for
31:             end if
32:         else
33:             $r(c) \leftarrow$ true
34:         end if
35:     end while
36:     return true
37: end procedure
```

recursive learned clause minimization. In order to prevent stack overflows, we actually implement this algorithm iteratively. Algorithm 3.4 shows the procedure that checks if a given literal a can be removed from the conflict clause C by recursive minimization.

Call a literal c *implied* if either $c \in C$ or if the literals of the antecedent of c are all recursively implied. Recall that a can be removed by minimization if all literals of the antecedent of a are implied. We use a stack to perform the recursive implication check and cache both positive and negative results using the variable r. The stack stores items of the form (c, p), where c is a literal and p is a Boolean value that determines whether c has already been expanded (i.e. whether the literals of the antecedent of c have already been pushed onto S). The algorithm expands the literals on S in a depth-first order. Note that literals can occur multiple times on S (if it occurs in the antecedent of multiple expanded literals) but the caching mechanism prevents literals from being expanded multiple times. Thus, the number of items that the algorithm processes is bounded by the number of edges in the conflict graph.

At the start of the algorithm, the literals of the antecedent of a are pushed onto S. Once all these literals are processed and marked as implied by the algorithm, we know that a can be removed by recursive minimization. The algorithm iteratively removes literals c from S. If c is part of C, it is implied. Because of the check in line 18, it is not expanded. Instead, it is cached as implied in line 33. Otherwise, if we reached a decision literal c, the literal cannot be implied. In this case, we also know that all successors of c are not implied. Thus, the loop in line 23 marks all of them as non-implied and returns. In this case, a cannot be removed by recursive minimization. After the whole stack has been emptied, the algorithm returns that a can indeed be removed from C.

3.2.6 Compact clause representation

In this section, we will discuss how the satUZK solver stores and manages its clause data. This section will discuss a “compact” clause representation, while the next section presents an “indexed” representation. The compact layout is generally faster and uses less memory, but it is also less flexible. Furthermore, unit propagation needs to modify clauses in-place when the compact layout is used, rendering this layout unsuitable for algorithms that want to share clause spaces between different threads.

The data that we need to store for each clause includes its length, its literals, some flags (e.g. if the clause is redundant or irredundant), its LBD and activity scores and some other information that is required for algorithms that we want to run in addition to CDCL (like the CNF simplification techniques that will be discussed in chapter 5) [9].

In the *compact* clause representation, all data that belongs to a clause is stored in a single chunk of memory. In order to reduce the amount of memory that this clause data consumes, we divide the clause space into "permanent" and "reducible" clauses. For *permanent* clauses, we do not store the LBD and activity values of the clause. We do store these values for *reducible* clauses. This division implies that clause database reduction heuristics cannot evaluate permanent clauses; thus, those clauses are never removed by clause database reduction. Note that permanent clauses are not necessarily essential (and reducible clauses are not necessarily non-essential) and can still be removed by other means (e.g. when a CNF simplification technique decides that they are not useful). In a typical satUZK configuration, this reduces the size of permanent clauses by 16 bytes [10]; a non-permanent binary clause may consume as little as 32 bytes. Clause database reduction heuristics do not remove binary clauses anyway (see subsection 3.1.5), so storing them as permanent clauses has no drawbacks. As a large fraction of the clauses of industrial CNF formulas are binary, this optimization can greatly reduce memory consumption.

Figure 3.4 depicts the layout of clauses in compact representation. We use a flag to determine if a clause is permanent or reducible. Algorithms need to test this flag before they access fields that are not shared by both layouts. Note that some fields are stored at negative offsets from pointers into the data structure (highlighted by the "Handle" arrow in figure 3.4). The layout is chosen so that fields that are traversed during unit propagation are stored at positive offsets, while the other fields (which are accessed less frequently) are stored at negative offsets. This design minimizes cache misses during unit propagation. The use of negative offsets is inspired by the SAT solver Splatz [19].

As discussed in section 3.2.3, clause spaces have to mark two literals of each clause as watched literals. In the compact clause representation, we do not explicitly mark those literals. Instead, the first and the second literal of each clause correspond to the watched literals of the clause. Thus, clauses have to be permutated during unit propagation.

A naive way of managing the clause data would be using the system memory allocator to allocate each clause individually. However, the overhead of this approach is not acceptable: General purpose memory managers usually require expensive updates of internal data structures to perform

[9]For example, some simplification algorithms store a 64-bit bloom filter for checking if a clause is a subset of another clause. Some parallel algorithms store a globally unique "clause name" to identify clauses originating from different threads.

[10]This reduction comes from the fact that activity values are stored as double-percision floating point numbers and require an alignment of 8 bytes.

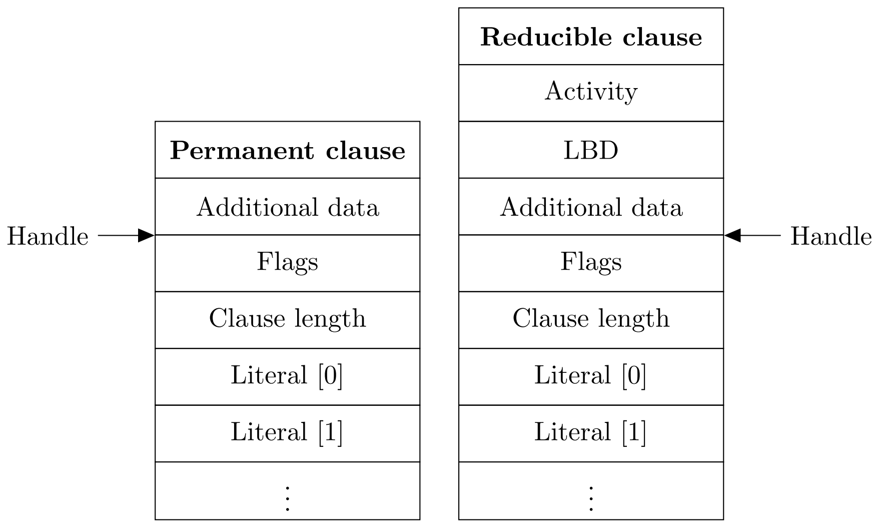


Figure 3.4: Compact representation: Clause layout
Permanent and reducible clauses are stored in different layouts. "Handle" indices where clause handles point to. The flags field contains a bit that determines if the clause is permanent or reducible. Additional data fields may be needed for some algorithms.

allocation and deallocation. Furthermore, they may need to attach auxiliary information to each of the memory chunks they return, increasing the amount of memory that the system consumes. Most allocators also require padding to ensure that the allocation size satisfies alignment constraints (e.g. the allocator might only hand out 2^k-sized chunks with $k \in \mathbb{N}$).

In order to avoid these overheads, we store all clauses in a contiguous block of memory. This idea is taken from the Chaff [69] and MiniSat [34] solvers. Clause handles are implemented as 32-bit offsets into this block [11]. Allocating a new clause thus simplifies to just incrementing a pointer into the memory block. If there is no space left (i.e. the whole block of memory is occupied by clauses), we allocate a new block of memory and copy all clauses into it. As clause handles are relative to the start of the block, they do not change as result of this operation.

However, this strategy prevents us from deleting clauses in the middle of the memory block. Instead, we mark clauses as deleted and clean up the whole memory block once a significant amount of clauses is deleted. We call this operation *garbage collection*. Garbage collection does indeed change clause handles; all clause handles in watch lists and other data structures have to be updated afterwards.

[11]Technically speaking, the offset is constructed by shifting the 32-bit handle by two bits to the left (i.e. multiplying the handle by four). This takes advantage of the fact that clause data is aligned to four bytes on most CPU architectures as it contains 32-bit integers. The trick enables the solver to address 16 GiB of clause data instead of 4 GiB. On the x86 architecture the shift is almost free, as most instructions support memory operands of the form $r_1 + (r_2 \ll c)$ where r_1 and r_2 are registers and c is a constant.

3.2.7 Indexed clause representation

In our second subsection regarding clause spaces, we present our indexed clause representation.

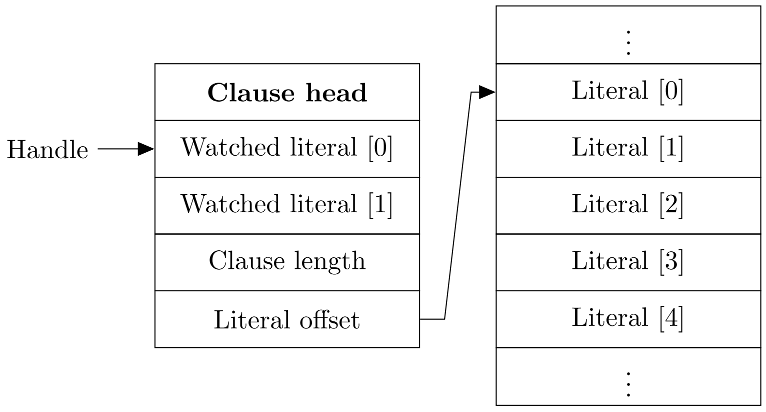


Figure 3.5: Indexed representation: Clause layout
The unit propagation algorithm uses the clause head to keep track of watched literals. The remaining literals are stored in a separate array.

Instead of storing clauses compactly in a single chunk of memory, we assign sequentially increasing integers to the clauses. These integers form the clause handles of the indexed clause space. We store the actual clause data in arrays that are indexed by clause handles.

Literals are a special case here, as clauses have variable numbers of literals. To accommodate for this, we store the literals of all clauses in a single array with each clause occurring contiguously in this array. For each clause we store its offset into the literal array seperately. Note that this design still forces us to perform garbage collection to compact the array of literals.

Furthermore, we group all propagation-related information about a clause in a single structure that we call the *clause head*. This improves locality and prevents cache-misses when clauses are accessed during unit propagation. Figure 3.5 depicts the clause head structure. Compared to the compact clause representation, we can expect that accessing the literals of a clause through its head incurs one additional cache-miss.

Nevertheless, as we store watched literals explicitly in the indexed representation, clauses do not need to be permuted during unit propagation. This enables the indexed representation to be shared between multiple threads. Furthermore, algorithms that extend CDCL can easily store clause-specific information in arrays as each clause has a sequentially increasing handle.

3.2.8 Occurrence lists

We quickly discuss a data structure that allows the solver do find all clauses that contain a certain literal.

When we review the data structures that we discuss so far, it is obvious that this information cannot be obtained from any of them without performing a linear search over all clauses. In fact, such a data structure is not necessary to run the CDCL algorithm. Nonetheless, some CNF simplification techniques depend on this information. In order to run these simplification techniques, we need to manage *occurrence lists*: For each literal, we store a list of clauses in which this literal occurs. It should be noted, that these lists are not updated when literals are assigned; the state represented by the occurence lists corresponds to the empty assignment.

Managing occurrence lists for learned clauses incurs a non-negligible performance cost. Whenever a new clause is allocated, we need to add it to multiple occurrence lists and after garbage collection, all occurrence lists need to be rebuilt. Furthermore, occurrence lists can consume large amounts of memory: For each literal, an additional clause handle needs to be stored, effectively doubling the memory footprint of literls.

For these reasons, the sequential satUZK solver only maintains occurrence lists during preprocessing. However, some extensions of the algorithm may require maintaining occurrence lists even during search; for example, some lookahead evaluation heuristics that we discuss in chapter 4 require occurrence lists.

3.2.9 Choice of heuristics

We shortly discuss the choice of heuristics in satUZK.

The current version of satUZK does not feature its own heuristics for decisions, clause database reduction and restarts [12]. Instead it relies on well-known heuristics. As a decision heuristic, we use VSIDS in all configurations. For clause database reduction and restarts, two *emulation modes* are implemented: MiniSat and Glucose emulation.

In MiniSat-emulation mode, the solver uses a geometric clause database reduction heuristic and Luby restarts. In Glucose-emulation mode, an aggressive, linear clause database reduction heuristic based on LBD and a dynamic restart heuristic is used.

One benefit of emulating the heuristics of other solvers is that optimization and debugging of solvers is greatly simplified. For example, the emulation modes of satUZK are accurate enough to compare propagation

[12]Versions of satUZK that participated at previous SAT Competitions did feature their own heuristics, see [96, 90, 89].

speed and the numbers of conflicts per instance (both of which vary heavily among different reduction and restart heuristics) of satUZK and the emulated solver.

3.2.10 C++ implementation details

While we presented most of satUZK's algorithms in a language-agnostic way, this subsection will discuss some details of satUZK's C++ implementation.

A unique feature of satUZK is that it is implemented in a modular way and many parts of the implementation can be modified by inserting code into well-defined hooks. For example, this mechanism is used to switch between different clause spaces [13]. Because dynamic polymorphism is too expensive to be used in the performance-critical parts of the solver, we use static polymorphism via C++ templates instead. The entire solver is implemented as a collection of templates that are compiled by a single invocation of the C++ compiler. This means that the solver has to be recompiled to modify hooks; this is unavoidable to achieve acceptable performance. Nevertheless, the template mechanism allows multiple instances of the solver that differ in their configurations in the same process [14]. This is not possible for manual configuration mechanisms like C `typedef`s and `#ifdef`s.

3.3 Extensions to CDCL

This section discusses some techniques that are not part of the core CDCL framework but that augment the CDCL algorithm in various ways.

3.3.1 Incremental SAT solving

Incremental SAT solving is the process of solving similar CNF formulas successively via multiple calls to the CDCL algorithm. This subsection will discuss techniques that speed up incremental calls to a CDCL solver.

Typically, between each of those incremental calls, clauses are added and/or removed from the formula. If a CDCL solver remembers its learned

[13]It is also used to parameterize the unit propagation procedure for the hyper binary resolution and distillation algorithms that we discuss in chapter 5.

[14]For example, this technique was used in the satUZK-par solver [89] that was submitted to SAT Competition 2014. satUZK-par was a parallel shared memory solver that used a dedicated thread to strengthen learned clauses along with traditional CDCL threads. The solver instances in both threads were based on different template instantiations. Hence, both instances of the algorithm could be adapted to their use cases without sacrificing performance of the other instance. We do not discuss satUZK-par in detail here, as we consider it inferior to the satUZK-ddc algorithm that we discuss in chapter 4

clauses between calls, incrementally solving a sequence of formulas is often much faster than solving each formula independently.

Adding new clauses before each call is not a problem for the CDCL algorithm. At the worst, new clauses can invalidate models that have already been found by the solver but they do not impact the validity of clauses that have already been learned. However, care must be taken to ensure that learned clauses are still valid after some of the original input clauses have been removed. If a set ψ of clauses that might eventually be removed is known in advance, this problem can be solved by allocating a new variable x, adding $\bar{x}$ to all clauses of ψ and forcing the solver to always assign x before performing other decisions. Using this scheme, all learned clauses that depend on clauses in ψ will also contain the literal $\bar{x}$. When the clauses from ψ are removed from the formula, all learned clauses containing $\bar{x}$ also need to be removed. This is equivalent to assigning the literal $\bar{x}$ instead of x; per construction, the latter literal does not even occur in the CNF formula.

Note that using this idea it is possible to remove individual clauses as well as sets of clauses from the formula. In practice, however, we want to keep the number of additional variables low, as each of those variables is likely to show up in learned clauses. This increases their length and thus reduces the efficiency of unit propagation. Luckily, in many applications, we want to remove sets of clauses, and not individual clauses anyway. For example, this is the case with [43] and with the distributed algorithm that we present in chapter 4.

In order to implement the aforementioned concept, we need to be able to force the CDCL solver to assign a certain literal before any decisions are done. An “assumption” mechanism [35] can be used to do that. A sequence of literals $(a_1, \dots, a_k)$ is designated as the current sequence of *assumptions*. Whenever the solver needs to perform a decision, it first checks if the current decision level l is smaller or equal to k. If that is the case, no decision is done and a_l is chosen as a decision literal instead.

satUZK (similar to MiniSat [34]) implements such an assumption mechanism. Subsection 4.3.1 of the next chapter discusses the interaction between this assumption mechanism and the LBD score.

3.3.2 Unsatisfiability proofs

In this subsection we discuss suitable formats for certificates of unsatisfiability.

As a certificate for satisfiablility, most SAT solvers can trivially output a model of a formula once they prove this formula to be satisfiable. However, in many cases we are also interested in certificates for unsatisfiability. In other words, given an unsatisfiable CNF formula ϕ, we want an efficiently checkable proof that ϕ is indeed unsatisfiable.

It is an open problem if short unsatisfiability proofs for CNF formulas exist. In particular, UNSAT is *coNP*-complete. If there is a proof system that can prove the unsatisfiability of an arbitrary CNF formula using only a polynomial number of steps, then the classes NP and $coNP$ coincide [27].

Lemma 3.4 states that clause learning can be translated to resolution, so it is possible to modify a CDCL solver to output a resolution proof for unsatisfiable CNF formulas. Early SAT Competitions, starting from 2005, indeed used a resolution based proof system for unsatifiability certificates [78]. However, resolution proofs are both large (see 2.2.6) and inefficient to validate.

A much more convenient proof system for CDCL solvers is the *reverse unit propagation* (RUP) system. RUP unsatisfiability proofs consist of a sequence of clauses $(C_1, C_2, \ldots, C_\ell = \varnothing)$ so that applying unit propagation to the assignment $\{\bar{a} : a \in C_i\}$ on the formula $\phi \cup \{C_1, \ldots, C_{i-1}\}$ leads to a conflict, for all $i = 1, \ldots, \ell$. Specifically, this property is true for all learned clauses that are produced during CDCL. Therefore, a CDCL solver can generate a RUP proof simply by outputting all its learned clauses.

It is easy to see that the RUP proof system is sound; this will be discussed in detail later in subsection 5.1.2 of chapter 5. Because of its simplicity, the SAT Competition switched to the RUP format in 2012 [78]. RUP has been extended to the DRUP format which also stores clause deletion information in order to improve the performance of a proof verifier. Other extensions include the introduction of RAT clauses [93] which can be used to model some simplification techniques that cannot be modeled via RUP clauses. RAT clauses will be discussed in chapter 5. Another extension to RUP is the introduction of even stronger clausal proofs in [48, 49].

satUZK provides RUP based unsatisfiability proofs. All of satUZKs simplification techniques can be simulated via RUP and RAT.

3.4 Experimental evaluation

In this section, we will experimentally evaluate the performance of satUZK. In a first experiment, we compare satUZK-seq's CDCL implementation to other state-of-the-art CDCL implementations. After that, we evaluate the effectiveness of its heuristics among different families of instances. Finally, we study the performance impact of certain data structures and of extensions to the CDCL algorithm.

It should be noted that we do not compare satUZK-seq with the state of the art of sequential SAT solvers here. As the primary objects of interest in this dissertation are the parallel algorithms of chapters 4 and 5, chasing the latest sequential state of the art is not absolutely necessary for us. While the implementation of CDCL in satUZK is competitive against the state of the art, it does not implement sophisticated inprocessing (like in Lingeling [15]) or the very successful LRB heuristic[15] that was discovered in the last year. satUZK-seq is, however, competitive with state-of-the-art solvers when these techniques are disabled.

All experiments that we discuss in this section ran on a cluster of dual-socket Intel Xeon E5-2690 v2 nodes with 128 GiB of RAM per node. The processors run at a reference frequency of 3.0 GHz. The cluster runs on Linux, using the Debian Stretch distribution. We used GCC 6.2.0 to build all solvers.

3.4.1 Performance characteristics

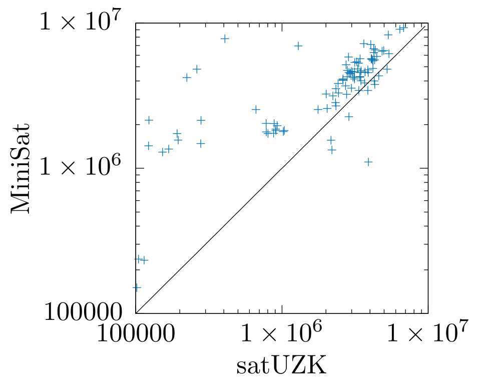
In our first experiment, we compare the quality of our CDCL implementation with that of MiniSat and Glucose. While MiniSat is not able to solve as many instances as state-of-the-art solvers, it still has one of the fastest CDCL engines and is often used as a basis for other solvers (e.g. Glucose, Syrup [10], CryptoMiniSat [80] and the Maple family [64]). Glucose is a state-of-the-art solver that won five gold medals at SAT Competitions since 2009 [78]. As we want to evaluate the performance of the CDCL implementation, we disabled Glucose's adaption heuristics that tune its behavior to the input CNF formula. In order to obtain meaningful results, we compare satUZK-seq's MiniSat-emulation with MiniSat and its Glucose-emulation to Glucose.

We use MiniSat version 2.2 and Glucose version 4.1 and the version of satUZK that was submitted to SAT Competition 2017[16]. As benchmarks, we use all 350 instances from the Main track of this competition. These instances include industrial applications and combinatorial problems. In order to evaluate the performance of the CDCL implementations, we compare the number of unit propagations per second and the number of conflicts per second that the solvers are able to handle. We ran all solvers with a timeout of one hour. For our comparison we only use those instances that could be solved both by satUZK-seq and the other solver.

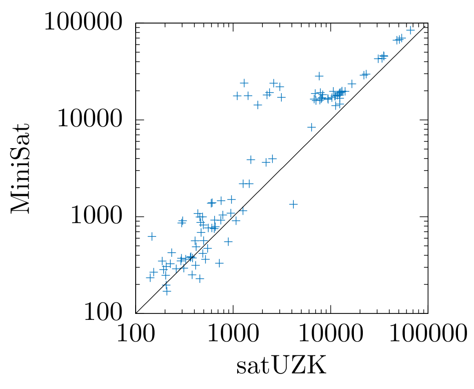
Figures 3.6 and 3.7 depicts the performance characteristics of the solvers. Both emulation modes seem to mimic the characteristics of the original solvers quite well. MiniSat seems to be slightly faster than satUZK-seq, which might be due to the fact that MiniSat is much smaller and thus its

[15]The Maple family of SAT solvers [64] that implements LRB solved 20 more instances than any other solver at SAT Competition 2017; this is a huge improvement over the previous state of the art.

[16]Specificially, the version differs from the SAT Competition 2017 only by minor bugfixes.



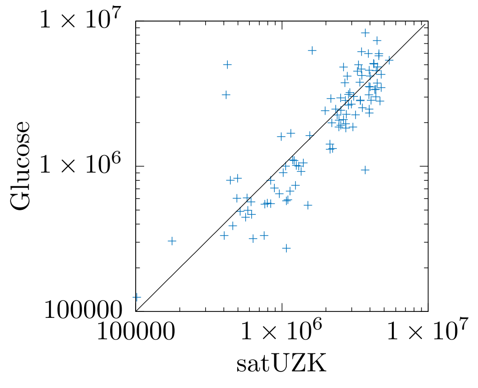
(a) Propagations per second



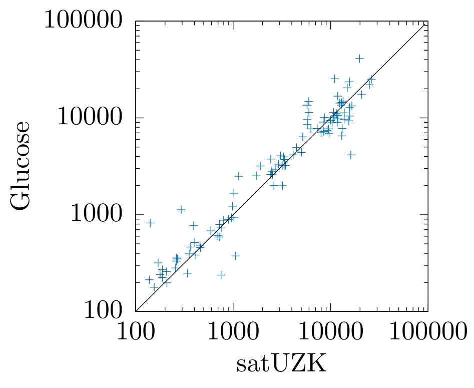
(b) Conflicts per second

Figure 3.6: Performance of satUZK vs. MiniSat

Compares MiniSat with satUZK's MiniSat-emulation. Each point corresponds to one instance of the SAT Competition 2017 benchmark set. Only instances that are solved both by satUZK-seq and by MiniSat are reported.



(a) Propagations per second



(b) Conflicts per second

Figure 3.7: Performance of satUZK vs. Glucose

The same experimental design as in figure 3.6. Here, Glucose is compared with satUZK's Glucose-emulation.

Table 3.1: Effectiveness of heuristics on SC'17 families

The "Uniq." columns give the number of instances that could only be solved in the specific configuration and not by the other set of heuristics.

		MiniSat-emu.		Glucose-emu.	
Family	Inst.	Solved	Uniq.	Solved	Uniq.
Integer prefix	59	**22 (37%)**	10	18 (31%)	6
`ACG`	3	**3 (100%)**	1	2 (67%)	0
`ak128`	30	10 (33%)	0	10 (33%)	0
`blockpuzzle`	20	14 (70%)	0	**20 (100%)**	6
`bsat`	4	0 (0%)	0	**3 (75%)**	3
`gss`	10	0 (0%)	0	0 (0%)	0
`hwmcc`	41	1 (2%)	0	**5 (12%)**	4
`klieber`	20	6 (30%)	0	**11 (55%)**	5
`mizh`	4	4 (100%)	0	4 (100%)	0
`modgen`	5	**2 (40%)**	2	1 (20%)	1
`Nb`	19	2 (11%)	0	2 (11%)	0
`ps`	40	5 (12%)	0	**6 (15%)**	1
`rubikcube`	20	7 (35%)	0	**8 (40%)**	1
`slp-synthesis`	6	**2 (33%)**	2	0 (0%)	0
`squ`	10	5 (50%)	0	**10 (100%)**	5
`T`	40	18 (45%)	0	**21 (52%)**	3
`tri`	5	5 (100%)	0	5 (100%)	0
`UCG`	3	2 (67%)	0	**3 (100%)**	1
Uncategorized	11	7 (64%)	0	**10 (91%)**	3
Sum	350	115	15	139	39

code exhibits better caching behavior. satUZK-seq's performance generally matches that of Glucose.

3.4.2 Effectiveness of heuristics

In the next experiment, we evaluate the differences between MiniSat- and Glucose-emulation. In particular, we are interested in determining which families of instances are best solved by which heuristic. This information will be useful when we design our parallel solver that can potentially use different sets of heuristics in different threads. We ran satUZK-seq both in MiniSat- and in Glucose-emulation mode with a timeout of 5000 seconds. We use the same set of benchmarks as in our previous experiment.

In table 3.1 we report the numbers of instances per instance family that could be solved by each emulation mode. Here, an instance family is defined by taking the common prefixes of the SAT Competition 2017 file names. If a prefix occurred less than three times, we put it into the *uncategorized* family.

Overall, Glucose-emulation outperforms MiniSat-emulation. However, MiniSat-emulation is still able to solve 15 instances that Glucose-emulation cannot solve. In particular, MiniSat seems to be strong on `ACG`, `modgen` and `slp-synthesis` instances and on the family of instances with an integer

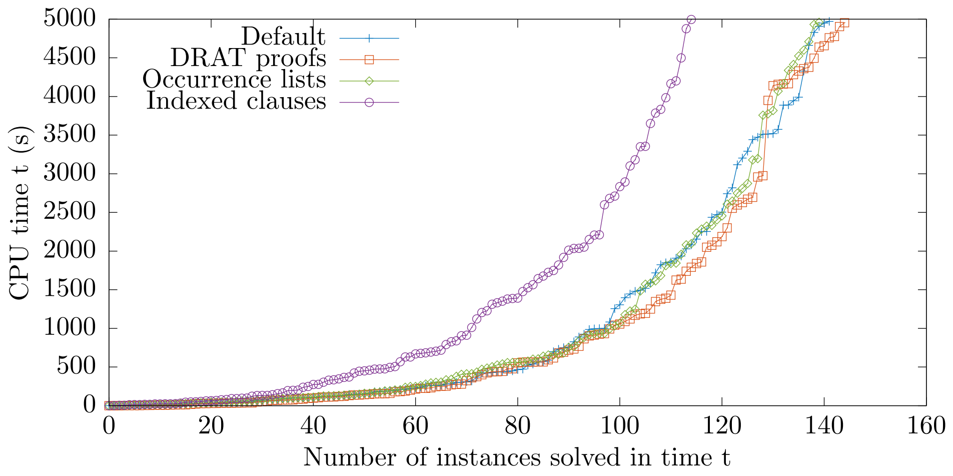


Figure 3.8: Extensions of satUZK-seq: Performance impact

prefix.

3.4.3 Performance impact of data structures and extensions

In our last experiment regarding our sequential solver, we study the performance impact of DRAT proofs, the maintenance of occurrence lists and the indexed clause space.

We enable Glucose-emulation for this experiment, use the SAT Competition 2017 instances as benchmarks and set a timeout of 5000 seconds. Proofs are generated in the binary DRAT format [93] but discarded to `/dev/null`.

The results are depicted in figure 3.8 in the form of a cactus plot. Fortunately, the generation of DRAT proofs and the maintenance of occurrence lists do not seem to affect the performance of satUZK-seq negatively. For proof generation, however, one has to keep in mind that in applications the proof has to be fed into a verifier. This has a non-zero cost which depends on the implementation quality of the verifier and on the I/O performance of the operating system. This cost is not represented in our plot.

On the other hand, the indexed clause representation has a huge negative impact on satUZK's performance. We conjecture that this is not due to the slightly worse caching behavior of the data structure (when compared to the compact clause representation) but due to the fact that reordering the clauses is actually a huge performance advantage in the compact representation. Hence, we conclude that sharing the clause space among different threads is not feasible in practice.

Chapter 4

Distributed Divide-and-Conquer

In this chapter we present our distributed SAT algorithm. First we discuss multiple approaches on parallel SAT solving in section 4.1. The section will also give an overview of the previous work on parallel SAT solvers. In section 4.2 we present the basic idea of our novel "Distributed Divide-and-Conquer" (DDC) algorithm. Section 4.3 will discuss the satUZK-ddc implementation of this algorithm as well as some refinements of it. At the end of the chapter, we will give some experimental results in section 4.4.

There is some overlap between this chapter and our DDC paper [88]. However, this dissertation describes the DDC algorithm and its extensions in more detail. We also include an extensive evaluation of variants of the algorithm that is not present in the paper. satUZK-ddc was submitted to SAT Competition 2017.

4.1 Approaches to parallel SAT solving

In the following, we will discuss different approaches to parallel SAT solving. This will help us to categorize existing solvers and introduce important techniques that will be reused later in our Distributed Divide-and-Conquer (DDC) solver.

4.1.1 Portfolio solvers

We start by discussing the "portfolio" approach to parallel SAT solving. Conceptually, this is the most simple parallel solving approach and it also requires the least modifications to a sequential CDCL solver.

A *portfolio solver* incorporates multiple different SAT solving strategies in order to exploit the fact that no single strategy is optimal on all instances.

We call these strategies *engines*. The solving process ends as soon as the first engine finds a solution or proves the unsatisfiability of the formula. If this happens, all other engines are stopped. In a (parallel) portfolio, typically all engines are run in parallel. Alternatively, it would be possible to enable only a subset of all engines on a per-instance basis. There are sequential portfolio solvers (e.g. [97]) based on this idea; however, we are more interested in the parallelization here.

ManySAT [40] was the first notable parallel portfolio solver. Its engines are based on a CDCL algorithm and use different restart and decision heuristics and different clause learning schemes. In addition to that the solver utilizes *clause sharing*: Engines exchange some of the learned clauses which they produce during the CDCL algorithm. These additional clauses reduce the search space that the other engines have to explore. ManySAT shares clauses with length smaller or equal to 8. Most of the later portfolio solvers incorporate some form of clause sharing; however, most of them use more sophisticated heuristics to decide whether a given clause should be exchanged with other engines. ManySAT won the parallel track of the SAT 2009 competition [78].

A problem of clause sharing is that clauses which are imported from other workers might negatively interact with the VSIDS heuristics by encouraging the workers to explore the same search space as the solver that produced the clauses. Therefore, the successful Plingeling solver [15, 16] uses a minimalistic clause sharing strategy: It only shares unit clauses and literal equivalences (i.e. clause sets of the form $\{\{\bar{a}, b\}, \{a, \bar{b}\}\}$, corresponding to the implications $a \to b$ and $b \to a$) between worker threads.

Rather than restricting clause exchange, the PeneLoPe solver [4] uses clause freezing [5] to prevent imported clauses from disturbing the CDCL search. Instead of immediately adding imported clauses to the watch lists, *clause freezing* only allocates the clauses first, without traversing them during unit propagation yet. Frozen clauses are *activated* later when they become relevant to the search. To judge whether a clause is relevent to the search, PeneLoPe utilizes the dynamic "*psm*" score of a clause.

Definition 4.1. *For a clause C, the* progress-saving measure $psm(C)$ *at a given point of time during a run of a CDCL solver is defined as the number of literals of C that are true under the total assignment that is given by progress-saving.*

Note that if $psm(C)$ is zero, the clause is conflicting under the progress-saving assignment; clauses with $psm(C)$ equal to one are unit under the progress-saving assignment. Furthermore, because CDCL solvers always assign variables to the values given by progress-saving, it is likely that this assignment will indeed arise. PeneLoPe therefore unfreezes clauses once their *psm* falls below a fixed constant.

The successful Syrup solver [10] uses a lazy clause export scheme. Clauses are not exported when they are generated. Instead, the solver only exports a clause after it appeared in conflict analysis at least once. On the import side, Syrup also performs clause freezing. However, it does not use *psm* but rather activates a clause once that clause causes a conflict. Clauses that cause a conflict can cheaply be detected by a single watched literal scheme.

All of the afforementioned solvers are shared-memory solvers. HordeSat is a portfolio solver [11] that runs in a distributed environment instead. HordeSat is a parallel interface to MiniSat [34] or Lingeling [15] and exchanges learned clauses via MPI. The authors of HordeSat report a decent speedup on large instances for up to 2048 cores.

It should be noted that there are inherent limits on the effectiveness of clause sharing. In particular, CDCL solvers are limited by the depth of resolution proofs that they produce, regardless of the amount of clause sharing in a solver [59].

In contrast to these sophisticated solvers, the ppfolio [76] solver just runs a collection of multiple different SAT engines in parallel and does not employ any form of communication. Its engines consist of separate SAT solvers that have proven to be efficient in previous SAT competitions. Despite the lack of communication, ppfolio won in the wall clock ranking of the "Application category", SAT instances only track at the SAT Competition 2011 [78]. A version of ppfolio that incorparates our sequential satUZK solver won the "Parallel Track" of the SAT Challenge 2012 [78].

Even though ppfolio was very successful at the SAT Competition, the value of its approach is questionable. While using different non-communicating SAT engines certainly helps to solve problems that are randomly selected from a large pool of instances, it does not help to solve families of SAT instances for which a best sequential solver is known. In industrial applications of SAT solving, this is often the case: Instances that encode similar problems and that are encoded by the same procedure are very likely to be solved by the same engine every time, so there is little advantage in running multiple engines. Furthermore, the approach of running different non-communicating engines of course is of little theoretical interest.

4.1.2 Search space partitioning

In this subsection, we discuss the *search space partitioning* approach to parallel SAT solving. This approach does not try to solve the same CNF formula using multiple engines but instead tries to divide the search space so that individual workers explore different parts of the search space.

This partitioning is usually done by adding additional clauses that partition the search space before starting the search procedure. Such a set of clauses is often called a *guiding path* in the literature. Each guiding path determines a *subproblem* of the input formula. In practice, many solvers generate guiding paths that only contain unit clauses.

In order to be able to prove unsatisfiability, guiding paths have to be chosen so that each assignment of the input CNF formula is contained in at least one guiding path. We also want to choose them so that each assignment is indeed part of exactly one guiding path. This motivates the next definition.

Definition 4.2. *Let $\gamma_1, \dots, \gamma_\ell$ be sets of clauses. $\{\gamma_1, \dots, \gamma_\ell\}$ is called a* collection of guiding paths *(for a formula ϕ) if the sets of models of the $\gamma_1, \dots, \gamma_\ell$ form a partition of the set of assignments of the variables in ϕ.*

This condition from definition 4.2 helps to prevent different workers from accidentally exploring the same part of the search space; however, it is not sufficient to ensure this. As a pathological example, consider a variable x that does not even occur in the input formula. The guiding paths $\{x\}$ and $\{\bar{x}\}$ do not constrain the search space; if these guiding paths are given to two workers, the workers will explore exactly the same search space. In practice it is of course easy to avoid picking literals that do not occur in the formula, but the general problem is still relevent. Remember that the CDCL algorithm explores the search space mostly using unit propagation. In industrial CNF formulas only a subset of all literals will be "important" and fix lots of literals through unit propagation; most literals will only fix a small amount of new literals. Hence, if such a literal is part of a guiding path, different workers are still suspectible to exploring the same search space.

The first parallel SAT solver that appears in the literature was written by Böhm [24] and utilizes the search space partitioning approach. Böhm's solver predates the CDCL method. Instead, it performs a DPLL algorithm without clause learning to solve subproblems. The solver uses a relatively simple branching heuristic: In each step it selects a decision literal so that the inverse of this literal has a maximal number of occurrences in clauses of shortest length. This heuristic was constructed to favor the production of unit clauses in random SAT formulas. Böhm's solver uses the same heuristic to generate guiding paths and to perform decisions while solving subproblems via DPLL.

Another approach that is outlined in [53] consists of first splitting an instance using guiding paths and then submitting all subproblems to a cluster where they are solved by CDCL solvers. The advantage of this scheme is that it does not depend on any communication and poses minimal requirements to the cluster infrastructure. On the other hand, the partition has to be chosen statically. Its quality has a large impact on the solvers performance as the wallclock time of the whole system depends on the difficulty of the hardest subproblem. Furthermore, the lack of communication also implies that techniques like clause sharing cannot be applied to the solver.

More recent search space partitioning solvers that do employ communication are Dolius [3] and AmPharoS [7]. Both solvers use a distributed client-server architecture. They are based on both search space partitioning and clause sharing. The solvers use the VSIDS heuristic to partition the search space. AmParoS contains heuristics that try to find a balance between exploring different parts of the search space while still generating learned clauses that are relevant to all workers.

4.1.3 Cube-and-Conquer

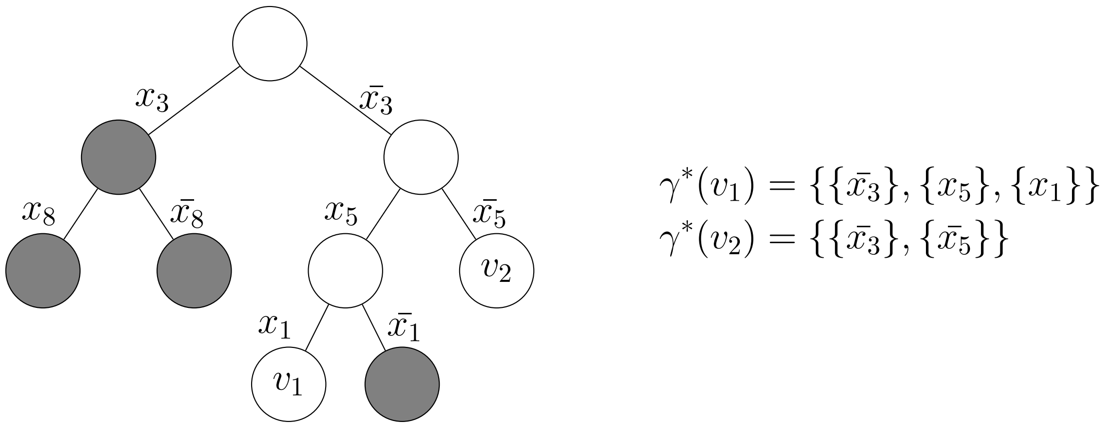
As a modern refinement of the search space partitioning approach from the last section, we discuss Cube-and-Conquer solvers.

Cube-and-Conquer refers to a SAT solving architecture that was introduced in [51]. It was not necessarily developed as a parallel SAT solver. Instead, Cube-and-Conquer is built on the idea of extending the concept of a *lookahead solver* to large industrial CNF formulas. To make this idea feasible, Cube-and-Conquer solves "easy" subproblems by CDCL.

Cube-and-Conquer proceeds in two sequential phases: In the *cube-phase*, the solver first performs a DPLL search with a lookahead-based decision heuristic. Let T be the DPLL search tree that is expanded during this search. We label each edge $e \in E(T)$ with the decision literal that was fixed while expanding the tree. Consequently, the children of each vertex define a collection of guiding paths $\gamma(v)$, where $\gamma(v)$ only contains of the single unit clause corresponding to the literal that the incoming edge of v is labeled with. Furthermore, the leafs u of T define a collection of guiding paths $\gamma^*(u)$, where $\gamma^*(u)$ is given by the union of all $\gamma(v)$ along the path from the root of T to u. Note that $\gamma^*(u)$ is essentially a cube of literals; this gives the algorithm the name "Cube-and-Conquer". Figure 4.1 depicts an example for such a DPLL search tree.

Note that some vertices are already *closed* by the lookahead procedure. That means, either the lookahead procedure was able to prove that at a given vertex $u \in V(T)$, the formula $\phi \cup \gamma^*(u)$ is unsatisfiable or all children of u are closed themselves. Vertices that are not closed yet are called *open*.

In contrast to a pure lookahead solver the Cube-and-Conquer algorithm



x_3
$\bar{x_3}$
x_8
$\bar{x_8}$
x_5
$\bar{x_5}$
v_2
x_1
$\bar{x_1}$
v_1
$\gamma^*(v_1) = \{\{\bar{x_3}\}, \{x_5\}, \{x_1\}\}$
$\gamma^*(v_2) = \{\{\bar{x_3}\}, \{\bar{x_5}\}\}$

Figure 4.1: DPLL search tree after the cube-phase
Possible result of a cube-phase that produces two subproblems $\phi \cup \gamma^*(v_1)$ and $\phi \cup \gamma^*(v_2)$. Gray denotes vertices that have been closed by the lookahead procedure.

does not continue to expand T until completion [1]. Instead it employs a *cutoff heuristic* to determine for each vertex $u \in V(T)$ of the DPLL search tree whether expanding it is worthwhile. If that is not the case, u is left open and the solver backtracks to a parent of u. After the cube-phase completes (i.e. after the cutoff heuristic has been met for all open leafs) it suffices to solve all remaining open leafs. For each of those open leafs $u \in V(T)$, the Cube-and-Conquer algorithm now solves the subproblems $\phi \cup \gamma^*(u)$ using a CDCL solver. This second phase is called the *conquer-phase.*

Note that we do not specify a particular cutoff heuristic. The original Cube-and-Conquer cutoff heuristic [51] is based on the number of decision literals (i.e. the length of the path from the root to a vertex) and the number of literals that are fixed by unit propagation, compared to the number of variables in the input formula. As our DDC algorithm does not employ a cutoff heuristic, we will not discuss it here in detail.

It is obvious that the conquer-phase of the algorithm can be naturally parallelized. As stated earlier, this parallelization can be seen as a special case of the search space partitioning approach. One problem with the approach, however, is that the cube-phase is still sequential. Just executing the conquer-phase in parallel thus might run into bottlenecks during the cube-phase.

The paper [91] introduced the concept of *concurrent* Cube-and-Conquer, a parallel algorithm that runs a CDCL algorithm in parallel to the sequential cube-phase. This CDCL algorithm always runs on the same DPLL search tree vertex as the lookahead procedure. If the CDCL algorithm finds a model of the formula, the whole solver terminates. Otherwise, if the CDCL algorithm is able to prove a vertex unsatisfiable faster than the lookahead

[1]A pure lookahead solver would expand T until either all leafs are proved to be unsatisfiable (i.e. by unit propagation derives a conflict) or a solution is found.

is able to split it, both solvers backtrack. In order to utilize multiple cores, it is desireable to use a parallel CDCL algorithm (e.g. a CDCL portfolio solver) to fully utilize the available processors.

The Treengeling solver [17] that won gold medals at the SAT Competitions in 2013, 2014 and 2016 implements a variant of this concurrent Cube-and-Conquer algorithm. Instead of running the CDCL algorithm in parallel to the lookahead procedure, each worker of the solver runs CDCL and lookahead interleaved. During this process a limit is applied to the number of open leafs. Additionally, the CDCL algorithm is run with a conflict limit; this limit is dynamically adjusted depending on the number of vertices that are solved by CDCL versus the number of vertices that are produced by lookahead [18].

A remaining problem of the concurrent Cube-and-Conquer approach is that if the CDCL algorithm is unable to solve a vertex, the CPU time it spent is essentially wasted. This happens especially often when the solver starts and no literals are fixed yet. The distributed divide-and-conquer solver that we will present in section 4.2 tries to avoid this problem by ensuring that workers contribute to a solution (or unsatisfiability proof) at all stages of the solving process.

4.1.4 Parallelizing CDCL

In addition to using a parallel high-level algorithm it is also possible to parallelize its building blocks to a certain extent. We shortly discuss algorithms that try to parallelize CDCL at the level of unit propagation.

Most of the time that a DPLL-based (or CDCL-based) SAT solver consumes is spent on doing unit propagation. Therefore it seems evident that one could try to parallelize the unit propagation itself and some work has been done on that [67, 54]. Unfortunately there are both theoretical and practical hurdles that impair the scalability of this method. From a theoretical point of view unit propagation is P-complete[2] and thus it is an open question whether there is an efficient parallel algorithm for unit propagation at all.

In practice it is unlikely to achieve a decent speedup by parallelizing unit propagation. Because the propagation of a single literal can result in the examination of arbitrary parts of the input formula, frequent synchronization between worker threads is required if these parts of the formula are handled by different workers. Even the author of the parallel, shared-memory unit propagation implementation in [67] remarks that his algorithm does

[2]This can easily be seen because HORN-SAT is already P-complete and unit propagation solves HORN-SAT in linear time. To see that HORN-SAT is P-complete, it suffices to reduce the CVP (see section 2.3.1) to HORN-SAT.

not scale to more than two cores. It seems likely that such an algorithm cannot be ported to a distributed computer without introducing a negative speedup even for two nodes.

Despite these problems, the parallel unit propagation algorithm of [67] does have one advantage over the other approaches that we consider here: It does not need to store the whole input formula in RAM and thus can be used to solve formulas that are too large to fit into a machine's memory. Instead, only an arbitrarily small subformula of the input formula needs to be stored by each individual worker. Therefore, it can be used in a complementary way with the other approaches to enable the solver to operate on formulas that would otherwise be to large to be handled.

However, we will concentrate on improving the scalability of parallel SAT algorithms in the remainder of this thesis. Hence, we will not consider parallel unit propagation any further.

4.2 The DDC algorithm

In this section, we present a novel parallel SAT algorithm that we call the *distributed divide-and-conquer* (DDC) algorithm.

4.2.1 The basic algorithm

Here, we first present the basic algorithm. The following subsections will discuss specific parts of this algorithm.

DDC is a search space partitioning solver that operates similar to the Cube-and-Conquer approach and also builds a DPLL search tree T (like the one in figure 4.1) to generate guiding paths. Similar to Cube-and-Conquer, a lookahead based algorithm is used to compute the branching heuristic. As opposed to Cube-and-Conquer however, the lookahead scores are not computed sequentially and no cutoff heuristic is required to decide when to run CDCL.

Instead, we attach (possibly empty) sets of workers to vertices of the DPLL search tree. Those workers are routed through the tree until they reach an open leaf vertex. When that happens, one of two possible operations is performed. We call these operations *divide-* and *conquer-steps*. Let u be a leaf vertex. Divide- and conquer-steps operate on u as follows:

Divide-step The workers perform a parallel lookahead operation. After this lookahead completes, guiding paths are computed and for each of those guiding paths, the vertex u is expanded with a new child vertex. As a by-product this operation yields a set of failed literals.

For each failed literal a, we learn a clause that consists of $\bar{a}$ and the inverse[3] of all elements of the guiding path $\gamma^*(u)$ that contributed to the conflict when a was assigned. We denote the set of these clauses by $\eta(u)$. $\eta(u)$ is added to the formulas associated with each new child vertex to further reduce the search space.

Conquer-step The workers run the CDCL algorithm in order to solve u directly. If one of the workers finds a solution, the whole solver terminates. Otherwise, if one of the workers proves the formula associated to u to be unsatisfiable, the vertex u is closed.

If the solver is able to close the root vertex, the input CNF formula ϕ is proven to be unsatisfiable. It should be noted that there is a number of corner cases that can result from divide- and conquer-steps: Divide-steps might discover a satisfying assignment (e.g. assigning a literal or after propagating new failed literals). Both divide- and conquer-steps can directly prove the formulas associated to parent vertices of u to be unsatisfiable. For example, this can happen if CDCL reaches a conflict that includes only a subset of the guiding path. In an exceptional case, this can even directly prove ϕ to be unsatisfiable.

Our heuristic of choosing between divide- and conquer-steps is easy. First, we always perform a conquer-step when only a single worker is attached to the leaf u. If there is more than one worker, we run a conquer-step until all workers combined reach a conflict limit. After that we interrupt the conquer-step and run a divide-step instead. If workers arrive at u while a conquer-step of u is already in progress, the workers join the active conquer-step. If a divide-step is in progress, the workers have to wait until it is completed.

Initially, we attach all workers to the root vertex of T (which corresponds to the empty guiding path). Workers that are attached to an inner vertex v of the tree are routed through T until they reach a leaf vertex. Routing moves workers into two directions: If the given vertex v is open, the workers attached to it are routed to child vertices of v. We distribute the workers so that each subtree of v contains the same number of workers. If v is already closed, the workers attached to it are routed to the parent of v until they reach an open vertex.

Because this routing operation prefers to move workers locally, the guiding path does not change chaotically after routing. This ensures that clauses that are learned during CDCL stay relevant at the new guiding path.

[3]The use of the assumption mechanism that we discuss in 4.3.1 ensures that this is possible, even if the guiding path has non-unary clauses.

4.2.2 Parallel lookaheads

Our DDC algorithm performs a parallel lookahead procedure during divide-steps. We shortly discuss different parallel algorithms to perform this lookahead.

There are two natural ways to parallelize the sequential lookahead algorithms 2.3 and 2.4 from chapter 2. The first way consists of computing the lookahead score for multiple literals concurrently and the second is to parallelize the lookahead evaluation procedure itself. The second approach might be desirable in situations with few variables (e.g. less variables than the number of cores) and expensive lookahead evaluation algorithms. However, we can expect industrial CNF formulas to have many times more variables than we have cores available, so that the first approach is likely to yield better performance.

Disregarding communication costs, we expect a linear speedup when computing the lookahead procedure for multiple variables in parallel. However, the runtime of this algorithm is still bounded by the runtime of the unit propagation algorithm. A natural theoretical question is thus, whether it is possible to design a parallel lookahead algorithm with a smaller runtime than the sequential unit propagation algorithm. This, however, is unlikely as unit propagation is P-complete (as discussed in subsection 4.1.4). In that sense, the runtime of our parallel lookahead algorithm is optimal under the assumption that $P \neq NC$.

In order to further speed up the lookahead, we can combine it with a (sequential or parallel) preselection phase. This might be especially worthwhile if the number of available workers is low. If enough workers are available, preselection becomes less important as it becomes possible to compute more lookaheads in reasonable time. Therefore, we run just a simple sequential preselection phase.

Suppose that we want to run a divide-step using the set Ω of workers that are attached to a vertex u of the DPLL search tree. We randomly choose one of the workers from Ω as the *leader* of the divide-step. The leader is responsible for coordinating all other workers in Ω. In particular, it runs the sequential preselection phase before starting the parallel lookahead and it generates the new child vertices of u after the lookahead is finished.

The naive lookahead algorithm 2.3 can easily be parallelized by employing a load balancer to distribute the variables over all available workers. After the lookahead of all variables is completed, the result of the whole lookahead evaluation can be computed by a parallel reduction operation that computes the maximum of all lookahead scores. If the number of available workers is close to the number of variables, this parallelization of the naive lookahead algorithm is optimal in the following sense: The number of variables that have to be probed by each worker is bounded by a constant. Thus, the walltime performance of this algorithm is determined by the most expensive unit propagation run and this is true for any other lookahead algorithm as well.

However, in the general case (i.e. when the number of variables is far greater than the number of available workers) we can try to parallelize the tree-based algorithm 2.4 in order to archive better performance. In this case, the load balancer does not distribute variables; instead it distributes the literals that are roots of the DFS forest which is constructed by the algorithm. The individual workers compute the lookahead scores for all literals that are in the trees with these specific roots. After lookahead of all literals is completed, the algorithm still needs to combine the scores of the positive and negative literals of each variable to obtain a lookahead score for the variable itself. Therefore, it does not suffice to perform a parallel reduction operation like in the naive case. All workers need to send the scores of all literals to the leader. The leader performs a sequential search to determine the variable with best lookahead score. The fact that it requires more communication and sequential runtime to determine the best variable score from the scores of individual literals is a drawback of the algorithm compared to the naive one.

So far, we did not specifiy a lookahead evaluation heuristics for the algorithm. An obvious choice for the score of a literal x is the number of literals that are fixed as the result of applying unit propagation to the assignment $\{x\}$. This number is cheap to compute and computing it does not require dedicated data structures. Other choices would be any of the evaluation heuristics that established lookahead solvers use (see e.g. [14] for an overview). However, most of these heuristics require at least occurrence lists to be computed and often need eager data structures for unit propagation (see 3.2.3) for efficiency.

4.2.3 Managing the distributed search tree

In order to actually implement the DDC algorithm, we have to specify how workers will be mapped to threads and processes. This subsection will discuss that mapping and the organization of the DPLL search tree that the solver has to manage.

In an implementation, a worker will be represented by a thread, with multiple worker threads running inside a single process. In general, it is desirable to either start one process per node or per CPU socket and then start as many worker threads as there are cores available on this node or socket.

In addition to the individual worker threads, each process also manages some state that is shared among all worker threads. This state includes the DPLL search tree itself. In the remainder of this section we will discuss how the DPLL search tree is organized.

While running the DDC algorithm, each of the workers has to access the formula $\phi \cup \gamma^*(u)$, where u is the vertex that the given worker is attached to. There are generally two ways of how this information can be provided to each worker: Let v be some vertex that is on the path from the root of T to u; either the worker requests $\gamma(v)$ from a centralized server that manages the whole DPLL search tree or the management of the search tree itself is distributed over multiple processes.

In order to avoid a design where a central server has to communicate with every other process, our algorithm uses the second approach. Vertices of the DPLL search tree are uniquely identified by IDs that consist of pairs (p, n), where p is the index of a process that we call the *owner* of the vertex and n is an integer. We allocate n by maintaining a per-owner counter that is incremented whenever a vertex ID is created by this owner. This ensures that no communication is required to allocate vertex IDs.

Recall that vertices are only added to the DPLL search tree as the result of a divide-step. When that happens, the leader of this divide-step becomes the owner of all vertices that are generated by the divide-step. Let v be such a vertex. The owner is then responsible for routing workers to the children of v.

In order to run the algorithm, all processes that have workers which are attached to a vertex in the subtree of v have to store $\gamma(v)$. The owner of v has to store the number of child vertices of v and the number of completed child vertices. In addition to that, it has to store, for each child vertex of v, whether that child vertex is already closed. This information allows the owner to determine whether v itself is already closed and it is required for routing decisions. The next subsection will detail the communication that is required to share $\gamma(v)$ among workers.

4.2.4 Communication model

In this section we discuss how the DDC algorithm can be implemented on top of a message passing framework.

In our DDC implementation processes send commands to each other in order to route workers through the DPLL search tree and to invoke divide- and conquer-steps. Commands are always sent and received between processes and never between individual threads. Some commands still affect specific workers. Those commands are forwarded to the workers via shared-memory queues. However, it can happen that a process sends a command

Table 4.1: Command overview

Category	Message
Vertex management	QUERYVERTEX(u), VERTEXDATA$(u, p, \gamma, \eta(p))$
Routing	ROUTE(u, Ω), CLOSE(u, Ω)
Divide-/conquer-step	DIVIDE(u, Ω), CONQUER(u, ω), REQUESTINTERRUPT(u, ω)
Result of divide-step	DIVIDECOMPLETE$(u, \Gamma, \eta(u))$, DIVIDETERMINATE(u)
Result of conquer-step	PROGRESS(u, k), CONQUERINTERRUPT(u, ω)
	CONQUERCOMPLETE(u, ω), CONQUERTERMINATE(u, ω)
Solution	SATISFIABLE(τ), UNSATISFIABLE
Shutdown	SHUTDOWNSOLVER, SHUTDOWNWORKER(ω)

The meaning of the symbols $u, p, \gamma, \Gamma, \eta, \omega, \Omega, k$ and τ is as follows:

Symbol	Meaning
u	Vertex of the DPLL search tree
p	Parent vertex of u
γ and Γ	Guiding path and set of guiding paths
$\eta(\_)$	Set of redundant clauses that are relevant at some vertex
ω and Ω	Single worker and set of workers
k	Number of conflicts since last PROGRESS command
τ	Model of the input CNF formula

to itself[4].

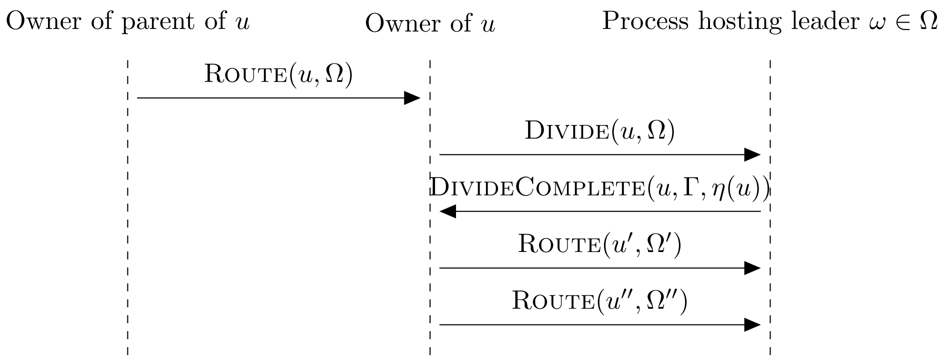
Table 4.1 states all commands that our implementation uses. It should be noted that commands are not the only messages that are exchanged in our DDC implementation. For example, divide-steps employ additional messages for load balancing and to exchange intermediate results between workers.

Using the QUERYVERTEX command, processes can enquire information about vertices u that are owned by other processes. The owner processes respond with a VERTEXDATA command that contains the relevant information. Note that VERTEXDATA contains the set $\eta(p)$ of redundant clauses and not $\eta(n)$. This is because $\eta(p)$ consists of clauses that assign failed literals which were found during the divide-step that generated u. These failed clauses apply to all child vertices of p and not only to n.

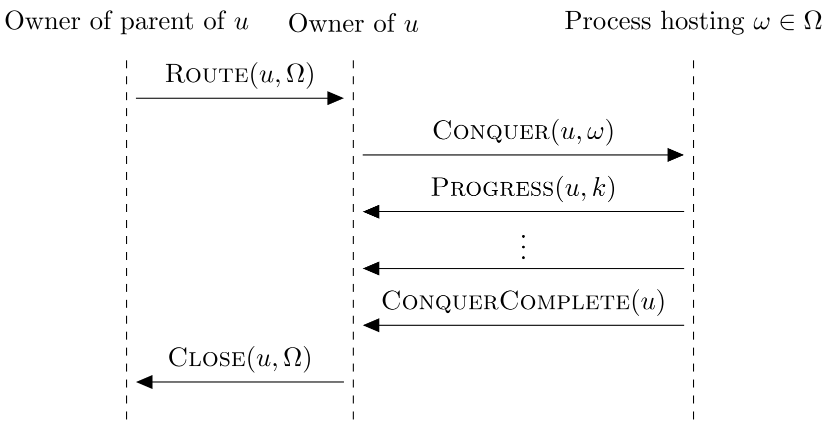
The commands ROUTE and CLOSE route workers through the DPLL search tree. ROUTE moves the set Ω of workers to the vertex u, while CLOSE moves the workers from Ω to the parent of u after u has been closed. Thus, ROUTE is always sent to the owner process of u and CLOSE is always sent to the owner of the parent of u.

After workers arrive at a leaf vertex u, the owner of u sends DIVIDE or CONQUER commands to the workers attached to u. Those workers usually

[4]Of course, that is not strictly necessary as those commands could also be sent over shared memory. However, it simplifies the implementation of the algorithm. We expect the overhead of those messages to be negligible as high-quality message passing libraries (e.g. MPICH and Open MPI) will fall back to shared memory anyway.


Owner of parent of u
Owner of u
Process hosting leader $\omega \in \Omega$
$\textsc{Route}(u, \Omega)$
$\textsc{Divide}(u, \Omega)$
$\textsc{DivideComplete}(u, \Gamma, \eta(u))$
$\textsc{Route}(u', \Omega')$
$\textsc{Route}(u'', \Omega'')$

(a) Divide-step
We assume that the divide-step generates two child vertices u' and u''. Ω' and Ω'' denote the sets of workers that are routed to these new vertices.


Owner of parent of u
Owner of u
Process hosting $\omega \in \Omega$
$\textsc{Route}(u, \Omega)$
$\textsc{Conquer}(u, \omega)$
$\textsc{Progress}(u, k)$
$\vdots$
$\textsc{ConquerComplete}(u)$
$\textsc{Close}(u, \Omega)$

(b) Conquer-step

Figure 4.2: Successful completion of divide- and conquer-steps

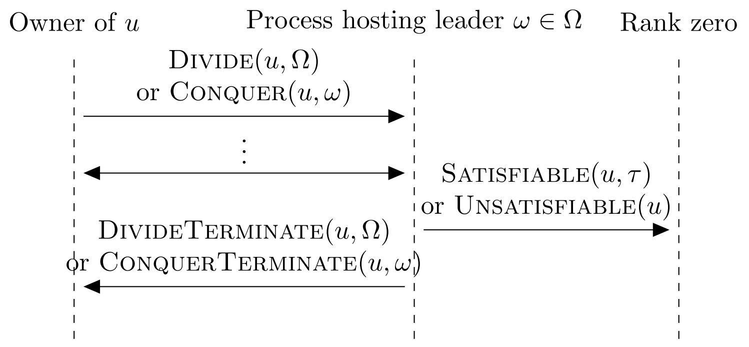

Owner of u
Process hosting leader $\omega \in \Omega$
Rank zero
DIVIDE(u, Ω) or CONQUER(u, ω)
SATISFIABLE(u, τ) or UNSATISFIABLE(u)
DIVIDETERMINATE(u, Ω) or CONQUERTERMINATE(u, ω)

Figure 4.3: Termination after a divide- or conquer-step

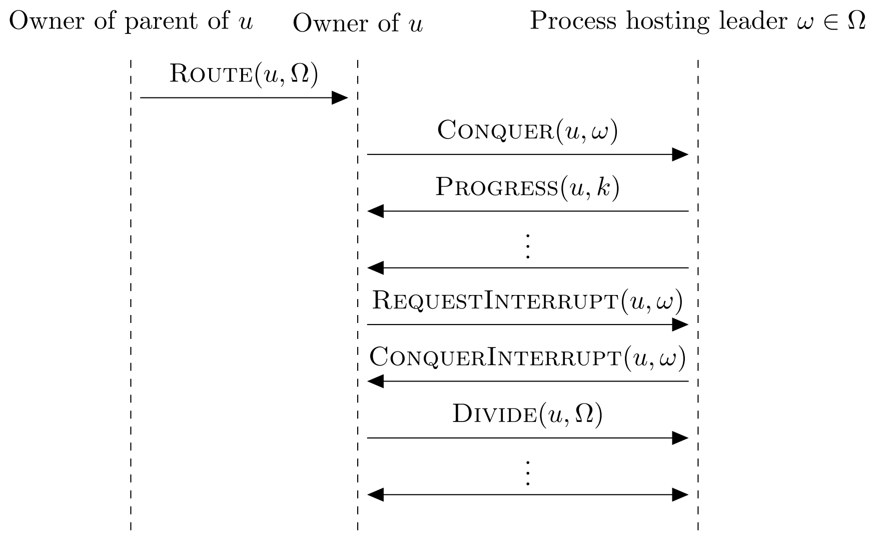


Figure 4.4: Interruption of a conquer-step

send a DivideComplete or ConquerComplete command back to the owner of u, after the divide- or conquer-step finishes. Figure 4.2 shows the message flow in these two cases.

If a divide- or conquer-step finds a satisfying assignment for the input formula or proves that the input formula is unsatisfiable, the workers instead send a DivideTerminate or ConquerTerminate command to the owner. This ensures that the workers do not enter the routing algorithm again. Instead, they wait until the whole solver terminates. Figure 4.3 depicts this situation.

In addition to that, conquer-steps can finish by sending a ConquerInterrupt command to the owner of u. This happens only after the owner has requested to interrupt the conquer-step by sending a RequestInterrupt command. In order to allow the owner to estimate if conquer-steps should be interrupted, workers send Progress commands at regular intervals during conquer-steps. This is visualized in figure 4.4.

Finally, the ShutdownSolver command is broadcasted to all workers when the solver terminates. As a response to that, each process sends ShutdownWorker commands to all workers that are attached to vertices that the process owns. The latter command causes the workers to exit. After all workers that are part of a process exited, a global barrier is issued and the solver itself exits. We remark that the termination process is considerably difficult to implement correctly in practice, even though it is conceptually simple. The mechanism outlined above prevents deadlocks that would otherwise arise when a worker waits for the response of anther worker (e.g. during divide-steps) and that second worker terminated already.

4.3 satUZK-ddc: Implementation and refinements

This section discusses implementation details and some refinements that we made to our basic DDC algorithm.

We implemented the DDC algorithm as part of our satUZK framework. The resulting solver is called *satUZK-ddc*. The implementation is written in C++ and uses MPI for message passing. We use the same CDCL engine as satUZK-seq to implement conquer-steps.

4.3.1 Assumptions and local LBD

We shortly discuss how assumptions can be used to force the solver to explore a certain guiding path.

In order to be able to efficiently change the guiding path as a result of a completed cube- or conquer-step, we assign the current guiding path using the assumption mechanism. This allows us to change the guiding path by only changing the current set of assumptions and without touching other data structures (e.g. it does not require deleting clauses from the clause database). Unary clauses can directly be assigned by assumptions. However, if the guiding path $\gamma(u)$ associated with a vertex u contains a non-unary clause, we need to introduce a new variable x_u. We add $\bar{x_u}$ to each non-unary clause of $\gamma(u)$ and assign x_u as an assumption. However, as learned clauses containing $\bar{x_u}$ will not be useful in other parts of the search tree[5]. Therefore, it is desirable to avoid guiding paths that contain non-unary clauses.

A consequence of using assumptions is that the meaning of the LBD as an indicator of clause quality changes: We can expect that most of the clauses which are learned during conquer-steps contain a large number of assumptions; some of them will even contain the whole guiding path. In those cases, as assumptions are all assigned on different decision levels, the LBD will be dominated by the number of assumptions that are part of the clause. While the LBD is still fine as a measure of a clause's quality with respect to the input formula, including assumptions in the LBD leads to a poor measure of the quaility of the clause with respect to the current guiding path.

This problem was already recognized in [6], altough not in the context of parallel solving. The solution which that paper discusses is simply dropping assumptions from the LBD calculation. While this strategy seems to perform well for the paper's application of minimal unsatisfiable cores (MUS), it is not a good idea in our case: If we just drop assumptions from the LBD calculation, clauses that are useless on the current guiding path will not be deleted in case they were useful on a previous guiding path.

Therefore, we adopt the following solution: For each clause, we store a *global* LBD and a *local* LBD. The global LBD counts all assumptions as usual. The local LBD does not count assumptions at all. Hence, the global LBD is an upper bound for the local LBD. The usual mechanism to improve the LBD of clauses that participate in conflicts (see 3.1.5) only affects the local LBD and we use the local LBD in the clause database reduction heuristics.

When we change the guiding path, we reset the local LBD to the global LBD. Clauses with small LBD before reset are protected for one round of clause database reduction. Due to the LBD improvement mechanism, they get the chance to reduce their local LBD if they are still useful on the new

[5]Per construction, learned clauses containing x_u are not produced by CDCL.

guiding path.

In addition to enabling this reset mechansim, the global LBD can also be used as an indicator for the quality of clauses regarding clause sharing. We will discuss clause sharing in the context of the DDC algorithm in subsection 4.3.3.

4.3.2 Branching schemes

This subsection will discuss alternatives to the usual branching scheme that generates two guiding paths as the result of each divide-step.

Scattering is an alternative technique to generate guiding paths, or equivalently, to expand vertices when branching in a DPLL search tree. Scattering was introduced in [52]. While the traditional DPLL method generates 2^ℓ guiding paths from ℓ branching literals, scattering takes ℓ cubes of literals as input and produces $\ell + 1$ guiding paths.

Definition 4.3. *Let $D_1, \dots, D_\ell$ be sequence of cubes. Note that the permutation of the D_i does matter.* Scattering *generates the $\ell + 1$ guiding paths*

$$\begin{aligned}
\gamma_1 &:= \{\{a\} : a \in D_1\} \\
\gamma_2 &:= \{\{a\} : a \in D_2\} \cup \{\{\bar{a} : a \in D_1\}\} \\
\gamma_3 &:= \{\{a\} : a \in D_3\} \cup \{\{\bar{a} : a \in D_1\}, \{\bar{a} : a \in D_2\}\} \\
&\ \ \vdots \\
\gamma_{\ell+1} &:= \{\{\bar{a} : a \in D_1\}, \dots, \{\bar{a} : a \in D_\ell\}\}
\end{aligned}$$

In other words, the i-th guiding path has the form

$$\gamma_i := \{\{a\} : a \in D_i\} \cup \{\{\bar{a} : a \in D_1\}, \dots, \{\bar{a} : a \in D_{i-1}\}\}$$

with $D_{\ell+1}$ defined as $D_{\ell+1} = \varnothing$.

Note that if $\ell = 1$ and D_1 contains only a single literal c, scattering degrades to the usual branching scheme that creates two guiding paths fixing the literals c and $\bar{c}$. We call this branching scheme *binary branching.* Furthermore, if ℓ is arbitrary but all D_i are unary cubes, then all γ_i only contain unit clauses. This interacts nicely with our use of assumptions that we discussed in the previous subsection.

For the general case the paper [52] contains some guidelines how $D_1, \dots D_\ell$ should be chosen so that all $\phi \cup \gamma_i$ have a similar difficulty. However, these guidelines depend on an estimation of the size of both search trees that might be inacurate in the presence of unit propagation.

Because of these complications with $|D_i| > 1$, we only deal with D_i that have $|D_i| = 1$ in satUZK-seq. In order to generate child vertices as the result of a divide-step, we set ℓ to the number of workers that participated in the divide-step. We set D_i to the unary cube that contains the i-th best literal as determined by the lookahead evaluation heuristic. We call the resulting branching scheme *unary scattering*. While the scheme does not guarantee that all γ_i have comparable difficulty, it ensures that workers quickly get to explore different parts of the search space. In our experimental section, we compare this scattering scheme to binary branching.

4.3.3 Clause sharing

Another enhancement of our DDC algorithm is the addition of clause sharing. We shortly discuss the interaction between clause sharing and DDC.

Even though DDC performs search space partitioning, clauses that are generated in one part of the search space can still be relevant to other parts. For example, one of the workers might be able to learn a unit clause that fixes a variable for all workers. Clause sharing is made possible because the assumption mechanism ensures that even though a clause is learned by a worker with a certain guiding path, it is still globally valid.

As sharing only unit clauses has proven to be effective in the Plingeling solver, we implemented this strategy in our satUZK-ddc solver. In the experimental section, we compare this strategy to a strategy that shares all clauses with an LBD of two or less.

We use clause freezing (as presented in section 4.1.1) to avoid resetting active CDCL searches when new clauses arrive. Clauses which get imported from other workers are frozen after they are allocated. The solver then unfreezes those clauses during clause database reduction. Clauses are unfrozen based on their PSM; this implies that unit clauses are always unfrozen.

The actual clause exchange is implemented by a combination of shared-memory queues and message passing. Generated clauses are passed to different workers of the same process via shared memory queues. At the same time, the worker that generated the clause prepares a message to send the clause to a different process. For purposes of clause sharing, the processes are arranged in a ring: Each process always sends clauses to exactly one other process and receives clauses from exactly one other process. The same model is used for the shared memory queues that pass clauses between workers of the same process.

4.3.4 Diversification

We quickly discuss the role of diversification as part of the DDC algorithm.

In portfolio solvers, the primary purpose of diversification is ensuring that workers do not explore the same search space and do not perform redundant work. This is not a concern for us, as satUZK-ddc already avoids this problem by partitioning the search space. Diversification can still improve performance for two reasons: First, the best heuristics for restarts and clause database reduction simply differ among different CNF formulas. Secondly, different heuristics might be able to derive different useful learned clauses that can be shared between the workers.

For these reasons, we implemented a simple diversification scheme in satUZK-ddc: satUZK-ddc runs one half of the worker threads in Glucose emulation mode and the other half in MiniSat emulation mode (see 3.2.9). We expect Glucose emulation to be effective on the majority of industrial CNF formulas, while we expect MiniSat emulation to be effective on some classes of hard combinatorial formulas where Glucose emulation performs badly (see the evaluation in 3.4.2).

4.3.5 Idle-time inprocessing

In this subsection we discuss techniques to reduce the idle time of satUZK-ddc.

The construction of the DDC algorithm tries to reduce the idle time of worker threads to a minimum by ensuring that workers can almost independently enter divide- and conquer-steps and by applying load balancing when multiple workers need to work on the same task. Unfortunately, it is still possible for workers to idle for non-negligible amounts of time. For example, that can happen if a worker is waiting for the completion of a divide-step or if it is waiting for the response to a command. In order to prevent wasting time, we run *inprocessing* when a solver is waiting for external events.

In contrast to sequential SAT solvers that make heavy use of inprocessing (like Lingeling [15]), we do not need to care about the efficiency of inprocessing or about sophisticated scheduling schemes as we will only run inprocessing when nothing else can be done. The version of satUZK that was submitted to SAT Competition 2017 runs subsumption inprocessing during idle-time. Subsumption removes clauses that are contained in other clauses as subsets. Subsumption interacts favorably with clause sharing as duplicated learned clauses can be removed by subsumption. In an experiment we also compare this inprocessing configuration to a configuration that performs resolution subsumption. Chapter 5 discusses both of these thechniques in depth.

4.3.6 Asynchronous MPI via fibers

In this subsection we shortly discuss how the communication infrastructure is organized in satUZK.

From subsection 4.2.4 it is obvious that many messages (e.g. completion messages for divide- and conquer-steps or interrupt messages) are sent and received asynchronously: Those messages are not generated at predetermined points during the algorithm but instead result from events that are hard to predict. Furthermore, it is evident that nodes need to be responsive to messages from other nodes even while their workers are computing divide- and conquer-steps.

One way to deal with those circumstances would be allocating a dedicated communication thread per process. Messages sent by individual workers would then be routed over the communication thread. Workers in the some process could interact with the communication thread over shared memory queues.

Early versions of satUZK-ddc implemented this approach. During profiling we found the communication thread to be underutilized in many situations. Therefore, we switched to a completely asynchronous model, where all threads are equal and implement workers of the DDC algorithm. In order to deal with asynchronous messages, we use user-space threads that are called fibers [6]. A fiber, just like a thread, is an execution context. However, unlike threads, fibers are not scheduled by the OS but by the solver itself. We implemented three priority-levels for fibers:

Batch priority Fibers in this priority-level are scheduled with the highest priority. They are only suspended to wait for the completion of blocking operations but never yield the CPU cooperatively.

Computation priority These fibers are only scheduled when no fiber with batch priority is available. They are expected to regularly yield the CPU to allow fibers with polling priority to run.

Polling priority These fibers only run after a fiber with computation priority yields and if no fibers with batch priority are available.

We implemented adapters for many of the MPI communication functions that only block the current fiber instead of the current thread. In order to realize this, each process runs one *MPI progress* fiber with polling priority that continuously checks if pending MPI operations are completed. This fiber then wakes other suspended fibers that block for these operations.

[6]In particular, we are using the user-space threading library Boost.Fiber [25].

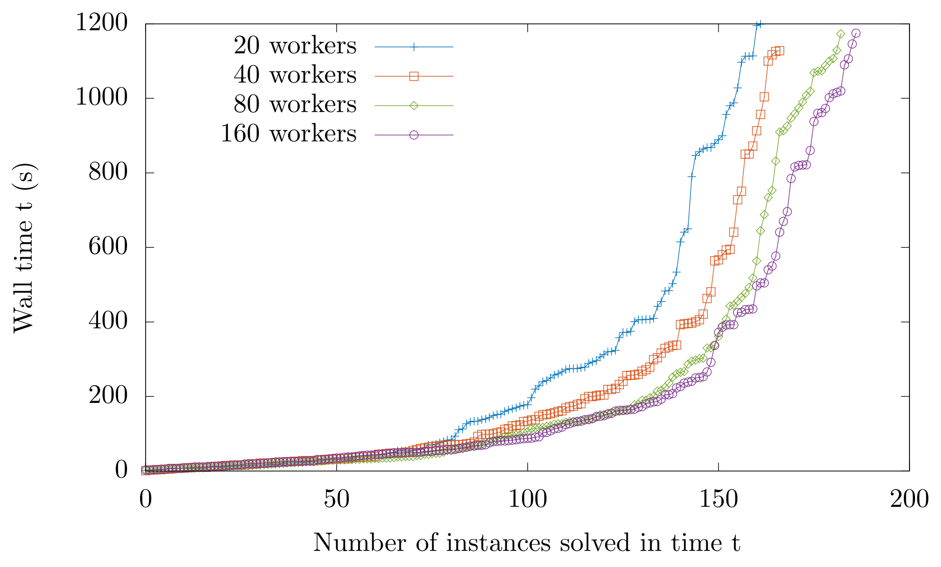


Figure 4.5: Scalability of satUZK-ddc on SC'17 instances

4.4 Experimental results

In this section, we evaluate the performance of our DDC algorithm empirically.

All experiments ran on a cluster of dual-socket Intel Xeon E5-2690 v2 nodes. Each of those CPUs consists of 10 individual cores[7], resulting in 20 cores per node. The cores run at a reference frequency of 3.0 GHz. Each node has access to 128 GiB of memory. The cluster runs on Linux, using the Debian Stretch distribution. As in the last chapter, we used the GCC 6.2.0 compiler to build our solvers.

When evaluating the DDC algorithm we always start one process per socket and one thread per core on this socket. This amounts to two processes per node and ten worker threads per process. The number of nodes that we use varies among different experiments.

4.4.1 Scalability of DDC

Our first experiment concerns the scalability of the satUZK-ddc solver. In order to evaluate this, we run satUZK-ddc on the 350 instances from SAT Competition 2017 and vary the number of workers from 20 to 160 (more specificially, we vary the number of nodes from 1 to 8). No CNF simplification is applied before starting the DDC algorithm. In this experiment, we set a per-instance timeout of 20 minutes. This timeout is much lower than the timeout of 5000 seconds from the Main track of SAT Competition 2017,

[7]Technically, Intel's hyper-threading provides two hardware threads per core, yielding 40 hardware threads per core. However, we disabled hyper-threading for our experiments. The solvers that we consider are limited by memory bandwidth and hyper-threading does not significantly improve their performance.

however, in contrast to the competition we run our benchmark on more than one node and thus invest more CPU time (at least in configurations with more than 100 workers).

Figure 4.5 depicts the results of this experiment as a cactus plot. The time that we report is the total wall time that is needed by the DDC algorithm to solve a given instance. Instances that could not be solved within the timeout are not reported in the figure.

As expected, figure 4.5 shows that the runtime per instance decreases when more workers are available. At the same time, the number of instances that can be solved within the timeout increases. It should be noted that the relation between the number of workers and the number of solved instances is not linear. This is no shortcoming of the algorithm but instead is caused by the fact that the instance difficulty is non-linear itself.

Note that although satUZK is written for a distributed computing architecture, configurations with 20 workers or less are essentially shared-memory solvers: The 10-workers configuration only performs communication within the same process and the 20-workers configuration only performs communication within the same node. All other configurations perform communication even between nodes. The configurations still all use the same message-passing API. As can be seen from figure 4.5, scalability does not noticably differ between the shared-memory and distributed configurations. This suggests that the performance of the DDC algorithm is not suspectible to latency differences originating from the communication fabric.

While the cactus plot confirms that increasing the number of workers is always beneficial, it does not visualize whether the solver uses its resources efficiently and whether the parallel speedup is justified relative to the number of invested workers. Unfortunately, in case of the SAT problem, it is hard to accurately determine the efficiency of the solver. Because solving times often vary chaotically after slight modifications of the algorithm, the intersection of instances that are solved by a sequential reference algorithm and instances that are solved by our parallel algorithm is relatively small. In table 4.2 we summarize the speedup of the satUZK-ddc solver compared with satUZK-seq. Here, satUZK-seq ran with a timeout of 3200 minutes, which is equal to the CPU time spent by the 160 worker configuration[8]. We report both the total speedup of the whole benchmark set and the median of the speedups of individual instances. Those statistics are provided both for all instances and for "difficult" instances only. Here, difficult means that the instance required more than $10n$ seconds, where n is the number of worker threads [9]. This allows us to estimate the weak scalability of the

[8]Using this timeout, the sequential solver was able to solve 239 of the 350 instances.

[9]This notion of difficulty is adopted from [11].

Table 4.2: Speedup of satUZK-ddc on SC'17 instances

(a) Over all runs

If the sequential reference algorithm could not solve an instance, we use the timeout for speedup calculation. The fifth column states the number of hard instances that could be solved. As the solver was not able to solve all instances, this number does not always decrease.

		All instances (Strong scaling)		Difficult instances (Weak scaling)		
Workers	Solved	Total	Median	Samples	Total	Median
20	162	127.49	5.14	145	129.20	6.36
40	167	412.24	5.49	139	418.45	11.89
80	183	494.20	7.76	149	505.51	22.23
160	187	553.36	9.02	134	595.61	39.34

(b) Over runs completed both by DDC and reference solver

		All instances (Strong scaling)			Difficult instances (Weak scaling)		
Workers	Solved	Samples	Total	Median	Samples	Total	Median
20	162	160	29.22	5.03	143	29.62	6.31
40	167	161	57.52	5.19	133	58.52	11.47
80	183	174	60.32	6.86	140	61.93	18.22
160	187	177	65.97	5.70	124	71.55	32.29

algorithm. Furthermore, table 4.2 is split into two parts: 4.2a aggregates data from all runs while 4.2b only includes those runs that were successfully completed both by the sequential and by the parallel solver. Because of the afforementioned chaotic behavior, we overestimate the real speedup in the first case and underestimate it in the second case. In particular, the total speedup over all runs exhibits superlinear behavior. This is a well-known phenomenon [82] that is also experienced by other SAT solvers. Overall, the speedups are comparable with those that are reported by the portfolio-based, state-of-the-art, distributed HordeSat solver [11].

4.4.2 DDC versus the state of the art

In the second experiment, we compare the performance of satUZK-ddc with other state-of-the-art solvers.

As references, we picked Syrup, Plingeling, Treengeling and HordeSat. Syrup and Plingeling respectively won the first and second place in the Parallel Track at SAT Competition 2017. Treengeling is the most successful Cube-and-Conquer solver and is architecturally the most similar solver to satUZK-ddc. HordeSat did not participate in SAT Competitions but represents the state of the art in distributed SAT solving (with the only other recently published, distributed SAT solver, AmPharoS, performing badly at SAT Competition 2016). For Syrup, Plingeling and Treengeling we use

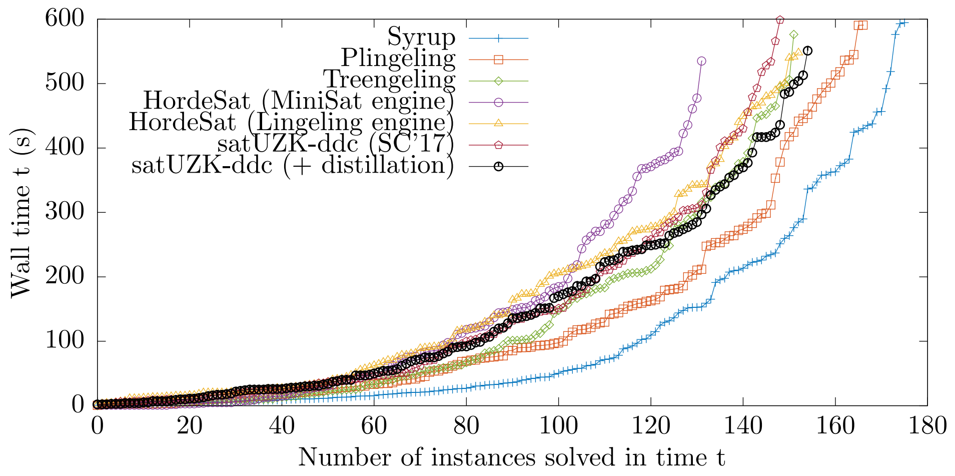


Figure 4.6: satUZK-ddc versus the state of the art

the SAT Competition 2017 versions. HordeSat was downloaded from the authors' website `https://baldur.iti.kit.edu/hordesat/`. The version of satUZK-ddc that we use here differs slightly from the SAT Competition 2017 version. In particular, we fixed a termination-related bug that might have had an impact on satUZK's performance at the competition. We choose the same set of benchmarks from SAT Competition 2017 as before. The experiment ran with a timeout of 10 minutes. All solvers run on 20 cores. The shared-memory solvers run a single process with 20 cores while the distributed solvers HordeSat and satUZK-ddc are configured as described in the beginning of section 4.4.

The results of the experiment are presented in figure 4.6. The search space partitioning solvers Treengeling and satUZK-ddc seem to perform worse than the portfolio-based solvers Plingeling and Syrup. Considering that the same, unmodified [20] version of Treengeling won the SAT Competition 2016 [78] (which also featured the same version of Plingeling), we explain this as a property of the chosen benchmark set.

The comparison between satUZK-ddc and HordeSat versus Syrup, Plingeling and Treelingeling is unfair in the sense that satUZK-ddc and HordeSat target distributed environments while the other solvers only communicate over shared memory. However, with the parallel distillation preprocessor enabled (see chapter 5), satUZK-ddc solves more instances than Treengeling. This is despite the fact that Treengeling's CDCL engine Lingeling outperforms satUZK-seq[10].

If we compare satUZK-ddc with the distributed HordeSat solver, satUZK-ddc yields decent results. satUZK-ddc solves more instances than HordeSat, even when HordeSat uses the more powerful Lingeling engine. It should be noted that HordeSat is significantly slower than Plingeling, even though

[10]Lingeling solved 7 instances more than satUZK-seq in the last SAT Competition [78]

both solvers are using the same Lingeling engine and the same portfolio-based approach. This can be seen as evidence for the difficulty of porting the portfolio approach to distributed architectures.

As can be seem from the previous subsection, all shared-memory solvers are of course eventually outscaled by satUZK-ddc (and also HordeSat).

4.4.3 Runtime of divide- vs. conquer-steps

In the next experiment, we evaluate where the satUZK-ddc solver spends its runtime.

Table 4.3: Divide vs. conquer-steps

Divide-, conquer- and inprocessing-time is given as a percentage of the total runtime. The sixth column reports the average degree of the tree. Column seven is the percentage of uninterrupted conquer-steps relative to the number of all conquer-steps.

Workers	Size of tree	Divide time	Conquer time	Inproc. time	Degree of tree	Conquer success	Shared unaries
20	14072	3.93	64.90	7.30	3.14	38.20	41479
40	32522	3.42	60.25	7.78	2.94	37.00	76188
80	48378	2.13	53.09	5.59	3.20	37.47	138031
160	83918	1.76	46.20	4.84	3.40	37.75	239481

We ran satUZK-ddc with a timeout of 5 minutes and varied the number of workers between 20 and 160. Again, benchmarks from SAT Competition 2017 are used. During this run, we measured the time that the solver spends in divide-steps, conquer-steps and inprocessing. In addition to that we recorded the size of the distributed search tree that the DDC algorithm generates.

Table 4.3 presents the results of this experiment. Only successfully solved instances are reported. The size of the constructed DPLL search tree scales linearly with the number of workers, while the degree of vertices in the tree increases slightly (because of our unary scattering strategy). The success rate of conquer-steps stays constant over all configuration. This suggests that the quality of the branching does not degrade when the degree of vertices is increased via scattering. The number of shared clauses also increases significantly; this is a trivial consequence of increasing the number of workers. However, with larger numbers of workers, some workers are likely to produce multiple copies of the same clause.

Furthermore, the time spent in divide- and conquer-steps and in inprocessing decreases when more workers are added to the solver. This is because the solver has to spend more time on book-keeping tasks and on communication. Additionally, both divide- and conquer-steps become gradually easier when the DPLL search tree becomes larger.

4.4.4 Comparison of lookahead strategies

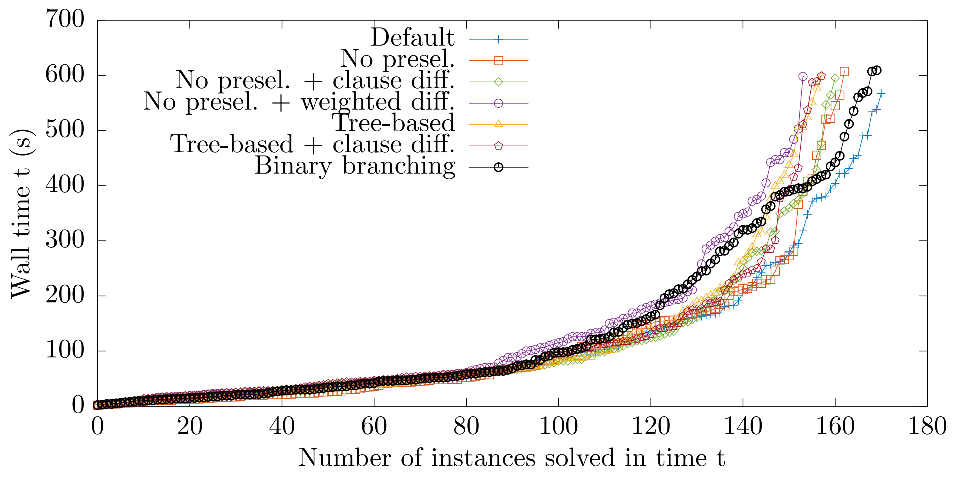


Figure 4.7: Comparison of lookahead strategies

Our next experiment compares different lookahead implementations and branching heuristics.

The default configuration from our SAT Competition 2017 version preselects the 10000 variables that occur most frequently in the current formula and then runs the parallel, naive lookahead algorithm. The number of literals that are fixed after unit propagation is used as the score for variables. We compare that configuration with a configuration which does not apply preselection and a configuration that performs tree-based lookahead (and also applies no preselection). As we conjecture that sophisticated strategies like tree-based lookahead are more important when the number of workers is small, we evaluate both 20-worker and 160-worker configurations. Furthermore, the experiment compares three different lookahead evaluation heuristics. By default, satUZK-ddc computes variable scores based on the number of literals which are fixed by unit propagation after a given variable is assigned. We compare that to a "clause difference" scheme that counts the number of clauses that are affected but not satisfied by unit propgation. Here, a clause is affected if it contains literals that are assigned to false by unit propagation. Finally, we consider a "weighted difference" scheme that weights those clauses based on the number of literals that remain unassigned. A clause with $k \geq 2$ unassigned literals contributes a score of $2^{-(k-2)}$.

The results of our comparison for the 160-worker configurations are illustrated in figure 4.7. None of the alternative strategies is able to improve the solver's performance. This is not necessarily unexpected as the default configuration has already been tuned before it was submitted to SAT Competition 2017. In order to better understand the implications of each

Table 4.4: Lookahead strategies: Statistics

The fourth column reports the average number of vertices in DPLL search trees. The remaining columns match those of table 4.3.

Configuration	Workers	Solved	Avg. tree	Divide time	Conq. time	Conq. success
Default	20	141	157	3.64	68.35	38.97
No presel.	20	145	156	10.63	62.59	39.86
Tree-based	20	144	160	5.70	66.77	38.63
Default	160	170	714	2.65	50.39	36.81
No presel.	160	163	706	3.61	48.32	39.55
No presel. + clause diff.	160	161	595	9.67	45.69	36.07
No presel. + weighted diff.	160	154	567	8.20	46.63	37.57
Tree-based	160	158	684	1.90	50.57	38.65
Tree-based + clause diff.	160	157	671	3.75	51.68	35.39
Binary branching	160	170	367	1.54	54.28	16.59

strategy, we summarize their statistics in table 4.4. This table also includes data on the 20-workers case.

All alternative lookahead and branching schemes reduce the number of DPLL search tree nodes that the solver has to explore. Despite solving the least amount of instances, the weighted difference scheme also generates the least number of vertices. Hence, it seems that the weighted difference heuristic is not able to generate subproblems that are considerably easier than their parents and conquer-steps get stuck at hard subproblems.

Tree-based lookahead reduces the time required for divide-steps considerably compared to the configurations without preselection. Unfortunately it seems that this reduction is not enough to offset the greater amount of communication that is required to collect the results after the tree-based algorithm is completed.

An interesting observation is that both tree-based lookahead and lookahead without preselection are beneficial in the 20-workers case: The number of search tree vertices that workers explore in these configurations is small enough to justify more expensive lookahead strategies that yield results of better quality.

The binary branching configuration is a special case in this experiment. When scattering is disabled, the number of DPLL search tree vertices is greatly reduced and workers spend considerably more time in conquer-steps. However, as workers are routed after they finish their conquer-steps, a significantly more conquer-steps need to be interrupted.

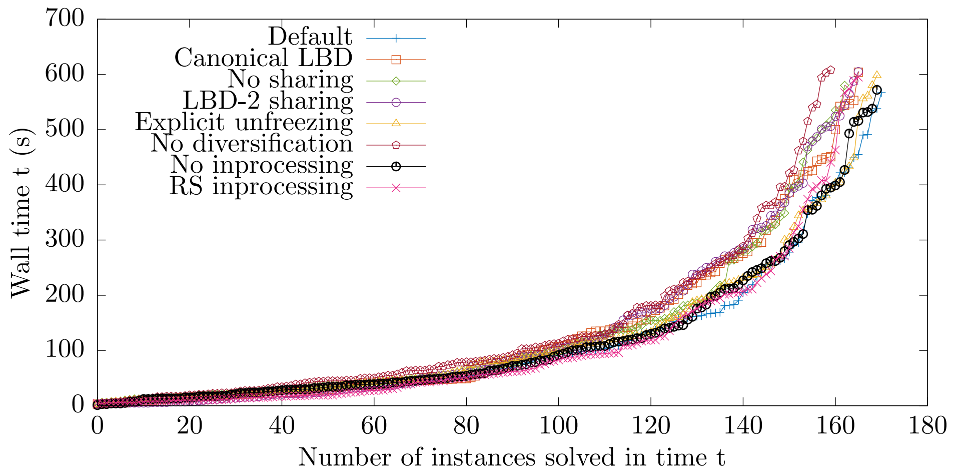


Figure 4.8: Comparison of DDC variants

4.4.5 Variants of the algorithm

In the last experiment regarding satUZK-ddc, we compare the performance of different variants of the algorithm. In particular, we compare different clause sharing and inprocessing stratgies and evaluate the performance of our local LBD scheme.

The experiment ran with 160 workers and a timeout of 10 minutes. As before, we use the SAT Competition 2017 benchmarks to compare different configurations. We evaluate the following variants of the DDC algorithm:

Default The default configuration from SAT Competition 2017. Performs subsumption inprocessing, sharing of unary clauses and our local LBD measure (see subsection 4.3.1).

Canonical LBD Uses a global LBD measure instead. Assumptions do count towards the LBD.

No sharing and LBD-2 sharing The former disables clause sharing. The latter exports clauses with a global LBD of 2 or less instead of just exporting unary clauses. Clauses are frozen during import and later unfrozen using a PSM-based policy.

Explicit unfreezing Performs an unfreezing pass over all clauses before each conquer-step. The default configuration only performs unfreezing during clause database reduction.

No diversification Uses Glucose-emulation mode for all workers.

No inprocessing and RS inprocessing The former disables subsumption inprocessing during idle time. The latter enables resolution subsumption in addition to subsumption during idle-time inprocessing.

Figure 4.8 depicts the results of this experiment. The first fact that we notice from the graph is that no configuration improves upon the default configuration. Again, this is not necessarily unexpected as the default configuration has already been tuned.

The local LBD measure certainly improves the runtime of the algorithm compared with the canonical LBD score.

Neither disabling clause sharing nor sharing more clauses improves the performance of the solver. Both of these configurations solve a few less instances than the default configuration. The explicit unfreezing pass has a less significant effect on performance but is still slightly slower than the default configuration.

Inprocessing does not seem to have a great impact on the algorithm's runtime either. Turning off inprocessing leads to a slightly slower configuration. Increasing the amount of inprocessing seems to hurt the solver even more.

Diversification seems to improve the performance of the solver significantly. The additional MiniSat-emulation configurations generate helpful learned clauses and enable the solver to solve instances where Glucose-emulation alone does not perform well.

Chapter 5

Parallel CNF simplification

This chapter will discuss the design and implementation of efficient parallel CNF simplification algorithms. At the end of the chapter we present experimental results to evaluate the practical performance of those algorithms.

What is CNF simplification? We understand CNF simplification as a collection of techniques that modify a CNF formula in order to make it more tractable for CDCL solvers. Simplification techniques will usually try to reduce the size of a CNF formula without impairing their underlying structure, or they will try to add useful clauses to the formula that fix variables or guide the solver by enhancing unit propagation.

As part of a CDCL solver, simplification techniques can be applied both as preprocessing (i.e. before starting the CDCL search) or as inprocessing. Here, inprocessing means scheduling the simplification and CDCL search algorithms in an interleaved fashion.

Usually, simplification techniques will not be strong enough to solve the CNF formula by itself. This implies that we want them to be highly efficient, as we still need to run a CDCL search. Furthermore, we want to be able to verify their result. To be able to do this, we will formulate a proof system that is strong enough to verify the correctness of our CNF simplifications in polynomial time.

The remainder of this chapter will discuss simplification techniques in depth. In section 5.1 we will discuss the theoretical basis of simplification techniques. Section 5.2 will present individual simplification techniques that are successfully applied in state-of-the-art CDCL solvers. In section 5.3 we will try to parallelize those techniques. Finally, we present experimental results in section 5.4.

5.1 Clause redundancy properties

It turns out that all established simplifcation techniques can be modelled as a sequence of adding and removing redundant clauses to and from a CNF formula. This motivates us to study classes of redundant clauses, before we look at individual simplification techniques.

5.1.1 The AT and RAT properties

This subsection will define various useful clause redundancy properties. These redundancy properties will form the basis of a proof system that we will introduce in the next subsection.

All redundancy properties will be properties of clauses C with respect to a CNF formula ϕ. Here C does not necessarily belong to ϕ.

Definition 5.1. *We call a clause C (not necessarily part of ϕ)* redundant (Red) *with respect to a CNF formula ϕ, if and only if ϕ and $\phi \cup \{C\}$ are satisfiability-equivalent. If ϕ and $\phi \cup \{C\}$ are not only satisfiability-equivalent, but in fact equivalent, we say that C has property* $\mathrm{Red}_{\mathrm{equiv}}$ *with respect to ϕ.*

The notion of a clause redundancy property was introduced in [58].

In the following discussion, we need to compare the strength of different redundancy properties. Formally, we introduce the following definiton:

Definition 5.2. *Let $\mathcal{P}$ and $\mathcal{P}'$ be two clause properties. We say that $\mathcal{P}'$* is weaker than *$\mathcal{P}$, if every clause that has property $\mathcal{P}'$ also has property $\mathcal{P}$ (w.r.t to the same CNF formula ϕ). In this case we write: $\mathcal{P}' \preceq \mathcal{P}$.*

In this language, $\mathrm{Red}_{\mathrm{equiv}} \preceq \mathrm{Red}$.

Unfortunately, our general redundancy property is too coarse to be useful in practice, as checking whether a clause is redundant is already $coNP$-hard; this follows from the fact that the empty clause is redundant if and only if ϕ is unsatisfiable. Furthermore, while ϕ and $\phi \cup \{C\}$ are satisfiability-equivalent if C is a redundant clause, redundancy alone does not allow us to efficiently compute a model of $\phi \cup \{C\}$ from a model of ϕ. Therefore, we need to study weaker redundancy properties that are efficiently computable. The most simple of these properties is given by the following definition.

Definition 5.3. *Let C be a clause. We call C a* tautology (T) *if and only if there is a variable $x \in \mathcal{V}$ so that $x, \bar{x} \in C$. It should be noted that the tautology property is independent from the CNF formula under consideration and only depends on the given clause.*

It is easy to see that tautologies are indeed redundant. Symbolically, we can write $\mathrm{T} \preceq \mathrm{Red}_{\mathrm{equiv}}$.

While the tautology property is certainly efficiently computable, it is too weak to derive useful CNF simplification algorithms from it. In the following, we consider a construction that gives us stronger redundancy properties. It was introduced in [45].

Definition 5.4. *Let C be a clause.* Asymmetric Literal Addition *of C with respect to a CNF formula ϕ defines the smallest set $\mathrm{ALA}_\phi(C)$ so that $C \subseteq \mathrm{ALA}_\phi(C)$ and if $c_1, \dots, c_\ell \in \mathrm{ALA}_\phi(C)$ and there is a clause $\{c_1, \dots, c_\ell, \bar{d}\}$ in ϕ then $d \in \mathrm{ALA}_\phi(C)$. The set $\mathrm{HLA}_\phi(C)$, denoted* Hidden Literal Addition*, is defined in the same way, with ℓ restricted to $\ell = 1$ (i.e. only binary clauses of ϕ are considered).*

Let $\mathcal{P}$ be a clause property. We define the associated asymmetric *property $\mathrm{A}\mathcal{P}$ as follows. Let C be a clause. C has property $\mathrm{A}\mathcal{P}$ if and only if $\mathrm{ALA}_\phi(C)$ has property P. We similarly define the* hidden *property $\mathrm{H}\mathcal{P}$. For the tautology property T, this defines the* asymmetric tautology *property* AT *as well as the* hidden tautology *property* HT.

Suppose that property $\mathcal{P}$ in definition 5.4 is *stable under literal addition*, that is, if a clause C has property $\mathcal{P}$, then every superset of C must also have property $\mathcal{P}$. In this case, $\mathcal{P} \preceq \mathrm{H}\mathcal{P} \preceq \mathrm{A}\mathcal{P}$ and the latter two properties are also stable under literal addition. Clearly, this assumption is satisfied for the tautology property T.

One might wonder why we consider $\mathrm{H}\mathcal{P}$ when $\mathrm{A}\mathcal{P}$ is stronger and also computable in polynomial time. The reason is that there are often better algorithms for $\mathrm{H}\mathcal{P}$ than for $\mathrm{A}\mathcal{P}$ as HLA_ϕ can be constructed by traversing the binary implication graph of ϕ, while computing ALA_ϕ is equivalent to full unit propagation. This will be made clear in the next proof.

We still need to check that $\mathrm{H}\mathcal{P}$ and $\mathrm{A}\mathcal{P}$ define redundancy properties. This is already proven in [45].

Lemma 5.5. *For every CNF formula ϕ and every clause C, it holds that $\phi \cup \{C\}$ is equivalent to $\phi \cup \{\mathrm{ALA}_\phi(C)\}$. In particular, $\mathcal{P} \preceq \mathrm{Red}_{\mathrm{equiv}}$ implies $\mathrm{A}\mathcal{P} \preceq \mathrm{Red}_{\mathrm{equiv}}$ and $\mathcal{P} \preceq \mathrm{Red}$ implies $\mathrm{A}\mathcal{P} \preceq \mathrm{Red}$. The same holds for HLA_ϕ and $H\mathcal{P}$.*

Proof. Let τ be a model of ϕ. Per construction, every model of $\{C\}$ is a model of $\{\mathrm{ALA}_\phi(C)\}$. On the other hand, assume that τ is not a model of $\{C\}$. Observe that $\mathrm{ALA}_\phi(C)$ contains exactly the inverses of the literals in $\mathrm{UP}_\phi(\{\bar{a} : a \in C\})$. As the literals $\bar{a}$ for $a \in C$ are all true under τ and τ is a model of ϕ, $\mathrm{UP}_\phi(\{\bar{a} : a \in C\})$ is a subset of τ, so τ is not a model of $\mathrm{ALA}_\phi(C)$.

The statement for HLA follows directly from $\mathrm{HLA}_\phi(C) \subseteq \mathrm{ALA}_\phi(C)$. □

The proof of lemma 5.5 gives another intuition for asymmetric tautologies: A clause C is an asymmetric tautology if and only if unit propagation in ϕ, starting with the assignment $\{\bar{a} : a \in C\}$ derives a conflict.

As an example for clauses that have the AT property, consider the following:

Lemma 5.6. *Let $C, C' \in \phi$ with $x \in C$ and $\bar{x} \in C'$. Then the resolvent $C \otimes_x C'$ has the* AT *property.*

Proof. It is clear that $\mathrm{ALA}_\phi(C \otimes_x C')$ contains both x and $\bar{x}$ as C and C' contribute those literals to $\mathrm{ALA}_\phi(C \otimes_x C')$. $\square$

However, it should be noted that this does not apply to resolvents that require more than one resolution step to be derived.

While the asymmetric tautology property is already strong enough to build useful simplification techniques (e.g. as a simplification technique, we can remove all clauses that have the AT property), it is still not strong enough to model all simplification techniques as the addition and removal of ATs. The following extension will enable us to do this:

Definition 5.7. *Let $\mathcal{P}$ be a clause property. The clause C has property* $\mathrm{R}\mathcal{P}$ *with respect to ϕ if and only if C itself has property $\mathcal{P}$ with respect to ϕ or there is a literal $a \in C$ so that for every clause $C' \in \phi$ with $\bar{a} \in C'$, the resolvent of C and C' has property $\mathcal{P}$ with respect to ϕ. In the latter case C is called a-*$\mathrm{R}\mathcal{P}$*.* $\mathrm{R}\mathcal{P}$ *is called the* resolution *version of $\mathcal{P}$. For* AT*, this defines the* resolution asymmetric tautology (RAT) *property.*

The next lemma proves that properties $\mathrm{R}\mathcal{P}$ that are constructed by this mechanism are indeed redundancy properties, for a large class of choices for $\mathcal{P}$. In [58] this lemma was proven for $\mathcal{P} = \mathrm{AT}$. We prove it here with greater generality.

Lemma 5.8. *If $\mathcal{P} \preceq \mathrm{Red}_{\mathrm{equiv}}$, then $\mathrm{R}\mathcal{P} \preceq \mathrm{Red}$.*

Furthermore, if C is a-$\mathrm{R}\mathcal{P}$ with respect to ϕ, and τ is a model of ϕ, then either τ or $(\tau \setminus \{\bar{a}\}) \cup \{a\}$ satisfies $\phi \cup \{C\}$.

Proof. Let C be a clause that has property $\mathrm{R}\mathcal{P}$ with respect to ϕ. We need to show that ϕ and $\phi \cup \{C\}$ are satisfiability-equivalent. If C is a tautology, there is nothing to prove, so we consider the case that C is not a tautology.

If C itself has property $\mathcal{P}$, there is again nothing to show. Therefore, we consider the case that C has property a-$\mathrm{R}\mathcal{P}$. Let $\phi_{\bar{a}} \subseteq \phi$ be the set of all clauses that contain $\bar{a}$. As $C \otimes_a C'$ has property $\mathcal{P}$, ϕ and $\phi \cup \{C \otimes_a C'\}$ are equivalent, for every $C' \in \phi_{\bar{a}}$. This means that ϕ and $\phi \cup \{C \otimes_a C' : C' \in \phi_{\bar{a}}\}$ are equivalent.

Let τ be a model of ϕ (so τ also satisfies $C \otimes_a C'$ for all $C' \in \phi_{\bar{a}}$). If τ satisfies C, there is nothing to prove. We consider the case where τ does not satisfy C. In this case all $C \otimes_a C'$ with $C' \in \phi_{\bar{a}}$ are satisfied by a literal from $C' \setminus \{\bar{a}\}$ (since $\bar{a} \notin C \otimes_a C'$, otherwise C would be a tautology and we already eliminated that case). So all those C' are still satisfied by $\tau' := (\tau \setminus \{\bar{a}\}) \cup \{a\}$. But τ' also satisfies C. On the other hand, if τ is a model of $\phi \cup \{C\}$, then τ is obviously also a model of ϕ.

As the above argument holds for every clause C, this concludes the proof. □

In addition to the correctness guarantee, lemma 5.8 gives a simple way to reconstruct solutions if clauses with property R$\mathcal{P}$ are removed from the CNF formula.

An obvious question is, whether the requirement $\mathcal{P} \preceq \text{Red}_{\text{equiv}}$ in the statement of lemma 5.8 can be relaxed to $\mathcal{P} \preceq$ Red. However, this is not the case. To demonstrate that, it suffices to show that RRed is not a redundancy property. Consider the satisfiable formula $\phi = \{\{\bar{a}, x\}, \{\bar{a}, \bar{x}\}\}$. The clause $\{a\}$ does not have property Red but RRed can add $\{a\}$ to ϕ (as the resolvents $\{x\}$ and $\{\bar{x}\}$ both have property Red with respect to ϕ), resulting in an unsatisfiable formula.

5.1.2 The RAT deduction system

In the following, we discuss the RAT deduction system that allows modeling both CDCL search and simplification algorithms. The system is based on the AT and RAT properties from the last subsection and will allow us to prove the correctness of simplification techniques.

The RAT deduction system deals with RAT *triples* of the form $\phi\,[\rho]\,\sigma$ where ϕ and ρ are CNF formulas, with ϕ and $\phi \cup \rho$ being satisfiability-equivalent, and σ is a finite sequence $(a_1, C_1)(a_2, C_2) \dots (a_\ell, C_\ell)$ of literal-clause pairs so that $a_i \in C_i$ for all $i = 1, 2, \dots, \ell$. ϕ is the set of *irredundant* clauses while ρ is the set of *redundant* clauses. σ is the *reconstruction sequence*. This sequence allows us to reconstruct models of the original formula from models of ϕ, using the principle of lemma 5.8. The intuition is that the SAT solver operates on the formula $\phi \cup \rho$; however, clauses from ρ might be dropped at any moment, while clauses from ϕ must not be deleted. In other words, the notions of redundant and irredundant clauses match those that we discussed in the context of CDCL in chapter 3.

The deduction rules of the RAT deduction system are depicted in figure 5.1. They operate on RAT triples by transforming them. Most of the rules have a precondition that needs to be satisfied before the rule can be applied. The rules are defined as follows:

$$\frac{\phi\,[\rho]\,\sigma}{\phi\,[\rho \cup \{C\}]\,\sigma}\ \textsc{Learn}(C) \qquad \frac{\phi\,[\rho \cup \{C\}]\,\sigma}{\phi\,[\rho]\,\sigma}\ \textsc{Forget}(C) \qquad \frac{\phi\,[\rho \cup \{C\}]\,\sigma}{\phi \cup \{C\}\,[\rho]\,\sigma}\ \textsc{Strengthen}(C)$$

$$\frac{\phi \cup \{C\}\,[\rho]\,\sigma}{\phi\,[\rho \cup \{C\}]\,\sigma}\ \text{AT-}\textsc{Weaken}(C) \qquad \frac{\phi \cup \{C\}\,[\rho]\,\sigma}{\phi\,[\rho \cup \{C\}]\,(a, C)\ \sigma}\ \text{RAT-}\textsc{Weaken}(a, C)$$

Figure 5.1: Deduction rules of the RAT deduction system

LEARN(C) Allows addition of a new redundant clause. The precondition is that C has to be RAT with respect to $\phi \cup \rho$. For correctness, it is important that ρ is considered here[1].

FORGET(C) Removes a redundant clause. Matching our intuition, redundant clauses can be removed at any moment, so there is no precondition.

STRENGTHEN(C) Turns a redundant clause into an irredundant one. There is no precondition to the STRENGTHEN rule, however it is not directly reversible.

AT-WEAKEN(C) Turns an irredundant AT clause C into a redundant clause. The precondition is that C has to be AT with respect to ϕ.

RAT-WEAKEN(a, C) Turns a irredundant a-RAT clause C into a redundant clause. The precondition is that C has to be a-RAT with respect to ϕ. This is the only rule that changes the reconstruction sequence.

It is easy to see that the RAT deduction rules preserve satisfiability-equivalence. Formally, if $\phi\,[\rho]\,\sigma$ is transformed to $\phi'\,[\rho']\,\sigma'$ by the application of a RAT deduction rule, then ϕ and ϕ' are satisfiability-equivalent (and thus $\phi \cup \rho$ and $\phi' \cup \rho'$ are also satisfiability-equivalent to these formulas). This follows directly from lemmas 5.5 and 5.8.

In some situations, σ will not matter when dealing with RAT triples. If that is the case, we will drop σ from the notation and just write $\phi\,[\rho]$.

It turns out that the RAT deduction system is able to model all CNF transformations that are performed by a CDCL-based SAT solver. In particular, it is strong enough to model clause learning and most of the estab-

[1]Consider a formula that has two models. Learning a clause that excludes one of those models is fine as long as the other model is preserved. If ρ was not considered during learning, two clauses could be learnt that each exclude one on the models.

lished simplification techniques. For example, it is easy to see that every learned clause is an asymmetric tautology by construction.

5.1.3 Other redundancy properties

Instead of starting with the tautology property T and defining other properties based on that, one can also consider other redundancy properties. Here, we discuss the subsumption property that forms the theorical basis for the subsumption elimination technique that we will discuss in the next section.

Definition 5.9. *Let C be a clause. We say that C is* subsumed *in ϕ, if and only if there is a clause $C' \in \phi$ with $C \supseteq C'$. This defines the* subsumption *(*S*) property. Using the definitions from section 5.1.1 we also define* HS*,* AS*,* RHS *and* RAS*.*

The S property is independent of the T property. However, it is easy to see that the following relation (that was proved by [45]) between AS and AT holds.

Lemma 5.10. AS $\preceq$ AT *and thus* RAS $\preceq$ RAT*.*

Proof. Let C be *AS* with respect to ϕ. That means that $\mathrm{ALA}_\phi(C)$ is subsumed by some other clause $D \in \phi$. But then D contributes the inverses of all its literals to $\mathrm{ALA}_\phi(C)$ and that set is a tautology. □

5.2 A survey of simplification techniques

In this section, we review most of the established simplification techniques. This prepares for the study of their parallelizability.

5.2.1 CNF simplification as RAT deduction

We start by giving a general definition of the term "simplification technique" and by discussing some basic properties of simplification techniques.

As we have already remarked in section 5.1.2, the most commonly used CNF simplification techniques can all be modelled as RAT deduction. This motivates us to define the term "simplification technique" via RAT deduction.

Definition 5.11. *A* simplification technique *or* simplification step[2] *is a function that takes a* RAT *triple $\phi\,[\rho]\,\sigma$ and maps it to a finite sequence of applicable* RAT *deduction rules, starting from $\phi\,[\rho]\,\sigma$.*

We remark that our definition of a simplification technique is somewhat unusual given the fact that most authors see simplification techniques as algorithms that operate on ϕ and/or ρ and not as functions that produce RAT triples. However, we will see that this notion enables us to rigorously study properties like the parallelizability of simplification techniques.

In practice we also want the f from the previous definiton to be efficiently computable. Here, efficiency is not synonymous with polynomial time - especially in the sequential world, that might be too slow. It is desirable to have linear or almost linear algorithms to be able to handle industrial CNF formulas with millions of clauses and variables.

Simplification steps can be chained, by applying the second simplification step to the RAT triple that is produced by applying the result of the first simplification step to the input $\phi\,[\rho]\,\sigma$.

We will see that most of the well-known simplification techniques apply a certain set of rules until a fixed point is reached. The following definition distinguishes simplification techniques that have a unique fixed point.

Definition 5.12. *Let s be a simplification technique. If s has a natural fixed point in the sense that this fixed point does not depend on arbitrary choices like the order in which variables or clauses are considered, for every input triple $\phi\,[\rho]$, we call s* confluent*.*

Confluence is mostly interesting from a theoretical point of view. It can be argued that the confluence of simplification techniques is also useful practically as it implies that no additional heuristics are required to implement the technique. It is not primarily important for our purpose but we discuss it for the sake of completeness.

5.2.2 Quality of simplification techniques

We shortly discuss means of estimating the quality of simplification techniques.

What is missing from the discussion so far are criteria for the quality of simplification techniques. Unfortunately such criteria are difficult to state. Instead, the quality of such techniques is often evaluated empirically. In

[2]Technically, we define simplification techniques and simplification steps identically. We do this as simplification techniques are conventionally understood as algorithms that simplify large parts of the formula, while we also need to consider simplifications that only affect fragments of the formula (e.g. single clauses).

practice, many techniques either try to reduce the size of the formula by removing redundant clauses or they try to fix variables to constrain the search space.

Nevertheless, we try to define at least some criteria. One property that is important from the CDCL algorithm's point of view is that unit propagation is not obstructed by simplification algorithms.

Definition 5.13. *We call a simplification step* BCP-preserving *if and only if* $\mathrm{UP}_{\phi\cup\rho}(\tau) \subseteq \mathrm{UP}_{\phi'\cup\rho'}(\tau)$ *holds for every input triple* $\phi\,[\rho]$*, result triple* $\phi'\,[\rho']$ *and every partial assignment* τ*.*

We will later see that most of the commonly used simplification techniques indeed preserve unit propagation. Some of these results are already studied in [45].

Another approach of evaluating simplification techniques theoretically is to examine their relative strength. For simplification techniques that eliminate redundant clauses, we already introduced such a measure through the $\preceq$ relation. It is less straightforward to generalize this definition to simplification techniques that do introduce new clauses[3]. Instead of trying to give a general definition, we will informally state whether a simplification technique S *simulates* another technique, with the precise meaning depending on the situation.

5.2.3 Clause elimination techniques

In this subsection, we review simplification techniques that remove clauses from the input formula.

Each clause redundancy property $\mathcal{P} \preceq AT$ canonically defines a $\mathcal{P}$ *elimination* technique that removes all clauses $C \in \phi$ that have property P with respect to ϕ using AT-Weaken, possibly followed by forgetting all clauses $C \in \rho$ that have $\mathcal{P}$ with respect to $\phi \cup \rho$ by Forget. This procedure is repeated until a fixed point is reached. Similarly, for $\mathcal{P} \preceq RAT$, a canonical elimination technique based on RAT-Weaken can be defined.

In the context of a CDCL solver, however, not all $\mathcal{P}$ elimination techniques will improve CDCL performance. For example, AT elimination is strong enough to remove all learned clauses from the formula. Certainly, forgetting all learned clauses will degrade CDCL performance. In contrast to that, for other redundancy properties like S, forgetting all S clauses will indeed improve performance. In general, forgetting redundant clauses will be beneficial if BCP is still preserved afterwards.

[3]The $\preceq$ relation does not yield a good measure for clause quality here: For example, we expect performance to drop dramatically if we just add all asymmetric tautologies to a formula.

Table 5.1: Well-known clause elimination techniques

Property	Elimination technique	Forget
T	Tautology elimination	Yes
S	Subsumption	Yes
RT	Blocked clause elimination (BCE)	Yes
HT	Hidden tautology elimination (HTE)	Yes
AT	Asymmetric tautology elimination (ATE)	No

While we could define an elimination technique for each of the redundancy properties that we studied before, we concentrate on well-known simplification techniques first. Figure 5.1 lists commonly applied clause elimination techniques.

Tautology elimination can be considered as a CNF simplification technique but it is usually not applied explicitly in practice. Instead, CDCL solvers discard tautologies when parsing their input and never produce tautologies themselves (e.g. as part of clause learning). Tautology elimination obviously preserves BCP. As the property T does not depend on any clauses of ϕ but only on the given clause itself, tautology elimination is also confluent.

Subsumption as a CNF simplification technique was introduced in [33] as part of the successful SatELite preprocessor. SatELite efficiently implements *backward* subsumption, that is finding all clauses that are subsumed by a given clause C. This is made possible by observing that to find those clauses, it suffices to iterate through the shortest occurrence list of any literal in C and by using a bloom filter[4] to speed up the check if C is contained in some other clause. Similar to tautology elimination, it is clear that subsumption preserves BCP. Because of the transitivity of the subset relation, subsumption is confluent: If C is subsumed by C' and C' itself is subsumed by some other clause, then C can be eliminated even after removing C'.

Blocked clause elimination as a simplification technique was first considered in [57]. It is based on the resolution tautology property RT which is also called the blocked clause property: If C is a-RT we also say that C is a *blocked clause* and a is a literal that *blocks* C. This notation was introduced much earlier [62] than the notation of R$\mathcal{P}$ for arbitrary $\mathcal{P}$ in [58].

It is easy to construct examples in which BCE does not preserve BCP. However, blocked clauses do not contribute to unit propagation in the following sense: Let C be a clause and $a \in C$ be a literal that blocks C. If a is propagated by C, no additional literals will be fixed by UP as every

[4]Implementations typically store a 64-bit *signature* $\text{sig}(C)$ of each clause C, where bit i of $\text{sig}(C)$ is set if and only if there is a variable with index congruent to $i \bmod 64$ in C.

clause that contains $\bar{a}$ is already satisfied by another literal. This holds as one of the literals of those clauses must occur inverted in C. On the other hand, if the rest of the formula propagates the literal $\bar{a}$ then the clause C is not unit as it is satisfied by some other literal. This means that BCE can only obstruct the propagation of a literal that blocks the clause in question and even if that does happen, no other propagations (or conflicts) would be possible. Therefore, forgetting blocked clauses is always beneficial in a CDCL solver.

BCE is confluent. Let C be a-RT. All resolvents of C by literal a are tautologies. Furthermore, they are still tautologies after clauses from ϕ have been removed. Hence, C is still a-RT in this case.

Hidden tautology elimination was introduced in [45]. HTE can be efficiently approximated by the *unhiding* algorithm [47] that is based on a depth-first search through the binary implication graph. The paper [45] proved that HTE is BCP preserving if the binary implication graph of the input CNF formula does not contain failed literals.

Asymmetric tautology elimination was first described in [42] although the notion of asymmetric tautologies did not exist when that paper was written. This paper defined a *distillation* algorithm that performs ATE and simulates some other simplification techniques. The term "asymmetric tautology elimination" was later defined in [45]. While the authors of [42] observed that distillation improves CDCL runtime on some hard CNF instances, they could not demonstrate that distillation performs well on average over large sets of benchmarks. The reason for that is that distillation is quite expensive (compared with other simplification techniques and even with CDCL search) and that ATE can actually remove useful clauses as it does not preserve BCP.

5.2.4 Clause addition techniques

We now consider simplification techniques that add clauses to the formula.

Similar to the case clause elimination techniques, each clause redundancy property $\mathcal{P}$ also defines a $\mathcal{P}$ *addition* technique. This technique adds all clauses that have property $\mathcal{P}$ with respect to $\phi \cup \rho$ via Learn to ρ, possibly followed by Strengthen to move them to ϕ. This procedure is repeated until a fixed point is reached.

In contrast to clause elimination techniques, it is less clear that adding redundant clauses improves CDCL performance. Certainly, for large classes of redundancy properties $\mathcal{P}$, including T or S, performing $\mathcal{P}$ addition decreases CDCL performance (otherwise there would be no point in eliminating these clauses by clause elimination techniques). Instead of considering these redundancy properties, we need to consider properties that only affect "useful" clauses and improve BCP performance. Just preserving BCP is not enough to be useful, as all clause addition techniques naturally preserve BCP.

We say that a clause $\{a\}$ has the *failed literal* (FL) property with respect to ϕ, if and only if $\mathrm{UP}_\phi(\{\bar{a}\})$ is conflicting. a is called the failed literal in this case[5]. It is easy to see that FL $\preceq$ AT, as a clause C is AT with respect to ϕ exactly when $\mathrm{ALA}_\phi(C)$ is conflicting. This implies that FL is indeed a redundancy property. *Failed literal elimination (FLE)* is the clause addition technique that is based on the failed literal property. In order to avoid adding superfluous clauses, we further restrict failed literal elimination on $\phi\,[\rho]$ to clauses $\{a\}$ with $a \notin \mathrm{UP}_{\phi\cup\rho}(\varnothing)$.

There are multiple algorithms for FLE. The naive probing based algorithm simply assigns all $2n$ literals iterativly and performs unit propagation to determine whether some of those literals are failed literals. Another method to find a subset of all failed literals is the unhiding algorithm [47] that runs a depth-first search in the binary implication graph of ϕ to find implications of the form $\bar{a} \to a$. Due to this approach, unhiding is unable to find failed literals that require inspecting non-binary clauses.

The naive probing algorithm can be improved by noticing that if $\mathrm{UP}_\phi(\{\bar{a}\})$ contains a literal b and a is not a failed literal, then b cannot be a failed literal either. If there are many of such propagations (for example because ϕ is an industrial CNF formula with a large binary implication graph), this optimization greatly reduces the number of necessary probes. However, it is not enough to get acceptable performance.

Another interesting redundancy property for clause additon is the hyper-binary property. We say that a clause $C = \{a, b\}$ has the *hyper-binary* (HB[6]) property with respect to ϕ if and only if there is a clause $\{a, c_1, \ldots, c_\ell\} \in \phi$ and there are paths $\bar{c}_i \to b$ in the binary implication graph of ϕ, for each $i = 1, \ldots, \ell$. In other words, a binary clause C has the hyper-binary property if C can be generated by repeatedly resolving a (not necessarily binary) clause from ϕ with binary clauses from ϕ. Similar to the failed literal case, it is easily verified that HB is a redundancy property; it suffices to observe that HB $\preceq$ AT. Thus HB defines the *hyper-binary resolution* (HBR) clause addition technique. Again, we have to restrict the technique to clauses $\{a, b\}$ so that $\bar{a} \to b$ (or equivalently $\bar{b} \to a$) is not already a path in the binary implication graph of $\phi \cup \rho$, for an input triple $\phi\,[\rho]$. Without this restriction, hyper-binary resolution would add

[5]Instead saying that $\{a\}$ has the failed literal property, one usually states that a is a failed literal. However, viewing the failed literal property as a clause redundancy property unifies the presentation of failed literal elimination and other clause addition techniques.

[6]The H at the beginning of HB should not be confused with the H in HT, HTE and similar terms where it stands for *hidden*.

the transitive closure of the binary implication graph to ϕ, which decreases the performance of CDCL as transitive clauses do not make additional unit propagations possible.

Both FLE and HBR can be efficiently computed by the tree-based lookahead algorithm from chapter 2. Failed literal elimination can trivially be included in the procedure. For HBR, we need to propagate all binary clauses before clauses of length greater than two are propagated. If unit propagation still fixes a literal during propagation of a non-binary clause, a hyper-binary clause can be added to the formula. We refer the reader to [46] for more details.

5.2.5 Clause shortening techniques

The last remaining category of simplification techniques that operate on individual clauses are clause shortening techniques; we shortly discuss them in this subsection.

Clause shortening techniques are defined as simplification techniques that first apply $\textsc{Learn}(C')$ and $\textsc{Strengthen}(C')$ to add a new clause $C' \subseteq C$ to ϕ and then use AT-$\textsc{Weaken}(C)$ and $\textsc{Forget}(C)$ to remove C. The $\textsc{Strengthen}$ and AT-$\textsc{Weaken}$ rules must of course be dropped if we want to shorten redundant clauses. Theoretically, it would be possible to also define RAT based clause shortening techniques, however, we do not encounter any of such techniques in practice. By construction, all clause shortening techniques preserve BCP. In fact, we can hope that BCP is actually improved by shortening clauses, which is the main motivation to consider such techniques.

The first clause shortening technique that we will consider is self-subsumption. The technique was introduced by SatELite [33]. *Self-subsumption*[7] replaces $C \in \phi$ by its resolvent $C \otimes D$ with some other clause $D \in \phi$, if $C \otimes D \subseteq C$. Applying $\textsc{Learn}$ to add $C \otimes D$ is well-defined because of lemma 5.6. After $C \otimes D$ is added, $C \supseteq C \otimes D$ has the subsumption property S so AT-$\textsc{Weaken}(C)$ can be applied.

Note that by design, self-subsumption will remove a single literal from a given clause in each step. However, this literal is not uniquely defined and self-subsumption is not confluent, as demonstrated by the following example.

Example 5.14. *Consider the formula $\phi = \{\{x, y, z\}, \{\bar{x}, y, z\}, \{x, \bar{y}, z\}\}$ and let C be the first clause. Self-subsumption can either remove x or y but not both from C.*

[7]Self-subsumption is often called “resolution subsumption” or “self-subsuming resolution”. We do not use this term here to avoid confusion with the resolution subsumption property RS.

Similar to the subsumption algorithm that we discussed in section 5.2.3 *backward* self-subsumption can be efficiently implemented by traversing only the literal with shortest occurrence list and using bloom filters. By backward self-subsumption we mean the problem of finding all clauses, that can be shortened by resolving them with a given clause. Again, this trick was invented by the author's of SatELite [33].

Other clause shortening techniques are hidden and asymmetric literal elimination. *Hidden* and *asymmetric* literal elimination replace clauses with subsets that have the same HLA or ALA set as the original clause. This is possible because if C and C' are clauses with $\mathrm{HLA}_\phi(C') = \mathrm{HLA}_\phi(C)$, then C' has the HT property with respect to $\phi \cup \{C\}$: As $\mathrm{HLA}_\phi(C') = \mathrm{HLA}_\phi(C) \supseteq C$, the clause C contributes the inverses of all its literals to $\mathrm{HLA}_{\phi\cup\{C\}}(C')$, so that set becomes a tautology. The same holds for ALA and the AT property.

Computing hidden and symmetric literal elimination can be done by similar algorithms as the related hidden and symmteric tautology elimination techniques. Hidden literal elimination (as well as hidden tautology elimination) can be computed during the unhiding algorithm [47]. Asymmetric literal elimination (as well as asymmetric tautology elimination) can be computed by distillation [42].

5.2.6 Variable elimination and addition techniques

This subsection will discuss simplification techniques that do not add, shorten or remove individual clauses but operate on multiple clauses. While the techniques are harder to categorize than the ones we studied, they all operate on clauses that share a specific variable.

Bounded variable elimination (BVE) is a well-known CNF simplification method that is applied in most CDCL based SAT solvers. It was introduced in [83] and is also part of the SatELite preprocessor [33]. Bounded variable elimination applies Davis-Putnam resolution to selected variables of the CNF formula ϕ. Here, *Davis-Putnam resolution* means applying resolution to all pairs of clauses that contain a given variable. The resulting resolvents are then added to ϕ while the original clauses are removed.

Unrestricted Davis-Putnam resolution is a complete procedure to solve SAT on CNF formulas. Specificially, it has exponential space and time requirements. Bounded variable elimination therefore restricts the set of variables that are considered for Davis-Putnam resolution so that a variable is only eliminated if its elimination decreases the number of clauses in the formula.

If we think of BVE as RAT deduction, BVE first applies $\textsc{Learn}(C \otimes_x C')$ and $\textsc{Strengthen}(C \otimes_x C')$ for all clauses C and C' with $x \in C$,

$\bar{x} \in C'$. After that, RAT-Weaken(x, C) and Forget(C) as well as RAT-Weaken$(\bar{x}, C')$ and Forget(C') is used to remove the original clauses. Strengthen is only applied if both C and C' are irredundant clauses. Likewise, RAT-Weaken() is only applied if C or C' are irredundant clauses. The Learn rules can be applied because of lemma 5.6. RAT-Weaken can be applied because after addition of $C \otimes_x C'$ for all C', the clause C has the $x - \mathrm{RS}$ property: All resolvents of C are already present as clauses in the formula. The same holds for C' as C' has the $\bar{x} - \mathrm{RS}$ property.

The inverse of variable elimination is variable addition. Suppose that a CNF formula contains nm clauses of the form $C_i \cup D_j$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Variable addition [68] introduces a new variable x and replaces all $C_i \cup D_j$ by $C_i \cup \{x\}$ and $D_j \cup \{\bar{x}\}$. Similar to BVE, variable addition is applied in a bounded fashion: *Bounded variable addition* (BVA) only introduces new variables if the number of clauses decreases (i.e. if $m + n < nm$).

In the RAT deduction system, BVA is done analogous to BVE: First, Learn is used to introduce the new clauses. This is possible because all new clauses are $x - \mathrm{RS}$ and $\bar{x} - \mathrm{RS}$ respectively. After that, AT-Weaken can be used to remove all clauses from the $C_i \cup D_j$.

5.3 Parallelization

Finally, the theory that we developed in the previous sections will allow us to reason about parallel simplification techniques. In this section we will define a notion of easily parallelizable simplification techniques. After that, we will inspect the simplification techniques that we introduced in the last section for parallelizability and present parallel algorithms to compute some of those techniques.

5.3.1 CNF-parallelization

In this subsection, we discuss conditions that allow different simplification steps to be run in parallel.

We want to be able to compute the modifications of those simplification steps to the CNF formula individually on different workers and then merge these modifications and apply them in all workers. The next definition captures this idea.

Definition 5.15. *Let $S = \{s_1, \ldots, s_\ell\}$ be simplification steps. A new simplification step that maps $\phi\,[\rho]\,\sigma$ to the concatenation $s_1(\phi\,[\rho]\,\sigma) \ldots s_\ell(\phi\,[\rho]\,\sigma)$ of sequences of deduction rules, is called a* CNF-parallelization *of S. Note that there is a CNF-parallelization for every choice of indices. If all those*

CNF-parallelizations are valid simplification techniques, that is, if the application of all sequences $s_1(\phi\,[\rho]\,\sigma)\dots s_\ell(\phi\,[\rho]\,\sigma)$ to $\phi\,[\rho]\,\sigma$ is well-defined, then S is called CNF-parallelizable*.*

We illustrate the concept of CNF-parallelizability by giving a simple example.

Example 5.16. *Consider the tautology technique. Instead of applying tautology elimination as defined in 5.2.3 to a formula ϕ we consider the collection $\{s_C : C \text{ is a tautology}\}$ of simplification steps, where $s_C(\phi\,[\rho])$ is the sequence*

$$\text{AT-Weaken}(C)\ \text{Forget}(C)$$

if $C \in \phi$ and the empty sequence otherwise. The elements of this collection can be applied in an order to $\phi\,[\rho]$ to recover the tautology elimination technique. In this sense, tautology elimination is CNF-parallelizable.

In the remainder of this subsection, we will discuss conditions under which simplification techniques from section 5.2 are CNF-parallelizable. As a first result, it is easy to see that AT-based clause addition techniques are always CNF-parallelizable.

Lemma 5.17. *Let $s_C(\phi\,[\rho])$ be the sequence*

$$\text{Learn}(C)\ \text{Strengthen}(C)$$

if C is AT *with respect to $\phi \cup \rho$. Any set of s_C is CNF-parallelizable.*

Proof. It is clear from the definition that adding clauses to a formula never causes clauses to lose the AT property. More generally, adding clauses to a formula ψ can only add literals to $\mathrm{ALA}_\psi(C)$ but never remove literals. □

Furthermore, it turns out that AT-based clause shortening techniques, as presented in 5.2.5 are always CNF-parallelizable. To see this, we need the following lemma:

Lemma 5.18. *Let ϕ be a CNF formula and C, D be clauses. Let $D' \subseteq D$ be some subset, $D \neq \varnothing$. Furthermore, let C be* AT *with respect to $\phi \cup \{D\}$. Then C is also* AT *with respect to $\phi \cup \{D'\}$.*

Proof. Recall that C is AT if and only if $\mathrm{ALA}_{\phi\cup\{D\}}(C)$ is a tautology. We perform induction over the contributions of D to $\mathrm{ALA}_{\phi\cup\{D\}}(C)$. If the clause D does not contribute to $\mathrm{ALA}_{\phi\cup\{D\}}(C)$ then $\mathrm{ALA}_{\phi\cup\{D'\}}(C) \supseteq \mathrm{ALA}_{\phi\cup\{D\}}(C)$ and both of these sets are tautologies. Note that in the last relation, we only get $\supseteq$ and not equality as D' can still contribute to $\mathrm{ALA}_{\phi\cup\{D'\}}(C)$.

Consider the case where $D = \{\bar{a}, d_1, \dots, d_k\}$ contributes the literal a to $\text{ALA}_{\phi \cup \{D\}}(C)$. If $a \in D'$, then we can replace this contribution by the contribution of D' and continue with the next contribution of D to $\text{ALA}_{\phi \cup \{D\}}(C)$. Otherwise we have $D' \subseteq \{d_1, \dots, d_k\} \subseteq \text{ALA}_\phi(C)$ and D' contributes the inverses of all of its literals to $\text{ALA}_{\phi \cup \{D'\}}(C)$ so that set is a tautology. □

Now we can formulate the CNF-parallelizability of clause shortening techniques as a corollary.

Corollary 5.19. *For two clauses C and C' with $C \supseteq C'$, let $s_{C,C'}(\phi\,[\rho])$ be the sequence*

$$\text{Learn}(C')\ \text{Strengthen}(C')\ \text{AT-Weaken}(C)\ \text{Forget}(C)$$

if $C \in \phi$ and C' is AT *with respect to $\phi \cup \rho$ and the empty sequence otherwise. In other words, $s_{C,C'}$ is a single step of an* AT*-based clause shortening technique. All sets of $s_{C,C'}$ with pairwise different C are CNF-parallelizable.*

Note that the pairwise-different condition is required to avoid deleting the same clause multiple times.

Proof. Applying $s_{C,C'}$ to $\phi\,[\rho]$ shortens clauses from ϕ and leaves ρ invariant. The lemma immediately gives us the desired result. □

It should be noted that the CNF-parallelization from corollary 5.19 generally does not recover the full sequential algorithm. For example, taking the $s_{C,C'}$ that are induced by resolution subsumption and applying them as a CNF-parallelization is not equivalent to resolution subsumption until a fixed point is reached. The reason for this is that applying a single resolution subsumption might enable additional resolution subsumptions later on.

In contrast to these strong results for clause addition and shortening techniques, we need to impose more constraints on CNF-parallelizable clause elimination techniques.

Lemma 5.20. *Let $\mathcal{P}$ be a clause property that is* stable under clause removal*, meaning that if C has property $\mathcal{P}$ with respect to some formula ψ, then we require C to also have property $\mathcal{P}$ with respect to any $\psi' \subseteq \psi$.*

Furthermore, if $\mathcal{P} \preceq$ AT*, let $s_{\mathcal{P},C}(\phi\,[\rho])$ be the sequence*

$$\text{AT-Weaken}(C)\ \text{Forget}(C)$$

*if $C \in \phi$ and C has property $\mathcal{P}$ with respect to ϕ, and otherwise the empty sequence. If $\mathcal{P} \preceq a$-*RAT *define $s_{a,\mathcal{P},C}(\phi\,[\rho])$ similarly by the sequence*

$$\text{RAT-Weaken}(a, C)\ \text{Forget}(C)$$

Then any set of $s_{\mathcal{P},C}$ and $s_{a,\mathcal{P},C}$ with pairwise different C is CNF-parallelizable, even for varying $\mathcal{P}$.

Proof. The statement follows directly from the fact that $\mathcal{P}$ is stable under clause removal. □

The interesting task now is to find properties $\mathcal{P}$ which satisfy the stability requirements. Certainly AT is not stable under clause removal: Removing clauses can remove literals from $\text{ALA}_\psi(C)$ and thus leads to clauses C losing the AT property. Indeed, it is easy to construct an example that demonstrates that for AT, the CNF-parallelization from lemma 5.20 is not valid.

However, the RT property is stable under clause removal; this proves that BCE is CNF-parallelizable. This follows from the fact that the T property does not depend on any clauses from the given CNF formula. It is, in a sense, equivalent to the statement that BCE is confluent.

Subsumption is also CNF-parallelizable: While subsumption is not stable under arbitrary clause removal, it is stable under removal of other clauses that also have the subsumption property. As discussed before, this follows directly from the transitivity of the subset relation. The only complication here is that we have to prevent the simultaneous elimination of all copies of a duplicate clause. This can be done by ordering the clauses (e.g. by taking the order in which they appear in the input formula) and restricting subsumption so that a duplicate clause is only eliminated if it is not the first copy of this clause in the chosen order.

5.3.2 Data structures for parallel simplification

We shortly discuss additional data structures that we need to efficiently implement parallel simplification algorithms.

During CNF simplification, we need to be able to globally identify clauses, even if they originate from different workers or if their clause handles are different for different workers[8]. Thus, to perform this identification we introduce a unique *name* for each clause. This name consists of a 64-bit integer where the first 16 bits identify the workers that produced the clause and the remaining 48 bits form a sequential per-worker counter that is incremented whenever a new clause is generated by this worker.

To support our algorithm, we maintain a hash table that maps clause names to proper clause handles. Because of the large number of clauses in industrial formulas, we cannot use a hash table with separate chaining. Instead, we use open addressing with linear probing. We found, that

[8]Remember that clause handles change as the result of garbage collection.

while this scheme provides better performance than other open addressing schemes, its performance drastically degrades if the output of the hash function is not uniformly distributed. We tried several simple hash functions with unsatisfying results and settled for the murmur3_32 [2] function.

5.3.3 Straightforward parallelizations

In this section we will consider relatively simple approaches for parallelizing self-subsumption, subsumption and blocked clause elimination. As the statements in section 5.3.1 guarantee CNF-parallizability for those techniques, implementing actual parallel algorithms for them is easy. The resulting algorithms tend to be embarrassingly parallel.

We model the self-subsumption technique as a set of simplification steps $s_{C,C'}$ as defined in corollary 5.19. Our goal is to compute the results $s_{C,C'}(\phi\,[\rho])$ concurrently on different workers. We have to ensure that all sC are pairwise different. Our approach consists of distributing the clauses C of ϕ over all workers. Each worker can then find a suitable C' for each C, thus computing $s_{C,C'}$. After all $s_{C,C'}$ have been computed, we gather the results (i.e. the names of removed clauses and the literals of new clauses) at all workers.

For blocked clause elimination and subsumption, we model the technique as a set of simplification steps $s_{a,\mathrm{RT},C}$ or $s_{\mathrm{S},C}$ as defined in lemma 5.20. Again, we have to ensure that all C are pairwise different. Similar to our self-subsumption approach, we distribute the clauses of ϕ over all workers. Each worker then tries to find a blocking literal a for each of its assigned clauses C. This computes the result $s_{a,\mathrm{RT},C}$ or $s_{\mathrm{S},C}$, that we have to gather (as a list of names of removed clauses) at all workers afterwards.

5.3.4 Parallel clause addition

We shortly discuss the parallelization of clause addition techniques.

As we already saw a proof for the CNF-parallizability of clause addition techniques, there is not much that still needs to be done. We can integrate FLE and HBR into the parallel tree-based lookahead algorithm that we discussed in chapter 4. In the case of FLE, this algorithm is embarrassingly parallel with the caveat that the size of the trees which the algorithm produces is not balanced. This problem, however, can be solved easily by using a load balancer that takes the size of the trees into account.

Unfortunately, regarding HBR, there is another problem: The size of the output is so large that applying the resulting simplification sequentially consumes non-negligible amounts of time. In experiments, we found that for many industrial problems, millions to tens of millions of binary clauses can be added by HBR. Often, a significant fraction of those clauses can be removed after failed literals are found. Another significant fraction are binary clauses that becomes transitive after HBR completes. These clauses can be removed by unhiding. Therefore, when applying parallel HBR in practice, we apply parallel tree-based lookahead twice; the first run only eleminates failed literals while the second run performs HBR. We remove all satisfied binary clauses from the output before exchanging them with other workers and follow the HBR run by unhiding to remove the transitive clauses.

5.3.5 Parallel distillation

In this section we formulate a parallel algorithm that performs both ATE and ALE.

As noted in 5.2.3, ATE might remove clauses that actually benefit the CDCL solver's performance. To prevent that from happening we do not remove clauses from ϕ. Instead we only mark those clauses as non-essential. The clause database reduction heuristics of the CDCL solver will then remove the clauses if it turns out that they do not help the solver.

As an additional optimization we do not apply ATE and ALE to binary clauses. Applying ATE does not make sense as solvers generally do not remove binary clauses[9] as they improve the number of possible propagations. ALE on binary clauses is simulated by FLE: Consider the case that $C = \{c, d\}$ is reduced to $\{c\}$ by ALE. Then $d \in \text{ALA}_{\phi \setminus \{C\}}(\{c\})$. In other words, unit propagation of $\bar{c}$ (on $\phi \setminus \{C\}$) assigns $\bar{d}$ and $\bar{c}$ is a failed literal.

Our algorithm uses a load balancer to distribute the set of clauses over all workers. Each worker is given a clause C and performs unit propagation to compute $ALA_{\phi \setminus \{C\}}(C)$. The worker then decides if C is an AT or if C can be shortened by ALE. This procedure is iterated until all clauses are processed. After that, all ATs and all clauses shortened by ALE are exchanged by the workers in an all-to-all operation. Each worker applies the changes made by other workers to its own formula.

We have to take into account that applying ATE to two different clauses does not commute. In order to gurantee correctness we track the *dependencies* of each ATE operation: During unit propagation, we track all clauses that contribute to $ALA_{\phi \setminus \{C\}}(C)$. Because we never apply ATE to binary clauses, we only have to track clauses of length > 2. In practice this greatly reduces the number of dependencies as industrial CNF formulas have large 2-CNF subsets.

Algorithm 5.1 shows our distributed distillation algorithm. In line 13 we

[9]This applies to non-transitive binary clauses. Transitive binary clauses are removed by unhiding.

Algorithm 5.1 Distributed distillation

```
 1: procedure distributedDistillation
 2:     $B$ : LoadBalancer
 3:     $R$ : List
 4:     if Rank $= 0$ then
 5:         $\mathcal{U} \leftarrow$ allClauses
 6:         $B$.initialize($\mathcal{U}$)
 7:     end if
 8:     loop
 9:         $C \leftarrow B$.fetch
10:         if $C =$ nil then
11:             break
12:         end if
13:         $(ate, ale, \mathcal{D}, S) \leftarrow$ distill($C$)
14:         $R$.append($(C, ate, ale, \mathcal{D}, S)$)
15:     end loop
16:     $R \leftarrow$ allgather($R$)
17:     $R$.sort        ▷ Sort order has to be consistent over all ranks
18:     $\mathcal{E} = \varnothing$
19:     $\mathcal{F} = \varnothing$
20:     for $(C, ate, ale, \mathcal{D}, S) \in R$ do
21:         if $ale$ then
22:             shorten($C, S$)
23:         end if
24:         if $ate$ then
25:             if $C \in \mathcal{F}$ or $\mathcal{E} \cap \mathcal{D} \neq \varnothing$ then
26:                 continue
27:             end if
28:             markAsIrredundant($C$)
29:             $\mathcal{E} \leftarrow \mathcal{E} \cup \{C\}$
30:             $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{D}$
31:         end if
32:     end for
33: end procedure
```

invoke a distillation procedure that determines if a clause is an asymmetric tautology and if it can be shortened by ALE. For asymmetric tautologies, the procedure returns a set $\mathcal{D}$ of dependencies. If ALE can be applied, a subset $S \subseteq C$ that later replaces C is returned. This information is stored in a list R. The data from R is used to replay the ATE and ALE operations at all workers.

In line 17 we start to apply those operations. First we sort the elimination information list R so that all workers consistently apply the same modifications to the CNF formula. If a clause can be shortened by ALE we unconditionally apply this modification. However, before applying ATE to a clause C, we need to ensure that no dependency of C is already eliminated and that C is not a dependency of an eliminated clause. Therefore we maintain a set $\mathcal{E}$ that stores all eliminated clauses and a set $\mathcal{F}$ of clauses that must not be eliminated.

Algorithm 5.2 lists the sequential distillation procedure itself. The algorithm operates on a clause C. Line 2 defines a filter function that we pass to propagateFilter. propagateFilter performs unit propagation while only considering clauses for which the filter function evaluates to `true`. This is required as non-irredundant clauses can be dropped from the formula at any time, thus eliminating ATs based on them is unsound. In line 5 we check for some trivial cases. In particular we check if a clause is the antecedent of a literal on decision level zero. Such clauses cannot be removed by ATE. As outlined earlier we do not consider clauses of length two for performance reasons.

The loop in line 15 assigns all literals of C to `true`. Each literal is assigned on a different decision level. If the loop encounters a literal c that

Algorithm 5.2 Distillation procedure

```
1:  procedure DISTILL($C$)
2:      procedure F($D$)
3:          return $C \neq D$
                and ISIRREDUNDANT($D$)
4:      end procedure
5:      if LENGTH($C$) $\leq 2$
                or ISANTECEDENT($C$) then
6:          return $(false, false, \varnothing, \varnothing)$
7:      else if $\exists_{c \in C}$ISTRUE($C$) then
8:          return $(true, false, \varnothing, \varnothing)$
9:      end if
10:     $ate \leftarrow false$
11:     $ale \leftarrow false$
12:     $\mathcal{D} \leftarrow \varnothing$
13:     $S \leftarrow \varnothing$
14:     $E \leftarrow \varnothing$
15:     for $c \in C$ do
16:         if ISTRUE($c$) then
17:             $E \leftarrow E \cup \{c\}$
18:         end if
19:         if ISASSIGNED($c$) then
20:             continue
21:         end if
22:         NEWDECISIONLEVEL
23:         ENQUEUE($\neg c$)
24:         PROPAGATEFILTER(F)
25:         if ATCONFLICT then
26:             break
27:         end if
28:     end for
29:     $S$ : Stack
30:     procedure TRACE
31:         while not $S$.EMPTY do
32:             $c \leftarrow S$.POP
33:             if DECLEVEL($c$) $== 0$ then
34:                 continue
35:             end if
36:             $\lambda \leftarrow$ ANTECEDENT($c$)
37:             if $\lambda =$ DECISIONANTECEDENT then
38:                 $S \leftarrow S \cup \{\neg c\}$
39:             else
40:                 $\mathcal{D} \leftarrow \mathcal{D} \cup \{$clause of $\lambda\}$
41:                 for $a \in$ REASONS($\lambda$) do
42:                     $S$.PUSH($a$)
43:                 end for
44:             end if
45:         end while
46:     end procedure
47:     if ATCONFLICT then
48:         for $a \in C$ do
49:             $S$.PUSH($a$)
50:         end for
51:         TRACE
52:         $ate \leftarrow true$
53:     else if $|E| > 0$ then
54:         $a \leftarrow \text{argmin}_{c \in E}$ DECLEVEL($c$)
55:         $\lambda \leftarrow$ ANTECEDENT($a$)
56:         $\mathcal{D} \leftarrow \mathcal{D} \cup \{$clause of$\lambda\}$
57:         for $a \in$ REASONS($\lambda$) do
58:             $S$.PUSH($a$)
59:         end for
60:         TRACE
61:         $S \leftarrow S \cup \{a\}$
62:         $ate \leftarrow true$
63:     else
64:         for $a \in C$ do
65:             $S$.PUSH($a$)
66:         end for
67:         TRACE
68:     end if
69:     if $|S| < |C|$ then
70:         $ale \leftarrow true$
71:     end if
72:     BACKJUMP(0)
73:     CLEARCONFLICT
74:     return $(ate, ale, \mathcal{D}, S)$
75: end procedure
```

is already assigned to true, this assignment necessarily happened at a non-zero decision level and C is not part of the antecedent graph of c. In this case, we have $\bar{c} \in \mathrm{ALA}(C)$ and C is an AT. We memorize c in a set E so that we can traverse the antecedent graph and determine all dependencies of this AT later. When there are multiple such literals we prioritize the literal with lowest decision level to get a small antecedent graph.

In line 47 we finally determine if ATE and/or ALE can be applied to C. In the first two cases, assigning all literals of C to false leads to a conflict that does not involve C itself. Here C is an AT. In all three cases we scan the antecedent graph to compute all dependendencies and a subclause of C that can be derived by ALE. This completes the distillation algorithm.

5.3.6 A note on variable elimination

It is conspicuous that we did not consider parallel variable elimination in this section. We shortly remark why parallel variable elimination is a hard problem.

A sufficient condition for CNF-parallelizability of BVE applied to two different variables x and y is that no clause that contains x shares a variable with any clause that contains y. This means that the variable incidence graph has a vertex separator[10] so that x and y are in different partitions w.r.t. this separator. Finding minimal separators is an NP-complete problem in itself [63]. Thus, it would be necessary to apply an approximation algorithm to find such separators. Because it is unclear, whether a good approximation can be obtained in acceptable time (especially because we need as many partitions as there are processors), we do not study parallel variable elimination here; the same applies for variable addition.

It should be noted that there is work on a parallel, shared-memory algorithm for variable elimination [36] via fine-grained locking. However, as this algorithm will not scale to distributed computers, we will not further discuss it here.

5.4 Experimental results

In this section, we evaluate the effectiveness and performance of our parallel and sequential CNF simplification techniques. For this evaluation we use the same execution environment as in chapter 4.

[10] A vertex separator of a graph G is a partition of the vertex set $V(G)$ into three sets A, B and S so that there is no edge between A and B.

5.4.1 Effectiveness of CNF simplifications

In a first experiment, we compare the effectiveness of different simplification techniques.

We will focus on parallelizable simplification techniques here. For the effectiveness of sequential simplification algorithms, we refer to our survey [94].

Here, we consider preprocessing only: We apply CNF simplification and CDCL search in separate phases. There are two kinds of data that we are interested in: First, we want to evaluate which CNF simplification techniques accelerate the CDCL solver's performance. Secondly, we need to take the runtime that CNF simplifications themselves consume into account. In order to obtain this information, we run the sequential satUZK solver on the 350 CNF instances from SAT Competition 2017. We set a timeout of one hour, both for the preprocessing and for the CDCL phase. In reality, we do not want to allocate such a large fraction of the available runtime to CNF simplification. Later experiments will try to reduce the runtime that CNF simplifcation requires by parallelizing the simplifcation techniques. We compare the following configurations:

Default As a reference, we use the sequence of CNF simplification techniques from the SAT Competition 2017 version of satUZK. This sequence consists of an initial unit propagation pass and then applies unhiding (i.e. resolution subsumption on binary clauses, failed literal elimination using binary clauses only, HLE and HTE), BCE, subsumption, resolution subsumption and BVE. This sequence is repeated until less than 5% of the remaining clauses and variables are affected by the last iteration of the sequence.

Long distillation Runs distillation on clauses with length greater than two only. The distillation algorithm only runs once, after the initial iteration of the preprocessing sequence completes.

Simple/iterative distillation First runs the tree-based lookahead algorithm to determine failed literals. After that, it runs the distillation algorithm on clauses with a length greater than two. The simple version only runs distillation during the initial iteration of preprocessing while the iterative version runs it in each iteration.

Simple/iterative HBR Runs the tree-based lookahead algorithm to perform hyper binary resolution and to determine failed literals. HBR is scheduled before unhiding in the preprocessing sequence. The difference between the simple and iterative version is the same as for distillation.

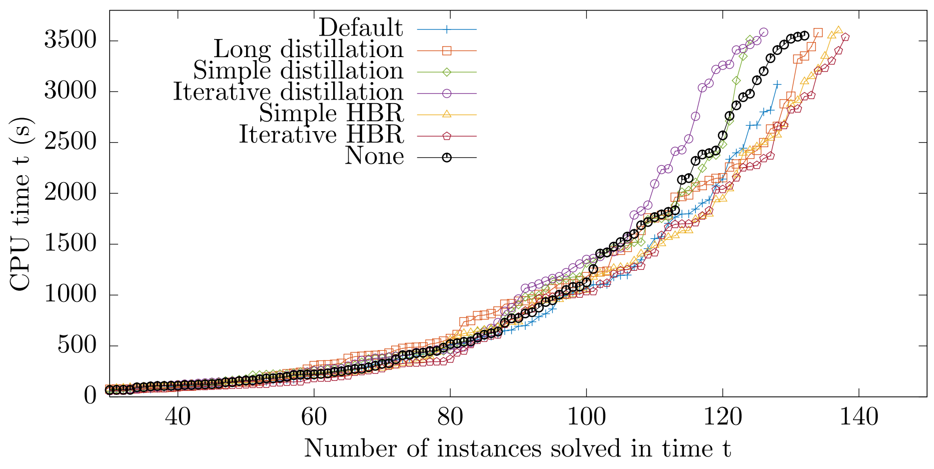


Figure 5.2: Effectiveness of CNF simplification, effect on CDCL runtime
Reports only the runtime of the CDCL algorithm not the simplification time.

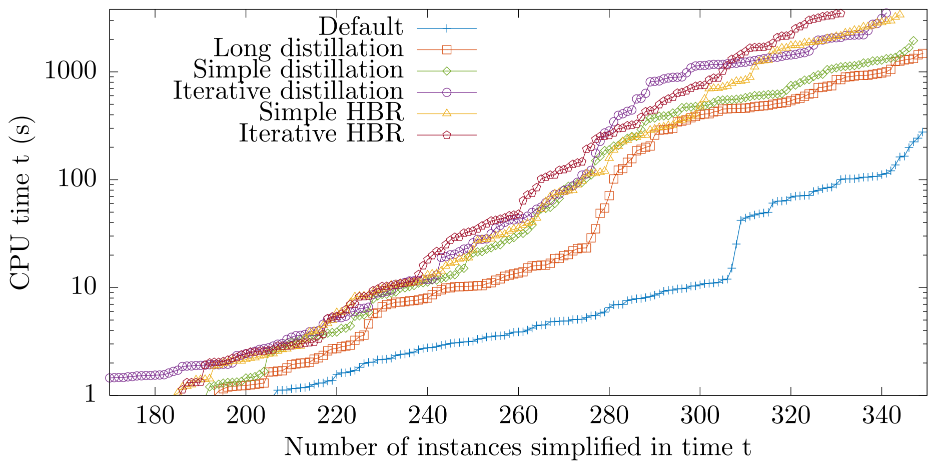


Figure 5.3: Effectiveness of CNF simplification, preprocessing runtime
Reports the simplification time in a logarithmic plot. Note that the first 170 instances are not plotted as they are too easy for all simplification techniques.

Table 5.2: Effectiveness of simplification

States the number of instances that could be simplified or solved within the timeout. Instances that could not be simplified are reported as unsolved. Maximal simplification time is given in seconds.

Configuration	Simplified	Solved	Max simp. time
None	-	133	-
Default	350	129	276.72
Long dist.	350	135	1481.17
Simple dist.	348	125	> 3600.00
Iterative dist.	342	127	> 3600.00
Simple HBR.	345	138	> 3600.00
Iterative HBR	332	139	> 3600.00

Let us discuss the results of the experiment. Table 5.2 contains the number of instances that could be simplified and solved within the timeout. In order to better understand the results, we visualize them in cactus plots. Figure 5.2 depicts the runtime of the CDCL algorithm after preprocessing has been applied. The first interesting observation is that the configuration without preprocessing solves more instances than the default preprocessing configuration. This is unexpected, especially because it does not match the results of satUZK at SAT Competition 2017, where a larger timeout of 5000 seconds is used. The second observation is that both HBR and distillation of clauses with length greater than two improve considerably over the configuration without preprocessing. The effectiveness of HBR is even more impressive given the fact that many instances could not be preprocessed within the timeout and are thus excluded from the results. However, applying HBR multiple times does not seem to improve the runtime of the CDCL algorithm substantially.

On the other hand, distillation in combination with failed literal elimination seems to have a negative impact on the number of solved instances. We conjecture that in these configurations, too many clauses are weakened and later removed as the result of asymmetric tautology elimination. While those asymmetric tautologies are of course redundant, they still guide the CDCL solver and enable it to find a solution (or unsatisfiability proof) more quickly. This suggests that further heuristics are required to restrict the set of asymmetric tautologies that should be weakened from the input formula.

Figure 5.3 presents the runtime of the CNF simplifcation techniques itself. Note that the plot has a logarithmic time axis to account for the large range of occurring runtimes. The default preprocessing configuration is by far the cheapest configuration, with a maximal runtime of under 200 seconds. Even the simple HBR and distillation configurations reach a runtimes over 3600 seconds. Only the default and long distillation configurations were able to preprocess all benchmark formulas within the timeout.

On large ranges of instances, simple distillation is not significantly slower than distillation of long clauses only. The iterative versions of both distillation and HBR are much slower than the simple versions. Together with the fact that the iterative versions do not improve considerably upon the runtime of the simple versions, this suggests that running iterative distillation and HBR as preprocessing is not worthwhile.

5.4.2 Simplification scalability

In the second experiment regarding simplification, we evaluate the scalability of our parallel simplification techniques.

All simplifcation techniques were ran with 20, 40, 80 and 160 workers on the SAT Competition 2017 benchmarks. We evaluate the following combinations of simplification techniques:

Subsumption + BCE Runs the parallel subsumption, self-subsumption and BCE algorithms.

(FLE +) Distillation First runs the parallel tree-based lookahead algorithm to determine failed literals and then runs the parallel distillation algorithm on all clauses with length greater than two.

HBR Runs the parallel tree-based lookahead algorithm two times. The first run determines failed literals, while the second run performs hyper binary resolution.

Table 5.3: Parallel subsumption + BCE: Speedup

	All instances (Strong scaling)		Difficult instances (Weak scaling)		
Workers	Total	Median	Samples	Total	Median
20	16.37	12.90	31	16.38	18.89
40	19.97	23.71	32	20.02	35.76
80	38.88	40.06	31	39.25	57.66
160	73.74	47.58	29	75.52	110.12

Let us first analyze the combination of subsumption and BCE. As the sequential computation of these techniques takes less than ten seconds on all SAT Competition 2017 instances, we cannot use these instances to benchmark our parallel implemenations. Instead, we take the `barman`, `sokoban` and `stone` families of instances from SAT Competition 2016 as benchmarks. This results in a set of 60 instances. The sequential implementation requires hundreds to thousands of seconds to simplify these instances. Table 5.3 reports the speedup of the subsumption + BCE algorithm. As we expect, the algorithm has almost ideal weak scaling properties and decent strong scaling.

Secondly, we discuss distillation. In contrast to the preceeding evaluation, we use the SAT Competition 2017 benchmarks here. Figure 5.5 visualizes the results of the experiment. Only the 80 most difficult instances are reported in the plot as many of the instances are actually too small when 20 or more workers are used. In contrast to the sequential case, all parallel configurations are able to simplify all 350 benchmark instances.

We compute the speedup of the distillation algorithm on this set of benchmarks. Table 5.4 reports the strong and weak scalability of parallel distillation. We use the same difficulty measure as in subsection 4.4.1 of the previous chapter. As many instances are too simple for the parallel algorithm, the median speedup over all instances is less than 1; the program runtime is increased by the additional parallelism. However, when difficult instances are considered, we get satisfactory speedup values over all configurations.

Table 5.4: Parallel distillation: Speedup

	All instances		Difficult instances		
Workers	Total	Median	Samples	Total	Median
20	15.12	< 1	74	15.52	13.57
40	25.03	< 1	69	26.05	22.79
80	37.13	< 1	65	39.82	33.27
160	50.22	< 1	60	55.64	48.71

Finally, we evaluate HBR. Figure 5.4 depicts the results of this technique. All parallel configurations are much faster than their sequential counterpart. However, beyond 20 workers, added parallelism does not improve the algorithm's performance. The reason for this behavior is that HBR generates such a large amount of clauses that importing clauses from other workers takes more time than running the actual HBR algorithm. Furthermore, for large instances, the parallel HBR algorithm exhausts the available amount of memory while the resulting clauses are exchanged. We suggest that additional heuristics could be developed to restrict the number of generated clauses but leave this challenge to future work.

5.4.3 Effectiveness of satUZK-ddc plus simplification

In our final experiment, we benchmark the performance of satUZK-ddc with preprocessing enabled.

For the experiment, we consider our default sequential preprocessing (as described in subsection 5.4.1), the combination of distillation and failed

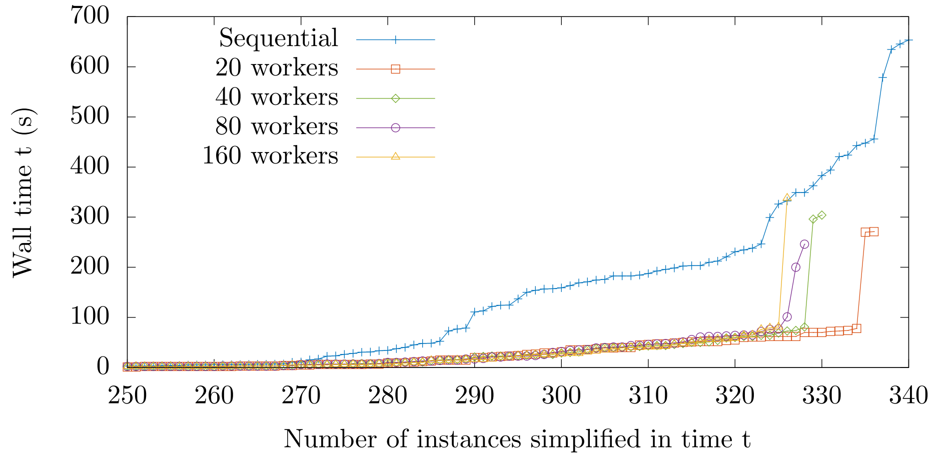


Figure 5.4: Parallel HBR runtime

The first 250 instances are not reported as they are simplified in a negligible amount of time.

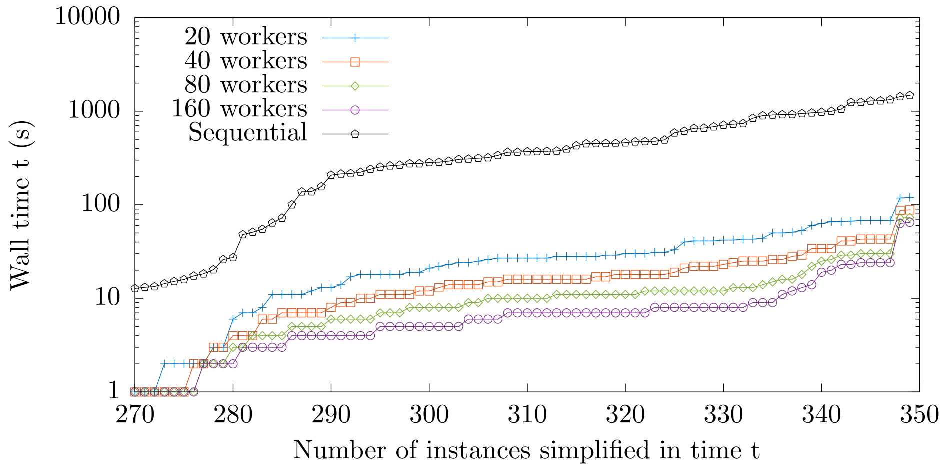


Figure 5.5: Parallel distillation runtime

Again, the first 270 instances are not reported.

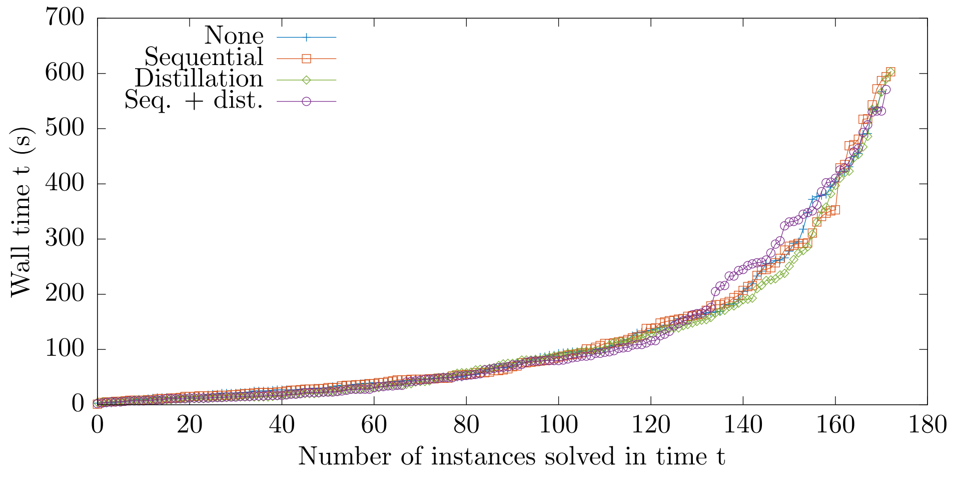


Figure 5.6: Effectiveness of satUZK-ddc + simplification

literal elimination from the last subsection and a combination of these techniques. We use the SAT Competition 2017 instances and set a timeout of 10 minutes.

Figure 5.6 depicts the results of the evaluation. All configurations are equally viable and solve between 171 and 173 instances. This is in contrast to sequential configurations, where satUZK-seq solves a significantly smaller number of instances if distillation is enabled. Hence, the high computational cost of distillation is offset by its good scaling to many processors.

Table 5.5: satUZK-ddc + simplification: Comparison of configurations

Reports the numbers of instances that could only be solved in some configurations.

		Instances solved by			
		None	Sequential	Distillation	Seq. + dist.
versus	None	-	8	7	9
	Sequential	6	-	7	9
	Distillation	5	7	-	8
	Seq. + dist.	8	10	9	-

It should be noted that the configurations do not solve the same instances. Table 5.5 lists the number of instances that were solved only in some configurations. It is evident from this table that our parallel distillation algorithm enables satUZK-ddc to solve multiple instances that could not be solved otherwise.

Chapter 6

Conclusion and future work

In this dissertation, we presented our CDCL implementation, introduced the novel DDC algorithm for distributed SAT solving and developed parallelizations of several established CNF simplification techniques.

In a benchmark, our DDC algorithm turned out to be faster than other state-of-the-art, distributed SAT solvers while solving at least as many instances. We studied performance characteristics of the algorithm and introduced extensions that boost its effectiveness. While the solver is not necessarily able to solve more instances when parallel simplification is enabled, we demonstrated that several instances could only be solved when the parallel distillation technique was used.

Future work could either focus on improvements to the sequential CDCL algorithm or on improvements to the parallel DDC architecture. For the parallel architecture, there are still some successful technologies that have not been integrated into satUZK-ddc. For example, the clause exchange scheme from the Syrup solver seems to be an obvious candidate for integration to the DDC framework.

Orthogonally, the communication between individual workers could be further restricted in order to allow the algorithm to better scale to thousands and tens of thousands of cores. The DDC algorithm already performs routing decisions and assignment of work locally relative to the DPLL search tree that it constructs. However, no effort is made to match this search tree to the underlying topology of the interconnect. Another interesting challenge could be the study of clause exchange schemes that prefer to exchange clauses between local clusters of workers instead of broadcasting them to all workers. Again, these considerations should take the interconnect topology into account in order to be successful.

Bibliography

[1] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 448–456, New York, NY, USA, 2002. ACM.

[2] Austin Appleby. smasher. Available from GitHub: `https://github.com/aappleby/smhasher`.

[3] G. Audemard, B. Hoessen, S. Jabbour, and Piette C. Dolius: A distributed parallel SAT solving framework. In *Pragmatics of SAT*, POS '14, 2014.

[4] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2012*, pages 200–213. Springer, 2012.

[5] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Theory and Applications of Satisfiability Testing - SAT 2011*, SAT '11, pages 188–200. Springer, 2011.

[6] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013*, pages 309–317. Springer, 2013.

[7] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An Adaptive Parallel SAT solver. In *Principles and Practice of Constraint Programming*, CP '16, pages 30–48. Springer, 2016.

[8] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint*

Conference on Artifical Intelligence, IJCAI'09, pages 399–404. Morgan Kaufmann Publishers Inc., 2009.

[9] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In Michela Milano, editor, *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 118–126, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 197–205, Cham, 2014. Springer International Publishing.

[11] Tomáš Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing - SAT 2015*, pages 156–172. Springer, 2015.

[12] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP Look-back Techniques to Solve Real-world SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997.

[13] Paul Beanie, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1194–1201, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[14] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability.* IOS Press, 2009.

[15] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, 2010.

[16] Armin Biere. Lingeling and Friends at the SAT Competition 2011. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[17] Armin Biere. Lingeling and Friends Entering the SAT Challenge 2012. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, pages 33–34, 2012.

[18] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proceedings of SAT Competition 2013*, pages 51–52, 2013.

[19] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the Sat Competition 2016. In *Proceedings of SAT Competition 2016*, pages 44–45, 2016.

[20] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2017*, pages 14–15, 2017.

[21] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[22] Armin Biere and Andreas Fröhlich. Evaluating cdcl restart schemes. In *Pragmatics of SAT*, POS '15, 2015.

[23] Armin Biere and Andreas Fröhlich. Evaluating cdcl variable scoring schemes. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pages 405–422, Cham, 2015. Springer International Publishing.

[24] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, pages 381–400, 1996.

[25] boost.org. Boost C++ libraries. `http://www.boost.org/`.

[26] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[27] Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *Proceedings of the Sixth*

Annual ACM Symposium on Theory of Computing, STOC '74, pages 135–148, New York, NY, USA, 1974. ACM.

[28] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC'71, pages 151–158. ACM, 1971.

[29] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 03 1979.

[30] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, 1989.

[31] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[33] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing SAT 2015*, pages 61–75. Springer, 2005.

[34] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, SAT '03, pages 502–518. Springer, 2004.

[35] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543 – 560, 2003. BMC'2003, First International Workshop on Bounded Model Checking.

[36] Kilian Gebhardt and Norbert Manthey. Parallel Variable Elimination on CNF Formulas. In Ingo J. Timm and Matthias Thimm, editors, *KI 2013: Advances in Artificial Intelligence: 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings*, pages 61–73, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[37] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[38] Shai Haim and Marijn Heule. Towards ultra rapid restarts. Technical report, 2014.

[39] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(Supplement C):297 – 308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[40] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2009.

[41] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34(2):99–106, 2013.

[42] Hyojung Han and Fabio Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *44th ACM/IEEE Design Automation Conference*, pages 582–587. ACM, 2007.

[43] Federico Heras, Antonio Morgado, and Joao Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, pages 36–41. AAAI Press, 2011.

[44] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In *Theory and Applications of Satisfiability Testing*, SAT '04, pages 345–359. Springer, 2004.

[45] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 357–371. Springer, 2010.

[46] Marijn Heule, Matti Järvisalo, and Armin Biere. Revisiting hyper binary resolution. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, pages 77–93, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[47] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 201–215. Springer, 2011.

[48] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In Leonardo de Moura, editor, *Automated Deduction – CADE 26: 26th International Conference on Automated De-*

duction, Gothenburg, Sweden, August 6–11, 2017, Proceedings, pages 130–147, Cham, 2017. Springer International Publishing.

[49] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. PRuning Through Satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, pages 179–194, Cham, 2017. Springer International Publishing.

[50] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 228–245, Cham, 2016. Springer International Publishing.

[51] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT solvers by lookaheads. In *Proceedings of the 7th international Haifa Verification conference on Hardware and Software: Verification and Testing*, HVC '12, pages 50–65. Springer, 2012.

[52] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving sat in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*, pages 430–435, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[53] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '10, pages 372–386. Springer, 2010.

[54] Antti E. J. Hyvärinen and Christoph M. Wintersteiger. Approaches for multi-core propagation in clause learning satisfiability solvers. Technical report, 2012.

[55] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256 – 274, 2008. International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004).

[56] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

[57] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '10, pages 129–144. Springer, 2010.

[58] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 355–370. Springer Berlin Heidelberg, 2012.

[59] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of sat solvers. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 481–488. AAAI Press, 2013.

[60] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley Professional, 2015.

[61] Boris Konev and Alexei Lisitsa. A sat attack on the erdős discrepancy conjecture. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 219–226, Cham, 2014. Springer International Publishing.

[62] O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(Supplement C):149 – 176, 1999.

[63] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Vieweg+Teubner Verlag, Wiesbaden, 1992.

[64] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 123–140, Cham, 2016. Springer International Publishing.

[65] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173 – 180, 1993.

[66] Inês Lynce and João Marques-Silva. Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 43(1):137–152, 2005.

[67] Norbert Manthey. Parallel SAT Solving - Using More Cores. In *Pragmatics of SAT*, 2011.

[68] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, pages 102–117, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[69] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th ACM/IEEE Design Automation Conference*, DAC '01, pages 530–535. ACM, 2001.

[70] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 294–299, Berlin, Heidelberg, 2007. Springer-Verlag.

[71] Knot Pipatsrisawat and Adnan Darwiche. On the Power of Clause-Learning SAT Solvers with Restarts. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings*, pages 654–668, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[72] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293 – 304, 1986.

[73] Antonio Ramos, Peter van der Tak, and Marijn J. H. Heule. Between restarts and backjumps. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, pages 216–229, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[74] Jussi Rintanen. Planning and SAT. In *Handbook of Satisfiability.* IOS Press, 2009.

[75] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[76] Olivier Roussel. Description of ppfolio. Technical report, 2011. `http://www.cril.univ-artois.fr/~roussel/ppfolio/`.

[77] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2002. Master thesis.

[78] satlive.org. The international SAT Competitions web page. `http://www.satcompetition.org/`.

[79] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided Design*, ICCAD '96, pages 220–227. IEEE, 1996.

[80] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 244–257, Berlin, Heidelberg, 2009. Springer-Verlag.

[81] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 237–243, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[82] E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In *Proceedings of the 1st International Conference on Supercomputing*, pages 985–993, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[83] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for preprocessing sat instances. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pages 276–291, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[84] Niklas Sörensson and Niklas Een. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization, 2005. Poster at SAT'05.

[85] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 1970.

[86] Alasdair Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[87] Alexander van der Grinten. satUZK-seq and satUZK-ddc: Solver description. In Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2017*, 2017.

[88] Alexander van der Grinten and Ewald Speckenmeyer. SAT solving on clusters and supercomputers based on real-time communication and efficient load balancing, 2017. Submitted.

[89] Alexander van der Grinten, Andreas Wotzlaw, and Ewald Speckenmeyer. satUZK: Solver description. In Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2014*, page 75, 2014.

[90] Alexander van der Grinten, Andreas Wotzlaw, Ewald Speckenmeyer, and Stefan Porschen. satUZK: Solver description. In Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013*, page 82, 2013.

[91] Peter van der Tak, Marijn J. H. Heule, and Armin Biere. Concurrent Cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2012*, SAT '12, pages 475–476. Springer, 2012.

[92] Peter van der Tak, Antonio Ramos, and Marijn Heule. Reusing the Assignment Trail in CDCL Solvers. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, pages 133–138, 2011.

[93] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 422–429, Cham, 2014. Springer International Publishing.

[94] Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer. Effectiveness of pre- and inprocessing for CDCL-based SAT solving. *ArXiv*, 2013.

[95] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, and Stefan Porschen. ppfolioUZK: Solver description. In Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors, *Proceedings of SAT Challenge 2012*, page 45, 2012.

[96] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, and Stefan Porschen. satUZK: Solver description. In Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors, *Proceedings of SAT Challenge 2012*, pages 54–55, 2012.

[97] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:65–606, 2008.

[98] Hantao Zhang and Mark Stickel. Implementing the Davis-Putnam Method. *Journal of Automated Reasoning*, 24:277–296, Feb 2000.

[99] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided Design*, ICCAD '01, pages 279–285. IEEE, 2001.

Danksagungen

Ich bedanke mich zunächst bei Prof. Dr. Speckenmeyer, der diese Arbeit betreut hat und mir mit vielen nützlichen Ratschlägen und Denkanstößen zur Seite stand. Desweiteren danke ich Andreas Wotzlaw für zahllose Diskussionen, ohne die satUZK wahrscheinlich nicht entstanden wäre.

Mein weiterer Dank gilt Prof. Dr. Meyerhenke, der diese Arbeit als zweiter Gutachter erhält. Dem Institut für Informatik der Universität zu Köln gilt mein Dank unter anderem für die enormen Mengen an Rechenzeit, die in die Entwicklung, Optimierung und Evaluation von satUZK geflossen sind. Meinen Kollegen danke ich für die angenehme Zeit am Institut.

Nicht zuletzt gilt mein Dank meiner Familie für ihre Unterstützung. Insbesondere gilt das für Viktor van der Grinten, der sehr viel Zeit damit verbracht hat, diese Arbeit auf Fehler zu überprüfen.

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie - abgesehen von unten angegebenen Teilpublikationen - noch nicht veröffentlicht worden ist, sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Speckenmeyer betreut worden.

(Alexander van der Grinten)

Bei den oben genannten Teilpublikationen handelt es sich um:

A. Wotzlaw, A. van der Grinten, E. Speckenmeyer, S. Porschen: satUZK: Solver Description in Proceedings of SAT Challenge 2012, 2012 [96]

A. Wotzlaw, A. van der Grinten, E. Speckenmeyer, S. Porschen: ppfolioUZK: Solver Description in Proceedings of SAT Challenge 2012, 2012 [95]

A. van der Grinten, A. Wotzlaw, E. Speckenmeyer, S. Porschen: satUZK: Solver Description in Proceedings of SAT Competition 2013, 2013 [90]

A. van der Grinten, A. Wotzlaw, E. Speckenmeyer: satUZK: Solver Description in Proceedings of SAT Competition 2014, 2014 [89]

A. Wotzlaw, A. van der Grinten, E. Speckenmeyer: Effectiveness of pre- and inprocessing for CDCL-based SAT solving on ArXiv, 2013 [94]

A. van der Grinten: satUZK-seq and satUZK-ddc: Solver description in Proceedings of SAT Competition 2017, 2017 [87]

A. van der Grinten, E. Speckenmeyer: SAT solving on clusters and supercomputers based on real-time communication and efficient load balancing, eingereicht, 2017 [88]