Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3200

Skalierbare parallele Verfahren zur Boolean Constraint Propagation auf Multicore-Architekturen

Jan Stöcklin

Studiengang:	Informatik
Prüfer:	Prof. Dr. Kurt Rothermel
Betreuer:	PD. Dr. Wolfgang Blochinger
begonnen am:	1. Juni 2011
beendet am:	1. Dezember 2011
CR-Klassifikation:	D.1.3, D.2.8

Für Rolf und Gina, die mir das Studium ermöglicht haben

Zusammenfassung

Beim SAT-Solving werden boolesche Formeln auf Erfüllbarkeit überprüft. Seit den frühen 1990ern gewann das SAT-Solving für Industrie und Forschung stetig an Bedeutung, da sich viele Probleme der Hard- und Softwareentwicklung auf das Erfüllbarkeitsproblem reduzieren lassen. Ein Großteil der Laufzeit eines SAT-Solvers (ca. 80 % - 90 %) entfällt auf die Unit-Propagation. Unit-Propagation ist die iterative Anwendung von Implikationen, die aufgrund der Formel und der aktuellen Belegung gelten. Um Beschleunigungen des Solving-Prozesses zu erzielen, werden in dieser Arbeit verschiedene Parallelisierungsansätze der Unit-Propagation untersucht. Da moderne Prozessoren mit immer mehr Rechenkernen ausgestattet sind, werden Parallelisierungsansätze entwickelt, die auf Mehrkernarchitekturen basieren.

In der vorliegenden Arbeit werden zunächst die von aktuellen SAT-Solvern verwendeten Algorithmen und Implementierungstechniken vorgestellt. Insbesondere wird der DPLL-Algorithmus und dessen Erweiterung um Klauselllernen (CDCL-Solver) präsentiert. Danach wird mit dem Two-Watched-Literal Schema ein effizientes Verfahren zur Unit-Propagation beschrieben. Im nächsten Schritt wird untersucht, welche Teile der Unit-Propagation sich besonders für eine Parallelisierung eignen und welche Probleme dabei zu lösen sind. Darauf aufbauend wird ein sperrbasierter und ein sperrfreier Lösungsansatz sowie parametrisierbare Alternativen davon entwickelt. Die Algorithmen werden anschließend in Minisat – einem minimalistischen klauselllernenden SAT-Solver – implementiert. Nach Auswahl eines geeigneten Formelsatzes werden die Algorithmen evaluiert. Dazu wird der Formelsatz für jeden Algorithmus mehrfach ausgeführt und Parallelisierungskennzahlen (Beschleunigung, parallele Effizienz) ermittelt. Abschließend werden die Ergebnisse beider Ansätze miteinander verglichen und bewertet.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

1.1. Motivation

Das Erfüllbarkeitsproblem der Aussagenlogik – auch als SAT-Problem[1] bezeichnet – untersucht die Frage, ob es zu einer gegebenen aussagenlogischen Formel eine erfüllende Belegung gibt. Ein SAT-Solver ist ein Programm, das das Erfüllbarkeitsproblem einer Formel (auch SAT-Instanz) entscheidet. Das SAT-Problem ist sowohl in der theoretischen als auch in der angewandten Informatik von großer Bedeutung.

Bedeutung des SAT-Problems für die theoretische Informatik

Anfang der 1970er wurde das P $\stackrel{?}{=}$ NP-Problem durch Cook und Levin formuliert [11]. Das Problem stammt aus der Komplexitätstheorie und stellt die Frage, ob die Komplexitätsklassen P und NP die gleichen Sprachen enthalten. Die Inklusion P $\subseteq$ NP folgt direkt aus der Definition der Turingmaschinen. Ob auch die Umkehrung, also NP $\subseteq$ P gilt, ist jedoch noch ungeklärt. Die Klärung der zweiten Inklusion und damit die Lösung des P $\stackrel{?}{=}$ NP-Problems hat für viele Anwendungsgebiete der Informatik (z. B. industrielle Optimierungsprobleme, Kryptografie und künstliche Intelligenz) weitreichende Konsequenzen [11]. Im Jahr 2000 wurde das P $\stackrel{?}{=}$ NP-Problem durch das Clay Mathematics Institute zu einem von sieben Millennium-Problemen erklärt, deren Lösung mit jeweils einer Millionen US-Dollar belohnt wird. Trotz seiner Bedeutung und des finanziellen Anreizes ist das Problem nach Kenntnis des Autors bis heute ungelöst.

Innerhalb der Klasse NP gibt es *NP-vollständige* Sprachen. Eine Sprache ist NP-vollständig, wenn sie aus NP ist und sich eine beliebige Sprache der Klasse NP in polynomieller Zeit auf L reduzieren lässt. Ihre besondere Bedeutung erlangen NP-schwere Sprachen dadurch, dass diese entscheidend zur Lösung der P $\stackrel{?}{=}$ NP-Frage beitragen können. Zur Beantwortung der

[1] abgeleitet vom englischen satisfiability

P $\stackrel{?}{=}$ NP-Frage genügt es lediglich für eine NP-schwere Sprache zu zeigen, dass diese in P ist oder nicht. Gelingt der Nachweis, dass eine NP-vollständige Sprache in P ist, so folgt daraus NP $\subseteq$ P und damit P = NP. Kann man andernfalls zeigen, dass eine NP-vollständige Sprache nicht in P liegen kann, so gilt P $\subsetneq$ NP und damit P $\neq$ NP.

1971 bewies Steven Cook in [12] die NP-Vollständigkeit des SAT-Problems. Die Klärung der P $\stackrel{?}{=}$ NP-Frage kann daher auf die Frage reduziert werden, ob sich das SAT-Problem in Polynomialzeit lösen lässt. Das seit Jahrzehnten ungelöste P $\stackrel{?}{=}$ NP-Problem zeigt jedoch, wie schwer es ist einen solchen Algorithmus anzugeben oder seine Nicht-Existenz zu beweisen. In einer Umfrage aus dem Jahr 2002 [17] gehen 61 von 100 befragten Experten davon aus, dass P $\neq$ NP gilt, es demnach also gar keinen Polynomialzeit-Algorithmus für das SAT-Problem gibt.

Bedeutung des SAT-Problems für die angewandte Informatik

Bis etwa Ende der 1980er war das SAT-Problem überwiegend von theoretischer Bedeutung [29]. Anfang der 1990er gewann das SAT-Solving[2] zunehmend auch in der Praxis an Relevanz. Verbesserte Algorithmen und Implementierungstechniken (siehe Kapitel 2) sowie die gestiegene Rechenleistung führten zu einer Beschleunigung der SAT-Solver um mehrere Größenordnungen. Dadurch konnten selbst Instanzen mit mehreren hunderttausend Variablen und Klauseln in akzeptabler Zeit bearbeitet werden. Der naive Ansatz, der alle Belegungen durchprobiert, kann dagegen schon bei verhältnismäßig kleinen Eingaben (zwei- bis dreistellige Variablenzahl) extreme Laufzeiten haben. Industrie und Forschung entdeckten in der Folge viele Probleme, die sich effizient in SAT-Instanzen übersetzen lassen. Im Folgenden werden einige Anwendungsgebiete vorgestellt.

Im Bereich der Hardwareverifikation lässt sich die Fragestellung, ob zwei kombinatorische Schaltkreise dieselbe Funktion berechnen auf SAT reduzieren. Damit kann man nachweisen, dass ein neuer, gegebenenfalls ressourcensparender Schaltkreis, die gleiche Funktion berechnet wie ein bereits vorhandener. Im Bereich des Hardwaretests werden im Rahmen des Automatic Test Pattern Generation (ATPG) Testfälle für spezifische Hardwaredefekte (z. B. stuck-at-0 oder stuck-at-1) generiert. Als Eingabe dient der Schaltkreis sowie der Fehler, dessen Abhandensein getestet werden soll. Der SAT-Solver berechnet daraus diejenige Eingabe, die bei einem

[2] (programmgestützte) Entscheidung ob eine aussagenlogische Formel erfüllbar oder unerfüllbar ist

fehlerhaften Schaltkreis zu einer vom korrekten Schaltkreis verschiedenen Ausgabe führt und somit detektiert werden kann [28]. Da Fehler in der Hardware zum Austausch selbiger und damit zu enormen Kosten und Imageschäden führen können (z. B. Intel FDIV-Bug), ist die Funktionsabsicherung von Hardware von großer wirtschaftlicher Bedeutung.

Im Bereich der Softwareverifikation wird das SAT-Solving im Rahmen des Bounded Model Checking (BMC) verwendet. Dabei werden im Gegensatz zum Model Checking nur Programmläufe bis zu einer vorgegebenen „Tiefe“ daraufhin untersucht, ob Korrektheitsbedingungen (z. B. keine Null-Pointer-Dereferenzierungen) verletzt werden. Als Eingabe dienen das als SAT-Instanz codierte Programm sowie die Korrektheitsbedingungen. Der SAT-Solver findet genau dann eine erfüllende Belegung, wenn das Programm einen Ablauf (innerhalb der vorgegebenen Tiefe) besitzt, der eine Korrektheitsbedingung verletzt. Zur Fehleranalyse kann der Programmlauf durch Interpretation der erfüllenden Belegung rekonstruiert werden [28]. Das BMC kommt z. B. beim Testen von Betriebssystem-Kerneltreibern zum Einsatz [31]. Dadurch können Fehler frühzeitig gefunden und die Systemstabilität verbessert werden.

Als abschließendes Beispiel für die vielfältigen Einsatzmöglichkeiten eines SAT-Solvers seien noch Paketmanager moderner Linux-Betriebssysteme genannt [26] [44]. Diese kodieren Paketabhängigkeiten und -unverträglichkeiten als SAT-Problem und überprüfen jedes Mal, wenn sich der Umfang der installierten Pakete ändert, ob die angeforderte Aktion (z. B. Installation eines Softwarepakets) zulässig ist. Da dies eine häufig verwendete Aktion ist, ist es wünschenswert, dass die Gültigkeitsprüfung so wenig Zeit wie möglich beansprucht. Eine ausführlichere Darstellung der angesprochenen Anwendungsgebiete sowie eine Reihe weiterer Anwendungen, die sich auf das SAT-Solving stützen, findet sich in [4].

Die Vorgehensweise, ein Problem als SAT-Instanz zu formulieren, kann als eine Form der deklarativen Programmierung angesehen werden. Bei dieser Art der Programmierung wird nur das Problem und seine Regeln beschrieben, nicht aber die Art und Weise wie die Lösung ermittelt wird. Bei SAT-Solvern, die auf dem DPLL-Algorithmus basieren, erfolgt die Lösungsfindung implizit durch systematische Traversierung des Lösungsraumes. Die Lösung für das Ursprungsproblem erhält man durch Interpretation der erfüllenden Belegung – sofern diese existiert. Der Vorteil bei der Verwendung eines SAT-Solvers gegenüber eines speziell für ein Problem entwickelten Algorithmus besteht darin, dass man keinen Lösungsalgorithmus für das Problem entwickeln muss. Stattdessen kann man sich darauf beschränken, nur die Transformation in eine SAT-Instanz anzugeben. Dies ist in vielen Fällen deutlich einfacher. Darüber hinaus werden

Anstrengungen für Verbesserungen an einer zentralen Stelle (dem SAT-Solver) gebündelt. Von einer Verbesserung der SAT-Solver profitieren alle Anwendungen.

Entwicklung der Prozessortechnik

Neben dem Bedeutungsgewinn der SAT-Solver ist die Entwicklung der Prozessortechnik eine weitere Rahmenbedingung dieser Arbeit. Etwa bis zum Jahre 2005 waren Prozessoren der jeweils neueren Generation stets höher getaktet als die der vorhergehenden. Dadurch liefen die Programme – auch ohne jede Änderung an ihnen – automatisch schneller ab. Aufgrund von Abwärme- und Energieeffizienzproblemen gingen die Chiphersteller dazu über, statt der Taktfrequenz die Anzahl der Rechenkerne eines Prozessors zu erhöhen [33]. Anders als bei der Steigerung der Taktfrequenz profitieren Anwendungen nicht mehr automatisch von der potentiellen Leistung mehrerer Rechenkerne, sondern müssen dafür erst angepasst werden. Die Anpassung eines sequentiellen Programms erfolgt durch Aufteilung in mehrere unabhängige Teile (Dekomposition), die parallel von mehreren Rechenkernen ausgeführt werden können. Diese Entwicklung wurde 2005 von Herb Sutter mit dem Slogan „The Free Lunch Is Over“ charakterisiert [40]. In [41] vertreten Sutter und Larus die Meinung, dass die Anpassung keine einfache Aufgabe ist und noch ein Mangel an Tools und Sprachen besteht. Sie bezeichnen Semaphoren als eine Art Assembler für Parallelität und fordern eine darüber hinausgehende Abstraktionsschicht, die es erleichtert parallele Programme zu entwickeln („OO for concurrency“). Diese Rolle könnte in Zukunft durch den transaktionalen Speicher eingenommen werden. Dieser überträgt transaktionale Konzepte aus dem Datenbankumfeld auf die Ebene der Hauptspeicherzugriffe. Da die Technik eine einfache Schnittstelle zu den Programmiersprachen anbietet, soll die parallele Programmierung vereinfacht und weniger fehleranfällig gemacht werden. Da der transaktionale Speicher aktiver Gegenstand der Forschung ist, wird die Arbeit noch ohne diese Technik umgesetzt.

1.2. Zielsetzung

Diese Arbeit knüpft an die skizzierten Entwicklungen der SAT-Solver und der Prozessoren an. Es soll geklärt werden, ob sich moderne Mehrkernarchitekturen nutzen lassen um den SAT-Solving Prozess zu beschleunigen. Dazu sollen anhand von Minisat, einem minimalistischen klauellernenden SAT-Solver [16], verschiedene Parallelisierungsansätze entwickelt und

überprüft werden. Der Fokus liegt hierbei auf der Unit Propagation, die einen Großteil der Laufzeit eines SAT-Solver ausmacht. Zur Bewertung der Ergebnisse sollen Laufzeitmessungen durchgeführt und mit klassische Metriken des parallelen Rechnens bewertet werden (z. B. die Beschleunigung und die parallele Effizienz).

1.3. Gliederung der Arbeit

Nach der Einführung in das Thema, werden in Kapitel 2 die Grundlagen und Implementierungstechniken moderner SAT-Solver sowie der Zusammenhang zu anderen Arbeiten des Gebiets vorgestellt. Kapitel 3 stellt die Implementierung der Unit-Propagation in Minisat vor und analysiert, welche Teile sich besonders für eine Parallelisierung eignen. Kapitel 4 zeigt zunächst, welche Probleme bei der Parallelisierung überwunden werden müssen und stellt anschließend einen sperrbasierten und einen sperrfreien Ansatz sowie Varianten davon vor. In Kapitel 5 werden die parallelen Algorithmen mit der sequentiellen Variante verglichen und deren Beschleunigung sowie Effizienz bestimmt und bewertet. In Kapitel 6 werden die Ergebnisse zusammengefasst und bewertet.

2. Stand der Forschung

Ein SAT-Solver ist ein Programm, das zu einer gegebenen aussagenlogischen Formel entscheidet, ob diese eine erfüllende Belegung besitzt oder unerfüllbar ist. Grundsätzlich kann man zwischen vollständigen und unvollständigen SAT-Solving Verfahren unterscheiden. Vollständige Verfahren garantieren, dass sie nach endlicher Zeit entweder die Unerfüllbarkeit feststellen oder eine erfüllende Belegung finden. Im Gegensatz dazu können unvollständige Verfahren auch unendlich lange laufen, also zu keinem Ergebnis kommen. In der Praxis versuchen unvollständige Verfahren die Erfüllbarkeit einer Formel, durch das Finden einer erfüllenden Belegung zu zeigen. Sie sind aber nicht in der Lage die Unerfüllbarkeit der Formel nachzuweisen [4]. Als Beispiel für ein unvollständiges Verfahren lässt sich Greedy-SAT anführen. Dieser startet mit einer beliebigen Belegung und versucht diese schrittweise zu einer erfüllenden Belegung auszubauen. Dabei wird in jedem Schritt die Belegung derjenigen Variablen invertiert, die die Anzahl an neu erfüllten Klauseln maximiert. Der Algorithmus endet sobald eine erfüllende Belegung gefunden wurde oder die Obergrenze an Iterationen erreicht wurde. In letzterem Fall bleibt der Status der Formel ungewiss. Der im folgenden vorgestellte DPLL-Algorithmus gehört dagegen zu den vollständigen Verfahren. Er beruht auf einem Backtracking-Suchalgorithmus und bildet die Grundlage für das später vorgestellte Klausellernen.

2.1. Notation

Die im folgenden vorgestellte Notation und Darstellung der Algorithmen orientiert sich an [29]. Eine aussagenlogische Formel wird mit ϕ bezeichnet. Es wird im weiteren stets davon ausgegangen, dass alle verwendeten Formeln in *konjunktiver Normalform (KNF)* vorliegen. Ein Literal ist eine atomare Aussage oder deren Negation. Keine der Disjunktionen enthält ein Literal und seine Negation zugleich. Eine Belegung α wird als Sequenz der mit *wahr* belegten Literale dargestellt. Dabei entspricht die Reihenfolge der Literale in α der Reihenfolge ihrer Belegung. Die Erweiterung einer Belegung α um ein Literal l wird als αl notiert, die Auswertung einer Formel ϕ mit der möglicherweise partiellen Belegung α als $\phi \mid \alpha$.

Formel ϕ	$(x_0 \vee x_1) \wedge (\neg x_0 \vee x_2)$
Mengendarstellung von ϕ	$\{\{x_0, x_1\}, \{\neg x_0, x_2\}\}$
Belegung α mit $x_0 = \mathit{wahr}$, $x_1 = \mathit{falsch}$	$\alpha = x_0 \neg x_1$
Erweiterung um $x_2 = \mathit{falsch}$	$\alpha' = \alpha \neg x_2 = x_0 \neg x_1 \neg x_2$
Auswertung $\phi \mid \alpha$	$\{\{x_2\}\}$
Auswertung $\phi \mid \alpha'$	$\{\{\}\} \equiv \mathit{falsch}$

Tabelle 2.1.: Beispiele zur Notation

Für die Darstellung der Algorithmen wird die Mengendarstellung der Formel verwendet (Klausel-Normalform). Dabei entspricht jeder Disjunktion die Menge seiner Literale („Klausel“) und der gesamten Formel entspricht die Menge ihrer Klauseln. Die Auswertung $\phi \mid \alpha$ lässt sich in diesem Schema berechnen, indem man jede erfüllte Klausel aus ϕ streicht sowie jedes mit *falsch* belegte Literal aus allen Klauseln entfernt. Ist α eine erfüllende Belegung für ϕ, so gilt $\phi \mid \alpha = \{\}$. Ist α dagegen eine nichterfüllende Belegung, so enthält $\phi \mid \alpha$ mindestens eine leere Klausel.

2.2. DPLL-Algorithmus

Der DPLL-Algorithmus wurde Anfang der 1960er von Davis, Putnam, Logemann und Loveland entwickelt [14, 13, 29]. Er durchsucht den zu einer Variablenreihenfolge gehörenden Binärbaum aller möglichen Belegungen mittels Backtracking nach einer erfüllenden Belegung. Den Binärbaum der möglichen Belegungen erhält man, indem man in der Reihenfolge der Variablenordnung jede Variable mit den Werten *wahr* und *falsch* belegt. Eine Variablenordnung kann statisch (z. B. $x_0 < x_1 < x_2$) oder dynamisch sein (z. B. wähle jeweils diejenige Variable, die am häufigsten in einer Klausel minimaler Größe vorkommt). Eine statische Ordnung ändert sich während des Programmlaufs nicht. Eine dynamische Ordnung kann dagegen von veränderlichen Größen wie der ausgewerteten Formel oder der Belegung abhängen. Wurde der gesamte Baum erfolglos traversiert und damit alle möglichen Belegungen ausprobiert, so ist die Formel unerfüllbar. Andernfalls hat man mit einer erfüllenden Belegung den Nachweis für die Erfüllbarkeit der Formel erbracht.

Der DPLL-Algorithmus ist in Listing 2.1 angegeben und implementiert das eben beschriebene Suchverfahren. In Zeile 2 wird überprüft, ob eine erfüllbare Belegung gefunden wurde. Ist α

```
1  DPLL(φ, α):
2  if φ | α ist die leere Menge return SAT (Belegung gefunden)
3  if φ | α enthält eine leere Klausel return UNSAT (Konflikt)
4  if φ | α enthält eine Einheitsklausel {l} return DPLL(φ, αl) (Einheitsklausel)
5  if φ | α enthält ein reines Literal l return DPLL(φ, αl) (Reines Literal)
6      Sei l ein Literal aus einer Klausel mit minimaler Größe von φ | α (Suche)
7          if DPLL(φ, αl) == SAT
8              return SAT
9          else
10             return DPLL(φ, α¬l)
```

Listing 2.1: DPLL-Algorithmus

hingegen eine nicht-erfüllende Belegung, so wird in Zeile 3 ein Backtrackingschritt ausgeführt. Alle Teilbäume unterhalb einer partiellen nicht-erfüllenden Belegung enthalten nur nicht-erfüllende Belegungen und müssen daher nicht traversiert werden. In den Zeilen 4 und 5 werden weiter unten erläuterte Schlussregeln angewandt um den Suchraum einzuschränken, ohne dabei mögliche Lösungen zu übersehen. In Zeile 6 wird ein unbelegtes Literal einer Klausel minimaler Größe gewählt und mit diesem der Rekursionsschritt vollzogen. Bei der Literalwahl handelt es sich um eine Heuristik, die sicher stellen soll, dass Klauseln, die nur durch wenige Literale erfüllt werden können, früh bzw. oben im Suchbaum belegt werden. Heuristiken sind Variablenordnungen und bestimmen zusammen mit der Formel die Struktur des Suchbaums. Neben der verwendeten gibt es eine Reihe weiterer Heuristiken. Allen gemeinsam ist, dass sie die Suche zwar beschleunigen können, ihre Effektivität jedoch von der Struktur der Formeln abhängt [36].

Um den Suchraum weiter einzuschränken, verwendet der DPLL-Algorithmus die folgenden Schlussregeln. Diese werden sowohl auf die Ausgangsformel ϕ als auch auf die Zwischenergebnisse $\phi \mid \alpha$ angewandt.

Reines-Literal (Pure-Literal): Kommt ein Literal in einer Formel in nur einer Polarität vor, so heißt das Literal *rein* und wird mit *wahr* belegt. Besitzt eine Formel eine erfüllende Belegung α, die ein reines Literal l mit *falsch* belegt, so besitzt sie auch eine erfüllende Belegung α', wobei α' mit α übereinstimmt, jedoch l mit *wahr* belegt. Dies liegt daran, dass die Belegung α' mindestens genauso viele Klauseln erfüllt wie α, da sich durch die Änderung der Belegung keine der Klauseln zu *falsch* auswertet. Dazu müsste mindestens

eine Klausel $\neg l$ enthalten, was aufgrund der Reinheit jedoch nicht gegeben ist. Im Suchbaum reicht es zur Ermittlung der Erfüllbarkeit demnach aus, wenn man nur denjenigen Teilbaum traversiert, der ein reines Literal mit *wahr* belegt.

Die Schlussregel wird in aktuellen Solvern meist nicht mehr implementiert, da der zusätzliche Berechnungsaufwand zur Feststellung reiner Literale als nicht lohnend bewertet wird [29].

Einheitsklausel (Unit-Propagation, Boolean Constraint Propagation): Enthält eine Formel eine Klausel, die unter der Belegung α ein unbelegtes Literal l enthält und alle restlichen Literale der Klausel mit *falsch* belegt sind, so wird l mit *wahr* belegt. Teilbäume, die das Literal mit *falsch* belegen, werten mindestens eine Klausel und damit die gesamte Formel zu *falsch* aus und müssen daher nicht weiter untersucht werden. Die Erweiterung der Belegung entsprechend dieser Regel, kann zu weiteren Einheitsklauseln führen. Moderne SAT-Solver verwenden zur effizienten Feststellung und Bearbeitung von Einheitsklauseln das Two-Watched-Literal Schema. Auf dieses wird in Abschnitt 2.4 näher eingegangen.

2.3. CDCL-Algorithmus

Um weitere Teilbäume von der Suche auszuschließen und damit das Verfahren zu beschleunigen, wurde der DPLL-Algorithmus Mitte der 1990er [4] um das Klausellernen erweitert. Klausellernende SAT-Solver werden auch als CDCL-Solver *(Conflict-Driven-Clause-Learning)* bezeichnet. Die Grundidee besteht darin, dass man Belegungen, die ein Backtracking notwendig machen, analysiert und die so gewonnenen Erkenntnisse nutzt, um „gleichartige“ Belegungen bei der weiteren Suche auszuschließen. Die Idee stammt aus dem Umfeld der Constraint Satisfaction Probleme (CSP) [2] und wurde z. B. in [37] erweitert und auf SAT-Solver übertragen. Das Klausellernen führt auf vielen Instanzen zu signifikanten Beschleunigungen [46]. Einen Erklärungsansatz, warum das so ist, liefert [3]. Das Klausellernen erfolgt in drei Schritten, die der Reihe nach wie folgt ablaufen:

1. **Informationssammlung:** Eine Variable kann entweder im Rahmen der Suche oder durch Unit-Propagation belegt werden. Die Informationen, welche Variable aus welchem Grund und in welcher Reihenfolge gesetzt wurde, wird aufgezeichnet. Die erfassten Informationen werden im Falle eines Konflikts in der Konfliktanalyse benötigt.

2. Konfliktanalyse: Eine Belegung, in der sich mindestens eine Klausel zu *falsch* auswertet, heißt Konflikt. Tritt während der Suche ein Konflikt auf, so werden die gesammelten Informationen im Rahmen der Konfliktanalyse ausgewertet. Die Konfliktanalyse erzeugt eine Konfliktklausel, welche die Ursache des Konflikts repräsentiert. Die Konfliktklausel soll bei der weiteren Traversierung diejenigen Teilbäume abschneiden, die zu ähnlichen Konflikten führen würden.

3. Lernen: Der Lernschritt besteht aus dem Hinzufügen der erzeugte Konfliktklausel C zur Menge der Klauseln von ϕ. Dadurch wird die Ursprungsformel im Lauf des Verfahrens erweitert ($\phi' = \phi \cup \{C\}$). Indem die Konfliktklausel C nur durch Resolution von Klauseln aus ϕ und Resolventen davon erzeugt wird, gilt[1] $\phi \equiv \phi \cup \{C\}$. Damit ist das vom Algorithmus ermittelte Ergebnis sowohl für die um Konfliktklauseln erweiterte Formel ϕ' als auch für die unveränderte Ursprungsformel ϕ gültig. Durch die Hinzunahme der Konfliktklausel zur Formel ϕ werden die Schlussregeln ebenfalls auf die Menge der gelernten Klauseln angewandt.

Konfliktgraph und Konfliktanalyse

Zur Erläuterung der Schritte 1 und 2 des Klauselllernens wird zunächst der Konfliktgraph [3] eingeführt. Er repräsentiert die Struktur einer konfliktverursachenden Belegung. Eine Variablenbelegung kommt sowohl beim DPLL- als auch beim CDCL-Algorithmus aus einem von zwei möglichen Gründen zustande: Entweder wird ein Literal im Rahmen der Suche oder durch Erkennung einer Einheitsklausel gesetzt. Wurde ein Literal im Rahmen der Suche gesetzt, so heißt es *Entscheidungsliteral*, andernfalls *impliziertes Literal* (entsprechend heißen Variablen Entscheidungsvariablen bzw. implizierte Variablen). Jedem Literal wird ein *Entscheidungslevel (kurz Level)* zugeordnet. Das Entscheidungslevel einer Variable x zu einer Belegung α gibt an, wie viele *Entscheidungsvariablen* von α nicht nach x belegt wurden. Variablen, die durch die Unit-Propagation noch vor der Suche (also dem Belegen von Entscheidungsvariablen) gesetzt werden, haben daher das Level 0.
Das Beispiel in Tabelle 2.2 veranschaulicht dies: Die Variablen x_1, x_3, x_9 werden noch vor der eigentlichen Suche gesetzt und haben daher das Level 0. Als erste Entscheidungsvariable wird x_4 belegt. Durch Unit-Propagation werden in der Folge keine weiteren Variablen belegt. Als

[1] Wenn F eine Formel in KNF und Mengendarstellung ist, K_1 und K_2 Klauseln von F, sowie R der Resolvent aus K_1 und K_2, so sind F und $F \cup \{R\}$ äquivalent [35].

Belegung α:	x_1 $\neg x_3$ x_9	$\neg\mathbf{x_4}$	$\mathbf{x_{12}}$ x_{15} x_6	$\mathbf{x_5}$
Entscheidungslevel:	0	1	2	3

Tabelle 2.2.: Beispiel zum Entscheidungslevel (Entscheidungsliterale hervorgehoben)

zweite Entscheidungsvariable wird x_{12} belegt, dies führt zu weiteren implizierten Variablen (x_{15}, x_6), die ebenfalls vom Level 2 sind. Als dritte Entscheidungsvariable wird x_5 gesetzt, was wiederum zu keinen weiteren implizierten Variablen führt.

Formal ist der Konfliktgraph G ein gerichteter azyklischer Graph, dessen Kanten mit je einer Klausel beschriftet sind. Er lässt sich in Anlehnung an [21] im Konfliktfall wie folgt zu einer gegebenen Belegung berechnen:

1. Für jedes Entscheidungsliteral wird ein gleichnamiger Knoten erzeugt. Alle so erzeugten Knoten haben auch am Ende den Eingangsgrad 0.

2. Solange eine Klausel $\omega = \{l_1, \ldots, l_k, l\}$ existiert, so dass $\neg l_1$, …, $\neg l_k$ Knoten von G sind:

   a) Wenn der Knoten $\neg l$ nicht in G enthalten ist (Unit-Propagation):

      - Füge den Knoten l zu G hinzu, sofern noch nicht vorhanden.

      - Für i mit $1 \leq i \leq k$ füge die mit ω beschriftete Kante (l_i, l) ein.

   b) Sonst (Konflikt):

      - Füge den Knoten Λ zu G hinzu.

      - Füge die mit ω beschriftete Kante $(\neg l, \Lambda)$ sowie die mit ω beschriftete(n) Kante(n) $(\neg l_i, \Lambda)$ für $1 \leq i \leq k$ ein.

      - Entferne alle Knoten, die keinen Pfad zum Konfliktknoten Λ haben.

Eine anschauliche Erklärung für das Lernen basiert auf Schnitten[2] des *Konfliktgraphen*, die alle Entscheidungsliterale auf einer Seite (*konfliktverursachende Seite, reason side*) und den Konfliktknoten Λ auf der anderen Seite (*Konfliktseite, conflict side*) enthalten. Verknüpft man die Literale deren ausgehende Kanten im Schnitt liegen per Konjunktion, so erhält man

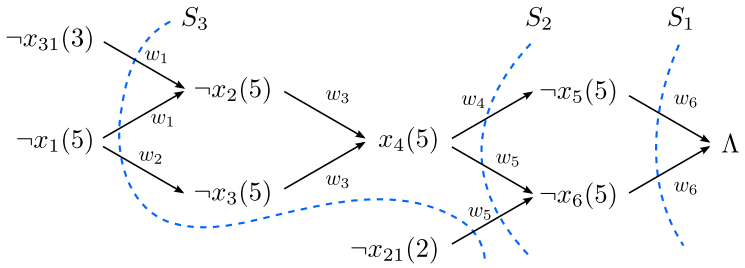
[2] Der Schnitt $S(X, V \setminus X)$ eines Graphen $G(V, E)$ ist die durch $S(X, X \setminus V) := \{(u, v) \in E \mid u \in X \wedge v \in V \setminus X\}$ definierte Kantenmenge

$$\phi = \{w_1, w_2, w_3, w_4, w_5, w_6, \ldots\}$$
$$w_1 = \{x_1, x_{31}, \neg x_2\} \quad w_2 = \{x_1, \neg x_3\} \quad w_3 = \{x_2, x_3, x_4\}$$
$$w_4 = \{\neg x_4, \neg x_5\} \quad w_5 = \{x_{21}, \neg x_4, \neg x_6\} \quad w_6 = \{x_5, x_6\}$$

(a) Formel ϕ

Belegung α:	–	...	$\neg\mathbf{x_{21}}$	$\neg\mathbf{x_{31}}$	...	$\neg\mathbf{x_1}, \neg x_2, \neg x_3, \neg x_4$
Entscheidungslevel:	0	1	2	3	4	5

(b) Belegung α



(c) Konfliktgraph für ϕ und α mit verschiedenen Schnitten (gestrichelt)

Abbildung 2.1.: Beispiel zur Konfliktanalyse

eine hinreichende Bedingung für das Auftreten des Konflikts. Damit ist die Negation der so gebildeten Bedingung eine notwendige Voraussetzung für die Erfüllbarkeit der Formel und genau diese bildet die Konfliktklausel [37, 29].
Abbildung 2.1 zeigt dies an einem Beispiel. Am Konfliktgraph ist erkennbar, dass die Entscheidungsliterale mit dem Entscheidungslevel 1 und 4 nicht am Konflikt beteiligt sind. Ferner sind für den Konflikt drei mögliche Schnitte (S_1, S_2, S_3) eingezeichnet. Als hinreichende Bedingungen für den Konflikt ergeben sich $\neg x_5 \wedge \neg x_6$ (für S_1), $x_4 \wedge \neg x_{21}$ (für S_2) und $\neg x_{31} \wedge \neg x_1 \wedge \neg x_{21}$ (für S_3). Entsprechend erhält man zum jeweiligen Schnitt die Konfliktklausel: $C_1 = \{x_5, x_6\}$, $C_2 = \{\neg x_4, x_{21}\}$, $C_3 = \{x_{31}, x_1, x_{21}\}$.

Grundsätzlich kann man bei einem Konflikt jede Konfliktklausel lernen (auch mehrere). Anhand des Schnittes S_1 sieht man jedoch, dass nicht jede Konfliktklausel gleich viel Erkenntnisgewinn liefert, da C_1 genau der konfliktverursachenden Klausel entspricht. Die verschiedenen Möglichkeiten einen Schnitt zu bilden werden auch Lern-Schemata genannt. Die Arbeit [46]

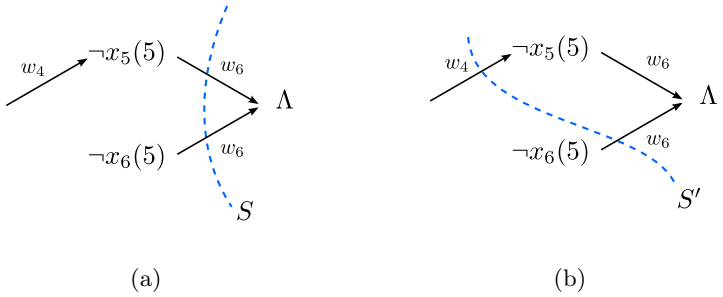


Abbildung 2.2.: Durch Resolution der Konfliktklausel eines Schnittes S mit w_4, erhält man die Konfliktklausel des um einen Knoten verschobenen Schnittes S'

vergleicht experimentell verschiedene Lern-Schemata miteinander. Minisat verwendet zum Lernen das First-Implication-Point Schema (auch 1-UIP abgekürzt) [42]. Es soll kleinere und damit allgemeinere Konfliktklauseln als andere Schemata erzeugen [4].

Neben der Herleitung am Konfliktgraphen, lässt sich jede Konfliktklausel auch iterativ über Resolution erzeugen. Dies wird anhand des Beispiels aus Abbildung 2.1 in Abbildung 2.2 skizziert. Zunächst wird die Konfliktklausel C mit der konfliktverursachenden Klausel (w_6) initialisiert. Der zu dieser Konfliktklausel korrespondierende Schnitt S ist in Abbildung 2.2(a) gezeigt. Durch einmalige Anwendung des Resolutionsoperators auf C und den Vorgänger (w_4) der Resolutionsvariablen (x_5) erhält man die Konfliktklausel C'. Diese entspricht dem Schnitt S' (siehe Abbildung 2.2(b)). Auf diese Art und Weise lässt sich der Schnitt des Konfliktgraphen um jeweils einen Knoten verschieben und damit jede zu einem Schnitt gehörende Konfliktklausel berechnen. Einziges Unterscheidungsmerkmal der verschiedenen Lernschemata mit Resolution ist die Abbruchbedingung. Ausführlichere Darstellungen zum Zusammenhang des Konfliktgraphen und der Resolution finden sich in [29, 3].

Der Basisalgorithmus klauselllernender SAT-Solver ist in Listing 2.2 angegeben. Der Algorithmus startet mit der Ausgangsformel ϕ und der leeren Belegung. Die Belegung α wird solange durch Entscheidung und Unit-Propagation erweitert bis ein Konflikt auftritt (Zeile 5) oder eine erfüllbare Belegung gefunden wurde (Zeile 13). Tritt der Konflikt auf Level Null auf, so ist die Formel unerfüllbar, da der Konflikt nur auf Variablenbelegungen durch Unit-Propagation beruht. Tritt der Konflikt auf einem höheren Level als Null auf, so wird wie oben gezeigt, eine Konfliktklausel generiert. Durch Anwendung des 1-UIP Lernschemas (First Unique Implication Point) wird für die Konfliktklausel garantiert, dass sie genau ein Literal enthält, das auf

```
1  CDCL(φ):
2  Γ = φ, α = ∅, level = 0
3  repeat:
4      UnitPropagation(Γ, α)
5      if Konflikt aufgetreten
6          if level = 0 return UNSAT
7          C = die aus dem Konflikt gelernte Klausel
8          p = das einzige Literal aus C, das auf dem Konfliktlevel gesetzt wurde
9          level = max{level(x)  | x ∈ C − p}
10         α = α ohne jede Belegungen die auf einem Level größer als level gemacht wurde
11         (Γ, α) = (Γ ∪ {C}, αp)
12     else
13         if α ist total return SAT
14         Wähle ein Literal p aus Γ | α
15         α = αp
16         level = level + 1
```

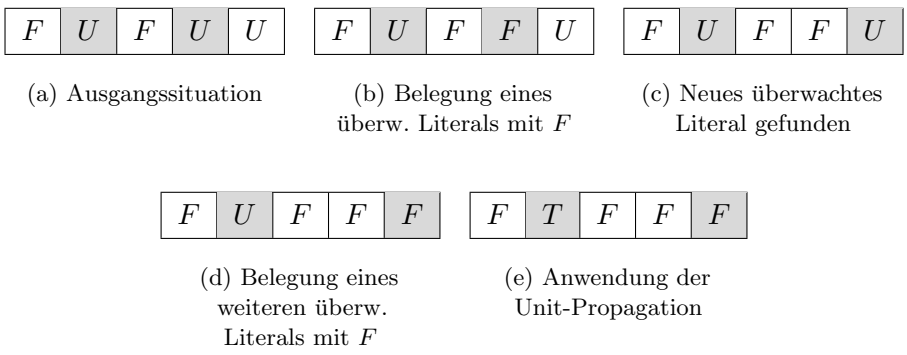
Listing 2.2: CDCL-Algorithmus

dem Konfliktlevel gesetzt wurde (*asserting clause*). Alle anderen Literale haben ein kleineres Level. Durch die Rücknahme aller Belegungen des höchsten Entscheidungslevels wird C damit zur Einheitsklausel (alle Literale bis auf p sind mit *falsch* belegt). Das Backtracking fährt mit der Unit-Propagation auf dem höchsten Level fort, auf dem die Klausel C gerade noch Einheitsklausel ist. Dies stellt sicher, dass Belegungen die nicht unmittelbar zum Konflikt beigetragen haben, rückgängig gemacht werden. Die Vollständigkeit des Verfahrens bleibt davon unberührt [4]. Da die Konfliktklausel C nur über Resolution gewonnen wurde, ist jede Erweiterung von Γ um C äquivalent zur Ausgangsformel ϕ. Da alle Literale bis auf p von C mit *falsch* belegt sind, wird die Belegung um p erweitert und mit der Unit-Propagation fortgefahren.

Reale Implementierungen erweitern diesen Basisalgorithmus um weitere Funktionen wie z. B. Neustarts, Löschen von Klauseln und Heuristiken. Eine Übersicht über diese Erweiterungen gibt [29].

2.4. Two-Watched-Literal Schema für die Unit-Propagation

Einen Großteil der Laufzeit (ca. 80 % – 90 %, siehe Abschnitt 3) wird mit der Unit-Propagation verbracht. Es ist daher für die Gesamtlaufzeit des SAT-Solvers bedeutend, einen effizienten Algorithmus zur Erkennung und Propagierung von Einheitsklauseln zu verwenden. Eine einfache Art der Implementierung ordnet jeder Klausel einen Zähler zu, der die Anzahl der mit *falsch* belegten Literale zählt. Der Zähler wird immer dann inkrementiert, wenn ein Literal der Klausel mit *falsch* belegt wird. Eine Klausel ist genau dann eine Einheitsklausel, wenn ihr Zähler um eins kleiner als die Anzahl der Literale der Klausel ist. D. h. nach der Belegung eines Literals l werden alle Klauseln, in denen $\neg l$ vorkommt konsultiert und deren Zähler überprüft. Der Nachteil der Methode ist, dass der Zähler jedes Mal angepasst werden muss, wenn *ein Literal der Klausel* mit *falsch* belegt wird. Außerdem müssen die Zähler beim Backtracking wieder dekrementiert werden.



(a) Ausgangssituation

(b) Belegung eines überw. Literals mit F

(c) Neues überwachtes Literal gefunden

(d) Belegung eines weiteren überw. Literals mit F

(e) Anwendung der Unit-Propagation

Abbildung 2.3.: Ablauf des Two-Watched-Literal Schemas

Das vom Chaff-Solver eingeführte Two-Watched-Literal Schema [30] vermeidet die genannten Nachteile. Die Idee besteht darin, für jede Klausel zwei (beliebige) Literale, die anfangs nicht mit *falsch* belegt sind, auf Belegungen mit *falsch* zu überwachen. Diese werden im folgenden überwachte Literale genannt. Wenn eine Klausel das erste Mal zur Einheitsklausel wird, so muss zwingend eines ihrer beiden überwachten Literale mit *falsch* belegt worden sein (andernfalls kann es sich nicht um eine Einheitsklausel handeln). Es genügt daher, die Klausel nur noch dann zu konsultieren, wenn eines ihrer überwachten Literale mit *falsch* belegt wird. In diesen Fällen wird dann geprüft, ob ein Ersatz für das überwachte Literal gefunden werden

kann. Erst wenn sich kein solches Literal finden lässt, wird überprüft, ob es sich um eine Einheitsklausel handelt. Ein weiterer Vorteil des Schemas ist, dass die überwachten Literale beim Backtracking nicht angepasst werden müssen, da sie gültig bleiben. Der einzige kritische Fall ist die Situation einer Einheitsklausel. Hierbei ist ein überwachtes Literal mit *wahr*, das andere mit *falsch* belegt. Da das mit *wahr* belegte Literal durch Unit-Propagation nach dem mit *falsch* belegten Literal gesetzt wurde, besitzen beide das gleiche Entscheidungslevel. Damit kann es nicht vorkommen, dass nur das mit *wahr* belegte Literal zurückgenommen wird. Entweder werden beide Belegungen oder keine zurückgenommen. Abbildung 2.3 zeigt den Ablauf des Two-Watched-Literal Schemas an einem Beispiel, Abschnitt 3.1 die Implementierung in Minisat.

2.5. Verwandte Arbeiten zur Parallelisierung

Die bereits vorhandenen Ansätze zur Parallelisierung von SAT-Solvern lassen sich mit Hilfe von zwei Kriterien klassifizieren [38]. Das erste Kriterium vergleicht die Ansätze nach Art des verwendeten Rechenmodells. Hier lassen sich *Shared-Memory-* und *Message-Passing-Systeme* unterscheiden. Der Vorteil ersterer besteht darin, dass die Kommunikation über den gemeinsamen Speicher mit nur wenig Overhead und geringer Latenz abgewickelt werden kann. Dafür richtet sich die Anzahl der Rechenkerne nach den aktuellen technischen Möglichkeiten. Darüber hinaus kann eine, von den Rechenkernen gemeinsam genutzte Ressource, wie z. B. der Speicher, zum Flaschenhals werden. Ein Message-Passing-System besteht hingegen aus beliebig vielen Rechenknoten, die keinen gemeinsamen Speicher besitzen. Die Kommunikation erfolgt über das Versenden und Empfangen von Nachrichten.

Das zweite Kriterium unterscheidet auf welcher Abstraktionsebene die Parallelisierung stattfindet. Abbildung 2.4 veranschaulicht die verschiedenen Herangehensweisen. Der *Portfolioansatz* setzt auf der obersten Ebene an und führt mehrere SAT-Solver parallel auf der gleichen Eingabe aus. Durch die Verwendung verschiedener Suchstrategien (bzw. Heuristiken) sowie eine unterschiedliche Initialisierung erreicht man, dass die parallel ausgeführten Suchprozesse verschiedene Teilbereiche des Suchraums bearbeiten. Der *Partitionierungsansatz* befindet sich eine Ebene darunter und teilt den Suchraum *eines* SAT-Solvers auf mehrere Prozessoren bzw. Knoten auf (Explorative Dekomposition). Eine Schwierigkeit hierbei ist die gleichmäßige Auslastung der Knoten, da nicht a priori bekannt ist, wie lange die Bearbeitung eines Teilbaums dauert. Zur dynamischen, überlappungsfreien Lastenverteilung wurde hierzu der *guiding path* eingeführt [45].

Er gibt an, welche Teile eines Suchbaums gerade bearbeitet werden bzw. welche parallel bearbeitet werden können. Der Vorteil des Partitionierungsansatzes unter Verwendung des guiding path ist, dass garantiert werden kann, dass kein Bereich des Suchraums mehrfach bearbeitet wird. Der Ansatz auf der niedrigsten Abstraktionsebene parallelisiert die *Unit-Propagation*. Aufgrund des hohen Laufzeitanteils von ca. 80 % ist die Unit-Propagation ein Kandidat um Beschleunigungen des Solving-Prozesses zu erzielen. Dies ist auch die Herangehensweise dieser Arbeit.

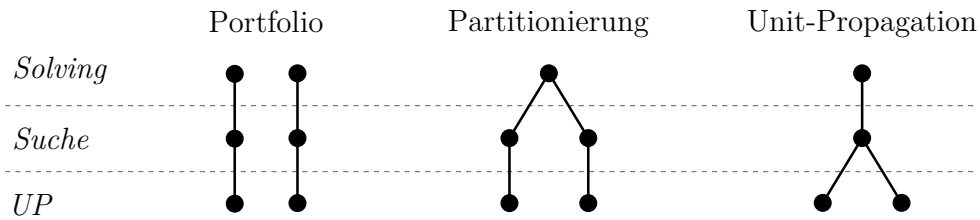


Abbildung 2.4.: Vergleich der Ansätze nach Parallelisierungsebene

Der Partitionierungsansatz wurde 1996 in [10] beschrieben und auf einem Parallelrechner implementiert. Ebenfalls auf dem Partitionierungsansatz basiert der SAT-Solver PaSAT [8, 39]. Eine der ersten Arbeiten, die den Portfolioansatz verwendet, stammt aus dem Jahr 2003 [7]. Hier wurden auf Basis des Parallelisierungsframeworks DOTS [6] und PaSAT die Ansätze Partitionierung und Portfolio miteinander kombiniert. Auf oberster Ebene wurden mehrere Instanzen von PaSAT auf verschiedenen Knoten gestartet. Sofern die Knoten über mehrere Rechenkerne bzw. Prozessoren verfügen, wurde das Problem unter diesen partitioniert. Beide Arbeiten, die auf PaSAT basieren, verwenden das sogenannte *Lemma-Exchange*, bei dem gelernte Konfliktklauseln zwischen Partitionen bzw. Knoten ausgetauscht werden. Die Projekte SATCIETY und ZetaSAT [34, 9] verwenden einen Partitionierungsansatz und wurden für Desktop Grids[3] entwickelt. Die Solver MiraXT [25] und SArTagnang [24] basieren im Gegensatz zu den bisher genannten Solvern nur auf dem Shared-Memory Modell. Ersterer verwendet einen Partitionierungsansatz, letzterer einen Portfolioansatz. Beide nutzen den effizienten Zugriff auf den gemeinsamen Speicher, in dem gelernte Klauseln gemeinsam benutzt werden (*clause sharing*). Da die Parallelisierung auch zu einer Randomisierung des Suchprozesses führt,

[3] Ein Desktop-Grid besteht aus einer Vielzahl von PCs, die sich in Hardware, Verfügbarkeit und Latenz deutlich voneinander unterscheiden können. Eines der ersten und bekanntesten Projekte dieser Art ist SETI@Home.

können die Laufzeit und die gefunden Modelle mehrerer Programmläufe stark variieren. In [20] wird ein Shared-Memory Portfolioansatz entwickelt, der vollständig deterministisch ist.

Diese Arbeit verfolgt ebenfalls einen Shared-Memory Ansatz, unterscheidet sich aber dadurch von anderen Arbeiten, dass sie auf der Ebene der Unit-Propagation ansetzt. Die einzige dem Autor bekannte Arbeit, die ebenfalls die Unit-Propagation parallelisiert, ist [27]. Hierin wird in einem Shared-Memory Ansatz die Klauseldatenbank partitioniert. Jeder Partition wird ein Thread zugeordnet, der die Unit-Propagation für diese ausführt. Die Threads tauschen Belegungen aus anderen Partitionen miteinander aus und iterieren solange bis keine Änderung mehr eintritt oder ein Konflikt vorliegt.

3. Problemanalyse

Die in diesem Kapitel vorgestellten Analysen basieren auf dem, von den Wissenschaftlern Eén und Sörensson in C++ entwickelten, SAT-Solver Minisat[1] [16, 42]. Er wurde wegen seines überschaubaren und strukturierten Quellcodes (ca. 2000 Codezeilen) sowie der guten Performance gewählt. Minisat belegte bei verschiedenen SAT-Solving Wettbewerben[2] mehrfach einen der ersten drei Plätze. Aufgrund der guten Erweiterbarkeit wurden sogar Unterwettbewerbe eingerichtet, bei denen verschiedene Minisat-Modifikationen gegeneinander antreten.

In diesem Abschnitt wird zunächst der von Minisat 2.2 verwendete Algorithmus in sequentieller Form präsentiert. Auf dieser Grundlage werden statistische Untersuchungen der sequentiellen Unit-Propagation durchgeführt und die Eignung verschiedener Parallelisierungsansätze diskutiert. Zur Vereinfachung der Logik wird auf die cache-optimierenden Blocker verzichtet [16]. Die Darstellung des Programmcodes erfolgt in einem an C++ angelehnten Pseudocode, bei dem der Übersichtlichkeit wegen, geschweifte Klammern durch eine entsprechende Einrückung ersetzt werden.

3.1. Unit Propagation in Minisat

Minisat verwendet das in Abschnitt 2.3 vorgestellte Two-Watched-Literal Schema. Da der Algorithmus Methoden und Datenstrukturen aus Minisat verwendet, werden diese zunächst erläutert.

Von Minisat verwendete Datenstrukturen und Funktionen

Literale und Variablen werden als Integer dargestellt. Die Funktion var(l) gibt die Variable eines Literals l zurück. Die Belegung eines Literals l kann mit der Funktion value(l) abgefragt werden

[1] http://minisat.se/
[2] SAT Competition: http://www.satcompetition.org, SAT Race

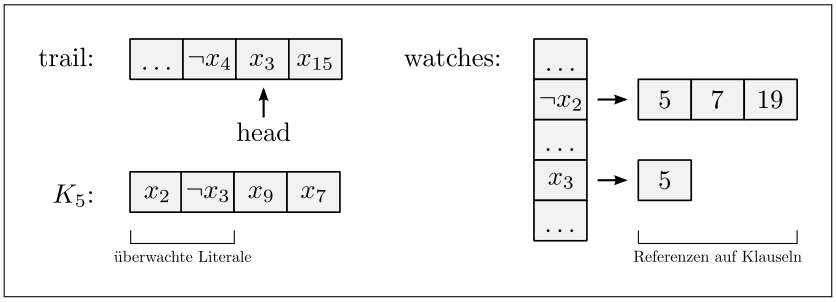


Abbildung 3.1.: Von den Algorithmen verwendete Datenstrukturen

und ist entweder *wahr*, *falsch* oder *undef*. Ein mit *undef* belegtes Literal wird im folgenden auch als unbelegt bezeichnet. Die Belegung wird als Array der mit *wahr* belegten Literale unter dem Namen trail gespeichert. Dabei entspricht die Reihenfolge im Array der Reihenfolge ihrer Belegung. Das trail-Array wird im folgenden auch als *Propagation-Queue* bezeichnet, da es die zu propagierenden Literale speichert. Zum schnellen Zugriff wird die Belegung in einem weiteren Array assigns[], nach Variablen indiziert, gespeichert. Ein Entscheidungsliteral l kann mit der Funktion uncheckedEnqueue(l) im assigns-Array mit *wahr* belegt und an das Ende der Propagation-Queue angefügt werden. Wird das Literal aufgrund einer Einheitsklausel gesetzt, so wird als weiterer Parameter die Einheitsklausel angegeben. Die Integervariable head gibt an, für welche Belegungen der Propagation-Queue bereits alle Klauseln konsultiert wurden. Alle Belegungen mit einem Index kleiner als head wurden bereits bearbeitet. Die Anzahl der Elemente eines Arrays a lässt sich mit a.size() ermitteln und ist eins größer als der letzte gültige Index. Da Literale und Variablen von Minisat als Integer repräsentiert werden, lassen sich Arrays auch mit Literalen und Variablen indizieren. Klauseln werden als Array von Literalen repräsentiert und lassen sich eindeutig über eine ganze, positive Zahl referenzieren. Das Array watches[l] speichert zu einem Literal l die Liste der Klauselreferenzen, in denen $\neg$l ein überwachtes Literal ist. Da die Unterscheidung zwischen einer Referenz auf eine Klausel und der Klausel selbst nicht immer von Bedeutung ist, werden diese synonym verwendet. Die Anzahl der Literale einer Klausel C lässt sich analog zu den Arrays mit C.size() ermitteln. Auf die Literale einer Klausel C kann mit eckigen Klammern (C[0], C[1], ..., C[C.size() - 1]) zugegriffen werden. Per Konvention speichert Minisat die beiden überwachten Literale am Anfang der Klausel (C[0] und C[1]).

Abbildung 3.1 veranschaulicht dies an einem Beispiel. Zu sehen ist das Endstück der aktuellen Belegung trail (Propagation-Queue). Die Variable head gibt an, dass die Klausellisten watches[x_3] und watches[x_{15}] noch nicht bearbeitet wurden. Für die Belegung von x_3 ist nur die Klausel 5

zu konsultieren, da x_3 nur in dieser Klausel ein überwachtes Literal ist. Da jede Klausel zu jedem Zeitpunkt genau zwei überwachte Literale hat, enthält auch das Array watches für jede Klausel genau zwei Referenzen auf die Klausel. Im Beispiel ist $\neg x_2$ das andere überwachte Literal der Klausel K_5.

Listing 3.1 zeigt den Pseudocode der Unit-Propagation in Minisat. Die äußerste Schleife läuft solange, bis alle Klauselllisten aller Belegungen bearbeitet wurden oder ein Konflikt auftritt. Die innere Schleife iteriert für die Belegung eines Literals p über alle Klauseln, die ¬p als überwachtes Literal enthalten. Bei der Bearbeitung einer Klausel wird zunächst sicher gestellt, dass das mit *falsch* belegte Literal, für das die Unit-Propagation aufgerufen wird, an Index 1 steht. Dadurch wird garantiert, dass das implizierte Literal immer an Position 0 steht. An verschiedenen Stellen im Code, wie z. B. bei der Implementierung des Resolutionsoperators, wird von dieser Annahme Gebrauch gemacht. Die Bearbeitung einer Klausel hat vier mögliche Ausgänge:

Fall 1 – Zweites überwachtes Literal mit wahr belegt: Ist C[0] – das zweite überwachte Literal – mit *wahr* belegt, so handelt es sich bei der Klausel nicht um eine Einheitsklausel. Zwar könnte man wie in Fall 2, nach einem neuen überwachten Literal als Ersatz für C[1] suchen, dies ist aber nicht notwendig. Stattdessen kann die Klausel so belassen und mit der nächsten fortgefahren werden. Das liegt daran, dass das Entscheidungslevel von C[1] größer oder gleich dem von C[0] ist, da C[1] aufgrund des zuletzt belegten Entscheidungsliterals bearbeitet wird. Daher kann die Klausel frühstens dann Einheitsklausel werden, wenn die Belegung von C[0] zurückgenommen wird. Da level(C[1]) ≥ level(C[0]) gilt, ist dann aber auch C[1] wieder unbelegt. Damit sind sowohl C[0], als auch C[1] unbelegt und gültige überwachte Literale. Hierbei handelt es sich also um einen Spezialfall von Fall 2, bei dem man die Suche nach dem neuen Literal einsparen kann.

Fall 2 – Neu überwachtes Literal gefunden: Nachdem Fall 1 nicht eingetreten ist, wird nach einem Literal gesucht, das mit *wahr* oder *undef* belegt ist. Wird dieses an der Position $k \geq 2$ gefunden, so werden C[1] und C[k] getauscht. Damit die Klausel bei der Belegung des neu überwachten Literals C[1] mit *falsch* konsultiert wird, wird eine Referenz auf die Klausel zu watches[¬C[1]] hinzugefügt. Wird kein Ersatz gefunden, so gilt, dass alle Literale C[1], . . . , C[C.size() - 1] mit *falsch* belegt sind. Da in Fall 1 bereits ausgeschlossen wurde, dass C[0] mit *wahr* belegt ist, kann C[0] nur mit *falsch* oder *undef* belegt sein.

Fall 3 – Konflikt: Ist C[0] mit *falsch* belegt, so handelt es sich um einen Konflikt, da sich die Klausel C zu *falsch* auswertet. Die globale Variable confl wird entsprechend auf die Referenz von C gesetzt. Nach abschließenden Arbeiten endet die Unit-Propagation und es folgt die in Kapitel 2 gezeigte Konfliktanalyse.

Fall 4 – Einheitsklausel: Ist C[1] unbelegt, so handelt es sich um eine Einheitsklausel. Daraufhin wird C[0] im assigns-Array auf *wahr* gesetzt und das Literal in die Propagation-Queue eingefügt. Um im Konfliktfall den Konfliktgraphen aufbauen zu können, wird in Form der Klauselreferenz gespeichert, welche Literale zur Belegung von C[0] geführt haben (dies sind C[1], ..., C[C.size() - 1]). Diese Aktionen sind in in uncheckedEnqueue gekapselt.

Wenn in Fall 3 ein neu überwachtes Literal l_{neu} gefunden wird, so muss die Referenz auf die Klausel, die wegen des alten überwachten Literals l_{alt} besteht, aus der Klauselliste watches[$\neg$l$_{alt}$] entfernt werden. Das Löschen erfolgt in Minisat implizit durch Überschreiben mit darauffolgenden Klauselreferenzen bzw. durch die Verkleinerung des Arrays am Ende der Bearbeitung der Klauselliste. Dazu gibt die Variable j die Position an, an die die gerade bearbeitete Klauselreferenz kopiert wird. Soll eine Klauselreferenz gelöscht werden, so genügt es, die Variable j nicht zu inkrementieren. In der Folge wird die Referenz entweder mit einer weiteren Referenz überschrieben oder durch die Verkleinerung des Arrays gelöscht. Nach der Bearbeitung der gesamten Klauselliste, wird das Array um die Anzahl der gelöschten Referenzen (i - j) verkleinert.

Reproduzierbarkeit von Programmläufen

Minisat wurde so implementiert, dass sich die Ergebnisse eines Programmlaufs reproduzieren lassen. Zwar hängt der Programmlauf von einem Pseudozufallsgenerator ab, dieser wird jedoch bei jedem Lauf mit dem gleichen Wert initialisiert. Führt man Minisat daher mehrmals auf der gleichen Eingabe aus, so wird jedes Mal der gleiche Suchbaum traversiert und die gleichen Konflikte sowie Einheitsklauseln gefunden. Eine Abweichung in den Laufzeiten mehrerer Läufe kann durch externe Faktoren (z. B. Systemlast, nicht reproduzierbares Scheduling) erklärt werden. Diese Schwankungen liegen jedoch im niedrigen einstelligen Sekundenbereich und werden vernachlässigt. Die im folgenden präsentierten Auswertungen beruhen daher auf jeweils einem Programmlauf.

```
1  Unit-Propagation :
2
3  while (head < trail.size()) // Alle Literale der Propagation-Queue
4      Lit p = trail[head++]
5
6      for (int i = 0, j = 0; i < watches[p].size();) // Alle Klauseln der Klauselliste
7
8          Sei C die durch watches[i] referenzierte Klausel
9          Vertausche C[0] und C[1], sodass C[1] == ¬p gilt
10
11         // Fall 1
12         if (value(C[0]) == wahr):
13             watches[j++] = watches[i++]
14             continue
15
16         // Fall 2
17         for (int k = 2; k < c.size(); k++)
18             if (value(C[k]) ≠ falsch):
19                 vertausche C[1] und C[k]
20                 füge Referenz auf Klausel C zu watches[¬C[1]] hinzu
21                 i++
22                 goto NextClause
23
24         // Fall 3 und 4
25         watches[j++] = watches[i++]
26         if (value(C[0]) == falsch)
27             Setzte confl so, dass confl auf Klausel C verweist
28             while (i < watches[p].size()) watches[j++] = watches[i++]
29             head = trail.size()
30         else
31             uncheckedEnqueue(C[0], Referenz auf C)
32
33         NextClause:
34
35     watches[p].shrink(i − j) // Klauselliste um gelöschte Referenzen verkleinern
```

Listing 3.1: Minisat: Sequentielle Unit-Propagation

3.2. Laufzeitanteil der Unit-Propagation

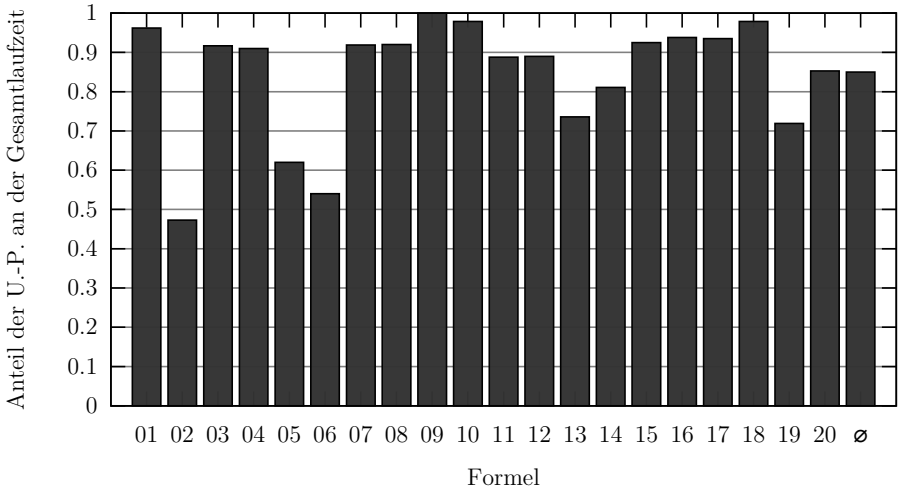


Abbildung 3.2.: Laufzeitanteil der Unit-Propagation

Zunächst wurde der Anteil[3] der Unit-Propagation an der Gesamtlaufzeit des SAT-Solvers für alle Testformeln (siehe Abschnitt 5.1) ermittelt. Über das Amdahlsche Gesetz lassen sich damit Rückschlüsse auf die theoretisch größt mögliche Beschleunigung des Gesamtverfahrens ziehen. Dazu wurde der Code aus Listing 3.1 so ergänzt, dass jeweils am Anfang und am Ende der Prozedur die zu einem Referenzzeitpunkt vergangenen Sekunden und Nanosekunden gemessen wurden. Die akkumulierte Differenz von Ende zu Start, ergibt die Laufzeit der Unit-Propagation. Implementierungsdetails der Zeitmessung sowie dabei auftretende Probleme und deren Lösung werden in Anhang B erläutert. Als Mittelwert für den Anteil der Unit-Propagation an der Gesamtlaufzeit erhält man 83 %, als Median 91 %. Diese Ergebnisse stehen in Einklang mit einer Untersuchung in [23], in der mit einem Sampling-Profiler ein Anteil der Unit-Propagation von 80,9 % für Formeln des Typs Application ermittelt wurde.

Amdahlsches Gesetz

Das Amdahlsche Gesetz [32] beschreibt um welchen Faktor ein Programm, bei einem sequentiellen Anteil s ($0 \leq s \leq 1$) und einer Anzahl an Prozessoren p, maximal beschleunigt werden

[3] Alle Zeiten beziehen sich (z. B. im Gegensatz zur CPU-Zeit) auf die tatsächlich vergangene Zeit, die sogenannte wall-clock time

kann.

$$S_{max}(p) = \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$$

Übertragen auf das SAT-Solving stellt die Unit-Propagation den parallelisierten Teil dar, der Rest des Solving-Prozesses läuft sequentiell ab. Die folgende Tabelle listet die maximal möglichen Beschleunigungen für verschiedene Anteile parallelisierbaren Codes auf. Geht man

parallelisierbarer Anteil $(1-s)$	0,75	0,8	0,85	0,9	0,95
S_{max}	4	5	6,7	10	20

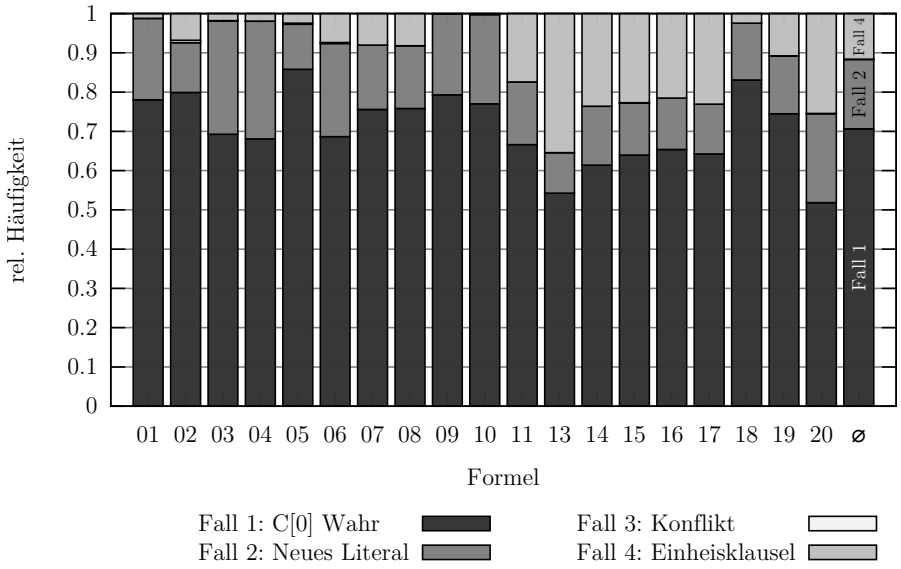
Tabelle 3.1.: Maximale mögliche Beschleunigung bei sequentiellem Anteil s

von einem Laufzeitanteil der Unit-Propagation von 80 % aus, so lässt sich der Solvingprozess damit maximal um den Faktor 5 beschleunigen. Superlineare Beschleunigungen, die z. B. durch Cacheffekte verursacht werden können, werden hierbei nicht berücksichtigt.

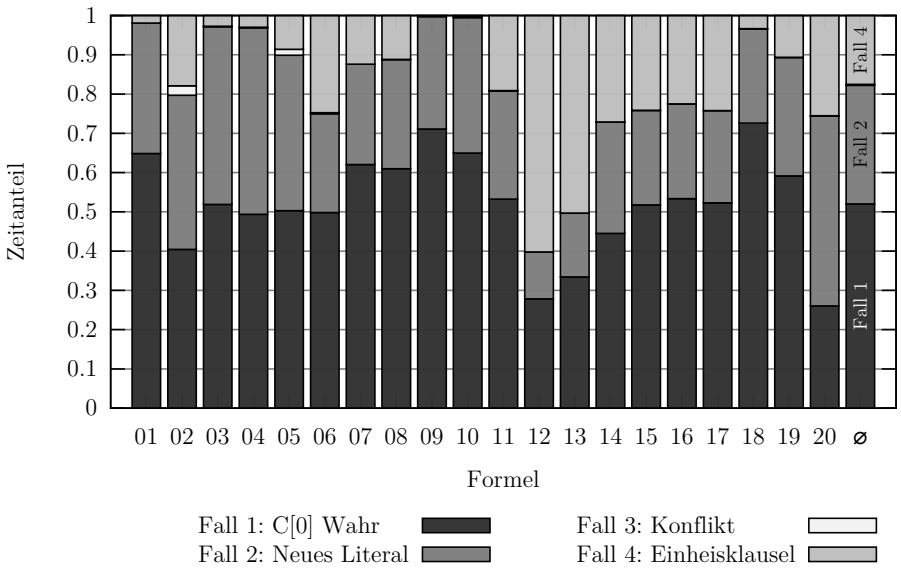
3.3. Fallhäufigkeiten und Fallzeitanteil

Um die Eignung verschiedener Parallelisierungsansätze zu beurteilen, wurde untersucht mit welcher relativen Häufigkeit jeder der vier möglichen Fälle eintritt. Dazu wurde der Code so instrumentalisiert, dass die Häufigkeit mit der ein Fall bei der Bearbeitung einer Formel eintrat, gezählt wurde. Das Ergebnis der anschließenden Auswertung zeigt Abbildung 3.3(a). Bildet man über alle Formeln den Durchschnitt, so tritt der Fall 1 (zweites überwachtes Literal ist *wahr*) in 70,6 %, Fall 2 (neu überwachtes Literal gefunden) in 17,6 % und Fall 4 (Einheitsklausel) in 11,6 % der Fälle ein. Ein Konflikt tritt in nur weniger als 1 % der Fälle ein.

Zwar tritt Fall 1 mit 70,6 % am häufigsten auf, da der in ihm ausgeführte Code jedoch nur zwei Zeilen umfasst, kann sein Anteil an der *Gesamtlaufzeit* geringer sein. Daher wurde neben der Fallhäufigkeit ermittelt, welchen Anteil an der Gesamtlaufzeit die einzelnen Fälle besitzen.



(a) relative Häufigkeiten der möglichen Fälle der Unit-Propagation



(b) relative Zeitanteil der möglichen Fälle an der Unit-Propagation

Abbildung 3.3.: Unit-Propagation: Fallhäufigkeit und Zeitanteil

Dazu wurde für jede bearbeitete Klauselreferenz und für jeden Fall getrennt erfasst, wie lange die Bearbeitung des Falles dauerte[4].

Die auf diesem Wege erhaltene Verteilung zeigt Abbildung 3.3(b). Danach wird etwa 52 % der Zeit mit der Bearbeitung des Falles 1 verwendet, 30 % der Zeit mit Fall 2 und 17 % mit Fall 4. Auch in der Zeitbetrachtung hat der Konfliktfall einen geringeren Anteil als 1 %.

3.4. Untersuchung des Parallelisierungspotentials

Bei der Parallelisierung von vorhandenem sequentiellem Code bieten sich insbesondere Schleifen an, da sie durch die wiederholte Ausführung einen potentiell hohen Laufzeitanteil haben. Sind die Schleifendurchläufe voneinander unabhängig, so lässt sich deren Ausführung parallelisieren. In diesem Abschnitt werden die drei vorhandenen Schleifen der Unit-Propagation jeweils unabhängig voneinander untersucht (siehe Listing 3.1). Dabei wird die Anzahl der Iterationen einer Schleife als Maß der verfügbaren Parallelität verwendet. Dies wird als Indikator dafür genutzt, ob eine Parallelisierung der untersuchten Schleife zu Beschleunigungen führen kann und wie viel Rechenkerne dabei ausgelastet würden. Um die Ergebnisse zu vergleichen, beschreibt der Wert P_i einer Schleife, in wie viel Prozent der Fälle mehr als i Iterationen ausgeführt wurden. So gibt $P_{10} = 80\,\%$ an, dass in 80 % der Fälle, zehn oder mehr Iterationen der Schleife durchgeführt werden.

Die äußerste while-Schleife iteriert über Belegungen der Propagation-Queue, um deren Klauselllisten zu bearbeiten. Eine Parallelisierung könnte die Klauselllisten verschiedener Belegungen parallel bearbeiten. In einem ersten Schritt wurde daher untersucht, wie oft die äußerste Schleife pro Aufruf der Unit-Propagation durchlaufen wird. Würden sich alle Schleifendurchläufe parallel ausführen lassen, so gibt Auswertung 3.4(a) Aufschluss über das Maß der vorhandenen Parallelität. Hierbei lassen sich erste Unterschiede zwischen den Formeltypen erkennen (die Formeltypen werden in Abschnitt 5.1 erläutert). Formeln des Typs Application führen pro Unit-Propagation-Aufruf mehr Iterationen der äußersten Schleife aus als Formeln des Typs Crafted. So beträgt für Formeln des Typs Crafted $P_{10} = 25\,\%$, für Formeln vom Typ Application ist $P_{10} = 71\,\%$.

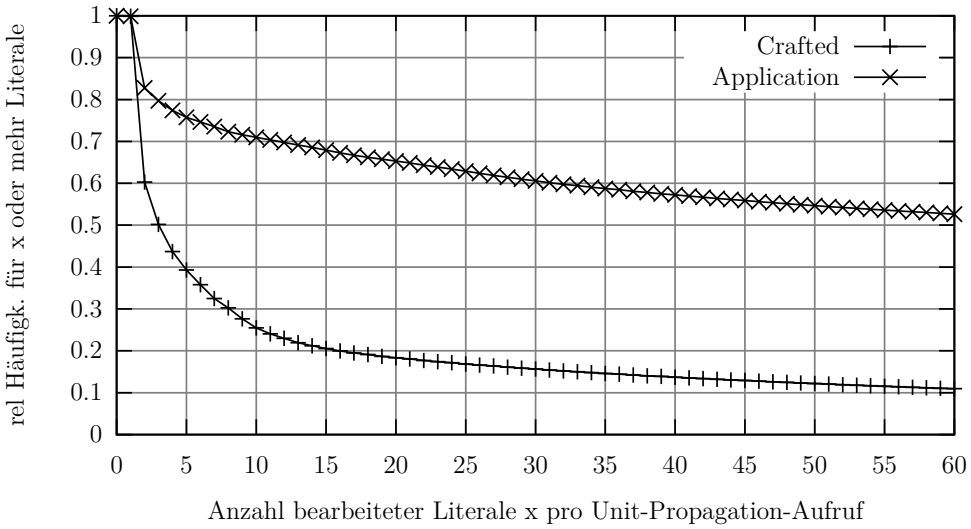
[4] so wurde z. B. für Fall 1 die Zeit gemessen die zur Ausführung der Zeile 7 - 13 benötigt wurde; Fall 2: 7-21, Fall 3: 7-29, Fall 4: 7-31

Diese Analyse betrachtet jedoch nur wie viele Belegungen insgesamt bearbeitet wurden, nicht wie viele davon gleichzeitig zur Verfügung stehen. Würde die Belegung eines Literals immer erst durch die Bearbeitung der letzten Klausel einer Klauselliste verursacht, so stünde zu jedem Zeitpunkt nur eine Belegung zur Verfügung. Um Aufschluss über die Anzahl der parallel bearbeitbaren Belegungen zu erhalten, wurde daher eine weitere Analyse durchgeführt. Hierbei wurde bei jeder Ausführung des Schleifenrumpfes die Anzahl der parallel bearbeitbaren Literale erfasst (trail.size() - head). Das Ergebnis in Abbildung 3.4(a) zeigt wieder deutliche Unterschiede zwischen den Formeltypen Application und Crafted. So konnten für Formeln des Typs Crafted zehn oder mehr Belegungen in nur 31 % der Fälle parallel bearbeitet werden. Für Formeln des Typs Application betrug der Wert dagegen 90 %.

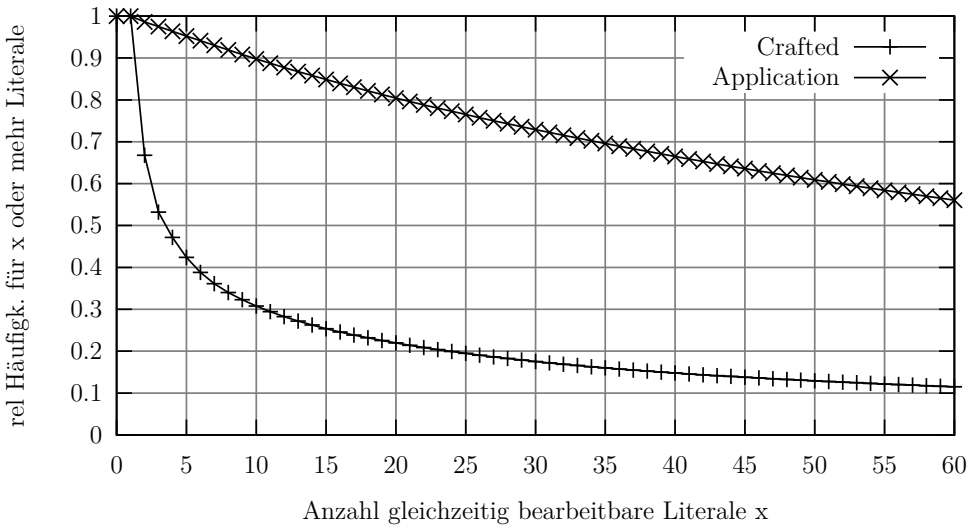
Die äußere for-Schleife iteriert für eine Belegung p über alle Klauseln, die $\neg p$ als überwachtes Literal enthalten. Die Länge der Klauselllisten wird als Indikator der verfügbaren Parallelität verwendet und entspricht der Anzahl an ausgeführten Iterationen der for-Schleife. Abbildung 3.5(a) zeigt eine Verteilung der bearbeiteten Listenlängen. Dazu wurde unmittelbar vor Ausführung der äußeren for-Schleife die Größe watches[p].size() erfasst. Hierbei zeigt sich, dass die Klauselllisten von Formeln des Typs Crafted länger sind als die des Typs Application. Für Formeln des Typs Crafted ist $P_{10} = 62\,\%$, für Formeln des Typs Application beträgt der Wert nur 12 %.

Die innere for-Schleife durchsucht eine Klausel nach einem neuen, nicht-überwachten Literal. Sie stoppt, sobald sie ein Literal gefunden hat, das nicht mit *falsch* belegt ist. Abbildung 3.5(b) zeigt die Auswertung über die Anzahl der Durchläufe. Überhaupt nicht betreten wird die Schleife, wenn eine Klausel aus nur zwei Literalen besteht. Für Formeln des Typs Application tritt dieser Fall bereits in etwa 60 % der Fälle ein, für Formeln des Typs Crafted nur in etwa 25 % der Fälle. Insgesamt ist das Parallelisierungspotential der Schleife gering, insbesondere für Formeln des Typs Application. Für diese beträgt bereits P_3 weniger als 3 %, aber auch für Formeln des Typs Crafted beträgt P_3 nur 34 %.

Die parallele Ausführung der innersten for-Schleife unterscheidet sich strukturell zu den beiden äußeren Schleifen. Bei der parallelen Ausführung der beiden äußeren Schleifen müssen, um die Korrektheit zu bewahren, jeweils alle Iterationen der Schleifen ausgeführt werden (mit Ausnahme des Konfliktfalls). Wird bei der parallelen Suche nach einem neuen Literal ein solches gefunden, können dagegen alle parallelen Suchprozesse abgebrochen werden (explorative

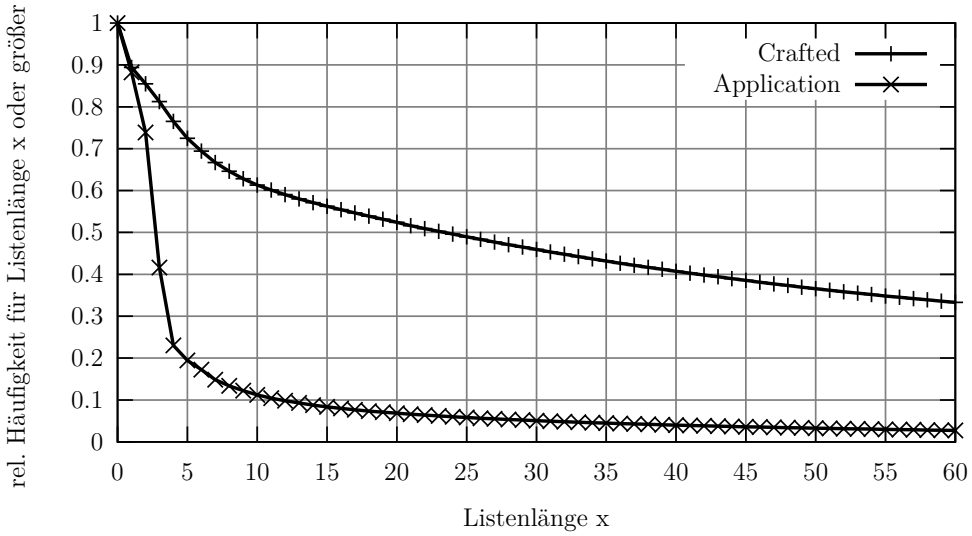


(a)

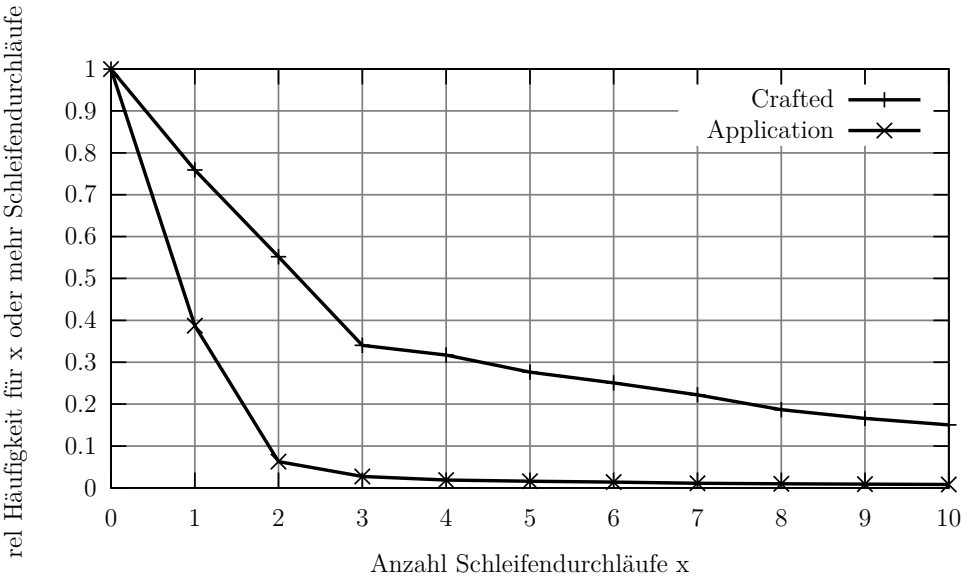


(b)

Abbildung 3.4.: Parallelisierungspotential der while-Schleife



(a) Länge bearbeiteter Klauselllisten



(b)

Abbildung 3.5.: Parallelisierungspotential der for-Schleifen

Dekomposition). Nur falls kein solches Literal gefunden wird, muss das Ergebnis aller parallelen Ausführungsstränge abgewartet werden.

3.5. Schlussfolgerungen

Zwar tritt Fall 1 in 70 % der Fälle ein, sein Anteil an der Gesamtlaufzeit beträgt jedoch nur 52 %. Würde man sich bei der Parallelisierung z. B. nur auf Fall 1 beschränken, so würde dessen Anteil an der Gesamtlaufzeit 42 % betragen. Für diesen Wert kann man durch die Parallelisierung des Falls jedoch nur eine Beschleunigung von weniger als zwei erreichen. Um größere Beschleunigungen zu erreichen, müssen die Lösungsansätze daher auch Fall 2 und Fall 4 berücksichtigen. Ansätze, die auf einer Parallelisierung der beiden äußeren Schleifen basieren, erfüllen diese Forderung. Aufgrund des geringen Anteils an der Gesamtlaufzeit, haben Konflikte nur einen geringen Einfluss auf mögliche Beschleunigungen und können daher nachrangig behandelt werden.

Bei einer Parallelisierung der while-Schleife wird aufgrund der Analysen erwartet, dass diese Maßnahme besonders bei Formeln des Typs Application Wirkung zeigt. Für Formeln des Typs Crafted ist die verfügbare Parallelität und damit mögliche Beschleunigungen jedoch gering. Genau umgekehrt verhält es sich mit der Parallelisierung der äußeren for-Schleife. Hier weisen Formeln des Typs Crafted einen deutlich höheren Anteil an parallel ausführbarer Arbeit auf als Formeln des Typs Application. Möchte man primär Formeln eines Typs beschleunigen, so kann man die dafür geeignete Schleife als ersten Ansatzpunkt wählen. Soll die Lösung für beide Formeltypen entworfen werden, so empfielt sich die gleichzeitige Parallelisierung beider Schleifen.

4. Parallele Unit-Propagation

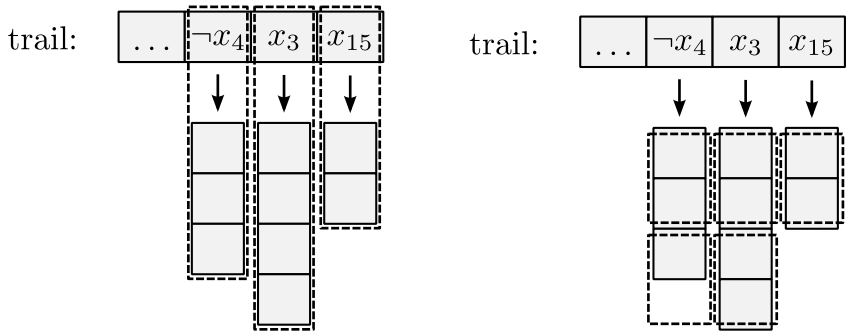
Zur Parallelisierung eines sequentiellen Programms werden im wesentlichen die Schritte *Dekomposition* und *Lastverteilung* [18] durchgeführt. Die Dekomposition zerlegt ein Programm in parallel ausführbare Einheiten, die Tasks genannt werden. Tasks spezifizieren die parallel ausführbaren Programmteile, legen aber noch nicht fest wie und wann diese ausgeführt werden. Die Abbildung von Tasks auf Prozessoren erfolgt erst durch die Lastverteilung. Als Prozessor ist hier und im folgenden eine unabhängige Ausführungseinheit wie z. B. ein Prozessorkern gemeint.

4.1. Parallelisierungsansätze

Zur Dekomposition von Programmen gibt es eine Reihe von Standardtechniken [18]. Die hier vorgestellten Lösungen basieren auf dem Ansatz der Daten-Dekomposition. Die Tasks ergeben sich hierbei durch die Partitionierung von Datenstrukturen. Ein Task bearbeitet diejenigen Aufgaben, die mit einer Partition der Datenstruktur verbunden sind. Auf diese Weise werden Tasks von der Partitionierung induziert. Die Arbeit verwendet den Daten-Dekompositionsansatz in zwei unterschiedlichen Granularitäten. Diese werden im folgenden erläutert und in Abbildung 4.1 dargestellt.

Grobgranularer Ansatz - Parallele Bearbeitung von Belegungen: Der erste Ansatz partitioniert die Belegungen der Propagation-Queue so, dass ein Task die gesamte Klauselliste einer Belegung bearbeitet. Dies entspricht der parallelen Ausführung der while-Schleife. Die Anzahl der gleichzeitig verfügbaren Belegungen bestimmt dabei den Grad der Parallelität. Sind weniger bearbeitbare Belegungen als Prozessoren verfügbar, so können nicht alle Prozessoren ausgelastet werden, obwohl sich die Bearbeitung einer Belegung in weitere unabhängige Teilaufgaben zerlegen lässt.

Feingranularer Ansatz - Parallele Bearbeitung von Klauseln: Um auch in Fällen, in denen nur wenige Belegungen in der Propagation-Queue verfügbar sind, eine hohe Zahl an parallel ausführbaren Tasks zu erzeugen, wurde in einem feingranularen Ansatz die Klauselllisten partitioniert (watches[p] für eine Belegung p). Dabei bearbeitet ein Task blockweise eine feste Anzahl von Klauseln einer Klauselliste. Der Ansatz entspricht damit der blockweisen Parallelisierung der for-Schleife. Wird die Blockgröße so gewählt, dass sie größer als jede während der Berechnung auftretende Klauselliste lang ist, so erhält man den grobgranularen Ansatz.



(a) ein Task bearbeitet die gesamte Klauselliste einer Belegung

(b) ein Task bearbeitet eine feste Anzahl an Klauseln (Blockgröße 2)

Abbildung 4.1.: Grob- und feingranulare Daten-Dekomposition (vertikal: Klauselllisten)

Die Bestimmung der Taskgranularität ist eine Abwägung zwischen der zur Verfügung stehenden Parallelität und dem durch die Taskverwaltung verursachten Mehraufwand (*Overhead*). Einerseits führt eine möglichst feine Aufteilung zu vielen Tasks und damit zu einer hohen Parallelität, andererseits ist mit jedem Task ein Mehraufwand verbunden (z. B. Task aus Taskpool entnehmen). Der Mehraufwand wird im Verhältnis zum Task umso größer, je kleiner der Task wird. Um diese Abwägung zu beeinflussen, werden zwei unterschiedlich granulare Ansätze entwickelt. Zudem lässt sich die Granularität des feingranularen Ansatzes über die Blockgröße steuern.

Eine Partitionierung der Klauseln bzw. die Parallelisierung der innersten for-Schleife wurde nicht durchgeführt. Dies wird damit begründet, dass das Parallelisierungspotential, insbesondere für Formeln des Typs Application, gering ist (vgl. Abschnitt 3.4). Darüber hinaus

unterscheidet sich die Literalsuche von den beiden äußeren Schleifen dadurch, dass das Finden eines neu überwachten Literals die Suche paralleler Ausführungsstränge überflüssig macht (*explorative Dekomposition*). Für die beiden äußeren Schleifen müssen dagegen, außer im seltenen Konfliktfall, alle Elemente bearbeitet werden. Andernfalls wäre die Korrektheit verletzt.

Die Lastverteilung erfolgt mit Hilfe eines (impliziten) Task-Pools [18]. Ein Task-Pool ist eine Datenstruktur, die die zur Verfügung stehende Parallelität verwaltet. Ein Prozessor kann Tasks aus dem Taskpool entnehmen und zu diesem hinzufügen. Bei einem impliziten Task-Pool werden keine separaten Datenstrukturen zur Verwaltung von Tasks benutzt, sondern auf bereits vorhandene zugegriffen. Mit Hilfe von Indizes kann angegeben werden, welche Tasks bereits bearbeitet wurden bzw. noch bearbeitet werden müssen. Immer wenn ein Prozessor leer läuft, versucht dieser einen Task aus dem Task-Pool zu entnehmen und zu bearbeiten (*Self-Sheduling*). Im grobgranularen Ansatz erfolgt die Taskvergabe mit Hilfe der Propagation-Queue (des trail-Arrays) sowie Indizes, die das nächste zu bearbeitende Element angeben. Im Falle des feingranularen Ansatzes wird ein weiteres Array für die Verwaltung der Blöcke eingeführt.

4.2. Probleme paralleler Ansätze

Wendet man die Fallbestimmung, wie in Listing 3.1 gezeigt, unverändert für mehrere Klauseln parallel an, so können eine Reihe von Problemen auftreten. Diese können sowohl grundsätzlicher Art als auch implementierungsspezifisch sein und werden im folgenden erläutert. Hierbei wird nicht nach der Granularität der Ansätze unterschieden, da die auftretenden Probleme bei beiden Ansätzen die gleichen sind.

4.2.1. Spontane Belegung

Durch parallel bearbeitete Einheitsklauseln können *unbelegte Literale* jederzeit mit *wahr* oder *falsch* belegt werden. Dies führt dazu, dass der Wahrheitswert von bestimmten booleschen Ausdrücken nur im Augenblick ihrer Auswertung gilt. Insbesondere führen Belegungen, die einen bereits ausgewerteten booleschen Ausdruck betreffen, zu Problemen. Listing 4.1 zeigt dies an einem Beispiel.

```
1  Sei l ein Literal das nur die Werte wahr, falsch und undef annehmen kann
2
3  if (value(l) == wahr)
4     // wahr-Zweig
5  else if(value(l) == falsch)
6    // falsch-Zweig
7  else
8    // else-Fall
```

Listing 4.1: Beispiel zur spontanen Belegung

In sequentiell ausgeführtem Code bedeutet das Eintreten des else-Falles, dass das Literal mit *undef* belegt sein muss. Dies gilt in der parallelen Version jedoch nicht mehr. Hier kann das Literal l durch eine parallel ausgeführte Belegung auch *wahr*, *falsch* oder *undef* sein. So tritt dieser Fall z. B. ein, wenn l bei den Prüfungen auf *wahr* bzw. *falsch* noch unbelegt ist, aber noch vor Eintritt des else-Falles belegt wird. Von diesem Problem betroffen sind alle booleschen Ausdrücke, die von einem unbelegten Literal impliziert werden, wie z. B. l != *wahr*,l != *falsch* und l == *undef*. Dieses Grundproblem tritt bei der parallelen Bearbeitung von Klauseln an den folgenden Stellen auf:

- in Fall 2: Tritt durch das Finden eines unbelegten Literals l Fall 2 ein, so wird l mit dem bisher überwachten Literal C[1] getauscht. Nachdem festgestellt wurde, dass l mit *undef* belegt ist, kann das Literal jedoch durch einen parallel laufenden Prozess mit *falsch* belegt werden. In diesem Fall würde ein mit *falsch* belegtes Literal als neu überwachtes Literal bestimmt werden. Damit wäre jedoch keine korrekte Umsetzung des Two-Watched-Literal Schemas gegeben.

- in Fall 3 und Fall 4 tritt das Problem auf, wenn das Literal C[0] bei der Prüfung auf Fall 3 und 4 mit *undef* belegt ist, vor Eintritt in den else-Fall jedoch mit *falsch* belegt wird. Dies führt dazu, dass anstatt eines Konflikts eine Einheitsklausel festgestellt wird.

- in Fall 3 und Fall 4: Ist im sequentiellen Algorithmus Fall 1 nicht eingetreten, so ist das Literal C[0] bei der Prüfung nicht mit *wahr* belegt. Diese Belegung gilt mindestens solange, bis der Algorithmus weitere Literale aufgrund einer Einheitsklausel belegt. Im sequentiellen Algorithmus wird daher bei der Prüfung, ob Fall 3 oder 4 eingetreten ist, der Wert von C[0] nicht mehr mit *wahr* verglichen. Wenn das Literal C[0] anfangs mit

undef belegt ist, gilt diese Voraussetzung in der parallelisierten Version jedoch nicht mehr. Daher kann in der parallelen Version Fall 4 eintreten, obwohl C[0] inzwischen mit *wahr* belegt ist.

4.2.2. Klauselvolatilität

Durch die parallele Bearbeitung der Klauselllisten kann es dazu kommen, dass zwei Prozesse simultan die gleiche Klausel bearbeiten. Es sind höchstens zwei Prozesse, da eine Klausel für jedes ihrer überwachten Literale genau einmal in einer Klauselliste vorkommt. Die gleichzeitige, nicht synchronisierte Veränderung einer Klausel kann jedoch zu einer falschen Anwendung des Two-Watched-Literal Schemas führen.

Grundsätzlich wird eine Klausel durch zwei Basisoperationen verändert:

- Wurde in Fall 2 ein neu überwachtes Literal $\mathsf{C[k]}_{k \geq 2}$ gefunden, so wird dieses mit dem bisher überwachten Literal C[1] getauscht.
- Bei der Bearbeitung einer Klausel, die aufgrund der Belegung von p bearbeitet wird, werden zu Beginn des Algorithmus die überwachten Literale so getauscht, dass $\neg\mathsf{p} == \mathsf{C[1]}$ gilt. Die Operation ist Minisat-spezifisch, da an verschiedenen Stellen (z. B. Resolutionsoperator, Klauselminimierung) in Minisat davon ausgegangen wird, dass C[0] das implizierte Literal der Klausel ist.

Werden die parallelen Zugriffe auf die Klauseln nicht synchronisiert, so wird das Two-Watched-Literal Schema nicht in allen Fällen korrekt angewendet. Dies wird in Abbildung 4.2 an einem Beispiel gezeigt. Dabei seien die Literale einer Klausel x_{f_0} und x_{f_1} beide mit *falsch* sowie x_u mit *undef* belegt. Der Prozess P_0 bearbeitet die Klauselliste von $\neg x_{f_0}$, P_1 die von $\neg x_{f_1}$. Nach dem Vertauschen der überwachten Literale, finden beide Prozesse das unbelegte Literal x_u als neu überwachtes Literal und tauschen dieses jeweils mit dem Literal C[1]. Die nicht-synchronisierten Tauschoperationen führen dazu, dass keine Einheitsklausel erkannt wird, obwohl beide überwachten Literale mit *falsch* belegt sind. Eine sequentielle Bearbeitung der Klauseln hätte dagegen zur Erkennung der Einheitsklausel geführt.

Ein ähnliches Beispiel lässt sich auch konstruieren, wenn man auf das Minisat-spezifische Vertauschen der beiden überwachten Literale C[0] und C[1] verzichtet und nur den Tausch von $\mathsf{C[k]}_{\mathsf{k}\geq 2}$ mit C[0] bzw. C[1] verwendet. So erhält man in Abbildung 4.2 durch das Streichen der

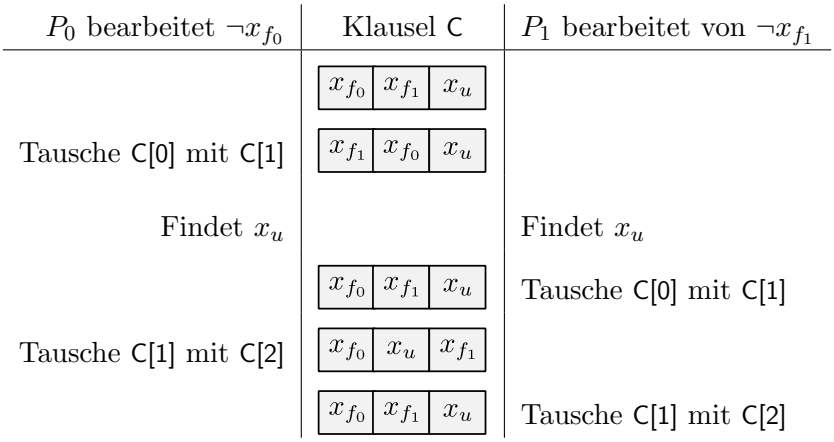


Abbildung 4.2.: Beispiel zum nicht synchronisierten Klauselzugriff zweier Prozesse

Tauschoperationen von C[0] mit C[1] einen Ablauf, in dem eine Einheitsklausel nicht in allen Fällen erkannt wird. In diesem Fall finden beide Prozesse das gleiche neu überwachte Literal C[k]. Nachdem der Prozess P_0 das alte überwachte Literal gegen das neue getauscht hat, ist das Literal C[2] mit *falsch* belegt. Tauscht anschließend der zweite Prozess sein altes überwachtes Literal gegen C[2] aus, so ist eines der beiden neu überwachten Literale mit *falsch* belegt. Dies führt im Beispiel dazu, dass eine Einheitsklausel nicht als solche erkannt wird.

4.2.3. Inkonsistenz gemeinsamer Datenstrukturen

Während der parallelen Bearbeitung mehrerer Klauseln greifen die Prozesse auf weitere gemeinsame Datenstrukturen zu. Dazu gehören z. B. die belegten Literale (der Propagation-Queue), die Informationen zum Aufbau des Konfliktgraphen, die Klauselllisten sowie der Taskpool. Die Zugriffe auf diese gemeinsamen Daten müssen so synchronisiert werden, dass deren Konsistenz gewährleistet ist. So dürfen z. B. parallel durchgeführte Belegungen nicht dazu führen, dass ein Thread die Belegung eines anderen überschreibt.

Auch die *Konsistenz* im engeren Sinne kann verletzt werden, wenn Operationen von anderen Threads nicht als atomar wahrgenommen werden:

- Nachdem ein neu überwachtes Literal l bestimmt wurde, wird eine Referenz auf die Klausel zur Liste watches[¬l] hinzugefügt. Wird das Literal ¬l mit *wahr* belegt, noch bevor die Referenz auf die Klausel zur Klauselliste hinzugefügt wurde, kann eine Einheitsklausel

bzw. ein Konflikt übersehen werden. Die Bestimmung des neu überwachten Literals und das Hinzufügen zur Klauselliste muss für alle Threads als atomare Operation wahrgenommen werden.

- Nachdem ein Literal im assigns-Array mit *wahr* belegt wurde, wird es in die Propagation-Queue eingefügt (siehe: Von Minisat verwendete Datenstrukturen und Funktionen, Abschnitt 3.1). Hierbei gilt es jedoch eine Einschränkung zu beachten. Damit der Konfliktgraph nicht explizit aufgebaut werden muss, wird die Reihenfolge der Propagation-Queue genutzt, um eine Breitentraversierung des Konfliktgraphen durchzuführen. Dabei muss für die Reihenfolge der Propagation-Queue gelten, dass ein Literal l vor den Literalen $l_1, \ldots, l_n$ steht, wenn l durch die Einheitsklausel $\{l, l_1, \ldots, l_n\}$ belegt wurde [16]. Diese Reihenfolge kann verletzt werden, wenn die Belegung und das Einfügen in die Propagation-Queue nicht atomar erfolgen.

  Folgendes Beispiel zeigt dies: Angenommen das Entscheidungsliteral x_0 wird belegt und in die Propagation-Queue eingefügt. In Folge der Belegung von x_0, wird das das implizierte Literal x_1 belegt aber noch nicht in die Propagation-Queue eingefügt. Wird nun von einem parallelen Thread die Einheitsklausel $\{x_2, \neg x_0, \neg x_1\}$ erkannt, so wird x_2 mit *wahr* belegt und in die Propagation-Queue eingefügt. Endet anschließend das Einfügen von x_1, so ist die Reihenfolge in der Propagation-Queue x_0, x_2, x_1. Diese so erhaltene Literal-Reihenfolge in der Propagation-Queue verletzt obige Gültigkeitsbedingung.

Die Schwierigkeit besteht in beiden Fällen darin, dass verschiedene Speicherbereiche atomar geändert werden müssen.

4.3. Sperrbasierte Lösungsansätze

Die sperrbasierten Lösungsansätze stellen durch den Erwerb von Sperren sicher, dass keine Verschränkung von parallelen Prozessen auftritt, die zu einem der oben genannten Probleme führen kann. Problematisch an diesem Ansatz ist, dass blockierte Prozesse nicht zum Fortschritt der Gesamtrechnung beitragen. Wenn viele Prozesse die gleichen Sperren anfordern, können diese zum Flaschenhals werden und die Skalierbarkeit des Ansatzes verringern (*Lock Contention*)[1]. Dieses Problem wird in einem zweiten, sperrfreien Ansatz vermieden.

Sowohl der grobgranulare als auch der feingranulare Ansatz unterscheiden sich nicht grundsätzlich darin, wie sie die oben aufgeführten Probleme ausschließen. Es wird daher zunächst der grobgranulare Lösungsansatz vorgestellt und im Anschluss daran gezeigt, wie sich dieser vom feingranularen Ansatz unterscheidet. Der sperrbasierte, grobgranulare Ansatz löst die in Abschnitt 4.2 angeführten Probleme wie folgt:

Das Problem spontaner Belegungen wird gelöst, indem für jede *Variable* eine Sperre eingeführt wird. Die Variablensperre steht für das ausschließliche Recht eine unbelegte Variable zu belegen. Bevor ein Prozess eine Variable belegen darf, muss dieser die zugehörige Variablensperre erwerben. Möchte ein lesender Prozess sicherstellen, dass sich eine Variable in einem bestimmten Codeabschnitt nicht ändert, so kann er dies durch den Erwerb der zugehörigen Sperre sicherstellen.

Die durch die Klauselvolatilität verursachten Probleme werden vermieden, indem für jede Klausel eine Sperre eingeführt wird. Bevor ein Prozess Änderungen an einer Klausel vornimmt, muss er sicher stellen, dass er die zur Klausel gehörende Sperre besitzt. Dies stellt damit sicher, dass eine Klausel zu keinem Zeitpunkt von mehr als einem Prozess verändert wird.

Die Konsistenz gemeinsamer Datenstrukturen wird sichergestellt, indem der gemeinsame Zugriff durch den Sperrerwerb synchronisiert wird. Wenn es sich bei den manipulierten Datenstrukturen nicht um komplexe Objekte, sondern um einen grundlegenden Datentyp von C++ handelt, können auch spezielle Prozessor-Befehle, die eine atomare Ausführung garantieren, verwendet werden (z. B. Compare and Swap).

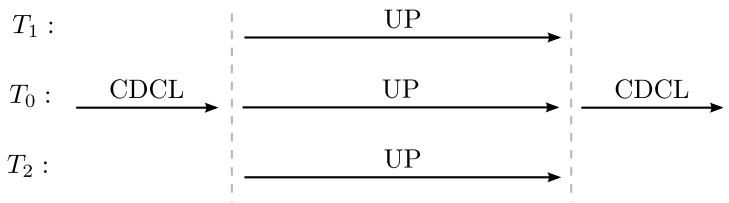


Abbildung 4.3.: Parallele Ausführung der Unit-Propagation

4.3.1. Grobgranulare, parallele Unit-Propagation

Der grobgranulare Ansatz nutzt genauso viele Threads wie dem Programm an Rechenkernen zur Verfügung gestellt wird. Der Kontroll-Thread T_1 führt den CDCL-Algorithmus aus, bis er an eine Stelle kommt, an der die Unit-Propagation ausgeführt werden soll. Daraufhin aktiviert er die Threads $T_2, \ldots, T_n$ durch das Setzen einer gemeinsamen Variablen, die laufend von den wartenden Threads abgefragt wird (*Polling*). Daraufhin führen alle Threads, also auch T_1, die Unit-Propagation bis zum Eintreten des Abbruchkriteriums aus. Abbildung 4.3 veranschaulicht dies für $n = 3$.

Taskverwaltung

Die Taskverwaltung erfolgt durch atomare Manipulation der Variablen head sowie der threadspezifischen Variablen head_i. Listing 4.2 zeigt den von jedem Thread ausgeführten Algorithmus, um einen Task zu erhalten. Methoden mit der Vorsilbe atomic geben an, dass die von ihr ausgeführte Operation atomar durchgeführt werden. Zunächst initialisiert jeder Thread seine Variable head_i, indem er den Wert von head atomar liest und inkrementiert. Nachdem dieser Schritt von allen Threads ausgeführt wurde, ist aufgrund der atomaren Operationen garantiert, dass sich alle threadspezifischen Variablen head_i voneinander unterscheiden. Wenn für einen Thread $\mathsf{head}_\mathsf{i} \geq \mathsf{trail.size()}$ gilt, so zeigt head_i auf Belegungen, die erst in der Zukunft durchgeführt werden. Der Thread geht daher in eine Schleife und wartet bis das Literal gültig wird oder das Abbruchkriterium eintritt. Alle Threads, deren Variable head_i auf ein gültiges Literal zeigen, lesen dieses und bearbeiten dessen Klauselliste. Danach setzt jeder Thread seine Variable head_i neu und dekrementiert die Anzahl an noch zu bearbeitenden Belegungen (openProps). Durch die Verwendung von atomaren Operationen wird garantiert, dass jede Belegung *genau einmal* bearbeitet wird (mit Ausnahme des Konfliktfalls).

Endeerkennung

Ein Thread geht wieder in den Wartezustand, wenn ein Konflikt aufgetreten ist oder alle Belegungen bearbeitet wurden. Erster Fall wird mit Hilfe der gemeinsamen Variablen confl erkannt, die im Konfliktfall auf die Konfliktklausel verweist. Andernfalls wird das Ende der Bearbeitung über die gemeinsame Variable openProps erkannt. Die Variable openProps wird

```
1  Thread i :
2
3  int head_i = atomic_fetch_and_add(&head, 1)
4  do
5      if (head_i < trail.size())
6          Lit l = trail[head_i]
7          bearbeite die Klauselliste von l
8          head_i = atomic_fetch_and_add(&head, 1)
9          atomic_sub_and_fetch(&openProps, 1)
10
11 while (openProps != 0 && kein Konflikt aufgetreten)
```

Listing 4.2: Minisat: Taskvergabe der Parallele Unit-Propagation

vor der Ausführung der Unit-Propagation mit 1 (bzw. tail.size() - head) initialisiert und gibt an, wie viele Belegungen noch bearbeitet werden müssen. Die Variable openProps wird jedes Mal inkrementiert bevor ein neues Literal belegt wird. Sie wird dekrementiert, nachdem eine Belegung vollständig bearbeitet wurde. Die Inkrementierung von openProps kann abgesehen von der Initialisierung nur *während* der Bearbeitung einer Klausel erfolgen. Der Zustand, in dem openProps == 0 gilt, ist daher stabil und beendet die parallel ausgeführte Unit-Propagation. Die Variable wird dann erst wieder nach dem Setzen von weiteren Entscheidungsliteralen des CDCL-Algorithmus erhöht. Die Reihenfolge, in der die Variable openProps manipuliert wird, stellt sicher, dass kein Thread vorzeitig stoppt.

Grobgranulare Bearbeitung der Klauselllisten

Der Algorithmus zur Bearbeitung der Klauselllisten wurde in zwei Listings aufgeteilt. Listing 4.3 zeigt die Bearbeitung der Fälle 1 und 2. Listing 4.4 schließt unmittelbar am Ende der Bearbeitung von Fall 2 an und zeigt die Bearbeitung der Fälle 3 und 4. Die Anweisungen lock x und unlock x stehen für den Erwerb bzw. die Freigabe der Sperre von x. Sollen mehrere Sperren gleichzeitig erworben bzw. freigegeben werden, so kann auch eine Liste als Parameter für lock angegeben werden. Der Erwerb bzw. ihre Freigabe erfolgt in der Reihenfolge ihrer Auflistung.

```
1  Bearbeitung der Klauselliste für das Literal p :
2
3  for (int i = 0, j = 0; i < watches[p].size();)
4
5      Sei C die durch watches[i] referenzierte Klausel
6      lock C
7      Vertausche C[0] und C[1], so dass C[1] == ¬p gilt
8
9      if (value(C[0]) == wahr):
10         watches[j++] = watches[i++] // Fall 1
11         unlock C
12         continue
13
14     for (int k = 2; k < c.size(); k++)
15         if (value(C[k] == undef)
16             lock var(C[k])
17             if (value(C[k] != falsch)
18                 vertausche C[1] und C[k] // Fall 2
19                 lock watches[¬C[1]]
20                 füge Referenz auf Klausel C zu watches[¬C[1]] hinzu
21                 unlock watches[¬C[1]], var(C[1]), C
22                 i++
23                 goto NextClause
24             unlock var(C[k])
25         else if (value (C[k]) == wahr)
26                 vertausche C[1] und C[k] // Fall 2
27                 lock watches[¬C[1]]
28                 füge Referenz auf Klausel C zu watches[¬C[1]] hinzu
29                 unlock watches[¬C[1]], C
30                 i++
31                 goto NextClause
32
33     watches[j++] = watches[i++]
```

Listing 4.3: Sperrbasierte, grobgranulare Bearbeitung der Klauseln

```
34      if (value(C[0]) == undef)
35          lock var(C[0])
36          if (value(C[0]) == undef)
37              atomic_add_and_fetch(&openProps, 1)
38              lock global
39              uncheckedEnqueue(other, Referenz auf C) // Fall 4
40              unlock global, var(C[0]), C
41              continue;
42          unlock var(C[0])
43
44      if (value(C[0]) == falsch)
45          Setzte confl atomar so, dass confl auf Klausel C verweist // Fall 3
46          while (i < watches[p].size()) watches[j++] = watches[i++]
47
48       // Fall 1 und Fall 3
49      unlock C
50      NextClause:
51  watches[p].shrink(i − j)
```

Listing 4.4: Sperrbasierte, grobgranulare Bearbeitung der Klauseln (Fortsetzung)

Noch bevor die Literale C[0] und C[1] vertauscht werden, erwirbt der bearbeitende Thread die Klauselsperre. Die Sperre wird solange gehalten, bis die Klausel bearbeitet wurde. Durch diese Maßnahme werden Threads, die die gleiche Klausel bearbeiten, sequentialisiert.

Literalsperren werden nur erworben wenn sie unbelegt sind. Ist dies der Fall, so wird die Sperre erworben und in einer anschließend Prüfung sicher gestellt, dass sich die Belegung des Literals in der Zwischenzeit nicht geändert hat bzw. die Änderung mit dem Fall verträglich ist (*Double Checked Locking*). Bei Eintreten des Falles 1 ist C[0] mit *wahr* belegt, daher wird in diesem Fall keine Literalsperre angefordert. Fall 2 führt die erläuterte Fallunterscheidung danach durch, ob eine Sperre erworben werden muss. Dies ist dann der Fall, wenn das Literal C[k] unbelegt ist. Da das Literal C[k] zwischen Prüfung und Zuteilung der Sperre belegt worden sein kann, wird dessen Belegung nochmals geprüft. In diesem Fall ist es zulässig, wenn das Literal in der Zwischenzeit mit *wahr* belegt worden ist. Anschließend wird die Sperre für die Klauselliste erworben, da mehrere Prozesse gleichzeitig darauf zugreifen könnten. Derjenige Thread, der die Sperre besitzt, hat das exklusive Schreibrecht und darf weitere Klauselreferenzen an die Liste anfügen. Nach dem Tausch der Literale und dem Hinzufügen der Klauselreferenz werden

alle erworbenen Sperren wieder freigegeben. Wurde das Literal nach Anforderung der Sperre mit *falsch* belegt, so wird die Suche am nächsten Index fortgesetzt. War das Literal C[k] bereits bei der ersten Prüfung mit *wahr* belegt, so ist keine Literalsperre erforderlich, da sich einmal belegte Literale innerhalb der Unit-Propagation nicht mehr ändern. Entsprechend wird in diesem Fall nur die Sperre für die Klauselliste erworben.

Sind weder Fall 1 noch Fall 2 eingetreten, so wird zunächst geprüft, ob eine Einheitsklausel (Fall 4) vorliegt. Die Reihenfolge dieser Prüfung stellt sicher, dass anschließend nur noch Fall 3 (Konflikt) und Fall 1 (zweites überwachtes Literal ist *wahr*) möglich sind. Eine andere Anordnung der Prüfungen, bei der das Eintreten von Fall 4 zuletzt überprüft wird, müsste innerhalb des Falles 4, nochmals das Eintreten der Fälle 1 und 3 prüfen (vgl. Abschnitt 4.2.1).

Ist Fall 4 eingetreten, so wird die Anzahl der noch zu bearbeitenden Belegungen inkrementiert, das Literal belegt und in die Propagation-Queue eingefügt. Diese Änderungen werden über eine globale Sperre sequentialisiert. Falls ein Konflikt auftritt, wird die gemeinsame Variable confl atomar gesetzt. Ist weder Fall 4 noch Fall 3 eingetreten, so wurde das Literal C[0] in der Zwischenzeit mit *wahr* belegt (Fall 1). In diesem Falle ist außer der Freigabe der Klauselsperre, genauso wie in Fall 1, nichts zu tun.

Anmerkungen

Threads, die die gleiche Klausel bearbeiten, werden durch die Klauselsperre sequentialisiert. Dadurch werden Probleme, die durch die Klauselvolatilität hervorgerufen werden, ausgeschlossen. Die spontane Belegung von Literalen wird durch den Erwerb von Sperren ausgeschlossen. Da alle Sperranforderungen entweder gemäß der Ordnung $Klauselsperre < Literalesperre < Klauselllistensperre$ oder der Ordnung $Klauselsperre < Globalsperre$ erfolgen, ist der Algorithmus frei von Deadlocks [43], da keine zyklische Abhängigkeiten auftreten können.

Eine Race Condition beim Anfügen an die Klauselliste in Fall 2 kann nicht auftreten, da Klauselreferenzen nie an Listen angefügt werden, die simultan von einem Thread bearbeitet werden. Dies sieht man wie folgt: Parallele Instanzen der Unit-Propagation arbeiten nur auf Klauselllisten watches[l], deren Indexliteral l mit *wahr* belegt ist. In Fall 2 ist das neu gefundene Literal C[1] entweder mit *undef* oder mit *wahr* belegt. In der Folge wird an die Liste watches[¬C[1]] angefügt. Da ¬C[1] folglich entweder *undef* oder *falsch* und damit verschieden von *wahr* ist, wird keine Klausel an eine simultan bearbeitete Liste hinzugefügt. Falls C[1]

unbelegt ist, wird über das Halten der Sperre sicher gestellt, dass sich die Belegung, bis nach dem Anfügen an die Liste, nicht ändert.

Verzichtet man in Fall 2 auf das Double-Checked-Locking und fordert die Sperre für var(C[k]) unabhängig von ihrer Belegung an, so kann man auf den Erwerb der Klauselllistensperre verzichten. Dies liegt daran, dass alle Codestellen, die schreibend auf die Klauselliste zugreifen, bereits die Sperre für var(C[k]) halten. Auf die Klauselsperren kann man vollständig verzichten, indem man die überwachten Literale nicht am Anfang der Klausel speichert, sondern zu jeder Klausel zwei Indizes einführt, die die Position der überwachten Literale in der Klausel angeben. Die Indizes dürfen nur atomar geändert werden. Falls eine atomare Tauschoperation fehl schlägt, muss die Fallbestimmung erneut ausgeführt werden.

Ob Fall 1 eingetreten ist, kann auch mit (value(C[0]) == true || value(C[1]) == true) ohne vorheriges Tauschen überprüft werden. Damit ließe sich die Klauselsperre und das Vertauschen der Literale noch hinter Fall 1 verschieben. Würde die Maßnahme zu einer Beschleunigung führen, so wäre dies aber auf algorithmische Verbesserungen (die auch im sequentiellen Fall durchgeführt werden können), nicht jedoch auf die Parallelisierung an sich zurückzuführen.

4.3.2. Feingranulare Bearbeitung der Klausellisten

Im feingranularen Ansatz bearbeitet ein Task statt der gesamten Klauselliste nur Blöcke fester Größe. Der eigentliche Algorithmus zur Klauselbearbeitung ändert sich hierbei nur geringfügig. Die meisten Änderungen betreffen die Vergabe von Tasks bzw. Blöcken. Folgende Erweiterungen wurden am grobgranularen Ansatz vorgenommen:

Taskverwaltung Um die blockweise Taskvergabe zu ermöglichen, wurde das Array idx[] eingeführt. Das Array gibt für ein Literal l den Startindex des nächsten unvergebenen Blocks an. Die um das Array idx[] erweiterte Taskvergabe ist in Listing 4.5 dargestellt. Ein Thread versucht nun, einen Block derjenigen Klauselliste zu bekommen, auf die seine private Variable head_i zeigt. Dazu setzt jeder Thread seine private Variable head_i zunächst auf den Wert der gemeinsamen Variablen head. Anschließend versucht jeder Thread einen Block der Klauselliste durch atomare Inkrementierung der Variablen idx[(var(l)] zu bekommen, wobei l das Literal ist, auf das head_i zeigt.

Ist die Liste leer, so wird die Anzahl der noch zu bearbeitenden Belegungen (openProps) nur von demjenigen Thread dekrementiert, der den ersten Block der Liste erhalten hat.

Danach beendet der Thread die Bearbeitung des Literals l, erhöht im nächsten Durchlauf die Variable head_i und führt anschließend die Prüfung durch, ob head_i bereits auf ein gültiges Literal zeigt.

Erhält der Thread einen Block der Liste ($\mathsf{idx}_i < k$), so bearbeitet er nur die Klauseln des ihm zugeordneten Blocks. Endet die Bearbeitung des letzten Blocks (dies muss nicht der zuletzt vergebene sein), so wird die Anzahl an offenen Belegungen (openProps) atomar dekrementiert. Das Löschen kann beim feingranularen Ansatz nicht mehr direkt durch überschreibendes Kopieren erfolgen, da nicht bekannt ist, ob Klauseln anderer Blöcke bereits bearbeitet wurden. Daher erfolgt das Löschen in einem extra Schritt, der Kompaktifizierung genannt wird. Hierbei wird nochmals über die gesamte Klauselliste iteriert und alle als gelöscht markierten Klauseln überschrieben bzw. durch Verkleinerung des Arrays gelöscht.

Tritt für einen Thread der else-Fall ein, so werden bereits alle Blöcke des Literals l bearbeitet. Daraufhin inkrementiert er die private Variable head_i und fährt mit der Prüfung fort, ob head_i auf ein gültiges Literal zeigt.

Bearbeitung der Blöcke Der Algorithmus zur Bearbeitung der Klauseln (Listing 4.3 und 4.4) wurden nur geringfügig wie folgt verändert. Zum einen iteriert die äußerste for-Schleife nicht mehr über alle Klauseln der Liste, sondern nur über die des Blocks. Das Löschen durch Kopieren von Klauseln könnte daher nur innerhalb eines Blocks erfolgen. Hierbei lassen sich durch gelöschte Klauseln entstandene Löcher nicht vermeiden. Daher werden Klauseln in Fall 2 zunächst als gelöscht markiert und erst nachdem alle Blöcke der Klauselliste bearbeitet wurden, in einem Durchlauf über die gesamte Klauselliste gelöscht. Im Konfliktfall wird die Blockbearbeitung sofort abgebrochen, da alle verbleibenden Klauseln des Blocks automatisch als nicht gelöscht markiert sind und daher bei der abschließenden Kompaktifizierung kopiert werden.

```
1  Thread i :
2
3  int head_i = head;
4  do
5      if (head_i < trail.size())
6          Lit l = trail[head_i]
7          Sei k die Anzahl der Klauseln von watches[l]
8          Sei b = ⌈k / CHUNK_SIZE⌉ die Anzahl der Blöcke von watches[l]
9
10         while (true)
11             idx_i = atomic_fetch_and_add(&idx[var(l)], CHUNK_SIZE);
12             if (idx_i == 0 && k == 0)
13                 atomic_sub_and_fetch(&openProps, 1);
14                 break
15             else if(idx_i < k)
16                 bearbeite den Block watches[l][idx_i] bis watches[l][idx_i + CHUNK_SIZE − 1]
17                 if (atomic_add_and_fetch(&processedChunks[l], 1) == b)
18                         atomic_sub_and_fetch(&openProps, 1);
19                         kompaktifiziere die Klauselliste watches[l]
20                         break
21             else
22                 head_i++
23                 break
24 while (openProps != 0 && kein Konflikt aufgetreten)
```

Listing 4.5: Task-Verwaltung des feingranularen Ansatzes

4.4. Sperrfreie Lösungsansätze

Die Grundidee des sperrfreien Ansatzes besteht darin, dass nicht alle parallelen Verschränkungen, die zu einem Problem führen können, ausgeschlossen werden. Stattdessen beschränkt man sich darauf, problematische Ausführungen zu erkennen und erneut zu berechnen. Durch den Verzicht auf Sperren vermeidet man, dass parallele Threads aufgrund des Sperrerwerbs blockieren und so die vorhandene Parallelität herabsetzen (*Lock Contention*).

Bei den sperrfreien Ansätzen werden ebenfalls so viele Threads benutzt, wie dem Programm an Prozessoren zur Verfügung gestellt werden, mindestens jedoch zwei. Der Kontroll-Thread T_1 steuert analog zum sperrbasierten Ansatz die Threads $T_2, \ldots, T_n$. Im sperrfreien Ansatz sind aber nicht mehr alle Threads gleichberechtigt. Die Threads $T_2, \ldots, T_n$ dürfen gemeinsam benutzte Datenstrukturen nur lesen, nicht aber verändern. Der Kontroll-Thread T_1 ist der einzige Thread der gemeinsame Datenstrukturen manipuliert. Mit den in Abschnitt 4.2 aufgeführten Problemen wird wie folgt umgegangen:

Vom Problem der parallel durchgeführten Belegungen sind nur die Threads $T_2, \ldots, T_n$ betroffen, nicht jedoch T_1, da dieser der einzige Thread ist, der Änderungen an gemeinsamen Datenstrukturen durchführen darf. Die Berechnung wird in zwei Phasen aufgeteilt: Zunächst wird von den lesenden Threads, wie im sequentiellen Algorithmus ermittelt, welcher der möglichen Fälle für die bearbeitete Klausel vorliegt (z. B. neu überwachtes Literal gefunden). Die Konsequenzen dieser Berechnung werden jedoch nicht angewendet (z. B. Tausch der Literale). Stattdessen wird in einem eigenen Speicherbereich mit einigen Zusatzinformationen gespeichert, welcher Fall eingetreten ist. In der zweiten Phase verarbeitet der Kontrollthread T_1 die berechneten Ergebnisse und validiert, dass der berechnete Fall auch mit den in der Zwischenzeit durchgeführten Änderungen eintritt. Da T_1 der einzige Thread ist, der Änderungen an gemeinsamen Datenstrukturen durchführen darf, ist sichergestellt, dass zwischen Validierung und Anwendung der Konsequenzen keine Literalbelegung erfolgt.

Aus diesem Schema ergibt sich, dass der Aufwand zur Validierung im Vergleich zur Berechnung verhältnismäßig klein sein sollte. Wenn die Validierung eines Ergebnis fehlschlägt, so wird die Klausel nochmals von einem der lesenden Threads bearbeitet und der Vorgang wiederholt sich. Die maximale Anzahl an Neuberechnungen ist durch die Anzahl der Literale einer Klausel beschränkt.

Von der Klauselvolatilität betroffen sind ebenfalls nur die Threads $T_2, \ldots, T_n$. Der für eine Klausel ermittelten Fall (z. B. neu überwachtes Literal gefunden), kann durch Veränderungen der Klausel invalidiert werden (z. B. Tausch des so gefunden Literals). Auch dies wird während der Validierungsphase erkannt.

Die Inkonsistenz gemeinsamer Datenstrukturen wird vermieden, indem der schreibende Zugriff auf gemeinsame Datenstrukturen nur aus je einem Thread erlaubt ist. Daher sind zur Synchronisation keine Sperren erforderlich.

Fasst man die Bearbeitung einer Klausel als Transaktion auf, so lässt sich der Ansatz auch mit den Begriffen des Transaktionalen Speichers beschreiben [22]. Kommt es in diesem Modell bei einer Transaktion zu einer unzulässigen Überschneidung, so muss diese wiederholt und ihre Änderungen rückgängig gemacht werden. Dabei gibt es zwei Optionen: werden die Änderungen der Transaktion direkt am Objekt durchgeführt, so bedarf es einer Sicherungskopie des Originals (*undo log*). Werden die Änderungen dagegen in einem privaten Speicherbereich durchgeführt, so ist nichts weiter zu tun, als die Berechnung neu zu starten. Der erste Ansatz hat den Nachteil, dass Transaktionen die auf den Wert einer abgebrochenen Transaktion zugegriffen haben, ebenfalls zurück gesetzt werden müssen. Dies kann zu kaskadierendem Rücksetzten von Transaktionen führen. Daher verwendet der sperrfreie Ansatz einen privaten Speicherbereich, in dem eine Transaktion ihre Änderungen puffert (*buffered updates*). Erst wenn die Transaktion erfolgreich beendet wird, werden die Daten aus dem privaten Speicher übernommen. Weiterhin unterscheidet man beim Transaktionalen Speicher, wie Konflikte erkannt (*conflict detection*) und behandelt werden *(conflict resolution)*. Eine Variante prüft während der Transaktion ob ein Konflikt aufgetreten ist (*eager conflict detection*), eine andere erst am Ende (*lazy conflict detection*). Da die Transaktionen im sperrfreien Ansatz nur aus wenigen Operationen bestehen, wird die Konfliktprüfung erst am Ende einer Transaktion durchgeführt. Wenn ein Konflikt auftritt, werden die Änderungen der Transaktion verworfen und diese neu gestartet (*choice*).

4.4.1. Grobgranulare Bearbeitung der Klauselllisten

Zunächst wird die Organisation der Threads anhand von Abbildung 4.4 erläutert. Danach wird auf die Details der Implementierung eingegangen. Abbildung 4.4 zeigt den Kontroll-Thread T_1, der das Schreibrecht für die gemeinsame Datenstrukturen besitzt, sowie die Threads T_2 und T_3, die nur private Speicherbereiche manipulieren. Die Pfeile geben dabei den Informationsfluss von Klauseln bzw. Nachrichten an. Jedem Thread $T_{i,i\geq 2}$ sind drei Ringpuffer zugeordnet. Ein

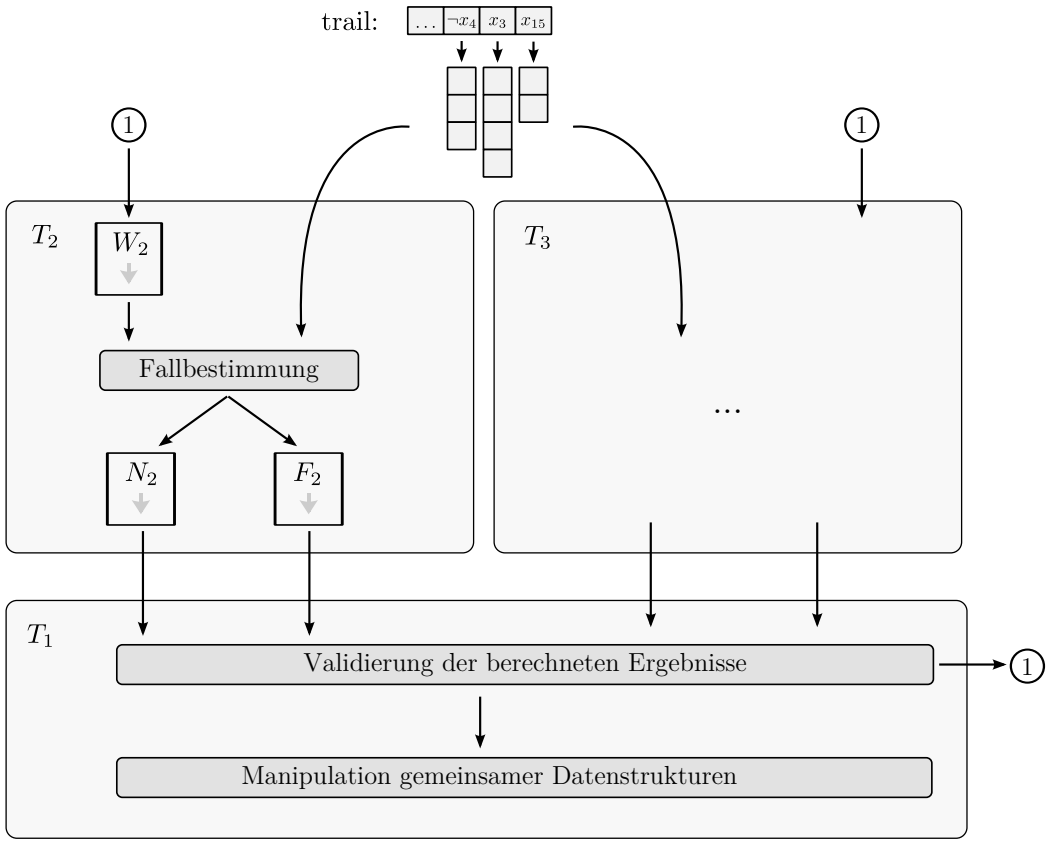


Abbildung 4.4.: Architektur des sperrfreien Ansatzes

Thread $T_{i,i\geq 2}$ greift dabei lesend auf den Puffer W_i und schreibend auf den Puffer N_i und F_i zu. Thread T_1 liest von den Puffern $N_{i,i\geq 2}$ und $F_{i,i\geq 2}$ und beschreibt die Puffer $T_{i,i\geq 2}$. Damit wird jeder Ringpuffer gleichzeitig von einem Thread gelesen und von einem Thread beschrieben (*single reader, single writer*). Alle Ringpuffer speichern Steuernachrichten oder das Ergebnis der Berechnung einer Klausel. Das Ergebnis einer Klausel setzt sich aus der Klauselreferenz, dem eingetretenen Fall und weiteren Metadaten zusammen (z. B. bis zu welchem Index bereits nach einem neuen Literal gesucht wurde). Die Implementierung des Ringpuffers wird in Anhang C dargestellt.

In den Puffern W_i werden Ergebnisse gespeichert, die aufgrund einer fehlgeschlagenen Validierung neu berechnet werden müssen. In den Puffern N_i werden neben Steuernachrichten, Ergebnisse des Falles 2 gespeichert (neu überwachtes Literal gefunden). Die Puffer F_i speichern diejenigen Klauseln, für die weder Fall 1 noch Fall 2 eingetreten ist. Welcher Fall vorliegt wird jedoch erst von T_1 ermittelt, da die Fallbestimmung und die Validierung gleich aufwendig sind. Die Trennung der Ergebnistypen erlaubt eine priorisierte Bearbeitung verschiedener Ergebnisse. So können z. B. die Ringpuffer F_i, die möglicherweise weitere Tasks durch Einheitsklauseln erzeugen, priorisiert bearbeitet werden. Klauseln, die von T_1 nicht validiert werden konnten, werden zur wiederholten Berechnung im Rundlaufverfahren (*round robin*) in die Puffer W_i geschrieben.

Taskverwaltung

Die Threads $T_2, \ldots, T_n$ werden vom Kontroll-Thread, der auch den Rest des CDCL-Algorithmus ausführt, über das Setzen einer gemeinsamen Variablen aufgeweckt. Anschließend bearbeiten sie, bis zum Eintritt des Abbruchkriteriums oder dem Auftreten eines Konflikts, die vorhandenen Tasks. Die Taskvergabe erfolgt analog zur grobgranularen, sperrbasierten Taskvergabe durch atomare Manipulation von Indexvariablen (siehe Abschnitt 4.3.1). Diese wird jedoch noch um Tasks erweitert, für die eine Neuberechnung durchgeführt werden muss. Die Threads $T_2, \ldots, T_n$ bearbeiten daher alternierend die Klauselllisten sowie die Klauseln des Puffers W_i.

Bearbeitung von Tasks durch die Threads $T_2, \ldots, T_n$

Für einen Thread $T_{i,i\geq 2}$, der eine Belegung bearbeitet, ergibt sich der folgende Ablauf: Für jede Klausel der Klauselliste wird eine Fallbestimmung, wie in Listing 4.6 gezeigt, durchgeführt. Tritt Fall 2 ein, so wird das Ergebnis in Puffer N_i gespeichert. Tritt weder Fall 1 noch Fall 2 ein, so wird das Ergebnis im Puffer F_i gespeichert. Fall 1 (zweites überwachtes Literal mit *wahr* belegt) nimmt aus den folgenden Gründen eine Sonderrolle ein: Zum einen muss sein Eintreten nicht validiert werden, da einmal belegte Literale für die Dauer der Unit-Propagation belegt bleiben. Zum anderen sind mit seinem Eintreten keine Aktionen verbunden, daher kann Fall 1 abschließend durch Thread T_i bearbeitet werden. Zwar kann es durch das parallele Vertauschen des anderen überwachten Literals durch T_1 dazu kommen, dass das alte Literal gelesen wird, dies hat jedoch keinen Einfluss auf die Korrektheit. Dies führt lediglich dazu, dass die in Abschnitt 3.1 beschriebene Optimierung nicht angewendet wird. Bei diesem Vorgehen, kann der am häufigsten eintretenden Fall 1 (vgl. Abschnitt 3.3) genauso effizient wie im sequentiellen Verfahren behandelt werden.

Im Puffer W_i befinden sich Klauseln, für die der Fall 2 festgestellt, aber nicht erfolgreich validiert werden konnte. Der Thread T_i setzt die Suche nach einem neuen überwachten Literal an der Stelle in der Klausel fort, an der beim letzten Mal aufgehört wurde. Abhängig davon, ob ein neu überwachtes Literal gefunden wird, wird das Ergebnis in Puffer N_i bzw. F_i geschrieben. Wiederholte Berechnungen für andere Fälle sind nicht notwendig, da die einzige weitere Entscheidung, die von einem unbelegten Literal abhängt (Fall 4, Einheitsklausel), abschließend von T_1 behandelt wird.

Validierung und Anwendung der Ergebnisse durch T_1

Die Validierung prüft, ob Änderungen der Klausel oder Belegungen, die nach der Fallermittlung der Klausel durchgeführt wurden, dazu geführt haben, dass inzwischen ein anderer als der ermittelte Fall vorliegt. Hat einer der lesenden Threads ein neu überwachtes Literal gefunden, so könnte dies in der Zwischenzeit mit *falsch* belegt oder durch ein anderes Literal ersetzt worden sein (durch einen Tausch des zweiten überwachten Literals). Zur Validierung reicht es daher, nur die Belegung des potentiell neu überwachten Literals zu prüfen. Ist dessen Belegung nach wie vor ungleich *falsch*, so wird es getauscht, die Klauselreferenz als gelöscht markiert und eine neue Klauselreferenz für das neue Literal hinzugefügt. Ist das Literal dagegen mit

```
1  Fallbestimmung für die Klauselliste von p durch Thread_i :
2
3  int pending = 0;
4  for (int i = 0, j = 0; i < watches[p].size(); i++)
5
6        if(watches[p][i] ist als gelöscht markiert)
7              continue;
8
9        watches[p][j] = watches[p][i]
10       Sei C die durch watches[p][j] referenzierte Klausel
11
12       Sei idx der Index an dem ¬p steht(idx = 0 oder idx = 1)
13       if (value(c[idx]) == wahr)
14             goto NextWatcher;
15
16       pending++;
17       for (int k = 2; k < c.size(); k++)
18             if (value (c[k]) != falsch)
19                   wr = ClauseResult::NewWatch(p, j, cr, k);
20                   N_i.enqueue(wr);
21                   goto NextWatcher;
22
23       wr = WatcherResult::NewWatchFailed(p, j, cr);
24       F_i.enqueue(wr);
25
26       NextWatcher: j++;
27
28 ClauseResult wr = ClauseResult::AllClauses(p, ws.size() − pending);
29 N_i.enqueue(wr);
30 watches[p].shrink(i − j);
```

Listing 4.6: Fallbestimmung des sperrfreien Ansatzes

falsch belegt, so wird die Klausel zur Neuberechnung in einer der Puffer W_i eingefügt. Für eine Klausel kann der Fall 2 maximal so oft im Laufe eines Unit-Propagation Aufrufs auftreten, wie sie Literale hat. Es verbleiben die Fälle, in denen für eine Klausel weder Fall 1 noch Fall 2 eingetreten ist. In diesem Fall sind alle Literale C[0],...,C[C.size() -1] sowie mindestens eines der überwachten Literale mit *falsch* belegt. Die Validierung besteht also aus dem Lesen der Belegung des anderen überwachten Literals. Entweder es liegt Fall 1 (zweites überwachtes Literal mit *wahr* belegt), Fall 3 (Konflikt) oder Fall 4 (Einheitsklausel) vor.

Kompaktifizierung und Endeerkennung

Im Gegensatz zum sequentiellen Ansatz kann nicht während der Bearbeitung einer Klausel festgestellt werden, ob sie in der Liste verbleiben soll. Die Referenz müsste genau dann gelöscht werden, wenn ein neu überwachtes Literal gefunden und erfolgreich validiert wird. Da das Warten auf das Ergebnis der Validierung die Parallelität verringern würde, wird stattdessen das folgende Schema verwendet: Der Kontrollthread markiert eine Klauselreferenz als gelöscht, wenn die Validierung des neuen Literals erfolgreich war. In jedem Durchlauf über die Klauselliste werden zuerst diejenigen Klauselreferenzen überschrieben, die im vorhergehenden Durchlauf als gelöscht markiert wurden. Das Löschen wird also solange verzögert, bis die Klauselliste erneut traversiert wird (*lazy deletion*).

Die Bearbeitung eines Threads endet, sobald alle Belegungen vollständig bearbeitet wurden oder ein Konflikt eintrat. Letzteres kann über eine gemeinsame Variable von allen Threads detektiert werden. Zur Erkennung, wann alle Belegungen vollständig bearbeitet wurden, wird das Array clausesProcessed[] eingeführt. Es speichert zu einem Literal die Anzahl seiner bereits validierten Klauseln. Die Anzahl an zu bearbeitenden Belegungen wird analog zu den sperrbasierten Ansätzen mit der Variablen openProps erfasst. Diese wird vor Durchführung der Unit-Propagation mit 1 bzw. trail.size() - head initialisiert. Die Anzahl an zu bearbeitenden Belegungen openProps wird jedes Mal dekrementiert werden, wenn clausesProcessed[l] für ein Literal l gerade der Anzahl aller Klauseln der Liste entspricht.

Durch die oben beschriebene Kompaktifizierung ist diese Größe jedoch nicht stabil. Daher wird ein weiteres Array toProcess[l] eingeführt, das die Anzahl der Klauseln vor Beginn der Kompaktifizierung enthält. Dieses wird vor Bearbeitung des Literals mit dem Wert von watches[l].size() initialisiert.

Prinzipiell müssten sowohl die lesenden Threads $T_{i,i\geq 2}$, als auch der schreibende Thread T_1 clausesProcessed[] inkrementieren. Erstere, falls Fall 1 eintritt oder eine leere Liste bearbeitet wird. Letzterer, wenn die Ergebnisse einer Klausel erfolgreich validiert werden. Da es sich hierbei um gemeinsame Daten handelt, wird das eingeführte Schema fortgesetzt und nur T_1 darf schreibend auf das Array clausesProcessed[] zugreifen. Daher wurde wie in Listing 4.6 gezeigt, eine Steuernachricht eingeführt, die an den Kontrollthread gesendet wird, um diesem mitzuteilen, wie viele Klauseln durch den lesenden Thread bearbeitet wurden. Beim Verarbeiten der Nachricht, inkrementiert der Kontrollthread den Zähler clausesProcessed[l]. Falls danach clausesProcessed[l] == toProcess[l] gilt, wird openProps dekrementiert.

4.4.2. Feingranulare Bearbeitung der Klauselllisten

Der feingranulare Ansatz erweitert den grobgranularen Ansatz, indem ein Task statt der gesamten Klauselliste nur einer festen Anzahl an Klauseln entspricht. Die Blockvergabe erfolgt analog zum sperrbasierten, feingranularen Ansatz über ein Array, das den jeweils nächsten Block angibt (vgl. Abschnitt 4.3.2).

Klauseln der Klauselliste werden gelöscht, indem sie während ihrer Bearbeitung durch den Kontroll-Thread als gelöscht markiert werden. Über ein Array wird Buch geführt, ob bereits alle Klauseln bzw. Blöcke bearbeitet wurden. Sobald der letzte Block bearbeitet wurde, erfolgt die Kompaktifizierung der Liste. Dazu wird eine Steuernachricht eingeführt, die der Thread T_1 an die lesenden Prozesse schickt. Da zu diesem Zeitpunkt nicht mehr auf die Klauselliste zugegriffen wird, können die Threads die Klausellisten ohne weitere Synchronisation kompaktifizieren.

Implementierung

Zur Implementierung aller Varianten werden POSIX *pthreads* benutzt, die bei Programmbeginn gestartet werden. Die Sperren werden explizit in Form von Integervariablen bzw. Integerarrays umgesetzt, die über atomare Operationen manipuliert werden. Möchte ein Thread eine Sperre erwerben, so versucht er in einer Schleife den betreffenden Speicherbereich, unter Verwendung einer atomaren *compare and swap* Operation, mit 1 zu belegen. Er besitzt die Sperre sobald die Operation erfolgreich durchgeführt wurde. Die Freigabe erfolgt analog durch das Zurücksetzen

der Variablen auf 0. Dieses Vorgehen stellt sicher, dass der Code beim Erwerb einer Sperre keine Systemaufrufe benutzt, die eine höhere Latenz als dieses Vorgehen haben.

4.5. Partitionierungsansatz

In [27] wird ein Shared-Memory Parallelisierungsansatz der Unit-Propagation vorgestellt, der auf der Partitionierung der Klauseldatenbank basiert. Dabei werden die Klauseln der Formel im Round-Robin-Verfahren einer Partition zugewiesen. Jeder Thread führt die Unit-Propagation nur auf den Klauseln seiner Partition durch. Sobald ein Thread fertig ist, wartet dieser auf die anderen und prüft, ob ein Konflikt oder weitere Literale von anderen Threads belegt wurden (da diese auch in Klauseln der eigenen Partition vorkommen können). Ist letzteres der Fall, wird erneut propagiert bis sich keine Änderungen mehr ergeben oder ein Konflikt auftritt. Die Berechnung besteht also aus einem Wechsel von Unit-Propagation und Synchronisation. Der Ansatz erzielt bei der Ausführung auf einem Dual-Core Prozessor, eine Beschleunigung des gesamten Solving-Prozesses[1] von 1,06 und auf einem Quad-Core eine Beschleunigung von 1,11. Da nur geringe Beschleunigungen erzielt werden können, werden die folgenden Erweiterungen des Schemas zur Diskussion gestellt.

Der Vorschlag besteht aus einer Kombination von zwei Maßnahmen:

- Es wird vorgeschlagen, dass die Struktur der Formeln bei der Partitionierung berücksichtigt wird. Als Partitionierungskriterium wird in Anlehnung an [5] der Klauselgraph vorgeschlagen. Der Klauselgraph besitzt als Knoten die Klauseln der Formel und verbindet je zwei Knoten, wenn die Klauseln eine gemeinsame Variable besitzen. Die Struktur von SAT-Formeln, insbesondere wie sich diese in Komponenten zerlegen lassen, wurde in [5] untersucht.

  Betrachtet man den Extremfall, in dem eine Partition des Klauselgraphen keine ausgehenden Kanten in eine andere Partition besitzt, so kann die Unit-Propagation für diese Partition ausgeführt werden ohne dass dies zu Iterationen einer anderen Partition führt. Besitzt eine Partition dagegen Kanten in eine andere Partition, so versucht der Ansatz die Anzahl der synchronisationsbedingten Iterationen zu reduzieren. Ein Nachteil des Ansatzes ist, dass Einheitsklauseln dazu tendieren, nur Literale der eignen Partition zu belegen. Würde z. B. ein Entscheidungsliteral in einer Partition des Klauselgraphen

[1] nicht nur der Unit-Propagation

gesetzt, die keine Kanten in andere Partitionen hat, so kann dies nur zur implizierten Literalen innerhalb der Partition führen. Damit wäre auch keine parallele Bearbeitung der einzelnen Partitionen möglich.

- Daher wird weiter vorgeschlagen, die Heuristiken zur Variablenwahl so anzupassen, dass bevorzugt Variablen gewählt werden, die in Klauseln mehrerer Partitionen vorkommen.

  Alternativ dazu, ließe sich auch der Solving-Prozess so modifizieren, dass pro Entscheidungslevel nicht nur ein Literal, sondern mehrere Literale aus verschiedenen Partitionen gleichzeitig belegt werden. Damit wäre aber auch eine vollständige Anpassung der Konfliktanalyse und des Lernschemas notwendig.

Der erste Vorschlag zielt darauf ab, die Kommunikation und die durch sie verursachten Iterationen zwischen den Partitionen zu minimieren. Dies kann jedoch dazu führen, dass nur selten Literale in mehreren Partitionen gleichzeitig propagiert werden können. Das Problem versucht der zweiter Vorschlag durch Anpassung der Heuristik bzw. des gesamten Suchprozesses zu vermeiden. Die Änderungen haben das Ziel, dass die Unit-Propagation in möglichst vielen Partitionen gleichzeitig und unabhängig voneinander ablaufen kann. Insbesondere der Vorschlag mehrere Literale auf einmal zu belegen, zieht aber weitreichende Änderungen nach sich. Die vorgeschlagene Heuristik wählt Literale, die günstig für die parallele Ausführung der Unit-Propagation sind. In wie weit sich diese Literalwahl orthogonal zu den bisherigen Heuristiken verhält ist unklar.

5. Experimentelle Bewertung

In diesem Kapitel wird überprüft, wie sich die vorgeschlagenen Lösungen auf die Laufzeit[1] eines Testformelsatzes auswirken. Hierzu werden die Laufzeiten der Verfahren anhand der folgenden Kennzahlen des parallelen Rechnens verglichen.

$$\text{Beschleunigung} = \frac{\text{Laufzeit}_{\text{sequentiell}}}{\text{Laufzeit}_{\text{parallel}}} \qquad \text{Effizienz} = \frac{\text{Beschleunigung}}{\text{Anzahl Rechenkerne}}$$

Der durch die Parallelisierung eingeführte Mehraufwand (*Overhead*) lässt sich nach [18] in drei Kategorien einteilen: Excess Computation, Leerlauf und Kommunikation. Excess Computation ist algorithmischer Mehraufwand, der von der Parallelisierung verursacht wird (hier z. B. die Taskverwaltung, Sperrerwerb). Leerlauf ist diejenige Zeit, in der ein Prozess keinen Fortschritt im Algorithmus macht (z. B. durch Warten auf eine Sperre). Im folgenden wird nur zwischen Excess Computation und Leerlauf unterschieden, da sich die Kommunikation als Kombination von Excess Computation und Leerlauf auffassen lässt. Ferner wird der Sperrerwerb durch das beständige Abfragen eines Variablenwertes zum Leerlauf gezählt, da hierbei kein Fortschritt im Algorithmus erfolgt. Die gemessenen Laufzeiten können auch auf der gleichen Eingabe schwanken, da die Parallelisierung eine Randomisierung des Suchprozesses bewirkt. Dies kann sowohl zu einer Verkürzung als auch zu einer Verlängerung der Laufzeiten führen. Um ihren Einfluss auf das Ergebnis zu erfassen, werden pro Formel mehrere Wiederholungen ausgeführt und die Streuung der Laufzeit ermittelt.

5.1. Testformelsatz

Der Testformelsatz besteht aus 20 unerfüllbaren Formeln, die aus den Wettbewerben *SAT Race* und *SAT-Competition* ausgewählt wurden. Dazu wurden diese mit Minisat 2.2 ohne

[1] Mit Laufzeit ist im folgenden immer die zwischen Start und Ende des Programms verstrichene Zeit gemeint (wall clock time).

Verwendung der cache-optimierenden Blocker, bearbeitet. Es wurden nur *unerfüllbare* Formeln mit einer Gesamtlaufzeit von mindestens zwei und maximal zehn Minuten als Testformelkandidaten berücksichtigt. Die Beschränkung auf unerfüllbare Formeln soll die Wahrscheinlichkeit senken, dass Beschleunigungen gemessen werden, die primär durch die Randomisierung entstanden sind. Die durch die Parallelisierung verursachte Randomisierung kann dazu führen, dass bei mehreren Läufen auf der gleichen Eingabe verschiedene Suchbäume traversiert werden. Dies kann bei erfüllbaren Formeln zur Folge haben, dass sich auch die Lage von erfüllenden Belegungen im Suchbaum ändert. Unter Umständen können diese daher in der parallelisierten Version in kürzerer Zeit gefunden werden als in der sequentiellen Version. Wird eine Lösung sogar mit weniger Schritten gefunden als das sequentiellen Verfahren benötigt, so spricht man von der Beschleunigungsanomalie [19]. Der Effekt kann durch das Lernen unterschiedlicher Konfliktklauseln auch bei unerfüllbaren Formeln auftreten. Es wird jedoch erwartet, dass diese Effekte für unerfüllbare Formeln geringer sind, da der Solver hier Informationen über den gesamten Suchbaum besitzen muss und nicht nur über Teile davon.

SAT-Formeln lassen sich nach der Art ihrer Erzeugung klassifizieren[2] [15]. Dabei werden die Typen *Crafted, Application und Random* unterschieden. Formeln des Typs Random werden zufällig erzeugt, Formeln des Typs Application stammen dagegen aus industriellen Anwendungen (siehe Abschnitt 1.1). Zu Formeln des Typs Crafted gehören alle Formeln, die nicht in eine der beiden anderen Kategorien fallen. Sie sind oft mit dem Ziel entworfen, für einen SAT-Solver schwer lösbar zu sein. Da zufällig erzeugte Formeln nur eine geringe Relevanz für die Praxis besitzen, werden zur Bewertung nur Formeln des Typs Crafted und Application verwendet.

Aus den unerfüllbaren Formeln, die innerhalb der vorgegebenen Laufzeit lagen, wurde anschließend eine Teilmenge von 20 Formeln gewählt, von denen 10 vom Typ Crafted (01-10) und 10 vom Typ Application (11-20) sind. Dabei wurden diese so gewählt, dass keine einseitige Auswahl hinsichtlich der Laufzeit, der Anzahl der Variablen und der Anzahl der Klauseln entstand. Abbildung 5.1 zeigt die Verteilung der Gesamtlaufzeit der Testformeln, Anhang A die Verteilung nach der Anzahl der Klauseln und Variablen.

[2] http://www.satcompetition.org/

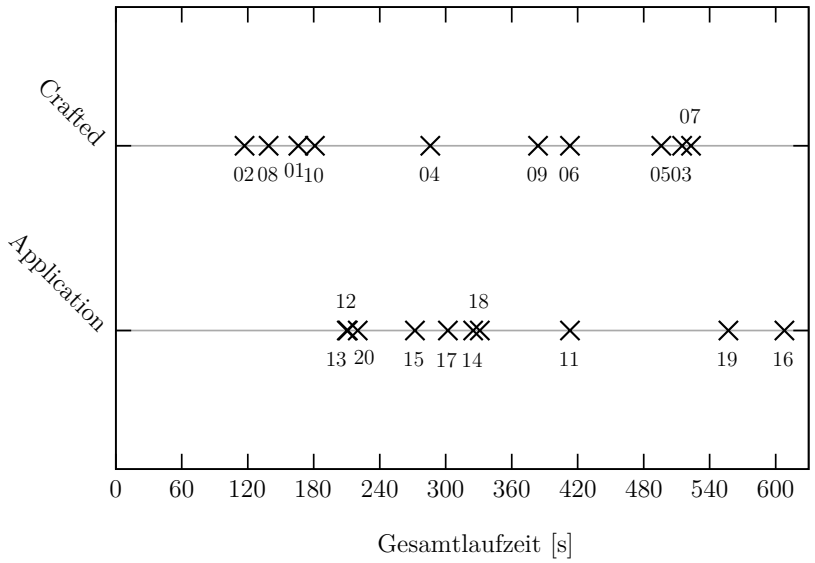


Abbildung 5.1.: Gesamtlaufzeit der Testformeln (sequentiell)

5.2. Verwendete Hard- und Software

Zum Einsatz kam ein AMD Athlon II X4 635 Prozessor mit 2,9 GHz, 512 KB Cache und 4 GB RAM, auf dem ein Ubuntu Linux 10.04 LTS läuft. Als Compiler wurde gcc in Version 4.4.3 verwendet. Kompiliert wurden die Programme mittels[3] `make r` und der Optimierungsstufe -O3. Das Makefile wurde zur Zeitmessung um die Linking Option -lrt zur Verwendung der lib-realtime erweitert.

Als zweites Testsystem, mit einer größeren Anzahl an Rechenkernen, wurde ein Rechner mit NUMA-Architektur verwendet. Er besteht aus acht Quad-Core AMD Opteron Prozessoren mit 2,3 GHz, 512 KB Cache und 280 GB Speicher. Der Rechner wird mit CentOS 5.7 betrieben und verfügt über die gcc Version 4.1.2.

Auf welchem System die Messungen durchgeführt wurden, ist jeweils angegeben. Das erste System wird mit `PC1` das zweite mit `PC2` bezeichnet.

[3] nicht mit `make rs`, da vorübergehend die Bibliothek cilk verwendet wurde, die das statische Linking nicht unterstützt

5.3. Ergebnisse

Im folgenden werden die Laufzeiten der Unit-Propagation für die verschiedenen Varianten präsentiert. Zunächst werden die Ergebnisse des sperrbasierten, anschließend die des sperrfreien Ansatzes vorgestellt. Um Einflüsse der Konfliktanalyse auszuschließen, wurde nur die Laufzeit der *Unit-Propagation* gemessen. Die gemessenen Zeiten wurden wie in Anhang B beschrieben um den Overhead, der durch die Zeitmessung selbst anfällt, bereinigt. Die Laufzeit einer Formel wird nur dargestellt, wenn alle Programmläufe innerhalb des Zeitlimits[4] von 2000 Sekunden gelöst werden konnten. Dabei werden die Ergebnisse in Bezug zur Laufzeit des sequentiellen Verfahrens (ohne Blocker) normiert. Dies hat bei den sperrbasierten Ansätzen den Vorteil, dass man den eingeführten Mehraufwand an der Laufzeit des Programms, das nur einen Rechenkern nutzt, ablesen kann. Außerdem ist sofort ersichtlich, ob eine Beschleunigung erfolgte. Die Streuung der Laufzeiten einer Formel wird als Balken dargestellt, dessen Länge der Standardabweichung[5] entspricht. Bei der Ermittlung von Kennzahlen werden nur diejenigen Formeln berücksichtigt, bei denen jeder Lauf innerhalb des Zeitlimits lag.

5.3.1. Sperrbasierter Ansatz

Grobgranularer Ansatz

Die Ergebnisse der drei Läufe jeder Formel (auf PC1) zeigt Abbildung 5.2. Insgesamt wird in der Mehrzahl der Fälle keine Beschleunigung erzielt. Die folgende Tabelle fasst dies zusammen:

# Rechenkerne	2	3	4
# in Zeit gelöster Formeln	19	18	18
# Beschleunigung > 1	1	5	6
# Beschleunigung ≤ 1	18	13	12
Mittlere Beschleunigung	0,79	0,87	0,87

[4] wall-clock time bzw. bei p Rechenkernen das p-fache der wall-clock time an CPU-Zeit

[5] Als Standardabweichung der Messwerten $x_1, \ldots x_n$ von ihrem Mittelwert $\bar{x}$ wird die Formel $stawb(x_1, \ldots, x_n) = \sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}$ verwendet.

Betrachtet man nur die Formeln, bei denen eine Beschleunigung erfolgte, wurde für zwei Rechenkerne eine Beschleunigung von 1,06 (Effizienz: 0,52), für drei Rechenkerne eine Beschleunigung von 1,2 (Effizienz 0,4) und für vier Rechenkerne eine Beschleunigung von 1,15 (Effizienz 0,29) erzielt. Die Fälle, in denen eine Beschleunigung eintrat, setzten sich zu gleichen Teilen aus den Formeltypen Crafted und Application zusammen.

Betrachtet man das sperrbasierte Programm, das von nur einem Rechenkern ausgeführt wird, so kann man den durch Excess Computation eingeführten Mehraufwand erfassen. Die für den Sperrerwerb und -freigabe eingeführten Anweisungen führen zu einer mittleren Laufzeitsteigerung von 69 %. Bei der Verwendung von mehr als einem Thread, kann der durch den Sperrerwerb verursachte Leerlauf den Mehraufwand weiter erhöhen und die parallele Effizienz des Verfahren so weiter reduzieren.

Das Testset wurde je drei Mal auf dem System PC2 für 2, 4, und 6 Rechenkerne ausgeführt. Eine weitere Erhöhung der Rechenkerne führte dazu, dass weniger Formeln innerhalb des Zeitlimits gelöst werden konnten. Außerdem erhöhte sich die Gesamtzeit der gelösten Formeln, obwohl deren Anzahl abnahm.

# Rechenkerne	1_{seq}	2	4	6
# in Zeit gelöster Formeln	20	16	14	12
Gesamtlaufzeit gelöster Formeln [s]	4335	10125	10644	11654

Feingranularer Ansatz

Der feingranulare Ansatz wurde zweifach mit den Blockgrößen 2, 8, 16 und 32 auf drei Rechenkernen ausgeführt und mit dem grobgranularen Ansatz verglichen. Die Blockgröße 2 wurde gewählt, damit auch die kurzen Klauselllisten des Formeltyps Application in mehrere Blöcke zerfallen (vgl. Abschnitt 3.4). Die Laufzeit des feingranularen Ansatzes ist mit drei Rechenkernen, ausgenommen zweier Formeln, stets größer als die des grobgranularen Ansatzes (je nach Anzahl der Rechenkerne um den Faktor 1,5 bis 2). Eine mögliche Erklärung dafür ist, dass der Mehraufwand des feingranularen Ansatz im Verhältnis zur Taskgröße zu groß ist. Die vollständige Auswertung befindet sich in Abbildung 5.3. Die Laufzeiten wurden auch hier auf die Laufzeit der sequentiellen Unit-Propagation normiert.

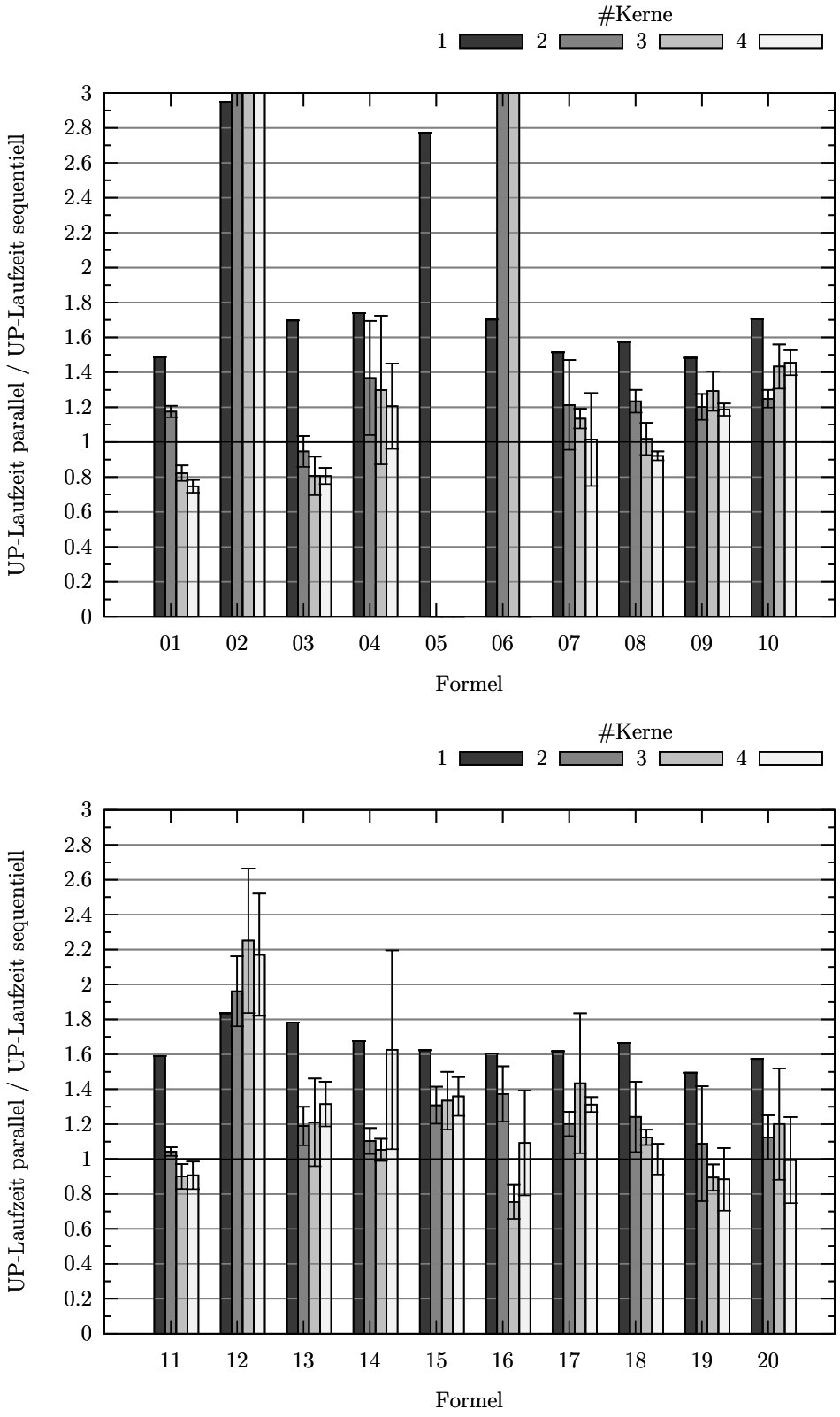


Abbildung 5.2.: Normierte Laufzeit des sperrb. grobr. Ansatzes (PC1)

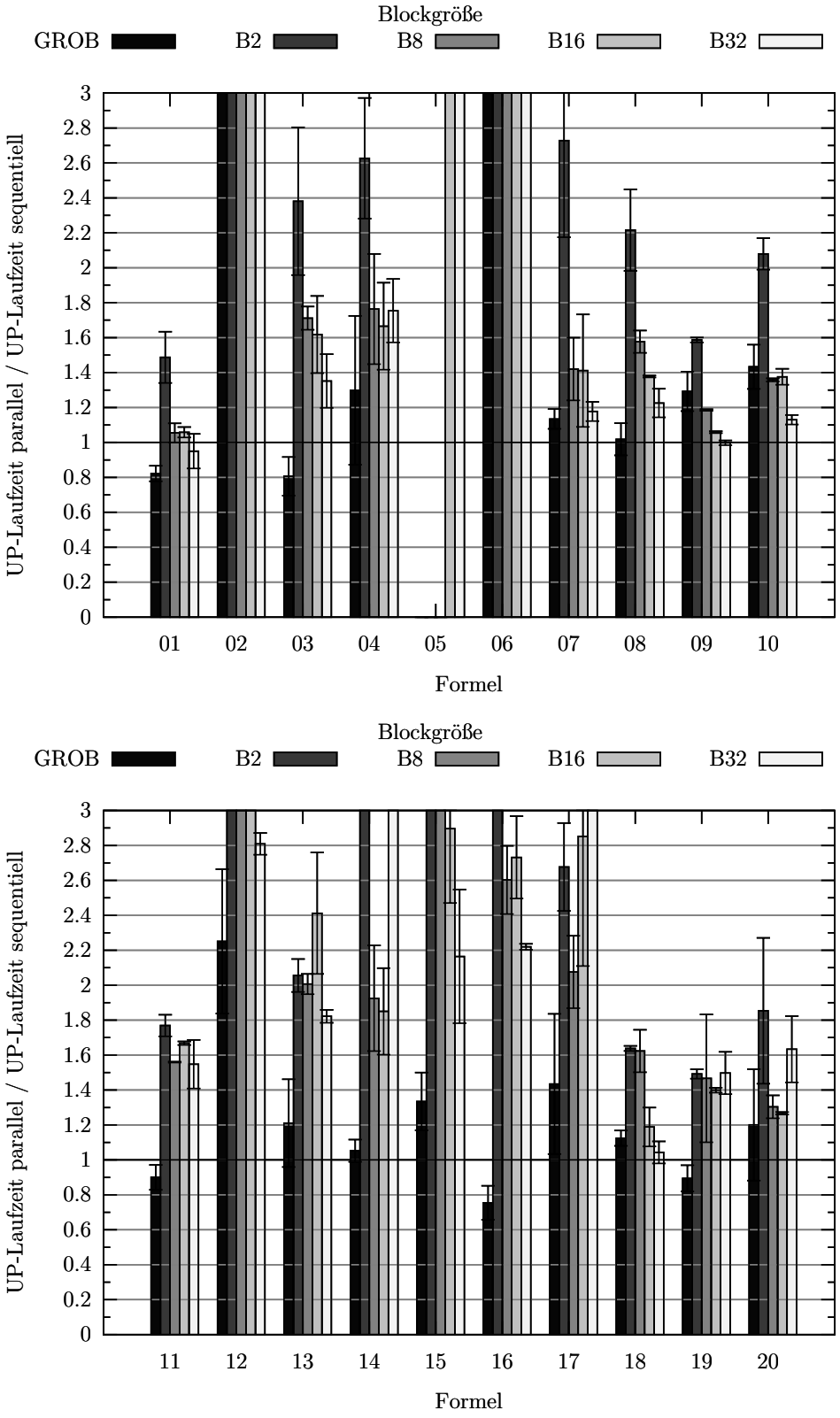


Abbildung 5.3.: Normierte Laufzeit des sperrb. feingr. Ansatzes mit drei Rechenkernen (PC1)

5.3.2. Sperrfreier Ansatz

Grobgranularer Ansatz

Beim sperrfreien Ansatz wird im Mittel bei keiner der zwanzig Formeln eine Beschleunigung gemessen. Die Gesamtlaufzeit der Formeln, die immer innerhalb des Zeitlimits gelöst werden konnten, ist in folgender Tabelle dargestellt:

# Rechenkerne	1_{seq}	2	3	4
# in Zeit gelöster Formeln	20	19	19	19
Gesamtlaufzeit gelöster Formeln [s]	5406	12802	10939	11548

Zu erkennen ist, dass die Ausführungszeit mit zwei Rechenkernen mehr als das Doppelte des sequentiellen Algorithmus beträgt. Die Verwendung von drei Rechenkernen führt zunächst zu einer Abnahme der Gesamtlaufzeit um ca. 17 %, ein weiterer Rechenkern erhöht diese jedoch wieder.

Feingranularer Ansatz

Der feingranulare Ansatz wurde ebenfalls zweifach mit den Blockgrößen 2, 8, 16 und 32 auf drei Rechenkernen ausgeführt. Im Vergleich zum grobgranularen Ansatz benötigt der feingranulare Ansatz bis auf wenige Ausnahmen stets länger. Für die Blockgröße 8 werden drei der Formeln in kürzerer Zeit gelöst als im grobgranularen Ansatz. Für jede andere Blockgröße sind es jeweils weniger. Die vollständige Auswertung zeigt Abbildung 5.5. Die Laufzeiten wurden wieder mit der Laufzeit des sequentiellen Algorithmus normiert.

5.3.3. Vergleich sperrfreier und sperrbasierter Ansatz

Abschließend wird der grobgranulare, sperrbasierte Ansatz mit dem grobgranularen, sperrfreien Ansatz vergleichen. Hierbei wird die Summe der mittleren Gesamtlaufzeit aller Formeln betrachtet, die in beiden Ansätzen innerhalb des Zeitlimits gelöst werden konnten (18 Formeln). Die kleinste Gesamtlaufzeit wird in beiden Fällen mit drei Rechenkernen erreicht. Allerdings

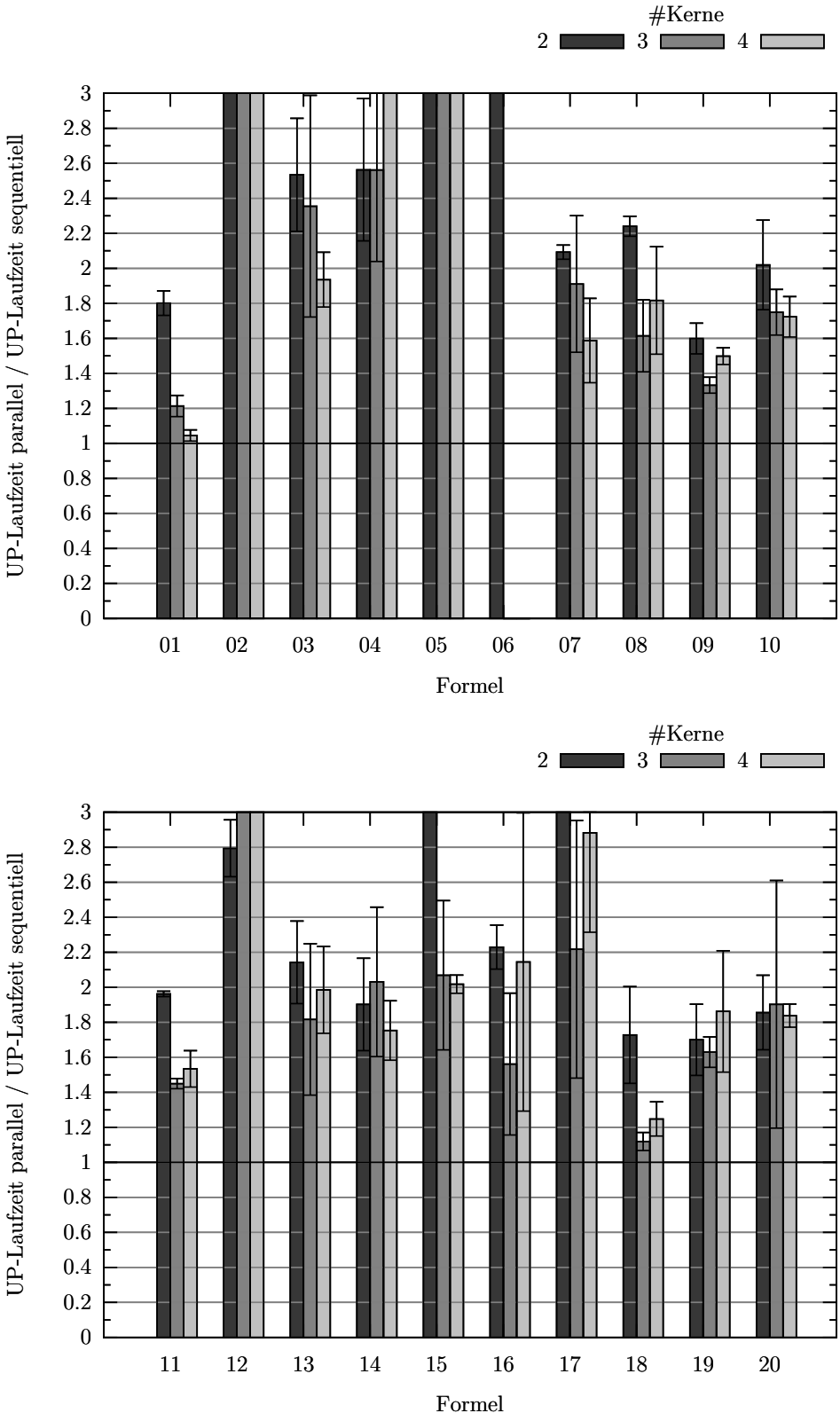
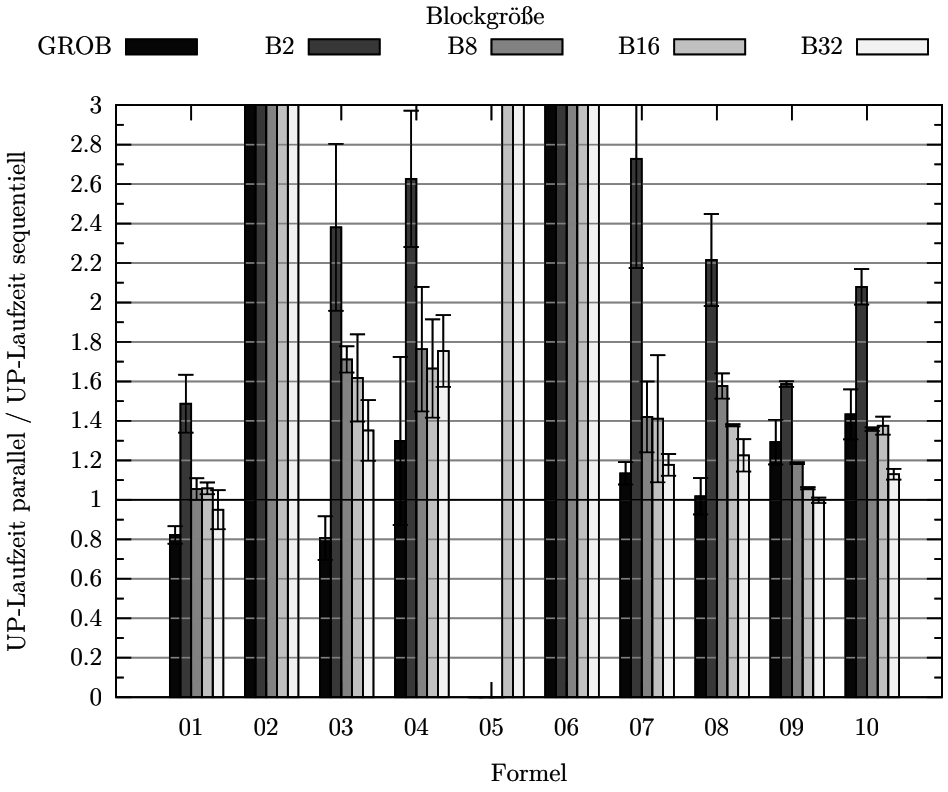


Abbildung 5.4.: Normierte Laufzeit des sperrfr. grobr. Ansatzes (PC1)



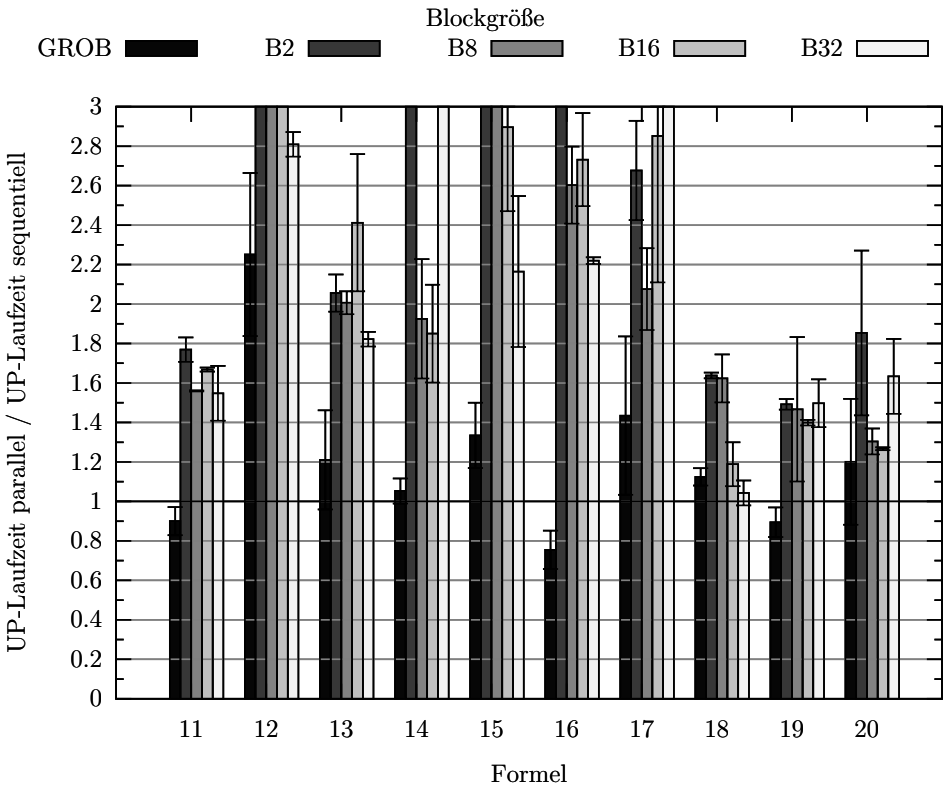


Abbildung 5.5.: Normierte Laufzeit des sperrfr. feingr. Ansatzes mit drei Rechenkernen (PC1)

# Rechenkerne		1_{seq}	1	2	3	4
Sperrbasiert	Gesamtzeit T_i	5099	8304	6433	5957	6127
	Veränderung	×	1,63	0,77	0,93	1,03
	Verhältnis T_i/T_{1seq}	1	1,63	1,26	1,17	1,20
Sperrfrei	Gesamtzeit T_i	5099	×	11617	9677	10245
	Veränderung	×	×	×	0,83	1,06
	Verhältnis T_i/T_{1seq}	1	×	2,28	1,90	2,01

Tabelle 5.1.: Skalierbarkeit der Verfahren

liegt die Laufzeit des sperrbasierten Ansatzes mit drei Rechenkernen 17 % über dem der sequentiellen Variante, die des sperrfreien sogar um 90 %.

Abschließend wird die Skalierbarkeit der Verfahren anhand von Tabelle 5.1 untersucht. Im sperrbasierten Verfahren sieht man, dass der eingeführte Code einen Mehraufwand von 63 % verursacht. Die Verwendung eines zweiten Rechenkerns reduziert die Gesamtlaufzeit zwar um 23 %, die des dritten jedoch nur noch um 7 %. Eine mögliche Erklärung hierfür ist, dass durch die Hinzunahme der weiteren Threads, mehr Leerlauf durch den Sperrerwerb entsteht (*Lock Contention*). Eine weitere Erhöhung der verwendeten Rechenkerne führt nicht zur weiteren Abnahme der Laufzeit. Im sperrfreien Ansatz führt die Verwendung von drei Rechenkernen im Vergleich zu zwei genutzten Rechenkernen zu einer Abnahme um 17 %. Eine weitere Erhöhung der benutzten Rechenkerne führt auch hier nicht zu einer weiteren Reduktion der Laufzeit.

Zusammenfassend lässt sich sagen, dass beide Verfahren nicht von mehr als drei Rechenkernen profitieren. Im sperrbasierten Ansatz liegen die Ursachen in einer Kombination von Excess Computation und Leerlauf. Ein möglicher Erklärungsansatz wäre, dass jeder weitere Thread den Wettbewerb um die Sperren erhöht und es zur Lock Contention kommt. Warum die Laufzeit des sperrfreien Ansatz bei der der Verwendung von mehr Rechenkernen nicht weiter reduziert werden kann, ließe sich damit erklären, dass der Kontrollthread ab drei benutzten Rechenkernen zum Flaschenhals wird. Diese Erklärungsansätze werden im nächsten Abschnitt stichprobenartig überprüft.

5.4. Ergebnisanalyse

Abschließend wird die Frage untersucht, warum es ab der Verwendung von drei Rechenkernen zur Stagnation der Beschleunigung kommt. Hierzu wurde das sperrbasierte und das sperrfreie Verfahren, stichprobenartig anhand von Formel 15 und 18, mit Hilfe des Sampling Profilers CodeAnalyst[6] von AMD analysiert.

Sperrbasiertes Verfahren

Zunächst wurde untersucht, inwieweit der Sperrerwerb zum Flaschenhals wird. Dazu wurde der Code aus Listing 4.4 (Fallbestimmung) einmal mit einem Rechenkern und einmal mit vier Rechenkernen ausgeführt. Anschließend wurde verglichen, welcher Zeitanteil pro Rechenkern für den Sperrerwerb benötigt wurde. Zu beachten ist, dass als Bezugswert nicht die Laufzeit der gesamten Unit-Propagation verwendet wird, sondern nur die Zeit, die ein Rechenkern insgesamt für die Fallbestimmung benötigt hat. Die Fallbestimmung hat bei den untersuchten Formeln im sequentiellen Fall einen Zeitanteil von 98 % an der Gesamtzeit der Unit-Propagation. In der parallelen Version mit vier Rechenkernen sind es nur 73 %. Tabelle 5.2 zeigt für die Formeln 15 und 18 die für den Sperrerwerb und Sperrfreigabe benötigte Zeit.

Formel	# Rechenkerne	Erwerb & Freigabe	Erwerb der glob. Sperre
15	1	0,12	0,01
	4	0,45	0,31
18	1	0,05	$< 0{,}01$
	4	0,13	0,01

Tabelle 5.2.: Zeitanteil des Sperrerwerbs an der Fallbestimmung

Für Formel 15 ist erkennbar, dass die Zeit für den Sperrerwerb mit der Anzahl an Threads deutlich zunimmt. Wohingegen die Version mit einem Thread ca. 12 % der Zeit für den Sperrerwerb benötigt, braucht die Version mit drei Threads mehr als das dreifache davon. Verursacht wird der erhöhte Zeitbedarf hauptsächlich durch den Erwerb der globalen Sperre (schützt die Propagation-Queue und die Literalbelegungen). Dies lässt sich damit begründen,

[6] http://developer.amd.com/tools/codeanalyst/ Version: 3.0.14-public

dass die Sperre bei einer Anforderung nicht wie im sequentiellen Fall sofort gewährt wurde, da sie noch im Besitz eines anderen Threads war. Die Stagnation der Beschleunigung ab der Verwendung von drei Rechenkernen lässt sich für Formel 15 also mit einer Erhöhung der Leerlaufzeit aufgrund des konkurrierenden Sperrerwerbs begründen. Kritisch bewertet wird, dass der Anteil von 12 % zu wenig von den ca. 63 % der Laufzeitsteigerung gegenüber der sequentiellen Version erklärt.

Für Formel 18 lässt sich der eben beschriebene Effekt nicht in gleichem Umfang beobachten. Zwar kommt es zu einer Erhöhung der Zeit, die für den Sperrerwerb und die Freigabe insgesamt benötigt wird, jedoch wird die zusätzliche Leerlaufzeit nicht durch den Erwerb der globalen Sperre verursacht. Stattdessen fällt der zusätzliche Mehraufwand hier bei den Klauselsperren (4 %), den Literalsperren (6 %) und den Sperren der Klauellisten (2 %) an. Formel 18 ist jedoch atypisch, da die Häufigkeit, mit der in der sequentiellen Version Einheitsklauseln auftreten, mit 2 % deutlich unter dem Mittelwert von 20 % aller Formeln des Typ Application liegt (vgl. Abbildung 3.3). Da die globale Sperre nur im Fall der Einheitsklausel erworben wird, erklärt dies den niedrigen Wert. Dies steht auch in Einklang mit den Laufzeiten von Formel 18, bei der die Verwendung eines vierten Rechenkerns zur weiteren Laufzeitreduktion führt.

# Rechenkerne	1_{ser}	1	2	3	4
Mittlere Laufzeit T_i	322	536	399	362	322
Veränderung	×	1,66	0,74	0,90	0,89

Tabelle 5.3.: Laufzeiten von Formel 18

Die Stagnation der Laufzeitverkürzung lässt sich damit aus der Kombination von Excess Computation und Leerlauf erklären. Für drei oder weniger Rechenkerne ist die Leerlaufzeit noch klein genug, dass weitere Laufzeitverkürzungen möglich sind. Ab vier und mehr Rechenkernen nimmt die Leerlaufzeit durch Lock Contention jedoch soweit zu, dass es nicht mehr zu weiteren Laufzeitverkürzungen kommt. Diese Erklärung basiert nur auf einer kleinen Stichprobe und bedarf der Überprüfung am gesamten Formelsatz.

Sperrfreies Verfahren

Für die sperrfreie Variante wurde ebenfalls für die Formeln 15 und 18 ermittelt, zu welchem Grad der Kontrollthread leer läuft bzw. ausgelastet ist. Der Kontrollthread verarbeitet in einer Schleife die von den lesenden Threads generierten Ergebnisse. Als Maß für die Leerlaufzeit wurde die Zeit verwendet, die der Kontrollthread mit der Überprüfung der Schleifenbedingung verbringt. Dies ist eine Näherung, da auch während des Leerlaufs versucht wird aus den Puffern zu lesen. Da die Leseversuche selbst auch Zeit benötigen, führt der Ansatz zu einer Unterschätzung der Leerlaufzeit.

Formel	# Rechenkerne	Zeitanteil
15	2	0,15
	4	0,01
18	2	0,15
	4	0,01

Tabelle 5.4.: Leerlaufzeit des Kontrollthreads

Das Ergebnis für Formel 15 und 18 zeigt Tabelle 5.4. Dabei ist erkennbar, dass der Zeitanteil, der mit der Überprüfung der Schleifenbedingung verbracht wird, mit der Anzahl der Rechenkerne deutlich abnimmt. Die Ergebnisse bekräftigen daher die These, dass der Kontrollthread zum Flaschenhals wird und weitere Rechenkerne daher nicht zu einer Verkürzung der Laufzeit führen.

5.5. Modifikation der Verfahren

Unter der Annahme, dass sich die in Abschnitt 5.4 entwickelten Erklärungsansätze auf den Formelsatz übertragen lassen, werden in diesem Abschnitt Vorschläge gemacht, wie die Verfahren verbessert werden können.

5.5.1. Sperrbasierter Ansatz

Der sperrbasierte Ansatz leidet unter zwei Problemen. Erstens wird durch den zusätzlichen Code ein Mehraufwand von ca. 60 % eingeführt. Zweitens wird die globale Sperre ab drei Threads zur kritischen Ressource, die dazu führt, dass Threads beim Warten auf eine Sperre leer laufen. Daher werden folgende Verfeinerungen zum Umgang der in Abschnitt 4.2 eingeführten Probleme vorgeschlagen:

Spontane Belegungen werden wie bisher durch Literalsperren vermieden.

Zur Lösung des Problems der Klauselvolatilität gibt es zwei Optionen. Erstens kann die bisherige Lösung der Klauselsperre beibehalten werden. Zweitens kann, um den Mehraufwand zu senken, auch eine neue Klauselstruktur eingeführt werden. Hierbei wird jede Klausel um zwei Indizes erweitert, die die Position der überwachten Literale angeben. Wichtig dabei ist, dass sich beide Indizes über eine spezielle Prozessoranweisung atomar schreiben lassen (*Compare and Swap*). Bei Verwendung dieses Schemas, kann auch die Klauselsperren verzichtet werde. Allerdings kann eine Neuberechnung einer Klausel notwendig werden, wenn die atomare Operation fehlschlägt. Dies wäre dann der Fall, wenn ein anderer Thread in der Zwischenzeit ein neues überwachtes Literal bestimmt hat.

Die Konsistenz gemeinsamer Datenstrukturen kann auch ohne Sperren gewährleistet werden, wenn diese durch sperrfreie Datenstrukturen ersetzt werden, die das gleichzeitige Lesen und Schreiben mehrerer Threads erlauben (*multiple reader, multiple writer*). Ersetzt man das Array, das bisher zur Darstellung der Klauselllisten und der Propagation-Queue verwendet wird, durch eine sperrfreie Alternative, so kann man auf die globale Sperre sowie die Sperren für Klauselllisten verzichten. Besonders beim Einsatz für die Klauselllisten sollte die sperrfreie Datenstruktur dynamisch vergrößer- und verkleinerbar sein. Wenn diese Anforderung der dynamischen Größe zu stark ist, so kann die sperrfreie Datenstruktur nur für die Propagation-Queue verwendet werden, da deren Größe durch die Anzahl der Literale der Formel beschränkt ist. Der Zugriff auf die Klauselllisten kann in diesem Fall weiterhin über Sperren synchronisiert werden. Besonders der Zugriff auf die Propagation-Queue sowie auf das Array assigns werden ab drei Threads zum Flaschenhals.

Das Problem, dass bestimmte Änderungen nur atomar erfolgen dürfen (siehe Abschnitt 4.2.3), lässt sich durch die Anpassung der Algorithmen erreichen. Die Reihenfolge der Propagation-Queue lässt sich z. B. auch über eine nachträgliche Bearbeitung oder die umgekehrte Einfügereihenfolge (erst Propagation-Queue, dann Belegung) einhalten.

Obige Vorschläge zielen darauf ab, den durch den Sperrerwerb verursachten Mehraufwand zu reduzieren und den Wettbewerb um Sperren zu verringern. Die Kombination beider Maßnahmen kann zu einer höheren Effizienz und besseren Skalierbarkeit führen.

5.5.2. Sperrfreier Ansatz

Beim sperrfreien Ansatz deutet sich an, dass der Kontrollthread ab einer Anzahl von drei Threads ausgelastet ist. Um diesen Umstand zu beheben, können mehrere Kontrollthreads verwendet werden. Damit trotzdem keines der in Abschnitt 4.2 aufgeführten Probleme auftritt, werden alle Variablen der Formel in gleich große Partitionen zerlegt und jede Partition je einem Kontrollthread zugewiesen. Jeder Kontrollthread darf nur die Variablen seiner ihm zugewiesenen Partition belegen. Damit werden die Probleme wie folgt vermieden:

Spontane Belegungen werden ausgeschlossen, indem eine Variable nur von genau einem Thread belegt werden darf. In zwei Fällen hängt das Ergebnis der Fallbestimmung von einem unbelegten Literal ab. Im ersten Fall wurde ein neu überwachtes Literal gefunden. Ist dieses unbelegt, so wird das berechnete Ergebnis zur Validierung dem Kontrollthread übermittelt, der das neu überwachte Literal belegen darf. Ist das Literal bereits mit *wahr* belegt, darf es von einem beliebigen Thread bearbeitet werden. Der zweite Fall betrifft eine potentielle Einheitsklausel. Auch in diesem Fall darf die Klausel nur von demjenigen Kontrollthread bearbeitet werden, der aufgrund der Variablenpartitionierung berechtigt ist das Literal zu belegen. Durch die Sequentialisierung innerhalb eines Threads kann sich der Wert einer Variablen nicht während der Bearbeitung einer Klausel ändern.

Zur Lösung des Problems der Klauselvolatilität wird die erweitere Klauselstruktur mit Indizes verwendet und diese atomar geändert (siehe gleicher Punkt auf Seite 82).

Die Konsistenz gemeinsamer Datenstrukturen wird sichergestellt, indem für die Klauselllisten und die Propagation-Queue eine sperrfreie Datenstruktur verwendet wird. Damit die Atomarität bei den angegeben Operationen erhalten bleibt, müssen die Algorithmen

angepasst werden. Zur Einhaltung der Reihenfolge der Propagation-Queue kann man die Lösung des modifizierten, sperrbasierten Ansatzes übernehmen. Eine Race-Condition auf den Klausellisten (vgl. Abschnitt 4.2.3) kann nicht auftreten, da eine Belegung nur durch den bearbeitenden Kontrollthread zulässig ist.

6. Schlussfolgerungen und Fazit

In der Arbeit wurden neuartige Algorithmen zur Unit-Propagation entwickelt. Im Ergebnis konnten weder mit dem sperrbasierten noch mit dem sperrfreien Ansatz in der Mehrzahl der untersuchten Fälle Beschleunigungen erzielt werden. Insgesamt schneidet der sperrbasierte Ansatz deutlich besser ab. So erzielte dieser mit drei Rechenkernen auf fünf von 20 Formeln im Mittel eine Beschleunigung von 20 %. Darüber hinaus braucht dieser zur Lösung aller Formeln des Testset mit drei Rechenkernen das 1,16-fache der sequentiellen Laufzeit, der sperrfreie Ansatz dagegen das 1,9-fache des sequentiellen Algorithmus. Bei beiden Ansätzen schnitten die grobgranularen Varianten in der Mehrzahl der Fälle deutlich besser ab als die entsprechende feingranularen. Eine weitere Vertiefung der feingranularen Ansätze scheint daher nicht lohnenswert.

Beide grobgranularen Varianten erreichen die niedrigste Laufzeit mit drei – nicht mit vier oder mehr – Rechenkernen. In Abschnitt 5.4 wurden Erklärungsansätze entwickelt, warum dies so ist. Neben der hohen Excess Computation gibt es Hinweise darauf, dass die globale Sperre, die zum Zugriff auf die Propagation-Queue und die Literalbelegungen benötigt wird, mit steigender Zahl der Rechenkerne zum Flaschenhals wird. Im sperrfreien Ansatz sprechen die Stichproben dafür, dass der Kontrollthread bereits bei drei Rechenkernen voll ausgelastet ist. Eine weitere Erhöhung der Zahl der verwendeten Rechenkerne führt daher nicht mehr zu einer Reduktion der Laufzeit.

Ein abschließendes Urteil über die Tauglichkeit der vorgeschlagenen Algorithmen lässt sich noch nicht treffen. Die Schwächen beider Varianten wurden analysiert und Verbesserungen vorgeschlagen. Wenn es gelingt, die Excess Computation und den sperrbedingten Leerlauf durch die vorgeschlagenen Maßnahmen weiter zu senken, könnte sich der sperrbasierte Ansatz für kleine Anzahl an Rechenkernen durchaus eignen. Das Grundsatzproblem der Lock Contention wird aber mit zunehmender Anzahl an Threads nicht vermeidbar sein. Daher kann ein weiterentwickelter sperrfreier Ansatz trotz seiner hohen Excess Computation vorteilhaft sein, wenn der Algorithmus auf einer hohen Zahl von Rechenkernen skalieren soll. Für die

modifizierten Ansätze entscheidend ist die Verfügbarkeit einer sperrfreien Datenstruktur, die von mehreren Threads gleichzeitig gelesen und geschrieben werden kann.

A. Testformelsatz: Verteilung der Klauselgröße und Variablenanzahl

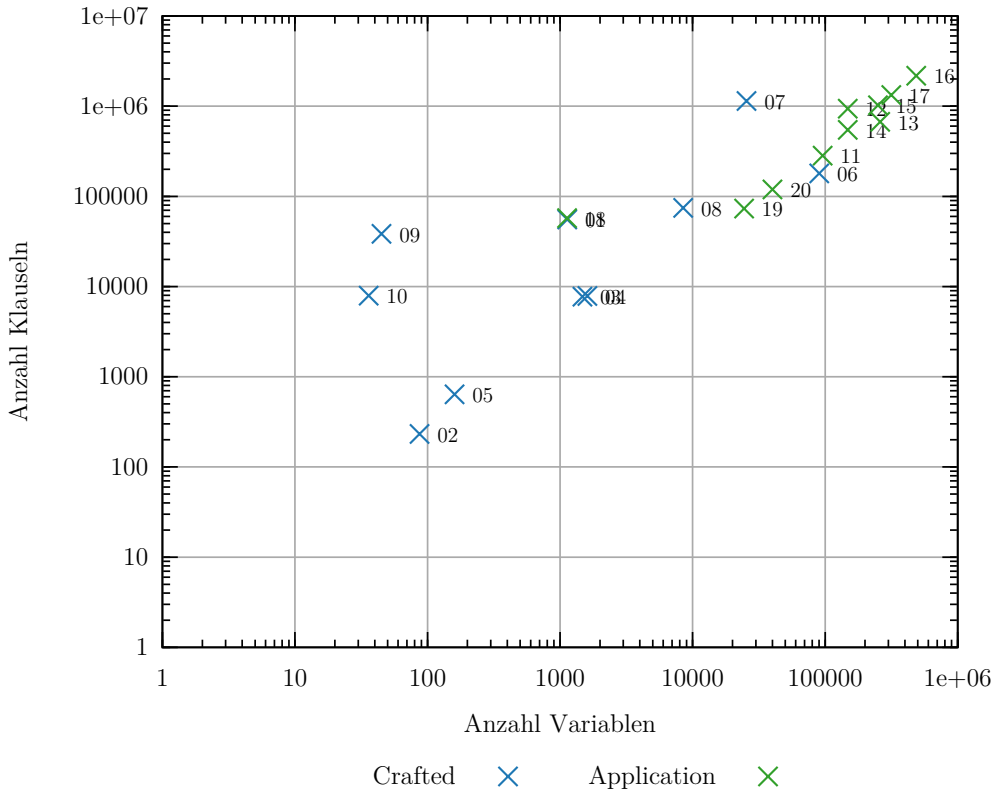


Abbildung A.1.: Verteilung der Klauselgrößen und Variablenanzahl je Formel

B. Durchführung der Zeitmessung

Für Zeitmessungen wurde die Echtzeituhr des Systems (zur verwendeten Hard- und Software siehe Abschnitt 5.2) mit der POSIX-Funktion clock_gettime abgefragt (Parameter: CLOCK-REALTIME). Durch die Zeitmessung selbst kann ein Overhead entstehen, der fälschlicherweise zur Zeit des ausgeführten Codes gezählt wird. Daher wurde in einem ersten Schritt in einem eigenen Programm ermittelt, wie lange der Aufruf zweier clock_gettime-Aufrufe (Start, Ende) dauert. Dazu wurden 10^8 Aufrufe in einer Schleife ausgeführt und die vergangene Zeit erfasst (13s). Im Mittel dauern zwei Aufrufe daher ca. 130ns.

Bei der Ermittlung des Laufzeitanteils der Unit-Propagation wurde die Anzahl der Aufrufe der Funktion clock_gettime erfasst, um so den maximalen Einfluss der Zeitmessung auf das Ergebnis abschätzen zu können. Es zeigte sich, dass der daraus entstandene Fehler für 17 von 20 Formeln vernachlässigbar klein ist (Mittelwert 0,2s, durchschnittliche Laufzeit der Formeln: 333s). Für drei Formeln wurden allerdings so viele Zeitmessungsaufrufe durchgeführt, dass deren Einfluss nicht mehr vernachlässigbar klein blieb (Mittelwert: 19s).

Noch größer war der Overhead bei der Messung, wie sich die Gesamtzeit der Unit-Propagation auf die einzelnen Fälle verteilt. Addiert man für eine Formel die für jeden Fall i erfasste Zeit t_i, so müsste wegen der Wiederholbarkeit die Summe alle Fälle ($t := t_1 + t_2 + t_3 + t_4$) der Gesamtzeit der Unit-Propagation t_{up} (wie oben ermittelt) entsprechen. Tatsächlich war t im Mittel jedoch um den Faktor Faktor 2,3 größer. Dies wird damit, dass bei der Zeitmessung selbst Zeit vergeht, die zur Ausführungszeit der Fälle hinzu gerechnet wird. Dies kann den Zeitanteil eines Falles verfälschen.

Abbildung B.1 erläutert, wie der Overhead zustande kommt und zeigt wie man den Einfluss auf die Zeitmessung verringern kann. Ein Aufruf der Funktion clock_gettime wird vom Compiler in eine Reihe von Assembler-Anweisungen übersetzt. Dabei verstreicht vor dem Auslesen des High-Performance Counters (HPC) die Zeit o_1, nach dem Auslesen die Zeit $o2$. Die Zeit o_1, o_2 kann z. B. durch Parameterübergabe, den Aufruf der Systemfunktion, das Sichern und Wiederherstellen von Registern verursacht werden. Es wird nicht vorausgesetzt, dass o_1 und o_2

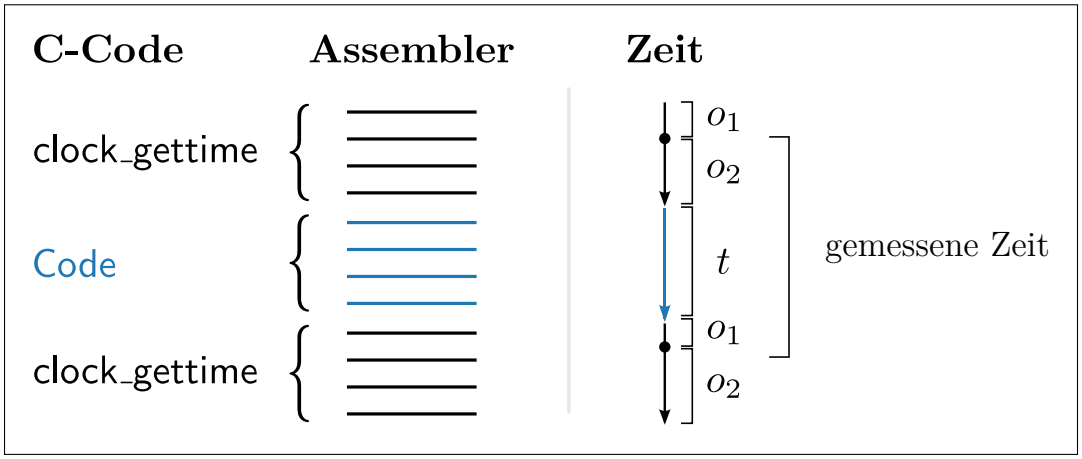


Abbildung B.1.: Fehler bei der Zeitmessung

gleich groß sind, jedoch wird angenommen, dass ihr Verhältnis konstant ist. Obwohl man in Abbildung B.1 also die Zeit t messen möchte, misst man stattdessen $t + o_1 + o_2$. Der Anteil an Overhead lässt sich rechnerisch verringern, da $o_1 + o_2$ gerade der Zeit entspricht, die benötigt wird um die Funktion clock_gettime einmal aufzurufen. Diese lässt sich wie oben beschrieben ermitteln und beträgt für das verwendete System 65 ns. Ist für eine Formel der Fall i c_i-mal eingetreten und hat dafür t_i Sekunden benötigt, so berechnet sich die um den Overhead bereinigte Zeit zu $t'_i = t_i - c_i * (o_1 + o_2)$.

Um den Ansatz zu überprüfen, wurden die bereinigten Zeiten für jede Formel summiert ($t'_{up} = t'_1 + t'_2 + t'_3 + t'_4$). Im Mittel unterscheidet sich dieser Wert (t'_{up}) von den tatsächlichen Laufzeiten (t_{up}) nur noch um den Faktor 0,1 (vorher: 2,3). Der verbesserte Ansatz, der den Overhead berücksichtigt, liefert also eine deutlich genauere Zeitverteilung. Daher wurde die Messung zur Ermittlung des Laufzeitanteils der Unit-Propagation nochmal mit dem eben vorgestellten Ansatz durchgeführt.

C. Sperrfreier Zyklischer Ringpuffer

Der Ringpuffer wird im sperrfreien Ansatz zum Nachrichtenaustausch zwischen Threads benutzt. Der Zugriff aus zwei Threads ist erlaubt, solange einer nur lesend (dequeue) und der andere nur schreibend (enqueue) zugreift. Die Elemente werden in einem Puffer fester Größe auf dem Heap gespeichert. Die Größe des Puffers ist eine Potenz von 2, damit die Modulo-Funktion effizient über Bitschiebe-Operationen berechnet werden kann. Der Ringpuffer verwendet die Variablen rCount und wCount, um den Füllstand und die nächste Schreib- bzw. Leseposition zu verwalten.

```
1  /**
2   * Single reader, single writer lock-free circular buffer
3   * Lockfree extension of:
4   * http://en.wikipedia.org/wiki/Circular_buffer#Read_.2F_Write_Counts
5   */
6  template <typename T>
7  class LFBuffer
8  {
9  private:
10     unsigned long long capacity;
11     volatile unsigned long long rCount, wCount;
12     T * buf;
13
14     // http://en.wikipedia.org/w/index.php?title=Power_of_two&oldid=461172157
15     // #Fast_algorithm_to_find_a_number_modulo_a_power_of_two
16     inline unsigned int fast_mod (unsigned int a) {return (a & (capacity - 1));}
17
18 public:
19     LFBuffer (unsigned int twoexp = 20) : rCount(0ULL), wCount(0ULL){
20         capacity = 1 << twoexp;
21         buf = new T [capacity];
22     }
23
```

```
24      ~LFBuffer() {delete [] buf;}
25
26      /**
27       * wCount is guaranteed to be stable, rCount may be incremented in parallel.
28       * Blocks if the buffer is full.
29       *
30       */
31      inline void enqueue (T & elem){
32            bool full;
33            do {
34                  full = ((rCount + capacity) == wCount);
35            }while (full);
36            int pos = fast_mod(wCount);
37            buf[pos] = elem;
38            __sync_add_and_fetch(&wCount, 1);
39      }
40
41      /**
42       * rCount is guaranteed to be stable, wCount may be incremented in parallel.
43       * returns false if buffer is empty
44       */
45      inline bool dequeue (T & elem){
46            if (rCount == wCount)
47                  return false; // empty
48            int pos = fast_mod(rCount);
49            elem = buf[pos];
50            __sync_add_and_fetch(&rCount, 1);
51            return true;
52      }
53
54      inline void clear()    {rCount = wCount;}
55  };
```

Listing C.1: Sperrfreier zyklischer Ringpuffer

Das Einfügen erfolgt, indem zunächst geprüft wird, ob der Puffer voll ist. Bei der Prüfung wird ein möglicher Integerüberlauf berücksichtigt. Wenn er voll ist, wird das Einfügen solange verzögert, bis durch das parallele Lesen eines Elements wieder Platz im Puffer ist. Die Position pos des Elements wird mit der Modulo-Funktion berechnet und das Element dort eingefügt. Als letzte Operation wird die Variable wCount inkrementiert, wodurch das eingefügte Element für lesende Threads sichtbar wird.

Beim Lesen wird zunächst überprüft, ob der Puffer leer ist. Wenn er das nicht ist, wird mit der Modulo-Funktion der Index berechnet, der gelesen werden soll. Nach dem Kopieren des Inhalts wird rCount inkrementiert und damit zum Überschreiben freigegeben.

Literaturverzeichnis

[1] Apiki, Steve: *Lock-Free Programming on AMD Multi-Core Systems.* Webseite, 5 2006. http://developer.amd.com/documentation/articles/pages/125200689.aspx (zuletzt geprüft: 21.11.2011).

[2] Bayardo, Jr., Roberto J. und 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, Seiten 203–208. AAAI Press, 1997.

[3] Beame, Paul, Henry Kautz und Ashish Sabharwal: *Towards understanding and harnessing the potential of clause learning.* Journal of Artificial Intelligence Research, 22:319–351, 2004.

[4] Biere, Armin, Marijn J. H. Heule, Hans van Maaren und Toby Walsh (Herausgeber): *Handbook of Satisfiability*, Band 185 der Reihe *Frontiers in Artificial Intelligence and Applications.* IOS Press, February 2009.

[5] Biere, Armin und Carsten Sinz: *Decomposing SAT Problems into Connected Components.* Journal on Satisfiability, Boolean Modeling and Computation, 2006. Accepted for publication.

[6] Blochinger, Wolfgang, Wolfgang Küchlin und Andreas Weber: *The Distributed Object-Oriented Threads System DOTS.* In: Ferreira, Afonso, José D. P. Rolim, Horst D. Simon und Shang-Hua Teng (Herausgeber): *IRREGULAR*, Band 1457 der Reihe *Lecture Notes in Computer Science*, Seiten 206–217. Springer, 1998.

[7] Blochinger, Wolfgang, Carsten Sinz und Wolfgang Küchlin: *A Universal Parallel SAT Checking Kernel.* In: Arabnia, Hamid R. und Youngsong Mun (Herausgeber): *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and*

Applications (PDPTA'03), Band 4, Seiten 1720–1725, Las Vegas, NV, June 2003. CSREA Press.

[8] Blochinger, Wolfgang, Carsten Sinz und Wolfgang Küchlin: *Parallel propositional satisfiability checking with distributed dynamic learning.* Parallel Computing, 29:969–994, 2003.

[9] Blochinger, Wolfgang, W. Westje, Wolfgang Küchlin und Sebastian Wedeniwski: *ZetaSAT - Boolean SATisfiability solving on Desktop Grids.* In: *CCGRID*, Seiten 1079–1086. IEEE Computer Society, 2005.

[10] Böhm, Max und Ewald Speckenmeyer: *A Fast Parallel SAT-Solver - Efficient Workload Balancing.* In: *Annals of Mathematics and Artificial Intelligence*, Seiten 40–0, 1996.

[11] Cook, Stephen: *The importance of the P versus NP question.* J. ACM, 50:27–29, January 2003.

[12] Cook, Stephen A.: *The complexity of theorem-proving procedures.* In: *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, Seiten 151–158, New York, NY, USA, 1971. ACM.

[13] Davis, Martin, George Logemann und Donald Loveland: *A machine program for theorem-proving.* Commun. ACM, 5:394–397, July 1962.

[14] Davis, Martin und Hilary Putnam: *A Computing Procedure for Quantification Theory.* J. ACM, 7:201–215, July 1960.

[15] Devlin, David: *B.: Satisfiability as a Classification Problem.* In: *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*, 2008.

[16] Eén, Niklas und Niklas Sörensson: *An Extensible SAT-solver.* In: Giunchiglia, Enrico und Armando Tacchella (Herausgeber): *SAT*, Band 2919 der Reihe *Lecture Notes in Computer Science*, Seiten 502–518. Springer, 2003.

[17] Gasarch, William I.: *The P=?NP Poll.* SIGACT News Complexity Theory Column 36, 33(2):34–47, 2002.

[18] Grama, A.: *Introduction to parallel computing*, Kapitel 3, Seite 85ff. Pearson Education. Addison-Wesley, 2nd Auflage, 2003.

[19] Grama, A.: *Introduction to parallel computing*, Kapitel 11.6, Seite 501ff. Pearson Education. Addison-Wesley, 2 Auflage, 2003.

[20] Hamadi, Youssef, Said Jabbour, Cédric Piette und Lakhdar Saïs: *Deterministic Parallel DPLL: System Description.* In: *Pragmatics of SAT(POS'11)*, jun 2011.

[21] Harmelen, Frank van, Vladimir Lifschitz und Bruce Porter (Herausgeber): *Handbook of Knowledge Representation*, Kapitel 2. Foundations of Artificial Intelligence. Elsevier Science, San Diego, USA, 1 Auflage, 2008.

[22] Harris, Tim, Adrian Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith und Mateo Valero: *Transactional Memory: An Overview.* IEEE Micro, 27:8–29, 2007.

[23] Hölldobler, Steffen, Norbert Manthey und Ari Saptawijaya: *Improving resource-unaware SAT solvers.* In: *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, Seiten 519–534, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] Kottler, Stephan und Michael Kaufmann: *A parallel portfolio SAT solver with lockless physical clause sharing.* Technischer Bericht, Universitaetsbibliothek Tuebingen, Wilhelmstr. 32, 72074 Tübingen, 2011.

[25] Lewis, Matthew, Tobias Schubert und Bernd Becker: *Multithreaded SAT Solving.* In: *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, Seiten 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Mancinelli, Fabio, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy und Ralf Treinen: *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions.* In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, Seiten 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Manthey, Norbert: *Parallel SAT Solving - Using More Cores.* In: *Pragmatics of SAT(POS'11)*, 2011.

[28] Marques-Silva, Joao: *Practical Applications of Boolean Satisfiability.* In: *Workshop on Discrete Event Systems (WODES'08).* IEEE Press, May 2008.

[29] Mitchell, David G.: *A SAT Solver Primer.* Bulletin of The European Association for Theoretical Computer Science, 85:112–132, 2005.

[30] Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang und Sharad Malik: *Chaff: engineering an efficient SAT solver.* In: *Proceedings of the 38th annual Design Automation Conference*, DAC '01, Seiten 530–535, New York, NY, USA, 2001. ACM.

[31] Post, Hendrik und Wolfgang Küchlin: *Integrated Static Analysis for Linux Device Driver Verification.* In: *IFM*, Seiten 518–537, 2007.

[32] Rauber, Rünger: *Parallele und verteilte Programmierung*, Kapitel 6.2, Seite 313. Springer, 2000.

[33] Ross, P.E.: *Why CPU Frequency Stalled.* Spectrum, IEEE, 45(4):72, april 2008.

[34] Schulz, Sven und Wolfgang Blochinger: *Parallel SAT Solving on Peer-to-Peer Desktop Grids.* J. Grid Comput., 8(3):443–471, 2010.

[35] Schöning, Uwe: *Logik für Informatiker*, Kapitel 1. Spektrum Akademischer Verlag, 5. Auflage, 2000.

[36] Silva, João P. Marques: *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms.* In: *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, Seiten 62–74, London, UK, 1999. Springer-Verlag.

[37] Silva, João P. Marques und 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, Seiten 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[38] Singer, Daniel: *Parallel Resolution of the Satisfiability Problem: A Survey*, Kapitel 5. Wiley Interscience, Oktober 2006.

[39] Sinz, Carsten, Wolfgang Blochinger und Wolfgang Küchlin: *PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications.* Electronic Notes in Discrete Mathematics, 9:205–216, 2001.

[40] Sutter, Herb: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3):202–210, 2005.

[41] Sutter, Herb und James Larus: *Software and the Concurrency Revolution.* Queue, 3:54–62, September 2005.

[42] Sörensson, Niklas: *MINISAT 2.2 and MINISAT++ 1.1.* Tech. rep. SAT-Race 2010, 2010.

[43] Tanenbaum, A.S.: *Moderne Betriebssysteme*, Kapitel 6, Seite 535ff. Pearson Studium. Pearson Studium, 3 Auflage, 2009.

[44] Tucker, Chris, David Shuffelton, Ranjit Jhala und Sorin Lerner: *OPIUM: Optimal Package Install/Uninstall Manager.* In: *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, Seiten 178–188, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Zhang, Hantao, Maria Paola Bonacina, Maria Paola, Bonacina und Jieh Hsiang: *PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems.* Journal of Symbolic Computation, 21:543–560, 1996.

[46] Zhang, Lintao, Conor F. Madigan, Matthew H. Moskewicz und 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, Seiten 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Jan Stöcklin)