SAT-Algorithmen und Systemaspekte: vom Mikroprozessor zum parallelen System

Dissertation zur Erlangung des Doktorgrades
der Fakultät für Angewandte Wissenschaften
der Albert-Ludwigs-Universität Freiburg im Breisgau

Tobias Schubert

März 2008

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg im Breisgau

Dekan:	Prof. Dr. Bernhard Nebel
Erstreferent:	Prof. Dr. Bernd Becker
Zweitreferent:	Prof. Dr. Rolf Drechsler
Datum der Promotion:	2. Mai 2008

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Rechnerarchitektur der Fakultät für Angewandte Wissenschaften der Albert-Ludwigs-Universität Freiburg. Sowohl die Forschungsarbeit als auch die Betreuung zahlreicher Lehrveranstaltungen in den vergangenen Jahren haben mir viel Freude bereitet. Insbesondere die Möglichkeit, im Rahmen des *Mobilen Hardware-Praktikums* neue Hardware-Komponenten entwickeln und diese erfolgreich in den alltäglichen Lehrbetrieb einbringen zu können, war eine große und spannende Herausforderung.

Mein Dank gilt in erster Linie den wichtigsten Personen in meinem Leben: meiner Familie. Euch ein herzliches Dankeschön für die geduldige und bedingungslose Unterstützung meiner Arbeit.

Ebenso möchte ich mich bei allen aktiven und ehemaligen Mitarbeitern des Lehrstuhls für Rechnerarchitektur bedanken. Besonders Herr Prof. Dr. Bernd Becker hat mich in meiner wissenschaftlichen Entwicklung nachhaltig geprägt. Nur durch sein in mich gesetztes Vertrauen wurde die vorliegende Arbeit überhaupt möglich.

Freiburg im Breisgau, März 2008 *Tobias Schubert*

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den beiden zurückliegenden Jahrzehnten haben Computer massiv Einzug in den Alltag gehalten und unser Leben dadurch nachhaltig verändert. Per E-Mail mit Menschen auf der ganzen Welt in Kontakt treten zu können ist mittlerweile genauso wenig wegzudenken wie der Zugang zum weltweiten Internet. Gerade das Internet mit all den darauf aufbauenden Geschäftsmodellen hat sich in den letzten Jahren zu einem beträchtlichen Wirtschaftszweig entwickelt. Der „klassische“ Computer, ausgestattet mit dem eigentlichen Rechner und diversen zusätzlichen Komponenten wie Tastatur, Maus, Bildschirm, Modem und Drucker, gehört daher in den meisten Haushalten heutzutage zur Grundausstattung.

Im Wesentlichen sind es aber nicht die klassischen Computer, sondern die so genannten *eingebetteten Systeme*, mit denen man täglich und fast überall in Berührung kommt. Diese Systeme sind, wie der Name andeutet, in eine „Umgebung“ eingebettet und übernehmen dort komplexe Regelungs-, Steuerungs- und Datenverarbeitungsaufgaben. Die Realisierung basiert im Allgemeinen auf Spezialhardware, bestehend aus einem oder mehreren Mikroprozessoren und diversen, von den Mikroprozessoren mit Hilfe entsprechender Programme gesteuerten Sensoren und Aktoren. Exemplarisch seien Haushaltsgeräte (Kühlschränke, Waschmaschinen), der Medizin-Sektor (Hörgeräte, Herzschrittmacher) und die Verkehrsbranche (Airbagsteuerung in Autos, Autopilot in Flugzeugen) genannt.

Besonders die beiden letztgenannten Beispiele verdeutlichen, dass mit einer steigenden Verbreitung und Akzeptanz von Computern und eingebetteten Systemen auch deren Einsatz in sicherheitskritischen Anwendungen steigt. Ein Fehlverhalten während des Betriebs, unabhängig ob durch ein fehlerhaftes Design der Hardware oder der darauf ausgeführten Software bedingt, verursacht hier üblicherweise nicht nur einen erheblichen finanziellen Schaden, sondern gefährdet auch Menschenleben. Beschränkt man sich in diesem Zusammenhang auf Aspekte des rechnergestützten Schaltkreisentwurfs, so ist offensichtlich, dass dem Entwickler unter anderem Werkzeuge bereitgestellt werden müssen, mit denen entworfene Schaltungen während des gesamten Entwurfsprozesses immer wieder getestet und verifiziert werden können. Dies ist umso bedeutender, wenn die entstehenden Komponenten in sicherheitskritischen Anwendungen eingesetzt werden sollen.

Als Lösungsansatz für zahlreiche Fragestellungen aus dem Bereich des rechnergestützten

Schaltkreisentwurfs, nicht allein beschränkt auf die zuvor genannten Aspekte Test und Verifikation, haben sich in den letzten Jahren verstärkt *SAT-Algorithmen* etabliert [80]. Dabei hat sich die Kurzform *SAT* beziehungsweise *SAT-Problem*, in Anlehnung an die englische Übersetzung *Boolean Satisfiability Problem*, in der Literatur als Synonym für das Erfüllbarkeitsproblem der Aussagenlogik durchgesetzt. Bei SAT-Algorithmen handelt es sich um Verfahren, die in Form einer Booleschen Formel eine Instanz des Erfüllbarkeitsproblems der Aussagenlogik entgegennehmen und versuchen, diese Formel durch geeignete Zuweisungen an die darin enthaltenen Variablen zu erfüllen. Beispielsweise kann die Spezifikation einer kombinatorischen Schaltung zusammen mit dem anhand dieser Vorgaben entwickelten Schaltkreis so in ein SAT-Problem transformiert werden, dass eine erfüllende Belegung genau dann ermittelt werden kann, wenn Spezifikation und Implementierung der Schaltung für zumindest eine Eingabe ein unterschiedliches Ausgabeverhalten zeigen. Existiert eine solche Variablenbelegung nicht, das heißt, ist die Formel nicht erfüllbar, so sind Spezifikation und Implementierung bezüglich ihres Ein- und Ausgabeverhaltens identisch. In diesem Fall ist die Korrektheit der entwickelten Hardware-Komponente gegenüber der geforderten Spezifikation formal bewiesen.

Wie von Cook im Jahr 1971 gezeigt werden konnte, gehört das Erfüllbarkeitsproblem der Aussagenlogik zur Klasse der NP-vollständigen Probleme [25]. Unter der Annahme $NP \neq P$ ist daher nicht mit einem Verfahren zu rechnen, das beliebige Probleminstanzen mit stets polynomieller Laufzeit (mit vertretbarem Zeitaufwand) löst. Dennoch haben zahlreiche neue Techniken und Optimierungen auf Seiten der Implementierung dazu geführt, dass moderne SAT-Algorithmen wie MiniSat [32], PicoSAT [112], RSat [113] oder auch zChaff [86] heute in der Lage sind, eine Vielzahl an industriell relevanten Fragestellungen erfolgreich zu lösen.

Es ist aber zu beobachten, dass durch den vermehrten und erfolgreichen Einsatz von SAT-Algorithmen auch die Komplexität der zu lösenden Probleme kontinuierlich ansteigt. Jedes Verfahren ist folglich nur eine Momentaufnahme, die stetig verbessert und optimiert werden muss, um den steigenden Anforderungen auch künftig gerecht zu werden. Neben Optimierungen der zumeist sequentiellen Algorithmen bietet sich mit der Parallelisierung, bei der mehrere sequentielle SAT-Prozeduren zu einem parallelen Algorithmus zusammengefasst werden, eine alternative Methode zur Leistungssteigerung an.

In der Regel basiert das Vorgehen dabei auf einer dynamisch zur Laufzeit durchgeführten Partitionierung der Probleminstanz in jeweils voneinander disjunkte Teilbereiche, die dann von den verschiedenen sequentiellen SAT-Prozeduren parallel gelöst werden. Auf diese Weise bearbeitet jede an der Suche nach einer erfüllenden Belegung beteiligte SAT-Prozedur lediglich einen Teil, alle zusammen aber das Gesamtproblem. Je nach Implementierung dient die Kommunikation zwischen den einzelnen Prozessen dabei nicht nur dem Austausch von Teilproblemen und Statussignalen, sondern wird auch dazu genutzt, relevante Informationen über die zu lösende Probleminstanz zu transferieren, was üblicherweise in

Form so genannter *Konflikt-Klauseln* erfolgt. Dadurch ist es einer einzelnen SAT-Prozedur möglich, direkt von der Arbeit der restlichen Prozesse zu profitieren, was sich bei einer effizienten Umsetzung positiv auf die benötigte Laufzeit auswirkt. Bedingt durch die Aufspaltung des Problems, die parallele Bearbeitung der einzelnen Bereiche und insbesondere durch den Austausch geeigneter Daten zwischen den sequentiellen SAT-Prozeduren sollte eine gegebene Probleminstanz daher mit einem parallelen SAT-Algorithmus stets schneller bearbeitet werden können als mit einem vergleichbaren sequentiellen Verfahren.

Die Größenordnung, in der sich der Performance-Gewinn eines parallelen SAT-Algorithmus gegenüber einem vergleichbaren sequentiellen Verfahren bewegt, hängt maßgeblich von zwei Faktoren ab: wie gut ist der parallele Algorithmus an die spezifischen Eigenschaften der zugrunde liegenden Hardware-Umgebung angepasst worden und in welchem Umfang können die einzelnen Prozesse durch den Austausch geeigneter Daten voneinander profitieren. In diesem Zusammenhang gilt es daher Aspekte wie die Anzahl der verfügbaren Prozessoren, die Art der Kommunikation zwischen diesen sowie die Anbindung an den Speicher beim Entwurf eines effizienten parallelen Verfahrens zu beachten. Weiterhin sollten die eingesetzten sequentiellen SAT-Prozeduren bereits über ein gewisses Leistungspotenzial verfügen, um die Konkurrenzfähigkeit zu anderen *State-of-the-Art* Ansätzen in diesem Bereich zu gewährleisten.

An genau diesem Punkt setzt die vorliegende Arbeit an. Für unterschiedliche Hardware-Plattformen wird untersucht, mit welchen speziell an die jeweilige Architektur angepassten Datenstrukturen und Routinen parallele SAT-Algorithmen einen im Vergleich zu einem sequentiellen Verfahren „maximalen“ Geschwindigkeitsvorteil erzielen. In einem ersten Schritt sind zunächst leistungsstarke sequentielle SAT-Prozeduren entwickelt worden, die dann im zweiten Schritt, unter Ausnutzung der spezifischen Eigenschaften der jeweils anvisierten Zielplattform, zu parallelen SAT-Algorithmen erweitert wurden. Die Palette der entwickelten Verfahren deckt sowohl die hardwarenahe Programmierung in Maschinensprache als auch die Programmierung in einer Hochsprache ab. Im Einzelnen umfasst der Beitrag dieser Arbeit die folgenden parallelen SAT-Algorithmen:

- PIChaff: eine in Maschinensprache und C erfolgte Implementierung für ein am Lehrstuhl für Rechnerarchitektur entwickeltes Multiprozessorsystem auf Basis von *Microchip PIC17C43* und *Motorola MC68340* Mikroprozessoren.

- MiraXT: eine threadbasierte Entwicklung auf Basis von C/C++, die speziell auf solche Systeme zugeschnitten wurde, bei denen alle Prozessoren an einen gemeinsamen Speicher angebunden sind. Dies trifft beispielsweise auf aktuelle Dual-/Multi-Core Prozessoren [4, 55, 56, 109] zu, aber auch auf so genannte Mehrprozessorsysteme [6, 7, 53], bei denen mehrere Prozessoren in jeweils eigenen Fassungen auf der Hauptplatine des Rechners untergebracht sind. Der von allen Threads gemeinsam genutzte Speicher ermöglicht es, dass jede sequentielle SAT-Prozedur jederzeit auf alle Daten,

die für den von ihr aktuell bearbeiteten Bereich des Gesamtproblems relevant sind, zugreifen kann, auch wenn diese ursprünglich von einem anderen Thread, ausgeführt auf einem anderen Prozessor, bereitgestellt wurden.

– PaMiraXT: eine Erweiterung von MiraXT, bei der mittels einer zweistufigen Form der Parallelität der Einsatz auch in Rechnernetzwerken ermöglicht wird, bei denen die einzelnen Rechner zwar per Ethernet-Verbindung miteinander verknüpft sind, aber nicht alle Prozessoren Zugriff auf einen gemeinsamen Speicher besitzen. Auf der ersten Stufe wird auf allen an der Lösung eines SAT-Problems beteiligten Rechnern des Netzwerks MiraXT ausgeführt und zwar je nach Anzahl der lokal auf diesem Rechner vorhandenen Prozessoren beziehungsweise CPU-Kerne entweder in der sequentiellen (mit einem Thread) oder in der parallelen Variante (mit mehreren Threads). Dem übergeordnet werden auf der zweiten Stufe alle so gestarteten „MiraXT-Kopien“ zum parallelen SAT-Algorithmus PaMiraXT zusammengeführt.

Im Gegensatz zu den beiden anderen Ansätzen war bei PIChaff die Umsetzung des eigentlichen SAT-Algorithmus nur eine der zu lösenden Aufgaben. Zudem mussten insbesondere auch Methoden zur Realisierung der interruptgestützten Kommunikation zwischen den Mikroprozessoren auf unterster Hardware-Ebene entwickelt werden. Bei MiraXT beziehungsweise PaMiraXT konnte an dieser Stelle mit PThread [22] und MPICH [48] auf bestehende Funktionsbibliotheken beziehungsweise Software-Pakete zurückgegriffen werden, ohne deren explizite Umsetzung der Kommunikation auf Hardware-Ebene im Detail berücksichtigen zu müssen.

Bei allen drei genannten SAT-Algorithmen handelt es sich um *vollständige* Verfahren im Stil der Davis-Logemann-Loveland Prozedur [27]. Für eine gegebene Probleminstanz kann bei diesen Ansätzen garantiert werden, dass, sofern eine erfüllende Belegung existiert, diese auch gefunden wird. Unter Umständen wird dazu der gesamte Suchraum systematisch durchsucht, was auch den Begriff „vollständig“ erklärt. Im Umkehrschluss sind vollständige SAT-Algorithmen, im Gegensatz zu GSAT [100], WSAT [99] oder ähnlichen Ansätzen, dadurch auch in der Lage, die Unerfüllbarkeit eines Problems nachzuweisen. In einigen Teildisziplinen des Schaltkreisentwurfs, wie etwa der zuvor angedeuteten Verifikation kombinatorischer Schaltkreise (das so genannte *Combinational Equivalence Checking*), ist der Nachweis der Unerfüllbarkeit das vorrangige Ziel, zeigt es doch die Korrektheit der entworfenen Schaltung gegenüber der geforderten Spezifikation. Aufgrund der Problemstellung sind in derartigen Szenarien nur vollständige SAT-Algorithmen anwendbar.

1.1 PIChaff

Den Anfang macht mit PIChaff ein paralleler SAT-Algorithmus, der speziell an ein am Lehrstuhl für Rechnerarchitektur entwickeltes Multiprozessorsystem angepasst wurde. Im

Kern besteht diese in Abbildung 1.1 dargestellte Hardware-Plattform aus den im Folgenden genannten Komponenten. Als Trägerboard fungiert eine ISA-Steckkarte, die in jedem Rechner mit entsprechender Schnittstelle genutzt werden kann. Sie bietet Platz für bis zu neun so genannte Recheneinheiten, welche die Arbeitstiere des Systems darstellen und ein gestelltes SAT-Problem parallel lösen. Es handelt sich hierbei um Mikroprozessoren vom Typ *Microchip PIC17C43* [84] mit jeweils 64 kWord externem Speicher (1 Word entspricht 16 Bit), die mit 32 MHz Taktfrequenz betrieben werden. Die Kommunikation zwischen den Recheneinheiten wird durch eine zur Laufzeit rekonfigurierbare *Switch-Matrix* der Firma I-Cube [52] ermöglicht, an der die seriellen Schnittstellen aller PIC17C43 Mikroprozessoren direkt angeschlossen sind. Vereinfacht ausgedrückt handelt es sich bei diesem Baustein um eine Leitungsmatrix, bei der durch Setzen und Löschen von Verknüpfungspunkten beliebige I/O-Pins und somit beliebige Recheneinheiten miteinander verknüpft werden können. Die Konfiguration der Switch-Matrix wird durch einen separaten, mittig auf dem Trägerboard platzierten Kommunikationsprozessor vom Typ *Motorola MC68340* [39] gesteuert, der über 256 kByte externen Speicher verfügt und mit 16,78 MHz Taktfrequenz betrieben wird. Dieser ist ebenso wie die Recheneinheiten in Form eines eigenständigen Moduls auf das Trägerboard aufgesteckt und regelt über den ISA-Bus auch den gesamten Datenverkehr zum angeschlossenen Rechner.

Abbildung 1.1: Multiprozessorsystem

Die in PIChaff eingesetzten sequentiellen SAT-Prozeduren, die auf den verschiedenen Recheneinheiten des Multiprozessorsystems parallel ausgeführt werden, beinhalten alle elementaren Techniken, die ein modernes Verfahren heutzutage auszeichnen: eine effiziente Entscheidungsheuristik, einen *Boolean Constraint Propagation* Mechanismus auf Basis so genannter *Watched Literals*, die von zChaff bekannte Konflikt-Analyse gemäß des *1UIP*-Prinzips und damit einhergehend auch *Non-Chronological Backtracking*. Aus Speicher- und Performance-Gründen erfolgte die Umsetzung der SAT-Prozeduren vollständig in Maschinensprache, wobei an einigen Stellen zChaff als Ideengeber fungierte, was zusammen mit den Microchip PIC17C43 Mikroprozessoren der Recheneinheiten auch die Namensgebung erklärt.

Die Parallelisierung erfolgte gemäß eines *Master/Client-Modells*, bei dem der Kommunikationsprozessor als Master agiert, während die Recheneinheiten als Clients das gestellte Problem gemeinsam lösen. Abbildung 1.2 zeigt schematisch das Design von PIChaff sowie die Zuordnung der Funktionseinheiten zu den Komponenten des Multiprozessorsystems.

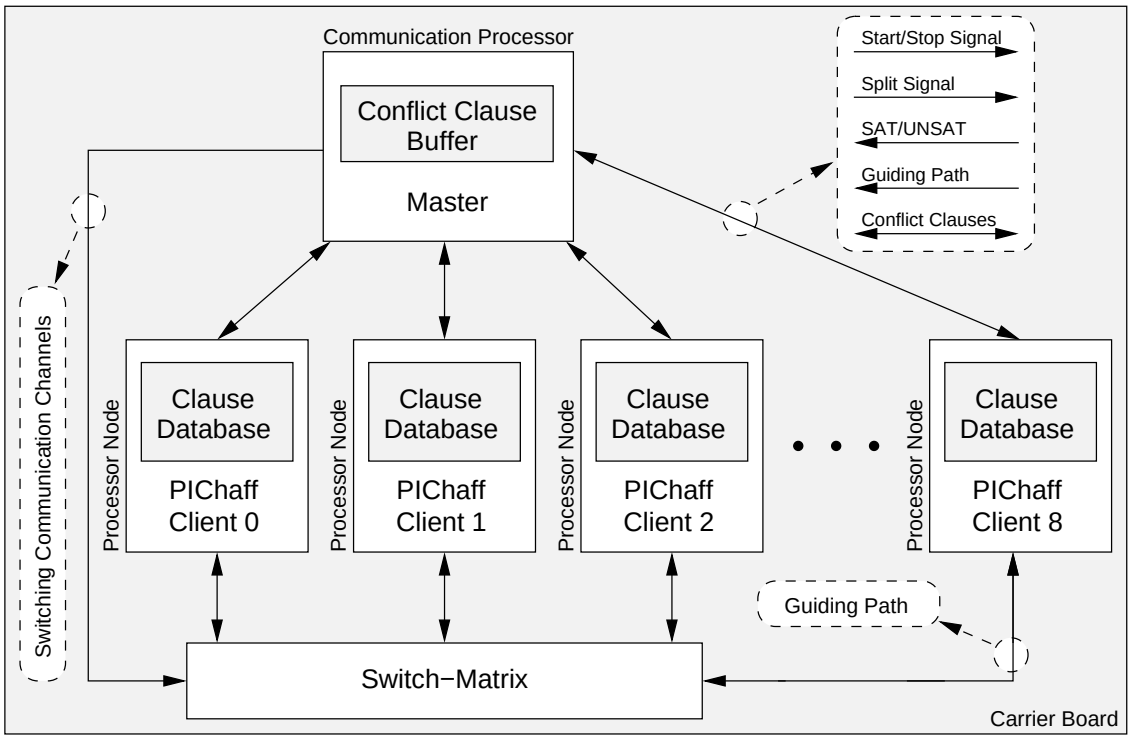


Abbildung 1.2: Design PIChaff

Der Master entscheidet dabei gegebenenfalls, welcher Client sein aktuelles Teilproblem in zwei disjunkte Bereiche aufteilt und einen dieser Bereiche über die Switch-Matrix an einen derzeit „inaktiven“ PIC17C43 Mikroprozessor abgibt. Weiterhin ist der Master-Prozess dafür verantwortlich, die von einem Client ermittelten relevanten Informationen über die gegebene Probleminstanz an die restlichen an der Suche beteiligten Clients weiterzuleiten. Im Bereich der SAT-Algorithmen erfolgt die Kodierung derartiger Informationen üblicherweise in Form so genannter *Konflikt-Klauseln*, die unerfüllbare (Teil-)Belegungen der Variablen charakterisieren, das heißt Zuweisungen an Variablen beschreiben, mit denen die gegebene Formel nicht erfüllt werden kann. Durch den Austausch geeigneter Konflikt-Klauseln kann der gesamte Suchprozess dahingehend optimiert werden, dass die Clients daran gehindert werden, Variablenbelegungen zu wählen, die bereits als unerfüllbar identifiziert wurden.

Die in [89, 90, 91, 92, 93] für verschiedene Entwicklungsstufen von PIChaff durchgeführten Experimente zeugen von einer erfolgreichen Implementierung. Im parallelen Betrieb konnte eine im Vergleich zum sequentiellen Szenario lineare Beschleunigung erzielt werden, so dass sich beim Einsatz aller neun Recheneinheiten (neun Clients) die Laufzeit zum Lösen einer Probleminstanz im Vergleich zur Laufzeit einer einzelnen Recheneinheit (ein Client) im Mittel auf ein Neuntel reduziert.

1.2 MiraXT

Mit MiraXT, dem zweiten in dieser Arbeit entwickelten parallelen SAT-Algorithmus, wird die hardwarenahe Programmierung verlassen. Zugleich verschiebt sich der Schwerpunkt der anvisierten Hardware-Plattformen von Systemen mit in ihren Ressourcen eingeschränkten Mikroprozessoren hin zu „klassischen“ Computern. Insbesondere werden Computer betrachtet, die intern über mehrere Prozessoren (Multiprozessorsysteme) beziehungsweise über Prozessoren mit mehreren CPU-Kernen (Dual-/Multi-Core Prozessoren) verfügen. Der Vorteil derartiger Architekturen ist darin zu sehen, dass alle Prozessoren beziehungsweise CPU-Kerne an einen gemeinsamen Speicher angebunden sind, was dazu genutzt werden kann, die Kommunikation zwischen den auf den einzelnen CPUs ausgeführten Prozessen mit Hilfe des Speichers und entsprechenden Datenstrukturen abzuwickeln.

MiraXT folgt einer threadbasierten Programmierung, die unter Zuhilfenahme der Funktionsbibliothek PThread [22] in C/C++ vorgenommen wurde. Der Zusatz „XT“ steht dabei abkürzend für „x Threads“ und deutet an, dass MiraXT im Gegensatz zur Vorgängerversion Mira [69, 70] in der Lage ist, ein gestelltes Problem mit mehreren Threads parallel zu lösen. Abbildung 1.3 zeigt schematisch das umgesetzte Konzept.

Wie einige andere SAT-Algorithmen auch, verfügt MiraXT über eine so genannte *Preprocessing*-Einheit. Diese hat zum Ziel, die zu lösende Probleminstanz im Vorfeld der eigentlichen Suche nach einer erfüllenden Belegung so zu vereinfachen, dass sich dadurch die Laufzeit des nachfolgenden Suchprozesses möglichst stark reduziert. Im Unterschied zu anderen parallelen SAT-Verfahren wie PaSAT [104] und ySAT [38], die ebenfalls auf einem Thread-Konzept aufbauen, zeichnet MiraXT aus, dass alle Threads auf einer einzigen Klauselmenge operieren, der so genannten *Shared Clause Database*. Jeder Thread legt die von ihm generierten Konflikt-Klauseln in der Shared Clause Database ab, so dass alle Threads neben dem Zugriff auf die „eigenen“ Klauseln insbesondere auch den direkten Zugriff auf Konflikt-Klauseln anderer Threads haben und damit unmittelbar von diesen profitieren können. Ein ausgeklügeltes System so genannter *Locks* zur Vergabe von exklusiven Schreibrechten gewährleistet die Datenkonsistenz der Shared Clause Database und minimiert zugleich die beim Einfügen neuer Konflikt-Klauseln in die Klauselmenge unvermeidbaren Wartezeiten einzelner Threads.

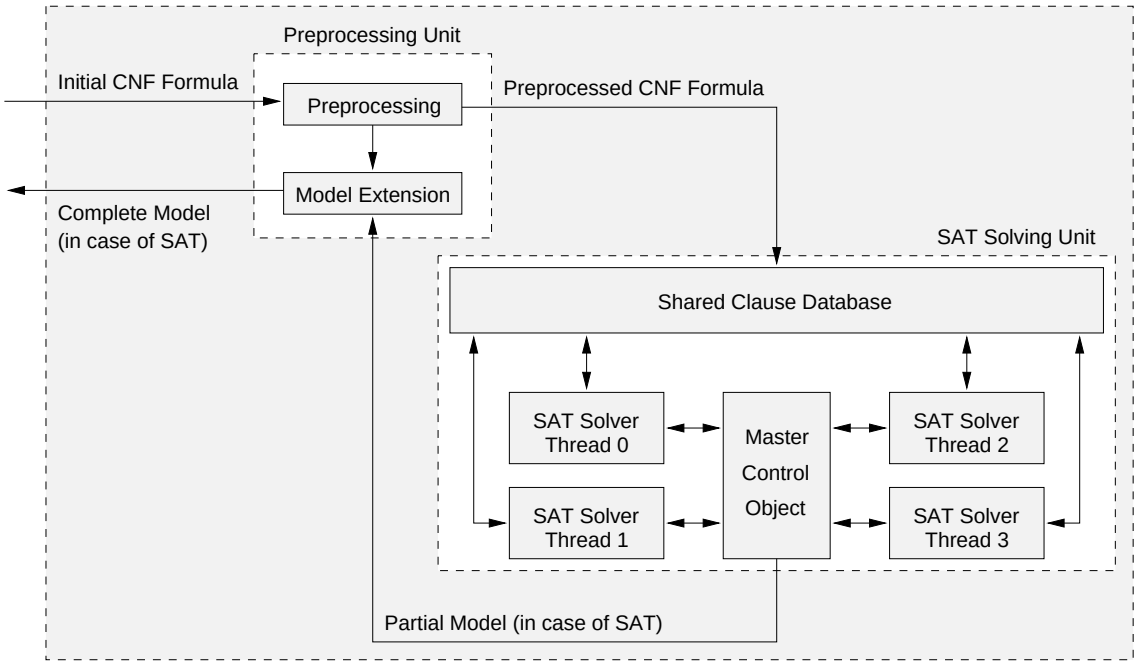


Abbildung 1.3: Design MiraXT

Anders als bei PIChaff wird in MiraXT kein Master/Client-Modell umgesetzt und auf einen separaten, aktiven Master-Prozess verzichtet. Stattdessen wird mit dem *Master Control Object* lediglich eine „passive“ Datenstruktur eingesetzt, mit der die Threads Statusinformationen, insbesondere aber auch noch zu analysierende Teilprobleme untereinander austauschen können. Letzteres ist derart realisiert, dass ein inaktiver Thread eine entsprechende Anfrage im Master Control Object hinterlegt, die von einem aktiven Thread gelesen und durch die Bereitstellung eines noch nicht bearbeiteten Bereichs seines eigenen Teilproblems beantwortet wird. Das dabei abgespaltete Teilproblem wird ebenfalls im Master Control Object gespeichert und kann von dort vom inaktiven Thread entgegengenommen werden.

Die bei der Lösung anerkannt schwieriger Probleminstanzen erzielten Ergebnisse demonstrieren eindrucksvoll das Potenzial von MiraXT. Bereits im sequentiellen Modus mit nur einem Thread ist MiraXT den als Referenz herangezogenen SAT-Algorithmen RSat, MiniSat2 und PicoSAT überlegen. Alle drei Verfahren gehören zu den aktuell leistungsstärksten sequentiellen SAT-Algorithmen. Durch die Verwendung mehrerer gemeinsam agierender Threads reduziert sich nicht nur die benötigte Laufzeit, zugleich erhöht sich auch die Anzahl der innerhalb eines vorgegebenen Zeitlimits gelösten Probleme. Die im parallelen Betrieb gegenüber der sequentiellen MiraXT-Variante erzielte Beschleunigung variiert je nach

Hardware-Plattform und Problemklasse. Im besten Fall konnte bei industriell relevanten Fragestellungen der Problemklasse *SAT 2007 Industrial*, gelöst auf einem Doppelprozessorsystem mit zwei *Dual-Core AMD Opteron 280* Prozessoren und gemittelt über alle 247 Instanzen dieser Kategorie, eine Beschleunigung um den Faktor 2,47 (zwei Threads) beziehungsweise 3,51 (vier Threads) erzielt werden. Die Ergebnisse werden durch die in [71] für eine andere Klasse von Problemstellungen durchgeführten Ergebnisse bestätigt.

1.3 PaMiraXT

Mit MiraXT wurde ein leistungsstarker, threadbasierter SAT-Algorithmus realisiert, der selbst auf Rechnern, die nur über eine CPU verfügen, eine sehr gute Performance zeigt. Gegenüber anderen *State-of-the-Art* Ansätzen bietet MiraXT zudem die Möglichkeit, zusätzlich vorhandene Prozessoren und CPU-Kerne, die bei sequentiellen SAT-Algorithmen ansonsten brach liegen, gewinnbringend in die Suche nach einer erfüllenden Belegung einzubinden. Zur Umsetzung des Thread-Konzepts ist es allerdings erforderlich, dass alle am Suchprozess beteiligten CPUs, und damit die darauf ausgeführten Threads, Zugriff auf einen gemeinsamen Speicher besitzen, was in Rechnernetzwerken, bei denen mehrere Computer per Ethernet-Verbindung miteinander verknüpft sind, nicht gegeben ist. MiraXT ist in derartigen Hardware-Umgebungen daher nur auf einem einzelnen Knoten eines Rechnernetzwerks ausführbar.

PaMiraXT überwindet diese Einschränkung durch ein zweistufiges Design, das sich wie folgt charakterisieren lässt: auf jedem am Suchprozess beteiligten Rechner eines Netzwerks wird MiraXT ausgeführt und zwar je nach Hardware-Ausstattung entweder in der sequentiellen Variante mit einem Thread oder im parallelen Betriebsmodus mit mehreren Threads. Die so gestarteten „Kopien“ von MiraXT werden analog zum Master/Client-Modell von PIChaff als Clients aufgefasst und unter der Regie eines separaten Master-Prozesses zum parallelen SAT-Algorithmus PaMiraXT zusammengeführt. Abbildung 1.4 illustriert das Konzept am Beispiel von vier Clients und dem sich als Mittler zwischen den Clients befindenden Master. Dieser ist verantwortlich für das Starten und Stoppen der Clients als auch das Weiterleiten von Teilproblemen und Konflikt-Klauseln auf der Ebene der Clients, das heißt zwischen verschiedenen MiraXT-Kopien. Die Kommunikation zwischen Master und Clients erfolgt über den Austausch von Nachrichten, das so genannte *Message Passing*, wobei zur Umsetzung auf das Software-Paket MPICH [48] zurückgegriffen wurde.

Zum Zweck des Datenaustauschs mit dem Master ist die in PaMiraXT für die Clients eingesetzte Variante von MiraXT um einen so genannten *MPI-Thread* erweitert worden. Wie Abbildung 1.5 zeigt, besitzt dieser analog zu den die eigentliche SAT-Prozedur ausführenden Threads (im Folgenden als SAT-Threads bezeichnet) einen Zugang zur Shared Clause Database. Diese Zugriffsmöglichkeit wird durch den MPI-Thread dazu genutzt, anhand der in der Klauseldatenbank „seines“ Clients enthaltenen Klauseln zu entscheiden, welche da-

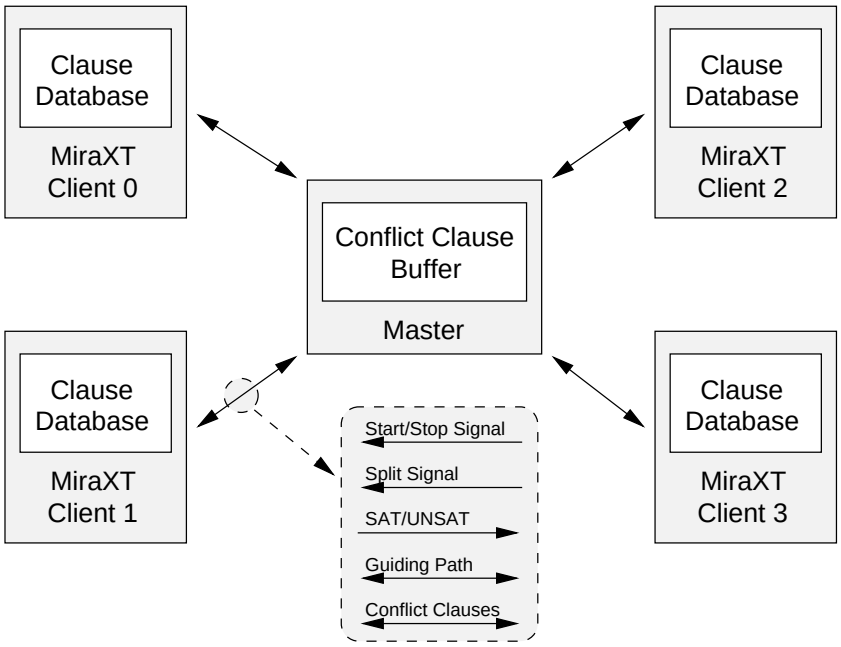


Abbildung 1.4: Design PaMiraXT

von potenziell auch für andere Clients und deren SAT-Threads relevant sein könnten. Die Menge der dahingehend positiv bewerteten Klauseln wird durch den MPI-Thread zunächst an den Master-Prozess und von diesem dann an die restlichen Clients weitergeleitet. Auf diesem Weg vom Master erhaltene Konflikt-Klauseln werden von den MPI-Threads der jeweils eigenen Shared Clause Database hinzugefügt und sind dadurch allen SAT-Threads des entsprechenden Clients direkt zugänglich.

Ein Austausch von Teilproblemen auf der Ebene der Clients wird in PaMiraXT immer dann vollzogen, wenn alle SAT-Threads eines Clients inaktiv sind, das heißt, dass diese das ursprünglich an sie übertragene Teilproblem komplett abgearbeitet haben (allerdings ohne eine erfüllende Belegung ermittelt zu haben). In diesen Fällen wird, durch den Master-Prozess initiiert, von einem aktiven Client ein noch unbearbeitetes Teilproblem angefordert und an den inaktiven Client weitergeleitet, wo es von den dortigen SAT-Threads gelöst wird.

Auf Seiten der PaMiraXT-Clients wird dieser Austausch ebenfalls von den MPI-Threads gesteuert, die zu diesem Zweck auch einen Zugang zum Master Control Object besitzen (siehe Abbildung 1.5). Auf diesem Weg sind die MPI-Threads in der Lage, ein vom Master-Prozess erhaltenes Teilproblem eines anderen Clients an die eigenen SAT-Threads abzugeben. Umgekehrt besteht dadurch ebenso die Möglichkeit, dass ein MPI-Thread einen noch

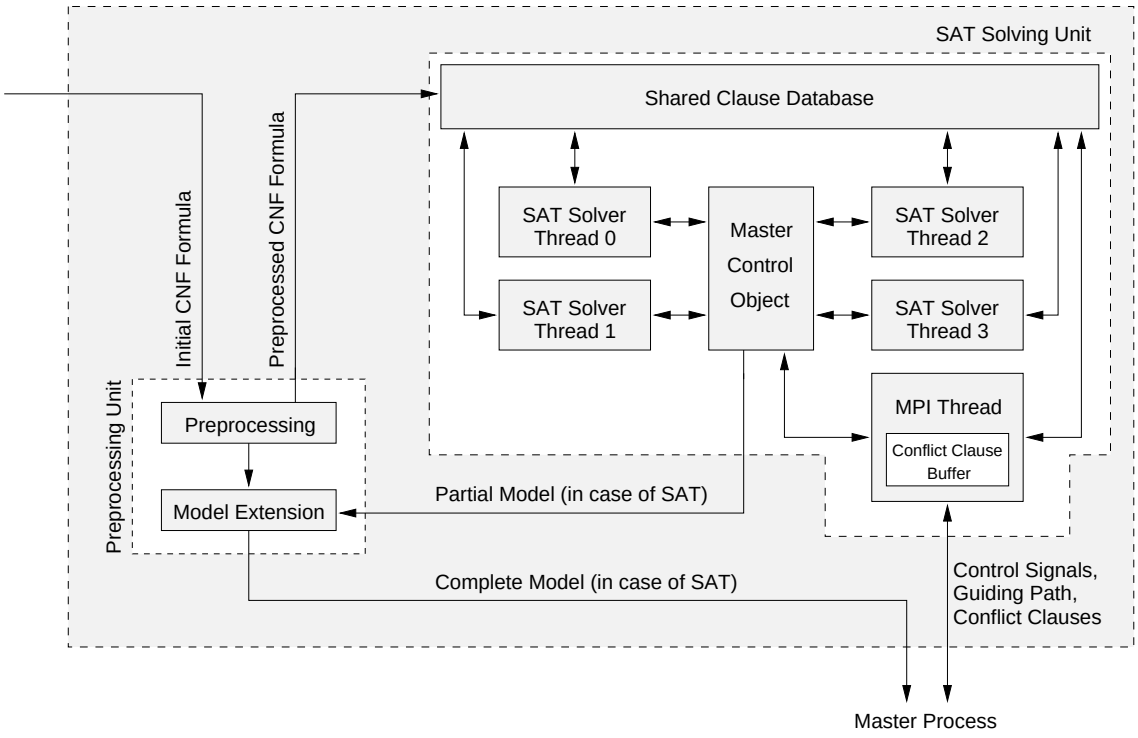


Abbildung 1.5: Design der in PaMiraXT eingesetzten Variante von MiraXT

unbearbeiteten Bereich des von seinen SAT-Threads aktuell untersuchten Teilproblems entgegennimmt und an den Master-Prozess weiterleitet, der das erhaltene Teilproblem dann an einen inaktiven Client transferiert.

In verschiedenen Konfigurationen wurde PaMiraXT intensiv getestet. Im Vergleich zu MiraXT konnte dabei gezeigt werden, dass insbesondere bei zeitintensiven Probleminstanzen, bei denen die vergleichsweise langsame Kommunikation per Message Passing nicht ins Gewicht fällt, zusätzliche SAT-Threads, wenngleich verteilt auf verschiedene Clients, für eine weitere Erhöhung der Performance sorgen.

1.4 Gliederung der Arbeit

Nachfolgend wird der Aufbau der Arbeit dargestellt. Im zweiten Kapitel werden die Grundlagen gelegt und die benötigten Begriffe der Aussagenlogik eingeführt. Darauf aufbauend wird das NP-vollständige Erfüllbarkeitsproblem der Aussagenlogik (kurz: SAT-Problem) definiert. Weiterhin wird mit der Resolution eine syntaktische Umformungsvorschrift vorgestellt. Abschließend wird anhand des Davis-Putnam (DP) beziehungsweise des Davis-Logemann-Loveland (DLL) Verfahrens aufgezeigt, wie mit Hilfe der Resolution die Frage

der Erfüllbarkeit von Instanzen des SAT-Problems entschieden werden kann [28, 27].

Kapitel 3 behandelt mögliche Einsatzgebiete von SAT-Algorithmen. Stellvertretend für eine Vielzahl weiterer Anwendungen werden mit *Automatic Test Pattern Generation* und *Combinational Equivalence Checking* zwei Teildisziplinen aus dem Bereich des rechnergestützten Schaltkreisentwurfs herausgegriffen und näher beleuchtet. Es wird unter anderem skizziert, wie sich die dortigen Fragestellungen so als Erfüllbarkeitsproblem der Aussagenlogik formulieren lassen, dass eine von einem SAT-Algorithmus ermittelte Lösung auch eine Lösung für das Ausgangsproblem darstellt.

Der Fokus dieser Arbeit liegt auf *vollständigen* SAT-Verfahren im Stil der klassischen DLL-Prozedur, das heißt Ansätzen, die neben der Erfüllbarkeit auch die Unerfüllbarkeit einer Probleminstanz belegen können (ein geeignet großes Zeitlimit vorausgesetzt). Kapitel 4 widmet sich detailliert den über das Grundgerüst der DLL-Prozedur hinausgehenden, leistungssteigernden Techniken. Ein Schwerpunkt liegt auf den drei Kernfunktionen vollständiger SAT-Algorithmen: der Entscheidungsheuristik, der so genannten *Boolean Constraint Propagation* sowie der Konflikt-Analyse. Weiterhin werden das mögliche *Preprocessing* der zu untersuchenden Probleminstanz, das Löschen von Konflikt-Klauseln und das Konzept der Neustarts diskutiert. Dabei wird an den entsprechenden Stellen angedeutet, welche der Konzepte und Methoden, angepasst an die jeweilige Hardware-Plattform, auch in PIChaff, MiraXT und PaMiraXT integriert wurden.

Im fünften Kapitel wird ein Überblick über parallele SAT-Algorithmen gegeben. Insgesamt werden vier unterschiedliche Ansätze vorgestellt, bei denen die Parallelisierung, wie auch bei den in dieser Arbeit entwickelten Verfahren, darauf beruht, dass die zu lösende Probleminstanz in disjunkte Teile aufgeteilt wird, die dann parallel bearbeitet werden. Im Hinblick auf die Implementierung von PIChaff, MiraXT und PaMiraXT steht dabei eine Analyse der eingesetzten Kommunikationsmodelle im Vordergrund, von denen maßgeblich die Art des Informationsaustauschs zwischen den Prozessen beziehungsweise Threads und damit auch das Gesamtdesign des parallelen SAT-Algorithmus abhängt.

Die technischen Aspekte des am Lehrstuhl für Rechnerarchitektur entwickelten Multiprozessorsystems werden in Kapitel 6 erläutert. Neben der Vorstellung der Hardware-Komponenten liegt ein besonderes Augenmerk auf den für die Kommunikation zwischen den Mikroprozessoren zur Verfügung stehenden Datenpfaden. Aufbauend darauf wird in Kapitel 7 die Realisierung von PIChaff aufgezeigt. Neben einer Anwendung auf Seiten des angeschlossenen Rechners, mit dem das Multiprozessorsystem mit Daten versorgt wird, stehen die für einen reibungslosen Ablauf benötigten Routinen des Kommunikationsprozessors und die auf den Recheneinheiten ausgeführte sequentielle SAT-Prozedur im Mittelpunkt.

Kapitel 8 behandelt die Realisierung von MiraXT und beleuchtet insbesondere die *Shared Clause Database* und das *Master Control Object*, deren Implementierung von entscheiden-

der Bedeutung für die Performance von MiraXT ist. Weiterhin wird die von den Threads ausgeführte SAT-Prozedur thematisiert und es wird beschrieben, wie die in Kapitel 4 eingeführten Methoden moderner SAT-Algorithmen integriert wurden. Zahlreiche Versuchsreihen runden das Kapitel ab.

Daran anschließend wird in Kapitel 9 mit PaMiraXT eine Erweiterung von MiraXT vorgestellt. Aufbauend auf den Arbeiten des vorherigen Kapitels werden die MiraXT-Clients sowie das diesen übergeordnete Master/Client-Modell erläutert. Für unterschiedliche Konfigurationen von PaMiraXT, die Anzahl an Clients und die Zahl der jeweils pro Client gestarteten SAT-Threads betreffend, wird das Leistungsvermögen dieser Variante von MiraXT analysiert und den Resultaten aus Kapitel 8 gegenübergestellt.

Eine Zusammenfassung und Bewertung der erzielten Ergebnisse in Kapitel 10 schließen die Arbeit ab. Es sei darauf hingewiesen, dass einzelne Aspekte der Entwicklung von PIChaff, MiraXT und PaMiraXT in folgenden Publikationen thematisiert wurden:

- M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.
- M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. In *JSAT Special Issue on SAT/CP Integration*, 2007.
- E. Abraham, T. Schubert, B. Becker, M. Fränzle, and C. Herde. Parallel SAT Solving in Bounded Model Checking. In *5th International Workshop on Parallel and Distributed Methods in Verification*, 2006.
- T. Schubert, M. Lewis, and B. Becker. Accelerating Boolean SAT Engines Using Hyper-Threading Technology. In *3rd Asian Applied Computing Conference*, 2005.
- T. Schubert, M. Lewis, and B. Becker. PaMira - A Parallel SAT Solver with Knowledge Sharing. In *6th International Workshop on Microprocessor Test and Verification*, 2005.
- M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.
- T. Schubert and B. Becker. Knowledge Sharing in a Microcontroller based Parallel SAT Solver. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2005.
- T. Schubert and B. Becker. Lemma Exchange in a Microcontroller based Parallel SAT Solver. In *IEEE Symposium on VLSI*, 2005.

- T. Schubert and B. Becker. Parallel SAT Solving with Microcontrollers. In *2nd Asian Applied Computing Conference*, 2004.
- T. Schubert and B. Becker. PICHAFF^2 - A Hierarchical Parallel SAT Solver. In *5th International Workshop on Microprocessor Test and Verification*, 2004.
- M. Lewis, T. Schubert, and B. Becker. Early Conflict Detection Based BCP for SAT Solving. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.
- T. Schubert and B. Becker. A Distributed SAT Solver for Microcontrollers. In *7th Workshop on Parallel Systems and Algorithms*, 2004.
- M. Lewis, T. Schubert, and B. Becker. Early Conflict Detection Based SAT Solving. In *GI/ITG/GMM Workshop on „Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen“*, 2004.
- T. Schubert and B. Becker. PICHAFF: A Distributed SAT Solver for Microcontrollers. In *Work in Progress Session held in connection with the 29th Euromicro Conference*, 2003.
- T. Schubert, E. Mackensen, N. Drechsler, R. Drechsler, and B. Becker. Specialized Hardware for Implementation of Evolutionary Algorithms. In *4th International Workshop on Boolean Problems*, 2000.
- R. Drechsler, N. Drechsler, E. Mackensen, T. Schubert, and B. Becker. Design Reuse by Modularity: A Scalable Dynamical (Re)Configurable Multiprocessor System. In *26th Euromicro Conference*, 2000.
- T. Schubert, E. Mackensen, N. Drechsler, R. Drechsler, and B. Becker. Specialized Hardware for Implementation of Evolutionary Algorithms. In *Genetic and Evolutionary Computing Conference*, 2000.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen für die vorliegende Arbeit gelegt. Es wird eine Einführung in die Aussagenlogik gegeben und das Erfüllbarkeitsproblem der Aussagenlogik definiert. Des Weiteren werden aufbauend auf der syntaktischen Umformungsregel der Resolution mit dem Davis-Putnam (DP) und dem Davis-Logemann-Loveland (DLL) Algorithmus zwei Verfahren erläutert, mit Hilfe derer die Frage der Erfüllbarkeit bei derartigen Probleminstanzen beantwortet werden kann.

Die dargestellten Aspekte sind bewusst ausführlich gehalten, da sie für weite Teile dieser Arbeit von erheblicher Bedeutung sind. So wird sich beispielsweise in Kapitel 4 zeigen, dass alle modernen SAT-Verfahren auf Basis des DLL-Algorithmus während der Konflikt-Analyse auf die Resolutionsregel zurückgreifen. Gleiches gilt je nach Implementierung auch für das so genannte *Preprocessing*, das zum Ziel hat, eine gegebene Probleminstanz so zu modifizieren, dass sich die Laufzeit des anschließenden Suchprozesses möglichst stark verringert.

2.1 Aussagenlogik

In der Aussagenlogik wird der Wahrheitswert von Aussagen untersucht. Es gilt festzustellen, ob ein bestimmter durch eine Formel beschriebener Sachverhalt *wahr* oder *falsch* ist. Formeln der Aussagenlogik bestehen auf der einen Seite aus nicht weiter zerlegbaren, atomaren Grundbausteinen, die für einfache Aussagen stehen und entweder wahr oder falsch sind. Auf der anderen Seite können diese atomaren Formeln, im Folgenden als Variablen bezeichnet, mit den logischen Operatoren UND ($\wedge$) und ODER ($\vee$) sowie der Negation ($\neg$) zur Beschreibung komplexer Sachverhalte verknüpft werden. In Anlehnung an [36, 97] kann die Syntax der Aussagenlogik wie folgt definiert werden:

Definition 2.1 (Syntax der Aussagenlogik)
Sei eine Menge von Variablen $x_1, \ldots, x_n$ gegeben. Eine Formel der Aussagenlogik ist durch folgenden induktiven Prozess definiert:

1. *Jede Variable x_i ist eine Formel.*
2. *Für alle Formeln F_1 und F_2 sind auch*

   - *die Konjunktion* $(F_1 \wedge F_2)$ *und*
   - *die Disjunktion* $(F_1 \vee F_2)$ *Formeln der Aussagenlogik.*

3. *Für jede Formel* F *ist auch die Negation* $(\neg F)$ *eine Formel der Aussagenlogik.*

4. *Zur Menge der Formeln der Aussagenlogik gehören nur die Formeln, die mittels einer endlich oft durchgeführten Anwendung der drei erstgenannten Regeln gebildet werden können.*

Um Schreibaufwand zu sparen und um mehr Übersichtlichkeit zu erhalten, wird im weiteren Verlauf der Arbeit eine *abkürzende Schreibweise* verwendet, bei der weitgehend auf Klammerpaare verzichtet wird. Dazu wird folgende Prioritätsliste bezüglich der Auswertungsreihenfolge der Operatoren festgelegt (wobei sich die eigentliche Auswertung der Operatoren aus der in Definition 2.2 gegebenen Semantik der Aussagenlogik ergibt):

$$\text{„}\neg\text{“} \rightharpoonup \text{„}\wedge\text{“} \rightharpoonup \text{„}\vee\text{“}$$

Dabei bedeutet $a \rightharpoonup b$, dass die Auswertung des Operators a eine höhere Priorität besitzt als die Auswertung des Operators b oder anders formuliert, dass a stärker bindet als b. Weiterhin binden Klammerpaare stärker als jeder Operator. Mit Hilfe dieser Regeln können Klammerpaare immer dann weggelassen werden, wenn das Entfernen von Klammern die Auswertungsreihenfolge der Operatoren nicht verändert. Beispielsweise sind $F = (x_1 \wedge (\neg(x_2 \vee (x_3 \vee x_4))))$ und $F = x_1 \wedge \neg(x_2 \vee x_3 \vee x_4)$ in abkürzender Schreibweise gleichwertige Darstellungsformen für F.

Weiterhin werden folgende in der Aussagenlogik geläufige Schreib- und Sprechweisen vereinbart:

- Ein *Literal* L ist entweder eine Variable $(L = x_i)$ oder deren Negation $(L = \neg x_i)$. Im ersten Fall spricht man von einem *positiven*, im zweiten Fall von einem *negativen* Literal.

- Die Negation $\neg L$ eines Literals L wird definiert als

$$\neg L = \begin{cases} \neg x_i \text{ , falls } L = x_i, \\ x_i \text{ , falls } L = \neg x_i. \end{cases}$$

- Eine Formel $C = (L_1 \vee \ldots \vee L_k)$ mit den Literalen $L_1, \ldots, L_k$ wird auch als *Klausel* bezeichnet. Im Folgenden wird davon ausgegangen, dass, sofern nicht anders angegeben, die Literale $L_1, \ldots, L_k$ stets paarweise verschieden sind:

$$\forall i, j \in \{1, \ldots, k\} \text{ mit } i \neq j \text{ gilt: } L_i \neq L_j.$$

- Eine *Unit Clause* ist eine Klausel, die aus genau einem Literal besteht.

Nach der rein syntaktischen Definition, die das Aussehen von Formeln der Aussagenlogik festlegt, wird nun die Semantik definiert, um entscheiden zu können, wann eine Formel wahr oder falsch ist.

Definition 2.2 (Semantik der Aussagenlogik)
Eine Belegung $\mathcal{A}_x : \{x_1, \ldots, x_n\} \to \{wahr, falsch\}$ *ist eine Abbildung, die allen Variablen* $x_1, \ldots, x_n$ *einer Formel der Aussagenlogik den Wahrheitswert wahr oder falsch zuordnet.* $\mathcal{A}_x$ *wird erweitert zur Abbildung* $\mathcal{A} : \{F \mid F \text{ aussagenlogische Formel}\} \to \{wahr, falsch\}$, *die jeder Formel* F *der Aussagenlogik gemäß den nachfolgenden Regeln einen Wahrheitswert der Menge* $\{wahr, falsch\}$ *zuweist:*

1. *Für jede in* F *enthaltene Variable* x_i *gilt:* $\mathcal{A}(x_i) = \mathcal{A}_x(x_i)$.
2. *Für alle Teilformeln* F_1 *und* F_2 *von* F *gilt:*
   - $\mathcal{A}(F_1 \wedge F_2) = wahr \Leftrightarrow \mathcal{A}(F_1) = wahr$ *und* $\mathcal{A}(F_2) = wahr$.
   - $\mathcal{A}(F_1 \vee F_2) = wahr \Leftrightarrow \mathcal{A}(F_1) = wahr$ *oder* $\mathcal{A}(F_2) = wahr$.
3. *Für jede Teilformel* F' *von* F *gilt:* $\mathcal{A}(\neg F') = wahr \Leftrightarrow \mathcal{A}(F') = falsch$.

Ist nur eine Teilbelegung der Variablen einer Formel gegeben, so werden alle Variablen, denen kein Wahrheitswert zugewiesen wurde, als *frei* bezeichnet. Die dazu korrespondierenden positiven als auch negativen Literale sind dann *unbelegt*.

Ausgehend von der Belegung der Variablen kann gemäß Definition 2.2 sowie der zuvor festgelegten Prioritätsliste bezüglich der Auswertungsreihenfolge der Operatoren der Wahrheitswert einer Formel bestimmt werden, wobei die Wahrheitswerte *falsch* und *wahr* üblicherweise durch 0 und 1 ersetzt werden (Boolesche Wertemenge $\mathbb{B} = \{0, 1\}$).

Beispiel 2.1
Sei die Formel $F = x_1 \vee (x_2 \wedge \neg x_3)$ *sowie die Belegung* $\mathcal{A}$ *mit* $\mathcal{A}(x_1) = 0$, $\mathcal{A}(x_2) = 1$ *und* $\mathcal{A}(x_3) = 0$ *gegeben. Der Wahrheitswert von* F *lässt sich wie folgt bestimmen:*

$$\begin{aligned}\mathcal{A}(F) &= \mathcal{A}(\mathcal{A}(x_1) \vee \mathcal{A}(\mathcal{A}(x_2) \wedge \mathcal{A}(\neg x_3))) \\ &= \mathcal{A}(0 \vee \mathcal{A}(1 \wedge 1)) \\ &= \mathcal{A}(0 \vee 1) \\ &= 1\end{aligned}$$

Definition 2.3 (Erfüllbarkeit)
Eine Formel F *der Aussagenlogik ist genau dann erfüllbar, wenn eine Belegung* $\mathcal{A}$ *mit* $\mathcal{A}(F) = 1$ *existiert. Als Sprechweise wird vereinbart, dass eine derartige Belegung, die auch als Modell für* F *bezeichnet wird, die Formel* F *erfüllt, dargestellt durch* $\mathcal{A} \models F$. *Existiert hingegen keine Belegung* $\mathcal{A}$ *mit* $\mathcal{A}(F) = 1$, *so ist* F *unerfüllbar. Für alle Belegungen* $\mathcal{A}$ *gilt dann:* $\mathcal{A} \not\models F$.

Beispiel 2.2
Die in Beispiel 2.1 angegebene Formel F ist offensichtlich erfüllbar. Im Gegensatz dazu ist $G = x_1 \wedge \neg x_1$ unerfüllbar, da G durch keine Belegung der Variablen x_1 erfüllt werden kann.

Sofern aus dem Kontext ersichtlich, wird im Folgenden anstelle der Schreibweise $\mathcal{A}(x_i) = w$ mit $w \in \{0, 1\}$ für eine gegebene Variable x_i nur die Kurzform $x_i = w$ verwendet und eine entsprechende Belegung $\mathcal{A}$ implizit angenommen.

Alle in dieser Arbeit betrachteten Verfahren zum Lösen von Instanzen des im folgenden Abschnitt definierten Erfüllbarkeitsproblems der Aussagenlogik akzeptieren als Eingabe nur Formeln, die eine bestimmte syntaktische Struktur aufweisen: die konjunktive Normalform.

Definition 2.4 (konjunktive Normalform)
Eine Formel F der Aussagenlogik ist genau dann in konjunktiver Normalform, wenn sie aus der Konjunktion von Klauseln besteht:

$$F = \bigwedge_{j=1}^{m} C_j \qquad \textit{mit } C_1, \ldots, C_m \textit{ Klauseln}$$

Aus der Definition ergibt sich, dass eine Formel F in konjunktiver Normalform genau dann erfüllbar ist, wenn eine Belegung der in F enthaltenen Variablen existiert, die alle Klauseln erfüllt. Existiert keine derartige Belegung, so ist F unerfüllbar.

In Anlehnung an die englische Übersetzung der konjunktiven Normalform, *Conjunctive Normal Form (CNF)*, hat sich der Begriff *CNF-Formel* eingebürgert und wird auch in dieser Arbeit als Standard für Formeln in konjunktiver Normalform verwendet.

Definition 2.5 (Äquivalenz)
Zwei Formeln F und G der Aussagenlogik sind genau dann (logisch) äquivalent, dargestellt durch $F \equiv G$, wenn für alle zu F und G passenden Belegungen $\mathcal{A}$ gilt: $\mathcal{A}(F) = \mathcal{A}(G)$.

Durch Induktion über den Formelaufbau kann gezeigt werden, dass sich jede aussagenlogische Formel in eine äquivalente CNF-Darstellung überführen lässt [97], die konjunktive Normalform schränkt die Ausdruckskraft der Aussagenlogik somit nicht ein. Folgende Vorgehensweise kann zur Umformung einer beliebigen Formel F in eine äquivalente CNF-Darstellung angewendet werden:

1. Ersetze in F jede Teilformel der Form

   $\neg\neg F_1$ durch F_1,

   $\neg(F_1 \wedge F_2)$ durch $(\neg F_1 \vee \neg F_2)$,

   $\neg(F_1 \vee F_2)$ durch $(\neg F_1 \wedge \neg F_2)$,

   bis derartige Teilformeln in F nicht mehr existieren.

2. Ersetze in F jede Teilformel der Form

   $(F_1 \vee (F_2 \wedge F_3))$ durch $((F_1 \vee F_2) \wedge (F_1 \vee F_3))$,

   $((F_1 \wedge F_2) \vee F_3)$ durch $((F_1 \vee F_3) \wedge (F_2 \vee F_3))$,

   bis derartige Teilformeln in F nicht mehr vorkommen.

Die resultierende Formel liegt dann in konjunktiver Normalform vor und kann eventuell noch weiter vereinfacht werden. So können etwa Klauseln, die ein Literal L_i sowohl in positiver (L_i) als auch negativer Polarität ($\neg L_i$) enthalten, entfernt werden, da sie von allen Belegungen $\mathcal{A}$ erfüllt werden. Man spricht in diesem Fall von einer *Tautologie*. Ein entscheidender Nachteil der skizzierten äquivalenzerhaltenden Transformation liegt in der Tatsache, dass die entstehende CNF-Formel exponentiell größer als die Ausgangsformel sein kann, wie der aus [64] entnommene Satz 2.1 zeigt.

Definition 2.6 (Größe einer Formel)
Die Größe einer Formel F, gekennzeichnet durch $|F|$, sei definiert als die Anzahl aller in F vorkommenden Operatoren $\Diamond$ mit $\Diamond \in \{\wedge, \vee, \neg\}$.

Satz 2.1
Es existieren Formeln der Größe $(2 \cdot m - 1)$, für die jede äquivalente Formel in konjunktiver Normalform die Größe $(m \cdot 2^m - 1)$ aufweist.

Beweis: Man betrachte Formeln der Bauart

$$F_m = \bigvee_{j=1}^{m} (L_{j,1} \wedge L_{j,2})$$

mit paarweise verschiedenen, in diesem Fall nur positiv auftretenden Literalen

$$L_{1,1}, L_{1,2}, \ldots, L_{m,1}, L_{m,2}.$$

Die Größe derartiger Formeln beträgt offensichtlich $(2 \cdot m - 1)$. Eine minimale äquivalente Formel F'_m in konjunktiver Normalform hat die Form

$$F'_m = \bigwedge_{k_1,\ldots,k_m \in \{1,2\}} (L_{1,k_1} \vee \ldots \vee L_{m,k_m})$$

mit 2^m Klauseln. Für die Konjunktion der Klauseln werden $(2^m - 1)$ UND-Verknüpfungen benötigt. Da jede Klausel aus m Literalen besteht, was jeweils $(m-1)$ ODER-Verknüpfungen erforderlich macht, gilt insgesamt für die Größe von F'_m:

$$|F'_m| = 2^m - 1 + 2^m \cdot (m-1) = m \cdot 2^m - 1.$$

■

Eine alternative Methode der Überführung einer Formel der Aussagenlogik in eine CNF-Darstellung wird in Kapitel 3.1 erläutert: die Tseitin-Transformation [114]. Dabei handelt es sich um das Standardverfahren, eine durch einen Schaltkreis repräsentierte Funktion in eine CNF-Formel zu überführen. Der Vorteil dieser Vorgehensweise liegt darin, dass die Größe der entstehenden Formel in CNF-Darstellung stets linear in der Anzahl der Grundgatter des Schaltkreises und somit linear in der Größe der durch den Schaltkreis berechneten Funktion ist.[1] Bedingt durch die während der Transformation eingeführten zusätzlichen Hilfsvariablen ist anstelle der Äquivalenz allerdings lediglich die nachfolgend definierte Erfüllbarkeitsäquivalenz gewährleistet.

Definition 2.7 (Erfüllbarkeitsäquivalenz)
Zwei aussagenlogische Formeln F und G sind genau dann erfüllbarkeitsäquivalent, wenn gilt: F erfüllbar $\Leftrightarrow$ G erfüllbar.

2.2 Erfüllbarkeitsproblem der Aussagenlogik

Aufbauend auf den zuvor eingeführten Begriffen wird nun das Erfüllbarkeitsproblem der Aussagenlogik definiert. Wie bereits erwähnt, haben sich in der Literatur das englische Pendant *Boolean Satisfiability Problem* sowie die Abkürzungen *SAT* beziehungsweise *SAT-Problem* als Begriffe durchgesetzt.

Definition 2.8 (SAT-Problem)
Sei eine Formel F der Aussagenlogik in konjunktiver Normalform gegeben. Die zu beantwortende Fragestellung lautet: ist F erfüllbar, das heißt, existiert eine Belegung $\mathcal{A}$ für die in F enthaltenen Variablen, so dass $\mathcal{A}(F) = 1$ gilt?

Die hier vorgenommene Beschränkung auf Formeln in konjunktiver Normalform ist nicht zwingend erforderlich, sondern dadurch motiviert, dass alle gängigen SAT-Verfahren auf Basis des in Abschnitt 2.5 vorgestellten Davis-Logemann-Loveland Algorithmus zum Lösen derartiger Problemstellungen nur CNF-Formeln verarbeiten können.

Wie 1971 von Cook gezeigt werden konnte, gehört das SAT-Problem zur Klasse der NP-vollständigen Probleme [25]. Unter der Annahme NP $\neq$ P ist folglich nicht mit der Entwicklung eines effizienten Algorithmus mit stets polynomieller Laufzeit zu rechnen.

Neben dem in Definition 2.8 angegebenen allgemeinen Fall ist auch das 3SAT-Problem, bei dem jede Klausel der zu untersuchenden Formel aus maximal drei Literalen besteht, bereits NP-vollständig [117]. Die für die Reduktion des allgemeinen SAT-Problems auf 3SAT in polynomieller Zeit durchzuführende Transformation einer CNF-Formel in eine erfüllbarkeitsäquivalente Darstellung mit maximal drei Literalen pro Klausel ist denkbar

[1] Unter der Voraussetzung, dass für die CNF-Darstellung der durch das jeweilige Grundgatter repräsentierten Funktion nur konstant viele Klauseln benötigt werden (siehe Abschnitt 3.1).

einfach. Für jede Klausel $(L_1 \vee \ldots \vee L_k)$, die aus vier oder mehr Literalen besteht ($k \geq 4$), wird folgende Umformung durchgeführt: ersetze $(L_1 \vee \ldots \vee L_k)$ durch $k-2$ neue Klauseln unter Verwendung von $k-3$ neuen Variablen $h_1, \ldots, h_{k-3}$, die nur in diesen Klauseln vorkommen. Die neuen Klauseln haben dabei die folgende Form:

- $(L_1 \vee L_2 \vee h_1)$
- $(\neg h_t \vee L_{t+2} \vee h_{t+1})$ für $t = 1, \ldots, k-4$
- $(\neg h_{k-3} \vee L_{k-1} \vee L_k)$

Die Anwendung dieser Regel erhält offensichtlich die Erfüllbarkeitsäquivalenz zwischen der Ausgangs- und der auf diesem Weg erzeugten 3SAT-Formel, wie dies auch das folgende Beispiel verdeutlicht.

Beispiel 2.3
Sei die Formel $F = (L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5)$ mit paarweise verschiedenen Literalen $L_1, \ldots, L_5$ gegeben, die in eine 3SAT-Formel überführt werden soll. Die Anwendung der zuvor angegebenen Regel erzeugt eine Formel F', bei der jede Klausel aus genau drei Literalen besteht:

$$F' = (L_1 \vee L_2 \vee h_1) \wedge (\neg h_1 \vee L_3 \vee h_2) \wedge (\neg h_2 \vee L_4 \vee L_5)$$

Wie man leicht sieht, ist die Ausgangsformel F bei allen Belegungen, bei denen mindestens eines der Literale $L_1, \ldots, L_5$ den Wahrheitswert 1 annimmt, erfüllt. Ausgehend von einer derartigen Belegung lassen sich h_1 beziehungsweise h_2 stets so mit einem Wahrheitswert versehen, dass auch F' erfüllt ist. Lediglich bei der Belegung $L_1 = \ldots = L_5 = 0$ ist F nicht erfüllt, was in diesem Fall auch für F' gilt, da entweder h_1 oder h_2 zeitgleich die Wahrheitswerte 0 und 1 annehmen müsste, um alle Klauseln von F' zu erfüllen. Die Erfüllbarkeitsäquivalenz ist somit gezeigt. Die Situation ist analog bei Klauseln mit vier oder mehr als fünf Literalen.

Die Bedeutung des 3SAT-Problems aus theoretischer Sicht ergibt sich aus der Tatsache, dass durch eine Reduktion des 3SAT-Problems auf diverse andere Fragestellungen auch deren Zugehörigkeit zur Klasse der NP-vollständigen Probleme gezeigt werden konnte. Exemplarisch seien die Probleme *Clique*, *Travelling Salesman*, *Knapsack* und *Bin Packing* genannt [98, 117].

Dem gegenüber stehen allerdings auch Spezialfälle des SAT-Problems, bei denen die Frage der Erfüllbarkeit effizient entschieden werden kann. Das 2SAT-Problem, bei dem jede Klausel aus maximal zwei Literalen besteht, ist ein solcher Spezialfall [118]. Ebenso ist der Erfüllbarkeitstest für Hornformeln, bei denen jede Klausel höchstens ein positives Literal enthält, mit polynomiellem Zeitaufwand durchführbar [83, 97].

2.3 Resolution

Bei der *Resolution* handelt es sich um eine syntaktische Umformungsvorschrift, mit deren Hilfe für Instanzen des in Definition 2.8 angegebenen SAT-Problems die (Un-)Erfüllbarkeit nachgewiesen werden kann. Eine Möglichkeit besteht darin, eine in konjunktiver Normalform gegebene Formel F solange per Resolution um neue Klauseln, die so genannten *Resolventen*, zu erweitern, bis die leere Klausel erzeugt werden kann. Gelingt dies, so ist die Formel unerfüllbar, andernfalls ist F erfüllbar.

Als Vorarbeit werden einige Notationen vereinbart:

- Eine Klausel $C = (L_1 \vee L_2 \vee \ldots \vee L_k)$ kann auch als Menge von Literalen aufgefasst werden: $C = \{L_1, L_2, \ldots, L_k\}$. Beide Darstellungsformen werden im Folgenden als gleichwertig angesehen.
- Die *leere* Klausel beschreibt eine leere Menge von Literalen und wird durch $\square$ symbolisiert. Sie ist per Definition unerfüllbar.
- Die Vereinigung von zwei Klauseln C_1 und C_2 resultiert in einer Klausel C_3, die alle Literale der beiden Ausgangsklauseln enthält:

  $$C_3 = C_1 \cup C_2 = \{L \,|\, (L \in C_1) \vee (L \in C_2)\}$$

  Literale, die sowohl in C_1 als auch in C_2 auftreten, werden dabei nur einmal in C_3 aufgenommen.
- Die Differenz von zwei Klauseln sei wie folgt definiert:

  $$C_1 - C_2 = \{L \,|\, (L \in C_1) \wedge (L \notin C_2)\}$$

- Ebenso kann eine CNF-Formel $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ mit den Klauseln $C_1, \ldots, C_m$ als Menge aufgefasst werden: $F = \{C_1, C_2, \ldots, C_m\}$.
- In diesem Kontext beschreibt die *leere* Formel eine leere Menge von Klauseln und ist per Definition erfüllbar.
- Die Vereinigung von zwei CNF-Formeln F_1 und F_2 resultiert in einer CNF-Formel F_3, die alle Klauseln der beiden Ursprungsformeln enthält:

  $$F_3 = F_1 \cup F_2 = \{C \,|\, (C \in F_1) \vee (C \in F_2)\}$$

  Klauseln, die sowohl in F_1 als auch in F_2 auftreten, werden nur einmal in F_3 aufgenommen.

Nachfolgend wird die Resolution beziehungsweise die Resolutionsregel definiert, die angibt, unter welchen Voraussetzungen aus zwei gegebenen Klauseln eine dritte Klausel erzeugt werden kann.

Definition 2.9 (Resolution)
Seien zwei Klauseln C_1 und C_2 sowie ein Literal L mit folgender Eigenschaft gegeben: $L \in C_1$ und $\neg L \in C_2$. Dann kann eine Klausel R mit

$$R = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

gebildet werden, die als die Resolvente der Klauseln C_1 und C_2 bezüglich L bezeichnet wird. Als Notation dieses Sachverhalts wird $R = C_1 \otimes_L C_2$ verwendet.

Beispiel 2.4
Seien die beiden Klauseln $C_1 = (x_1 \vee x_2 \vee x_3)$ und $C_2 = (x_4 \vee \neg x_2)$ gegeben. Da $x_2 \in C_1$ und $\neg x_2 \in C_2$ kann gemäß obiger Definition durch Resolution die Resolvente $R = (x_1 \vee x_3 \vee x_4)$ gebildet werden.

Lemma 2.1 (Resolutions-Lemma)
Sei F eine CNF-Formel und R die Resolvente zweier Klauseln C_1 und C_2 aus F. Dann sind F und $F \cup \{R\}$ äquivalent: $F \equiv F \cup \{R\}$.

Beweis: Sei $\mathcal{A}$ eine Belegung, die $F \cup \{R\}$ erfüllt: $\mathcal{A} \models F \cup \{R\}$. Offensichtlich gilt dann auch $\mathcal{A} \models F$. Sei nun umgekehrt angenommen, dass die Belegung $\mathcal{A}$ die Formel F erfüllt ($\mathcal{A} \models F$), wodurch auch alle Klauseln $C_i \in F$ erfüllt sind. Sei weiterhin angenommen, dass die Resolvente R die Form $R = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$ mit $C_1, C_2 \in F$, $L \in C_1$ und $\neg L \in C_2$ hat. Zum Beweis der Äquivalenz kann eine Fallunterscheidung vorgenommen werden, da wegen $\mathcal{A} \models F$ entweder $\mathcal{A} \models L$ oder $\mathcal{A} \models \neg L$ gilt.

- Fall 1: $\mathcal{A} \models L$. Wegen $\mathcal{A} \models C_2$ und $\mathcal{A} \not\models \neg L$ folgt $\mathcal{A} \models (C_2 - \{\neg L\})$. Die Resolvente R ist folglich durch die Belegung $\mathcal{A}$ erfüllt und somit ist auch $F \cup \{R\}$ erfüllt.
- Fall 2: $\mathcal{A} \models \neg L$. Wegen $\mathcal{A} \models C_1$ und $\mathcal{A} \not\models L$ folgt $\mathcal{A} \models (C_1 - \{L\})$. Die Resolvente R ist folglich durch die Belegung $\mathcal{A}$ erfüllt und wiederum gilt: $\mathcal{A} \models F \cup \{R\}$. ∎

Beispiel 2.5
Sei noch einmal die unerfüllbare Formel $G = (x_1 \wedge \neg x_1)$ aus Beispiel 2.2 gegeben, bei der x_1 und $\neg x_1$ jeweils als Unit Clause, bestehend aus genau einem Literal, angesehen werden können: $G = C_1 \wedge C_2$ mit $C_1 = (x_1)$ und $C_2 = (\neg x_1)$. Auch hier kann die Resolution angewendet werden. Als Resolvente entsteht in diesem Fall die leere Klausel, da in den beiden Klauseln C_1 und C_2 außer x_1 und $\neg x_1$ keine weiteren Literale auftreten.

Aus dem Resolutions-Lemma ist klar, dass die Formel G äquivalent zu $G \cup \{R\}$ mit $R = \square$ ist. Beispiel 2.5 liefert somit ein Indiz, dass ein Zusammenhang besteht zwischen der Unerfüllbarkeit einer CNF-Formel und der Herleitbarkeit der leeren Klausel, was im Folgenden formalisiert wird und im *Resolutions-Satz* mündet.

Definition 2.10
Sei F eine CNF-Formel. Dann ist $Res(F)$ definiert als

$$Res(F) \quad = F \cup \{R \,|\, R \textit{ ist Resolvente zweier Klauseln in } F\}.$$

Ferner wird definiert:

$$\begin{aligned} Res^0(F) &= F \\ Res^{t+1}(F) &= Res(Res^t(F)) \textit{ für } t \geq 0 \\ Res^*(F) &= \bigcup_{t \geq 0} Res^t(F) \end{aligned}$$

Satz 2.2 (Resolutions-Satz)
Eine CNF-Formel F ist genau dann unerfüllbar, wenn $\square \in Res^*(F)$.

Beweis: Sei $\square \in Res^*(F)$ angenommen. Es gilt die Korrektheit der Resolution zu zeigen, das heißt, dass F in der Tat unerfüllbar ist. Die leere Klausel kann nur durch Resolution zweier Klauseln der Form $C_1 = (L)$ und $C_2 = (\neg L)$ entstanden sein. Da $\square$ in $Res^*(F)$ enthalten ist, muss ein $t \geq 0$ existieren, für das gilt: $\square, C_1, C_2 \in Res^{t+1}(F)$ und $C_1, C_2 \in Res^t(F)$. Offensichtlich kann keine Belegung existieren, die C_1 und C_2 zeitgleich erfüllt, $Res^t(F)$ ist somit bereits unerfüllbar. Mit Hilfe des Resolutions-Lemmas kann argumentiert werden, dass

$$F \equiv Res^1(F) \equiv Res^2(F) \equiv \ldots \equiv Res^t(F) \equiv Res^{t+1}(F) \equiv \ldots$$

gilt, womit aus der Unerfüllbarkeit von $Res^t(F)$ direkt die Unerfüllbarkeit von F folgt.

Im zweiten Teil des Beweises muss die Vollständigkeit der Resolution nachgewiesen werden. Es ist zu zeigen, dass ausgehend von einer beliebigen unerfüllbaren CNF-Formel F durch die wiederholte Anwendung der Resolutionsregel die leere Klausel hergeleitet werden kann. An dieser Stelle sei lediglich angedeutet, dass der Nachweis durch Induktion über die in F vorkommenden Variablen geführt werden kann. Für Details sei auf [97] verwiesen. ■

Aus dem Resolutions-Satz kann direkt ein Verfahren zum Testen der (Un-)Erfüllbarkeit einer Probleminstanz abgeleitet werden. Einer CNF-Formel F werden durch wiederholte Anwendung der Resolutionsregel solange Resolventen hinzugefügt, bis entweder die leere Klausel gefolgert oder die Resolution nicht mehr angewendet werden kann. Im ersten Fall ist F unerfüllbar und ansonsten erfüllbar.

Ausgehend von der Annahme, dass eine Variable entweder nur als positives Literal, nur als negatives Literal oder gar nicht in einer Klauseln auftritt, ist klar, dass mit n Variablen insgesamt $3^n - 1$ unterschiedliche Klauseln gebildet werden können. Die Terminierung des zuvor skizzierten, naiven Verfahrens ist somit gewährleistet, allerdings stoppt der Algorithmus unter Umständen erst nach der Generierung exponentiell vieler Resolventen. Aufgrund der NP-Vollständigkeit des SAT-Problems ist dies nicht verwunderlich und in der Tat gibt es CNF-Formeln, für die sich dieses *Worst Case*-Verhalten für jeglichen auf der Resolution basierenden Algorithmus nachweisen lässt. Die so genannten *Pigeon Hole* Probleme sind

ein solches Beispiel [83].

Die Resolution kann ebenfalls dazu genutzt werden, eine Variable x_i vollständig aus einer gegebenen CNF-Formel F zu entfernen, das heißt, alle Vorkommen von x_i sowohl als positives als auch als negatives Literal aus F zu löschen. Man spricht hierbei von der *Variablen-Elimination*. Sei x_i die zu eliminierende Variable und gelte $L = x_i$ beziehungsweise $\neg L = \neg x_i$. Dann kann folgende Partitionierung der Klauselmenge von F vorgenommen werden:

- Sei P die Menge aller Klauseln in F, die L enthalten:

$$P = \{C \,|\, (L \in C) \wedge (C \in F)\}$$

- Sei N die Menge aller Klauseln in F, die $\neg L$ enthalten:

$$N = \{C \,|\, (\neg L \in C) \wedge (C \in F)\}$$

- Sei W die Menge aller Klauseln in F, in denen weder L noch $\neg L$ auftritt:

$$W = \{C \,|\, (L \notin C) \wedge (\neg L \notin C) \wedge (C \in F)\}$$

Die gewählte Partitionierung erlaubt die Darstellung $F = P \wedge N \wedge W$ für die gegebene CNF-Formel F. Des Weiteren sei an die Wahl der zu eliminierenden Variablen x_i die Bedingung geknüpft, dass x_i in beiden Polaritäten in F auftritt. Als Menge von Klauseln aufgefasst, sind P und N somit nicht leer. Dann kann auf zwei beliebige Klauseln C_1 und C_2, für die $C_1 \in P$, $C_2 \in N$ gilt, die Resolutionsregel bezüglich x_i angewendet werden. Die Menge der Klauseln, die durch paarweise Resolution, angewendet auf alle derartigen Kombinationen, gebildet werden kann, sei als

$$P \otimes_{x_i} N = \{R \,|\, (R = C_1 \otimes_{x_i} C_2) \wedge (C_1 \in P) \wedge (C_2 \in N)\}$$

definiert. Aufbauend auf dieser Vorarbeit kann der folgende Satz gezeigt werden.

Satz 2.3
Sei F eine CNF-Formel und sei x_i eine Variable, die sowohl als positives Literal der Form $L = x_i$ als auch als negatives Literal der Form $\neg L = \neg x_i$ in F enthalten ist. Weiterhin seien die Mengen P, N und W der zuvor eingeführten Partitionierung der Klauselmenge von F gegeben. Dann sind $F = P \wedge N \wedge W$ und $F' = (P \otimes_{x_i} N) \wedge W$ erfüllbarkeitsäquivalent.

Beweis: Seien P' und N' als die Mengen definiert, die entstehen, wenn aus den Klauseln von P beziehungsweise N alle Vorkommen von $L = x_i$ beziehungsweise $\neg L = \neg x_i$ gelöscht werden:

$$P' = \{C' \mid (C' = C - \{L\}) \wedge (C \in P)\}$$
$$N' = \{C' \mid (C' = C - \{\neg L\}) \wedge (C \in N)\}$$

Zwischen P und P' sowie N und N' gilt folgender Zusammenhang:

$$P = (L \vee P')$$
$$N = (\neg L \vee N')$$

Ausgehend von diesem Sachverhalt kann nun gezeigt werden, dass aus der Unerfüllbarkeit von F auch die Unerfüllbarkeit von F' folgt und umgekehrt.

$$\begin{aligned}
F \text{ ist unerfüllbar} &\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models F \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models P \wedge N \wedge W \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models (L \vee P') \wedge (\neg L \vee N') \wedge W \\
&\Leftrightarrow \forall \mathcal{A}, \mathcal{A}(L) = 0 : \mathcal{A} \not\models (P' \wedge W) \text{ und} \\
&\quad \forall \mathcal{A}, \mathcal{A}(L) = 1 : \mathcal{A} \not\models (N' \wedge W) \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models (P' \wedge W) \vee (N' \wedge W) \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models (P' \vee N') \wedge W \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models [(\textstyle\bigwedge_{C_i \in P'} C_i) \vee (\bigwedge_{C_j \in N'} C_j)] \wedge W \\
&\overset{(*)}{\Leftrightarrow} \forall \mathcal{A} : \mathcal{A} \not\models \left[\bigwedge_{C_1 \in P', C_2 \in N'} (\underbrace{L_1 \vee \ldots \vee L_{k_1}}_{L_1, \ldots, L_{k_1} \in C_1} \vee \underbrace{L'_1 \vee \ldots \vee L'_{k_2}}_{L'_1, \ldots, L'_{k_2} \in C_2}) \right] \wedge W \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models (P \otimes_{x_i} N) \wedge W \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models F' \\
&\Leftrightarrow F' \text{ ist unerfüllbar}
\end{aligned}$$

Der mit $(*)$ gekennzeichnete Schritt stellt in die eine Richtung („$\Rightarrow$“) die Umwandlung der zuvor betrachteten Formel $(P' \vee N') \wedge W$ in eine äquivalente CNF-Darstellung dar. Ein entsprechender Umformungsmechanismus wurde in Abschnitt 2.1 vorgestellt. Für die umgekehrte Richtung („$\Leftarrow$“) müssen lediglich die dort angegebenen Ersetzungsregeln umgedreht werden. ■

Die Aussage des Satzes besteht darin, dass die Frage der Erfüllbarkeit einer CNF-Formel F mit Hilfe einer erfüllbarkeitsäquivalenten Formel F' beantwortet werden kann, bei der im Vergleich zu F eine geeignete Variable x_i vollständig eliminiert wurde. Ist F' unerfüllbar, so gilt dies auch für F, ansonsten sind beide Formeln erfüllbar. Für die beiden nachfolgenden Abschnitte und insbesondere auch Abschnitt 4.2 ist Satz 2.3 von zentraler Bedeutung. Sowohl für den DP- und den DLL-Algorithmus als auch die von heutigen Verfahren oftmals durchgeführte Vorverarbeitung einer Probleminstanz (das so genannte *Preprocessing*)

bildet die Erkenntnis dieses Satzes das theoretische Fundament und gewährleistet die Korrektheit der jeweiligen Vorgehensweise.

Beispiel 2.6
Sei die CNF-Formel F mit

$$F = \underbrace{(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3)}_{P} \wedge \underbrace{(\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2)}_{N} \wedge \underbrace{(x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_2)}_{W}$$

gegeben, aus der in einem ersten Schritt die Variable x_1 eliminiert werden soll. Die entsprechende Einteilung in die Klauselmengen P, N und W ist bereits in der Formel markiert. Die Anwendung der Resolution auf alle Kombinationen von je einer Klausel aus P und N bezüglich x_1 ergibt:

$$P \otimes_{x_1} N = \{(x_2 \vee x_3), (x_2 \vee \neg x_2), (\neg x_3 \vee x_3), (\neg x_3 \vee \neg x_2)\}$$

Bei den beiden mittleren Klauseln handelt es sich um Tautologien, die nicht weiter berücksichtigt werden müssen. Für die resultierende und zu F erfüllbarkeitsäquivalente Formel F' bedeutet dies:

$$F' = (P \otimes_{x_1} N) \wedge W = (x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2) \wedge (x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_2)$$

In einem zweiten Schritt wird die Variablen-Elimination auf x_2 bezüglich F' angewendet. Für P', N' und W' gilt in dieser Situation:

$$\begin{aligned} P' &= \{(x_2 \vee x_3), (\neg x_3 \vee x_2)\} \\ N' &= \{(\neg x_3 \vee \neg x_2), (x_3 \vee \neg x_2)\} \\ W' &= \emptyset \end{aligned}$$

Mit paarweiser Resolution zwischen je einer Klausel aus P' und N' bezüglich x_2 ist die Klauselmenge

$$P' \otimes_{x_2} N' = \{(x_3 \wedge \neg x_3), (x_3), (\neg x_3), (\neg x_3 \wedge x_3)\}$$

herleitbar, die in $F'' = (P' \otimes_{x_2} N') \wedge W' = (x_3) \wedge (\neg x_3)$ mündet. F'' ist offensichtlich unerfüllbar, was damit gemäß Satz 2.3 auch für F' und insbesondere auch für die Ausgangsformel F gilt.

2.4 Davis-Putnam Algorithmus

Bereits im Jahr 1960 wurde von Davis und Putnam ein Verfahren zum Lösen von Erfüllbarkeitsproblemen der Aussagenlogik vorgestellt, das die in Satz 2.3 erzielten Erkenntnisse ausnutzt [28]. Algorithmus 2.1 zeigt den im Allgemeinen rekursiv angegebenen DP-Algorithmus in einer C-ähnlichen Notation, der analog zu Beispiel 2.6 im Wesentlichen auf der wiederholten Anwendung der Variablen-Elimination beruht.

Algorithmus 2.1 Davis-Putnam Algorithmus

```
 1: bool DP(CNF F)
 2: {
 3:   if (F = ∅) { return SATISFIABLE; }                    // Leere Klauselmenge.
 4:   if (□ ∈ F) { return UNSATISFIABLE; }                  // Leere Klausel.

 5:   if (F contains a unit clause (L))                     // Unit Clause Regel.
 6:     {
 7:       // Unit Subsumption.
 8:       F' = F − {C | (L ∈ C) ∧ (C ∈ F) ∧ (C ≠ (L))};

 9:       // Unit Resolution.
10:       P = {(L)};
11:       N = {C | (¬L ∈ C) ∧ (C ∈ F')};
12:       W = F' − P − N;
13:       return DP([P ⊗_L N] ∧ W);
14:     }

15:   if (F contains a pure literal L)                      // Pure Literal Regel.
16:     {
17:       // Delete from F every clause containing L.
18:       F' = F − {C | (L ∈ C) ∧ (C ∈ F)};
19:       return DP(F');
20:     }

21:   L = SelectLiteral(F);                                 // Auswahl eines Literals.
22:   P = {C | (L ∈ C) ∧ (C ∈ F);                           // Variablen-Elimination.
23:   N = {C | (¬L ∈ C) ∧ (C ∈ F)};
24:   W = F − P − N;
25:   return DP([P ⊗_L N] ∧ W);
26: }
```

Die ersten vier if-Anweisungen werden auf jeder *Rekursionsebene* vorrangig behandelt und dienen der Abwicklung von Sonderfällen. Dabei repräsentiert die leere Klauselmenge ein erfüllbares Problem (Zeile 3), während die leere Klausel für ein unerfüllbares Problem steht (Zeile 4). In Zeile 5 wird die Anwendbarkeit der *Unit Clause* Regel geprüft. Eine aus genau einem Literal bestehende Unit Clause der Form (L) kann entweder bereits in der initialen Formel enthalten sein oder dadurch entstehen, dass während der Abarbeitung des DP-Algorithmus aus einer Klausel mit ursprünglich k Literalen $(k-1)$ Literale entfernt wurden. Ist die *Unit Clause* Regel anwendbar, so werden nacheinander *Unit Subsumption* und *Unit Resolution* durchgeführt, bevor rekursiv der DP-Algorithmus mit dem daraus resultierenden Teilproblem als Übergabeparameter aufgerufen wird. Anschaulich ausge-

drückt bewirken *Unit Subsumption* und *Unit Resolution* das Löschen aller Klauseln, die L enthalten, sowie das Entfernen aller Vorkommen von $\neg L$ aus den entsprechenden Klauseln.

Die Definition des Begriffs *Subsumption* ist in Definition 2.11 angegeben. Seien in diesem Zusammenhang C_1 und C_2 zwei Klauseln einer CNF-Formel F, wobei C_2 von C_1 subsumiert wird. Um F zu erfüllen, müssen alle Klauseln erfüllt sein, was insbesondere auch für C_1 gilt. Da alle Literale von C_1 auch in C_2 enthalten sind, erfüllt jede C_1 erfüllende Belegung automatisch auch die Klausel C_2, die folglich nicht mehr gesondert betrachtet werden muss und gelöscht werden kann. Bei der in Zeile 8 von Algorithmus 2.1 gewählten Variante werden nur die Klauseln aus F entfernt, die „echt“ von der Unit Clause (L) subsumiert werden, also diejenigen Klauseln, die aus L und zumindest einem weiteren Literal bestehen. Die Klausel (L) verbleibt noch in der betrachteten Teilformel. Dadurch kann direkt anschließend die *Unit Resolution* durchgeführt werden, die als Spezialfall von Satz 2.3 aufgefasst werden kann, da die dort definierte Klauselmenge P nur aus der Unit Clause (L) besteht. Als Folge dieser Operation wird (L) ebenfalls aus der an die nächste Rekursionsebene weitergeleiteten Teilformel entfernt.

Definition 2.11 (Subsumption)
Seien C_1 und C_2 Klauseln. C_1 subsumiert C_2 genau dann, wenn alle in der Klausel C_1 auftretenden Literale auch in C_2 enthalten sind: $C_1 \subseteq C_2$.

Im nächsten Schritt des DP-Algorithmus wird, wenn möglich, die *Pure Literal* Regel angewendet (Zeile 15). Diese greift immer dann, wenn ein Literal L in der aktuellen Teilformel F entweder nur positiv oder nur negativ auftritt, nicht aber, wenn L und $\neg L$ gemeinsam in F vorkommen. Ist L ein derartiges *Pure Literal*, können alle Klauseln, in denen das Literal L vorkommt, gelöscht werden. Die Korrektheit der *Pure Literal* Regel ergibt sich aus folgender Überlegung: sei L ein *Pure Literal* und C_L die Konjunktion aller Klauseln, in denen L auftritt. F kann folglich in der Form $F = C_L \wedge F'$ dargestellt werden, wobei F' die Menge der restlichen, nicht in C_L enthaltenen Klauseln von F repräsentiert. Nun gilt:

$$
\begin{aligned}
F \text{ ist unerfüllbar} &\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models F \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models (C_L \wedge F') \\
&\Leftrightarrow \forall \mathcal{A} : \mathcal{A} \not\models F' \\
&\Leftrightarrow F' \text{ ist unerfüllbar}
\end{aligned}
$$

Die Folgerung $\forall \mathcal{A} \not\models (C_L \wedge F') \Rightarrow \forall \mathcal{A} \not\models F'$ beruht auf der Tatsache, dass $\forall \mathcal{A} \not\models (C_L \wedge F')$ insbesondere auch für alle Belegungen $\mathcal{A}$ mit $\mathcal{A}(L) = 1$ gilt. In diesem Fall sind alle Klauseln in C_L erfüllt, woraus folgt, dass die Unerfüllbarkeit von F von der Unerfüllbarkeit von F' abhängt. Als Konsequenz genügt es, im weiteren Verlauf des Suchprozesses F' zu betrachten (Zeile 19).

Erst wenn keiner der Sonderfälle greift, wird in Zeile 21 durch SelectLiteral ein Literal L bestimmt und diesbezüglich die im Abschnitt zuvor vorgestellte Variablen-Elimination

durchgeführt. Aufgrund der vorgeschalteten Sonderfall-Behandlung kann dabei jedes noch in F enthaltene Literal gewählt werden. Die für die Anwendung von Satz 2.3 notwendige Voraussetzung, dass die Klauselmengen P und N nicht leer sind, ist in jedem Fall erfüllt. In [28] wird vorgeschlagen, stets ein Literal der aktuell kürzesten Klausel zu wählen.

Die Korrektheit und Terminierung des DP-Algorithmus ergibt sich direkt aus Satz 2.3. Spätestens wenn für alle in der Originalformel F enthaltenen Variablen die Variablen-Elimination durchgeführt wurde, stoppt das Verfahren. Wie bei dem direkt im Anschluss an den Resolutions-Satz skizzierten naiven Verfahren, ist die im schlechtesten Fall exponentielle Zunahme an Klauseln, bedingt durch die in Zeile 25 wiederholt durchgeführte Berechnung von $(P \otimes_L N) \wedge W$, auch beim DP-Algorithmus ein Problem.

Andererseits kann unter Umständen auch ein weitaus günstigerer Fall eintreten. Man betrachte hierzu erneut Beispiel 2.6. Die dortige Ausgangsformel F besteht aus 3 Variablen, 12 Literalen sowie 6 Klauseln. Gemäß Definition 2.6 gilt für die Größe: $|F| = 17$. Die zu F erfüllbarkeitsäquivalente Formel F', die sich als Folge der Elimination von x_1 ergibt, besteht dagegen aus 2 Variablen, 8 Literalen und nur 4 Klauseln bei einer Größe von $|F'| = 11$. Analog ist die Situation beim Übergang von F' zu F''.

Einige der Preprocessing-Routinen moderner SAT-Algorithmen setzen an diesem Punkt an und versuchen, im Vorfeld des eigentlichen Suchprozesses exakt diejenigen Variablen zu identifizieren und zu eliminieren, die genau diesen die Klauselmenge reduzierenden Effekt hervorrufen. Motiviert wird das Vorgehen durch die Beobachtung, dass eine kleinere CNF-Formel in der Regel auch schneller von einem SAT-Algorithmus gelöst werden kann. Abschnitt 4.2 greift diese Thematik erneut auf.

2.5 Davis-Logemann-Loveland Algorithmus

Mit dem Ziel, den gegebenenfalls exponentiellen Speicherbedarf des DP-Algorithmus zu vermeiden, wurde 1962 von Davis, Logemann und Loveland eine Weiterentwicklung vorgestellt: der Davis-Logemann-Loveland Algorithmus [27].

Wie Algorithmus 2.2 zeigt, betreffen die gegenüber dem DP-Verfahren vorgenommenen Änderungen einzig die Zeilen 22 bis 25. Anstelle der Variablen-Elimination wird vom DLL-Algorithmus eine Fallunterscheidung bezüglich des ausgewählten Literals L vorgenommen. Begründet werden kann dies dadurch, dass für jede Belegung $\mathcal{A}$, welche die auf der aktuellen Rekursionsebene betrachtete Teilformel F erfüllt, entweder $\mathcal{A}(L) = 1$ oder $\mathcal{A}(L) = 0$ gelten muss. Beide Fälle werden der Reihe nach überprüft. Im ersten Fall geschieht dies durch die Hinzunahme der Unit Clause (L) zu F und einem rekursiven Aufruf des DLL-Algorithmus, womit geprüft wird, ob eine erfüllende Belegung $\mathcal{A}$ mit $\mathcal{A}(L) = 1$ existiert. Durch die eingefügte Unit Clause wird auf der nächsten Rekursionsebene automatisch die *Unit Clause*

Algorithmus 2.2 Davis-Logemann-Loveland Algorithmus

```
 1: bool DLL(CNF F)
 2: {
 3:   if (F = ∅) { return SATISFIABLE; }                    // Leere Klauselmenge.
 4:   if (□ ∈ F) { return UNSATISFIABLE; }                  // Leere Klausel.

 5:   if (F contains a unit clause (L))                     // Unit Clause Regel.
 6:     {
 7:       // Unit Subsumption.
 8:       F' = F − {C | (L ∈ C) ∧ (C ∈ F) ∧ (C ≠ (L))};

 9:       // Unit Resolution.
10:       P = {(L)};
11:       N = {C | (¬L ∈ C) ∧ (C ∈ F')};
12:       W = F' − P − N;
13:       return DLL([P ⊗_L N] ∧ W);
14:     }

15:   if (F contains a pure literal L)                      // Pure Literal Regel.
16:     {
17:       // Delete from F every clause containing L.
18:       F' = F − {C | (L ∈ C) ∧ (C ∈ F)};
19:       return DLL(F');
20:     }

21:   L = SelectLiteral(F);                                 // Auswahl eines Literals.
22:   if (DLL(F ∪ {(L)}) == SATISFIABLE)                    // Fallunterscheidung.
23:     { return SATISFIABLE; }
24:   else
25:     { return DLL(F ∪ {(¬L)}); }
26: }
```

Regel aktiv und die Formel entsprechend vereinfacht. Falls $(F \cup \{(L)\})$ erfüllbar ist, so ist auch F erfüllbar und die Abarbeitung kann mit dem Rückgabewert *SATISFIABLE* gestoppt werden (Zeilen 22 und 23). Andernfalls wird für L der komplementäre Wahrheitswert angenommen, wiederum die entsprechende Unit Clause zu F hinzugefügt und rekursiv das Teilproblem $(F \cup \{(\neg L)\})$ untersucht. Liefert auf den nachfolgenden Rekursionsebenen auch diese Analyse das Ergebnis *UNSATISFIABLE*, so ist gezeigt, dass die betrachtete Teilformel F unerfüllbar ist, denn das Literal L muss einen der beiden Wahrheitswerte annehmen.

Man beachte in diesem Zusammenhang, dass sich der Rückgabewert *UNSATISFIABLE*

immer nur auf das auf der jeweiligen Rekursionsebene betrachtete Teilproblem bezieht und nicht zwangsläufig auch für die Ausgangsformel gelten muss. Eine unerfüllbare Probleminstanz liegt erst dann vor, wenn auch für die Rekursionsebene, auf der die erste Fallunterscheidung getätigt wurde, gilt, dass für das auf dieser Ebene ausgewählte Literal L weder die Annahme $\mathcal{A}(L) = 1$ noch die Annahme $\mathcal{A}(L) = 0$ zu einer erfüllenden Belegung $\mathcal{A}$ führt. In dieser Situation wurde dann der gesamte durch die CNF-Formel aufgespannte Suchraum durchsucht, ohne dass eine erfüllende Belegung bestimmt werden konnte.

Die Bedeutung des DLL-Algorithmus für die heutige Forschung im Bereich der SAT-Algorithmen ergibt sich aus der Tatsache, dass das Grundgerüst nahezu aller modernen und vollständigen Verfahren weiterhin auf den vor über 45 Jahren entwickelten Konzepten beruht. In Kapitel 4 werden die auf diesem Basis-Algorithmus aufbauenden, leistungssteigernden Techniken beschrieben. Ein hier gänzlich ausgesparter Aspekt ist in geeigneten Datenstrukturen und Funktionen zu sehen, um beispielsweise eine effiziente Durchführung der *Unit Clause* Regel zu gewährleisten.

Kapitel 3

Anwendungsgebiete von SAT-Algorithmen

Dank des enormen Leistungssprungs moderner SAT-Algorithmen hat das Erfüllbarkeitsproblem der Aussagenlogik in den letzten Jahren vermehrt an praktischer Relevanz gewonnen. Viele anwendungsbezogene Fragestellungen lassen sich als SAT-Problem formulieren und können von heutigen Verfahren mit vertretbarem Zeitaufwand gelöst werden. Beispielhaft werden in diesem Kapitel mit *Automatic Test Pattern Generation* und *Combinational Equivalence Checking* zwei Teildisziplinen aus dem Bereich des rechnergestützten Schaltkreisentwurfs aufgegriffen und mögliche Kodierungen als Erfüllbarkeitsproblem der Aussagenlogik aufgezeigt. Darüber hinaus existiert eine Reihe weiterer Anwendungsgebiete, in denen SAT-Algorithmen eingesetzt werden. Stellvertretend seien an dieser Stelle *Bounded Model Checking* [1, 13, 23, 103], ebenfalls ein Teilgebiet des Schaltkreisentwurfs, sowie Planungsprobleme der *Künstlichen Intelligenz* genannt [62, 63].

3.1 Tseitin-Transformation

Eine im weiteren Verlauf dieses Kapitels immer wiederkehrende Aufgabe besteht darin, die durch einen Schaltkreis repräsentierte Funktion als CNF-Formel darzustellen, wobei für die folgenden Betrachtungen vorausgesetzt wird, dass ein kombinatorischer Schaltkreis vorliegt. Die Handhabung sequentieller Schaltkreise mit speichernden Elementen fällt in den Bereich *Bounded Model Checking*, der in dieser Übersicht ausgespart wird, hier sei auf die zuvor angegebenen Literaturstellen verwiesen. Ferner sei angenommen, dass die betrachteten Schaltkreise nicht in hierarchischer Form vorliegen und nur aus so genannten *Grundgattern* zusammengesetzt sind. Bezüglich der Menge der Grundgatter genügt es, sich auf die Gatter zu beschränken, die für die Berechnung der UND- und ODER-Verknüpfung sowie der Negation benötigt werden, da sich mit diesen drei Operatoren alle aussagenlogischen Formeln beschreiben lassen. Zusätzlich zu diesen drei Typen von Gattern wird in diesem Kapitel allerdings davon ausgegangen, dass das EXOR-Gatter zur Berechnung der EXOR-Verknüpfung ebenfalls der Menge der Grundgatter angehört, was dazu dient, die nachfolgend beschriebenen Sachverhalte übersichtlicher darstellen zu können.

Die natürliche Vorgehensweise bei der Umwandlung der durch einen Schaltkreis repräsentierten Funktion in eine CNF-Formel besteht aus zwei Schritten. Im ersten Schritt werden die Grundgatter des Schaltkreises separat in entsprechende Teilformeln in konjunktiver

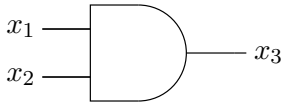
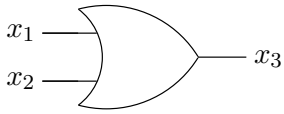
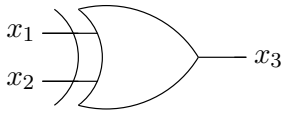
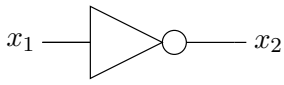
Gatter	Funktion	CNF-Formel
	$x_3 \equiv x_1 \wedge x_2$	$(\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_2) \wedge (x_3 \vee \neg x_1 \vee \neg x_2)$
	$x_3 \equiv x_1 \vee x_2$	$(x_3 \vee \neg x_1) \wedge (x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_1 \vee x_2)$
	$x_3 \equiv x_1 \oplus x_2$	$(\neg x_3 \vee x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee \neg x_2) \wedge$ $(x_3 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_1 \vee \neg x_2)$
	$x_2 \equiv \neg x_1$	$(x_2 \vee x_1) \wedge (\neg x_2 \vee \neg x_1)$

Tabelle 3.1: Die vier Grundgatter und ihre CNF-Darstellung

Normalform übersetzt. Tabelle 3.1 zeigt die Kodierung der vier Grundgatter. Die aufgelisteten CNF-Formeln charakterisieren das Verhalten der Gatter, so dass eine bezüglich der Gatter-Funktionalität „gültige“ Wertekombination der Ein- und Ausgänge wie etwa $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ für ein ODER-Gatter bewirkt, dass die entsprechende CNF-Formel erfüllt ist. Eine im Gegensatz dazu ungültige Kombination wie beispielsweise $x_1 = x_2 = 1$ für einen Inverter erfüllt die entsprechende CNF-Formel $(x_2 \vee x_1) \wedge (\neg x_2 \vee \neg x_1)$ nicht, da diese Belegung nicht der intendierten Funktionalität eines Inverters entspricht.

Im zweiten Schritt werden die so für alle Grundgatter eines gegebenen Schaltkreises gewonnen Teilformeln per UND-Verknüpfung zur endgültigen CNF-Darstellung der durch den Schaltkreis repräsentierten Funktion zusammengefasst. Das folgende Beispiel verdeutlicht das Vorgehen.

Beispiel 3.1
Sei der in Abbildung 3.1 dargestellte Schaltkreis SK gegeben, der die Funktion

$$F_{SK} = (x_1 \wedge x_2) \vee \neg x_3$$

berechnet. Die Funktion F_{SK} soll in eine CNF-Darstellung überführt werden. Als erstes werden unter Zuhilfenahme der beiden zusätzlichen Variablen x_5 und x_6, die zu den internen Signalen des Schaltkreises korrespondieren, die drei Gatter von SK jeweils separat in die entsprechende CNF-Formel überführt:

- $F_\wedge = (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2)$ *(UND-Gatter)*

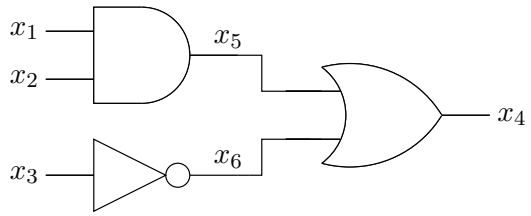


Abbildung 3.1: Schaltkreis SK zur Berechnung der Funktion $F_{SK} = (x_1 \wedge x_2) \vee \neg x_3$

- $F_\neg = (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3)$ *(Inverter)*
- $F_\vee = (x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_5 \vee x_6)$ *(ODER-Gatter)*

Die endgültige Umwandlung von F_{SK} in eine Formel in konjunktiver Normalform wird dann im zweiten Schritt durch die UND-Verknüpfung der drei zuvor gebildeten Teilformeln $F_\wedge$, $F_\neg$ und $F_\vee$ erreicht:

$$\begin{aligned} F_{SK}^{CNF} = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge \\ & (x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_5 \vee x_6) \end{aligned}$$

Diese Art der Überführung einer aussagenlogischen Formel in eine CNF-Darstellung geht zurück auf Tseitin [114] und ist daher auch unter dem Namen *Tseitin-Transformation* bekannt. Beschränkt man sich in diesem Zusammenhang (wie in Tabelle 3.1 geschehen) auf Grundgatter, für welche die CNF-Darstellung der durch das jeweilige Gatter repräsentierten Funktion nur konstant viele Klauseln erfordert, so ist die Größe einer per Tseitin-Transformation erzeugten Formel linear in der Größe der Ausgangsformel. Dies ist ein klarer Vorteil gegenüber den in Abschnitt 2.1 angegebenen Umformungsregeln, bei denen die resultierende CNF-Formel unter Umständen exponentiell größer ist als die Ausgangsformel. Weiterhin kann gezeigt werden, dass bei der Tseitin-Transformation durch die zusätzlich eingeführten Variablen zwar nicht die (logische) Äquivalenz gegeben ist, aber die Erfüllbarkeitsäquivalenz gemäß Definition 2.7 gilt. Bezogen auf Beispiel 3.1 bedeutet dies, dass F_{SK} genau dann erfüllbar ist, wenn F_{SK}^{CNF} erfüllbar ist. Die Frage der Erfüllbarkeit von F_{SK} kann folglich mit Hilfe von F_{SK}^{CNF} beantwortet werden und umgekehrt.

3.2 Miter-Schaltkreis

Ebenso wie die Tseitin-Transformation ist auch das Konzept des *Miters* [18] für die weiteren Abschnitte dieses Kapitels von Bedeutung. Abbildung 3.2 gibt ein Beispiel wie der Miter-Schaltkreis im Bereich *Combinational Equivalence Checking* Verwendung findet.

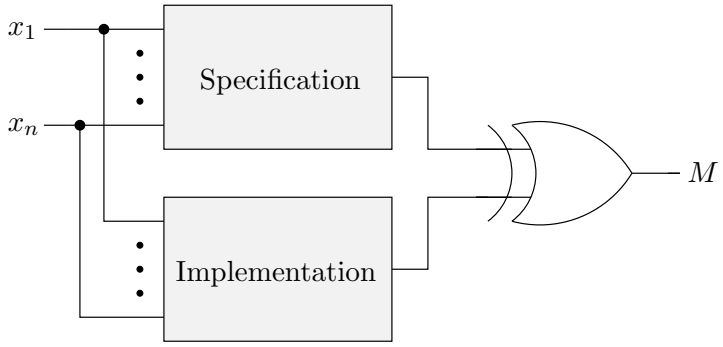


Abbildung 3.2: Miter-Schaltkreis

Das Ziel dieses Anwendungsbereichs, der im Folgenden genauer behandelt wird, besteht darin, zu verifizieren, ob ein anhand einer vorgegebenen Spezifikation entwickelter Schaltkreis (in der Abbildung als *Implementation* bezeichnet) auch in der Tat den geforderten Vorgaben entspricht. Dazu werden sowohl Spezifikation als auch Implementierung in einen so genannten *Miter-Schaltkreis* eingebettet, wobei jeweils zueinander korrespondierende Eingänge miteinander verbunden sind. Die zueinander korrespondierenden Ausgänge von Spezifikation und Implementierung werden jeweils mit den Eingängen eines EXOR-Gatters verbunden, während die Ausgänge aller EXOR-Gatter so zusammengeführt werden, dass der Ausgang M des Miters der ODER-Verknüpfung aller EXOR-Gatter entspricht. Abbildung 3.2 zeigt ein dahingehend vereinfachtes Szenario, bei dem Spezifikation und Implementierung nur über je einen Ausgang verfügen.

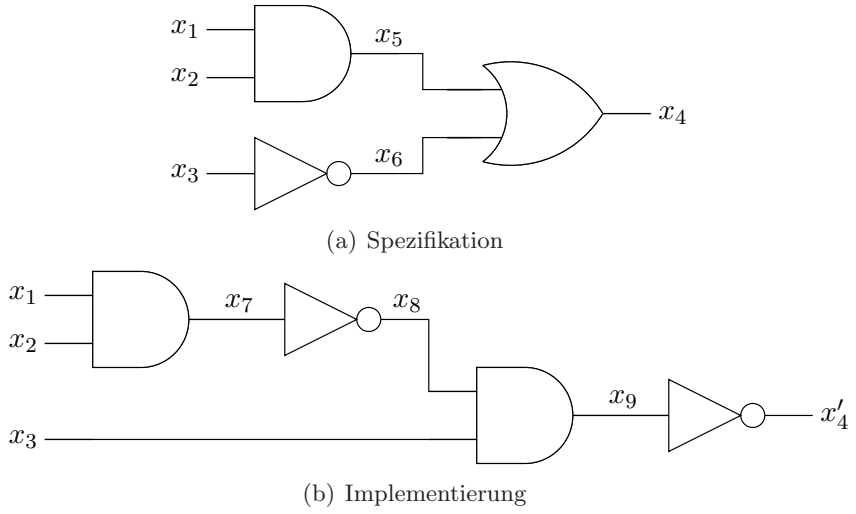
Mittels Tseitin-Transformation wird der Miter-Schaltkreis in eine CNF-Formel übersetzt. Gelingt es einem SAT-Algorithmus nun, eine erfüllende Belegung zu bestimmen, bei dem für den Ausgang des Miters $M = 1$ gilt, so ist gezeigt, dass Spezifikation und Implementierung für zumindest eine Belegung der Eingangs-Pins, die identisch auf beide Teilschaltungen geführt sind, ein unterschiedliches Ausgabeverhalten zeigen. Die entwickelte Schaltung entspricht somit (noch) nicht den Vorgaben. Zudem gibt das gefundene Modell gegebenenfalls auch einen Hinweis darauf, in welchem Bereich der Schaltung der Fehler zu suchen ist. Durch das Hinzufügen der Unit Clause (M) zur CNF-Darstellung der durch den Miter-Schaltkreis repräsentierten Funktion kann die Problemstellung so modifiziert werden, dass bei allen erfüllenden Belegungen $M = 1$ gilt. Ohne diese Unit Clause repräsentiert eine erfüllende Belegung zwar eine gültige Wertekombination der Ein- und Ausgänge des Miters sowie aller internen Signale, es muss aber nicht zwangsläufig $M = 1$ gelten.

Auch im Bereich *Automatic Test Pattern Generation* (siehe Abschnitt 3.4) ist der Miter-Schaltkreis gängige Praxis, anstelle Spezifikation und Implementierung spricht man in diesem Zusammenhang von fehlerfreiem und fehlerbehaftetem Schaltkreis. Das Ziel ist es, ein Testmuster (eine Belegung der Eingangs-Pins) zu bestimmen, bei dem sich das Aus-

gabeverhalten von fehlerfreiem und fehlerbehaftetem Schaltkreis unterscheidet und somit erneut $M = 1$ für den Miter-Ausgang gilt. Gelingt dies, das heißt, ist der SAT-Algorithmus in der Lage, eine entsprechende Belegung mit $M = 1$ zu bestimmen, so ist ein Testmuster gefunden, mit dem sich dieser spezielle Fehler testen lässt.

3.3 Combinational Equivalence Checking

Im Bereich *Combinational Equivalence Checking* (CEC) soll verifiziert werden, ob eine kombinatorische Schaltung mit einer geforderten Spezifikation übereinstimmt. In der Praxis wird dieser Abgleich mehrfach und zu unterschiedlichen Zeitpunkten während des gesamten Entwurfsprozesses durchgeführt. Im Folgenden wird der Einsatz von SAT-Algorithmen in dieser Anwendungsdomäne anhand eines Beispiels näher erläutert. Seien dazu die beiden in Abbildung 3.3 dargestellten Schaltkreise gegeben. Beim oberen Schaltkreis handelt es sich um das Beispiel aus Abbildung 3.1, das an dieser Stelle als Spezifikation eines zu implementierenden Schaltkreises dient, während angenommen wird, dass es sich beim unteren Schaltkreis um eine Implementierung handelt, von der geprüft werden soll, ob sie der Spezifikation genügt.



(a) Spezifikation

(b) Implementierung

Abbildung 3.3: Spezifikation und Implementierung eines Schaltkreises

Wie im vorherigen Abschnitt erläutert, werden dazu zunächst Spezifikation und Implementierung in Form eines Miter-Schaltkreises zusammengeführt, der für das hier betrachtete Beispiel in Abbildung 3.4 dargestellt ist. Der Äquivalenznachweis kann derart erfolgen, dass die durch den Miter-Schaltkreis repräsentierte Funktion in eine CNF-Darstellung überführt und anschließend mit einem SAT-Algorithmus gelöst wird. Gelingt es dem SAT-Algorithmus, eine erfüllende Belegung mit $M = 1$ zu bestimmen, so ist gezeigt, dass

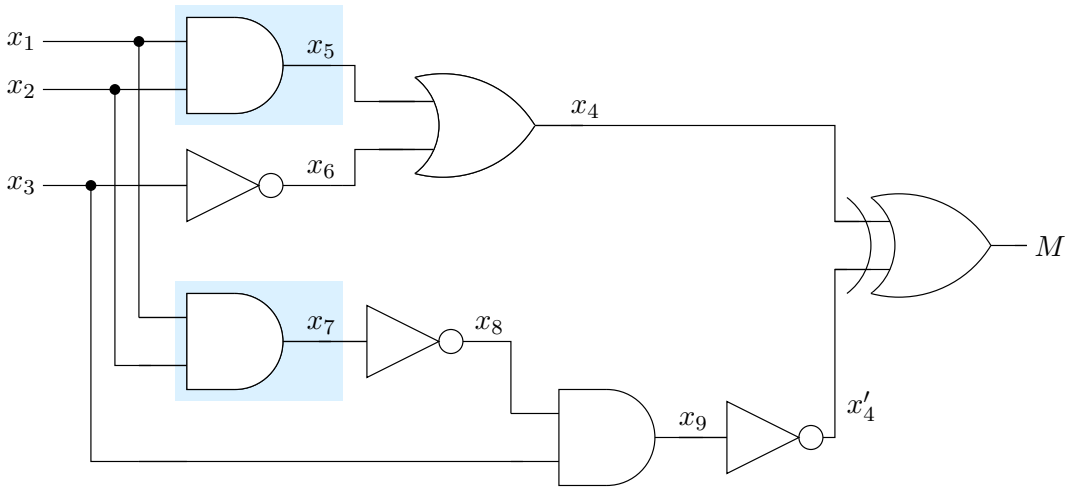


Abbildung 3.4: Miter-Schaltkreis zur Überprüfung der Äquivalenz

zumindest eine Belegung der Eingangs-Pins existiert, bei der die beiden Schaltkreise ein unterschiedliches Ausgabeverhalten zeigen. Ist die CNF-Formel hingegen unerfüllbar, so sind beide Schaltkreise äquivalent und die Implementierung erfüllt die vorgegebene Spezifikation.

In der Regel ist die bei diesem Vorgehen zu lösende CNF-Formel allerdings so groß, dass sie von einem SAT-Algorithmus entweder gar nicht oder nur mit einem erheblichen Zeitaufwand erfolgreich bearbeitet werden kann. Typischerweise wird daher versucht, innerhalb des Miters äquivalente Signale zu bestimmen, anhand derer der Miter-Schaltkreis vereinfacht werden kann, was schlussendlich zu einer (gegenüber der ursprünglichen Variante) kleineren CNF-Formel führt.

Die Identifikation äquivalenter Signale, welche die gleiche (Teil-)Funktion repräsentieren, kann wie folgt durchgeführt werden: für einige zufällig oder heuristisch bestimmte Belegungen der Eingangs-Pins des Miters wird analysiert, welchen logischen Wert die beiden potenziell zueinander äquivalenten Signale jeweils annehmen [11]. Weisen die getesteten Signale bei mindestens einer Belegung der Eingangs-Pins einen Unterschied auf, so ist klar, dass die beiden Leitungen nicht die gleiche Funktion repräsentieren. Beispielsweise sind die Variablen x_4 und x_8 aus Abbildung 3.4 nicht äquivalent, da sie für die Belegung $x_1 = 0$, $x_2 = 0$ und $x_3 = 1$ der Eingangs-Pins des Miters unterschiedliche logische Werte annehmen: für x_4 gilt $x_4 = 0$, während x_8 den logischen Wert $x_8 = 1$ annimmt.

Bestehen die potenziell äquivalenten Signale diesen ersten Test, ist prinzipiell die Chance gegeben, dass beide Signale in der Tat die gleiche (Teil-)Funktion repräsentieren. Sei zur Verdeutlichung angenommen, dass die in Abbildung 3.4 blau unterlegten Signale x_5 und

x_7 bei allen getesteten Belegungen der Eingangs-Pins stets den gleichen Wahrheitswert aufwiesen. Um in dieser Situation die Äquivalenz der beiden Signale formal nachzuweisen, werden die Teilschaltkreise, die den an x_5 und x_7 jeweils anliegenden logischen Wert beeinflussen, in einen separaten Miter-Schaltkreis eingebettet. Abbildung 3.5 verdeutlicht das Vorgehen für das hier gewählte Beispiel.

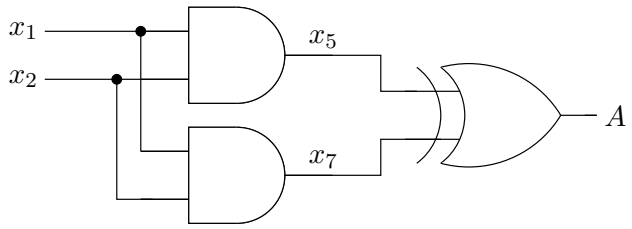


Abbildung 3.5: Miter-Schaltkreis der Signale x_5 und x_7

Der so entstandene Miter beziehungsweise die dadurch berechnete Funktion kann dann per Tseitin-Transformation in eine CNF-Darstellung überführt werden:

$$\begin{aligned} F_A = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (\neg x_7 \vee x_1) \wedge (\neg x_7 \vee x_2) \wedge (x_7 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (\neg A \vee x_5 \vee x_7) \wedge (\neg A \vee \neg x_5 \vee \neg x_7) \wedge \\ & (A \vee \neg x_5 \vee x_7) \wedge (A \vee x_5 \vee \neg x_7) \wedge (A) \end{aligned}$$

Die abschließende Unit Clause (A) wurde eingefügt, um zu gewährleisten, dass bei jeder die Formel F_A erfüllenden Belegung $A = 1$ gilt. Analog zur *Unit Clause* Regel des DLL-Algorithmus kann F_A bereits vor der Übergabe an einen SAT-Algorithmus dahingehend vereinfacht werden, dass alle durch die Unit Clause (A) subsumierten Klauseln sowie alle Vorkommen des Literals $\neg A$ gelöscht werden. F_A vereinfacht sich somit wie folgt:

$$\begin{aligned} F'_A = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (\neg x_7 \vee x_1) \wedge (\neg x_7 \vee x_2) \wedge (x_7 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (x_5 \vee x_7) \wedge (\neg x_5 \vee \neg x_7) \wedge (A) \end{aligned}$$

Da sowohl x_5 als auch x_7 die UND-Verknüpfung von x_1 und x_2 berechnen, ist F'_A offensichtlich unerfüllbar. Daraus kann gefolgert werden, dass es sich bei x_5 und x_7 um äquivalente Signale handelt, die nicht nur bei den (wenigen) getesteten Belegungen der Eingangs-Pins stets den gleichen logischen Wert annehmen, sondern bei allen möglichen Belegungen. Aufbauend auf diesem Resultat kann in dem in Abbildung 3.4 dargestellten Miter einer der beiden Teilschaltkreise zur Berechnung der durch x_5 beziehungsweise x_7 repräsentierten Funktion entfernt und mittels einer neu eingeführten Signalleitung durch den verbliebenen Teilschaltkreis ersetzt werden. Abbildung 3.6 zeigt diese Optimierung des Miter-Schaltkreises, bei der x_7 durch das dazu äquivalente Signal x_5 ersetzt wurde (die neu

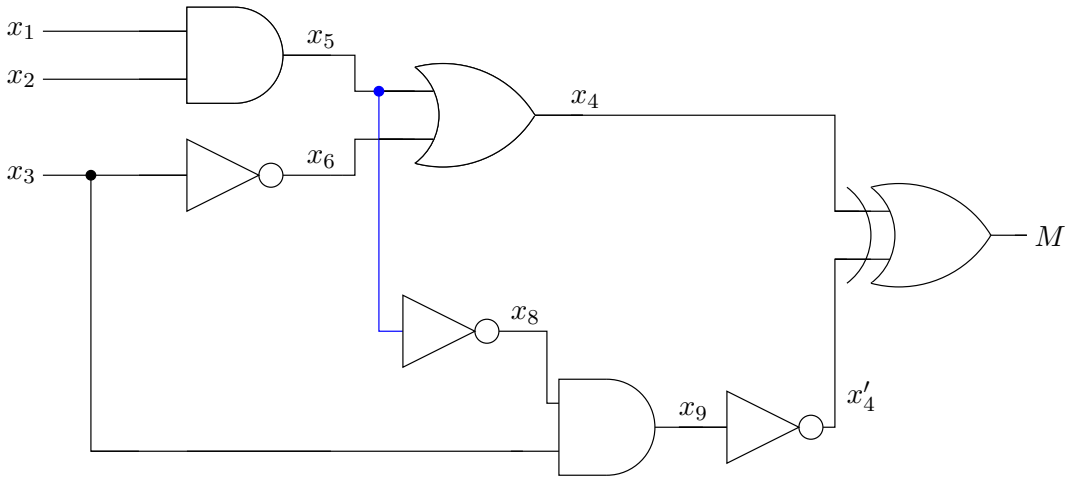


Abbildung 3.6: Optimierter Miter-Schaltkreis zur Überprüfung der Äquivalenz

eingeführte Signalleitung ist blau hervorgehoben).

Ausgehend von denjenigen Signalen, die sich direkt an die Eingangs-Pins anschließen, wird dieser Prozess solange fortgeführt, bis schlussendlich überprüft werden muss, ob die beiden zum Ausgang der Spezifikation beziehungsweise der Implementierung korrespondierenden Signale äquivalent sind. In dem hier gewählten Beispiel wäre dies der Test, ob die beiden Signale x_4 (Spezifikation) und x'_4 (Implementierung) äquivalent sind, was, eingebettet in einen entsprechenden Miter, dem in Abbildung 3.6 gezeigten Schaltkreis entspricht. Die durch den Miter berechnete Funktion lässt sich per Tseitin-Transformation in folgende CNF-Darstellung überführen:

$$\begin{aligned} F_M = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge \\ & (x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_5 \vee x_6) \wedge (x_8 \vee x_5) \wedge (\neg x_8 \vee \neg x_5) \wedge \\ & (\neg x_9 \vee x_8) \wedge (\neg x_9 \vee x_3) \wedge (x_9 \vee \neg x_8 \vee \neg x_3) \wedge (x'_4 \vee x_9) \wedge (\neg x'_4 \vee \neg x_9) \wedge \\ & (\neg M \vee x_4 \vee x'_4) \wedge (\neg M \vee \neg x_4 \vee \neg x'_4) \wedge (M \vee \neg x_4 \vee x'_4) \wedge (M \vee x_4 \vee \neg x'_4) \wedge \\ & (M) \end{aligned}$$

Leicht nachvollziehbar ist, dass die in Abbildung 3.3(b) gegebene Implementierung die geforderte Spezifikation aus Abbildung 3.3(a) erfüllt, somit verwundert es nicht, dass die CNF-Formel F_M von einem SAT-Algorithmus als unerfüllbar bewertet wird.

Insgesamt kann das Vorgehen beim *Combinational Equivalence Checking* dahingehend charakterisiert werden, dass die Äquivalenz von Implementierung und geforderter Spezifikation nicht auf ein einziges, unter Umständen sehr großes SAT-Problem, sondern auf eine Reihe kleinerer Problemstellungen zurückgeführt wird, die nacheinander bearbeitet werden. Können dabei äquivalente Signale bestimmt werden, besteht die Chance, dass das „finale“

SAT-Problem, mit dem die eigentliche Übereinstimmung von Spezifikation und Implementierung gezeigt oder widerlegt wird, substanziell kleiner ist als die ursprüngliche Variante, bei der keinerlei äquivalente Signale ausgenutzt wurden.

Neben SAT-Algorithmen gehören Ansätze wie [65, 82], die auf der Datenstruktur der *Binary Decision Diagrams* (BDDs) [17, 19, 106] aufbauen, zu den klassischen Werkzeugen im Bereich CEC. Speziell so genannte ROBDDs (*Reduced Ordered Binary Decision Diagrams*) bieten den Vorteil einer kanonischen und eindeutigen Repräsentation. Sind Implementierung und Spezifikation bezüglich des Ein- und Ausgabeverhaltens identisch, weisen sie (abgesehen von Isomorphie) auch eine identische Darstellung als ROBDD auf, was einen effizienten Äquivalenztest ermöglicht. Allerdings haben Binary Decision Diagrams im Allgemeinen den Nachteil eines sehr hohen Speicherbedarfs, so dass es je nach verfügbarem Hauptspeicher nicht möglich ist, die entsprechende ROBDD-Darstellung von Spezifikation und Implementierung aufzubauen. Hier bieten SAT-Algorithmen eine interessante Alternative, sind sie doch wesentlich weniger speicherintensiv, allerdings muss in der Regel ein höherer Zeitaufwand beim Nachweis der Äquivalenz in Kauf genommen werden.

Es konnte gezeigt werden, dass Problemstellungen existieren, die sich für den Einsatz von BDD-Ansätzen eignen, während SAT-Algorithmen diese nicht effizient bearbeiten können und umgekehrt [46]. Daher liegt es auf der Hand, die Vorteile von BDD-basierten und SAT-basierten Verfahren zu koppeln [20, 87]. Beispielsweise ist es denkbar, mit einem BDD-Ansatz zu beginnen und immer dann zu einem SAT-Ansatz zu wechseln, wenn Spezifikation und Implementierung mangels verfügbarem Hauptspeicher nicht als ROBDD dargestellt werden können.

Abschließend sei angemerkt, dass der in diesem Abschnitt skizzierte Ansatz zum Lösen von Fragestellungen aus dem Bereich *Combinational Equivalence Checking* auch angewendet werden kann, wenn (im Gegensatz zu dem hier gewählten Beispiel) sowohl Spezifikation als auch Implementierung über mehr als einen Ausgang verfügen. Arbeiten in diese Richtung finden sich in [29, 49].

3.4 Automatic Test Pattern Generation

In diesem Abschnitt wird mit *Automatic Test Pattern Generation* (ATPG), auch als Testmustergenerierung bezeichnet, ein zweites Teilgebiet des rechnergestützten Schaltkreisentwurfs aufgegriffen. Es wird erläutert, in welcher Form SAT-Algorithmen in diesem Bereich eingesetzt werden können. Die Aufgabe besteht darin, für einen in einem Schaltkreis möglicherweise vorhandenen Fehler eine Belegung der Eingangs-Pins (ein Testmuster) zu bestimmen, mit dem sich der Fehler, sofern dieser in der Tat auftritt, durch ein Fehlverhalten an den Ausgängen bemerkbar macht. Existiert ein entsprechendes Testmuster, so kann für die gefertigte Schaltung festgestellt werden, ob sie diesen speziellen Fehler aufweist

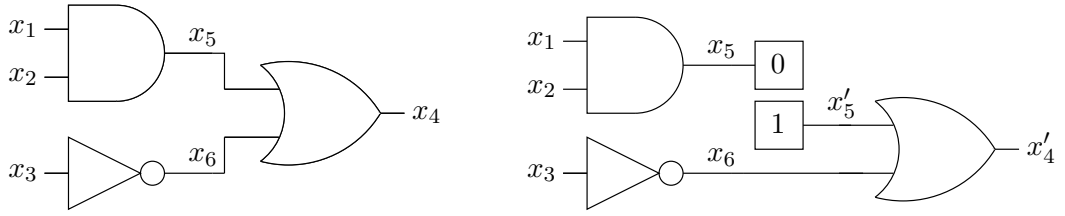
oder nicht. Ein solcher Fehler kann beispielsweise durch einen Fabrikationsfehler entstanden sein. Demgegenüber werden Fehler, für die kein Testmuster bestimmt werden kann, als *redundant* bezeichnet. Unabhängig von der Belegung der Eingangs-Pins eines Schaltkreises macht sich ein redundanter Fehler nicht durch ein vom fehlerfreien Schaltkreis abweichendes Ausgabeverhalten bemerkbar und kann daher nicht detektiert werden.

Die Bestimmung von Testmustern für unter Umständen in einer Schaltung enthaltene Fehler wird maßgeblich durch das betrachtete *Fehlermodell* beeinflusst. Im Folgenden liegt der Fokus auf dem so genannten *Single Stuck-At* Fehlermodell, einem der gängigsten Modelle im Bereich ATPG [2]. Wie der Name andeutet, basiert dieses Fehlermodell auf der Idee, dass maximal eine Leitung des Schaltkreises unabhängig von der diese Leitung beeinflussenden vorhergehenden Logik stets entweder den logischen Wert 0 (im Weiteren als *SSA-0-Fehler* bezeichnet) oder 1 (*SSA-1-Fehler*) führt. Prinzipiell kann dieser Effekt gleichgesetzt werden mit der Situation, in der die entsprechende Leitung dauerhaft mit der Versorgungsspannung V_{DD} (SSA-1-Fehler) oder 0V (SSA-0-Fehler) verbunden ist, was aber nicht die physikalische Ursache für den Fehler sein muss. Genauso ist es möglich, dass das die Leitung „treibende“ Gatter gegenüber der Spezifikation eine Fehlfunktion aufweist und unabhängig von der vorgesehenen Funktionalität dauerhaft den logischen Wert 0 oder 1 liefert.

Der Vorteil des *Single Stuck-At* Fehlermodells liegt in der Tatsache, dass es sich um ein *logisches* Fehlermodell handelt, welches von den möglichen physikalischen Ursachen einzelner Fehler abstrahiert und daher auch unabhängig von der Technologie ist, in der der betrachtete Schaltkreis umgesetzt wird. Zudem werden Aspekte wie das Zeitverhalten der Schaltung oder die Abhängigkeit von äußeren Einflüssen wie beispielsweise Temperatur und Luftfeuchtigkeit in diesem Fehlermodell nicht berücksichtigt. Trotz dieser vereinfachenden Annahmen hat sich gezeigt, dass das SSA-Fehlermodell in der Lage ist, viele der in der Realität auftretenden Design- und Fabrikationsfehler beim Entwurf und der Herstellung von Schaltkreisen zu detektieren.

In Abbildung 3.7 ist ein Beispiel aufgeführt, bei dem ausgehend von dem in Abbildung 3.7(a) dargestellten, fehlerfreien Schaltkreis angenommen sei, dass ein SSA-1-Fehler auf Leitung x_5 vorliegt. Dies bedeutet, dass unabhängig von der Belegung der Eingänge x_1 und x_2 des vorhergehenden UND-Gatters an x_5 dauerhaft der logische Wert 1 anliegt. Abbildung 3.7(b) zeigt dazu passend die graphische Darstellung dieses Fehlers, bei der die ursprüngliche Leitung aufgetrennt, mit x_5' ein neues Signal eingeführt und mit dem logischen Wert 1 beschaltet wird, während das ursprüngliche Signal x_5 mit der konstanten 0-Funktion verbunden wird.

Ein Testmuster für einen SSA-1-Fehler auf Leitung x_5 ist nun eine Belegung der Eingänge x_1, x_2 und x_3 des Schaltkreises, die so gewählt sein muss, dass zum Einen der Fehler am Fehlerort „eingestellt“ wird ($x_5 = 0$, $x_5' = 1$) und zum Anderen $x_4 \neq x_4'$ gilt, das heißt,



(a) Fehlerfreier Schaltkreis (b) *SSA-1-Fehler* auf Leitung x_5

Abbildung 3.7: Graphische Darstellung von SSA-Fehlern

fehlerfreier und fehlerbehafteter Schaltkreis bei genau diesem Testmuster ein unterschiedliches Ausgabeverhalten zeigen. Gelingt es, eine dahingehende Belegung der Eingänge zu bestimmen, kann für den gefertigten Schaltkreis geprüft werden, ob beispielsweise während der Produktion ein SSA-1-Fehler auf Leitung x_5 entstanden ist oder nicht. Dazu werden die Eingänge x_1, x_2 und x_3 mit dem entsprechenden Testmuster beschaltet und der am Ausgang der Schaltung anliegende logische Wert mit dem zuvor berechneten Wert des Ausgangs im fehlerfreien Fall verglichen. Liegt ein Unterschied vor, so ist die Schaltung fehlerhaft, ansonsten weist sie den getesteten SSA-1-Fehler nicht auf.

Die Generierung derartiger Testmuster zur Detektierung von SSA-Fehlern kann prinzipiell analog zum Vorgehen beim *Combinational Equivalence Checking* erfolgen, indem der Miter-Schaltkreis bestehend aus fehlerfreiem und fehlerbehaftetem Schaltkreis gebildet wird. Abbildung 3.8 zeigt den Miter für obiges Beispiel. Anstatt den Miter-Schaltkreis nun sofort in die entsprechende CNF-Darstellung zu überführen und die erzeugte CNF-Formel daraufhin mit einem SAT-Algorithmus zu lösen, können zunächst einige Optimierungen vorgenommen werden. Es fällt auf, dass die in Abbildung 3.8 blau unterlegten UND-Gatter jeweils die Funktion $x_5 \equiv x_1 \wedge x_2$ repräsentieren, wobei das untere UND-Gatter im fehlerbehafteten Schaltkreis dazu dient, mit $x_5 = 0$ den hier betrachteten SSA-1-Fehler am Fehlerort einzustellen. Der Sachverhalt kann durch Hinzunahme der Unit Clause $(\neg x_5)$ zur entstehenden CNF-Formel kodiert und das untere UND-Gatter daraufhin gelöscht werden. Abbildung 3.9 zeigt die dahingehende Optimierung des Miters. Weiterhin fällt auf, dass die beiden in Abbildung 3.9 blau hervorgehobenen Inverter mit $x_6 \equiv \neg x_3$ die identische Funktion berechnen, somit kann auf einen der beiden Inverter verzichtet werden. Der in diesem Sinne „finale“ Miter ist in Abbildung 3.10 dargestellt und die durch diesen Schaltkreis repräsentierte Funktion kann in die folgende erfüllbarkeitsäquivalente CNF-Darstellung überführt werden:

$$\begin{aligned} F_M = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge \\ & (x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_5 \vee x_6) \wedge \\ & (x'_4 \vee \neg x'_5) \wedge (x'_4 \vee \neg x_6) \wedge (\neg x'_4 \vee x'_5 \vee x_6) \wedge \\ & (\neg M \vee x_4 \vee x'_4) \wedge (\neg M \vee \neg x_4 \vee \neg x'_4) \wedge (M \vee \neg x_4 \vee x'_4) \wedge (M \vee x_4 \vee \neg x'_4) \wedge \\ & (M) \wedge (\neg x_5) \wedge (x'_5) \end{aligned}$$

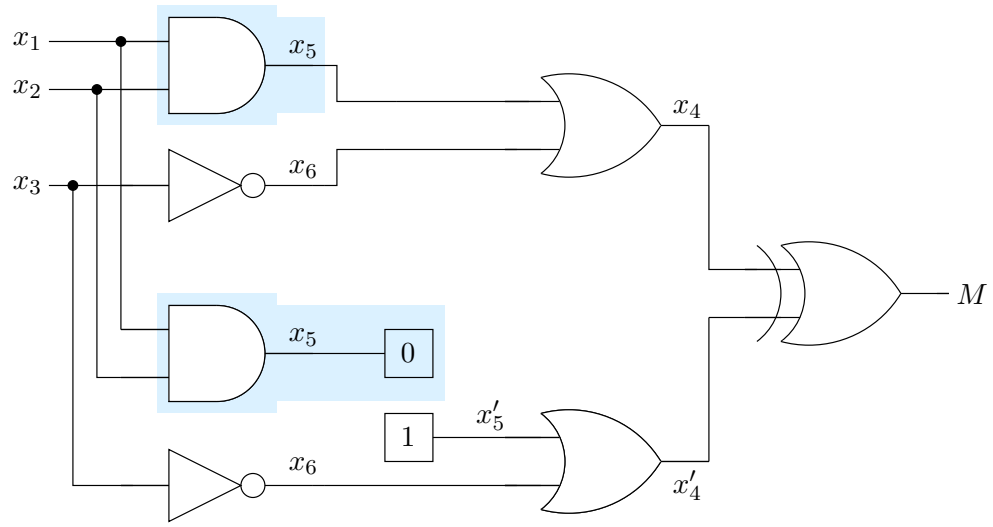


Abbildung 3.8: Miter-Schaltkreis bestehend aus fehlerfreiem und fehlerbehaftetem Schaltkreis mit *SSA-1-Fehler* auf Leitung x_5

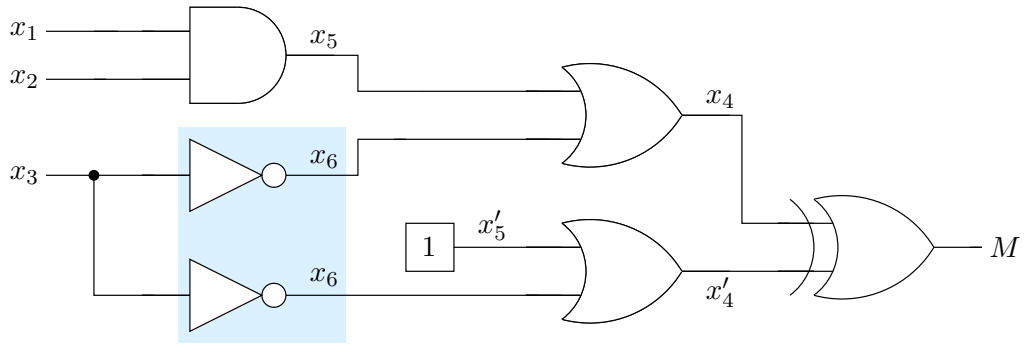


Abbildung 3.9: Optimierter Miter-Schaltkreis bestehend aus fehlerfreiem und fehlerbehaftetem Schaltkreis mit *SSA-1-Fehler* auf Leitung x_5

Erneut ist der CNF-Formel F_M die Unit Clause (M) hinzugefügt worden, um nur erfüllende Belegungen mit $M = 1$ zu erhalten, was auf einen Unterschied zwischen fehlerfreiem und fehlerbehaftetem Schaltkreis hinweist, während die Unit Clauses $(\neg x_5)$ und (x'_5) im fehlerbehafteten Schaltkreis die Existenz des SSA-1-Fehlers auf Leitung x_5 symbolisieren. Wie man leicht nachrechnet, kann F_M bereits vor der Übergabe an einen SAT-Algorithmus durch eine mehrmalige Anwendung der *Unit Clause* Regel des DLL-Algorithmus zu

$$F'_M = (\neg x_1 \vee \neg x_2) \wedge (x_3) \wedge (\neg x_6) \wedge (x'_4) \wedge (\neg x_4) \wedge (M) \wedge (\neg x_5) \wedge (x'_5)$$

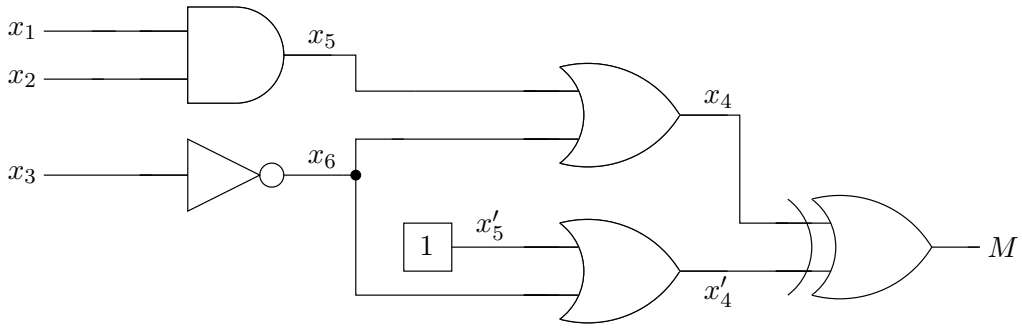


Abbildung 3.10: Finaler Miter-Schaltkreis bestehend aus fehlerfreiem und fehlerbehaftetem Schaltkreis (*SSA-1-Fehler* auf Leitung x_5)

vereinfacht werden. Die Bestimmung einer erfüllenden Belegung für die CNF-Formel F'_M ist trivial, da bis auf x_1 und x_2 alle Variablen per entsprechender Unit Clause in ihrem Wahrheitswert bereits festgelegt sind. Für x_1 und x_2 muss gelten, dass zumindest eine der beiden Variablen den Wahrheitswert 0 annimmt, um die Klausel $(\neg x_1 \vee \neg x_2)$ zu erfüllen. Insgesamt ergeben sich somit drei Testmuster, mit denen ein SSA-1-Fehler auf Leitung x_5 detektiert werden kann:

- $x_1 = 0, x_2 = 0, x_3 = 1$,
- $x_1 = 1, x_2 = 0, x_3 = 1$,
- $x_1 = 0, x_2 = 1, x_3 = 1$.

Bei allen drei Belegungen der Eingangssignale x_1, x_2 und x_3 nimmt der Ausgang x_4 des fehlerfreien Schaltkreises den Wert 0 an, während für den Ausgang des fehlerbehafteten Schaltkreises $x'_4 = 1$ gilt.

Das hier anhand eines Beispiels skizzierte, typische Vorgehen beim SAT-basierten *Automatic Test Pattern Generation* lässt sich in vereinfachter Form wie folgt zusammenfassen: zunächst wird in Abhängigkeit vom zu testenden Fehler der Miter bestehend aus fehlerfreiem und fehlerbehaftetem Schaltkreis gebildet und nach Möglichkeit vereinfacht. Danach wird die durch den Miter berechnete Funktion in eine CNF-Formel übersetzt und von einem SAT-Algorithmus bearbeitet. Stellt sich dabei heraus, dass die Probleminstanz erfüllbar ist, so ist der betrachtete Fehler „testbar“ und die gefundene erfüllende Belegung stellt ein entsprechendes Testmuster dar. Ansonsten handelt es sich um einen redundanten Fehler, der an den Ausgängen des Schaltkreises, unabhängig von der Beschaltung der Eingangs-Pins, kein Fehlverhalten verursacht. Das hat zur Folge, dass der gefertigte Schaltkreis gegebenenfalls diesen speziellen redundanten Fehler aufweist, aber keine Möglichkeit besteht, diesen Sachverhalt zu überprüfen.

Dieses Grundkonzept bietet eine Reihe von Optimierungsmöglichkeiten [66, 78, 102, 107]. Exemplarisch seien Ansätze genannt, bei denen der eigentlichen CNF-Formel in Form zusätzlicher Klauseln weitere strukturelle Informationen über den (Miter-)Schaltkreis hinzugefügt werden. Die Kernidee besteht darin, den Suchprozess des SAT-Algorithmus durch die zusätzlichen Klauseln dahingehend besser zu steuern, dass Kombinationen von Variablenzuweisungen unterbunden werden, die eine Propagation des Fehlers vom Fehlerort hin zu den Ausgängen der Schaltung unmöglich machen. Weiterhin sei angeführt, dass der Einsatz von SAT-Algorithmen im Rahmen der Testmustergenerierung nicht allein auf Szenarien beschränkt ist, in denen das *Single Stuck-At* Fehlermodell betrachtet wird. Beispielsweise wird in [35] ein SAT-basiertes Verfahren zur Testmustergenerierung vorgestellt, das auf dem Fehlermodell der *Resistive Bridging Faults* beruht, während in [34] das so genannte *Path Delay Fault Model* betrachtet wird.

Kapitel 4

Sequentielle SAT-Algorithmen

Dieses Kapitel widmet sich der Funktionsweise moderner sequentieller SAT-Algorithmen, wobei der Fokus auf so genannten vollständigen Verfahren im Stil der in Abschnitt 2.5 vorgestellten DLL-Prozedur liegt. Der Begriff *vollständig* bezieht sich in diesem Zusammenhang auf ein systematisches Durchsuchen des gesamten durch eine CNF-Formel aufgespannten Suchraums. Derartige Ansätze sind insbesondere in der Lage, die Unerfüllbarkeit einer Probleminstanz zu belegen (ein entsprechend großes Zeitlimit vorausgesetzt). Im Wesentlichen werden mit diesem Kapitel zwei Ziele verfolgt. Einerseits wird ein Überblick über die essentiellen Techniken leistungsstarker sequentieller SAT-Verfahren gegeben und aufgezeigt, mit welchen Methoden Algorithmen wie BerkMin [44], MiniSat [32], SatELite [31], Siege [67] und zChaff [86] versuchen, den Suchprozess zu beschleunigen. Andererseits wird als Vorarbeit für Kapitel 7 bis 9 dargelegt, auf welchen Konzepten die sequentiellen SAT-Prozeduren von PIChaff, MiraXT und PaMiraXT aufbauen. Auf eine explizite Diskussion darüber hinausgehender Aspekte wie etwa *Incremental SAT Solving* [33, 50] und *Multi-Valued SAT Solving* [73] wird verzichtet, hier sei auf die jeweilige Literatur verwiesen.

4.1 Überblick

Algorithmus 4.1 beschreibt das Grundgerüst heutiger sequentieller SAT-Algorithmen, das auf diesem abstrakten Niveau für nahezu alle Verfahren identisch ist. Zunächst fällt auf, dass es sich im Gegensatz zur DLL-Prozedur nicht um ein rekursives Verfahren handelt. Den Kern bildet die in den Zeilen 5 bis 20 angegebene while-Schleife, der je nach Implementierung eine so genannte *Preprocessing*-Routine vorgeschaltet ist. Diese als PreprocessCNF bezeichnete Funktion dient dazu, noch vor der eigentlichen Suche nach einer erfüllenden Belegung die gegebene CNF-Formel nach Möglichkeit so zu vereinfachen, dass der folgende Suchprozess beschleunigt werden kann. Motiviert ist dieses Vorgehen durch die Beobachtung, dass die zum Lösen einer Problemstellung benötigte Laufzeit eines SAT-Algorithmus in der Regel von der Größe der Formel abhängt, also eine „kleinere“ CNF-Formel im Allgemeinen schneller gelöst werden kann als eine „große“. Bei der Umsetzung gilt es zu beachten, das Preprocessing so effizient zu gestalten, dass der dafür benötigte Zeitaufwand nicht den zu erwartenden Laufzeitvorteil des nachfolgenden Suchprozesses dominiert, da ansonsten keine Reduktion der Gesamtlaufzeit des SAT-Algorithmus erreicht werden kann. Stellt sich bereits während der Vorverarbeitung die Unerfüllbarkeit

Algorithmus 4.1 Hauptroutine sequentieller SAT-Algorithmen

```
 1: bool SequentialSatEngine(CNF F)
 2: {
 3:   if (PreprocessCNF(F) == CONFLICT)    // Vorverarbeitung der CNF-Formel.
 4:     { return UNSATISFIABLE; }                    // Problem unerfüllbar.
 5:   while (true)
 6:     {
 7:       if (DecideNextBranch())              // Wahl der nächsten freien Variablen.
 8:         {
 9:           while (BCP() == CONFLICT)       // Boolean Constraint Propagation.
10:             {
11:               BLevel = AnalyzeConflict();                // Konflikt-Analyse.
12:               if (BLevel > 0)
13:                 { Backtrack(BLevel); }               // Zuweisungen aufheben.
14:               else
15:                 { return UNSATISFIABLE; }            // Problem unerfüllbar.
16:             }
17:         }
18:       else
19:         { return SATISFIABLE; }    // Alle Variablen belegt, Problem erfüllbar.
20:     }
21: }
```

der Probleminstanz heraus, stoppt der SAT-Algorithmus in Zeile 4 mit der Ausgabe *UNSATISFIABLE*.

Im Anschluss an die Vorverarbeitung der Probleminstanz beginnt die Hauptschleife des SAT-Algorithmus. Mit der Funktion DecideNextBranch wird eine freie Variable ausgewählt und dieser einer der beiden Wahrheitswerte 0 oder 1 zugewiesen. Die implementierte Strategie wird auch als *Entscheidungsheuristik* (*Decision Heuristic*) bezeichnet, die jeweils gewählte Variable als *Entscheidungsvariable* (*Decision Variable*).

Als Folge der von DecideNextBranch gewählten und mit einem Wahrheitswert belegten Decision Variable werden üblicherweise einige Klauseln zu Unit Clauses, das heißt, bis auf ein unbelegtes Literal sind alle weiteren Literale „falsch“ belegt und erfüllen die entsprechende Klausel nicht.[1] Mittels der so genannten *Boolean Constraint Propagation* (Funktion BCP, Zeile 9) werden alle Unit Clauses ermittelt und durch eine entsprechende Zuweisung an das verbliebene, unbelegte Literal erfüllt. Man spricht bei derartig erzwun-

[1] In diesem Zusammenhang sei angemerkt, dass im Gegensatz zur DLL-Prozedur bei heutigen SAT-Algorithmen falsch belegte Literale nicht explizit aus den entsprechenden Klauseln entfernt werden.

genen Zuweisungen, bei denen ein Literal auf einen bestimmt Wert gesetzt werden muss, um sowohl die entsprechende Unit Clause zu erfüllen als auch die Chance zu wahren, die gesamte Formel erfüllen zu können, von *Implikationen*. Sei beispielsweise die CNF-Formel $F = (x_1 \vee x_2)$ gegeben: in dieser Situation erzwingt (impliziert) die Zuweisung $x_1 = 0$ die Zuweisung $x_2 = 1$, da F ansonsten nicht erfüllt ist. Gleiches gilt für $x_2 = 0$, in diesem Fall wird die Implikation $x_1 = 1$ ausgelöst. Im Rahmen der Boolean Constraint Propagation werden dabei nicht nur alle Implikationen ermittelt, die sich direkt als Konsequenz einer gewählten Decision Variable ergeben, sondern auch diejenigen Implikationen, die „indirekt“ aufgrund vorangegangener Implikationen gefolgert werden können. Exemplarisch sei die CNF-Formel $G = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$ angenommen. Die Zuweisung $x_1 = 0$ impliziert zunächst nur $x_2 = 1$, aufgrund der zweiten Klausel erzwingen beide Zuweisungen zusammen aber darüber hinaus noch die Implikation $x_3 = 1$. Erst wenn mittels dieser Methodik keine weitere Implikation mehr bestimmt werden kann, stoppt die BCP-Routine, bei der es sich im Kern um die Umsetzung der *Unit Clause* Regel der DLL-Prozedur handelt.

Hat sich im Verlauf der Boolean Constraint Propagation keine widersprüchliche Belegung ergeben, bei der eine Variable x_i sowohl den Wahrheitswert 0 als auch 1 annehmen müsste, um alle Klauseln der gegebenen Formel F zu erfüllen, wird erneut die Entscheidungsheuristik aufgerufen und die nächste Decision Variable bestimmt. Kann zu einem Zeitpunkt des Suchprozesses keine Decision Variable mehr gewählt werden, so wurden alle Variablen widerspruchsfrei belegt und der Algorithmus stoppt mit dem Ergebnis *SATISFIABLE* (Zeile 19). Üblicherweise wird auch das ermittelte Modell ausgegeben (nicht explizit in Algorithmus 4.1 dargestellt). An dieser Stelle sei auf einen wesentlichen Unterschied zwischen der DLL-Prozedur und heutigen SAT-Algorithmen hingewiesen. Während beim DLL-Algorithmus nach jeder (implizit) vorgenommenen Variablenzuweisung in Zeile 3 von Algorithmus 2.2 geprüft wird, ob die Formel bereits durch die aktuelle (Teil-)Belegung der Variablen erfüllt ist, wird in modernen SAT-Algorithmen auf einen derartigen Test verzichtet. Der Grund ist darin zu sehen, dass ein solcher Test nur mit erheblichem Aufwand und dem Mitführen entsprechender Statusvariablen zu realisieren wäre, die wiederum insbesondere während jeder *Backtrack*-Operation aktualisiert werden müssten. Stattdessen wird der Test auf eine erfüllende Belegung darauf zurückgeführt, ob noch freie Variablen existieren, somit besteht jedes Modell immer aus Zuweisungen an alle in der Formel enthaltenen Variablen.

Wird während der Durchführung der Boolean Constraint Propagation eine widersprüchliche Belegung (ein so genannter *Konflikt*) festgestellt, wird mit der Funktion AnalyzeConflict die Konflikt-Analyse aufgerufen. Ein Konflikt liegt immer dann vor, wenn eine Variable x_i zeitgleich die Wahrheitswerte 0 und 1 annehmen müsste, um alle Klauseln der gegebenen CNF-Formel zu erfüllen. Das Ziel der Konflikt-Analyse ist es, diejenigen Variablenzuweisungen zu identifizieren, die verantwortlich für den Konflikt sind, was in der Regel nur auf einige wenige Zuweisungen der aktuellen (Teil-)Belegung zutrifft. Per Reso-

lution werden solange am Konflikt beteiligte Klauseln miteinander resolviert, bis schlussendlich in der „finalen“ Resolventen (der so genannten *Konflikt-Klausel*) all diejenigen Literale enthalten sind, die sich für die widersprüchliche Belegung verantwortlich zeigen. Bedingt durch das in Lemma 2.1 angegebene Resolutions-Lemma kann die so entstandene Konflikt-Klausel ohne den Verlust der Äquivalenz der bisherigen Klauselmenge der gegebenen CNF-Formel hinzugefügt werden. Sei als Beispiel mit $G = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge \ldots$ ein Ausschnitt einer CNF-Formel G gegeben und sei ferner angenommen, dass als erste Entscheidungsvariable x_1 mit der Belegung $x_1 = 0$ gewählt wurde. Offensichtlich führt diese Zuweisung zu einem Konflikt, da zeitgleich $x_2 = 1$ und $x_2 = 0$ gelten müsste, um sowohl die erste als auch die zweite Klausel von G erfüllen zu können. Als verantwortlich für diesen Widerspruch kann die Zuweisung $x_1 = 0$ identifiziert werden, was zur Konflikt-Klausel (x_1) führt (Anwendung der Resolutionsregel auf die Klauseln $(x_1 \vee x_2)$ und $(x_1 \vee \neg x_2)$ bezüglich x_2). Durch Hinzunahme dieser Unit Clause zur Klauselmenge von G wird für die gesamte noch verbleibende Suche nach einer erfüllenden Belegung die Zuweisung $x_1 = 1$ erzwungen, so dass auf diesem Weg ein erneutes Eintreten des identischen Konflikts verhindert wird. Das Vorgehen bestehend aus der Analyse eines Konflikts und dem Hinzufügen der hergeleiteten Konflikt-Klausel wird in der Literatur auch als *Conflict Driven Learning* bezeichnet. Jede Konflikt-Klausel repräsentiert einen Teil des durch die CNF-Formel aufgespannten Suchraums, der bereits als unerfüllbar identifiziert wurde und im weiteren Verlauf nicht erneut betrachtet werden muss. Während der Suche nach einer erfüllenden Belegung lernt ein SAT-Algorithmus auf diesem Weg immer mehr Konflikt-Klauseln und somit Informationen über die gegebene Probleminstanz, die das zu betrachtende Restproblem immer weiter einschränken. So verwundert es nicht, dass *Conflict Driven Learning* einen maßgeblichen Anteil an der enormen Performance heutiger SAT-Verfahren hat.

Neben der Identifizierung der einen Konflikt auslösenden Variablenzuweisungen besteht die zweite Aufgabe der Funktion AnalyzeConflict darin, einen so genannten *Backtrack Level* zu bestimmen. Vereinfacht ausgedrückt gibt der Backtrack Level an, welche der Variablenzuweisungen mittels der Funktion Backtrack rückgängig gemacht werden müssen, um den derzeitigen Konflikt aufzulösen und die Suche nach einer erfüllenden Belegung fortsetzen zu können. Lässt sich der Konflikt selbst bei einer Rücknahme der gesamten (Teil-)Belegung der Variablen nicht verhindern, was einem Backtrack Level von 0 entspricht, ist die Probleminstanz unerfüllbar und der SAT-Algorithmus stoppt in Zeile 15 mit der Meldung *UNSATISFIABLE*.

Wie zu Beginn angedeutet, folgen moderne SAT-Algorithmen auf Basis der DLL-Prozedur durchgängig dem zuvor skizzierten Grundgerüst und unterscheiden sich im Wesentlichen in der Umsetzung der Funktionen PreprocessCNF, DecideNextBranch, BCP, AnalyzeConflict und Backtrack und den dafür benötigten Datenstrukturen. In den Abschnitten 4.2 bis 4.5 werden effiziente Realisierungen für die einzelnen Bereiche vorgestellt. Anschließend werden mit dem Löschen von Konflikt-Klauseln (Abschnitt 4.6) und dem Konzept der Neustarts (Abschnitt 4.7) zwei weitere Routinen diskutiert, die der Über-

sichtlichkeit halber nicht explizit in Algorithmus 4.1 aufgeführt sind, mittlerweile aber ebenfalls zum Standardrepertoire eines SAT-Algorithmus gehören. Insbesondere wird in den entsprechenden Abschnitten als Vorarbeit für Kapitel 7 bis 9 aufgezeigt, auf welchen Methoden und Techniken die jeweiligen Funktionen in PIChaff, MiraXT und PaMiraXT aufbauen.

Zum Abschluss dieses Überblicks wird mit dem *Decision Stack* die zentrale Datenstruktur vorgestellt, mit der die Position moderner SAT-Algorithmen innerhalb des durch die CNF-Formel aufgespannten Suchraums repräsentiert wird. Zur Illustrierung sei die in Beispiel 4.1 dargestellte CNF-Formel F gegeben (die mit jeder Klausel assoziierte Nummer, die Klausel-ID, dient der eindeutigen Referenzierung der Klauseln und wird in den Abschnitten 4.4 und 4.5 benötigt).

Beispiel 4.1
Sei F eine CNF-Formel, für die ein Ausschnitt der Klauselmenge wie folgt gegeben sei:

$$F = \underbrace{(x_{23})}_{1} \wedge \underbrace{(x_7 \vee \neg x_{23})}_{2} \wedge \underbrace{(x_6 \vee \neg x_{17})}_{3} \wedge \underbrace{(x_6 \vee \neg x_{11} \vee \neg x_{12})}_{4} \wedge \underbrace{(x_{13} \vee x_8)}_{5} \wedge$$
$$\underbrace{(\neg x_{11} \vee x_{13} \vee x_{16})}_{6} \wedge \underbrace{(x_{12} \vee \neg x_{16} \vee \neg x_2)}_{7} \wedge \underbrace{(x_2 \vee \neg x_4 \vee \neg x_{10})}_{8} \wedge$$
$$\underbrace{(\neg x_{19} \vee x_4)}_{9} \wedge \underbrace{(x_{10} \vee \neg x_5)}_{10} \wedge \underbrace{(x_{10} \vee x_3)}_{11} \wedge \underbrace{(x_{10} \vee \neg x_8 \vee x_1)}_{12} \wedge$$
$$\underbrace{(\neg x_{19} \vee \neg x_{18} \vee \neg x_3)}_{13} \wedge \underbrace{(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)}_{14} \wedge \ldots$$

Bedingt durch den rekursiven Programmentwurf ist beim klassischen DLL-Algorithmus stets ersichtlich, in welcher Reihenfolge (auf welcher Rekursionsebene) die einzelnen Variablen gewählt, nacheinander die möglichen Zuweisungen geprüft und welche Implikationen dadurch jeweils ausgelöst wurden. Bei der in Algorithmus 4.1 aufgezeigten Version eines heutigen SAT-Algorithmus ist dies nicht gegeben, diese Informationen müssen mittels entsprechender Datenstrukturen mitgeführt werden. Deshalb wird mit jeder Variablen ein *Decision Level* assoziiert, der angibt, auf welchem Level eine Zuweisung an die jeweilige Variable vorgenommen wurde. Der *Decision Level* (auch als *Entscheidungsebene* bezeichnet) wird zu Beginn mit 0 initialisiert und lediglich vor der Wahl einer Decision Variable um eins inkrementiert. Das hat zur Folge, dass jede Entscheidungsvariable sowie alle von dieser ausgelösten Implikationen auf dem gleichen Decision Level belegt werden.

Decision Level 0 nimmt in diesem Zusammenhang eine Sonderrolle ein. Auf dieser Ebene werden alle Implikationen gespeichert, die sich aus Unit Clauses ergeben beziehungsweise als Folge davon hergeleitet werden können (unabhängig, ob in der initialen Formel enthalten oder während der Suche gefolgert). Die in Beispiel 4.1 gegebene CNF-Formel F enthält

mit (x_{23}) eine Unit Clause, anhand derer bereits beim Einlesen der Probleminstanz und noch vor dem Start des eigentlichen Suchprozesses des SAT-Algorithmus die Implikation $x_{23} = 1$ gefolgert werden kann. Wie zuvor festgelegt, wird mit jeder Variablen ein Decision Level assoziiert, der zu Beginn mit 0 initialisiert wird. Die Zuweisung $x_{23} = 1$ erfolgt daher auf Decision Level 0, was ebenfalls für die sich aus $x_{23} = 1$ mit Hilfe der Klausel $(x_7 \vee \neg x_{23})$ ergebende Implikation $x_7 = 1$ gilt. Das Identifizieren und Bearbeiten bereits in der initialen CNF-Formel enthaltener Unit Clauses kann als eine Art *Preprocessing* angesehen werden, dass von allen modernen SAT-Algorithmen unterstützt wird. In dieser Phase bereits erfüllte Klauseln werden im Allgemeinen aus der initialen CNF-Formel entfernt, da sie während des gesamten Suchprozesses erfüllt sind. Gleiches gilt für „falsch“ belegte Literale. In dem hier gewählten Beispiel hätte dies ein Löschen der beiden ersten Klauseln zur Folge.

Zur Abspeicherung der Reihenfolge der einzelnen Zuweisungen, um stets feststellen zu können, welche Zuweisungen welche Implikationen ausgelöst haben, bedienen sich SAT-Verfahren des so genannten *Decision Stacks*. Dieser speichert, geordnet nach Entscheidungsebenen, die Reihenfolge der auf dem jeweiligen Decision Level getätigten Zuweisungen. Abbildung 4.1 zeigt dies exemplarisch für die beiden Zuweisungen $x_{23} = 1$ und $x_7 = 1$, die genau in dieser Reihenfolge auf Decision Level 0 vorgenommen wurden.

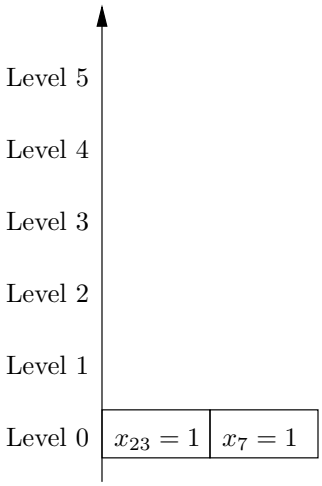


Abbildung 4.1: Decision Stack zu Beispiel 4.1; Zuweisungen auf Decision Level 0

Anhand der zuvor gemachten Ausführungen wird deutlich, warum in Zeile 12 von Algorithmus 4.1 ein Backtrack Level von 0 zu einem Abbruch des SAT-Algorithmus und zur Ausgabe *UNSATISFIABLE* führt. Da sich auf Decision Level 0 nur Zuweisungen befinden, die sich aus Unit Clauses ergaben (und damit unabänderlich sind), ist auf diesem Decision Level jede widersprüchliche Belegung wie $x_i = 0$ und $x_i = 1$ unabhängig von der Wahl einer Decision Variable. Es besteht somit keine Möglichkeit, eine Zuweisung an eine

Decision Variable zurückzunehmen und stattdessen den entsprechenden komplementären Wahrheitswert zu wählen, woraus folgt, dass die Probleminstanz unerfüllbar ist.

Nach dem Ende der Vorverarbeitung startet der eigentliche SAT-Algorithmus mit der Wahl der ersten Decision Variable, beispielsweise $x_6 = 0$. In Abbildung 4.2 ist der Decision Stack dargestellt. Wie zuvor festgelegt, wird vor jeder Wahl einer Decision Variable zunächst der Decision Level um eins inkrementiert, weshalb die Zuweisung $x_6 = 0$ auf Decision Level 1 erfolgt. Zur besseren Unterscheidung von Implikationen und Entscheidungsvariablen sind alle Implikationen weiß unterlegt, während die Entscheidungsvariablen grau unterlegt sind.

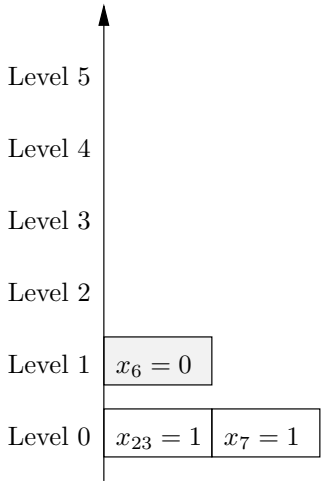


Abbildung 4.2: Decision Stack zu Beispiel 4.1; Wahl der ersten Decision Variable

Die Zuweisung $x_6 = 0$ löst zusammen mit der Klausel $(x_6 \vee \neg x_{17})$ die Implikation $x_{17} = 0$ aus, die ebenfalls auf Decision Level 1 vorgenommen und innerhalb des Decision Stacks direkt im Anschluss an $x_6 = 0$ abgelegt wird. Abbildung 4.3 illustriert diesen Sachverhalt.

Da sich aufgrund von $x_6 = 0$ und $x_{17} = 0$ keine weiteren Implikationen ergeben, verlässt der SAT-Algorithmus an dieser Stelle den Decision Level 1 und fährt mit der Auswahl der nächsten Decision Variable auf Decision Level 2 fort. Abbildung 4.4 veranschaulicht ein mögliches Vorgehen, bei dem

- auf Decision Level 2 die Decision Variable $x_{13} = 0$,
- auf Decision Level 3 die Decision Variable $x_{19} = 1$,
- auf Decision Level 4 die Decision Variable $x_{54} = 0$ und
- auf Decision Level 5 die Decision Variable $x_{11} = 1$

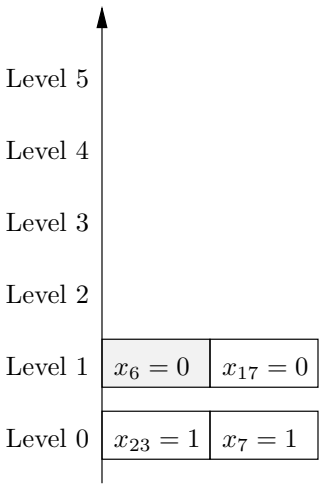


Abbildung 4.3: Decision Stack zu Beispiel 4.1; Implikationen auf Decision Level 1

ausgewählt wurde. Anhand der in Beispiel 4.1 angegebenen CNF-Formel F ist leicht nachzuvollziehen, dass die Wahl von $x_{11} = 1$ zusammen mit den Zuweisungen der vorherigen Entscheidungsebenen nacheinander die Implikationen $x_{12} = 0$, $x_{16} = 1$, $x_2 = 0$, $x_{10} = 0$, $x_5 = 0$, $x_3 = 1$, $x_1 = 1$ sowie $x_{18} = 0$ und $x_{18} = 1$ auslöst (in der Abbildung rot markiert und mit $x_{18} = 0/1$ angedeutet).

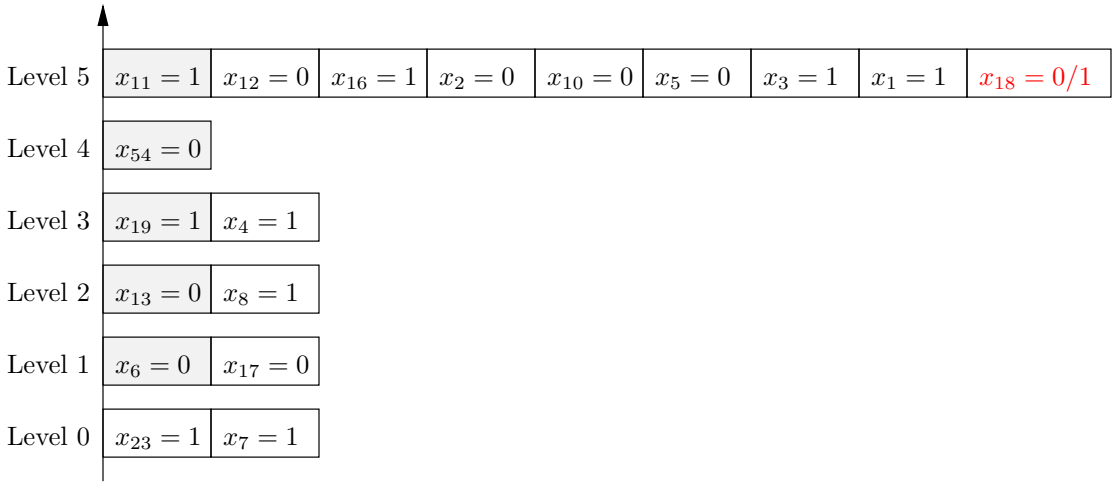


Abbildung 4.4: Decision Stack zu Beispiel 4.1; Konflikt auf Decision Level 5

Offensichtlich liegt ein Konflikt vor, da x_{18} sowohl den Wert 0 als auch den Wert 1 annehmen müsste, um alle Klauseln von F zu erfüllen. In dieser Situation werden beim klassischen

DLL-Algorithmus zunächst alle Variablenzuweisungen bis einschließlich der Zuweisung der letzten Fallunterscheidung rückgängig gemacht. Die im Rahmen der letzten Fallunterscheidung ausgewählte Variable wird dann in ihrer Belegung invertiert und der Suchprozess fortgesetzt. Man spricht hierbei auch vom so genannten *Chronological Backtracking*, da immer nur bis zur letzten Fallunterscheidung, bei der noch nicht beide möglichen Zuweisungen getestet wurden, zurückgesprungen und zunächst die sich aus der entsprechenden komplementären Zuweisung ergebende Situation analysiert wird, bevor gegebenenfalls noch weiter zurückgegangen wird.

Im Vergleich zur DLL-Prozedur gehen moderne SAT-Algorithmen im Falle eines Konflikts grundlegend anders vor. Zunächst wird während der Konflikt-Analyse per Resolution eine Konflikt-Klausel bestimmt, deren Literale die für die widersprüchliche Belegung verantwortlichen Variablenzuweisungen repräsentieren. Die miteinander resolvierten Klauseln hängen dabei von der Definition „für einen Konflikt verantwortlich“ ab, was zur Folge hat, dass die für einen bestimmten Konflikt hergeleitete Konflikt-Klausel je nach Implementierung variiert. In Abschnitt 4.5 werden diesbezüglich verschiedene Konzepte vorgestellt. In der vorliegenden Situation würde beispielsweise bei der Anwendung des dort eingeführten *1UIP*-Prinzips die Konflikt-Klausel $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$ hergeleitet und zur Klauselmenge von F hinzugefügt werden.

Auch bei der Backtrack-Operation unterscheiden sich heutige SAT-Verfahren erheblich von der DLL-Prozedur. Statt nur die Zuweisungen des aktuellen Decision Levels rückgängig zu machen und den Wahrheitswert der letzten Entscheidungsvariablen zu invertieren, wird nach Möglichkeit über mehrere Ebenen hinweg „zurückgesprungen“, was auch als *Non-Chronological Backtracking* bezeichnet wird. Betrachtet man die in diesem Beispiel gefolgerte Konflikt-Klausel $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$, so fällt auf, dass mit Ausnahme von x_{10} alle anderen Zuweisungen auf Decision Level 3 oder früher getätigt wurden. Das heißt, wenn diese Klausel Teil der initialen CNF-Formel F gewesen wäre, hätte sie auf Decision Level 3 die Implikation $x_{10} = 1$ ausgelöst. Beim Non-Chronological Backtracking wird dieser Idee Rechnung getragen, indem im Anschluss an die Konflikt-Analyse auf den Decision Level zurückgesprungen wird, auf dem die entsprechende Implikation ausgelöst wird. In dem hier diskutierten Beispiel werden dabei die Entscheidungsebenen 4 und 5 übersprungen, bevor der Suchprozess auf Decision Level 3 mit der Bearbeitung der aktuellen Konflikt-Klausel beziehungsweise der dadurch ausgelösten Implikation fortgesetzt wird. In diesem Zusammenhang werden Konflikt-Klauseln auch als *Asserting Clauses* bezeichnet, da sie direkt im Anschluss an das Backtracking eine Implikation auslösen [125]. Abbildung 4.5 illustriert das Vorgehen anhand des Decision Stacks, der sich nach der entsprechenden Backtrack-Operation und der Berücksichtigung der Implikation $x_{10} = 1$ ergibt, wobei anhand der Zuweisungen $x_4 = 1$ und $x_{10} = 1$ sowie der Klausel $(x_2 \vee \neg x_4 \vee \neg x_{10})$ unmittelbar die Implikation $x_2 = 1$ gefolgert werden kann.

Die Ausführungen verdeutlichen auch, warum in Algorithmus 4.1 nach der Durchführung

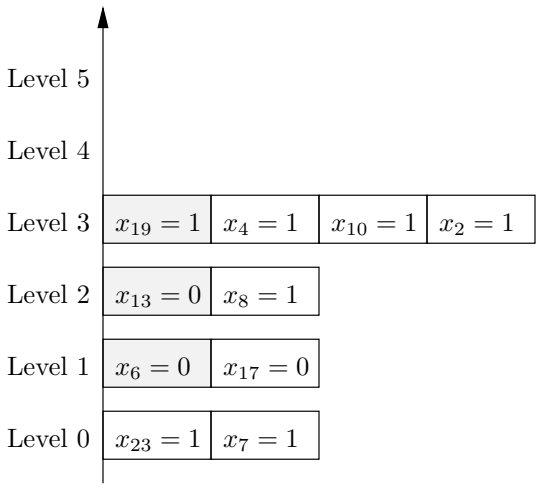


Abbildung 4.5: Decision Stack zu Beispiel 4.1; Backtracking auf Decision Level 3

einer Backtrack-Operation zunächst erneut die *Boolean Constraint Propagation* aufgerufen und die Suche nach einer erfüllenden Belegung nicht sofort mit der Wahl der nächsten Decision Variable fortgesetzt wird. Zunächst müssen die sich durch die hergeleitete Konflikt-Klausel „direkt“ ergebende Implikation (in diesem Beispiel $x_{10} = 1$) sowie alle Folgeimplikationen wie etwa $x_2 = 1$ bearbeitet werden.

4.2 Preprocessing

Unter dem Begriff *Preprocessing* versteht man eine im Vorfeld des eigentlichen Suchprozesses durchgeführte Modifikation der gegebenen CNF-Formel, so dass diese, selbstverständlich ohne deren Erfüllbarkeit zu verändern, schneller von einem SAT-Algorithmus gelöst werden kann. Das Ziel ist dabei, mit geeigneten Techniken Literale und Klauseln aus der originalen Problembeschreibung zu entfernen und so die Größe der CNF-Formel in Einklang mit Definition 2.6 zu verringern. Motiviert wird das Preprocessing durch die Beobachtung, dass im Allgemeinen die Größe einer CNF-Formel mit der zur Lösung benötigten Laufzeit eines SAT-Algorithmus korrespondiert, sich also eine kleinere Formel im Allgemeinen auch schneller lösen lässt. Besonders während der *Boolean Constraint Propagation* Phase kann sich eine kleinere Formel positiv bemerkbar machen, da weniger Klauseln bedeuten, dass potenziell auch weniger Klauseln als Auslöser von Implikationen und Konflikten in Frage kommen und dahingehend analysiert werden müssen. Wichtig für den Erfolg der Vorverarbeitung ist es, das Verhältnis zwischen der Laufzeit des SAT-Algorithmus und dem mit dem Preprocessing verbundenen, zeitlichen Mehraufwand zu berücksichtigen: ein Vorteil stellt sich nur dann ein, wenn die Vorverarbeitung so schnell und effizient ausgeführt werden kann, dass der Zeitgewinn durch eine beschleunigte Suche den Mehraufwand des Prepro-

cessings übertrifft.

Im vorherigen Abschnitt wurde bereits angedeutet, dass mit der Identifikation und Verarbeitung von in der Ausgangsformel enthaltenen Klauseln, die lediglich aus einem Literal bestehen, eine einfache Form des Preprocessings in allen modernen SAT-Algorithmen integriert ist. Kann eine Unit Clause (L) innerhalb der initialen CNF-Formel detektiert werden, so wird die notwendige Variablenzuweisung $L = 1$ auf Decision Level 0 fixiert und alle durch die Unit Clause subsumierten Klauseln sowie alle Vorkommen des Literals $\neg L$ aus der Formel entfernt. Das Vorgehen entspricht dabei der *Unit Clause* Regel der DP- beziehungsweise DLL-Prozedur. Die darüber hinausgehenden, aus der Literatur bekannten Techniken lassen sich in zwei Gruppen einteilen: einerseits Methoden, die auf der Anwendung der *Boolean Constraint Propagation* beruhen, andererseits Methoden auf Basis der Resolutionsregel.

Zur Gruppe der ersten Preprocessing-Routinen gehören Arbeiten wie etwa [68, 76], die sich wie folgt charakterisieren lassen: für jede Variable x_i der gegebenen CNF-Formel (beziehungsweise für eine festgelegte Teilmenge aller Variablen) werden der Reihe nach die beiden Zuweisungen $x_i = 1$ und $x_i = 0$ angenommen und per *Boolean Constraint Propagation* evaluiert, welche Implikationen daraus jeweils folgen. In Abhängigkeit vom Ergebnis können dann unter Umständen Schlussfolgerungen gezogen werden, mit denen sich die Klauselmenge vereinfachen lässt:

– *Identifizierung von notwendigen Variablenzuweisungen*: führt beispielsweise für eine Variable x_i die Zuweisung $x_i = 1$ zu einem Konflikt, kann sofort die Zuweisung $x_i = 0$ vorgenommen werden, um den Konflikt zu vermeiden. Gleiches gilt, falls sich aus $x_i = 0$ ein Konflikt ergibt, in dieser Situation kann direkt $x_i = 1$ gefolgert werden. Lässt sich in diesem Zusammenhang per Boolean Constraint Propagation belegen, dass sowohl $x_i = 0$ als auch $x_i = 1$ einen Konflikt auslösen, ist die Probleminstanz unerfüllbar, da in jeder erfüllenden Belegung die Variable x_i einen der beiden Wahrheitswerte annehmen muss.

  Eine weitere Möglichkeit der Identifizierung von notwendigen Variablenzuweisungen besteht, wenn sowohl $x_i = 1$ als auch $x_i = 0$ beispielsweise die Implikation $x_j = 1$ erzwingen. Da für jede erfüllende Belegung entweder $x_i = 1$ oder $x_i = 0$ gilt und beide Zuweisungen $x_j = 1$ implizieren, kann direkt auf die Zuweisung $x_j = 1$ geschlossen werden.

  Formal lässt sich der letzte Punkt folgendermaßen begründen: impliziert die Zuweisung $x_i = 1$ die Zuweisung $x_j = 1$, kann dies mittels der Klausel $(\neg x_i \vee x_j)$ repräsentiert werden. Gilt nun gleichermaßen, dass $x_i = 0$ die Zuweisung $x_j = 1$ impliziert, so lautet die entsprechende Klausel $(x_i \vee x_j)$. Per Resolution kann nun die Resolvente (x_j) gebildet und zur Klauselmenge hinzugenommen werden, so dass stets $x_j = 1$ gilt.

An dieser Stelle sei darauf hingewiesen, dass diese Art der Bestimmung notwendiger Zuweisungen eine der Kerntechniken der Methode von Stålmarck darstellt [101].

– *Identifizierung von äquivalenten Variablen*: lässt sich per BCP-Operation für $x_i = 1$ zeigen, dass $x_j = 1$ gilt und folgt umgekehrt aus $x_i = 0$ die Implikation $x_j = 0$, so ist gezeigt, dass x_i und x_j äquivalent sind. In diesem Fall können entweder alle Vorkommen von x_i in der Klauselmenge durch x_j oder alle Vorkommen von x_j durch x_i ersetzt werden.

Nach jeder Durchführung der probeweisen Zuweisung an eine Variable mit anschließender BCP-Operation wird zunächst geprüft, ob sich die CNF-Formel aufgrund der erzielten Erkenntnisse vereinfachen lässt. Kann etwa gefolgert werden, dass stets $x_j = 1$ gelten muss, können alle Vorkommen von x_j als negatives Literal der Form $\neg x_j$ aus den entsprechenden Klauseln entfernt und zudem alle Klauseln, die x_j als Literal enthalten, komplett gelöscht werden. Die entsprechenden Zuweisungen werden üblicherweise auf Decision Level 0 fixiert, um, sofern die gegebene Probleminstanz erfüllbar ist, ein vollständiges Modell zu gewährleisten, das Zuweisungen an alle in der Formel auftretenden Literale enthält. In diesem Zusammenhang werden Informationen über äquivalente Variablen separat gespeichert und nach Abschluss der Suche ebenfalls der erfüllenden Belegung hinzugefügt, da diese Art der Vereinfachung dazu führt, dass einige Variablen der originalen CNF-Formel nicht mehr in der modifizierten Formel vorkommen.

Ansätze, die auf einer probeweise durchgeführten BCP-Operation beruhen, in der Literatur auch unter dem Namen *Unit Propagation Lookahead* (UPLA) bekannt, haben einen entscheidenden Nachteil. Die Ausgangsformel muss binäre Klauseln enthalten, da sich ansonsten aufgrund einer getätigten Variablenzuweisung keine Implikationen ergeben und somit keine die Formel vereinfachenden Rückschlüsse möglich sind. Das bedeutet, dass derartige Verfahren nicht in allen Situationen zu einer Minimierung der gestellten Probleminstanz beitragen können. Als Vorteil kann allerdings angemerkt werden, das es sich bei Preprocessing-Routinen auf Basis von Unit Propagation Lookahead zumeist um sehr schnelle Methoden der Vorverarbeitung einer Probleminstanz handelt. Dies hat zwei Gründe: zum Einen muss die probeweise BCP-Operation nur auf diejenigen Variablen angewendet werden, die in binären Klauseln der initialen Formel enthalten sind oder in binären Klauseln auftreten, die sich während der Preprocessing-Phase ergeben haben. Bei allen anderen Variablen kann unabhängig von der gewählten Belegung keine Implikation gefolgert werden. Zum Zweiten wird die für das Preprocessing per Unit Propagation Lookahead benötigte Laufzeit maßgeblich durch die BCP-Routine beeinflusst. Diese macht, wie beispielsweise in [86] angeführt, etwa 80–90% der gesamten Laufzeit eines SAT-Algorithmus aus. Eine effiziente Implementierung ist daher unabdingbar für einen leistungsstarken SAT-Algorithmus, was dazu geführt hat, dass BCP-Routinen mit zu den am besten untersuchten und optimierten Funktionen gehören, wovon wiederum UPLA-Ansätze direkt profitieren.

Entsprechende Tests im Rahmen von MiraXT haben bestätigt, dass das Preprocessing per UPLA selbst bei sehr großen Problemstellungen innerhalb weniger Sekunden zu bewerkstelligen ist. Allerdings hat sich, bedingt durch die zumeist nur relativ geringe Anzahl an detektierten Vereinfachungen, im Mittel kein signifikanter Laufzeitvorteil für den nachfolgenden Suchprozess eingestellt, so dass auf eine Integration in PIChaff, MiraXT und PaMiraXT verzichtet wurde.

Die zweite Gruppe von Preprocessing-Verfahren basiert auf der Anwendung der Resolution. Dies bietet den Vorteil, dass der Erfolg der Vorverarbeitung nicht davon abhängt, ob die zu lösende CNF-Formel mit binären Klauseln eine bestimmte „Klasse“ von Klauseln enthält. Insbesondere hat sich die im Rahmen von SatELite im Jahr 2005 vorgestellte Methodik als vielversprechende Ausgangsbasis herauskristallisiert [31]. Im Kern werden bei diesem Ansatz alternierend solange die folgenden vier Techniken angewendet, bis sich keine weitere Veränderung der Klauselmenge mehr einstellt:

- *Self-Subsuming Resolution*: die Regel basiert auf der Beobachtung, dass im Verlauf des Suchprozesses oftmals die Situation eintritt, dass eine Klausel eine andere Klausel „fast“ subsumiert. Sei als Beispiel folgender Ausschnitt einer CNF-Formel F gegeben: $F = (x_1 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge \ldots$. Auf die beiden dargestellten Klauseln kann bezüglich x_3 die Resolutionsregel angewendet werden:

  $$(x_1 \vee \neg x_3) \otimes_{x_3} (x_1 \vee x_2 \vee x_3) = (x_1 \vee x_2)$$

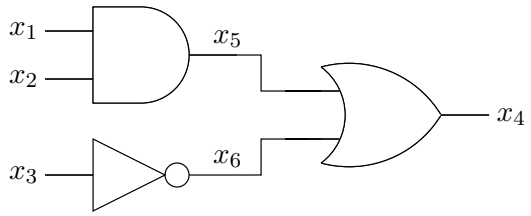
  In dieser speziellen Situation subsumiert die Resolvente $(x_1 \vee x_2)$ gemäß Definition 2.11 die in F enthaltene Klausel $(x_1 \vee x_2 \vee x_3)$, was bedeutet, dass die ursprüngliche Klausel durch die gebildete Resolvente ersetzt werden kann.

- *Subsumption*: Für jede neu zur Klauselmenge hinzugenommene Klausel wird geprüft, ob sie nicht bereits durch vorhandene Klauseln subsumiert wird. Ist dies der Fall, wird die Klausel verworfen.

- *Elimination by Clause Distribution*: die Regel entspricht exakt der in Abschnitt 2.3 diskutierten *Variablen-Elimination*. Zur Verdeutlichung sei auf Beispiel 2.6 verwiesen. Die dort betrachtete CNF-Formel $F = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_2)$ besteht aus insgesamt 3 Variablen, 12 Literalen und 6 Klauseln bei einer Größe von $|F| = 17$. Die zu F erfüllbarkeitsäquivalente Formel $F' = (x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2) \wedge (x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_2)$, die sich als Folge der Variablen-Elimination von x_1 ergibt, besteht dagegen lediglich aus 2 Variablen, 8 Literalen und 4 Klauseln bei einer Größe von $|F'| = 11$. Ersetzt man im Rahmen der Vorverarbeitung der Probleminstanz die Formel F durch F', ergibt sich eine Verringerung der durch einen SAT-Algorithmus zu analysierenden CNF-Formel. In SatELite wird eine derartige Ersetzung immer dann vorgenommen, wenn sich in der Tat eine Reduktion der Größe der Formel ergibt, obgleich auch Ansätze existieren, bei denen eine Zunahme der Größe um einen gewissen, geeignet klein gewählten Faktor erlaubt wird [108].

Für jede auf diesem Weg neu zur Klauselmenge hinzugefügten Klausel wird neben der zuvor aufgeführten Subsumption-Regel per *Backward Subsumption* ebenso getestet, ob die neue Klausel bereits existierende Klauseln subsumiert. Diese werden dann aus der CNF-Formel entfernt. Weiterhin werden die sich aus hergeleiteten Unit Clauses ergebenden Implikationen entsprechend gesetzt und per BCP-Operation die Folgeimplikationen bestimmt. Abschließend werden alle während dieses Schrittes durch eine entsprechende Zuweisung bereits erfüllten Klauseln sowie alle falsch belegten Literale aus der CNF-Formel entfernt. Klauseln, die eine Tautologie darstellen, werden ebenfalls gelöscht beziehungsweise nicht zur Klauselmenge hinzugenommen. Ergibt sich in Folge der Anwendung der *Elimination by Clause Distribution* Regel ein Widerspruch, ist die CNF-Formel unerfüllbar und das Preprocessing endet mit dem Rückgabewert *CONFLICT*, woraufhin Algorithmus 4.1 in Zeile 4 mit der Ausgabe *UNSATISFIABLE* stoppt.

– *Variable Elimination by Substitution*: Diese Regel kann als eine Erweiterung beziehungsweise Optimierung der in Abschnitt 3.1 vorgestellten Tseitin-Transformation angesehen werden. Konzeptuell besteht die Idee darin, nach Möglichkeit auf Variablen, die zu „inneren“ Signalleitungen eines Schaltkreis korrespondieren, zu verzichten. Zur Illustration sei erneut der in Abbildung 3.1 dargestellte Schaltkreis SK betrachtet, der $F_{SK} = (x_1 \wedge x_2) \vee \neg x_3$ berechnet. In diesem Beispiel korrespondieren die Variablen x_5 und x_6 zu den inneren Signalleitungen des Schaltkreises.



In CNF-Darstellung lässt sich F_{SK} wie folgt darstellen:

$$\begin{aligned} F_{SK}^{CNF} = {} & (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge \\ & (x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_5 \vee x_6) \end{aligned}$$

F_{SK}^{CNF} besteht insgesamt aus 6 Variablen, 18 Literalen und 8 Klauseln bei einer Größe von $|F_{SK}^{CNF}| = 26$. Wird diese Formel an SatELite übergeben, so wird durch die Preprocessing-Einheit erkannt, dass die drei ersten Klauseln ein UND-Gatter repräsentieren ($x_5 \equiv x_1 \wedge x_2$), so dass alle Vorkommen von x_5 durch die Teilformel $x_1 \wedge x_2$ ersetzt werden können. Die drei ersten Klauseln müssen daraufhin nicht mehr berücksichtigt werden. Als Zwischenschritt folgt:

$$F' = (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge$$

$$(x_4 \vee \neg(x_1 \wedge x_2)) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee (x_1 \wedge x_2) \vee x_6)$$

F' liegt nicht mehr in konjunktiver Normalform vor, kann aber mit den in Abschnitt 2.1 angegebenen Umformungsregeln wieder in eine Darstellung in konjunktiver Normalform überführt werden:

$$F'' = (x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_3) \wedge (x_4 \vee \neg x_1 \vee \neg x_2) \wedge (x_4 \vee \neg x_6) \wedge (\neg x_4 \vee x_1 \vee x_6) \wedge (\neg x_4 \vee x_2 \vee x_6)$$

F'' ist bedingt durch die Vorgehensweise offensichtlich erfüllbarkeitsäquivalent zu F_{SK}^{CNF}, besteht aber lediglich aus 5 Variablen, 15 Literalen und 6 Klauseln bei einer Größe von $|F''| = 21$. Wird die Ursprungsformel F_{SK}^{CNF} durch F'' ersetzt, ergibt sich in diesem Fall eine Verringerung der Problemgröße um 5.

Die hier für ein UND-Gatter gezeigte *Variable Elimination by Substitution* wird in SatELite auch für die Identifikation und Bearbeitung von Klauseln, die ein ODER-Gatter repräsentieren, eingesetzt. In diesem Szenario wird versucht, eine Teilformel der Form $(x_3 \vee \neg x_1) \wedge (x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_1 \vee x_2)$ zu identifizieren, alle Vorkommen von x_3 in der Ausgangsformel durch $x_1 \vee x_2$ zu ersetzen und abschließend die so entstandene Formel wieder in konjunktive Normalform zu überführen.

Analog zur vorherigen Regel wird die ursprüngliche Klauselmenge immer nur dann ersetzt, wenn sich durch die Anwendung der *Variable Elimination by Substitution* Regel eine Verkleinerung der Problemgröße ergibt, was nicht zwangsläufig der Fall sein muss.

Ebenso wird für jede auf diesem Weg neu zur Klauselmenge hinzugefügte Klausel wiederum per *Backward Subsumption* getestet, ob die neue Klausel bereits existierende Klauseln subsumiert, die dann aus der CNF-Formel entfernt werden. Weiterhin werden Unit Clauses bearbeitet und alle durch entsprechende Zuweisungen erfüllte Klauseln beziehungsweise falsch belegte Literale entfernt.

Ist es einem SAT-Algorithmus möglich, eine mit SatELite vereinfachte Probleminstanz zu lösen, das heißt eine erfüllende Belegung zu ermitteln, so stellt diese in der Regel nur ein partielles Modell der CNF-Formel dar, da durch die Anwendung von *Elimination by Clause Distribution* und *Variable Elimination by Substitution* gegebenenfalls Variablen aus der Formel entfernt wurden. Wie in Kapitel 2 angedeutet, wahrt die Variablen-Elimination zwar die Erfüllbarkeitsäquivalenz (Satz 2.3), ist aber nicht äquivalenzerhaltend, so dass eine Belegung für die durch SatELite vereinfachte Formel nicht zwangsläufig auch ein Modell für die Ausgangsformel darstellt. Man betrachte exemplarisch die im Folgenden dargestellten Formeln F, F' und F'', bei denen F' durch Anwendung der *Elimination by Clause Distribution* Regel bezüglich x_1 aus F hervorgegangen ist, während beim Übergang von F' zu F'' die Variable x_2 eliminiert wurde (ebenfalls per *Elimination by Clause Distribution*). Das Vorgehen ist dabei identisch zu der in Beispiel 2.6 diskutierten Verfahrensweise.

$$
\begin{aligned}
F &= (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_2) \\
F' &= (x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2) \wedge (\neg x_3 \vee x_2) \\
F'' &= (\neg x_3)
\end{aligned}
$$

Die Belegung $x_3 = 0$ ist zwar eine erfüllende Belegung für F'', erfüllt aber nicht F', da die Klausel $(x_2 \vee x_3)$ nicht erfüllt ist. Allerdings fällt auf, dass anhand des partiellen Modells für F'' und der Klausel $(x_2 \vee x_3)$ die Implikation $x_2 = 1$ gefolgert werden kann, wodurch die partielle Belegung $x_3 = 0$ zu einer erfüllenden Belegung für F' ($x_2 = 1$, $x_3 = 0$) erweitert werden kann. Diese Belegung erfüllt aber noch nicht die Ursprungsformel F, da die beiden Klauseln $(\neg x_1 \vee x_3)$ und $(\neg x_1 \vee \neg x_2)$ durch diese Belegung nicht erfüllt sind. Auch an dieser Stelle kann das partielle Modell für F' zu einer erfüllenden Belegung für F erweitert werden, da beide Klauseln die Implikation $x_1 = 0$ auslösen.

Das Beispiel deutet das Vorgehen bei der Modellerweiterung eines partiellen Modells an: die während der Vorverarbeitung aufgrund der *Elimination by Clause Distribution* beziehungsweise *Variable Elimination by Substitution* Regel gelöschten Klauseln werden separat gespeichert und im Anschluss an den eigentlichen Suchprozess herangezogen, um eine partielle erfüllende Belegung zu einem vollständigen Modell der Ausgangsformel zu erweitern. Die Erweiterung des Modells erfolgt schrittweise, wie dies im vorherigen Beispiel zunächst für den Übergang von F'' zu F' und dann für den Übergang von F' zu F demonstriert wurde. Wie man leicht sieht, ist dabei stets gewährleistet, dass ein partielles Modell der vorverarbeiteten CNF-Formel zu einem vollständigen Modell der Ursprungsformel erweitert werden kann [108]. In diesem Kontext sind Änderungen der Klauselmenge, die sich aus Unit Clauses ergaben, unproblematisch. Entweder werden die entsprechenden Variablenzuweisungen als feste Vorgaben für den sich anschließenden Suchprozess auf Decision Level 0 verankert oder separat gespeichert und im jeweiligen Schritt der partiellen erfüllenden Belegung hinzugefügt.

Auf der einen Seite deuten die gemachten Ausführungen an, dass es sich, im Gegensatz zu Unit Propagation Lookahead, bei der in SatELite implementierten Preprocessing-Einheit um eine sehr viel komplexere und damit auch zeitaufwendigere Methodik handelt. Andererseits belegen die in [31] durchgeführten Experimente, dass die aufgrund von *Self-Subsuming Resolution*, *Subsumption* und *Elimination by Clause Distribution* vorgenommenen Vereinfachungen der getesteten CNF-Formeln im Mittel dazu führen, dass die sich dadurch ergebende Laufzeiteinsparung des nachfolgenden Suchprozesses weitaus größer ausfällt als der für das Preprocessing benötigte Aufwand. Lediglich die *Variable Elimination by Substitution* Regel bildet hier eine Ausnahme: der Aufwand für die Durchführung dieser Regel wird in der Mehrheit der Fälle nicht durch die sich potenziell einstellende Laufzeiteinsparung bei der Suche nach einer erfüllenden Belegung ausgeglichen. Abgesehen davon sind die ansonsten positiven Ergebnisse neben einem vielversprechenden Konzept auch auf eine effiziente Realisierung zurückzuführen. So wird beispielsweise für jede Klausel eine so genannte *Signatur* mitgeführt, die so ausgelegt ist, dass der Test auf Subsumie-

rung einer Klausel durch eine andere schnell durchgeführt werden kann, was sowohl bei *Self-Subsuming Resolution*, *Subsumption* als auch *Backward Subsumption* entscheidend für eine effiziente Durchführung ist. Weiterhin werden analog zu den UPLA-Verfahren für die Variablen-Elimination (*Elimination by Clause Distribution* und *Variable Elimination by Substitution*) nicht alle Variablen der CNF-Formel in Betracht gezogen, sondern ebenfalls nur eine heuristisch bestimmte Teilmenge und dadurch der Berechnungsaufwand weiter minimiert.

Die in MiraXT und PaMiraXT umgesetzte Preprocessing-Einheit folgt dem hier diskutierten Konzept von SatELite, wobei, wie in MiniSat2 (dem Nachfolger von SatELite), auf eine Integration der *Variable Elimination by Substitution* Regel verzichtet wurde. Lediglich PIChaff verfügt über keine eigene Routine zur Vorverarbeitung von CNF-Formeln, stattdessen werden die Probleminstanzen zunächst mit SatELite modifiziert, danach von der das Multiprozessorsystem steuernden PC-Anwendung entgegengenommen und an die Recheneinheiten des Multiprozessorsystems weitergeleitet.

4.3 Entscheidungsheuristik

Die Entscheidungsheuristik (*Decision Heuristic*) hat einen erheblichen Einfluss auf die Performance eines SAT-Algorithmus, da mit jeder gewählten Entscheidungsvariablen (*Decision Variable*) der Teil des durch die CNF-Formel aufgespannten Suchraums ausgewählt wird, der als nächstes untersucht wird. Dieser Abschnitt konzentriert sich auf die im Rahmen von zChaff eingeführte *Variable State Independent Decaying Sum* Heuristik (VSIDS), die sich in den letzten Jahren aufgrund ihres geringen Berechnungsaufwands als Standard durchgesetzt hat und in verschiedenen Ausprägungen in nahezu jedem aktuellen SAT-Algorithmus zu finden ist. In der originalen, in [86] vorgestellten Form lässt sich die VSIDS-Strategie wie folgt charakterisieren:

- Für jede Variable x_i der gegebenen CNF-Formel werden mit P_{x_i} und N_{x_i} zwei Zähler mitgeführt. Diese geben an, wie oft die Variable x_i als positives Literal (P_{x_i}) beziehungsweise als negatives Literal (N_{x_i}) in der Klauselmenge vertreten ist. Zu Beginn werden alle Zähler mit dem Wert 0 initialisiert.
- Wird eine Klausel C zur Klauselmenge hinzugenommen, so wird für jedes in C enthaltene Literal L der jeweilige Zähler inkrementiert:

  $$\begin{aligned} P_{x_i} &= P_{x_i} + 1, \text{ falls } L = x_i \\ N_{x_i} &= N_{x_i} + 1, \text{ falls } L = \neg x_i \end{aligned}$$

  Diese Regel wird dabei sowohl auf alle während der Suche nach einer erfüllenden Belegung ermittelten Konflikt-Klauseln angewendet als auch auf sämtliche Klauseln der gegebenenfalls mit einer Preprocessing-Einheit vorverarbeiteten CNF-Formel, mit denen eine anfangs leere Klauselmenge initialisiert wird.

– Die Decision Variable wird dadurch bestimmt, dass immer diejenige freie Variable x_i ausgewählt wird, für die unter allen noch nicht belegten Variablen beziehungsweise den dazu korrespondierenden Zählern entweder P_{x_i} oder N_{x_i} maximal ist. Besitzen mehrere freie Variablen den gleichen maximalen Wert, wird eine dieser Variablen zufällig bestimmt.

  Die Belegung für x_i ergibt sich dann wie folgt: bei $P_{x_i} > N_{x_i}$ wird die Belegung $x_i = 1$ gewählt, bei $P_{x_i} < N_{x_i}$ die Belegung $x_i = 0$, bei Gleichstand entscheidet der Zufall.

– In regelmäßigen Abständen werden alle Zählerstände durch einen konstanten Faktor geteilt.

Bei der VSIDS-Heuristik werden vorrangig Variablen gewählt, die in der Klauselmenge oft vertreten sind, allerdings mit Schwerpunkt auf Variablen, die insbesondere zuletzt häufig in Konflikt-Klauseln enthalten waren. Bedingt durch das im letzten Punkt angegebene „Normalisieren“ der Zählerstände verlieren die aktuellen Werte der Zähler an Bedeutung, während jede nun folgende Erhöhung eines Zählers, ausgelöst durch eine neu zur Klauselmenge hinzugenommene Konflikt-Klausel, relativ an Gewicht gewinnt.

Die Zähler P_{x_i} und N_{x_i} werden im weiteren Verlauf der Arbeit auch als *Aktivität* der Variablen x_i bezeichnet. Je höher der Zählerstand, desto häufiger ist x_i als Literal in Klauseln der Klauselmenge und insbesondere in zuletzt hergeleiteten Konflikt-Klauseln vertreten. Die Variable x_i scheint daher zum aktuellen Zeitpunkt der Suche einen erhöhten Einfluss zu haben, ist in diesem Sinne folglich sehr *aktiv* und wird bevorzugt als Entscheidungsvariable gewählt.

Im Vergleich zu Strategien wie *Dynamic Largest Combined Sum*, *Dynamic Largest Individual Sum*, *Jeroslow-Wang* oder *Maximum Occurrences on Clauses of Minimal Size* besitzt die VSIDS-Heuristik durch einen erheblich geringeren Berechnungsaufwand einen entscheidenden Vorteil. Vereinfacht ausgedrückt wird bei den zuvor genannten, „klassischen“ Heuristiken die Decision Variable danach ausgewählt, wie oft eine freie Variable in aktuell unerfüllten Klauseln auftritt, was als „Status“ der Variablen angesehen werden kann. Zumeist wird die Vorkommenshäufigkeit zusätzlich noch mit der Länge der entsprechenden Klauseln gewichtet (für einen ausführlichen Überblick sei auf [77] verwiesen). Je häufiger eine Variable in unerfüllten Klauseln auftritt, umso mehr Klauseln könnten bei einer dahingehenden Wahl der Decision Variable erfüllt werden. Das Vorgehen hat allerdings zur Folge, dass in den Berechnungsaufwand derartiger Heuristiken die aktuelle Anzahl an (unerfüllten) Klauseln als entscheidende Größe einfließt.

Dem gegenüber spielt die zuvor gegebene, intuitive Definition des Status der Variablen bei der VSIDS-Strategie keine Rolle. Die Wahl der Decision Variable hängt einzig von den absoluten Vorkommenshäufigkeiten der freien Variablen ab, mit einem Schwerpunkt

auf zuletzt hergeleitete Konflikt-Klauseln, was durch ein periodisches Normalisieren der Zählerstände erreicht wird. Beides zusammen erklärt auch den Namen *Variable State Independent Decaying Sum*. Der Hauptaufwand der VSIDS-Strategie ist darin zu sehen, dass jeweils diejenige freie Variable bestimmt werden muss, deren Aktivität als positives (P_{x_i}) oder negatives Literal (N_{x_i}) maximal unter allen noch nicht belegten Variablen ist. Die Umsetzung kann so erfolgen, dass mit jedem Aufruf der Decision Heuristic alle freien Variablen der Reihe nach bezüglich ihrer Aktivität bewertet werden. Alternativ kann eine nach Aktivitäten sortierte Liste der aktuell freien Variablen mitgeführt werden, so dass, vom Sortieraufwand abgesehen, ein Zugriff auf die Variable mit der höchsten Aktivität stets effizient möglich ist. In beiden Fällen ist die Anzahl der Variablen, die im Allgemeinen um ein Vielfaches kleiner ist als die Anzahl der Klauseln, die entscheidende Größe für den Berechnungsaufwand. Ebenso wichtig für den geringen Aufwand, der im Rahmen der VSIDS-Strategie betrieben werden muss, ist die Tatsache, dass im Gegensatz zu den zuvor genannten Heuristiken insbesondere während einer Backtrack-Operation keinerlei Statusvariablen aktualisiert werden müssen (eine erfüllte Klausel kann durch die Rücknahme einer Zuweisung wieder in den Zustand *unerfüllt* übergehen). Die Überlegenheit der VSIDS-Strategie gegenüber den zuvor genannten Ansätzen wird durch die in [86] durchgeführten Experimente belegt.

Beim Einsatz einer nach Aktivitäten sortierten Liste der freien Variablen lässt sich eine weitere Reduktion des Berechnungsaufwands dadurch erzielen, dass die Liste nur semisortiert gehalten wird. Dies bedeutet, dass die Liste nicht zwangsläufig nach jedem Aufruf der Konflikt-Analyse (nur durch die Hinzunahme einer Konflikt-Klausel zur Klauselmenge ändern sich einzelne Zählerwerte) neu sortiert wird. Verschiebungen innerhalb der Variablenreihenfolge wirken sich dann zwar erst verspätet aus, allerdings verringert sich die für die Entscheidungsheuristik benötigte Laufzeit. Diese Variante findet sich in PIChaff, MiraXT, PaMiraXT und beispielsweise auch in der zChaff-Version des Jahres 2004 [42].

Die hier vorgestellte, originale Version der *Variable State Independent Decaying Sum* Heuristik bietet darüber hinaus zahlreiche Möglichkeiten, diverse Einstellungen zu variieren und das Vorgehen damit für eine bestimmte Problemklasse zu optimieren. Parameter sind hierbei der konstante Faktor, durch den die Zählerstände in regelmäßigen Abständen geteilt werden, sowie das Intervall zwischen zwei derartigen Normalisierungen. Zum Beispiel wird in PIChaff diese Operation alle 256 Konflikte vorgenommen und die Aktivitäten per *Rechts-Shift* jeweils halbiert. Wird der Faktor erhöht oder das Intervall verkleinert, lässt sich der Fokus auf Variablen, die in jüngster Zeit gehäuft in Konflikt-Klauseln aufgetreten sind, verstärken. Umgekehrt bewirkt eine Verkleinerung des Faktors oder eine Erhöhung des Intervalls, dass der Schwerpunkt auf freien Variablen liegt, die in der gesamten Klauselmenge oft vertreten sind, ohne eine Gruppe von Variablen gesondert hervorzuheben. Auch in BerkMin [44] und Siege [67] wird versucht, wenn auch auf einer anderen Methodik aufbauend, die dortigen VSIDS-Varianten „lokaler“ auf den aktuell untersuchten Bereich des durch die CNF-Formel aufgespannten Suchraums auszurichten. Es werden vorrangig

die freien Variablen ausgewählt, die besonders oft als positives oder negatives Literal in zuletzt hergeleiteten Konflikt-Klauseln vertreten sind.

Weiterhin besteht die Möglichkeit, die Aktivität einer Variablen nicht nur zu erhöhen, wenn diese in einer (Konflikt-)Klausel enthalten ist, sondern eine Inkrementierung auch in anderen Situationen vorzunehmen. So können beispielsweise auch die Zählerstände von Variablen erhöht werden, die zwar nicht Literal einer Konflikt-Klausel sind, aber derzeit dennoch wichtig für den Suchprozess erscheinen. In diesem Zusammenhang kann angeführt werden, dass die für einen Konflikt hauptverantwortlichen Variablenzuweisungen (die in der entsprechenden Konflikt-Klausel kodiert sind) die widersprüchliche Belegung üblicherweise nicht „direkt“ auslösen, sondern eine Folge von Implikationen erzwingen, die schlussendlich zum Konflikt führen. Diese Implikationen sind somit zwar nicht ursächlich für den Konflikt verantwortlich, haben aber doch einen gewissen Einfluss und sind in diesem Sinne ebenfalls *aktiv*. Daher werden sowohl bei PIChaff als auch bei MiraXT und PaMiraXT auch die Zählerstände derjenigen Variablen erhöht, bezüglich derer während der Konflikt-Analyse die Resolutionsregel angewendet wird. Vorgreifend auf Abschnitt 4.5 sind dies alle innerhalb der in Algorithmus 4.2 angegebenen while-Schleife betrachteten Variablen.

4.4 Boolean Constraint Propagation

Wie aus Algorithmus 4.1 ersichtlich, schließt sich an die Wahl der nächsten Decision Variable stets die so genannte *Boolean Constraint Propagation* Phase an (Funktion BCP, Zeile 9). Diese hat zum Ziel, ausgehend von der soeben getätigten Variablenzuweisung alle sich daraus ergebenden Implikationen und Konflikte festzustellen. Beispielsweise impliziert die Zuweisung $x_1 = 0$ bei der Formel $F = (x_1 \vee x_2)$ die Zuweisung $x_2 = 1$, während bei der CNF-Formel $G = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ dadurch ein Konflikt auftritt, da x_2 zeitgleich die Wahrheitswerte 0 und 1 annehmen müsste, um beide Klauseln von G zu erfüllen. Wie in [86] angeführt, macht der Anteil der *Boolean Constraint Propagation* etwa 80–90% der Gesamtlaufzeit eines SAT-Algorithmus aus, so dass eine effiziente Implementierung der BCP-Routine für einen leistungsstarken SAT-Algorithmus unabdingbar ist.

4.4.1 Überblick

Bevor im Folgenden die effiziente Umsetzung der *Boolean Constraint Propagation* beschrieben wird, soll zunächst das prinzipielle Vorgehen anhand der CNF-Formel F aus Beispiel 4.1 näher erläutert werden, wobei F der Übersichtlichkeit halber hier erneut angegeben sei (die mit jeder Klausel assoziierte Nummer, die Klausel-ID, dient im Folgenden der eindeutigen Referenzierung der Klauseln):

$$F = \underbrace{(x_{23})}_{1} \wedge \underbrace{(x_7 \vee \neg x_{23})}_{2} \wedge \underbrace{(x_6 \vee \neg x_{17})}_{3} \wedge \underbrace{(x_6 \vee \neg x_{11} \vee \neg x_{12})}_{4} \wedge \underbrace{(x_{13} \vee x_8)}_{5} \wedge$$

$$\underbrace{(\neg x_{11} \vee x_{13} \vee x_{16})}_{6} \wedge \underbrace{(x_{12} \vee \neg x_{16} \vee \neg x_2)}_{7} \wedge \underbrace{(x_2 \vee \neg x_4 \vee \neg x_{10})}_{8} \wedge$$
$$\underbrace{(\neg x_{19} \vee x_4)}_{9} \wedge \underbrace{(x_{10} \vee \neg x_5)}_{10} \wedge \underbrace{(x_{10} \vee x_3)}_{11} \wedge \underbrace{(x_{10} \vee \neg x_8 \vee x_1)}_{12} \wedge$$
$$\underbrace{(\neg x_{19} \vee \neg x_{18} \vee \neg x_3)}_{13} \wedge \underbrace{(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)}_{14} \wedge \ldots$$

Es sei angenommen, dass sich der SAT-Algorithmus in der in Abbildung 4.6 dargestellten Situation befindet, also aktuell auf Decision Level 5 mit $x_{11} = 1$ die nächste Decision Variable ausgewählt und belegt hat.

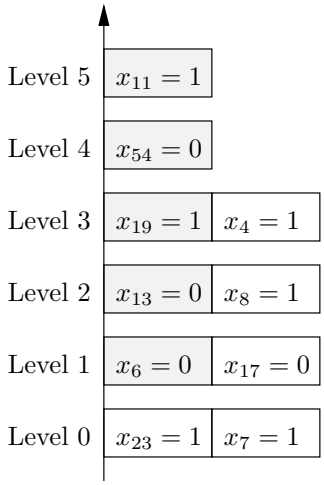


Abbildung 4.6: Ablauf der *Boolean Constraint Propagation* (1 von 10)

Die BCP-Phase läuft in mehreren aufeinanderfolgenden Schritten ab. Zunächst werden alle sich direkt aus der Zuweisung $x_{11} = 1$ ergebenden Implikationen bestimmt. Ausgehend von den dabei gefundenen Implikationen wird dann für jede dieser Zuweisungen separat geprüft, welche Konsequenzen sich hieraus ergeben, bevor wiederum deren Auswirkungen auf den Suchprozess (weitere Implikationen oder Konflikte) erneut der Reihe nach analysiert werden. Dieses Vorgehen wird solange fortgeführt, bis sich entweder ein Konflikt einstellt oder keine neuen Implikationen mehr bestimmt werden können. Unter der Annahme, dass durch die Wahl der Decision Variable kein Konflikt auftritt, sind mit Beendigung der BCP-Phase dann alle durch die jeweilige Decision Variable des aktuellen Decision Levels direkt und indirekt erzwungenen Implikationen identifiziert und bearbeitet worden.

Bezogen auf das hier gewählte Beispiel betrachte man die beiden mit den ID-Nummern 4 und 6 bezeichneten Klauseln $(x_6 \vee \neg x_{11} \vee \neg x_{12})$ beziehungsweise $(\neg x_{11} \vee x_{13} \vee x_{16})$. In

beiden Klauseln wurde mit $x_6 = 0$ beziehungsweise $x_{13} = 0$ bereits auf einem früheren Decision Level ein Literal falsch belegt. Ebenso ist das in beiden Klauseln vorkommende Literal $\neg x_{11}$, bedingt durch die auf Decision Level 5 gewählte Decision Variable, falsch belegt, so dass mit $x_{12} = 0$ (Klausel 4) beziehungsweise $x_{16} = 1$ (Klausel 6) zwei Implikationen ausgelöst werden.

Moderne SAT-Algorithmen verwenden zur Abspeicherung der gefundenen Implikationen die so genannte *Implication Queue*, in der zunächst alle aktuell detektierten Implikationen in der Reihenfolge ihrer Identifizierung eingetragen werden. Die einzelnen Elemente der Implication Queue werden dann in chronologischer Reihenfolge aus der Liste entnommen, die entsprechende Zuweisung vorgenommen und geprüft, welche Konsequenzen sich aus dieser Zuweisung ergeben (die dabei gefundenen Implikationen werden wiederum dem Ende der Implication Queue angehängt). Vereinfacht ausgedrückt speichert die Implication Queue all diejenigen Implikationen, die zwar bereits identifiziert werden konnten, allerdings noch nicht den erforderlichen Wahrheitswert zugewiesen bekommen haben und für die auch noch nicht geprüft wurde, welche Auswirkungen sich aus speziell dieser Zuweisung ergeben.

Abbildung 4.7 zeigt die Implication Queue für das hier betrachtete Beispiel und die sich direkt aus $x_{11} = 1$ ergebenden Implikationen $x_{12} = 0$ und $x_{16} = 1$. Aufgrund der Reihenfolge der beiden relevanten Klauseln innerhalb der CNF-Formel F wird angenommen, dass die Implikation $x_{12} = 0$ vor der Implikation $x_{16} = 1$ erkannt und daher zuerst in die Implication Queue eingetragen wurde. Die bei den gestrichelten Linien angegebenen Ziffern deuten jeweils die implikationsauslösende Klausel an.

Weitere Implikationen lassen sich aus $x_{11} = 1$ nicht ableiten, so dass im nächsten Schritt mit $x_{12} = 0$ das erste Element der Implication Queue entnommen, die entsprechende Zuweisung getätigt und geprüft wird, ob sich weitere Implikationen aus dieser Zuweisung ergeben. In Abbildung 4.8 ist schematisch der dazugehörige Decision Stack sowie die Implication Queue dargestellt, wobei die Zuweisung $x_{12} = 0$ in diesem Beispiel keine Implikationen auslöst. Durchgestrichene Elemente der Implication Queue sind als aus der Liste entfernte Einträge zu verstehen.

Dieses Vorgehen wiederholt sich zunächst mit der Bearbeitung der Implikation $x_{16} = 1$, gefolgt von $x_2 = 0$, $x_{10} = 0$, $x_5 = 0$, $x_3 = 1$ und $x_1 = 1$ (Abbildungen 4.9 bis 4.14). Bei der Prozessierung der nächsten zu bearbeitenden Implikation, $x_{18} = 0$, die notwendig ist, um aufgrund vorangegangener Zuweisungen die Klausel $(\neg x_{19} \vee \neg x_{18} \vee \neg x_3)$ zu erfüllen, stoppt die Boolean Constraint Propagation. Wie Abbildung 4.15 verdeutlicht, liegt in der Implication Queue bereits die Implikation $x_{18} = 1$ vor, welche die Erfüllbarkeit der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$ gewährleistet. Die Variable x_{18} müsste folglich zeitgleich die beiden Wahrheitswerte 0 und 1 annehmen, um beide Klauseln erfüllen zu können, das heißt, es liegt ein Konflikt vor. Die BCP-Funktion wird daher mit dem Rückgabewert *CONFLICT* gestoppt, so dass direkt im Anschluss die Konflikt-Analyse gestartet, ein Backtrack Level

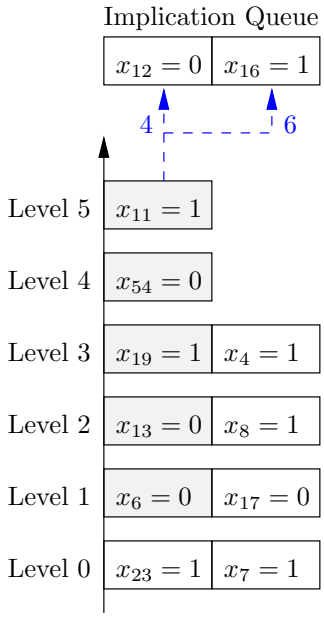


Abbildung 4.7: Ablauf der *Boolean Constraint Propagation* (2 von 10)

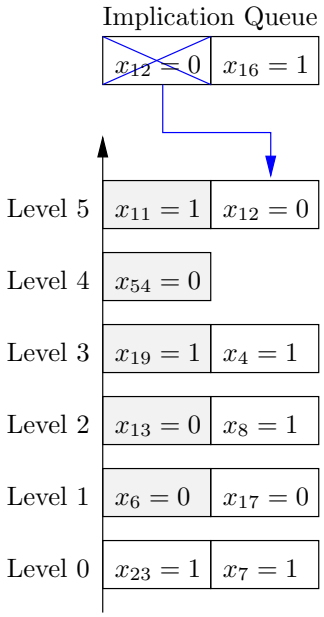


Abbildung 4.8: Ablauf der *Boolean Constraint Propagation* (3 von 10)

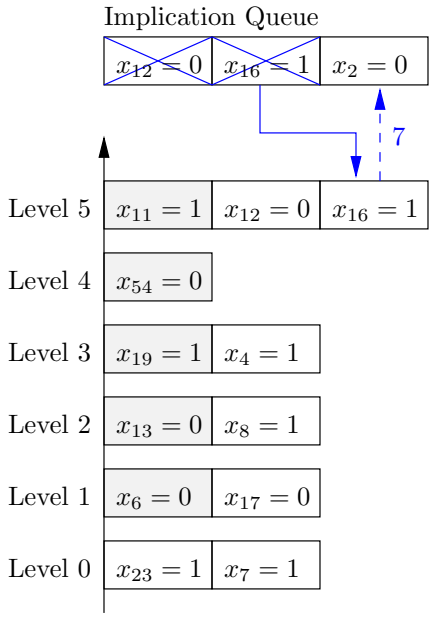


Abbildung 4.9: Ablauf der *Boolean Constraint Propagation* (4 von 10)

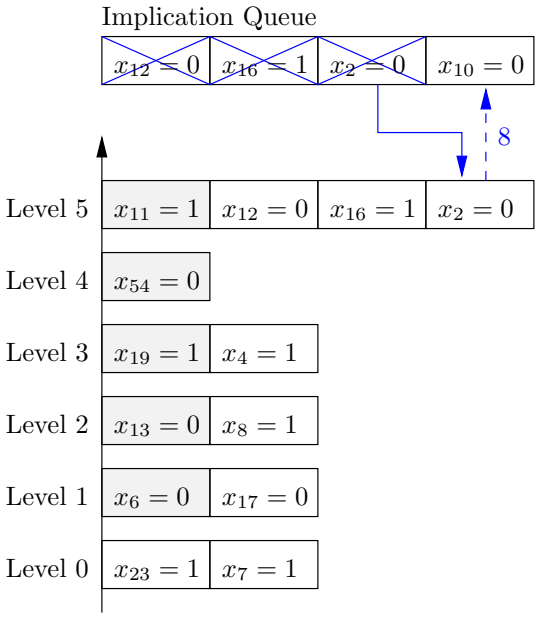


Abbildung 4.10: Ablauf der *Boolean Constraint Propagation* (5 von 10)

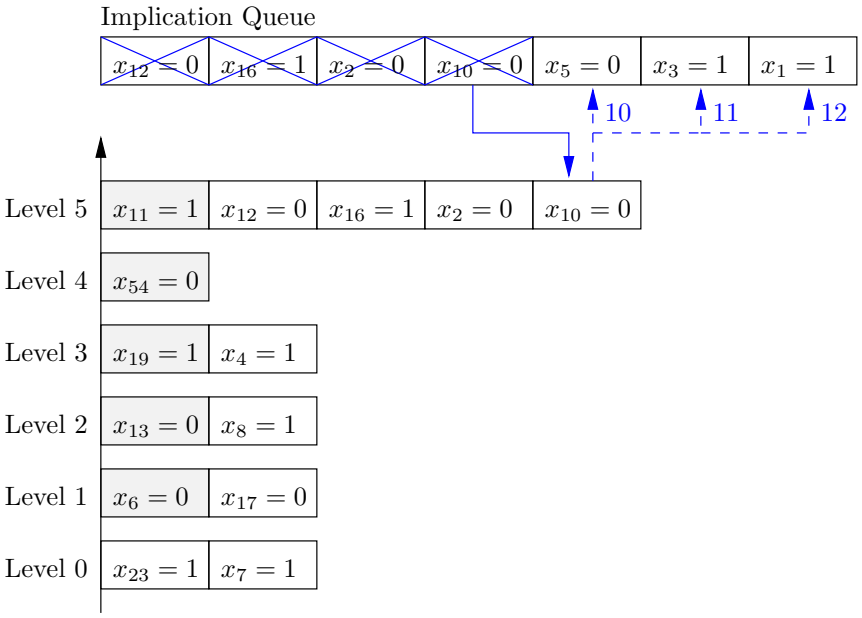


Abbildung 4.11: Ablauf der *Boolean Constraint Propagation* (6 von 10)

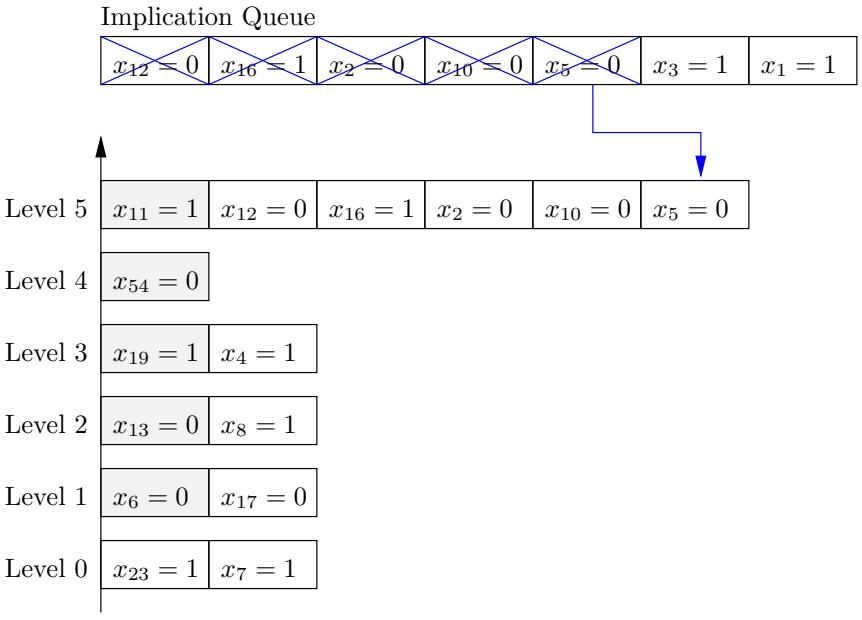


Abbildung 4.12: Ablauf der *Boolean Constraint Propagation* (7 von 10)

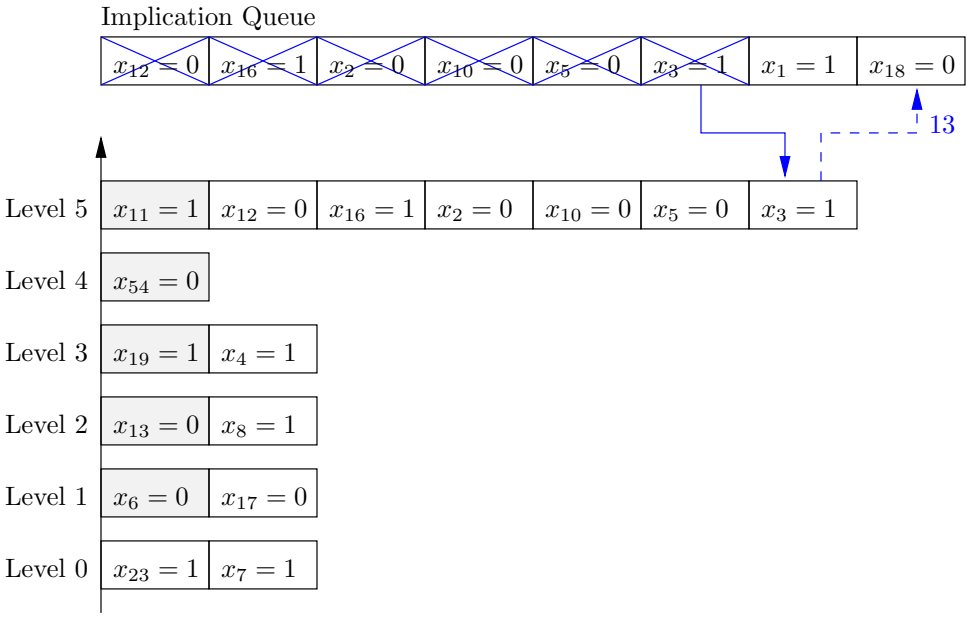


Abbildung 4.13: Ablauf der *Boolean Constraint Propagation* (8 von 10)

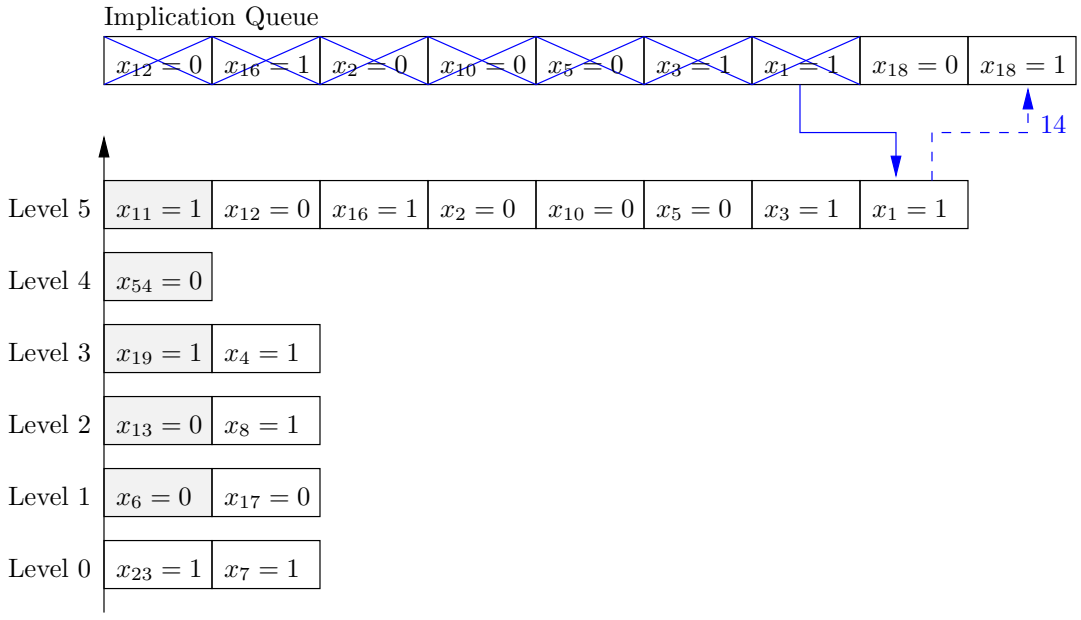


Abbildung 4.14: Ablauf der *Boolean Constraint Propagation* (9 von 10)

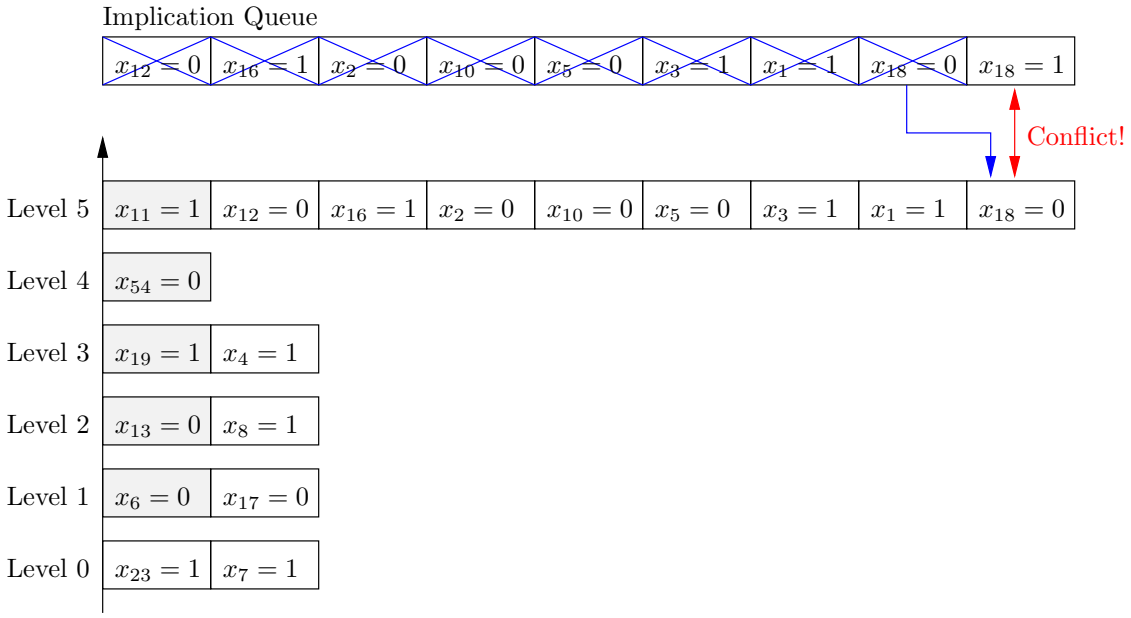


Abbildung 4.15: Ablauf der *Boolean Constraint Propagation* (10 von 10)

bestimmt und der Suchprozess in einem anderen Bereich des durch die gegebene CNF-Formel aufgespannten Suchraums fortgesetzt werden kann (Zeile 9, Algorithmus 4.1).

Ohne das Auftreten eines Konflikts stoppt die BCP-Phase, sobald sich keine Einträge mehr in der Implication Queue befinden, also alle aus der Wahl der aktuellen Decision Variable resultierenden Auswirkungen auf den Suchprozess identifiziert und bearbeitet wurden. Der SAT-Algorithmus setzt die Suche nach einer erfüllenden Belegung daraufhin mit der Wahl der nächsten Decision Variable fort.

4.4.2 Algorithmische Umsetzung

Ausgehend vom vorherigen Beispiel bleibt die Frage zu klären, wie die BCP-Phase realisiert werden kann, um Implikationen und Konflikte möglichst schnell zu identifizieren. Im Mittelpunkt dieses Abschnitts steht das im Rahmen von zChaff vorgestellte Konzept der *Watched Literals* [86], bei dem es sich um eine Verallgemeinerung des in SATO und Berk-Min integrierten *Head/Tail*-Ansatzes handelt [44, 120, 122]. Die Kernidee der Watched Literals besteht darin, nach jeder Variablenzuweisung nicht alle Klauseln zu evaluieren, sondern nur die Klauseln, bei denen die Chance besteht, Implikationen oder Konflikte bestimmen zu können. Vor diesem Hintergrund kann argumentiert werden, dass die Watched Literals zur Beobachtung des Status der verschiedenen Klauseln eingesetzt werden.

Zur Umsetzung werden pro Klausel zwei Literale gesondert markiert, für die stets die Inva-

riante gelten muss, dass entweder beide Literale der Klausel unbelegt sind oder mindestens eines der beiden Literale die Klausel erfüllt. Solange die Invariante für diese beiden als Watched Literals bezeichneten Literale gilt, erübrigt sich während der Boolean Constraint Propagation die Evaluierung der entsprechenden Klausel: entweder ist die Klausel bereits erfüllt oder von den insgesamt k Literalen der Klausel sind maximal $k-2$ Literale unerfüllt. In beiden Fällen kann die jeweilige Klausel weder eine Implikation noch einen Konflikt verursachen. Daher müssen beim Konzept der Watched Literals nach jeder Variablenzuweisung nur diejenigen Klauseln untersucht werden, bei denen eines der beiden Watched Literals die Invariante verletzt. Ist es möglich, einen Nachfolger für das entsprechende Watched Literal zu bestimmen, gilt weiterhin die zuvor angegebene Invariante, ansonsten liegt in Abhängigkeit vom Status des zweiten Watched Literals bei der jeweiligen Klausel entweder eine Implikation oder ein Konflikt vor (sofern das zweite Watched Literal nicht die Klausel erfüllt). Im Fall einer Implikation wird diese, wie zuvor erläutert, zunächst in die Implication Queue eingetragen und bearbeitet, sobald sie das erste Element der Implication Queue darstellt. Während der eigentlichen Bearbeitung der Implikation wird insbesondere die implizierte Zuweisung getätigt, so dass einerseits die entsprechende Klausel erfüllt ist und andererseits die Invariante der Watched Literals wieder gilt. Im Fall eines Konflikts wird ein Rücksprung auf einen früheren Decision Level vollzogen, der, wie sich zeigen wird, ebenfalls dazu führt, dass die angegebene Invariante wieder gilt.

Abbildung 4.16 illustriert das Vorgehen anhand der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$. Während der Startphase findet die Initialisierung der Watched Literals so statt, dass pro Klausel zwei beliebige Literale gesondert markiert werden und zudem festgelegt wird, in welche Richtung jeweils nach einem Nachfolger für ein falsch belegtes Watched Literal gesucht werden soll. Abbildung 4.16(a) beschreibt den Initialzustand für die hier gewählte Klausel, wobei die schwarzen Pfeile die beiden Watched Literals anzeigen, während die blau markierten Pfeile die Suchrichtung angeben. Grau unterlegte Literale symbolisieren in diesem Zusammenhang, dass das entsprechende Literal falsch belegt wurde. Es sei angemerkt, dass die Suchrichtung während der Initialisierungsphase für jede Klausel beliebig gewählt werden kann und nicht zwangsläufig für beide Watched Literals gleich sein muss.

Abbildung 4.16(b) zeigt die Situation, dass bedingt durch die Zuweisung $x_{17} = 0$ ein Nachfolger für das aktuelle Watched Literal, auf das der linke Zeiger verweist, gefunden werden muss. Aufgrund der festgelegten Suchrichtung wird rechts von der aktuellen Position des Zeigers ein Nachfolger gesucht, so dass das Literal $\neg x_1$, das (noch) unbelegt ist, zum neuen Watched Literal wird. Abbildung 4.16(c) skizziert das Vorgehen, wenn aufgrund der Zuweisung $x_5 = 0$ ein Nachfolger für das rechte Watched Literal bestimmt werden muss. Auch hier wird rechts von der aktuellen Position des Zeigers gesucht, da aber das Ende der Klausel bereits erreicht ist, wird die Suche am Anfang der Klausel fortgesetzt. Das erste in Frage kommende Literal der Klausel wäre $\neg x_1$ (x_{17} scheidet aus, da es zuvor falsch belegt wurde), allerdings handelt es sich dabei um das zweite Watched Literal, so dass auch dieses als Kandidat ausscheidet und die Wahl auf x_{18} fällt.

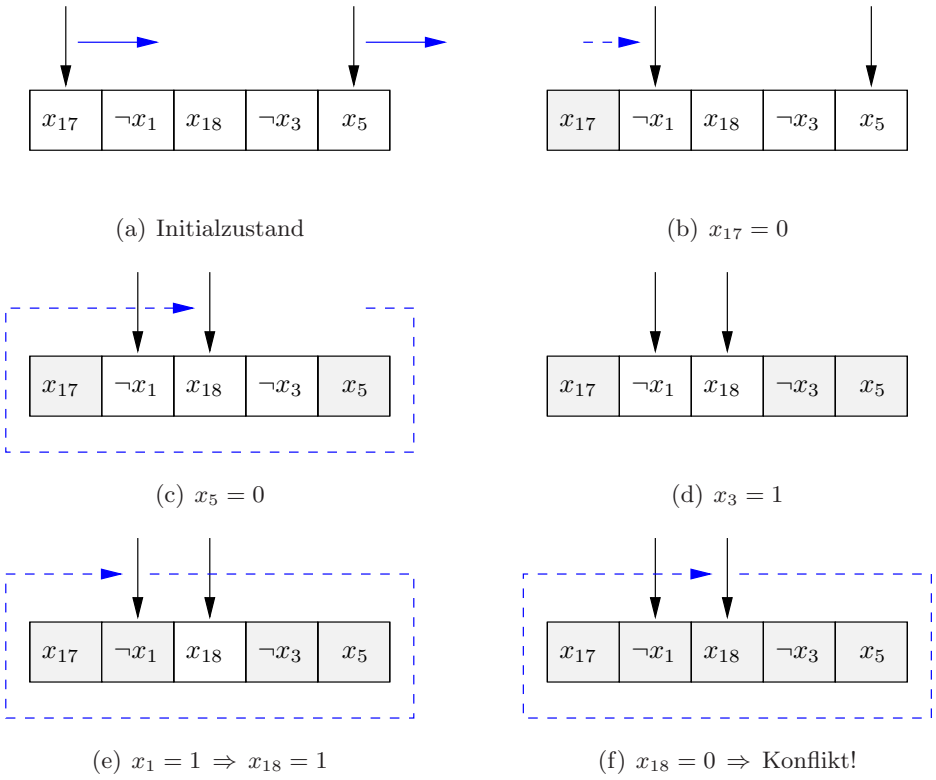


Abbildung 4.16: *Watched Literals*-Repräsentation der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$

In Fällen, in denen ein Literal, das aktuell kein Watched Literal darstellt, falsch belegt wird, besteht keine Notwendigkeit, die entsprechende Klausel zu evaluieren. Abbildung 4.16(d) bildet eine derartige Situation für die Zuweisung $x_3 = 1$ ab, wobei es sich beim Literal $\neg x_3$ um keines der beiden Watched Literals der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$ handelt.

Die beiden letzten Teilabbildungen verdeutlichen die Identifikation von Implikationen und Konflikten. In Abbildung 4.16(e) sei die Belegung $x_1 = 1$ angenommen, die es erforderlich macht, einen Nachfolger für das linke Watched Literal zu finden. Bei einer nach rechts gerichteten Suche kommt prinzipiell das direkt benachbarte und unbelegte Literal x_{18} in Frage. Es scheidet aber aus, weil es bereits ein Watched Literal repräsentiert, so dass die Klausel zunächst bis zum Klauselende und dann vom Klauselanfang bis zum „Ausgangsort“ untersucht wird. Da kein weiteres unbelegtes oder die Klausel erfüllendes Literal gefunden werden konnte, muss mit $x_{18} = 1$ eine Implikation vorliegen (wäre die Klausel durch eine Zuweisung an diese Variable erfüllt, so läge selbstverständlich keine Implikation

vor). Unabhängig davon verbleibt der Zeiger auf das linke Watched Literal in seiner Ausgangsposition. Die gefundene Implikation $x_{18} = 1$ wird daraufhin in die Implication Queue eingetragen und bearbeitet, sobald sie das erste Element der Implication Queue darstellt, wodurch dann auch die kurzfristig verletzte Invariante der Watched Literals wiederhergestellt wird.

Abbildung 4.16(f) geht von der Annahme aus, dass aufgrund der aktuellen Belegung die Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$ zwar die Implikation $x_{18} = 1$ auslöst, in der Implication Queue aber bereits die Implikation $x_{18} = 0$ vorliegt und diese zuerst bearbeitet wird. Für das hier gewählte Beispiel, das die in Abbildung 4.14 beziehungsweise Abbildung 4.15 gezeigte Situation wiedergibt, bedeutet dies, dass zum Einen das linke Watched Literal die Invariante verletzt und zum Anderen auch für das rechte Watched Literal ein Nachfolger gesucht werden muss. Eine dahingehende Suche führt zu keinem positiven Ergebnis, so dass auf einen Konflikt geschlossen werden kann und der Zeiger auf das rechte Watched Literal in seiner Ausgangsposition (auf das Literal x_{18} verweisend) verharrt. Als Konsequenz wird unmittelbar die Konflikt-Analyse gestartet und der Suchprozess nach einer entsprechenden Backtrack-Operation fortgesetzt. Bedingt dadurch, dass die aktuellen Watched Literals $\neg x_1$ und x_{18} der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$ beide auf dem konfliktauslösenden Decision Level belegt wurden und im Rahmen der Backtrack-Operation zumindest die Zuweisungen dieser Entscheidungsebene aufgehoben werden, wird durch das Backtracking auch die für die Watched Literals geforderte Invariante wiederhergestellt.

Die Durchführung einer BCP-Routine auf Basis von Watched Literals erfordert pro Variable das Mitführen von zwei Listen, die angeben, in welchen Klauseln die Variable als positives beziehungsweise negatives Literal ein Watched Literal darstellt. Nach jeder Variablenzuweisung muss lediglich die Liste der Klauseln abgearbeitet werden, in denen die Variable als Watched Literal auftritt und die Invariante verletzt. Für eine Zuweisung wie etwa $x_{11} = 1$ sind dies alle Klauseln, in denen $\neg x_{11}$ eines der beiden Watched Literals darstellt, da diese Klauseln potenziell eine Implikation oder einen Konflikt auslösen.

In den letzten Jahren haben sich Watched Literals als Standard durchgesetzt und sind im Rahmen der Boolean Constraint Propagation in nahezu jedem modernen SAT-Algorithmus integriert. Der entscheidende Vorteil liegt darin, dass während des Backtrackings keinerlei Aktualisierungen der Watched Literals notwendig sind. Dies lässt sich folgendermaßen begründen: für alle Klauseln, für welche die geforderte Invariante vor dem Backtracking galt, ist diese auch nach dem Backtracking erfüllt. Einzig die Situation, in der beide Watched Literals falsch belegt sind und die entsprechende Klausel einen Konflikt auslöst, scheint auf den ersten Blick problematisch. Da Konflikte aber sofort durch ein Backtracking auf einen niedrigeren Decision Level aufgelöst werden, gehen die beiden falsch belegten Watched Literals durch die Backtrack-Operation automatisch in einen unbelegten Zustand über, wodurch die Invariante wieder erfüllt ist (siehe auch Abschnitt 4.5). Die Überlegenheit von Watched Literals gegenüber anderen Ansätzen wurde sowohl in [74] als auch in [124] durch

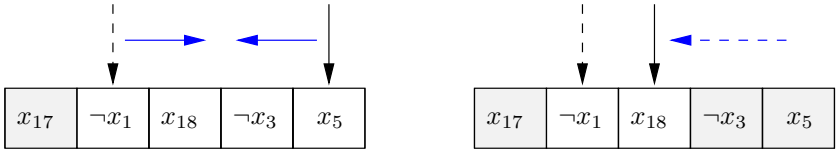
zahlreiche vergleichende Messungen bestätigt.

Die sequentiellen SAT-Prozeduren von PIChaff, MiraXT und PaMiraXT bauen aus den genannten Gründen ebenfalls auf dem Konzept der Watched Literals auf. Alle drei gehen allerdings noch einen Schritt weiter und setzen als Optimierung das so genannte *Early Conflict Detection Based BCP* (ECDB) [69] ein, das sich in ähnlicher Ausprägung auch in einer Reihe weiterer SAT-Algorithmen findet [12, 32, 67].

An dieser Stelle sei auf das in Abschnitt 4.4.1 gezeigte typische Vorgehen während der Boolean Constraint Propagation verwiesen, bei dem die gefundenen Implikationen zunächst in der Implication Queue gespeichert und zu einem späteren Zeitpunkt bearbeitet werden. Der Begriff „bearbeitet“ bedeutet in diesem Kontext, dass einerseits die durch die Implikation erzwungene Zuweisung vorgenommen wird, andererseits alle sich daraus ergebenden Konsequenzen bestimmt werden. Insbesondere wird die Zuweisung an die entsprechende Variable erst dann getätigt, wenn auch deren Auswirkungen (weitere Implikationen oder Konflikte) ermittelt werden.

Bei *Early Conflict Detection Based BCP* werden diese beiden Vorgänge getrennt, indem die implizierte Variablenzuweisung sofort vorgenommen wird, während die sich ergebenden Folgeimplikationen oder Konflikte gegebenenfalls erst zu einem späteren Zeitpunkt während der BCP-Phase bestimmt werden (sobald die entsprechende Implikation das erste Element der Implication Queue ist). Dies bietet zwei Vorteile: erstens werden Konflikte früher erkannt. Man betrachte in diesem Zusammenhang die Abbildungen 4.14 und 4.15. Ohne ECDB tritt die Situation ein, dass sowohl die Implikation $x_{18} = 0$ als auch die Implikation $x_{18} = 1$ erkannt und in der Implication Queue abgelegt werden, der daraus resultierende Konflikt aber nicht sofort festgestellt wird, da noch keine der beiden Implikationen bearbeitet wurde (Abbildung 4.14). Der Konflikt wird erst im darauffolgenden Schritt mit der Bearbeitung der Implikation $x_{18} = 0$ erkannt (Abbildung 4.15). Mit ECDB wäre der Konflikt sofort beim Eintragen der Implikation $x_{18} = 1$ in die Implication Queue erkannt worden, da die vorhergehende Implikation $x_{18} = 0$ unmittelbar zur entsprechenden Zuweisung $x_{18} = 0$ geführt hätte. Insgesamt lässt sich dadurch unter Umständen eine Vielzahl unnötiger Klauselevaluierungen vermeiden.

Zweitens kann die Wahl der Watched Literals effizienter gestaltet werden, wie dies Abbildung 4.17 für die Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$ andeutet. In der linken Teilabbildung ist der Initialzustand mit der Verteilung der beiden Watched Literals und deren Suchrichtung bei einer eventuellen Bestimmung eines Nachfolgers dargestellt. Nun sei angenommen, dass sich aus der Zuweisung $x_{10} = 0$ die beiden Implikationen $x_5 = 0$ und $x_3 = 1$ folgern lassen und diese auch in dieser Reihenfolge in der Implication Queue eingetragen sind, so dass ein Nachfolger für das rechte Watched Literal gefunden werden muss. Ohne ECDB wird als neues Watched Literal $\neg x_3$ gewählt. Zwar wurde mit $x_3 = 1$ eine Implikation gefunden, bei der dieses Literal als potenzielles Watched Literal ausscheidet, diese Impli-



(a) Initialzustand (b) $x_{10} = 0 \Rightarrow x_5 = 0, x_3 = 1$

Abbildung 4.17: Einfluss von ECDB auf die Wahl der *Watched Literals* am Beispiel der Klausel $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$

kation wird aber erst nach der derzeit durchgeführten Bearbeitung von $x_5 = 0$ betrachtet und die entsprechende Zuweisung wurde somit noch nicht getätigt. Das hat zur Folge, dass die Klausel erneut betrachtet werden muss, da im nächsten Schritt ein Nachfolger für $\neg x_3$ bestimmt werden muss. Mit ECDB werden die beiden implizierten Zuweisungen $x_5 = 0$ und $x_3 = 1$ sofort vorgenommen, so dass, wie in Abbildung 4.17(b) veranschaulicht, bei der Bestimmung eines Nachfolgers für das Watched Literal x_5 das Literal $\neg x_3$ bereits in diesem Schritt nicht mehr in Frage kommt und die Wahl direkt auf x_{18} fällt.

Insgesamt lassen die in diesem Abschnitt gemachten Ausführungen den Schluss zu, dass die Umsetzung des Konzepts der Watched Literals, bei dem pro Klausel stets zwei Literale gesondert betrachtet werden, um eine Sonderfallbehandlung erweitert werden muss, um Unit Clauses, die nur aus einem Literal bestehen, handhaben zu können. Allerdings sind in der initialen CNF-Formel enthaltene Unit Clauses in diesem Zusammenhang unproblematisch, da sie im Allgemeinen durch die Preprocessing-Einheit eliminiert und nur die sich daraus ergebenden Zuweisungen auf Decision Level 0 festgesetzt werden (siehe Abschnitt 4.2). Diese Idee lässt sich auf Unit Clauses, die während der Konflikt-Analyse generiert wurden, übertragen, indem zunächst eine Backtrack-Operation auf Decision Level 0 vorgenommen und danach die sich aus einer Unit-Clause ergebende Zuweisung unabänderlich für den weiteren Suchprozess auf Decision Level 0 verankert wird. Eine Evaluierung der entsprechenden Klausel während der Boolean Constraint Propagation ist dann überflüssig, so dass auf eine etwaige Sonderfallbehandlung verzichtet werden kann.

4.5 Konflikt-Analyse und Non-Chronological Backtracking

Die Konflikt-Analyse wird immer dann aktiv, wenn sich im Verlauf der Durchführung der Boolean Constraint Propagation ein Konflikt eingestellt hat, das heißt, die widersprüchliche Situation vorliegt, dass eine Variable zeitgleich die Wahrheitswerte 0 und 1 annehmen müsste, um die gegebene CNF-Formel zu erfüllen. Das Ziel der Konflikt-Analyse besteht darin, per Resolution einerseits alle für den Konflikt verantwortlichen Variablenzuweisungen zu identifizieren und andererseits dieses Wissen in einer *Konflikt-Klausel* zu-

sammenzufassen. Die Konflikt-Klausel wird abschließend der bisherigen Klauselmenge hinzugefügt und verhindert, dass die identische, die CNF-Formel nicht erfüllende Teilbelegung erneut gewählt wird. Ebenso wird von der Konflikt-Analyse der so genannte *Backtrack Level* bestimmt, der angibt, welche der Variablenzuweisungen im Rahmen einer *Backtrack-Operation* (auch als *Backtracking* bezeichnet) rückgängig gemacht werden müssen, damit der Konflikt aufgelöst und die Suche nach einer erfüllenden Belegung fortgesetzt werden kann.

Nachfolgend wird sowohl die Konflikt-Analyse als auch das Backtracking im Detail erläutert, wobei beides in dieser Arbeit als eine „Einheit“ angesehen wird, da einzig anhand der generierten Konflikt-Klausel entschieden wird, welche Variablenzuweisungen aufgehoben werden.

4.5.1 Implikationsgraph

Das wichtigste Hilfsmittel zur Durchführung der Konflikt-Analyse ist der so genannte *Implikationsgraph*, mit dem ausgedrückt wird, in welcher Beziehung die verschiedenen Variablenzuweisungen zueinander stehen, das heißt, welche Zuweisungen welche Implikationen ausgelöst haben. Beim Implikationsgraphen handelt es sich um einen gerichteten, azyklischen Graphen, bei dem jede Variablenzuweisung als Knoten repräsentiert wird, während die Kanten ausdrücken, welche Zuweisungen zusammen eine Implikation erzwungen haben. Abbildung 4.18 zeigt den Implikationsgraphen für Beispiel 4.1 und den in Abbildung 4.4 skizzierten Ausschnitt des Decision Stacks, bei dem auf Decision Level 5 die widersprüchliche Situation $x_{18} = 0$ und $x_{18} = 1$ vorliegt. Die Beschriftung der Knoten ist dabei so zu interpretieren, dass beispielsweise $x_6 = 0 \,@\, 1$ bedeutet, dass die Zuweisung $x_6 = 0$ auf Decision Level 1 vorgenommen wurde. Aus der Abbildung wird ersichtlich, dass es sich bei Entscheidungsvariablen, da nicht durch andere Zuweisungen impliziert, um die einzigen Knoten des Implikationsgraphen handelt, die keinerlei eingehende Kanten aufweisen.

Die Realisierung des Implikationsgraphen während des Suchprozesses erfolgt dadurch, dass pro Variable nicht nur der zugewiesene Wahrheitswert gespeichert wird (sofern die Variable belegt ist), sondern auch, ob es sich um eine Implikation handelt und wenn ja, welche Klausel der Auslöser für die Implikation ist. Man betrachte in diesem Zusammenhang den blau unterlegten Teilbereich von Abbildung 4.18, der die drei Zuweisungen $x_{13} = 0 \,@\, 2$, $x_{11} = 1 \,@\, 5$ und $x_{16} = 1 \,@\, 5$ betrifft. Die eingehenden Kanten zum Knoten $x_{16} = 1 \,@\, 5$ repräsentieren erstens, dass $x_{13} = 0$ zusammen mit $x_{11} = 1$ die Implikation $x_{16} = 1$ auslöst, und zweitens, dass es sich bei $(\neg x_{11} \vee x_{13} \vee x_{16})$ um die implikationsauslösende Klausel handelt. Das heißt, zum Zeitpunkt der Zuweisung $x_{16} = 1$ wird neben dem Decision Level, auf dem die Zuweisung erfolgt, zudem noch gespeichert, dass es sich um eine Implikation handelt, die ihre Ursache in einer entsprechenden Belegung der restlichen Literale der Klausel $(\neg x_{11} \vee x_{13} \vee x_{16})$ hat.

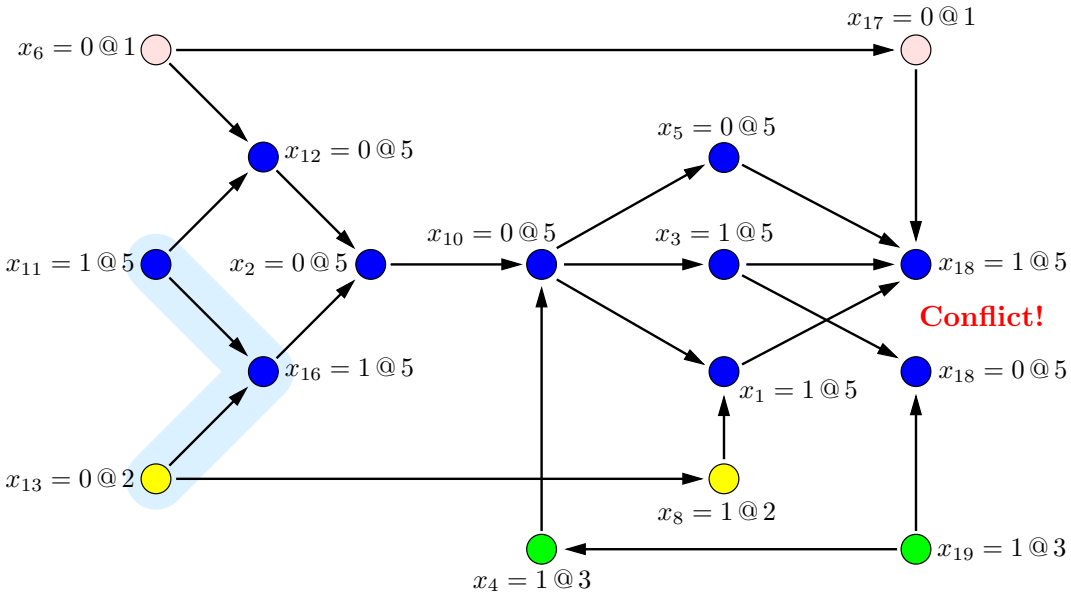


Abbildung 4.18: Implikationsgraph

Solange kein Konflikt vorliegt, gibt es im Implikationsgraphen für jede belegte Variable genau einen Knoten, der die entsprechende Zuweisung an diese Variable charakterisiert. Lediglich im Fall eines Konflikts existieren für eine Variable zwei Knoten, welche die widersprüchliche Belegung charakterisieren. Eine derartige Variable wird in der Literatur auch als *Conflicting Variable* bezeichnet. In Abbildung 4.18 stellt x_{18} die *Conflicting Variable* dar, da sowohl ein Knoten für die Zuweisung $x_{18} = 0$ als auch ein Knoten für $x_{18} = 1$ im Implikationsgraphen vorhanden sind. Beide Belegungen müssten gleichzeitig gelten, um die in diesem Beispiel gewählte Probleminstanz zu erfüllen, was zu einem Widerspruch führt.

Für die nachfolgenden Betrachtungen ist neben der *Dominanz* eines Knotens (gegenüber anderen Knoten) noch der so genannte *Unique Implication Point* von besonderer Bedeutung.

Definition 4.1 (Dominanz)
Ein Knoten a eines Implikationsgraphen dominiert einen Knoten b genau dann, wenn die mit den Knoten a und b assoziierten Variablen auf dem gleichen Decision Level belegt wurden und alle Pfade von der Decision Variable des entsprechenden Decision Levels zum Knoten b durch den Knoten a verlaufen.

Beispielsweise dominiert in dem in Abbildung 4.18 dargestellten Implikationsgraphen der mit $x_2 = 0 @ 5$ beschriftete Knoten den mit $x_{10} = 0 @ 5$ gekennzeichneten Knoten, da alle Pfade von der Decision Variable x_{11} des Decision Levels 5 zu $x_{10} = 0 @ 5$ durch den Knoten

$x_2 = 0 @ 5$ verlaufen. Gleichermaßen dominiert dieser Knoten auch die Knoten $x_1 = 1$, $x_3 = 1$, $x_5 = 0$, $x_{18} = 1$ und $x_{18} = 0$, die allesamt ebenfalls auf Entscheidungsebene 5 belegt wurden.

Definition 4.2 (Unique Implication Point)
Ein Unique Implication Point (UIP) ist ein Knoten des Implikationsgraphen, der im Falle eines Konflikts beide Knoten der Conflicting Variable dominiert.

Aus der Definition der *Dominanz* ist klar, dass jeder UIP auf dem gleichen Decision Level wie die Conflicting Variable belegt wurde. In dem hier gewählten Beispiel sind sowohl $x_{10} = 0$, $x_2 = 0$ als auch die Decision Variable $x_{11} = 1$ *Unique Implication Points*, da alle Pfade von der Entscheidungsvariablen hin zu den mit $x_{18} = 0$ beziehungsweise $x_{18} = 1$ beschrifteten Knoten stets durch diese Knoten führen (die Decision Variable ist immer ein UIP). Anschaulich ausgedrückt ist ein UIP hauptverantwortlich für den aktuell betrachteten Konflikt, da sich der Widerspruch allein als Folge der entsprechenden Zuweisung an den UIP ergibt.

Die Unique Implication Points eines Konflikts werden üblicherweise ausgehend von der widersprüchlichen Belegung hin zur Decision Variable des entsprechenden Decision Levels geordnet, das heißt, in Abbildung 4.18 ist $x_{10} = 0$ der erste, $x_2 = 0$ der zweite und $x_{11} = 1$ der dritte UIP, wobei im nächsten Abschnitt insbesondere der erste UIP von Bedeutung ist.

4.5.2 Konflikt-Analyse

Aufbauend auf den ausgeführten Vorarbeiten wird in diesem Abschnitt die von modernen SAT-Algorithmen durchgeführte Konflikt-Analyse vorgestellt. Algorithmus 4.2 deutet das algorithmische Vorgehen in einer C-ähnlichen Notation an [125]. Zur Illustration der Funktionsweise der Funktion AnalyzeConflict ist in Abbildung 4.19 erneut der Ausschnitt des Decision Stacks dargestellt, der in Abschnitt 4.1 und der dort gewählten CNF-Formel F zu einem Konflikt auf Decision Level 5 führte.

Zunächst wird in Zeile 3 überprüft, ob der aktuelle Decision Level ungleich 0 und die Durchführung der Konflikt-Analyse somit überhaupt notwendig ist. Ist der Decision Level gleich 0, so liegt bereits ein Konflikt vor, ohne dass auch nur eine Entscheidung getroffen wurde. Das heißt, dass die gegebene CNF-Formel an sich bereits einen Widerspruch aufweist. AnalyzeConflict endet mit dem Rückgabewert 0, was in Zeile 15 von Algorithmus 4.1 zur Meldung *UNSATISFIABLE* und dem Abbruch des Suchprozesses führt.

Ansonsten wird in Zeile 4 von Algorithmus 4.2 die so genannte *Conflicting Clause* bestimmt, bei der es sich um genau die Klausel handelt, die in Folge der getätigten Zuweisungen in den Status *unerfüllt* übergangen ist und dadurch den Konflikt ausgelöst hat. Man

Algorithmus 4.2 Umsetzung der Funktion AnalyzeConflict

```
1:  int AnalyzeConflict(void)
2:  {
3:    if (DecisionLevel == 0) { return 0; }// Konflikt auf Level 0, Problem unerfüllbar.
4:    C = ConflictingClause(C);            // Konfliktauslösende Klausel bestimmen.
5:    while(!StopCriterion(C))             // Abbruch-Bedingung prüfen.
6:      {
7:        L = MostRecentlyAssignedLiteral(C);    // Literal von C bestimmen.
8:        V = VariableOfLiteral(L);          // Entsprechende Variable bestimmen.
9:        A = Antecedent(V);           // Implikationsauslösende Klausel bestimmen.
10:       C = Resolve(C, A, V);      // Resolution zw. C und A bzgl. V durchführen.
11:     }
12:   AddClauseToDatabase(C);                // Konflikt-Klausel speichern.
13:   BLevel = ClauseAssertingLevel(C);      // Backtrack Level bestimmen.
14:   return BLevel;
15: }
```

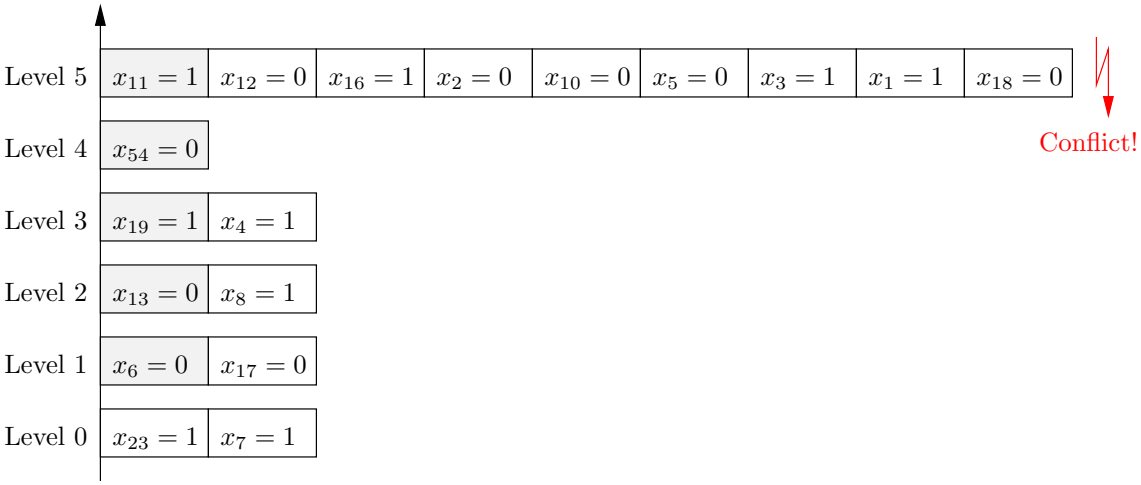


Abbildung 4.19: Ausschnitt des Decision Stacks, der in Abschnitt 4.1 und der dort gewählten CNF-Formel F zu einem Konflikt auf Decision Level 5 führte

beachte an dieser Stelle den Unterschied zwischen *Conflicting Clause* und *Conflict Clause* (Konflikt-Klausel). Bei der erstgenannten Klausel handelt es sich um die konfliktauslösende Klausel, bei der zweiten Klausel um diejenige, die während der Konflikt-Analyse hergeleitet und schlussendlich zur Klauselmenge hinzugefügt wird. In dem in Abbildung 4.19 dargestellten Szenario stellt die Klausel mit ID 14, $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$, die *Conflicting Clause* dar, weil diese, bedingt durch die Zuweisungen auf Decision Level 1 ($x_{17} = 0$) und

Decision Level 5 ($x_5 = 0$, $x_3 = 1$, $x_1 = 1$ und $x_{18} = 0$), aktuell unerfüllt ist. An dem Beispiel lässt sich gut ablesen, dass die Conflicting Clause neben der Variablenbelegung auch von der Reihenfolge abhängt, in der die einzelnen Klauseln während der Boolean Constraint Propagation verarbeitet werden. Werden beispielsweise die Klauseln mit ID 13 und 14, $(\neg x_{19} \vee \neg x_{18} \vee \neg x_3)$ und $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$, in umgekehrter Reihenfolge verarbeitet, das heißt die Klausel mit ID 14 vor der Klausel mit ID 13, wäre auf Decision Level 5 im Anschluss an die Implikation $x_1 = 1$ zunächst die Implikation $x_{18} = 1$ anstelle von $x_{18} = 0$ analysiert worden. Die widersprüchliche Belegung wäre dann aufgrund der Klausel mit ID 13 ausgelöst und somit $(\neg x_{19} \vee \neg x_{18} \vee \neg x_3)$ als Conflicting Clause bestimmt worden.

Im Anschluss daran werden in den Zeilen 5 bis 11 der while-Schleife solange am Konflikt beteiligte Klauseln miteinander resolviert, bis schlussendlich eine Konflikt-Klausel (die „finale“ Resolvente) erzeugt wurde, die das in Zeile 5 festgelegte Abbruchkriterium erfüllt. Dieses mit der Funktion StopCriterion überprüfte Abbruchkriterium dient dazu, festzulegen, welche Variablenzuweisungen als *am Konflikt beteiligt* beziehungsweise *verantwortlich für den Konflikt* angesehen werden und daher in der die widersprüchliche Belegung charakterisierenden Konflikt-Klausel enthalten sein sollen. Wie sich im Verlauf dieses Abschnitts zeigen wird, führen unterschiedliche Abbruchkriterien in der Regel auch zu unterschiedlichen Konflikt-Klauseln.

Innerhalb der Schleife wird zunächst dasjenige Literal der aktuellen Klausel C bestimmt, das zuletzt einen Wahrheitswert zugewiesen bekommen hat (Funktion MostRecentlyAssignedLiteral). In dem hier gewählten Beispiel wäre dies für die Klausel

$$C = (x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5)$$

das Literal $L = x_{18}$, da auf Decision Level 5 mit $x_{18} = 0$ die vor dem Konflikt letzte und zum Literal L im Widerspruch stehende Zuweisung vorgenommen wurde. An dieser Stelle sei erwähnt, dass in PIChaff, MiraXT und PaMiraXT, wie in Abschnitt 4.3 angedeutet, auch jeweils der zu L korrespondierende Aktivitätszähler der VSIDS-Entscheidungsheuristik inkrementiert wird (in diesem Fall der zu x_{18} korrespondierende Zähler), da bei einer dahingehenden Belegung von x_{18} die Klausel C nicht unerfüllt gewesen wäre (unabhängig von den Auswirkungen, die die Zuweisung $x_{18} = 1$ auf andere Klauseln hat). Sollte x_{18} im weiteren Verlauf der Suche einmal als Decision Variable in Frage kommen, so wäre die Zuweisung $x_{18} = 1$ gegebenenfalls eine Möglichkeit, den Suchprozess in einer vielversprechenderen Richtung fortzusetzen.

Nach der Bestimmung von L wird die dazu korrespondierende Variable bestimmt ($V = x_{18}$) und diejenige Klausel identifiziert, die für die Zuweisung $x_{18} = 0$ verantwortlich war, in diesem Beispiel die Klausel mit ID 13. Es gilt daher $A = (\neg x_{19} \vee \neg x_{18} \vee \neg x_3)$. Für die Bestimmung der implikationsauslösenden Klausel wird der im vorherigen Abschnitt eingeführte Implikationsgraph zu Hilfe genommen, beziehungsweise die dafür mit jeder Zuweisung gespeicherte Information, ob es sich um eine Implikation handelt und wenn ja, durch welche

Klausel verursacht.

Als Abschluss eines Durchlaufs durch die while-Schleife werden nun diese Informationen über die Klauseln A und C genutzt, um beide miteinander zu resolvieren und die Resolvente als neue Klausel C festzuhalten (Funktion RESOLVE):

$$\begin{aligned} C &= C \otimes_V A \\ &= (x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5) \otimes_{x_{18}} (\neg x_{19} \vee \neg x_{18} \vee \neg x_3) \\ &= (x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee \neg x_{19}) \end{aligned}$$

Sollte das in Zeile 5 festgelegte Abbruchkriterium der while-Schleife nach einem Durchlauf nicht erfüllt sein, wird ausgehend von der per Resolution neu bestimmten Klausel C ein erneuter Durchlauf begonnen, für den in diesem Beispiel gilt:

$$\begin{aligned} L &= \neg x_1 \\ V &= x_1 \\ A &= (x_{10} \vee \neg x_8 \vee x_1) \\ C &= C \otimes_{x_1} A = (x_{17} \vee \neg x_3 \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \end{aligned}$$

Wurde hingegen das Abbruchkriterium erreicht, wird die while-Schleife beendet und die soeben per Resolution generierte Konflikt-Klausel der bisherigen Klauselmenge hinzugefügt, bevor abschließend der Backtrack Level bestimmt wird, anhand dessen dann die entsprechende Backtrack-Operation durchgeführt werden kann. Die Bestimmung des von der hergeleiteten Konflikt-Klausel abhängigen Backtrack Levels wird im nächsten Abschnitt behandelt, während zum Abschluss dieses Abschnitts zwei mögliche Abbruchkriterien für die while-Schleife beziehungsweise die wiederholte Anwendung der Resolutionsregel vorgestellt werden.

Zunächst sei angemerkt, dass die während der Konflikt-Analyse bis zum Erreichen der gewünschten Konflikt-Klausel wiederholt durchgeführte Resolution den Implikationsgraphen in zwei Teile partitioniert: zum Einen in den Bereich, der direkt mit der widersprüchlichen Belegung zusammenhängt (die so genannte *Conflict Side*), sowie zum Anderen in den Bereich, der den Konflikt auslöst, die so genannte *Reason Side*. Die Menge der zur *Reason Side* gehörenden Knoten wird dabei auf diejenigen Knoten beschränkt, die über eine Kante zu einem Knoten auf der *Conflict Side* verfügen. Vor diesem Hintergrund gelten die zu den Knoten der *Reason Side* korrespondierenden Zuweisungen als verantwortlich für den Konflikt. Abbildung 4.20 stellt exemplarisch die vom SAT-Algorithmus relsat [10], einem der ersten Ansätze mit *Conflict Driven Learning*, vorgenommene Partitionierung des Implikationsgraphen (gestrichelt markiert) für das in diesem Abschnitt gewählte Beispiel dar.

Die in relsat vorgenommene Partitionierung sieht vor, dass die Decision Variable des konfliktauslösenden Decision Levels sowie die am Konflikt beteiligten Variablen, die auf früheren Entscheidungsebenen belegt wurden, auf der *Reason Side* liegen. Bezogen auf die in

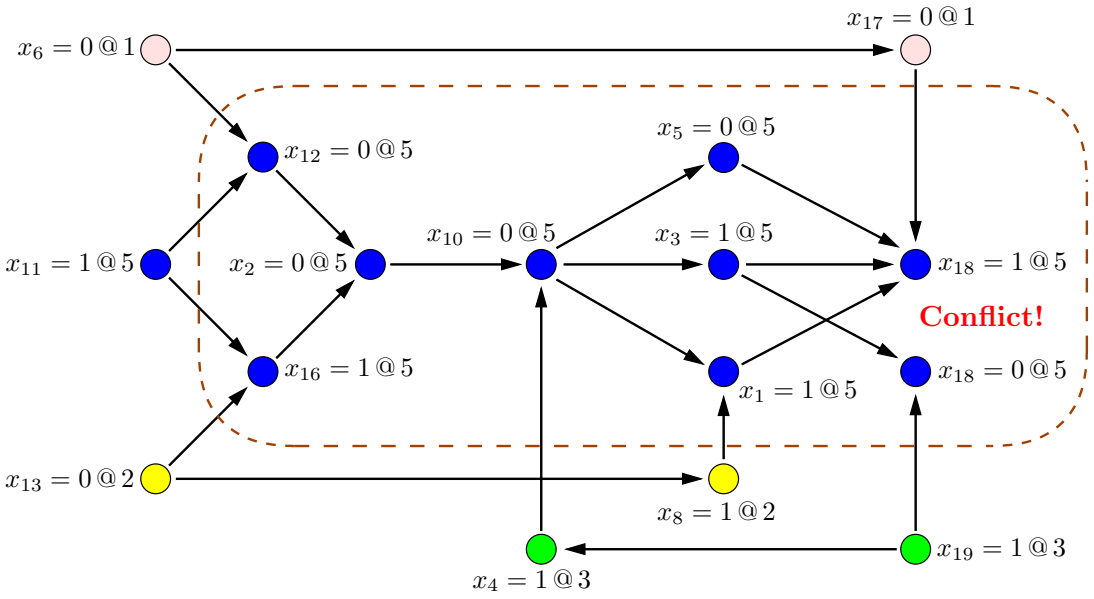


Abbildung 4.20: Partitionierung des Implikationsgraphen nach dem *relsat*-Prinzip

Abbildung 4.20 gestrichelt dargestellte Partitionierung liegen somit die mit $x_6 = 0 @ 1$, $x_{17} = 0 @ 1$, $x_{13} = 0 @ 2$, $x_8 = 1 @ 2$, $x_{19} = 1 @ 3$, $x_4 = 1 @ 3$ und $x_{11} = 1 @ 5$ beschrifteten Knoten auf der *Reason Side* und gelten als Auslöser des Konflikts.

Ausgehend von der Menge der Knoten auf der *Reason Side* und den dazugehörigen Variablenzuweisungen hätte der Konflikt vermieden werden können, wenn die Klausel

$$(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee \neg x_{11} \vee x_{13} \vee x_6)$$

Teil der initialen Klauselmenge gewesen wäre, da sie bei sechs falsch belegten Literalen die korrekte Belegung des siebten Literals erzwungen hätte. Diese Klausel lässt sich bei einem dahingehend gewählten Abbruchkriterium im Rahmen der von Algorithmus 4.2 durchgeführten Resolutions-Schritte wie folgt herleiten:

1. $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5) \otimes_{x_{18}} (\neg x_{19} \vee \neg x_{18} \vee \neg x_3) = (x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee \neg x_{19})$
2. $(x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee \neg x_{19}) \otimes_{x_1} (x_1 \vee x_{10} \vee \neg x_8) = (x_{17} \vee \neg x_3 \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$
3. $(x_{17} \vee \neg x_3 \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \otimes_{x_3} (x_{10} \vee x_3) = (x_{17} \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$
4. $(x_{17} \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \otimes_{x_5} (x_{10} \vee \neg x_5) = (x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$
5. $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \otimes_{x_{10}} (x_2 \vee \neg x_4 \vee \neg x_{10}) = (x_{17} \vee \neg x_{19} \vee \neg x_8 \vee x_2 \vee \neg x_4)$
6. $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee x_2 \vee \neg x_4) \otimes_{x_2} (x_{12} \vee \neg x_{16} \vee \neg x_2) = (x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee x_{12} \vee \neg x_{16})$

7. $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee x_{12} \vee \neg x_{16}) \otimes_{x_{16}} (\neg x_{11} \vee x_{13} \vee x_{16}) =$
   $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee x_{12} \vee \neg x_{11} \vee x_{13})$

8. $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee x_{12} \vee \neg x_{11} \vee x_{13}) \otimes_{x_{12}} (x_6 \vee \neg x_{11} \vee \neg x_{12}) =$
   $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee \neg x_{11} \vee x_{13} \vee x_6)$

Ein gegenüber dem *relsat*-Prinzip abgewandelter Ansatz wird in zChaff verfolgt. Das Ziel ist hierbei, die Partitionierung des Implikationsgraphen möglichst nah am Konflikt vorzunehmen, was, wie sich zeigen wird, in der Regel zu „kürzeren“ Konflikt-Klauseln führt. Erreicht wird dies dadurch, dass der erste UIP des Decision Levels, auf dem der Konflikt auftritt, bereits auf der *Reason Side* liegt, während alle nachfolgenden Knoten dieses Decision Levels auf der *Conflict Side* liegen. Analog zum *relsat*-Prinzip liegen alle am Konflikt beteiligten Knoten des Implikationsgraphen, die Zuweisungen auf früheren Ebenen repräsentieren, ebenfalls auf der *Reason Side*. Diese Art der Partitionierung wird auch als *1UIP*-Prinzip bezeichnet. Abbildung 4.21 zeigt die entsprechende Partitionierung für das hier gewählte Beispiel, wobei der mit $x_{10} = 0 \,@\, 5$ beschriftete Knoten den (vom Konflikt aus gesehen) ersten *Unique Implication Point* des Decision Levels 5 darstellt.

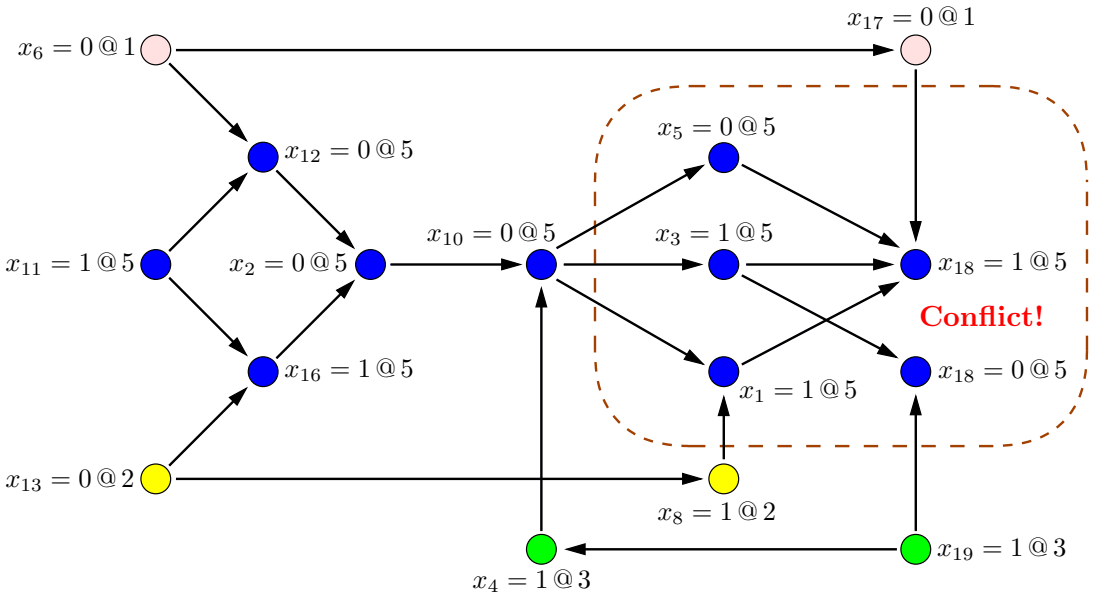


Abbildung 4.21: Partitionierung des Implikationsgraphen nach dem *1UIP*-Prinzip

Gemäß der Partitionierung gelten folglich die Knoten $x_{17} = 0 \,@\, 1$, $x_{19} = 1 \,@\, 3$, $x_8 = 1 \,@\, 2$ und $x_{10} = 0 \,@\, 5$ als Auslöser des Konflikts. Erneut hätte der Konflikt vermieden werden können, wenn die Klausel

$$(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$$

bereits in der Klauselmenge enthalten gewesen wäre, da sie bei drei falsch belegten Literalen die korrekte Belegung des vierten Literals erzwungen hätte. Ebenso wie bei relsat lässt sich diese Klausel bei einem entsprechenden Abbruchkriterium mit Algorithmus 4.2 herleiten:

1. $(x_{17} \vee \neg x_1 \vee x_{18} \vee \neg x_3 \vee x_5) \otimes_{x_{18}} (\neg x_{19} \vee \neg x_{18} \vee \neg x_3) = (x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee \neg x_{19})$
2. $(x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee \neg x_{19}) \otimes_{x_1} (x_1 \vee x_{10} \vee \neg x_8) = (x_{17} \vee \neg x_3 \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$
3. $(x_{17} \vee \neg x_3 \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \otimes_{x_3} (x_{10} \vee x_3) = (x_{17} \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$
4. $(x_{17} \vee x_5 \vee \neg x_{19} \vee x_{10} \vee \neg x_8) \otimes_{x_5} (x_{10} \vee \neg x_5) = (x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$

Bei beiden Beispielen wurde die jeweilige Konflikt-Klausel per Resolution hergeleitet und kann daher aufgrund des Resolutions-Lemmas bedenkenlos der Klauselmenge hinzugefügt werden. Da Konflikt-Klauseln Bereiche des durch die gegebene CNF-Formel aufgespannten Suchraums als unerfüllbar deklarieren, schränkt jede in die Klauselmenge aufgenommene Klausel das verbleibende und noch zu untersuchende Restproblem ein und verhindert, dass ein SAT-Algorithmus während des Suchprozesses den identischen „Fehler“ erneut begeht.

Es fällt auf, dass bei dem hier gewählten Konflikt die Partitionierung nach dem *1UIP*-Prinzip im Rahmen der Konflikt-Analyse zu einer kürzeren Konflikt-Klausel führt als dies bei der Anwendung des *relsat*-Prinzips der Fall ist. Dies konnte durch die in [123] durchgeführten Experimente auf einer repräsentativen Menge von Probleminstanzen bestätigt werden. Im Vergleich zu anderen Strategien ist es mit dem *1UIP*-Prinzip möglich, vergleichsweise kurze Klauseln zu generieren, was sich positiv auf die Performance eines SAT-Algorithmus auswirkt, da die Konflikt-Analyse an sich schneller durchgeführt werden kann. Zudem klassifizieren „kurze“ Konflikt-Klauseln im Vergleich zu „langen“ Konflikt-Klauseln einen größeren Teil des Gesamtproblems als unerfüllbar, das heißt, ein größerer Anteil des Suchraums muss nicht mehr explizit nach einer erfüllenden Belegung durchsucht werden.

4.5.3 Non-Chronological Backtracking

Wie bereits erwähnt, besteht bei heutigen SAT-Algorithmen zwischen der eigentlichen Konflikt-Analyse und dem sich daran anschließenden Backtracking eine enge Verbindung, da allein anhand der hergeleiteten Konflikt-Klausel entschieden wird, welche der Variablenzuweisungen rückgängig gemacht werden (müssen). Im Gegensatz zur DLL-Prozedur, die immer nur bis zur letzten Fallunterscheidung zurückspringt und den Wahrheitswert der dort gewählten Variablen invertiert, was bezogen auf den Decision Stack einem Aufheben aller Zuweisungen des aktuellen Decision Levels entspricht, versuchen moderne SAT-Algorithmen, wenn möglich, die Zuweisungen mehrerer Decision Level aufzuheben. Man spricht hierbei auch vom *Non-Chronological Backtracking* im Gegensatz zum *Chronological Backtracking* der DLL-Prozedur.

Das Vorgehen soll an den beiden im vorhergehenden Abschnitt für die in Abbildung 4.19 dargestellte Konflikt-Situation nach dem *relsat*- beziehungsweise *1UIP*-Prinzip hergeleiteten Konflikt-Klauseln demonstriert werden:

- $(x_{17} \vee \neg x_{19} \vee \neg x_8 \vee \neg x_4 \vee \neg x_{11} \vee x_{13} \vee x_6)$ (*relsat*-Prinzip)
- $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$ (*1UIP*-Prinzip)

Bei beiden Konflikt-Klauseln fällt auf, dass bis auf ein Literal, dem auf Decision Level 5 ein Wahrheitswert zugewiesen wurde (x_{11} beim *relsat*-Prinzip, x_{10} beim *1UIP*-Prinzip; beides sind UIPs des Decision Levels 5), alle anderen Literale auf Decision Level 3 oder früher mit einem Wahrheitswert versehen wurden. Das bedeutet, dass beide Klauseln, wären sie bereits zu Beginn des Suchprozesses Teil der Klauselmenge gewesen, auf Decision Level 3 die Implikation $x_{11} = 0$ beziehungsweise $x_{10} = 1$ ausgelöst hätten. In beiden Fällen wäre die entsprechende Konflikt-Klausel erfüllt gewesen und somit der auf Decision Level 5 aufgetretene Widerspruch verhindert worden.

Genau diese Idee macht man sich beim *Non-Chronological Backtracking* zu Nutze und bestimmt, abgesehen vom Literal der Konflikt-Klausel, das den UIP des Konflikts darstellt, unter allen anderen Literalen den maximalen Decision Level, der dann als *Backtrack Level* festgelegt wird. Handelt es sich bei der Konflikt-Klausel um eine Unit Clause, wird ein Rücksprung auf Decision Level 0 festgesetzt, um auf diesem Decision Level die sich aus der Unit Clause ergebende Implikation unabänderlich für die weitere Suche nach einer erfüllenden Belegung zu verankern.

Bei beiden hier betrachteten Konflikt-Klauseln lautet der Backtrack Level jeweils 3, das heißt, es wird über zwei Decision Level hinweg zurückgesprungen, dann die sich aus der Konflikt-Klausel ergebende Implikation bearbeitet und schlussendlich der Suchprozess fortgesetzt. In Abbildung 4.22 ist exemplarisch der Decision Stack dargestellt, der sich nach einer *Backtrack*-Operation auf Decision Level 3 und dem Verarbeiten der sich aus der Konflikt-Klausel $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$ ergebenden Implikation $x_{10} = 1$ einstellt. Aufgrund der Klausel $(x_2 \vee \neg x_4 \vee \neg x_{10})$ implizieren $x_4 = 1$ und $x_{10} = 1$ die Zuweisung $x_2 = 1$, die ebenfalls auf Decision Level 3 verankert wird.

Diese Vorgehensweise erfordert, dass die während der Konflikt-Analyse durchgeführten Resolutions-Schritte stets in einer Konflikt-Klausel münden, in der immer nur ein Literal des konfliktauslösenden Decision Levels enthalten ist (einer der möglicherweise mehreren *Unique Implication Points*), während alle anderen Literale der jeweiligen Konflikt-Klausel auf früheren Ebenen belegt wurden. Bezogen auf den Implikationsgraphen bedeutet dies, dass die Partitionierung so gewählt sein muss, dass unter allen Literalen des konfliktauslösenden Decision Levels nur ein Literal auf der *Reason Side* liegt, während sich sämtliche nachfolgenden Literale bereits auf der *Conflict Side* befinden. Dann ist gewährleistet, dass der Backtrack Level so bestimmt werden kann, dass die Konflikt-Klausel nach dem

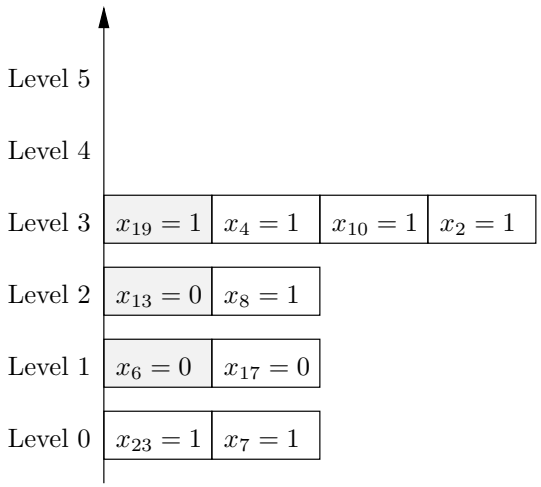


Abbildung 4.22: Decision Stack nach einer Backtrack-Operation auf Decision Level 3 mit anschließender Bearbeitung der Konflikt-Klausel $(x_{17} \vee \neg x_{19} \vee x_{10} \vee \neg x_8)$

Backtracking eine Implikation auslöst und der Suchprozess auf diesem Weg in eine neue Richtung fortgeführt werden kann. Beide im vorherigen Abschnitt diskutierten Methoden zur Herleitung von Konflikt-Klauseln erfüllen per Konstruktion diese Anforderung.

Abschließend sei festgehalten, dass alle drei in der vorliegenden Arbeit entwickelten parallelen SAT-Algorithmen die hier vorgestellte Art der Konflikt-Analyse (anhand des *1UIP*-Prinzips) in Kombination mit *Non-Chronological Backtracking* einsetzen.

4.6 Löschen von Konflikt-Klauseln

Der Übersichtlichkeit halber ist das Löschen von Konflikt-Klauseln nicht explizit in Algorithmus 4.1 dargestellt, gehört aber dennoch zur Standardfunktionalität moderner SAT-Algorithmen. Im Wesentlichen sprechen zwei Gründe dafür, eine derartige Operation in periodischen Abständen durchzuführen: da mit jedem Konflikt eine Konflikt-Klausel generiert und zur Datenbank hinzugenommen wird, wächst die Klauselmenge im Verlauf der Suche nach einer erfüllenden Belegung kontinuierlich an. Auf der einen Seite kann dies dazu führen, dass der Hauptspeicher des Rechners, auf dem der SAT-Algorithmus ausgeführt wird, nicht mehr ausreicht. Je nach Hardware-Ausstattung und Probleminstanz ist beispielsweise die sequentielle Variante von MiraXT bereits in der Lage, pro Sekunde fast 7000 Konflikt-Klauseln zu erzeugen (siehe Tabelle 8.6). Ohne das Löschen eines Teils der Konflikt-Klauseln wäre mangels Speicher zwangsläufig irgendwann der Punkt erreicht, an dem der Suchprozess abgebrochen werden müsste. Auf der anderen Seite bedeutet eine wachsende Klauselmenge auch, dass bei der Durchführung der BCP-Operation immer mehr

Klauseln analysiert werden müssen, was zu einer Verlangsamung der entsprechenden Routine und damit insgesamt zu einer Reduktion der Performance des SAT-Algorithmus führt.

Das Entfernen von Klauseln bezieht sich dabei einzig auf Konflikt-Klauseln, nicht aber auf Klauseln der initialen CNF-Formel, um nicht irrtümlich das gestellte Problem beziehungsweise dessen (Un-)Erfüllbarkeit zu verändern. Weiterhin müssen von der Menge der Konflikt-Klauseln diejenigen von einer Löschoperation ausgenommen werden, die aktuell eine Implikation auslösen und im Konflikt-Fall potenziell für einen der von der Konflikt-Analyse durchgeführten Resolutions-Schritte herangezogen werden. Darüber hinaus gilt es beim Löschen von Konflikt-Klauseln einen möglichst guten Kompromiss zu finden, bei dem der Nachteil durch das Löschen von Konflikt-Klauseln („Fehler“ können erneut auftreten) durch den Vorteil einer wieder beschleunigten BCP-Routine dominiert wird. Vereinfacht ausgedrückt liegt das Ziel darin, nur diejenigen Konflikt-Klauseln zu entfernen, die aufgrund eines festgelegten Kriteriums als irrelevant für den weiteren Suchprozess angesehen werden. Im Folgenden werden verschiedene Ansätze vorgestellt.

Das Löschen von Konflikt-Klauseln folgt in zChaff (in der originalen Version aus dem Jahr 2001) einem Konzept, das in [86] als *Scheduled Lazy Clause Deletion* bezeichnet wird, in anderen Veröffentlichungen aber auch unter dem Namen *Relevance Based Learning* [10, 75] bekannt ist. Für jede während der Konflikt-Analyse generierte und zur Klauselmenge hinzugenommene Konflikt-Klausel wird festgelegt, zu welchem Zeitpunkt während des Suchprozesses die Klausel wieder gelöscht werden soll, wobei der Zeitpunkt genau dann erreicht ist, wenn die Zahl der unbelegten Literale einen bestimmten Grenzwert überschreitet. Sei beispielsweise angenommen, dass während der Konflikt-Analyse eine Klausel bestehend aus 50 Literalen hergeleitet wird und diese wieder aus der Formel entfernt werden soll, wenn davon mindestens 30 Literale unbelegt sind. Sobald eine entsprechende Backtrack-Operation dazu geführt hat, dass 30 oder mehr Literale der Klausel in den Zustand *unbelegt* übergegangen sind, wird die Klausel als für den derzeitigen Bereich des Suchraums irrelevant angesehen und gelöscht, da sie auf „absehbare Zeit“ weder eine Implikation noch einen Konflikt auslösen kann.

Die in Grasp [79] eingesetzte Strategie ist in der Literatur unter dem Namen *Size-Bounded Learning* [10] beziehungsweise *k-Bounded Learning* [75] bekannt und kann wie folgt charakterisiert werden: alle Konflikt-Klauseln, die eine festgelegte Größe überschreiten, werden gelöscht, sobald sie keine Implikation mehr verursachen, also mindestens zwei Literale unbelegt sind, während hingegen „kurze“ Klauseln dauerhaft beibehalten werden.

Im Vergleich zu zChaff und Grasp geht BerkMin [44] beim Löschen von Konflikt-Klauseln einen Schritt weiter und berücksichtigt nicht nur die Länge der in Frage kommenden Klauseln, sondern kombiniert dies mit deren *Aktivität* und *Alter*. Dazu wird, analog zu den Aktivitäten der Variablen, auch für jede Klausel ein Aktivitätszähler eingeführt, der immer dann inkrementiert wird, wenn die entsprechende Klausel während der Konflikt-Analyse

an einem Resolutions-Schritt beteiligt war. Periodisch werden die Klauselaktivitäten durch einen konstanten Faktor geteilt, um den Fokus auf den aktuell untersuchten Bereich des Problems zu lenken. Das führt dazu, dass Konflikt-Klauseln immer dann sehr aktiv sind, wenn sie für eine Vielzahl von Konflikten mitverantwortlich sind und auf diesem Weg dazu beitragen, den noch zu analysierenden Teil des Suchraums zu reduzieren. Das Alter einer Klausel ergibt sich aus der Tatsache, dass Konflikt-Klauseln üblicherweise der Reihe nach in einer Liste abgelegt werden, so dass die am Listenende gespeicherten Konflikt-Klauseln zuletzt erzeugt wurden und gegenüber Klauseln am Listenanfang „jünger“ sind. Mit dem Argument, dass jüngere Klauseln besonders wertvoll sind, da es eines längeren Zeitraums zu deren Herleitung bedurfte, werden bei BerkMin vorrangig die Konflikt-Klauseln gelöscht, die sowohl ein gewisses Alter überschreiten als auch relativ inaktiv sind und somit kaum einen Einfluss auf den bisherigen Suchverlauf hatten.

Die in BerkMin umgesetzten Ideen finden sich mittlerweile in einer Vielzahl an SAT-Algorithmen, wobei, wie auch in MiraXT und PaMiraXT, zumeist auf die Berücksichtigung des Alters der Klauseln verzichtet wird. Etwas anders gestaltet sich die Situation bei PIChaff, wo die sequentiellen SAT-Prozeduren auf Mikroprozessoren ausgeführt werden, denen mit insgesamt 64 kWord nur sehr wenig Speicher zur Verfügung steht. Um mit jedem Aufruf der für das Löschen von Konflikt-Klauseln zuständigen Funktion möglichst viele Speicherzellen wieder freigeben zu können, wird in diesem Szenario ein sehr aggressives Auswahlkriterium verwendet, bei dem alle Konflikt-Klauseln entfernt werden, die aktuell für keine Implikation verantwortlich sind.

4.7 Neustarts

Bei Neustarts handelt es sich um ein probates Mittel, einen SAT-Algorithmus aus Bereichen des Suchraums herauszuführen, in denen aller Voraussicht nach keine erfüllende Belegung gefunden werden kann. Die Argumentation beruht auf der Idee, dass mit der Dauer einer erfolglosen Suche auch die Wahrscheinlichkeit steigt, dass sich das Verfahren in einem Teil des durch die CNF-Formel aufgespannten Suchraums befindet, in dem keine erfüllende Belegung ermittelt werden kann [61]. Damit einhergehend steigt auch die Wahrscheinlichkeit, dass bereits auf niedrigen Entscheidungsebenen die Wahl der dortigen Entscheidungsvariablen „ungünstig“ gewesen ist und der Suchprozess gegebenenfalls mit einer geänderten Ausrichtung neu gestartet werden sollte.

Diese potenziell ungünstigen Zuweisungen werden bei einem Neustart dadurch aufgehoben, dass zunächst die Suche gestoppt, mit Ausnahme der Zuweisungen auf Decision Level 0 die komplette Variablenbelegung rückgängig gemacht und die Suche nach einer erfüllenden Belegung dann erneut auf Decision Level 1 gestartet wird. Nicht geändert werden bei diesem Vorgang die Aktivitäten der einzelnen Variablen, so dass die nach einem Neustart auf Decision Level 1 gewählte Decision Variable in der Regel nicht identisch ist mit der

ehemaligen Decision Variable des ersten Levels. Bei der Umsetzung ist zu gewährleisten, dass sich die Folge der Variablenzuweisungen zwischen zwei Neustarts nicht gleicht, da der SAT-Algorithmus ansonsten in eine Endlosschleife gerät. Verhindert werden kann dies durch ein kontinuierlich ansteigendes Intervall zwischen zwei derartigen Operationen.

Neustarts gehören mittlerweile zu den Standardfunktionen moderner SAT-Algorithmen und sind in der Lage, die zum Lösen einer Probleminstanz benötigte Laufzeit zum Teil erheblich zu minimieren. Laufzeitvorteile stellen sich dabei nicht nur bei erfüllbaren, sondern auch bei unerfüllbaren CNF-Formeln ein, bei denen durch einen Neustart gegebenenfalls relativ schnell Konflikt-Klauseln hergeleitet werden können, die das Restproblem massiv einschränken, ohne Neustart aber erst erheblich später ermittelt worden wären. Aus diesem Grund verfügen auch MiraXT und PaMiraXT über einen derartigen Mechanismus, während in PIChaff auf eine Umsetzung verzichtet wurde, um weniger Speicher für das eigentliche Programm zu benötigen und mehr Speicherzellen für problemspezifische Daten zur Verfügung stellen zu können.

Kapitel 5

Parallele SAT-Algorithmen

Das vorangegangene Kapitel beschäftigte sich mit den Eigenschaften sequentieller SAT-Algorithmen und zeigte auf, mit welchen Methoden versucht wird, ein Maximum an Performance zu erzielen. Eine weitere Möglichkeit der Leistungssteigerung besteht in der Parallelisierung, bei der mehrere sequentielle SAT-Prozeduren zu einem parallelen Algorithmus zusammengefasst werden. Im vorliegenden Kapitel werden mit PSATO [121], //Satz [60], PaSAT [104] sowie ySAT [38] exemplarisch vier verschiedene parallele SAT-Algorithmen vorgestellt. Alle genannten Verfahren basieren, wie auch PIChaff, MiraXT und PaMiraXT, durchgängig auf der Idee, dass eine gegebene Probleminstanz unter den zum Einsatz kommenden sequentiellen SAT-Prozeduren aufgeteilt wird und die einzelnen Bereiche dann parallel bearbeitet werden. Auf eine detaillierte Darstellung anderer Arten der Parallelisierung wird verzichtet. Stellvertretend sei an dieser Stelle das Konzept der *Wettbewerbsparallelität* genannt, das auch unter dem Namen *Algorithm Portfolio* bekannt ist, für weitergehende Informationen sei auf [15, 45] verwiesen.

Allerdings unterscheiden sich die genannten Algorithmen hinsichtlich ihrer Konzeption erheblich, was unter anderem darauf zurückzuführen ist, dass sie für unterschiedliche Hardware-Systeme entwickelt wurden. PSATO und //Satz sind ausgelegt für den Einsatz auf Rechnernetzwerken, bei denen die einzelnen Rechner per Ethernet-Verbindung miteinander verbunden sind. In der Regel existiert bei derartigen Systemen keine Anbindung der verfügbaren Prozessoren an einen gemeinsamen Speicher, weshalb sie im Folgenden als *Rechnernetzwerke mit verteiltem Speicher* bezeichnet werden. Die sequentiellen SAT-Prozeduren des parallelen Algorithmus sind daher jeweils als eigenständiger Prozess realisiert, der über seinen eigenen, von den anderen Prozessen autarken Speicherbereich verfügt. Die Kommunikation, die sowohl bei PSATO als auch //Satz durch einen separaten Master-Prozess gesteuert wird, erfolgt einzig über den Austausch von Nachrichten, das so genannte *Message Passing*.

Die beiden anderen Verfahren, PaSAT und ySAT, wurden hingegen für den Einsatz auf Multiprozessorsystemen vorgesehen und optimiert, bei denen die zuvor angedeutete Anbindung aller Prozessoren an einen gemeinsamen Speicher existiert (nachfolgend als *Multiprozessorsysteme mit gemeinsamem Speicher* bezeichnet). Die verschiedenen Threads von PaSAT und ySAT nutzen daher zumindest einige der Datenstrukturen gemeinsam, was

beispielsweise den Austausch von Konflikt-Klauseln zwischen den Threads erleichtert.

In diesem Kapitel soll aufgezeigt werden, mit welchen Methoden SAT-Algorithmen parallelisiert werden können und welche der für den Informationsaustausch zwischen den Prozessen oder Threads eingesetzten Kommunikationsmodelle im Hinblick auf die Entwicklung von PIChaff, MiraXT und PaMiraXT besonders vielversprechend sind.

5.1 Aufteilung des Suchraums

Die im Folgenden diskutierten parallelen SAT-Algorithmen beruhen auf einer dynamischen Partitionierung des Suchraums, bei der die einzelnen Bereiche von den verschiedenen Prozessen beziehungsweise Threads parallel bearbeitet werden. In diesem Zusammenhang ist es erforderlich, eine Methode zur Verfügung zu stellen, mit der einerseits die Aufteilung des durch die CNF-Formel aufgespannten Suchraums effizient möglich ist. Andererseits sollte der Suchraum dabei in jeweils disjunkte Teile aufgespaltet werden, so dass keine „Überlappungen“ zwischen den einzelnen Teilproblemen vorhanden sind und somit kein Bereich mehrfach untersucht wird.

Prinzipiell ist die Aufteilung des Suchraums einer CNF-Formel in beispielsweise zwei Teile denkbar einfach und kann per Fallunterscheidung bezüglich einer Variablen x_i vorgenommen werden: die eine Hälfte des Suchraums wird durch die Annahme $x_i = 0$ charakterisiert, für die zweite Hälfte gilt die Annahme $x_i = 1$. Aufgeteilt auf zwei parallel agierende SAT-Prozeduren werden in dieser Situation zwei zueinander disjunkte Bereiche des Gesamtproblems untersucht, die sich bezüglich der Zuweisung an die Variable x_i unterscheiden. Um zu gewährleisten, dass während der Suche nach einer erfüllenden Belegung beide Teilprobleme dauerhaft disjunkt bleiben, wird die jeweilige Zuweisung an die Variable x_i für beide sequentiellen SAT-Prozeduren als fest und unabänderlich angenommen. Erreicht wird dies üblicherweise durch die Verankerung der entsprechenden, ein Teilproblem charakterisierenden Variablenbelegungen auf Decision Level 0. Wie in Abschnitt 4.1 erläutert, kommt Decision Level 0 dahingehend eine Sonderrolle zu, dass hier alle Implikationen, die sich aus Unit Clauses ergeben, sowie alle dadurch implizierten Variablenzuweisungen gespeichert sind. Es befindet sich aber keine Decision Variable auf Decision Level 0, so dass eine Backtrack-Operation, in deren Verlauf auch alle Zuweisungen auf Decision Level 0 rückgängig gemacht werden müssten, bei einem sequentiellen SAT-Algorithmus gleichbedeutend mit der Unerfüllbarkeit der CNF-Formel ist. Wird im Falle eines parallelen SAT-Algorithmus die ein Teilproblem spezifizierende (Teil-)Belegung der Variablen der zuvor skizzierten Fallunterscheidung ebenfalls auf Decision Level 0 abgelegt, wird verhindert, dass die entsprechende sequentielle SAT-Prozedur diese speziellen Zuweisungen während der Durchführung einer Backtrack-Operation ändert oder rückgängig macht. In diesem Szenario ist ein Rücksprung inklusive der Rücknahme aller Zuweisungen auf Decision Level 0 gleichbedeutend mit der Unerfüllbarkeit des durch den jeweiligen Prozess analysierten Teilproblems und muss nicht

zwangsläufig auch für die gesamte CNF-Formel gelten. Insbesondere ist aber durch dieses Vorgehen bei einem parallelen SAT-Algorithmus garantiert, dass kein Prozess beziehungsweise Thread den ihm übertragenen Teil des Gesamtproblems verlässt, somit wird kein Bereich mehrfach nach einer erfüllenden Belegung durchsucht.

Ausgehend von diesem Grundprinzip bleibt zu klären, welche Variablen für die Fallunterscheidung berücksichtigt werden dürfen und wie die Festlegung eines bestimmten Teilproblems kodiert werden kann. Zur Beantwortung dieser Fragen wird im Folgenden eine Methodik erläutert, die erstmals zusammen mit PSATO im Jahr 1996 vorgestellt wurde und sich mittlerweile als Standard etabliert hat. Im Kern basiert das Vorgehen darauf, in einem ersten Schritt die Position eines SAT-Algorithmus innerhalb des gesamten, durch die CNF-Formel aufgespannten Suchraums eindeutig zu spezifizieren und damit auch das verbleibende Restproblem zu charakterisieren. Im zweiten Schritt wird diese Information genutzt, um von dem noch zu analysierenden Restproblem einen Teilbereich abzutrennen, der dann von einem anderen Prozess oder Thread des parallelen SAT-Algorithmus gelöst werden kann. Zur Veranschaulichung dient das folgende Beispiel.

Beispiel 5.1
Sei mit $F = (x_{23}) \wedge (x_7 \vee \neg x_{23}) \wedge (x_6 \vee \neg x_{17}) \wedge (x_6 \vee \neg x_{11} \vee \neg x_{12}) \wedge \ldots$ *ein Ausschnitt einer CNF-Formel* F *gegeben. Es sei ferner angenommen, dass ein SAT-Algorithmus bereits beim Einlesen von* F *die Unit-Clause* (x_{23}) *durch die Zuweisung* $x_{23} = 1$ *auf Decision Level 0 erfüllt hat, wodurch sich anhand der zweiten Klausel,* $(\neg x_{23} \vee x_7)$*, die Implikation* $x_7 = 1$ *ergibt, was zur entsprechenden Zuweisung, ebenfalls auf Decision Level 0, führte. Zudem seien* $x_6 = 0$ *und* $x_{11} = 1$ *als Entscheidungsvariablen auf Decision Level 1 beziehungsweise 2 ausgewählt worden, die beide mit* $x_{17} = 0$ *beziehungsweise* $x_{12} = 0$ *je eine Implikation erzwungen haben.*

Abbildung 5.1 zeigt den entsprechenden Ausschnitt des Decision Stacks in der in Kapitel 4 eingeführten Darstellungsform, bei der die beiden Entscheidungsvariablen grau und alle Implikationen weiß unterlegt sind.

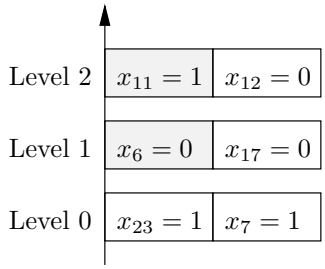


Abbildung 5.1: Decision Stack zu Beispiel 5.1

In der aktuellen Situation würde ein SAT-Algorithmus nun Decision Level 2 verlassen und den Suchprozess mit der Wahl der nächsten Entscheidungsvariablen auf Decision Level

3 fortsetzen, wobei der weitere Verlauf der Suche durch die bereits festgelegte Teilbelegung der Variablen beeinflusst wird. Das heißt, dass sich die aktuelle Position des SAT-Algorithmus innerhalb des gesamten Suchraums einer CNF-Formel durch die Sequenz aller bisher getätigten Variablenzuweisungen, jeweils kombiniert mit der Angabe, ob es sich um eine Decision Variable oder eine Implikation handelt, eindeutig beschreiben lässt. Im Falle von Beispiel 5.1 wäre dies die Folge $[(x_{23}, I), (x_7, I), (\neg x_6, D), (\neg x_{17}, I), (x_{11}, D), (\neg x_{12}, I)]$, wobei die Kürzel I und D für *Implikation* beziehungsweise *Decision* stehen. Eine derartige Sequenz von Zuweisungen wird in [121] als *Guiding Path* bezeichnet.

Als Zwischenfazit kann hier festgehalten werden, dass die aktuelle Position eines SAT-Algorithmus innerhalb des Suchraums einer CNF-Formel in Form eines Guiding Path spezifiziert werden kann. Dieses Wissen wird genutzt, um anhand einer Fallunterscheidung bezüglich einer innerhalb des Guiding Path enthaltenen Variablen das verbleibende Restproblem aufzuspalten, damit ein Teil davon von einem anderen Prozess beziehungsweise Thread bearbeitet werden kann. Das zuvor gegebene Beispiel deutet bereits an, dass nicht alle Variablen des Guiding Path für eine solche Fallunterscheidung in Frage kommen. Sämtliche Implikationen scheiden als potenzielle Kandidaten aus, da der jeweils komplementäre Wahrheitswert unweigerlich zu einem Konflikt führt. Allerdings können die Entscheidungsvariablen für eine Aufspaltung des Problems herangezogen werden: an der jeweiligen Position hat sich der SAT-Algorithmus für einen der beiden Wahrheitswerte entschieden und muss gegebenenfalls per Backtracking für den komplementären Wahrheitswert der entsprechenden Decision Variable prüfen, ob sich in dem dadurch spezifizierten Teil des Suchraums eine erfüllende Belegung bestimmen lässt.

An genau diesen Stellen, in Beispiel 5.1 sind dies die beiden Zuweisungen $x_6 = 0$ auf Decision Level 1 beziehungsweise $x_{11} = 1$ auf Decision Level 2, könnte eine zweite SAT-Prozedur einsteigen, den komplementären Wahrheitswert für die entsprechende Variable wählen und genau das dadurch spezifizierte Teilproblem untersuchen. Diese vom Ausgangsproblem abgespalteten Teilbereiche lassen sich wiederum durch eine Sequenz von Variablenzuweisungen beschreiben, die aus dem „originalen“ Guiding Path gewonnen werden kann. Bezüglich x_6 ist dies der Guiding Path

$$[(x_{23}, I), (x_7, I), (x_6, I)],$$

während bei einer Aufteilung bezüglich x_{11} der Guiding Path

$$[(x_{23}, I), (x_7, I), (\neg x_6, I), (\neg x_{17}, I), (\neg x_{11}, I)]$$

lautet. In beiden Fällen ist die Decision Variable, die für die Fallunterscheidung herangezogen wurde, gegenüber der ursprünglichen Zuweisung in ihrem Wahrheitswert komplementär gewählt. Zudem wurden alle Zuweisungen als Implikationen markiert, um ausschließlich das gewünschte Teilproblem zu spezifizieren. Im originalen Decision Stack wird die Abspaltung eines Teilproblems dadurch signalisiert, dass die Decision Variable, bezüglich derer die Fallunterscheidung vorgenommen wurde, ebenfalls als Implikation markiert wird. Dies deutet

der jeweiligen SAT-Prozedur an, dass der dazu korrespondierende Teil des Suchraums nicht mehr analysiert werden muss beziehungsweise darf. Bezogen auf das Beispiel führt dies anstelle des originalen Guiding Path

$$[(x_{23}, I), (x_7, I), (\neg x_6, D), (\neg x_{17}, I), (x_{11}, D), (\neg x_{12}, I)]$$

bei einer Aufteilung bezüglich x_6 zu

$$[(x_{23}, I), (x_7, I), (\neg x_6, I), (\neg x_{17}, I), (x_{11}, D), (\neg x_{12}, I)]$$

und bezüglich x_{11} zu

$$[(x_{23}, I), (x_7, I), (\neg x_6, D), (\neg x_{17}, I), (x_{11}, I), (\neg x_{12}, I)].$$

Wenngleich prinzipiell jede Decision Variable für eine Partitionierung des Suchraums in Frage kommt, wird bei parallelen SAT-Algorithmen üblicherweise der Suchraum bezüglich jener Decision Variable aufgeteilt, die auf dem niedrigsten Decision Level ausgewählt wurde. Dadurch wird der größte verbleibende Teil des Restproblems aufgespalten, bei dem die Zahl der nicht im Guiding Path enthaltenen Variablen unter allen möglichen Teilproblemen maximal ist. Im Allgemeinen korrespondiert die Anzahl der freien Variablen mit der Laufzeit, die zum Lösen des Problems benötigt wird. Mit diesem Vorgehen wird folglich zunächst das aller Wahrscheinlichkeit nach schwierigste und damit zeitintensivste Teilproblem abgegeben. Zudem verringert sich das Risiko, dass ein Bereich des Suchraums an einen zweiten Prozess abgegeben wird, der das Teilproblem in kürzester Zeit als unerfüllbar identifizieren kann, was eine neuerliche Aufteilung verbunden mit einem nicht zu vernachlässigenden Kommunikationsaufwand nach sich ziehen würde.

Abbildung 5.2 illustriert die algorithmische Umsetzung der zuvor gemachten Überlegungen an einem Beispiel. Auf der linken Seite ist der originale Decision Stack zu sehen, der dazu führt, dass eine SAT-Prozedur in dieser Situation mit der Wahl der nächsten Decision Variable auf Level 3 fortfahren würde. Es sei angenommen, dass zuvor ein Teil des Restproblems an eine zweite sequentielle SAT-Prozedur abgegeben werden soll. Ferner sei vorausgesetzt, dass die Aufspaltung bezüglich der auf Decision Level 1 gewählten Entscheidungsvariablen x_6 vorgenommen wird. Daraus ergibt sich, dass die zweite SAT-Prozedur den Guiding Path $[(x_{23}, I), (x_7, I), (x_6, I)]$ erhält, die entsprechenden Zuweisungen auf Decision Level 0 vornimmt und die Suche nach einer erfüllenden Belegung startet. Der entsprechende Decision Stack ist in Abbildung 5.2 rechts unten dargestellt. Wie zuvor diskutiert, muss die Abspaltung des Teilproblems beim Guiding Path der ersten SAT-Prozedur so vermerkt werden, dass die Zuweisung $x_6 = 0$ als Implikation markiert wird. Dies kann folgendermaßen umgesetzt werden: für jede aktuell belegte Variable wird der Decision Level, auf dem die entsprechende Variable einen Wahrheitswert zugewiesen bekam, um eins dekrementiert. Das heißt, dass alle Zuweisungen des Decision Levels 1 des ursprünglichen Decision Stacks zu zusätzlichen Zuweisungen des Levels 0, alle Zuweisungen auf Decision

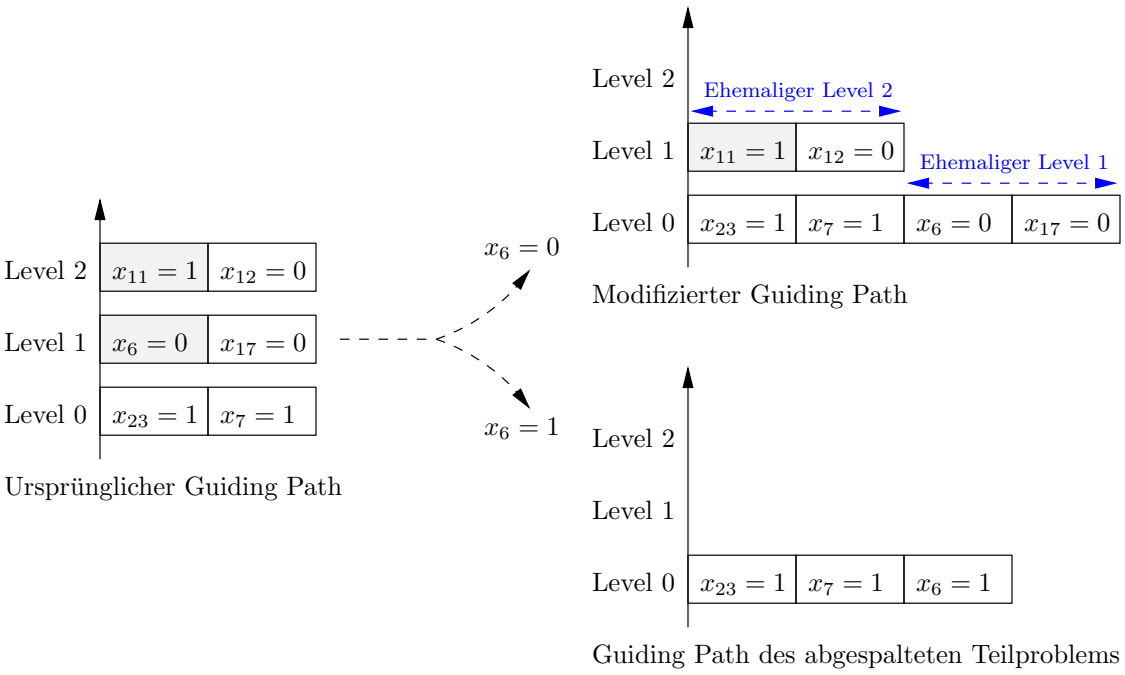


Abbildung 5.2: Mögliche Partitionierung des Decision Stacks aus Beispiel 5.1

Level 2 zu Belegungen auf Decision Level 1 werden und so fort (siehe Abbildung 5.2 rechts oben).

Mit diesen Modifikationen des ursprünglichen Decision Stacks wird zweierlei erreicht: zum Einen ist gewährleistet, dass die beiden Teilprobleme disjunkt sind, da die diesbezüglich relevanten Zuweisungen auf Decision Level 0 vorgenommen wurden und somit von jeglichen Backtrack-Operationen ausgenommen sind. Zum Anderen kann bei einer neuerlichen Aufteilung ein bereits abgegebener Bereich kein zweites Mal weitergeleitet werden.

Gegenüber einem sequentiellen Verfahren (siehe auch Algorithmus 4.1) müssen die sequentiellen SAT-Prozeduren eines parallelen SAT-Algorithmus zur Umsetzung der skizzierten Vorgehensweise so erweitert werden, dass sie als Übergabeparameter einen Guiding Path entgegennehmen und diesen in einem ersten Schritt abarbeiten. Vereinfacht ausgedrückt muss sich die SAT-Prozedur zunächst in den Zustand versetzen, der exakt der „Startposition“ des zu lösenden Teilproblems entspricht. Das wird dadurch erreicht, dass alle Zuweisungen des übermittelten Guiding Path auf Decision Level 0 in den Decision Stack eingetragen und für jede der darin enthaltenen Zuweisungen per *Boolean Constraint Propagation* die daraus folgenden Konsequenzen ermittelt werden. Alle im Folgenden diskutierten Verfahren bauen auf der in diesem Abschnitt eingeführten Art der Aufteilung des Suchraums auf, so dass an den entsprechenden Stellen nicht mehr gesondert auf diesen

Aspekt eingegangen wird.

5.2 Verfahren für Rechnernetzwerke mit verteiltem Speicher

Am Beispiel von //Satz [60] und PSATO [121] wird in diesem Abschnitt beschrieben, wie die Realisierung paralleler SAT-Algorithmen erfolgen kann, die für den Einsatz auf per Ethernet-Verbindung miteinander verknüpften Rechner ausgelegt sind. Eine wesentliche Eigenschaft derartiger Netzwerke liegt darin, dass die an der Ausführung eines parallelen SAT-Algorithmus beteiligten Rechner keinen Zugriff auf einen gemeinsamen Speicher haben. Das hat zur Folge, dass einerseits die verschiedenen sequentiellen SAT-Prozeduren weitgehend autarke Prozesse sind (und jeweils eine eigene Klauseldatenbank verwalten) und andererseits die Kommunikation zwischen den Prozessen nur über den Austausch von Nachrichten, das so genannte *Message Passing*, erfolgen kann.

Abbildung 5.3 zeigt das Design von //Satz [60], dessen sequentiellen SAT-Prozeduren auf dem SAT-Algorithmus Satz [72] aufbauen. Deutlich zu erkennen ist, dass //Satz einem *Master/Client*-Modell folgt, bei dem die von den Clients ausgeführten sequentiellen SAT-Prozeduren, gesteuert durch einen separaten Master-Prozess, gemeinsam eine CNF-Formel bearbeiten.

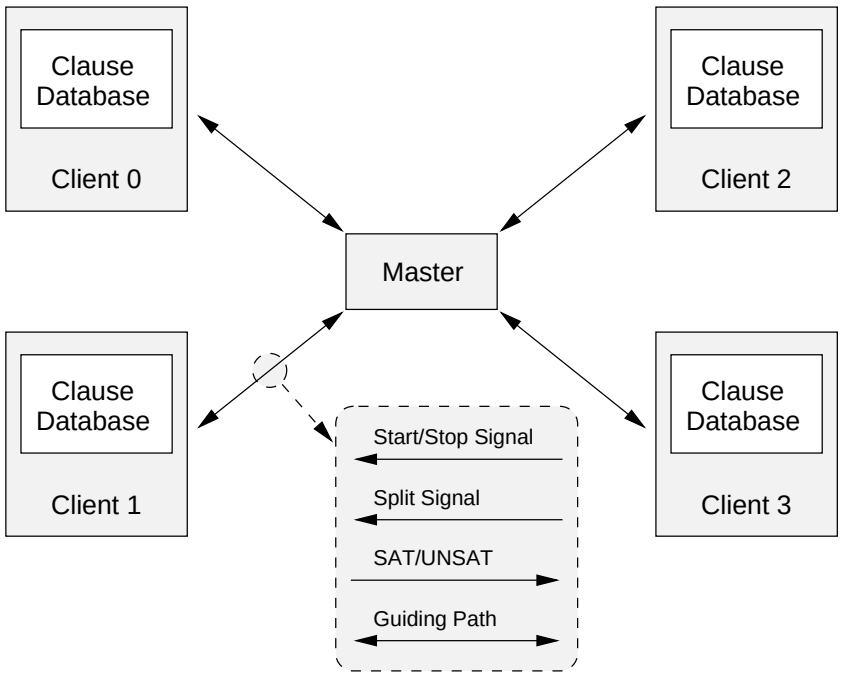


Abbildung 5.3: Design //Satz

Der Master ist für das Starten und Stoppen der Clients und die Weitergabe von Teilproblemen verantwortlich, wobei die Kommunikation stets über den Master abgewickelt wird. Das bedeutet, dass in //Satz kein Austausch von Informationen unmittelbar zwischen den verschiedenen Clients stattfindet. Die Clients nehmen vom Master-Prozess übermittelte und noch unbearbeitete Teilprobleme entgegen und bearbeiten diese. Die Aufteilung des durch die CNF-Formel aufgespannten Suchraums lässt sich im Fall von //Satz wie folgt charakterisieren: sobald ein Client in einen inaktiven Zustand übergeht, also den ihm zugewiesenen Bereich des Suchraums abgearbeitet hat, aber keine erfüllende Belegung ermitteln konnte, wird vom Master derjenige aktive Client bestimmt und zur Abspaltung eines Teilproblems aufgefordert, der aktuell das größte Restproblem besitzt. Analog zu den Überlegungen des vorherigen Abschnitts wird vom Master dasjenige Teilproblem als das „größte Restproblem“ angesehen, dessen Spezifikation als Guiding Path unter den Teilproblemen aller noch aktiven Clients aus den wenigsten Variablenzuweisungen besteht. Im ersten Schritt kontaktiert der Master daher alle aktiven Clients, nimmt von diesen den jeweiligen Guiding Path entgegen (alle Variablenzuweisungen bis einschließlich der ersten Decision Variable) und entscheidet dann im zweiten Schritt anhand dieser Daten, welcher Client tatsächlich seinen Bereich des Suchraums aufteilen soll. Das entgegengenommene Teilproblem des ausgewählten aktiven Clients wird schlussendlich vom Master-Prozess an den inaktiven Client weitergeleitet.

Während der Initialisierungsphase von //Satz wird durch den Master-Prozess lediglich ein Client gestartet, während alle weiteren Clients in einem inaktiven Zustand verharren. Zusammen mit der zuvor skizzierten Aufteilung des Suchraums bedingt dies, dass zunächst das Gesamtproblem solange aufgeteilt wird, bis jeder Client über ein initiales Teilproblem verfügt. Stellt sich während des Suchprozesses die Situation ein, dass alle Clients inaktiv sind, ist die Probleminstanz unerfüllbar; unterteilt in disjunkte Bereiche wurde der gesamte durch die CNF-Formel aufgespannte Suchraum von den sequentiellen SAT-Prozeduren analysiert, allerdings ohne eine erfüllende Belegung gefunden zu haben. In derartigen Fällen werden die Clients vom Master-Prozess gestoppt und //Satz beendet. Auch wenn ein Client ein Modell für das gestellte Problem ermittelt hat, initiiert der Master zuerst das Stoppen aller Clients, bevor //Satz beendet wird.

In Abbildung 5.4 ist schematisch das Design von PSATO [121] dargestellt, einer parallelen Variante des sequentiellen SAT-Algorithmus SATO [120]. Das Grundgerüst besteht aus diversen Clients, welche die sequentiellen SAT-Prozeduren ausführen, und einem separaten Master-Prozess, der sich für das Starten und Stoppen der Clients sowie die Weitergabe von Teilproblemen verantwortlich zeigt, und ist somit identisch zu //Satz. Der wesentliche Unterschied beider Verfahren liegt darin, dass der Master-Prozess von PSATO, im Gegensatz zu seinem Pendant bei //Satz, immer eine gewisse Anzahl noch unbearbeiteter Teilprobleme auf Vorrat hält. Vereinfacht ausgedrückt spalten die Clients dazu in festgelegten Intervallen jeweils einen Teil des von ihnen aktuell untersuchten Bereichs des Gesamtproblems ab und übergeben diesen an den Master, der das erhaltene Teilproblem, kodiert als

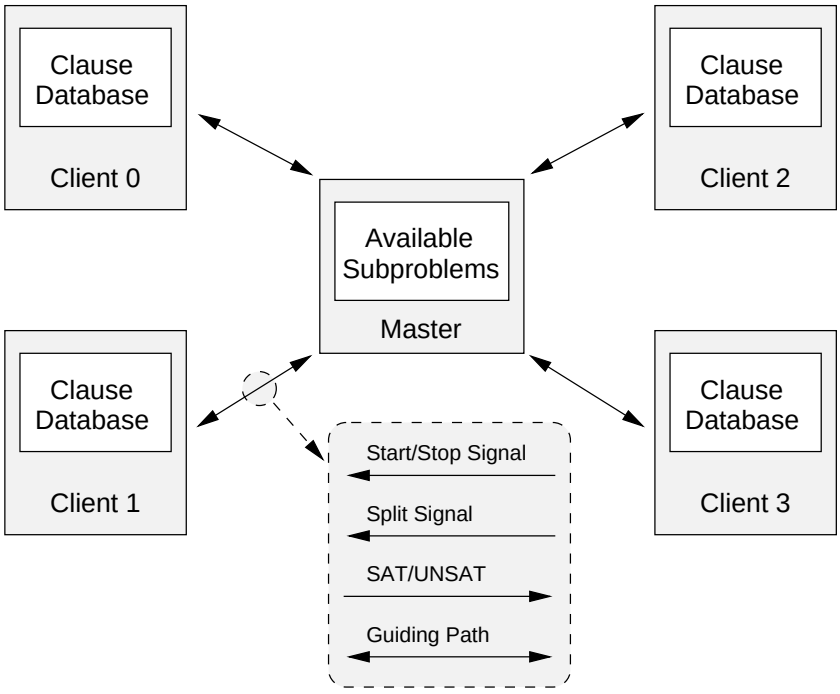


Abbildung 5.4: Design PSATO

Guiding Path, in *Available Subproblems* ablegt. Im Vergleich zu //Satz hat diese Strategie den Vorteil, dass der Master-Prozess einem inaktiv gewordenen Client sofort eines der in *Available Subproblems* gespeicherten Teilprobleme zuweisen kann, ohne vorher einen aktiven Client kontaktieren zu müssen. Dem inaktiven Client entsteht dadurch nur eine minimale Wartezeit.

Wie sich in den Kapiteln 7 und 9 zeigen wird, basieren sowohl PIChaff als auch PaMiraXT auf einem Konzept, das Ähnlichkeiten zu //Satz aufweist. Es wird ebenfalls versucht, bei der Aufteilung des Suchraums zunächst das größte Restproblem aufzuspalten. PIChaff und PaMiraXT erweitern das Design von //Satz um den Austausch von Konflikt-Klauseln, was zu einer signifikanten Reduktion der zum Lösen einer Probleminstanz benötigten Laufzeit führen kann (siehe auch den nachfolgenden Abschnitt). Auf das Speichern von noch unbearbeiteten Teilproblemen auf Seiten des Masters, wie in PSATO geschehen, verzichten beide Ansätze, da entsprechende Experimente gezeigt haben, dass die Wartezeiten der Clients auch ohne diese Technik so verschwindend gering sind, dass sie vernachlässigbar sind.

5.3 Verfahren für Multiprozessorsysteme mit gemeinsamem Speicher

Mit PaSAT und ySAT werden in diesem Abschnitt zwei parallele SAT-Algorithmen vorgestellt, die beide auf einem Thread-Konzept basieren und auf Multiprozessorsysteme zugeschnitten sind, bei denen die von den Threads ausgeführten sequentiellen SAT-Prozeduren Zugriff auf einen gemeinsamen Speicherbereich haben. Dies ist beispielsweise bei Dual- und Multi-Core Prozessoren und Mehrprozessorsystemen mit mehreren auf der Hauptplatine integrierten Prozessoren der Fall. Die Anbindung aller Prozessoren beziehungsweise CPU-Kerne an einen gemeinsamen Speicher bietet die Möglichkeit, die Kommunikation zwischen den Threads mit Hilfe entsprechender Datenstrukturen und Signale komplett über den Speicher abzuwickeln, was gegenüber einer Kommunikation per *Message Passing* die wesentlich schnellere Variante ist. Gerade im Hinblick auf kommende Generationen von Multi-Core Prozessoren mit mehreren Dutzend CPU-Kernen werden threadbasierte parallele Verfahren auf dem Gebiet der SAT-Algorithmen zunehmend an Bedeutung gewinnen, da nur diese die Chance bieten, die verfügbaren Hardware-Ressourcen optimal auszunutzen.

Abbildung 5.5 beschreibt schematisch das Design von PaSAT [104] anhand einer Konfiguration mit vier Threads. Im Gegensatz zu //Satz und PSATO verfügt PaSAT über keinen separaten Master-Prozess, stattdessen erfolgt jegliche Kommunikation direkt zwischen den Threads. Zu Beginn der Suche nach einer erfüllenden Belegung startet zunächst nur ein Thread mit der Bearbeitung des Gesamtproblems, während die anderen Threads in einem inaktiven Zustand verbleiben und auf die Zuweisung von Teilproblemen warten. Die Aufteilung des durch die CNF-Formel aufgespannten Suchraums ist in PaSAT so geregelt, dass ein zufällig bestimmter, aktiver Thread einen noch unbearbeiteten Bereich seines eigenen Teilproblems abgibt (in Abbildung 5.5 durch Pfeile zwischen den Threads symbolisiert). Es wird folglich keine Bewertung der durch die aktiven Threads bearbeiteten Teilprobleme vorgenommen, beispielsweise um immer das größte Restproblem aufzuspalten.

Weiterhin verfügen alle Threads über eine Anbindung an die so genannte *Conflict Clause Database*, die dem Austausch von Konflikt-Klauseln zwischen den Threads dient. Jeder Thread legt dazu sämtliche von ihm generierten Konflikt-Klauseln, die eine bestimmte Länge nicht überschreiten, in der *Conflict Clause Database* ab und überträgt im Gegenzug in regelmäßigen Abständen die von anderen Threads bereitgestellten Konflikt-Klauseln in die eigene lokale Klauseldatenbank. Jede von einem Thread entgegengenommene Konflikt-Klausel schränkt den Suchraum des aktuell von diesem Thread betrachteten Teilproblems ein, da die Klausel eine Kombination von Variablenzuweisungen beschreibt, mit der die gegebene CNF-Formel nicht erfüllt werden kann. Das beste Beispiel hierfür ist eine Konflikt-Klausel, die sofort nach Erhalt einen Konflikt auslöst. Der entsprechende Thread befindet sich folglich in einem Bereich, der von einem anderen Thread bereits als unerfüllbar identifiziert wurde. Die Bearbeitung kann an dieser Stelle sofort abgebrochen, anhand der

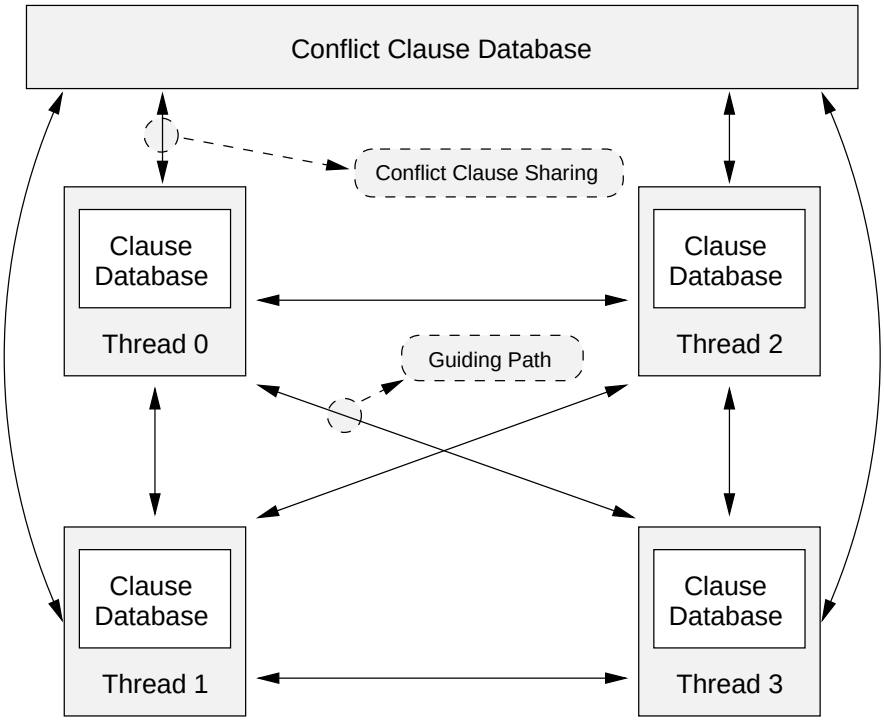


Abbildung 5.5: Design PaSAT

erhaltenen Klausel eine Backtrack-Operation durchgeführt und der Suchprozess in eine andere Richtung fortgesetzt werden. Gegebenenfalls kann dabei ein Thread auch komplett gestoppt werden, falls eine entsprechende Konflikt-Klausel dessen gesamtes Teilproblem als unerfüllbar deklariert.

Unabhängig von PaSAT gilt es beim Austausch von Konflikt-Klauseln eine gute Balance zwischen zwei gegenläufigen Effekten zu finden. Auf der einen Seite kann, wie zuvor angedeutet, die Weitergabe von Konflikt-Klauseln dazu führen, dass der Suchraum der von den sequentiellen SAT-Prozeduren bearbeiteten Teilprobleme erheblich eingeschränkt wird, was sich dann positiv auf die zum Lösen einer Probleminstanz benötigte Laufzeit auswirkt. Auf der anderen Seite ist der Austausch von Klauseln immer mit einem Mehraufwand verbunden, der die Laufzeit eines parallelen SAT-Algorithmus negativ beeinflusst. In diesem Zusammenhang ist neben der eigentlichen Weitergabe und dem Empfang von Konflikt-Klauseln zu bedenken, dass eine entgegengenommene Klausel direkt nach Erhalt bewertet werden muss, um im Fall einer Implikation oder eines Konflikts die entsprechenden Operationen einzuleiten. Ebenso verlangsamt jede in die Klauselmenge aufgenommene Klausel die BCP-Routine.

Aus diesen Gründen wurde in PaSAT daher festgelegt, dass Klauseln, die aus mehr als fünf Literalen bestehen, nicht weitergeleitet werden. Auf diesem Weg wurde erreicht, dass die Anzahl der für den Austausch potenziell in Frage kommenden Konflikt-Klauseln und damit der zu betreibende Aufwand in einem vertretbaren Rahmen bleibt, ohne dabei „kurze“ Klauseln, die den Suchraum weit stärker einschränken als „lange“ Klauseln, von einem Austausch auszuschließen. Für PIChaff und PaMiraXT, bei denen ebenfalls nur Klauseln bis zu einer bestimmten Länge weitergereicht werden, hat sich ein Limit von 3 als geeignet erwiesen.

Das letzte hier vorgestellte parallele SAT-Verfahren ist ySAT [38], das ebenfalls den Fokus auf Hardware-Plattformen setzt, bei denen alle sequentiellen SAT-Routinen, wiederum als Threads realisiert, auf einen gemeinsamen Speicher zugreifen können. Abbildung 5.6 zeigt schematisch die Konzeption von ySAT.

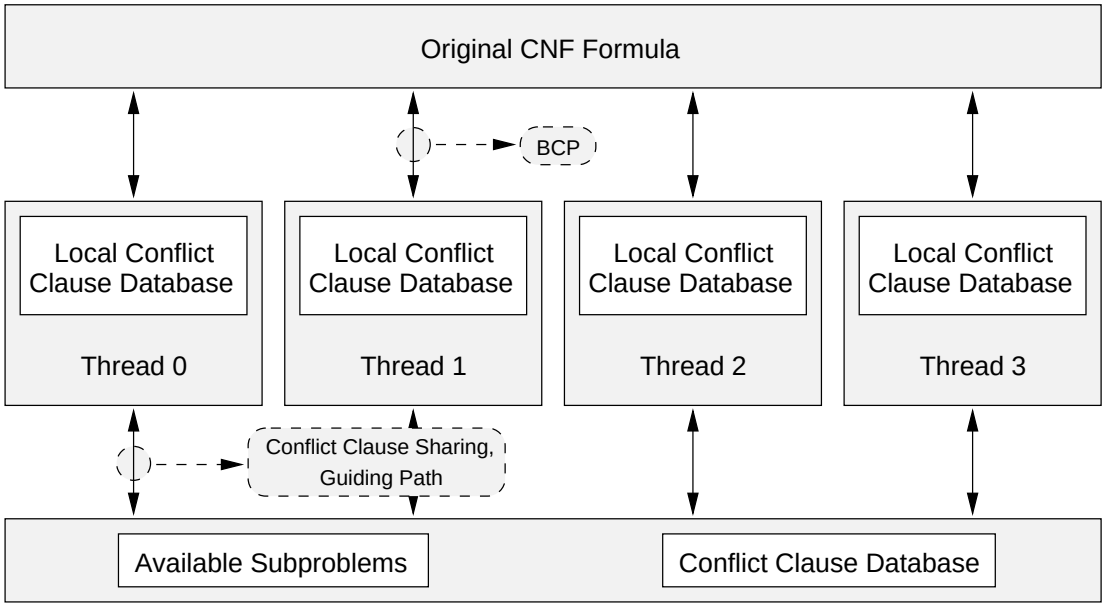


Abbildung 5.6: Design ySAT

Im Unterschied zu den drei bisher beschriebenen parallelen SAT-Algorithmen ist ySAT das einzige Verfahren, bei dem die sequentiellen SAT-Prozeduren die Klauseln der originalen CNF-Formel gemeinsam nutzen. Um auf der initialen Klauselmenge eine effiziente Durchführung der Boolean Constraint Propagation zu gewährleisten, sind nur lesende Zugriffe auf diese Klauseln erlaubt, was es den BCP-Routinen mehrerer Threads ermöglicht, zeitgleich dieselbe Klausel evaluieren zu können. Einzig die für die Klauseln der originalen Probleminstanz mitgeführten Watched Literals werden von den Threads lokal verwaltet.

Analog zu PSATO wird in ySAT eine Menge von noch nicht bearbeiteten Teilproblemen auf Vorrat gehalten. Jeder Thread prüft nach der Wahl einer Decision Variable, ob die Anzahl der aktuell in *Available Subproblems* gespeicherten Teilprobleme unter ein vorgegebenes Limit gefallen ist. Sollte dies der Fall sein, spaltet der Thread bezüglich der soeben gewählten Decision Variable sein eigenes Teilproblem auf und überträgt den neu erzeugten Guiding Path, der dem abgetrennten Bereich des eigenen Teilproblems entspricht, an *Available Subproblems*. Tritt die Situation ein, dass ein Thread sein Teilproblem gelöst hat, aber keine erfüllende Belegung ermitteln konnte, entnimmt er *Available Subproblems* ein Teilproblem und bearbeitet dieses. Erneut besteht der Vorteil darin, dass inaktiven Threads keinerlei Wartezeiten entstehen und sie den Suchprozess sofort in einem neuen Bereich des Suchraums fortführen können. Die in ySAT gewählte Variante hat aber den entscheidenden Nachteil, dass ein Thread nach jeder Wahl einer Decision Variable zunächst prüft, ob Bedarf an einer weiteren Aufspaltung des eigenen Teilproblems besteht, was aufgrund des damit verbundenen Aufwands nicht sinnvoll ist.

Eine Gemeinsamkeit von PaSAT und ySAT besteht im Austausch von Konflikt-Klauseln zwischen den Threads. In ySAT wurde er folgendermaßen realisiert: jede von einem Thread generierte Konflikt-Klausel wird unabhängig von deren Länge oder anderen Kriterien in die *Conflict Clause Database* eingetragen. In periodischen Intervallen prüfen die Threads, welche der darin enthaltenen Konflikt-Klauseln von anderen Threads erzeugt wurden und somit neue Informationen für sie selbst darstellen. Alle derartigen Konflikt-Klauseln werden in den lokalen Speicherbereich des jeweiligen Threads übertragen (in Abbildung 5.6 als *Local Conflict Clause Database* bezeichnet), die Watched Literals bestimmt und überprüft, ob die übernommene Klausel eine Implikation oder einen Konflikt auslöst, was gegebenenfalls weitere Operationen nach sich ziehen würde. Ein wichtiger Unterschied gegenüber PaSAT liegt darin, dass in ySAT jeder Thread Zugriff auf alle von den anderen Threads generierten Konflikt-Klauseln hat, da diese vor dem Einfügen in die *Conflict Clause Database* nicht hinsichtlich ihrer Länge bewertet und unter Umständen von einer Weitergabe ausgeschlossen werden. Da die Threads alle verfügbaren Konflikt-Klauseln in die eigene Klauseldatenbank übertragen, hat dies zugleich den Nachteil, dass potenziell auch Klauseln übernommen werden, die keinerlei positiven Einfluss auf den eigenen Suchprozess haben, aber unter Umständen die BCP-Routine massiv verlangsamen.

Als Vorgriff auf Kapitel 8 sei erwähnt, dass in MiraXT das Konzept einer einzigen Klauseldatenbank von ySAT übernommen wurde. Diese Idee wurde um das Speichern aller Klauseln, unabhängig ob in der initialen Klauselmenge enthalten oder während des Suchprozesses hergeleitet, in einer einzigen Datenstruktur erweitert. Bezogen auf Abbildung 5.6 entspricht dies einem Zusammenführen der beiden Speicherbereiche *Original CNF Formula* und *Conflict Clause Database*, wobei sichergestellt wird, dass sich mehrere, zeitgleich auf die Klauseldatenbank zugreifende Threads nicht gegenseitig blockieren. In MiraXT besteht daher im Gegensatz zu ySAT keine Notwendigkeit mehr, dass die Threads die Konflikt-

Klauseln in ihrem lokalen Speicherbereich halten, was zu einer signifikanten Reduktion des Speicherbedarfs führt.

Kapitel 6

Multiprozessorsystem

In den beiden vorangegangenen Kapiteln wurde ein Überblick über die in sequentiellen und parallelen SAT-Algorithmen eingesetzten Methoden und Techniken gegeben. Es liegt auf der Hand, dass erst ein vielversprechendes Konzept in Kombination mit geeigneten Datenstrukturen und einer effizienten algorithmischen Umsetzung zu einem leistungsstarken SAT-Algorithmus führen. Exemplarisch sei in diesem Zusammenhang das Konzept der *Watched Literals* zur Durchführung der *Boolean Constraint Propagation* genannt, von dem in [124] gezeigt werden konnte, dass es im Vergleich zu anderen Techniken den L2-Cache moderner Prozessoren am besten ausnutzt.

Besonders bei parallelen SAT-Algorithmen ist es wichtig, die zugrundeliegende Hardware-Plattform und Aspekte wie die Anzahl der verfügbaren Prozessoren, die Anbindung an den Speicher und die Kommunikationskanäle zwischen den einzelnen Prozessoren zu berücksichtigen. Nur bei einer optimalen Ausnutzung der zur Verfügung stehenden Hardware-Ressourcen lässt sich durch das Zusammenspiel mehrerer parallel agierender sequentieller SAT-Prozeduren ein maximaler Performance-Gewinn gegenüber einem sequentiellen Verfahren erzielen. In den Kapiteln 7, 8 und 9 werden mit PIChaff, MiraXT und PaMiraXT drei Entwicklungen vorgestellt, die allesamt speziell an die jeweils anvisierte Zielplattform angepasst wurden und so ein Maximum an Leistung erzielen.

Den Anfang macht dabei PIChaff, das als Hardware-Plattform ein am Lehrstuhl für Rechnerarchitektur entwickeltes Multiprozessorsystem nutzt, so dass im vorliegenden Kapitel zunächst die wichtigsten Eigenschaften dieses Systems thematisiert werden. Abbildung 6.1 zeigt die Kernmodule dieses Systems. Eine ISA-Steckkarte dient als Trägerboard und kann in jedem PC mit entsprechender Schnittstelle genutzt werden. Sie bietet Platz für bis zu neun Recheneinheiten, mit denen ein gestelltes Problem parallel gelöst werden kann, sowie für einen separaten Kommunikationsprozessor. Dieser mittig auf dem Trägerboard platzierte Baustein fungiert als Steuerzentrale und ist unter anderem für die Abwicklung der Kommunikation zwischen den einzelnen Mikroprozessoren verantwortlich. Im Folgenden wird die technische Ausstattung der drei Hauptkomponenten – Recheneinheiten, Kommunikationsprozessor und Trägerboard – vorgestellt. Dabei beschränken sich die einzelnen Abschnitte im Wesentlichen auf die Erläuterung derjenigen Aspekte, die für ein grundlegendes Verständnis unabdingbar sind. Zur Erhöhung der Übersichtlichkeit wird auf die

explizite Darstellung von Schaltplänen verzichtet. Die Datenpfade werden lediglich schematisch angedeutet. Weiterhin wird die notwendige Programmierung der diversen Logikbausteine ausgespart, dies stand im Mittelpunkt mehrerer Studien- und Diplomarbeiten [3, 37, 59, 88].

Abbildung 6.1: Multiprozessorsystem

6.1 Recheneinheiten

Die in Abbildung 6.2 dargestellte Recheneinheit (in Kapitel 7 auch als *Processor Node* bezeichnet) ist das „Arbeitstier“ des Multiprozessorsystems. Sie besteht aus einem *Microchip PIC17C43* Mikroprozessor, einem auf der Rückseite der Platine montierten, externen Speicher mit 64 kWord Speicherkapazität sowie einem programmierbaren Logikbaustein vom Typ *Atmel ATF1500*. Dieser ist durch die Generierung entsprechender Steuersignale verantwortlich für die Anbindung des Speichers an den PIC17C43 Mikroprozessor sowie für den Datenaustausch zwischen der Recheneinheit und dem Kommunikationsprozessor. Im Folgenden wird anstelle *Logikbaustein* oftmals die vom englischen Begriff *Programmable Logic Device* stammende Kurzform PLD verwendet.

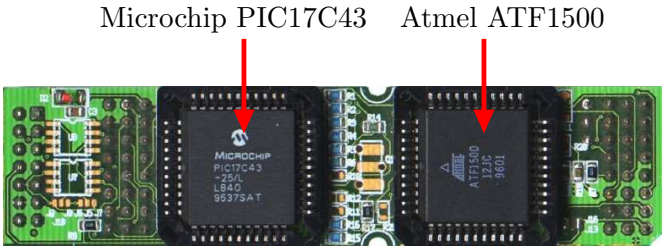


Abbildung 6.2: Recheneinheit

Beim Microchip PIC17C43 Mikroprozessor [84] handelt es sich um einen 8 Bit Prozessor in RISC-Technologie. Zu den wichtigsten Eigenschaften gehören:

- 8 Bit Datenbus,
- 16 Bit Adressbus, ermöglicht die Nutzung des externen Speichers,
- 454 interne Speicherplätze zu je 8 Bit,
- 50 Funktionsregister wie beispielsweise Arbeits- und Statusregister,
- 33 frei konfigurierbare I/O Leitungen,
- 8 Bit Hardware-Multiplizierer,
- 4 kWord internes, einmal beschreibbares ROM,
- 16 Ebenen Hardware-Stack, begrenzt den Einsatz rekursiver Funktionen,
- 11 verschiedene Interrupt-Quellen gruppiert in vier Prioritäts-Ebenen,
- 4 kaskadierbare 8 und 16 Bit Timer,
- eine serielle Schnittstelle mit einer Übertragungsrate von maximal 312,5 kBaud,
- 33 MHz maximale Taktfrequenz,
- etwa 100 mA Stromverbrauch bei 32 MHz Taktfrequenz und 5 Volt Spannung.

Da das 4 kWord große, interne ROM des PIC17C43 Mikroprozessors nur einmal beschreibbar ist (danach sind nur noch Lesezugriffe möglich), scheidet es als Speicherort für die von den Recheneinheiten auszuführenden Programme aus. Jede noch so minimale Änderung an einem bestehenden Programm als auch der Wechsel auf eine gänzlich neue Anwendung würden bei einer derartigen Vorgehensweise neue, wiederum nur einmal beschreibbare Mikroprozessoren erfordern, was allein aus Kostengründen nicht durchführbar ist. Daher wurde vereinbart, alle von den Recheneinheiten abzuarbeitenden Programme (wie etwa PIChaff) und die dafür benötigten Daten (Klauseln, Variablenbelegung, sonstige Statusvariablen) im externen, 64 kWord großen Speicher abzulegen und von dort auszuführen. Das ROM beinhaltet lediglich eine Art „Betriebssystem“, das alle Funktionen bereitstellt, die direkt nach dem Einschalten des Multiprozessorsystems ausgeführt werden müssen. In der in Abschnitt 7.1 skizzierten Variante sind dies die Konfiguration eines der 16 Bit Timer sowie die Aktivierung der benötigten Interrupt-Signale mitsamt entsprechender Routinen, die beim Eintreten des jeweiligen Interrupts automatisch aufgerufen werden. Vom Benutzer am Rechner entwickelte Anwendungen, die über die ISA-Schnittstelle an den Kommunikationsprozessor übertragen werden, können mittels entsprechender Funktionen des Betriebssystems entgegengenommen, im externen Speicher abgelegt und von dort ausgeführt

werden.

Die serielle Schnittstelle dient der Kontaktaufnahme mit anderen Recheneinheiten zwecks Austausch von anwendungsspezifischen Informationen. Wie in Abschnitt 6.3.4 dargestellt, werden hierzu die Sende- und Empfangsleitungen von zwei oder mehr Mikroprozessoren über eine so genannte *Switch-Matrix* miteinander verknüpft. Im Sinne der Funktionalität entspricht dies einer festen Verdrahtung. Maximal kann dabei eine Übertragungsrate von 312,5 kBaud im asynchronen Modus erreicht werden. Allerdings ist eine derartige Leistung nur bei ausreichender Abschirmung der entsprechenden Leitungen möglich, was bei der vorliegenden Version des Multiprozessorsystems nicht gegeben ist. Für PIChaff wurde daher mit 19,53 kBaud eine reduzierte Datenrate gewählt, die sich im Verlauf der Experimente als stabil und fehlerunanfällig herausgestellt hat.

Die Konfiguration der Switch-Matrix, das heißt das Setzen von Verknüpfungspunkten, um mehrere Recheneinheiten miteinander zu verbinden, wird durch den Kommunikationsprozessor vorgenommen. Der eigentliche Datenaustausch erfolgt schlussendlich ohne die Kooperation des Kommunikationsprozessors nur zwischen den beteiligten Recheneinheiten.

Die Aufgabe des programmierbaren Logikbausteins Atmel ATF1500 [8] besteht zum Einen in der Anbindung des externen Speichers an den PIC17C43 Mikroprozessor, zum Anderen dient er als Verbindungsglied zwischen der Recheneinheit und dem Kommunikationsprozessor, um den Austausch von Daten, Steuersignalen oder auch auszuführenden Anwendungen zu ermöglichen. Die notwendige Konfiguration des Bausteins ist identisch zu der in [88] vorgenommenen Implementierung und orientiert sich stark an den Vorgaben aus [9]. An dieser Stelle finden lediglich die wichtigsten Aspekte Erwähnung. Abbildung 6.3 zeigt dazu das entsprechende, vereinfacht dargestellte Schaltbild der Recheneinheit inklusive der Anbindung an das Trägerboard beziehungsweise an die Switch-Matrix.

Um einen Lese- oder Schreibzugriff auf den Speicher zu ermöglichen, werden die vom Mikroprozessor bereitgestellten, kombinierten Adress- und Datensignale durch einen Demultiplexer in getrennte Signale für Adressen und Daten aufgeteilt und an den externen Speicher weitergeleitet. Zum gleichen Zeitpunkt generiert das PLD aus den Adress- und Steuersignalen des PIC17C43 Mikroprozessors die notwendigen Signale für den Speicherbaustein wie etwa *Memory Chip Enable* und *Memory Write Enable*.

Der Datentransfer mit dem Kommunikationsprozessor erfolgt ebenfalls über das Atmel ATF1500. Zur Unterscheidung zwischen einem Zugriff auf den Speicher und einem Datenaustausch mit dem Kommunikationsprozessor wurden die Adressen $1000 und $1100 als *I/O-Bereich* definiert. Bei einem Schreibzugriff des Mikroprozessors auf erstgenannte Adresse werden die auf dem *Adress-/Daten-Bus* anliegenden Daten vom PLD zwischengespeichert und bei Bedarf über den *Daten-Bus* zwischen Logikbaustein und Trägerboard weitergegeben. Die Signalisierung neuer Daten geschieht hierbei mit Hilfe des Interrupt-

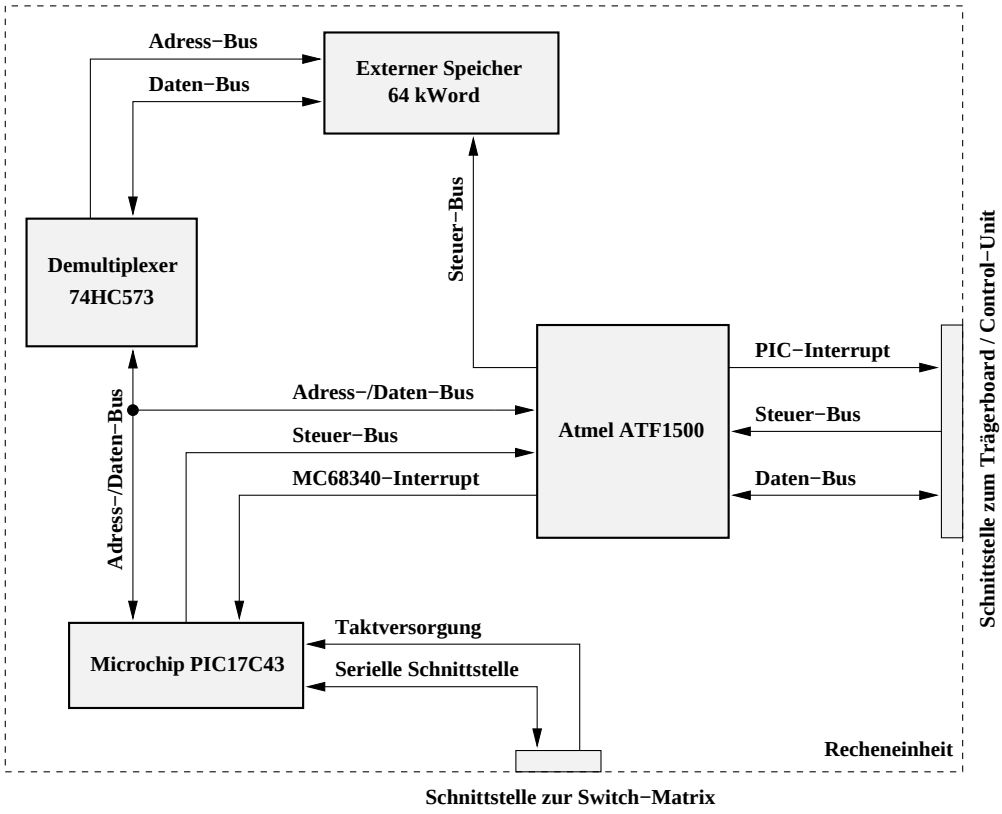


Abbildung 6.3: Schematische Darstellung der Recheneinheit

Signals *PIC-Interrupt*. Das Datenwort wird daraufhin auf dem Trägerboard von der in Abschnitt 6.3.1 eingeführten *Control-Unit* entgegengenommen und an den Kommunikationsprozessor weitergeleitet.

Analog dazu werden vom Kommunikationsprozessor per Control-Unit an das PLD gesendete Daten von diesem zwischengespeichert und dem PIC17C43 ebenfalls per Interrupt signalisiert (*MC68340-Interrupt*). Auf Seiten des Mikroprozessors geschieht die Entgegennahme der Daten vom Atmel ATF1500 durch eine Leseoperation mit Zieladresse $1100 und erfolgt über den *Adress-/Daten-Bus* zwischen Mikroprozessor und Logikbaustein.

Abschließend sei angemerkt, dass der 64 kWord große, externe Speicher nicht vollständig vom PIC17C43 Mikroprozessor genutzt werden kann. Abbildung 6.4 verdeutlicht die Problematik anhand der Partitionierung des Adressraums aus Sicht des Mikroprozessors. Der Adressbereich $0000 bis $1100 wird sowohl für das interne ROM des PIC17C43 als auch für den zuvor angedeuteten I/O-Bereich zwecks Datenaustausch mit dem Kommunikationsprozessor benötigt. Somit stehen vom externen Speicher lediglich die Zellen ab Adresse

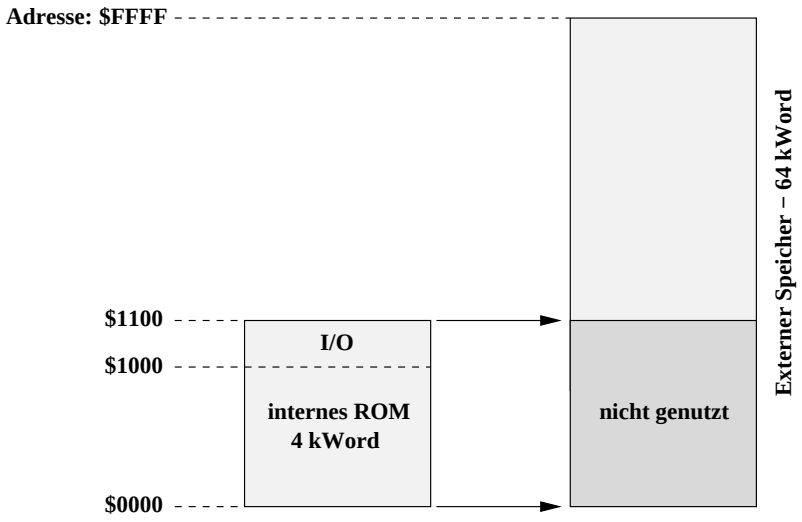


Abbildung 6.4: Partitionierung des Adressraums aus Sicht des PIC17C43 Mikroprozessors

$1101 für die von der Recheneinheit auszuführende Anwendung und die dafür benötigten Daten zur Verfügung, was einer Beschränkung auf 59,75 kWord entspricht.

6.2 Kommunikationsprozessor

Der Kommunikationsprozessor (in Kapitel 7 auch als *Communication Processor* bezeichnet) übernimmt die Aufgabe einer Steuerzentrale. Dies beinhaltet durch die Konfiguration der Switch-Matrix insbesondere den Austausch von Daten zwischen den Recheneinheiten. Des Weiteren ist das Zusammenspiel zwischen Trägerboard und angeschlossenem Rechner beziehungsweise einer entsprechenden Anwendung auf Seiten des Rechners Aufgabe des Kommunikationsprozessors. Hier steht die Weitergabe der vom Anwender implementierten Programme an die Recheneinheiten sowie das Übermitteln der bei der Abarbeitung der Programme erzielten Ergebnisse an den Computer im Vordergrund.

Abbildung 6.5 stellt den Kommunikationsprozessor dar, der aus einem *Motorola MC68340* Mikroprozessor, zwei Speichermodulen mit insgesamt 256 kByte Speicherkapazität und einem Flash-EEPROM mit 128 kByte Kapazität besteht. Im Gegensatz zum ROM-Baustein des PIC17C43 Mikroprozessors ist das Pendant des Kommunikationsprozessors, das EEPROM, mehrfach beschreibbar. Es enthält daher, wie sich in Kapitel 7 zeigen wird, neben anwendungsunabhängigen Routinen, die der Initialisierung des Multiprozessorsystems nach dem Einschalten dienen, auch Funktionen, die speziell auf PIChaff zugeschnitten sind.

Beim Motorola MC68340 Mikroprozessor [39, 40] handelt es sich um einen 32 Bit Prozessor

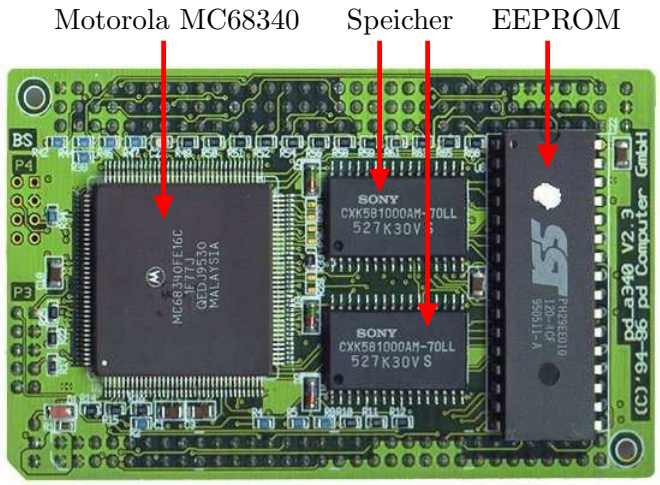


Abbildung 6.5: Kommunikationsprozessor

in CISC-Technologie, der unter anderem über folgende Eigenschaften verfügt:

- M68000 CPU-Architektur,
- 32 Bit Adress- und Datenbus,
- 32 Bit DMA Controller für schnellen Datentransfer (*Direct Memory Access*),
- 8 Daten- und 7 Adressregister,
- 16 frei konfigurierbare I/O Leitungen,
- 32 Bit Hardware-Multiplizierer,
- 7 externe Interrupt-Quellen,
- zwei kaskadierbare 16 Bit Timer,
- zwei serielle Schnittstellen mit einer maximalen Übertragungsrate von 76,8 kBaud,
- 16,78 MHz maximale Taktfrequenz,
- etwa 180 mA Stromverbrauch bei 16,78 MHz Taktfrequenz und 5 Volt Spannung.

In Abbildung 6.6 sind schematisch die Datenpfade zwischen den verschiedenen Modulen des Kommunikationsprozessors gezeigt. Durch die getrennten Adress- und Datenleitungen ist die Anbindung der Speichermodule und des EEPROMs an den Motorola MC68340 Mikroprozessor vergleichsweise einfach.

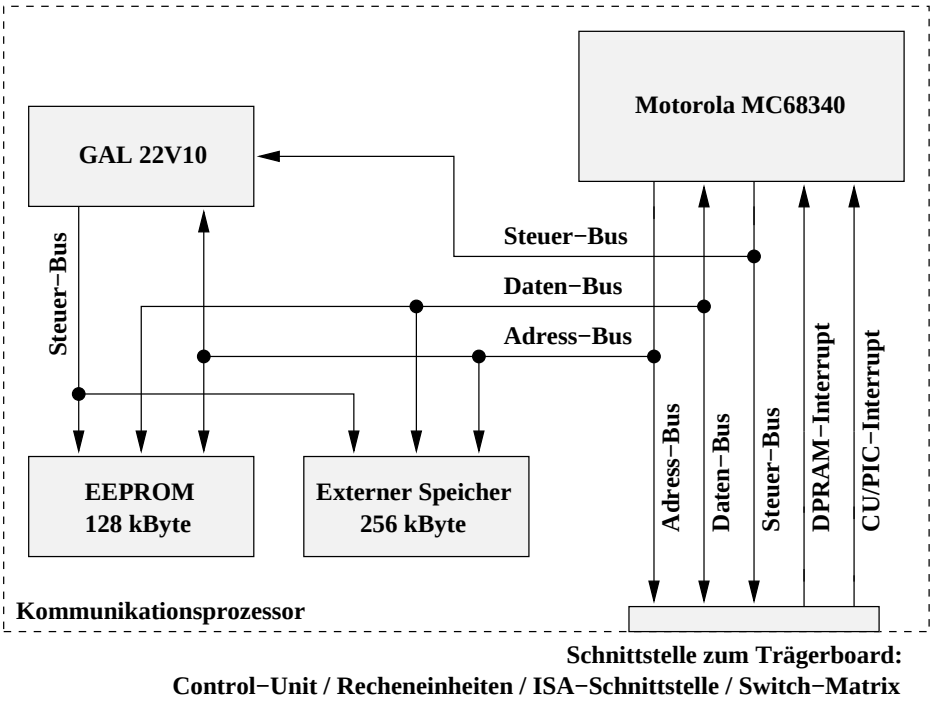


Abbildung 6.6: Schematische Darstellung des Kommunikationsprozessors

Die Aktivierung einer durch eine bestimmte Adresse spezifizierten Einheit wird mittels eines programmierbaren Bausteins vom Typ *GAL 22V10* gewährleistet. Dieser enthält die Partitionierung des Adressraums aus Sicht des Motorola Mikroprozessors und generiert für den jeweiligen Speicher die entsprechenden Steuersignale [59].

Der Kontakt zu Switch-Matrix, ISA-Schnittstelle und den Recheneinheiten erfolgt über die Schnittstelle zum Trägerboard, auf die *Steuer-*, *Adress-* und *Daten-Bus* geführt sind. Die notwendigen Steuersignale der vom Kommunikationsprozessor angesprochenen externen Komponente werden von der auf dem Trägerboard integrierten Control-Unit erzeugt. Die Existenz neuer Daten von Seiten des angeschlossenen Rechners oder der Recheneinheiten wird dem Kommunikationsprozessor mit Hilfe der Interrupt-Signale *DPRAM-Interrupt* und *CU/PIC-Interrupt* angezeigt. Das Datenwort kann daraufhin, wie in den Abschnitten 6.3.2 und 6.3.3 erläutert, vom Motorola MC68340 durch eine Leseoperation über den *Daten-Bus* entgegengenommen werden.

6.3 Trägerboard

Das in Abbildung 6.7 ohne Recheneinheiten und Kommunikationsprozessor abgebildete Trägerboard (in Kapitel 7 auch als *Carrier Board* bezeichnet) auf Basis einer ISA-

Steckkarte bildet das Rückgrat des Multiprozessorsystems und bietet die Infrastruktur, um bis zu neun Recheneinheiten effizient betreiben zu können. Mit den verschiedenen Jumpern kann der ISA-Karte ein aus Sicht des angeschlossenen Rechners eindeutiger Speicherbereich sowie Interrupt-Level zugeordnet werden. Ein Vorteil des für das Trägerboard gewählten Designs liegt in der Modularität. Die Recheneinheiten und der Kommunikationsprozessor werden lediglich aufgesteckt und können somit je nach vorgesehener Anwendung durch alternative Module ersetzt werden. Als Beispiel seien Recheneinheiten mit speziellen Sensoren oder Aktoren genannt. Einzig die Kompatibilität zu den Busprotokollen muss gewährleistet sein [14, 30].

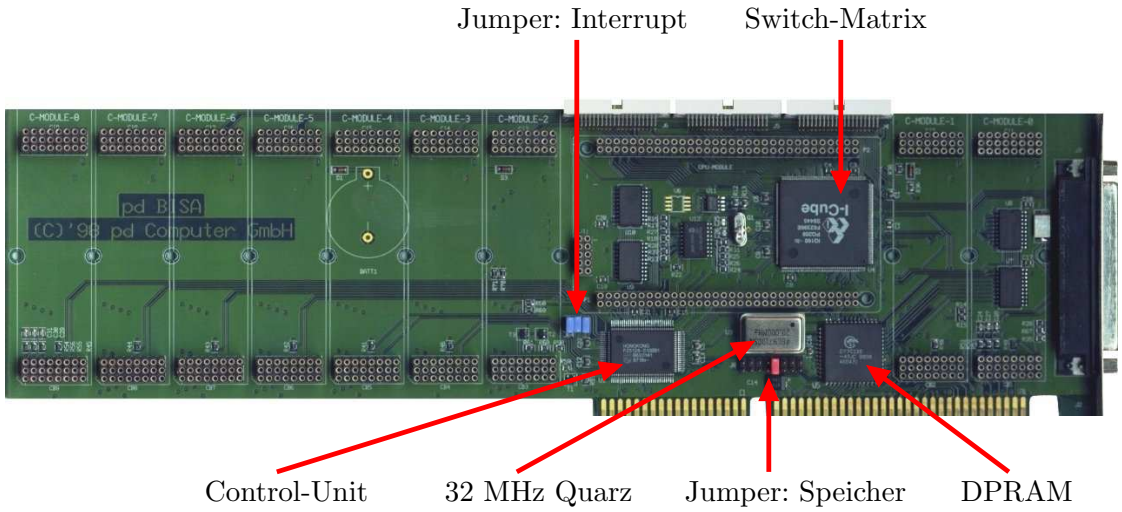


Abbildung 6.7: Trägerboard

Neben der Bereitstellung der Versorgungsspannung und eines globalen Taktsignals für nahezu alle Komponenten gehört im Zusammenspiel mit der nachfolgend beschriebenen Control-Unit die Abwicklung aller Arten von Kommunikation zu den Aufgaben des Trägerboards:

- zwischen dem Kommunikationsprozessor und den Recheneinheiten,
- zwischen dem Kommunikationsprozessor und dem angeschlossenen Rechner,
- zwischen den Recheneinheiten.

6.3.1 Control-Unit

Die *Control-Unit* ist für das Trägerboard von zentraler Bedeutung, da sie, wie Abbildung 6.8 widerspiegelt, an alle vorhandenen Bausteine angeschlossen ist und das Zusammenspiel

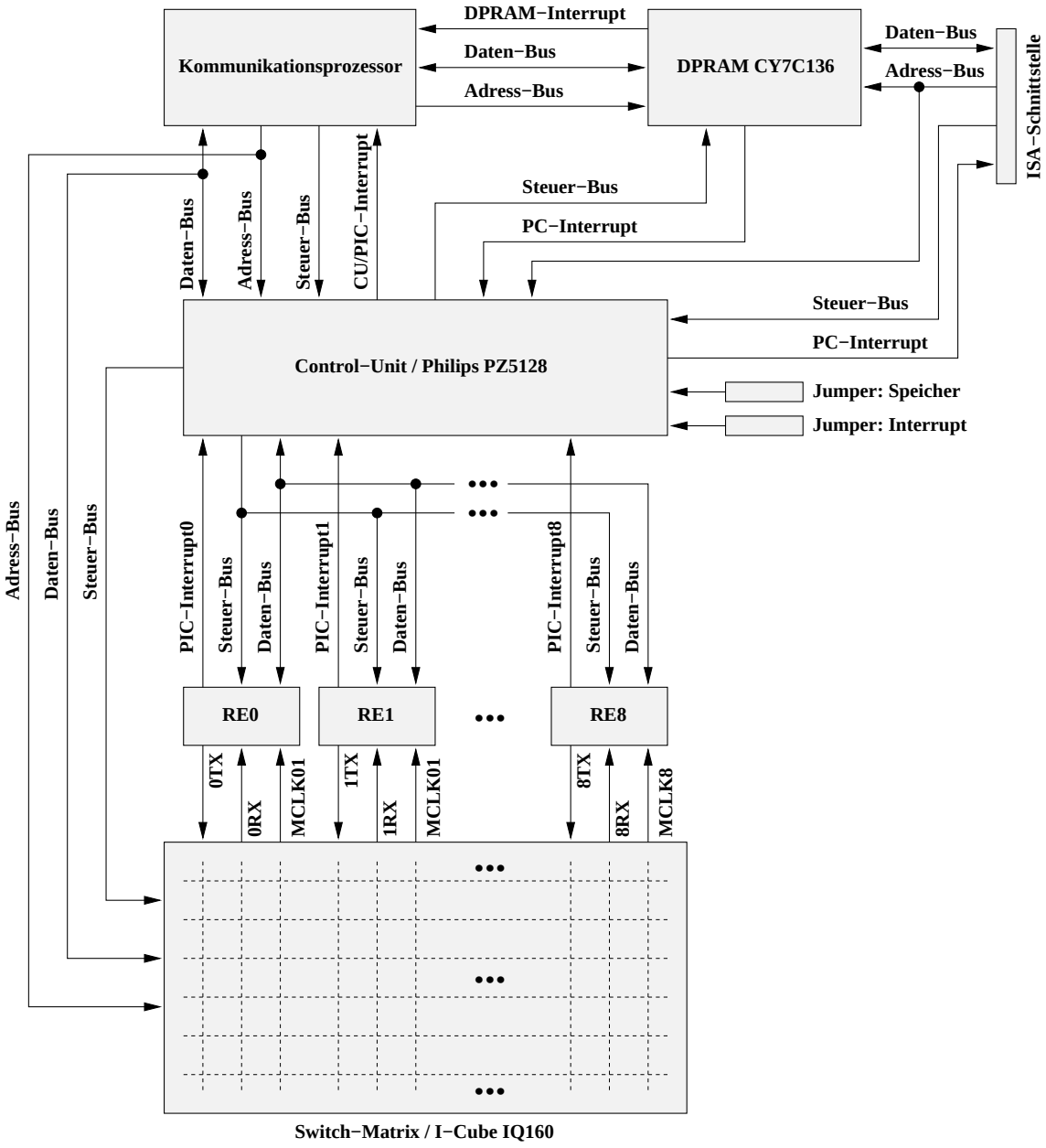


Abbildung 6.8: Einbettung der Control-Unit in das Multiprozessorsystem

zwischen diesen regelt. Verwendung findet ein programmierbares PLD vom Typ *Philips PZ5128*, das zur so genannten *CoolRunner* Familie gehört. Nach dem Verkauf dieser Serie an Xilinx im Jahr 1999 wurde die Produktion mittlerweile eingestellt, eine Übersicht über

die Nachfolgemodelle gibt [119]. Insgesamt stehen dem Anwender beim *Philips PZ5128* 128 Makrozellen zur Verfügung, die jeweils über einen I/O-Pin, ein Register und diverse interne Steuerleitungen verfügen. Mit der Möglichkeit, die realisierte (Teil-)Funktionalität der einzelnen Makrozellen über ein integriertes Bus-System koppeln zu können, lassen sich im Rahmen der Gesamtkapazität selbst komplexe Funktionen umsetzen.

Eingebettet zwischen Kommunikationsprozessor und den neun Recheneinheiten ermöglicht die Control-Unit den Datenaustausch zwischen diesen beiden Modulgruppen des Multiprozessorsystems (Abschnitt 6.3.2). Weiterhin werden die Steuersignale für das *Dual-Ported RAM* generiert, das als Verbindungsglied zwischen dem Computer, in den das Trägerboard eingesteckt ist, und dem Multiprozessorsystem beziehungsweise dem Kommunikationsprozessor fungiert (Abschnitt 6.3.3). Ebenso wird die Konfiguration der Switch-Matrix, initiiert durch einen Schreibbefehl des Motorola MC68340, durch das PLD gesteuert (Abschnitte 6.3.4 und 6.3.5).

Die Unterscheidung zwischen diesen drei Fällen geschieht durch eine eindeutige Partitionierung des Adressraums des Kommunikationsprozessors. Bei einem Lese- oder Schreibbefehl des Motorola MC68340 wird durch die Control-Unit nur die angeforderte Komponente des Multiprozessorsystems mittels spezifischer Kontrollsignale aktiviert. Die Zuordnung der Adressbereiche zu den verschiedenen Modulen des Multiprozessorsystems ist im Philips PZ5128 fest verankert und muss bei der Programmierung des Motorola MC68340 durch den Anwender eingehalten werden. Die implementierte Funktionalität der Control-Unit geht zurück auf die Arbeiten von Faller [37] und Jonas [59].

6.3.2 Datenaustausch zwischen dem Kommunikationsprozessor und den Recheneinheiten

In den Abschnitten 6.1 und 6.2 ist der Datenaustausch zwischen dem Kommunikationsprozessor und einer der insgesamt neun Recheneinheiten angedeutet worden. Das Hauptaugenmerk lag dabei auf der Fragestellung, wie die Daten vom jeweiligen Mikroprozessor auf der entsprechenden Platine zur Schnittstelle des Trägerboards gelangen. Da der Kommunikationsprozessor nicht über die Hardware-Möglichkeiten verfügt, alle neun Interrupts der Recheneinheiten mit je einem eigenen Pin zu verwalten (der MC68340 verfügt nur über 7 externe Interrupt-Leitungen), sind die beiden Schnittstellen nicht direkt miteinander verbunden. Stattdessen agiert die Control-Unit als Bindeglied.

Beim Transfer von Daten vom Kommunikationsprozessor hin zu einer der Recheneinheiten erfolgt die Spezifizierung des gewünschten PIC17C43 Mikroprozessors durch die Angabe einer ausgezeichneten Zieladresse von Seiten des Kommunikationsprozessors. Jede der neun Recheneinheiten, in Abbildung 6.8 mit RE0 bis RE8 gekennzeichnet, korrespondiert dabei zu einer eigenen, eindeutigen Adresse. Das heißt, dass anhand der auf dem *Adress-Bus* zwischen Control-Unit und Motorola MC68340 anliegenden Daten dem Logikbaustein be-

kannt ist, welche Recheneinheit mit entsprechenden Signalen auf dem Steuer-Bus aktiviert werden muss.

Initiiert eine der Recheneinheiten die Kommunikation, was der Control-Unit über die Signalleitungen *PIC-Interrupt0..8* mitgeteilt wird, so leitet diese nicht nur das entgegengenommene Datenwort, sondern auch die ID des auslösenden PIC17C43 Mikroprozessors über den *Daten-Bus* an den Kommunikationsprozessor weiter. Die ID-Nummern sind abhängig vom Slot des Trägerboards, in den die jeweilige Recheneinheit eingesteckt ist. Eine explizite Unterscheidung auf Interrupt-Ebene kann somit umgangen werden. Die Existenz neuer Daten wird dem Kommunikationsprozessor durch die Control-Unit auf einer einzigen Interrupt-Leitung (*CU/PIC-Interrupt*) signalisiert.

6.3.3 Datenaustausch mit dem angeschlossenen Rechner

Das *Dual-Ported RAM CY7C136* der Firma Cypress [26], im Folgenden kurz als *DPRAM* bezeichnet, dient als Interface zwischen Kommunikationsprozessor und angeschlossenem Rechner. Es verfügt über einen 2 kByte großen Puffer, auf den mit Hilfe zweier getrennter Schnittstellen sowohl der Kommunikationsprozessor als auch das Anwenderprogramm auf Rechnerseite zugreifen kann. Die Datenpfade sind im oberen Bereich von Abbildung 6.8 schematisch dargestellt und in Abbildung 6.9 gesondert hervorgehoben.

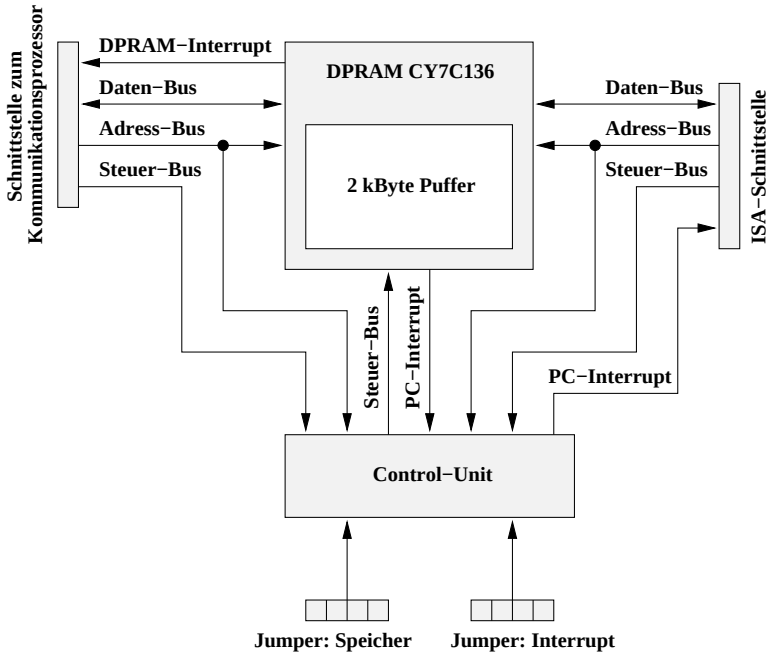


Abbildung 6.9: Anbindung der ISA-Schnittstelle an den Kommunikationsprozessor

Sowohl der ISA-Schnittstelle als auch dem Kommunikationsprozessor ist eine ausgezeichnete Speicherzelle des DPRAMs zugeordnet, die auf der jeweiligen Gegenseite automatisch einen Interrupt (*PC-Interrupt* oder *DPRAM-Interrupt*) erzeugen kann und als Indikator für neue Daten dient. Auszutauschende Daten wie die durch die Recheneinheiten abzuarbeitenden Programme oder die dabei erzielten Ergebnisse können, im Rahmen der gegebenen Puffer-Größe, blockweise übermittelt werden. Die Garantie, dass beide Seiten sich nicht gegenseitig den Inhalt der Speicherzellen überschreiben, muss dabei per Software durch eine geeignete Implementierung der Steuerroutinen gewährleistet werden.

Das DPRAM wird ebenfalls durch die Control-Unit gesteuert. Bei einem Zugriff des angeschlossenen Rechners beziehungsweise einer entsprechenden Anwendung auf das DPRAM wird von der Control-Unit anhand der von der ISA-Schnittstelle bereitgestellten Adresse ein Abgleich mit dem per Jumper eingestellten Adressbereich vorgenommen. Dieser Test ist notwendig, um zwischen Trägerboard und anderen Komponenten des Rechners wie Graphikkarte oder Festplatte unterscheiden zu können. Im positiven Fall werden vom Logikbaustein die für das DPRAM typischen Steuersignale generiert und so der Zugriff ermöglicht. Handelt es sich um einen Schreibbefehl auf die Speicherzelle mit Interrupt-Funktionalität, wird dem Kommunikationsprozessor über die Leitung *DPRAM-Interrupt* automatisch die Existenz neuer Daten gemeldet.

Analog dazu ist der Zugriff auf das DPRAM durch den Kommunikationsprozessor geregelt. Auch hier wird anhand der anliegenden Adresse von der Control-Unit entschieden, ob aus Sicht des Kommunikationsprozessors das DPRAM angesprochen werden soll. Ein gegebenenfalls an den Rechner gerichteter Interrupt (*PC-Interrupt*) zur Signalisierung eines neuen Datenblocks wird in diesem Fall allerdings nicht direkt durch das DPRAM ausgelöst, sondern an die Control-Unit weitergeleitet und dort für die ISA-Schnittstelle aufbereitet. Dieser Umweg ist notwendig, um gemäß des per Jumper auf dem Trägerboard eingestellten Interrupt-Levels nur den dazu korrespondierenden Pin der ISA-Schnittstelle zu aktivieren.

6.3.4 Datenaustausch zwischen den Recheneinheiten

Für den Datenaustausch zwischen den Recheneinheiten wird, wie bereits angeführt, die serielle Schnittstelle der PIC17C43 Mikroprozessoren genutzt. Hierzu werden Sende- und Empfangsleitung der üblicherweise zwei Partner kreuzweise miteinander verbunden. Diese Signalleitungen sind in Abbildung 6.8 sowie allen nachfolgenden Abbildungen als 0TX bis 8TX beziehungsweise 0RX bis 8RX bezeichnet. Die seriellen Schnittstellen aller neun Recheneinheiten sind auf dem Trägerboard nicht fest verdrahtet, da je nach Anwendung eine andere Ausgangs-Topologie sinnvoll sein kann. Zudem können auch während der Laufzeit unterschiedliche Verdrahtungen von Vorteil sein. Aus diesem Grund sind, wie Abbildung 6.8 zeigt, die Sende- und Empfangsleitungen mit einer so genannten *Switch-Matrix* verbunden. Bei dem hier eingesetzten Modell handelt es sich um ein dynamisch rekonfigurierbares *Field Programmable Interconnection Device* vom Typ *I-Cube IQ160* [52]. Dieses verfügt

über insgesamt 160 I/O-Pins, die intern auf eine Matrix von 160×160 Leitungen geführt sind. An jedem Kreuzungspunkt von zwei Leitungen kann dynamisch zur Laufzeit eine Verbindung geschaltet oder, falls bereits vorhanden, auch wieder gelöscht werden. Im ersten Fall sind dann die korrespondierenden I/O-Pins miteinander verbunden. Ist keine Verknüpfung hergestellt, überbrücken sich die beiden Leitungen, sie sind folglich an der Nahtstelle nicht unterbrochen. Das Setzen beziehungsweise Trennen von Kontakten erfolgt durch einen schreibenden Zugriff auf mit den Verknüpfungspunkten assoziierten Speicherzellen.

Zu den Eigenschaften des I-Cube IQ160 gehören eine maximale Taktfrequenz von 100 MHz, 10 ns Signalverzögerung zwischen den I/O-Pins und die Aktivierung einer Verbindung in 30 ns. Alle I/O-Pins können als Eingang, Ausgang oder auch bidirektional definiert werden. Sie verfügen zudem über ein Register zum Zwischenspeichern von Daten sowie über *Tri-State* Treiber zur Realisierung von Bus-Systemen. Die für diese Arbeit benötigten Einstellungen (keine Pufferung der Daten, keine Tri-State Treiber) sind im Vorfeld so gewählt worden, dass die nachfolgend beschriebene Funktionalität ermöglicht wird.

Zur Konfiguration der Switch-Matrix ist der Kommunikationsprozessor vorgesehen. Zu diesem Zweck sind *Adress-* und *Daten-Bus* des Motorola MC68340 Mikroprozessors mit den entsprechenden Anschlüssen der Switch-Matrix verbunden. Die Aktivierung des I-Cube Bausteins durch Bereitstellung entsprechender Steuersignale über den *Steuer-Bus* übernimmt wiederum die Control-Unit. Wie auch bei den anderen von der Control-Unit gesteuerten Komponenten wird anhand der Adress-Signale entschieden, ob die Switch-Matrix aktiviert werden muss, um die derzeitige Verknüpfungs-Topologie zu ändern. Somit können zu beliebigen Zeitpunkten während der Laufzeit unterschiedliche Recheneinheiten durch den Kommunikationsprozessor miteinander verbunden beziehungsweise getrennt werden. Der eigentliche Datenaustausch zwischen den PIC17C43 Mikroprozessoren erfolgt ohne die Kooperation des Motorola MC68340. Ein Beispiel für eine so genannte *One-to-One*-Verbindung, die zwei Recheneinheiten miteinander verknüpft, ist in Abbildung 6.10 gegeben. Ein wesentlicher Vorteil des gewählten Designs liegt darin, dass der Motorola MC68340 durch den Datenaustausch der PIC17C43 Mikroprozessoren nicht blockiert wird, sondern in der Zwischenzeit weitere Verknüpfungen herstellen beziehungsweise deaktivieren kann. Zusätzlich können zur gleichen Zeit auf verschiedenen Kanälen auch mehrere Paare von Recheneinheiten miteinander in Kontakt treten.

Neben *One-to-One*-Verbindungen ist auch der Datenaustausch zwischen mehr als zwei Recheneinheiten problemlos möglich, wenngleich diese Funktionalität in PIChaff nicht genutzt wird. Je nach Anwendung kann eine derartige *One-to-Many*-Verbindung aber durchaus sinnvoll sein, um, ausgehend von einer Recheneinheit, Daten gleichzeitig an mehrere Partner weiterzuleiten oder auch von dort entgegenzunehmen. Abbildung 6.11 zeigt ein Beispiel, bei dem die Recheneinheit 0 (abgekürzt durch RE0) diese Sonderstellung einnimmt und Daten an die Recheneinheiten 4, 6 und 8 weitergeben beziehungsweise von

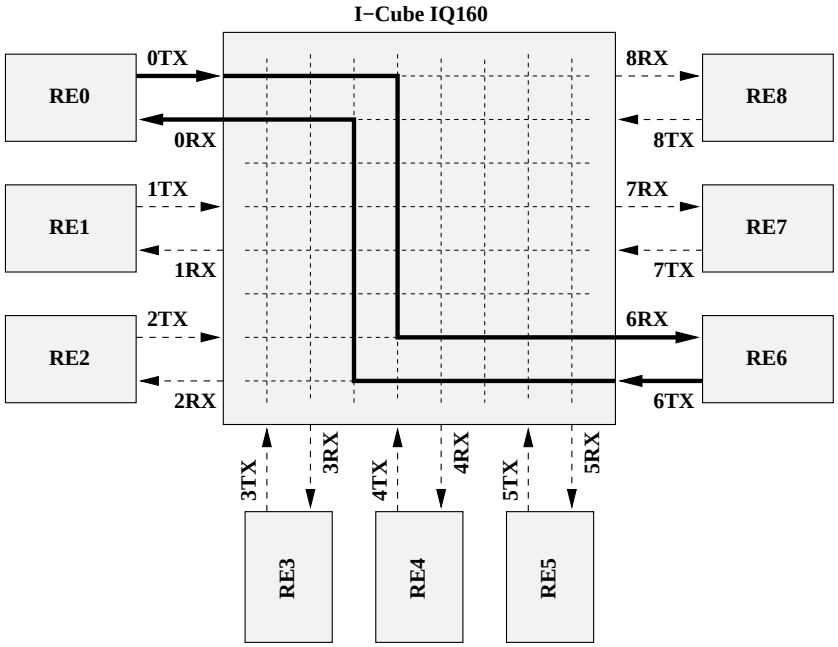


Abbildung 6.10: *One-to-One*-Verbindung

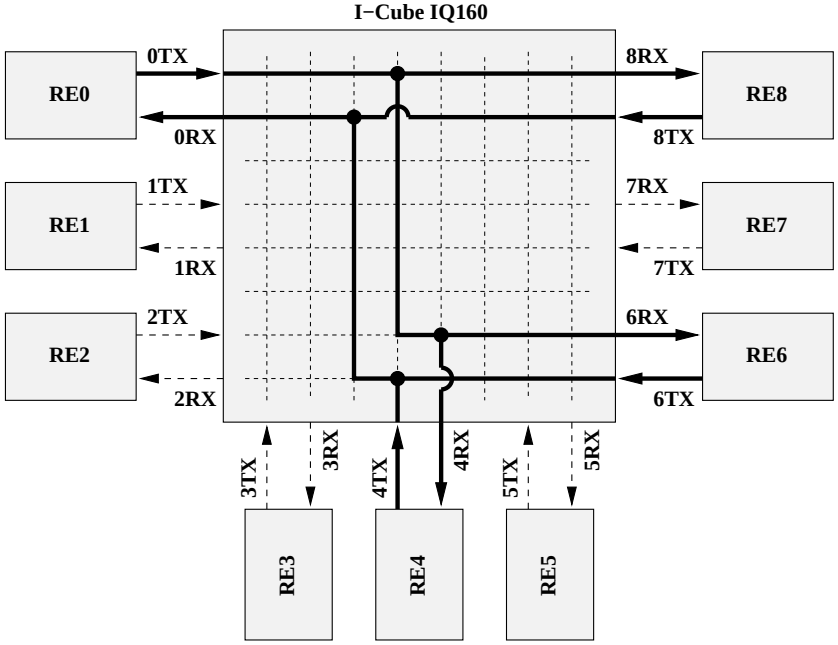


Abbildung 6.11: *One-to-Many*-Verbindung

dort empfangen kann. Dem gegenüber können die Module RE4, RE6 und RE8 nicht direkt miteinander in Kontakt treten, da die entsprechenden Sendeleitungen 4TX, 6TX und 8TX nicht kreuzweise mit den jeweiligen Empfangsleitungen 4RX, 6RX und 8RX verbunden sind.

6.3.5 Taktversorgung

Die Taktversorgung aller Logikbausteine und Mikroprozessoren des Trägerboards kann auf zweierlei Arten erfolgen: entweder global durch einen auf der ISA-Steckkarte aufgesteckten Quarz (siehe Abbildung 6.7) oder für jedes Modul getrennt durch entsprechende Bauteile auf den einzelnen Platinen. Da in der vorliegenden Version des Multiprozessorsystems lediglich die Platine des Kommunikationsprozessors über eine eigene Taktversorgung verfügt, ist folgender Ansatz realisiert worden: der Kommunikationsprozessor wird lokal mit 16,78 MHz Taktfrequenz betrieben, was der maximalen Taktrate des MC68340 Prozessors entspricht. Dem gegenüber werden die Recheneinheiten und auch die Logikbausteine des Trägerboards über einen auf der Steckkarte platzierten 32 MHz Quarz mit einem globalen Taktsignal versorgt. Die systemweite Verteilung dieses Signals regelt der Kommunikationsprozessor während der Initialisierungsphase durch eine Konfiguration der Switch-Matrix, wie sie in Abbildung 6.12 angedeutet ist. Dieses Vorgehen wird durch die Tatsache ermöglicht, dass neben den seriellen Schnittstellen der Recheneinheiten auch das Taktsignal des Quarzbausteins und die für die Taktversorgung notwendigen Pins der Recheneinheiten beziehungsweise der Logikbausteine mit I/O-Pins der Switch-Matrix verbunden sind.

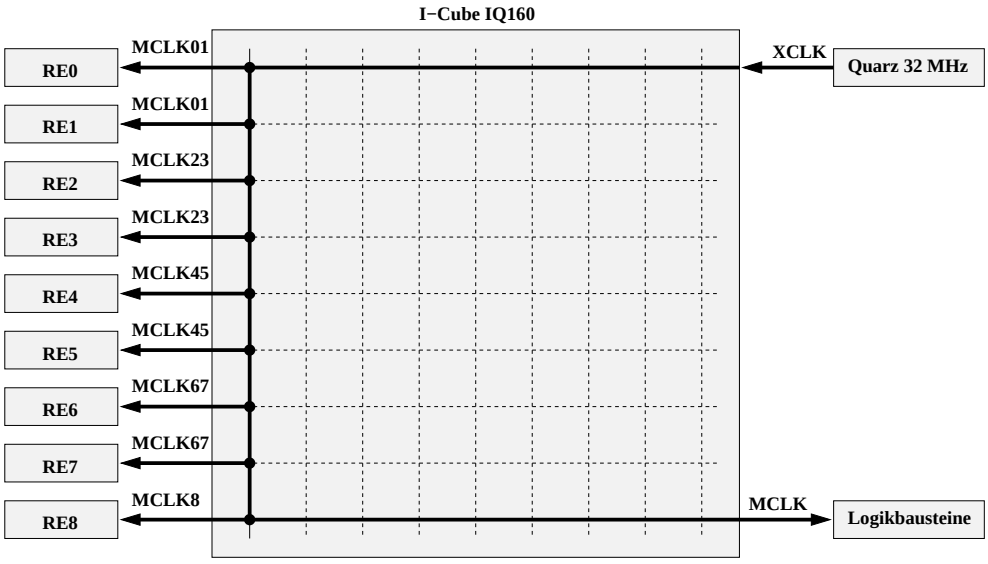


Abbildung 6.12: Taktversorgung

6.4 Abschlussbemerkung

Die Anfänge dieses Gemeinschaftsprojekts des Lehrstuhls für Rechnerarchitektur und der Firma *PD Computer – Gesellschaft für Prozess- und Datentechnik mbh* lassen sich bis etwa 1994 zurückverfolgen. Neben einem vertretbaren Preis der einzelnen Komponenten sowie einer für damalige Verhältnisse durchaus attraktiven Rechenleistung der Recheneinheiten lagen die Zielsetzungen in der Entwicklung eines flexiblen Multiprozessorsystems, das sich an verschiedene Szenarien anpassen lässt. Erreicht wurde dies durch ein modulares Design, bei dem der Kommunikationsprozessor und die Recheneinheiten auf dem Trägerboard lediglich aufgesteckt sind. Je nach Einsatzgebiet können beispielsweise Recheneinheiten mit speziellen Sensoren oder Aktoren vorteilhaft sein. Auch eine Kombination unterschiedlicher Module ist problemlos möglich.

Abschließend sei erwähnt, dass das hier vorgestellte Multiprozessorsystem nicht nur dem im nächsten Kapitel behandelten parallelen SAT-Algorithmus PIChaff als Hardware-Plattform dient, sondern auch in weiteren Anwendungsbereichen eingesetzt wurde: einerseits als Steuereinheit zur automatischen Auswertung von Stimmzetteln bei Aktionärsversammlungen mit mehreren tausend Teilnehmern [14], andererseits als Hardware-Umgebung für einen parallelen genetischen Algorithmus zum Lösen von Instanzen des *Travelling Salesman* Problems [96].

Kapitel 7

PIChaff

Den Anfang der in dieser Arbeit entwickelten parallelen SAT-Algorithmen macht PIChaff, welches speziell auf das im vorherigen Kapitel erläuterte Multiprozessorsystem zugeschnitten wurde. Wie sich zeigen wird, ist mit „PIChaff“ zwar in erster Linie die auf den Recheneinheiten ausgeführte, sequentielle SAT-Prozedur gemeint, es beinhaltet aber auch die entsprechenden Routinen auf Seiten des Kommunikationsprozessors. Die Namensgebung geht zurück auf die Microchip PIC17C43 Mikroprozessoren der Recheneinheiten und zChaff, das an einigen Stellen als Ideengeber fungierte.

Trotz der simplen Architektur der Microchip PIC17C43 Mikroprozessoren wurden für die von den Recheneinheiten ausgeführten sequentiellen SAT-Prozeduren mit einer Adaption der *Variable State Independent Decaying Sum* Entscheidungsheuristik, einem *Boolean Constraint Propagation* Mechanismus auf Basis von *Watched Literals* und einer Konflikt-Analyse gemäß des *1UIP*-Prinzips alle Eckpfeiler eines modernen SAT-Algorithmus implementiert. Weiterhin wurde eine Methode zum Löschen von Konflikt-Klauseln integriert, der aufgrund des mit 59,75 kWord sehr limitierten Speichers der Recheneinheiten eine besondere Bedeutung zukommt. Im Zusammenspiel mit dem Kommunikationsprozessor wurde für den parallelen Betriebsmodus von PIChaff, bei dem mehrere Recheneinheiten zusammen eine gestellte CNF-Formel lösen, ein *Master/Client*-Modell umgesetzt. Die Umsetzung weist dabei Ähnlichkeiten zu //Satz auf (siehe Abschnitt 5.2), wobei die Kodierung von noch unbearbeiteten Teilproblemen ebenfalls auf dem in Abschnitt 5.1 eingeführten *Guiding Path*-Konzept beruht. Abbildung 7.1 zeigt schematisch das Design von PIChaff sowie die Zuordnung zu den Komponenten des Multiprozessorsystems.

Der Kommunikationsprozessor agiert als Master und steuert den gesamten Suchprozess, indem inaktive Recheneinheiten mit bisher ungelösten Teilproblemen versorgt, der Austausch von Konflikt-Klauseln vollzogen und die von den Recheneinheiten erzielten Ergebnisse entgegengenommen werden. Die maximal neun Recheneinheiten stellen die Client-Prozesse dar und bearbeiten gemeinsam, aufgeteilt in disjunkte Bereiche, die gesamte zu lösende Probleminstanz.

Während der Initialisierungsphase wird vom Master lediglich ein Client mit dem gesamten Problem als Übergabeparameter (einem leeren Guiding Path) gestartet. Alle anderen Cli-

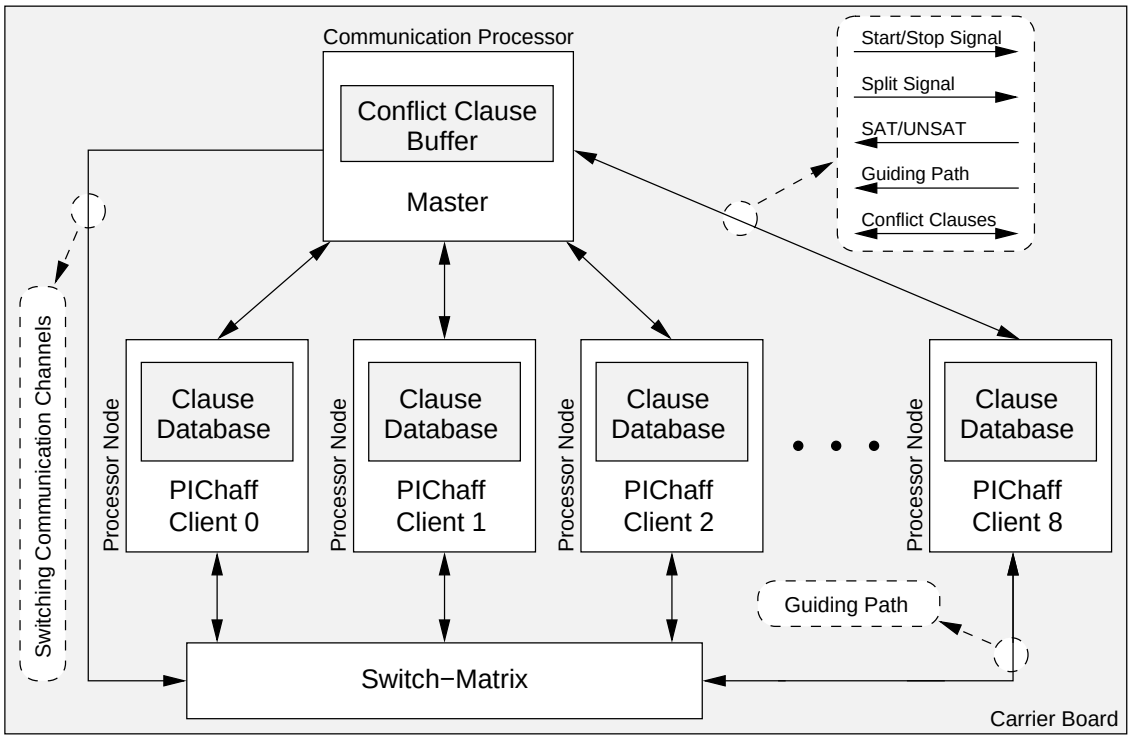


Abbildung 7.1: Design PIChaff

ents werden so aktiviert, dass sie als erstes ein Teilproblem vom Master-Prozess anfordern, was direkt im Anschluss an das Startkommando solange zu einer wiederholt durchgeführten Partitionierung des Gesamtproblems führt, bis alle Clients mit einem initialen Teilproblem versorgt sind.

Die dynamische Aufteilung des durch die CNF-Formel aufgespannten Suchraums lässt sich folgendermaßen charakterisieren: sobald ein Client-Prozess sein gestelltes Teilproblem gelöst hat, aber keine erfüllende Belegung ermitteln konnte, übermittelt dieser das Ergebnis *UNSAT* an den Master und verharrt in einem inaktiven Zustand. Ist zumindest noch ein weiterer Client aktiv, so kontaktiert der Master-Prozess einen dieser aktiven Clients und fordert ihn auf, sein aktuelles Teilproblem weiter aufzuteilen (per *Split Signal*, siehe Abbildung 7.1). Zum gleichen Zeitpunkt schaltet der Master entsprechende Kanäle der Switch-Matrix frei, so dass die seriellen Schnittstellen des aktiven und des inaktiven Clients direkt miteinander verbunden sind und der inaktive Client ein unbearbeitetes Teilproblem des aktiven Clients entgegennehmen kann.

Sind im Verlauf der Suche nach einer erfüllenden Belegung alle Clients inaktiv, so ist das gestellte Problem unerfüllbar (es verbleibt kein Teilproblem, das es zu analysieren gilt).

Daraufhin werden alle Clients durch den Master-Prozess gestoppt und in den Ausgangszustand überführt. Ebenso stoppt der Master die Clients, wenn eine erfüllende Belegung ermittelt werden konnte.

Weiterhin ist der Master-Prozess für den Austausch von Konflikt-Klauseln verantwortlich und nimmt daher alle von den Clients hergeleiteten und von diesen als „hilfreich“ für andere Clients bewerteten Konflikt-Klauseln von den Recheneinheiten entgegen. Temporär werden diese Daten im *Conflict Clause Buffer* zur Weiterverarbeitung zwischengespeichert. Die so erhaltenen Konflikt-Klauseln werden mit dem jeweils aktuell betrachteten Teilproblem der Clients verglichen und nur dann an diese weitergegeben, wenn sie nicht bereits durch den das aktuelle Teilproblem beschreibenden Guiding Path erfüllt sind. In diesem Fall wären sie nutzlos, eine Weitergabe wäre aufgrund des nicht unerheblichen Zeitaufwands für den Austausch von Konflikt-Klauseln und der begrenzten Kapazität des externen Speichers der Recheneinheiten besonders nachteilig für die Performance von PIChaff. Damit es dem Master möglich ist, die empfangenen Konflikt-Klauseln zu „filtern“, überträgt jeder Client, sobald er ein neues Teilproblem zugewiesen bekommen hat, den diesen Bereich spezifizierenden Guiding Path an den Master-Prozess. Wie sich im Verlauf des Kapitels zeigen wird, werden diese Informationen über die von den Clients jeweils bearbeiteten Teilprobleme vom Master auch genutzt, um im Fall eines inaktiven Clients denjenigen aktiven Client bestimmen zu können, der aktuell das größte verbleibende Restproblem bearbeitet und zur Aufteilung seines Bereichs aufgefordert wird.

Vor diesem Hintergrund stellt sich die Frage, warum der Austausch von Teilproblemen nicht komplett über den Master abgewickelt und auf den Einsatz der Switch-Matrix verzichtet wurde. Der Grund liegt darin, dass bei einem Verzicht auf die Switch-Matrix der Guiding Path, der einen noch nicht analysierten Bereich des Suchraums beschreibt, zunächst Literal für Literal vom „sendenden“ Client an den Master transferiert und dann vom Master wiederum Literal für Literal dem „empfangenden“ Client übergeben werden muss. Beide Schritte sind, wie in Abschnitt 7.2.3 in einem anderen Zusammenhang erläutert wird, aufgrund der zugrundeliegenden Hardware sehr zeitaufwendig, so dass bei der hier gewählten Vorgehensweise zumindest einer der beiden Datentransfers auf die ungleich schnellere Kommunikation per Switch-Matrix verlagert wurde.

In den weiteren Abschnitten des vorliegenden Kapitels wird nach einem Überblick über notwendige Vorarbeiten zunächst die sequentielle SAT-Prozedur, die auf den Recheneinheiten zum Einsatz kommt, erläutert. Die Abschnitte 7.3 und 7.4 widmen sich den Funktionen, die im Rahmen von PIChaff vom Kommunikationsprozessor und einer Anwendung auf Seiten des Rechners, die das Multiprozessorsystem mit Daten versorgt, bereitgestellt werden müssen. Im letzten Abschnitt werden die zur Evaluierung des Leistungsvermögens von PIChaff durchgeführten Experimente vorgestellt.

7.1 Vorarbeiten

Die Umsetzung des in Abbildung 7.1 dargestellten Designs war nur eine der Aufgaben, die auf dem Weg zu PIChaff bewältigt werden mussten. Wie in Abschnitt 6.2 angeführt, besteht auf Seiten des Kommunikationsprozessors die Möglichkeit, alle für PIChaff relevanten Routinen des Master-Prozesses im EEPROM des Kommunikationsprozessors abzulegen und gegebenenfalls zu aktualisieren, da dieser Speicher mehrfach beschreibbar ist. Auf Seiten der Recheneinheiten ist dieses Vorgehen nicht möglich, da das interne 4 kWord große ROM der PIC17C43 Mikroprozessoren nur einmal beschreibbar ist und danach nur noch Lesezugriffe erlaubt. Jede Änderung an einer einmal dort hinterlegten sequentiellen SAT-Prozedur hätte den Austausch der Mikroprozessoren zur Folge, was nicht nur aus finanziellen Gründen kein gangbarer Weg ist. Somit bleibt nur die Möglichkeit, die sequentiellen SAT-Prozeduren von PIChaff, die am Rechner mit Hilfe von *Microchip MPLAB* [85] implementiert und in ein PIC17C43-konformes Format übersetzt wurden, während der Startphase in den externen Speicher der Recheneinheiten zu übertragen und von dort auszuführen. Analog gestaltet sich die Situation für die zu lösende CNF-Formel, die ebenfalls vor dem eigentlichen Start von PIChaff in den 64 kWord großen RAM-Speicher der Recheneinheiten übertragen wird. Konsequenterweise sollten im ROM der PIC17C43 Mikroprozessoren dann nur diejenigen Funktionen hinterlegt werden, die von einer Anwendung wie PIChaff unabhängig sind und für die Initialisierung nach dem Einschalten oder für die Übertragung und die Ausführung von Programmen auf den Recheneinheiten benötigt werden.

Im ersten Arbeitsschritt wurde daher sowohl für die Microchip Mikroprozessoren der Recheneinheiten als auch den Motorola Mikroprozessor des Kommunikationsprozessors jeweils ein „Betriebssystem“ entwickelt, das dem typischen Vorgehen aus Entwickeln einer Anwendung, Übertragen an die Recheneinheiten des Multiprozessorsystems und Starten der Anwendung Rechnung trägt, in dem es die dafür benötigten Routinen bereitstellt. Beide Betriebssysteme bewirken, dass nach dem Einschalten der Versorgungsspannung und einer kurzen Initialisierungsphase das Multiprozessorsystem *betriebsbereit* ist und auf Kommandos durch den Benutzer wartet. Im Einzelnen verfügt das Betriebssystem der Recheneinheiten über folgende Funktionalität:

- Der mit der Leitung *MC68340-Interrupt* (siehe Abbildung 6.3) verbundene Pin des Microchip PIC17C43 Mikroprozessors wird als Interrupt-Signal konfiguriert und mit einer entsprechenden Interrupt-Routine verknüpft. Innerhalb der Interrupt-Routine wird beim Eintreten eines Interrupts (ausgelöst durch den Atmel ATF1500 Logikbaustein) ein spezielles Flag gesetzt, so dass es einer Zielanwendung wie PIChaff möglich ist, auf Änderungen dieses Flags zu reagieren und über den Logikbaustein mit dem Kommunikationsprozessor in Kontakt zu treten.
- Einer der 16 Bit Timer wird als so genannter *Overflow Timer* konfiguriert, das heißt, dass nach dem Erreichen des maximalen Zählerwerts ($2^{16} - 1 = 65535$) mit dem Zählerstand 0 fortgefahren wird. Jeder Überlauf löst einen internen Interrupt aus, der

dazu verwendet werden kann, die seit dem Aktivieren des Zählers verstrichene Zeit (in Abhängigkeit von der Taktfrequenz des Mikroprozessors) zu messen. Der Timer wird während der Initialisierungsphase lediglich konfiguriert, nicht aber aktiviert. Das Starten und Stoppen des Timers erfolgt bei Bedarf aus der Zielanwendung heraus, so dass neben der Gesamtlaufzeit eines Programms auch die Dauer einzelner Routinen exakt ermittelt werden kann.

– Aufbauend auf der zuvor durchgeführten Konfiguration und Aktivierung der benötigten Interrupt-Funktionalität stehen zwei Routinen bereit, die auf ein entsprechendes Kommando des Kommunikationsprozessors hin von diesem Programme und Daten interruptgestützt entgegennehmen, im externen Speicher der Recheneinheiten ablegen und im Fall übertragener Zielanwendungen dann ausführen.

Die Konfiguration und Aktivierung der seriellen Schnittstelle der PIC17C43 Mikroprozessoren, die im PIChaff-Szenario von den Recheneinheiten zum Austausch von Teilproblemen genutzt wird, ist bewusst kein Teil des Betriebssystems, da die entsprechenden Pins je nach Anwendung auch für andere Aufgaben als „klassische“ I/O-Pins eingesetzt werden können. Die Festlegung der benötigten Parameter der seriellen Schnittstelle erfolgt daher zu Beginn der von den Recheneinheiten ausgeführten sequentiellen SAT-Prozedur.

Angepasst an die Funktionalität der Recheneinheiten und die gegebene Architektur des Multiprozessorsystems bietet das Betriebssystem des Kommunikationsprozessors folgende Basisroutinen:

– Die mit den beiden in Abbildung 6.8 als *CU/PIC-Interrupt* beziehungsweise *DPRAM-Interrupt* bezeichneten Leitungen verbundenen Pins des Motorola MC68340 Mikroprozessors werden als Interrupt-Signale konfiguriert und ebenfalls mit einer *Interrupt-Service-Routine* verknüpft. Sie dienen dazu, dem Kommunikationsprozessor entweder von Seiten der Recheneinheiten (CU/PIC-Interrupt) oder von Seiten des Rechners (DPRAM-Interrupt) die Existenz neuer Daten zu signalisieren.

– Gemäß Abschnitt 6.3.5 wird die Switch-Matrix so konfiguriert, dass sowohl die Recheneinheiten als auch die Control-Unit des Trägerboards mit dem Taktsignal des externen Quarzes (32 MHz) versorgt werden. Der Motorola MC68340 Prozessor wird durch einen auf der Platine des Kommunikationsprozessors vorhandenen Quarz lokal mit einem Taktsignal von 16,78 MHz versorgt.

– Weiterhin stehen Funktionen bereit, mit denen Daten und PIC17C43-konforme Programme von einer entsprechenden Anwendung auf Seiten des Rechners über die ISA-Schnittstelle empfangen und an die Recheneinheiten weitergeleitet werden können, die das jeweilige Programm daraufhin abarbeiten.

Beide Betriebssysteme zusammen bewirken, dass dem Anwender nach dem Einschalten des Multiprozessorsystems eine einsatzbereite und empfangsbereite Hardware-Plattform

zur Verfügung steht. Die einzelnen zuvor skizzierten Routinen sind so allgemein gehalten, dass sie nicht nur für PIChaff, sondern ohne Änderung auch für andere Anwendungen genutzt werden können.

7.2 SAT-Prozeduren der Recheneinheiten

In diesem Abschnitt wird die Implementierung der sequentiellen SAT-Prozedur, die auf den Recheneinheiten ausgeführt wird, erläutert. Bevor im Detail auf die einzelnen Routinen eingegangen wird, sei an dieser Stelle darauf hingewiesen, dass die Programmierung vollständig in Maschinensprache vorgenommen wurde. Im Bereich der Mikroprozessor-Programmierung ist es allgemein akzeptiert, dass mit einem manuell optimierten Assembler-Code im Vergleich zu dem von einem C-Compiler aus einem C-Programm erzeugten Code eine zum Teil wesentlich kompaktere und somit auch schnellere Realisierung der gewünschten Funktionalität erzielt werden kann. Dahingehende Tests im Vorfeld haben dies auch im Fall von PIChaff bestätigt. Der Hauptgrund für den Einsatz von Maschinensprache ist allerdings weniger in der Beschleunigung des Programms als vielmehr in einer Verringerung des Speicherbedarfs der SAT-Prozedur zu sehen. Neben der Klauselmenge und allen weiteren Statusvariablen wird diese ebenfalls im externen Speicher der Recheneinheiten abgelegt, dessen Kapazität mit 64 kWord sehr gering ausfällt. Vereinfacht ausgedrückt bedeutet in diesem Zusammenhang ein kleineres Programm einen größeren „Spielraum“ für SAT-spezifische Daten.

7.2.1 Entscheidungsheuristik

Die bei den sequentiellen SAT-Prozeduren zum Einsatz kommende Entscheidungsheuristik basiert auf der *Variable State Independent Decaying Sum* Strategie (VSIDS). Für jede Variable der gegebenen CNF-Formel werden in PIChaff zwei Zähler mitgeführt, die angeben, wie *aktiv* die Variable als positives und negatives Literal ist. Als Decision Variable wird stets diejenige unbelegte Variable ausgewählt, deren Zählerstand (Aktivität) als positives oder negatives Literal maximal unter allen freien Variablen ist. Der höhere Zählerstand der beiden Polaritäten der Decision Variable gibt dabei den gewählten Wahrheitswert vor. Die ansonsten gegenüber der originalen VSIDS-Version von zChaff vorgenommenen Änderungen wurden bereits in Abschnitt 4.3 angedeutet, der Vollständigkeit halber sind sie an dieser Stelle nochmals kurz zusammengefasst:

– Für die Wahl der nächsten Decision Variable wird eine nach absteigenden Aktivitäten sortierte Liste der Variablen mitgeführt, in der im Gegensatz zu den Ausführungen in Abschnitt 4.3 nicht nur die derzeit freien, sondern alle Variablen enthalten sind. Das hat den Vorteil, dass Variablen, für die im Verlauf einer Backtrack-Operation eine Zuweisung rückgängig gemacht wurde, nicht wieder in die Liste eingetragen werden müssen. Andererseits handelt es sich dann beim ersten Element nicht zwangsläufig um eine freie Variable, was bedeutet, dass zur Bestimmung der aktuell aktivsten

Variablen, die zugleich noch nicht belegt ist, gegebenenfalls ausgehend vom Anfang der Liste mehrere Einträge der Reihe nach betrachtet werden müssen.

– Eine „Normalisierung“ der Aktivitäten der Variablen wird in PIChaff nach jeweils 256 Konflikten vorgenommen, wobei alle Zählerstände per *Rechts-Shift* halbiert werden.

– Neben den Literalen der Konflikt-Klauseln werden zudem die Zählerstände derjenigen Literale um eins inkrementiert, die zwar nicht Teil einer Konflikt-Klausel sind, aber an einem Resolutions-Schritt während der Herleitung einer Konflikt-Klausel beteiligt waren.

7.2.2 Boolean Constraint Propagation

Der in PIChaff gewählte Mechanismus zur Durchführung der *Boolean Constraint Propagation* basiert auf dem in Abschnitt 4.4 erläuterten Konzept der Watched Literals. Analog zu beispielsweise MiniSat2 werden die aktuell die Klausel beobachtenden Literale jeweils an die ersten beiden Positionen der Klausel gestellt, so dass bei einem falsch belegten Watched Literal zunächst effizient geprüft werden kann, ob das zweite Watched Literal die Klausel erfüllt. Ist dies der Fall, kann auf die Suche nach einer Alternative zum bisherigen Watched Literal verzichtet werden, denn solange das zweite Watched Literal die Klausel erfüllt, kann weder eine Implikation noch ein Konflikt eintreten.

Weiterhin wird in PIChaff auch die in Abschnitt 4.4 als *Early Conflict Detection Based BCP* bezeichnete Technik umgesetzt: jede gefundene Implikation zieht sofort die entsprechende Variablenbelegung nach sich, während der Test, welche Konsequenzen sich aus dieser Zuweisung ergeben, eventuell erst zu einem späteren Zeitpunkt durchgeführt wird. Das hat zum Einen den Vorteil, dass sich Konflikte innerhalb der *Implication Queue* im Vergleich zur „klassischen“ Vorgehensweise wesentlich früher feststellen lassen. Zum Anderen lassen sich dadurch effektiver Nachfolger für falsch belegte Watched Literals bestimmen, da ein Literal, von dem bereits bekannt ist, dass eine Implikation mit dem komplementären Wahrheitswert existiert, nicht mehr als potenzielles Watched Literal in Betracht gezogen wird.

Auch werden bei der in PIChaff eingesetzten BCP-Routine für jede Variable zwei Listen mit Verweisen auf Klauseln mitgeführt, die angeben, in welchen Klauseln die jeweilige Variable in Form eines negativen oder positiven Literals aktuell eines der beiden Watched Literals darstellt (im Folgenden als *WL-Listen* bezeichnet). Wird im Verlauf des Suchprozesses beispielsweise die Zuweisung $x_i = 1$ getätigt, so kann anhand der WL-Liste für den dazu komplementären Wahrheitswert (die Liste mit Verweisen auf Klauseln, in denen $\neg x_i$ eines der beiden Watched Literals darstellt) effizient die Menge der Klauseln bestimmt werden, bei denen eine Implikation oder ein Konflikt überhaupt nur möglich sein kann.

In diesem Zusammenhang gilt es zu beachten, dass mit jeder zur Klauselmenge hinzugefügten Klausel auch der Gesamtspeicherbedarf aller Listen von beobachteten Klauseln um zwei zusätzliche Einträge ansteigt, da pro Klausel zwei Watched Literals bestimmt werden. Das hat zur Folge, dass im Verlauf der Ausführung eines SAT-Algorithmus oftmals die Situation eintritt, das der initial für die Listen von beobachteten Klauseln vorgesehene Speicher nicht mehr ausreicht, um alle Verweise auf Klauseln ablegen zu können. Bei einem für einen „klassischen“ Rechner in einer Hochsprache wie C/C++ entwickelten SAT-Verfahren wird diese Situation im Allgemeinen dadurch gelöst, dass dynamisch zusätzlicher Speicher angefordert und reserviert wird. Durch eine ausgeklügelte Speicherverwaltung suggeriert das jeweilige Betriebssystem dem Anwender dabei weiterhin einen zusammenhängenden Speicherbereich, auch wenn sich dieser im Hauptspeicher über mehrere voneinander getrennte Bereiche erstreckt.

Bezogen auf PIChaff wäre die Nachbildung einer derartigen Speicherverwaltung für die Recheneinheiten nur mit einem unverhältnismäßig hohen Programmieraufwand möglich gewesen. Neben einem ungleich komplexeren und damit auch längeren Assembler-Programm hätte dies zudem Querverweise zwischen physikalisch getrennten, aber logisch zusammenhängenden Datenblöcken erfordert. Beides hätte die Anzahl der Speicherzellen, die für die problemspezifischen Daten wie Decision Stack, Klauselmenge und ähnliches benötigt werden, reduziert und somit die Menge der CNF-Formeln, die sich aufgrund ihrer Größe überhaupt mit PIChaff bearbeiten lassen, stark eingeschränkt. Daher ist bezüglich der Listen der von den Variablen beobachteten Klauseln folgender Weg eingeschlagen worden: vor dem eigentlichen Start von PIChaff wird innerhalb des externen Speichers ein zusammenhängender Bereich fester Größe für die WL-Listen reserviert. Dieser wird initialisiert, indem für alle Variablen die jeweiligen Listen von beobachteten Klauseln der originalen CNF-Formel hintereinander abgelegt werden (pro Variable sind dies zwei Listen mit Verweisen, eine für jede Polarität). Im Verlauf der Suche nach einer erfüllenden Belegung werden zusätzliche Einträge, die einer bestimmten Liste hinzugefügt werden müssen, dadurch ermöglicht, dass gegebenenfalls die vorangehenden oder nachfolgenden Datenblöcke zur Schaffung von freien Speicherzellen in Richtung Anfang beziehungsweise Ende des Speicherblocks verschoben werden. Erst wenn der gesamte vorgesehene Speicherbereich belegt ist, das heißt, das Maximum an Verweisen auf Klauseln erreicht ist, wird über das Löschen von Konflikt-Klauseln (siehe Abschnitt 7.2.4) versucht, wieder Speicherraum zur Verfügung zu stellen.

7.2.3 Konflikt-Analyse, Non-Chronological Backtracking und Weitergabe von Konflikt-Klauseln

Die im Fall einer widersprüchlichen Variablenbelegung von den Clients durchgeführte Konflikt-Analyse operiert gemäß des *1UIP*-Prinzips (siehe auch Abschnitt 4.5). Der Backtrack Level wird anhand der den Konflikt charakterisierenden Konflikt-Klausel derart bestimmt, dass die Klausel nach der vollzogenen Backtrack-Operation direkt eine Implikation

auslöst (eine *Asserting Clause* darstellt). Das Grundgerüst der in PIChaff zum Einsatz kommenden Konflikt-Analyse unterscheidet sich daher konzeptuell nicht von den entsprechenden Routinen anderer Verfahren.

Für den Austausch von Konflikt-Klauseln zwischen den Clients wurde jedoch eine zusätzliche Funktionalität eingeführt. Die Clients testen zum Abschluss der Konflikt-Analyse jeweils, ob die soeben hergeleitete Konflikt-Klausel auch für andere Clients von Interesse sein könnte, was anhand der Länge der Klausel entschieden wird. Liegt die Anzahl der Literale der Konflikt-Klausel unterhalb einer vorgegebenen Grenze, besteht die Chance, dass die Klausel so „allgemein gültig“ ist, dass sie auch bei anderen Clients zu einer erheblichen Reduktion des jeweiligen Teils des Suchraums führt. In diesen Fällen sendet der Client eine entsprechende Nachricht an den Master-Prozess und übermittelt daran anschließend die Konflikt-Klausel Literal für Literal.

Wie im vorherigen Kapitel dargestellt, erfolgt die Kommunikation zwischen einer Recheneinheit (Client) und dem Kommunikationsprozessor (Master) sowohl mit Hilfe der Control-Unit, die auf Seiten des Trägerboards für die Abwicklung der Kommunikation verantwortlich ist, als auch mit Hilfe des Atmel ATF1500 Logikbausteins, der als Mittler zwischen PIC17C43 Mikroprozessor und „Außenwelt“ dient. Von Nachteil für einen effizienten Datentransfer ist allerdings der lediglich 8 Bit breite Datenbus des Microchip PIC17C43 Mikroprozessors, der es erfordert, dass die Übertragung der Literale einer Konflikt-Klausel, die als 16 Bit Integer-Variablen deklariert sind, in zwei Schritten erfolgen muss. Das hat zur Folge, dass eine Klausel bestehend aus beispielsweise drei Literalen sowie einer abschließenden 0 zum Signalisieren des Klauselendes nicht in insgesamt vier Schritten an den Kommunikationsprozessor übertragen werden kann, sondern acht Schritte benötigt. Jeder übertragene 8 Bit Wert muss zudem noch vom Kommunikationsprozessor per *Acknowledge Signal* bestätigt werden, um der Recheneinheit die Bereitschaft zum Empfang des nächsten Datenworts zu signalisieren. Die Ausführungen verdeutlichen, dass der Austausch von Konflikt-Klauseln, bedingt durch die gegebene Hardware-Architektur, eine zeitaufwendige Prozedur darstellt, was sich auch im Rahmen der in Abschnitt 7.5 diskutierten experimentellen Ergebnisse bestätigen wird.

Es ist folglich von höchster Wichtigkeit, dass vom Master-Prozess keine Klauseln weitergeleitet werden, die nutzlos sind, da sie zum Beispiel bereits durch Zuweisungen auf Decision Level 0 des entsprechenden, die Klausel entgegennehmenden Clients erfüllt sind. Auf Decision Level 0 befinden sich einerseits die durch den Guiding Path spezifizierten Zuweisungen und andererseits alle Zuweisungen, die sich aus Unit Clauses beziehungsweise den daraus resultierenden Implikationen ergeben. Beide Gruppen von Variablenzuweisungen sind für das aktuelle Teilproblem fest vorgegeben und notwendig, um die Chance auf eine erfüllende Belegung zu wahren. Jede übertragene Konflikt-Klausel, die bereits durch eine Zuweisung auf Decision Level 0 erfüllt ist, hat daher im aktuellen Teilproblem keinerlei Bedeutung, da sie nie eine Implikation oder einen Konflikt auslösen kann.

Der Master nimmt aus diesem Grund stets einen Abgleich zwischen einer empfangenen Konflikt-Klausel und dem Guiding Path des Clients vor, an den die Klausel gegebenenfalls weitergeleitet werden soll. Stellt sich dabei heraus, dass die Klausel bereits durch die das Teilproblem des entsprechenden Clients spezifizierenden Zuweisungen erfüllt ist, wird auf eine Weitergabe verzichtet (siehe auch Abschnitt 7.3.2).

7.2.4 Löschen von Konflikt-Klauseln

Wie in Abschnitt 4.6 erklärt wurde, hat das Löschen von Konflikt-Klauseln bei modernen SAT-Algorithmen im Wesentlichen das Ziel, die Durchführung der Boolean Constraint Propagation zu beschleunigen. Ohne das Entfernen von gelernten Konflikt-Klauseln steigt die Gesamtzahl der Klauseln während des Suchprozesses kontinuierlich an, so dass auch tendenziell immer mehr Klauseln dahingehend analysiert werden müssen, ob sie Implikationen oder Konflikte auslösen. Werden dagegen in periodischen Abständen insbesondere die für die Suche nach einer erfüllenden Belegung als wenig hilfreich eingestuften Konflikt-Klauseln wieder entfernt, kann die BCP-Routine beschleunigt werden.

Im Rahmen von PIChaff kommt dem Löschen von Konflikt-Klauseln vor dem Hintergrund des in seiner Kapazität sehr limitierten externen Speichers der Recheneinheiten aber aus einem anderen Grund eine wichtige Rolle zu. Von den insgesamt 64 kWord stehen einer Anwendung wie PIChaff nur 59,75 kWord zur Verfügung, wobei in dem verbleibenden Speicher sowohl das auszuführende Programm als auch alle problemspezifischen Daten wie etwa die Variablenbelegung, der Decision Stack und die WL-Listen abgelegt sind. Das bedeutet, dass für die eigentliche Klauselmenge nur ein Bruchteil des gesamten Speichers zur Verfügung steht, was während der Ausführung der sequentiellen SAT-Prozeduren in der Regel mehrfach dazu führt, dass keine freien Speicherzellen mehr vorhanden sind, um neue Konflikt-Klauseln abspeichern zu können. Es müssen folglich bereits gelernte Klauseln wieder gelöscht werden, um den Suchprozess fortführen zu können.

Ebenso kann eine derartige Operation notwendig werden, wenn der Speicher für die WL-Listen vollständig erschöpft ist. Wie in Abschnitt 7.2.2 festgelegt wurde, wird vor dem Start der sequentiellen SAT-Prozeduren innerhalb des externen Speichers für die WL-Listen ein zusammenhängender Bereich fester Größe reserviert. Somit kann die Situation eintreten, dass zwar eine neue Klausel noch als solche gespeichert werden kann, aber die Watched Literals nicht mehr in den beiden entsprechenden Listen vermerkt werden können, so dass auch in diesem Szenario ein Löschen von Konflikt-Klauseln unabdingbar ist, um PIChaff weiter fortzuführen.

Um mit jedem Aufruf der für das Löschen von Konflikt-Klauseln zuständigen Funktion möglichst viele Speicherzellen wieder freizugeben, wurde für PIChaff folgendes Vorgehen gewählt: als erstes wird jede Klausel, die aktuell keine Implikation erzwingt und aus die-

sem Grund für eine gegebenenfalls anstehende Konflikt-Analyse nicht relevant ist, aus der Klauselmenge entfernt. In einem zweiten Schritt werden dann die entstandenen „Leerstellen“ innerhalb der Klauselmenge durch ein Aufrücken der jeweils nachfolgend abgelegten Konflikt-Klauseln gefüllt.

Zusammengefasst hat das Entfernen von Konflikt-Klauseln bei PIChaff nicht primär das Ziel, die BCP-Funktion zu entlasten, sondern dient vielmehr dazu, den Suchprozess überhaupt erfolgreich beenden zu können.

7.2.5 Austausch von Teilproblemen mit anderen Recheneinheiten

Für den Austausch von Teilproblemen zwischen den Clients wurden zwei Routinen entwickelt, mit denen die Clients zum Einen ihren aktuellen Bereich des Gesamtproblems aufteilen und einen Teil davon weitergeben und zum Anderen neue Teilprobleme empfangen können. Wie bereits mehrfach angedeutet und aus Abbildung 7.1 ersichtlich, erfolgt der Datentransfer in diesem Fall direkt zwischen den Clients mit Hilfe entsprechend freigeschalteter Kommunikationskanäle der Switch-Matrix.

Der Ablauf beider Szenarien lässt sich wie folgt skizzieren: sobald ein aktiver Client vom Master-Prozess das Signal zur Aufteilung seines Teilproblems erhält, wechselt er in den entsprechenden Modus, aktiviert seine serielle Schnittstelle und wartet auf das Start-Signal zur Datenübertragung durch den Master. Dieser konfiguriert in der Zwischenzeit freie Kommunikationskanäle der Switch-Matrix derart, dass die seriellen Schnittstellen von sendendem und empfangendem Client direkt miteinander verbunden sind (entspricht der in Abbildung 6.10 dargestellten *One-to-One*-Verbindung). Danach wird beiden Clients durch den Master das Start-Signal zur Datenübertragung gegeben, so dass der einen Bereich seines aktuellen Teilproblems abgebende Client den entsprechenden Guiding Path per serieller Schnittstelle und Switch-Matrix weitergibt. Dazu durchläuft der Client seinen eigenen Decision Stack beginnend bei der ersten Zuweisung auf Decision Level 0 und überträgt alle Zuweisungen bis zum Erreichen der ersten Decision Variable an den empfangenden Client. Als letzter Datensatz wird dann die Decision Variable mit dem komplementären Wahrheitswert übergeben. Zum Abschluss modifiziert der sendende Client seinen Decision Stack analog zu Abschnitt 5.1 dahingehend, dass alle Zuweisungen auf Decision Level 1 Teil des Decision Levels 0 werden und Zuweisungen auf höheren Ebenen jeweils auf den direkt vorangehenden Decision Level übertragen werden. Dies gewährleistet, dass es dem Client nicht erlaubt ist, in den durch das gerade abgegebene Teilproblem aufgespannten Teil des Suchraums zu wechseln und somit kein Bereich mehrfach analysiert wird. Weiterhin wird bei einer neuerlichen Anfrage verhindert, dass das identische Teilproblem ein zweites Mal abgegeben wird.

Der ein Teilproblem empfangende Client hingegen speichert alle Einträge des erhaltenen Guiding Path als Zuweisungen auf Decision Level 0 und prüft im Anschluss mittels der BCP-Funktion, welche Konsequenzen (Implikationen und Konflikte) sich vor dem Hin-

tergrund der eigenen Klauselmenge daraus ergeben. Danach sendet der Client das soeben erhaltene Teilproblem an den Master-Prozess, um diesem einerseits das „Filtern“ von Konflikt-Klauseln zu ermöglichen und andererseits die Größe des soeben erhaltenen Teilproblems mitzuteilen, was im Rahmen künftiger Aufteilungen des Suchraums für den Master von Bedeutung ist. Eine dahingehende Kontaktaufnahme signalisiert dem Master-Prozess zudem, dass die Datenübertragung über die Switch-Matrix beendet ist, so dass die Kommunikationskanäle wieder freigegeben beziehungsweise anderweitig genutzt werden können.

7.2.6 Integration von Konflikt-Klauseln anderer Recheneinheiten

Wie in Abschnitt 7.2.3 festgelegt wurde, bestimmen die Clients nach jeder Konflikt-Analyse, ob die hergeleitete Konflikt-Klausel aus so wenigen Literalen besteht, dass sie auch für andere Clients von Interesse sein könnte. Fällt der Test positiv aus, wird die Klausel an den Master transferiert, der diese nach einem filternden Schritt an die restlichen Clients weiterleitet. Anders als bei PaMiraXT (siehe Abschnitte 9.1 und 9.2), wo je nach Konfiguration mehrere Konflikt-Klauseln zusammen als ein Datenpaket transferiert werden, wird in PIChaff jede Klausel separat weitergereicht.

Liegt einem Client eine solche Mitteilung des Master-Prozesses vor, nimmt er in einem ersten Schritt die Klausel Literal für Literal entgegen und speichert diese als neue Klausel in seiner lokalen Datenbank ab. Als zweites wird der *Status* der soeben erhaltenen Konflikt-Klausel bestimmt, es wird untersucht, ob die Klausel erfüllt ist, eine Implikation oder sogar einen Konflikt im aktuellen Zustand des Suchprozesses des Clients auslöst. Je nach Status müssen dabei unterschiedliche Operationen vorgenommen werden:

- In der aktuellen Situation des Clients ist die erhaltene Konflikt-Klausel weder erfüllt noch unerfüllt und beinhaltet zumindest zwei aktuell unbelegte Literale. Es werden zwei beliebige unbelegte Literale der Klausel als Watched Literals bestimmt und an den Anfang der Klausel verschoben. Weitere Operationen sind nicht erforderlich.

- Die Konflikt-Klausel ist erfüllt. Auch hier sind außer der Bestimmung der beiden Watched Literals keine weiteren Operationen notwendig, wobei gewährleistet wird, dass zumindest eines der die Klausel erfüllenden Literale zu einem Watched Literal wird.

- In der Konflikt-Klausel tritt genau ein unbelegtes Literal auf, während alle anderen Literale falsch belegt sind und die Klausel nicht erfüllen: es liegt eine Implikation vor. Zunächst wird der Decision Level bestimmt, auf dem die Implikation ausgelöst wird, was nicht zwangsläufig mit dem aktuellen Decision Level identisch sein muss. Danach wird ein Backtrack-Schritt auf die entsprechende Entscheidungsebene vorgenommen, um genau die Situation widerzuspiegeln, die eingetreten wäre, wenn die entsprechende Konflikt-Klausel bereits zu Beginn der Suche Teil der Klauselmenge

des Clients gewesen wäre. Anschließend wird die implizierte Variable mit dem geforderten Wahrheitswert belegt und die SAT-Prozedur fortgesetzt (wird die Implikation auf dem aktuellen Decision Level ausgelöst, wird auf das Backtracking verzichtet).

Abbildung 7.2 skizziert die Vorgehensweise anhand eines fiktiven Decision Stacks (im linken Teil der Abbildung dargestellt), der die aktuelle Position des Clients innerhalb des Suchraums spezifiziert. Es sei angenommen, dass der Client sich auf Decision Level 5 befindet und vom Master die Konflikt-Klausel $(\neg x_8 \vee x_{22})$ entgegengenommen hat, die offensichtlich durch die auf Decision Level 2 getätigte Zuweisung $x_8 = 1$ die Implikation $x_{22} = 1$ auslöst. Als Folge dessen wird zunächst eine Backtrack-Operation bis auf Level 2 vorgenommen, die Variable x_{22} mit dem Wahrheitswert 1 belegt und von dort aus die Suche fortgesetzt.

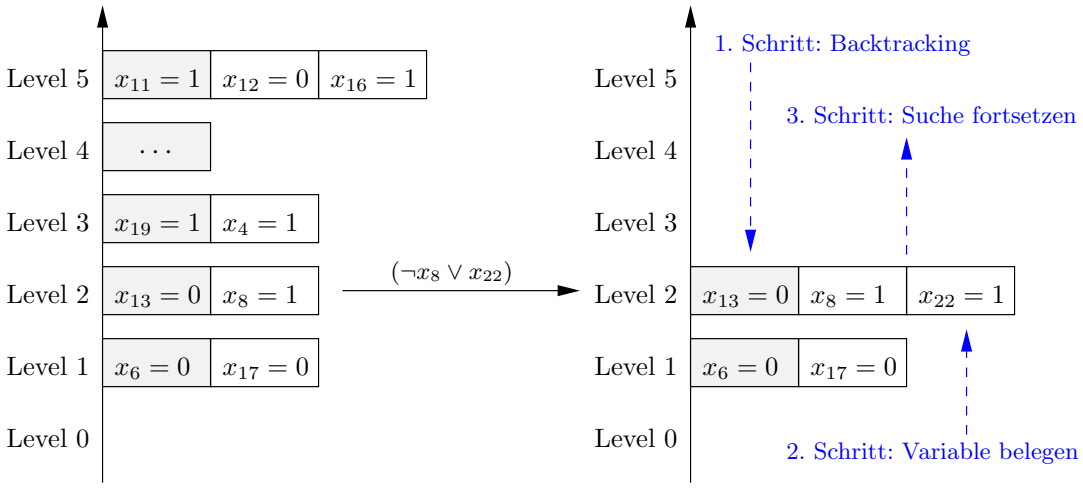


Abbildung 7.2: Vorgehensweise bei einer implikationsauslösenden Konflikt-Klausel

Sollte es sich bei der erhaltenen Klausel um eine Unit Clause handeln, wird zunächst auf Decision Level 0 zurückgesprungen und anschließend die sich aus der Unit Clause ergebende Zuweisung vorgenommen.

– Alle Literale der Konflikt-Klausel sind falsch belegt, die Klausel ist somit unerfüllt und löst einen Konflikt aus. Unter allen Literalen der Klausel werden die beiden Literale bestimmt, die zuletzt (auf den beiden „höchsten“ Entscheidungsebenen) falsch belegt wurden. Seien dies die beiden Decision Level DL_1 und DL_2. Gilt nun $DL_1 = DL_2$, so wird eine Backtrack-Operation auf diesen Decision Level vorgenommen und die Konflikt-Analyse durchgeführt (wird ein Konflikt auf dem aktuellen Decision Level des Clients ausgelöst, wird direkt die Funktion zur Konflikt-Analyse aufgerufen). Als Folge der Konflikt-Analyse werden die gemäß des ermittelten Backtrack Levels falschen Zuweisungen rückgängig gemacht, die erzeugte Konflikt-Analyse in die Klauselmenge eingetragen, eventuell an den Master-Prozess weitergegeben und

der Suchprozess fortgesetzt.

Gilt hingegen $DL_1 < DL_2$ (analog für $DL_2 < DL_1$), kann der Konflikt dadurch aufgelöst werden, dass per Backtracking auf Decision Level DL_1 zurückgesprungen wird, auf dem dann das vormals auf Decision Level DL_2 belegte Literal eine Implikation auslöst. Diese wird bearbeitet und anschließend die Suche nach einer erfüllenden Belegung fortgesetzt.

Handelt es sich bei der erhaltenen Klausel um eine Unit Clause, die im Widerspruch zu den auf Decision Level 0 getätigten Zuweisungen steht, so wird der Suchprozess abgebrochen, da die Unit Clause das gesamte vom jeweiligen Client aktuell betrachtete Teilproblem als unerfüllbar klassifiziert.

7.3 Aufgaben des Kommunikationsprozessors

Der Kommunikationsprozessor übernimmt in PIChaff die Rolle des Master-Prozesses. Das von diesem zu bewältigende Aufgabengebiet umfasst die Verteilung von Teilproblemen zwischen den Clients, die Weiterleitung von Konflikt-Klauseln und die Entgegennahme der erzielten Ergebnisse, die nach Beendigung aller Clients über die ISA-Schnittstelle an den angeschlossenen Rechner weitergegeben werden. Speziell die Umsetzung der beiden erstgenannten Punkte ist für das Leistungsvermögen von PIChaff von besonderer Bedeutung und wird in den Abschnitten 7.3.1 und 7.3.2 aus Sicht des Masters näher beleuchtet.

Weiterhin ist der Kommunikationsprozessor, wie in Abschnitt 7.1 festgelegt wurde, während der Startphase dafür verantwortlich, die sequentiellen SAT-Prozeduren von einer entsprechenden Anwendung auf Seiten des Rechners entgegenzunehmen und an die Recheneinheiten (Clients) weiterzuleiten. Wie bei anderen parallelen SAT-Verfahren auch, endet bei PIChaff die Initialisierungsphase damit, dass durch den Master ein Client aufgefordert wird, das gesamte Problem zu lösen. Die restlichen Clients werden so gestartet, dass sie als erstes eine Anfrage nach einem zu lösenden Teilproblem an den Master-Prozess stellen, so dass sich im Anschluss an das Startkommando sofort ein mehrfaches Aufteilen des Gesamtproblems anschließt, das erst endet, wenn alle Clients einen Teilbereich zugewiesen bekommen haben.

Die im Rahmen von PIChaff vom Master-Prozess bereitzustellende Funktionalität ist komplett im EEPROM des Kommunikationsprozessors verankert, so dass die entsprechenden Routinen im Gegensatz zu denen der Recheneinheiten nicht während der Startphase an das Multiprozessorsystem übertragen werden müssen.

7.3.1 Austausch von Teilproblemen zwischen den Recheneinheiten

Der Austausch von Teilproblemen zwischen zwei Clients wurde aus Sicht der Recheneinheiten bereits weitgehend durch die Ausführungen in Abschnitt 7.2.5 abgedeckt. Kann ein

Client sein gestelltes Teilproblem lösen, dabei aber keine erfüllende Belegung für die CNF-Formel bestimmen, so beendet er den aktuellen Suchprozess mit dem Ergebnis *UNSAT*, das er an den Master-Prozess weiter gibt (*UNSAT* bezieht sich an dieser Stelle nur auf das vom jeweiligen Client bearbeitete Teilproblem). Solange mindestens noch ein weiterer Client aktiv ist, wird einer davon durch den Master-Prozess kontaktiert und es werden entsprechende Kanäle der Switch-Matrix freigeschaltet, so dass der aktive Client einen Bereich seines aktuellen Teilproblems weitergeben kann. Ist kein weiterer Client mehr aktiv, was bedeutet, dass alle Clients ihre gestellte Aufgabe gelöst haben, aber keiner eine erfüllende Belegung finden konnte, ist die gestellte Probleminstanz unerfüllbar. Das Ergebnis wird an die das Multiprozessorsystem steuernde Anwendung auf Seiten des angeschlossenen Rechners weitergegeben und die Clients beenden auf Anweisung des Masters die Ausführung der sequentiellen SAT-Prozedur.

An dieser Stelle ist lediglich noch die Frage zu klären, welcher der aktiven Clients durch den Master kontaktiert wird, wenn ein Client inaktiv geworden ist und einen noch unbearbeiteten Bereich des durch die CNF-Formel aufgespannten Suchraums benötigt. Analog zu //Satz wird auch in PIChaff versucht, unter den von den aktiven Clients aktuell betrachteten Teilproblemen genau dasjenige Teilproblem aufzuspalten (beziehungsweise den entsprechenden Client zur Aufteilung aufzufordern), das wahrscheinlich am schwersten zu lösen ist und in aller Regel damit auch die meiste Laufzeit benötigt. Insbesondere soll dadurch verhindert werden, dass Teilprobleme weitergegeben werden, die sich in Sekundenbruchteilen lösen lassen, was einen neuerlichen Austausch verbunden mit einer weiteren, zeitaufwendigen Kommunikation nach sich zieht.

Dieses Ziel wird in PIChaff dadurch erreicht, dass der Master alle Teilproblemspezifikationen der noch aktiven Clients, die ihm diese nach dem Erhalt eines neuen Teilproblems jeweils mitteilen, intern speichert. Er kann somit die in Frage kommenden Clients beziehungsweise deren Guiding Paths einem Vergleich unterziehen und anhand des Ergebnisses dann denjenigen Client auswählen, dessen Guiding Path aus den wenigsten Zuweisungen besteht. Je weniger Zuweisungen bereits fest vorgegeben sind, desto höher ist die Zahl der Variablen der CNF-Formel, für welche unter Umständen noch beide Wahrheitswerte untersucht werden müssen. Üblicherweise korrespondiert dieser Wert mit der benötigten Laufzeit, so dass mit diesem Vorgehen der Client ausgewählt und kontaktiert wird, der aller Wahrscheinlichkeit nach das derzeit größte Teilproblem löst.

7.3.2 Austausch von Konflikt-Klauseln zwischen den Recheneinheiten

Abbildung 7.3 zeigt die Komponenten und Datenpfade des Multiprozessorsystems, die am Austausch von Konflikt-Klauseln beteiligt sind. Wie in Abschnitt 7.2.3 skizziert wurde, ist bereits die Übermittlung einer Konflikt-Klausel von dem die Klausel generierenden Client an den Master ein zeitaufwendiger Prozess (in der Abbildung grün hervorgehoben). Bedingt durch den nur 8 Bit breiten Datenbus der Microchip PIC17C43 Mikroprozessoren müssen

die einzelnen Literale der zu übertragenden Klausel in Paketen zu je 8 Bit transferiert werden. Bei der in Abbildung 7.3 blau markierten Weiterleitung der Konflikt-Klauseln vom Master an die restlichen Clients vervielfacht sich dieser Aufwand offensichtlich. Aus diesem Grund sollte der Austausch von Konflikt-Klauseln nur auf diejenigen Klauseln angewendet werden, bei denen die Chance besteht, dass sie bei den empfangenden Clients auch in der Tat zu einer Einschränkung des von diesen betrachteten Suchraums führen. Nur dann wird erreicht, dass die mit einer Reduzierung des Suchraums potenziell einhergehende Verringerung der Laufzeit des Suchprozesses den Mehraufwand durch den Austausch aufwiegt.

Als Ausweg aus dieser Problematik werden von den Clients nur solche Klauseln als hilfreich für andere Clients eingestuft und dann an den Master weitergegeben, die eine gewisse Anzahl an Literalen nicht übersteigen. Auf Seiten des Masters wird jede entgegengenommene Konflikt-Klausel wiederum nur an diejenigen Clients weitergeleitet, bei denen die Klausel nicht bereits durch den das jeweilige Teilproblem spezifizierenden Guiding Path erfüllt ist (ebenso wird der „Absender“ der Klausel von der Weiterleitung ausgenommen). Das kann zur Folge haben, dass eine Konflikt-Klausel an alle anderen, maximal acht Clients weitergegeben, nur an einzelne Clients transferiert oder auch gänzlich vom Master verworfen wird.

Abschließend sei erwähnt, dass der Austausch von Konflikt-Klauseln zwischen den diversen Clients prinzipiell auch auf direktem Weg über die Switch-Matrix hätte erfolgen können. Die Switch-Matrix muss dazu so konfiguriert werden, dass ein Client eine von ihm ermittelte und für relevant befundene Konflikt-Klausel gleichzeitig an mehrere andere Clients weiterleiten kann (entspricht der in Abbildung 6.11 dargestellten *One-to-Many*-Verbindung). Dies hätte aber einen stark erhöhten Synchronisationsaufwand zur Folge gehabt, so dass diese Idee verworfen wurde.

7.4 Aufgaben des angeschlossenen Rechners

Der Rechner, an den das Multiprozessorsystem angeschlossen ist, beziehungsweise ein auf dem Rechner ausgeführtes Programm hat im Wesentlichen die Aufgabe, sowohl Kommunikationsprozessor als auch Recheneinheiten mit den vom Anwender geforderten Einstellungen und Daten zu versorgen. Dies umfasst üblicherweise die auf der gewünschten Anzahl an Recheneinheiten auszuführende Zielanwendung sowie gegebenenfalls weitere problemspezifische Daten. Bezogen auf PIChaff ist mit der Entwicklungsumgebung *Microsoft Visual C++* ein entsprechendes Programm entwickelt worden, das über folgende Funktionalität verfügt (anders als bei MiraXT/PaMiraXT kam bei PIChaff ein Rechner mit einem *Microsoft Windows* Betriebssystem zum Einsatz):

- Der Anwender kann die Zahl der Recheneinheiten (Clients) festlegen, die eine gestellte CNF-Formel gemeinsam lösen sollen.
- Im zweiten Schritt wird die sequentielle SAT-Prozedur von PIChaff über die ISA-

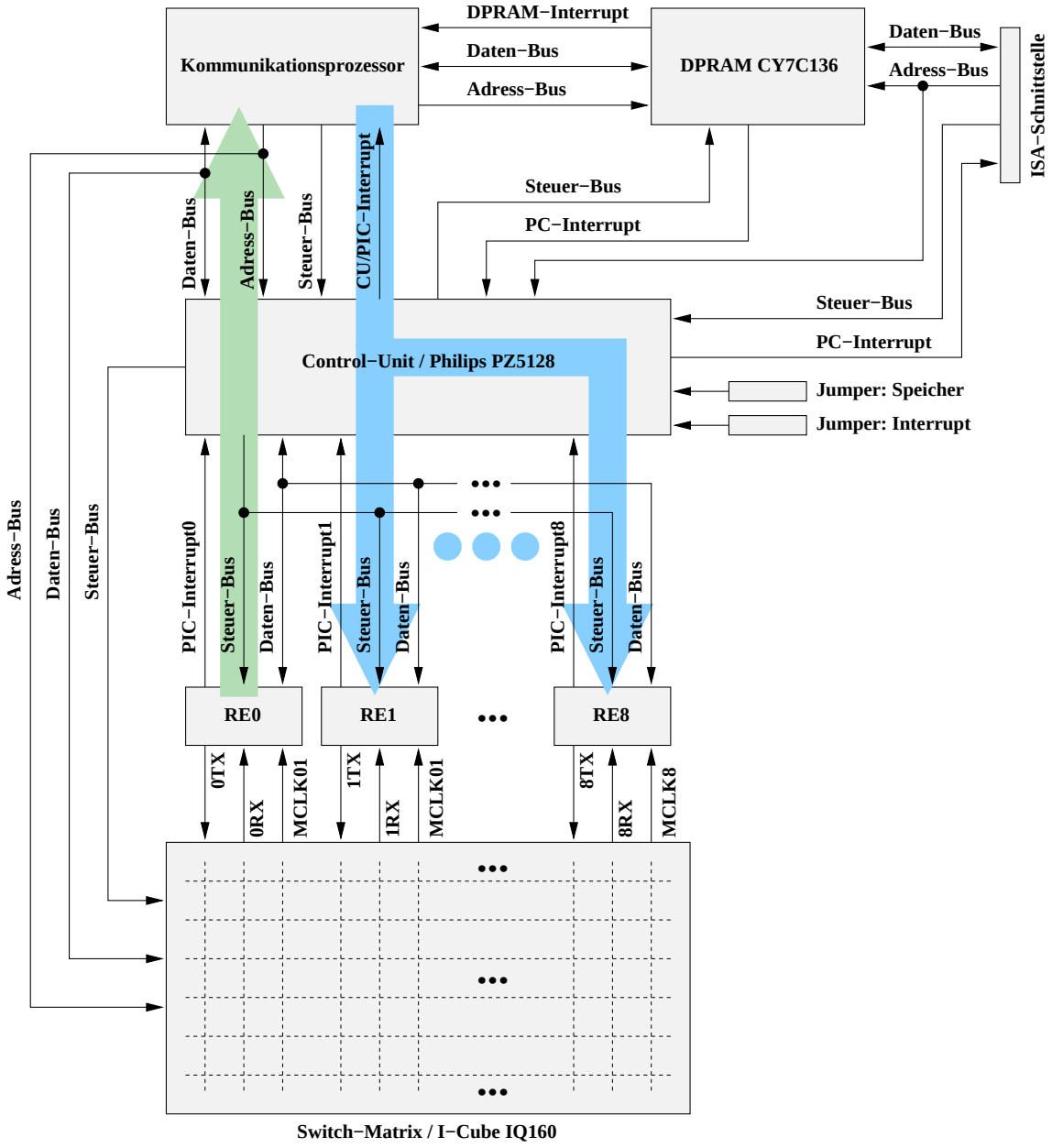


Abbildung 7.3: Am Austausch von Konflikt-Klauseln beteiligte Komponenten und Datenpfade des Multiprozessorsystems

Schnittstelle, das Dual-Ported RAM und den Kommunikationsprozessor an die zuvor festgelegten Recheneinheiten übertragen. An dieser Stelle wird vorausgesetzt, dass die entsprechenden Routinen vom Anwender bereits kompiliert und somit in ein PIC17C43-konformes Format übersetzt wurden.

– Danach wird die vom Anwender spezifizierte Probleminstanz eingelesen und ebenfalls via ISA-Schnittstelle, Dual-Ported RAM und Kommunikationsprozessor an die Recheneinheiten weitergeleitet. Wie auch beim vorherigen Punkt erfolgt die Übertragung blockweise, das bedeutet, dass der 2 kByte große Puffer des Dual-Ported RAM Bausteins erst komplett mit Daten gefüllt wird, bevor dem Kommunikationsprozessor durch Auslösen des Signals *DPRAM-Interrupt* (siehe Abbildung 6.9) ein neuer Datenblock signalisiert wird.

– Abschließend werden die Recheneinheiten und der Kommunikationsprozessor gestartet und nach Beendigung des Suchprozesses die bei der Lösung des Problems erzielten Ergebnisse entgegengenommen und dem Anwender am Bildschirm dargestellt.

Um diesem Programm den Zugriff auf die ISA-Schnittstelle und damit das Multiprozessorsystem zu ermöglichen, wurde mit Hilfe des Software-Pakets *NuMega DriverAgent* [24] ein Treiber realisiert, der auf Seiten des Rechners den Datentransfer auf Hardware-Ebene steuert und sich an den auf dem Trägerboard per Jumper eingestellten Speicherbereich und Interrupt-Level anpassen lässt.

7.5 Experimentelle Ergebnisse

In diesem Abschnitt werden die zur Evaluierung von PIChaff mit bis zu neun Clients durchgeführten Experimente und die dabei erzielten Ergebnisse präsentiert. Die Tabellen 7.1, 7.2 und 7.3 geben einen Überblick über die insgesamt 39 Probleminstanzen aus den Bereichen *Graph Colouring*, *Random Unsat* und *Random Sat*. Die Problemklasse *Graph Colouring* beinhaltet insgesamt 21 Instanzen des so genannten *Graphenfärbeproblems*, die unter anderem auch in der *SAT Competition 2002* eingesetzt wurden, während es sich bei den restlichen Instanzen um zufällig erzeugte Probleme handelt [51]. Alle CNF-Formeln wurden unter dem Gesichtspunkt ausgesucht, dass sie einerseits von einem Client mit vertretbarem Zeitaufwand gelöst werden können und andererseits auch für den parallelen Modus noch eine Herausforderung darstellen. Es sei darauf hingewiesen, dass die einzelnen Problemstellungen der drei Gruppen sich nur teilweise mit den in [89, 90, 91, 92, 93] eingesetzten Instanzen decken. Bedingt durch die jüngsten Optimierungen ist die aktuelle Version von PIChaff in der Lage, einige der in den genannten Veröffentlichungen betrachteten Probleme so schnell zu lösen, dass eine aussagekräftige Analyse insbesondere im parallelen Betriebsmodus nicht mehr möglich ist. Da PIChaff über keine eigene *Preprocessing*-Einheit verfügt, wurden alle CNF-Formeln im Vorfeld mit SatELite [31] vorverarbeitet und die modifizierten Problemstellungen dann an PIChaff übergeben.

Probleminstanz	SAT?	#Variablen	#Klauseln	#Klauseln (maximal)
3col120_5_2	SAT	202	867	4474
3col120_5_4	SAT	202	867	4474
3col120_5_5	SAT	202	867	4474
3col120_5_7	SAT	202	867	4474
3col120_5_10	SAT	202	867	4474
3col140_5_6	SAT	253	1058	4397
3col140_5_10	SAT	253	1058	4397
3col160_5_1	SAT	275	1182	4364
3col160_5_6	SAT	275	1182	4364
3col160_5_7	SAT	275	1182	4364
3col160_5_8	SAT	275	1182	4364
3col160_5_9	SAT	275	1182	4364
3col180_5_1	SAT	323	1380	4292
3col180_5_2	SAT	323	1380	4292
3col180_5_3	SAT	323	1380	4292
3col180_5_4	SAT	323	1380	4292
3col180_5_5	SAT	323	1380	4292
3col180_5_6	SAT	323	1380	4292
3col180_5_7	SAT	323	1380	4292
3col180_5_8	SAT	323	1380	4292
3col180_5_10	SAT	323	1380	4292

Tabelle 7.1: Übersicht der Problemklasse *Graph Colouring*

Probleminstanz	SAT?	#Variablen	#Klauseln	#Klauseln (maximal)
uuf225-01	UNSAT	222	957	4444
uuf225-02	UNSAT	220	956	4447
uuf225-06	UNSAT	223	958	4442
uuf225-07	UNSAT	221	958	4445
uuf225-011	UNSAT	222	957	4444
uuf225-012	UNSAT	219	951	4448
uuf225-013	UNSAT	219	953	4448
uuf225-014	UNSAT	216	951	4453
uuf225-015	UNSAT	220	956	4447

Tabelle 7.2: Übersicht der Problemklasse *Random Unsat*

Probleminstanz	SAT?	#Variablen	#Klauseln	#Klauseln (maximal)
uf225-01	SAT	222	957	4444
uf225-02	SAT	220	946	4447
uf225-04	SAT	222	956	4444
uf225-06	SAT	223	958	4442
uf225-07	SAT	220	955	4447
uf225-010	SAT	221	956	4445
uf225-012	SAT	221	956	4445
uf225-014	SAT	219	954	4448
uf225-015	SAT	218	951	4450

Tabelle 7.3: Übersicht der Problemklasse *Random Sat*

Bei der Betrachtung der drei Tabellen fällt auf, dass sich die Anzahl der Variablen und Klauseln auf einem Niveau von einigen hundert Variablen und um die tausend Klauseln bewegt. Der in der jeweils letzten Spalte der Tabellen 7.1, 7.2 und 7.3 angegebene Wert für *#Klauseln (maximal)* legt die maximale Anzahl an Konflikt-Klauseln fest, für die der in Abschnitt 7.2.2 angedeutete Speicherbereich zur Umsetzung des Konzepts der Watched Literals ausgelegt ist. Die Größe dieses Speicherblocks ergibt sich dabei aus der Gesamtkapazität des externen Speichers abzüglich des Speichers, der für die sequentiellen SAT-Prozeduren der Recheneinheiten, die Belegung der Variablen, den Decision Stack, die Klauselmenge der originalen CNF-Formel und einige weitere Speicherbereiche benötigt wird. Der verbleibende „Restspeicher“ wird dann so aufgeteilt, dass für eine neu zur Klauselmenge hinzugenommene Konflikt-Klausel nach Möglichkeit auch noch die Verweise auf die beiden Watched Literals innerhalb der entsprechenden WL-Listen gespeichert werden können. Die für *#Klauseln (maximal)* angegebenen Werte verdeutlichen, dass es mit PIChaff nicht möglich ist, große Fragestellungen zu bearbeiten. Allein die initiale Klauselmenge übersteigt in diesen Fällen die Speicherkapazität der Microchip PIC17C43 Mikroprozessoren.

Im Folgenden wird das Verhalten von verschiedenen Konfigurationen von PIChaff beim Lösen der 39 Probleminstanzen analysiert und bewertet. Die in den verschiedenen Tabellen angegebenen Zeiten sind dabei jeweils als Mittelwert von drei hintereinander ausgeführten Durchläufen zu verstehen. Das Vorgehen der einzelnen Clients im parallelen Betrieb von PIChaff wird unter anderem dadurch beeinflusst, welche Konflikt-Klauseln anderer Clients wann zur Verfügung stehen. Selbst eine Verzögerung von Sekundenbruchteilen verändert die Entscheidungsgrundlage, welcher Teil des Suchraums als nächstes analysiert wird, was offensichtlich zu unterschiedlichen Laufzeiten beim mehrmaligen Lösen der identischen Probleminstanz führen kann. Auch beim Austausch von Teilproblemen wird die Laufzeit unter Umständen dadurch beeinflusst, zu welchem Zeitpunkt welcher aktive Client an welchen inaktiven Client ein unbearbeitetes Teilproblem weitergibt.

Tabelle 7.4 gibt einen Vergleich zwischen der ursprünglichen *Variable State Independent Decaying Sum* Heuristik, wie sie im Rahmen von zChaff im Jahr 2001 vorgeschlagen wurde, und der für PIChaff gewählten Version, bei der die einzelnen Aktivitäten der Variablen nach einem abweichenden Schema inkrementiert werden (siehe Abschnitt 7.2.1). Auf allen drei Problemklassen ist die in PIChaff gewählte Adaption in der Lage, die jeweiligen Instanzen erheblich schneller zu lösen, insgesamt ergibt sich eine Beschleunigung um über das Doppelte. Die hier für den sequentiellen Modus von PIChaff mit einem Client durchgeführten Experimente wurden in internen Tests auch im parallelen Modus mit variierender Anzahl an Clients bestätigt. Die gegenüber dem Original vorgenommen Änderungen sind somit als Erfolg zu werten und wurden daher in alle im weiteren Verlauf diskutierten Varianten von PIChaff integriert.

Die beim Einsatz von einem, drei, sechs und neun Clients (ohne den Austausch von

Problemklasse	#Instanzen	Gesamtlaufzeit eines Clients [s]	
		$\text{VSIDS}_{original}$	$\text{VSIDS}_{PIChaff}$
Graph Colouring	21	6612,34	5193,15
Random Unsat	9	40775,99	19283,20
Random Sat	9	30500,03	14025,93
Gesamt	**39**	**77888,36**	**38502,28**

Tabelle 7.4: Vergleich zwischen der originalen VSIDS-Heuristik und der in PIChaff eingesetzten Variante

Konflikt-Klauseln im parallelen Modus) benötigten Gesamtlaufzeiten zum Lösen der einzelnen Problemklassen sind im oberen Teil von Tabelle 7.5 wiedergegeben, während im unteren Teil die jeweilige Beschleunigung im Vergleich zum sequentiellen Modus dargestellt ist. Es wird deutlich, dass sich über alle 39 CNF-Formeln gemittelt eine lineare Beschleunigung einstellt. Während der Speedup für die *Graph Colouring* Instanzen beim Einsatz von drei, sechs und neun Clients im zu erwartenden Rahmen liegt, stellt sich bei der Problemklasse *Random SAT* mit einem Wert von 28,10 (neun Clients) ein so genannter *superlinearer Speedup* ein, das heißt eine Beschleunigung, die höher ausfällt als die Anzahl der parallel agierenden Prozesse. Begründet werden kann dies bei erfüllbaren Instanzen dadurch, dass ein einzelner Client gegebenenfalls zunächst einen unerfüllbaren Teil des Suchraums komplett abarbeitet, bevor er in einen erfüllbaren Bereich wechselt und dort relativ schnell eine erfüllende Belegung ermittelt. Wird nun genau bezüglich der Variablen, die diese beiden Bereiche trennt, der Suchraum aufgeteilt, wird analog zum sequentiellen Fall ein Client den unerfüllbaren Bereich durchsuchen, während der zweite Client in kürzester Zeit die erfüllende Belegung findet, woraufhin der Algorithmus sofort gestoppt werden kann.

Obgleich dieses Argument nicht auf die Problemklasse *Random Unsat* zutrifft, ist PIChaff auch bei diesen CNF-Formeln in der Lage, sowohl mit drei, sechs als auch neun Clients eine superlineare Beschleunigung zu erzielen. Speziell bei dieser Kategorie hat sich herauskristallisiert, dass die Funktion, die für das Löschen von Konflikt-Klauseln zuständig ist, im parallelen Modus von PIChaff gegenüber der sequentiellen Variante bei der Mehrheit der Probleminstanzen seltener aufgerufen wird. Beispielsweise werden bei der Probleminstanz *uuf225-012* im sequentiellen Betrieb 24 dieser enorm zeitintensiven Lösch-Operationen durchgeführt, während im parallelen Betriebsmodus mit neun Clients insgesamt nur 22 Aufrufe gezählt wurden, die sich zudem auf die verschiedenen Clients verteilen, was schlussendlich zu einem Zeitvorteil führt.

Tabelle 7.6 fasst die Ergebnisse zusammen, die beim Einsatz von drei, sechs und neun Clients kombiniert mit dem Austausch von Konflikt-Klauseln mit maximal drei bezie-

Problemklasse	#Instanzen	Gesamtlaufzeit von PIChaff [s]			
		1 Client	3 Clients	6 Clients	9 Clients
Graph Colouring	21	5193,15	1718,54	1452,21	914,98
Random Unsat	9	19283,20	5525,33	2821,24	2022,28
Random Sat	9	14025,93	3191,44	1414,46	499,14
Gesamt	**39**	**38502,28**	**10435,31**	**5687,91**	**3436,40**

Problemklasse	#Instanzen	Speedup gegenüber 1 Client			
		1 Client	3 Clients	6 Clients	9 Clients
Graph Colouring	21	1,00	3,02	3,58	5,68
Random Unsat	9	1,00	3,49	6,84	9,54
Random Sat	9	1,00	4,39	9,92	28,10
Gesamt	**39**	**1,00**	**3,69**	**6,77**	**11,20**

Tabelle 7.5: Experimentelle Ergebnisse verschiedener Konfigurationen von PIChaff (ohne Austausch von Konflikt-Klauseln im parallelen Betrieb)

PIChaff mit 3, 6 und 9 Clients	Problemklasse		
	Graph Colouring	*Random Unsat*	*Random Sat*
Ohne Austausch von Konflikt-Klauseln			
Gesamtlaufzeit von 3 Clients [s]	1718,54	5525,33	3191,44
Gesamtlaufzeit von 6 Clients [s]	1452,21	2821,24	1414,46
Gesamtlaufzeit von 9 Clients [s]	914,98	2022,28	499,14
Mit Austausch von Konflikt-Klauseln bis zu einer Länge von 3			
Gesamtlaufzeit von 3 Clients [s]	**1670,72**	5830,49	3962,07
Gesamtlaufzeit von 6 Clients [s]	**1282,12**	3242,62	1843,63
Gesamtlaufzeit von 9 Clients [s]	926,90	2518,32	808,55
Mit Austausch von Konflikt-Klauseln bis zu einer Länge von 4			
Gesamtlaufzeit von 3 Clients [s]	1875,08	6702,28	4873,48
Gesamtlaufzeit von 6 Clients [s]	**1332,27**	4688,14	2753,08
Gesamtlaufzeit von 9 Clients [s]	1002,56	4953,70	1451,13

Tabelle 7.6: Experimentelle Ergebnisse verschiedener Konfigurationen von PIChaff (mit Austausch von Konflikt-Klauseln im parallelen Betrieb)

hungsweise vier Literalen ermittelt werden konnten. Der Austausch von Klauseln mit einer maximalen Länge von 1 und 2 ist mangels entsprechender Konflikt-Klauseln bei den getesteten Probleminstanzen gleichzusetzen mit einem Deaktivieren des Austauschs. Jeweils fett gedruckt sind die Konfigurationen, bei denen der Austausch von Konflikt-Klauseln zu einer Verringerung der Laufzeit führte. Es zeigt sich, dass diese Option auf der Problemklasse *Graph Colouring* zu einer Verbesserung führt, während sich der Austausch von Konflikt-Klauseln bei den beiden anderen Problemklassen stets zu einem Nachteil entwickelt. Beschränkt man sich in diesem Kontext auf die *Graph Colouring* Instanzen, wird deutlich, dass die Variante mit sechs Clients den besten Betriebsmodus darstellt, da sich beim Austausch von Konflikt-Klauseln mit maximal drei beziehungsweise vier Literalen jeweils eine Reduktion der Gesamtlaufzeit einstellt. Im ersten Fall (Austausch von Klauseln mit einer maximalen Länge von 3) führt dies zu einer Verbesserung der Performance um etwa 13%. Im Gegensatz dazu ist die Konfiguration mit neun Clients in beiden Szenarien nicht in der Lage, vom Austausch von Konflikt-Klauseln zu profitieren, die Anzahl der getauschten Konflikt-Klauseln und der damit verbundene Kommunikationsaufwand ist so groß, dass etwaige Beschneidungen des Suchraums und daraus resultierende Beschleunigungen des Suchprozesses in den Hintergrund geraten.

Tabelle 7.7 deutet in diesem Zusammenhang die Problematik eines zu hohen Kommunikationsaufwands, bedingt durch zu viele weitergeleitete Konflikt-Klauseln, exemplarisch anhand einer Konfiguration von PIChaff mit sechs Clients an. Ohne den Austausch von Konflikt-Klauseln liegt der Anteil der Kommunikation, die in diesem Szenario nur der Weitergabe von Teilproblemen dient, bei der Problemklasse *Graph Colouring* bei 3,2% und bei der Problemklasse *Random Unsat* bei 8,2% der Gesamtlaufzeit.[1] Wird der Austausch von Konflikt-Klauseln aktiviert, verändert sich die Situation: der Gesamtaufwand der Kommunikation, nun verursacht durch den Austausch von Konflikt-Klauseln und Teilproblemen, steigt bei den 21 Instanzen der Gruppe *Graph Colouring* in der Summe von 3,2% auf 12,6% der Gesamtlaufzeit bei durchschnittlich pro Instanz 127,43 durch den Master weitergeleiteten Konflikt-Klauseln. Dem gegenüber ist die Anzahl der vom Master weitergeleiteten Konflikt-Klauseln bei den neun zufällig generierten, unerfüllbaren Instanzen der Problemklasse *Random Unsat* deutlich höher und beträgt im Mittel 733,56 Konflikt-Klauseln pro Instanz. In Folge dessen steigt auch der Kommunikationsaufwand rasant an und macht bei dieser Gruppe von Problemen bereits 51,1% der Gesamtlaufzeit aus, was im Endeffekt zu einer Verschlechterung der Performance von PIChaff führt.

Die in der Spalte *Gesamtaufwand „SAT-Solving“* angegebenen Werte, die sich einzig auf den zum Lösen der Probleminstanz benötigten Zeitaufwand ohne etwaige Kommunikation beziehen, zeigen allerdings, dass auch bei der Problemklasse *Random Unsat* der Austausch

[1] Die Messung des Kommunikationsaufwands konnte aufgrund einer dahingehenden Nutzung des 16 Bit Timers *TMR0* der Microchip PIC17C43 Mikroprozessoren für jede Recheneinheit exakt ermittelt werden (siehe Abschnitt 7.1).

PIChaff, 6 Clients	Problemklasse	
	Graph Colouring	*Random Unsat*
Ohne Austausch von Konflikt-Klauseln		
Gesamtlaufzeit [s]	1452,21	2821,24
Gesamtaufwand Kommunikation [s]	46,40	232,50
Gesamtaufwand „SAT-Solving“ [s]	1405,81	2588,74
Mit Austausch von Konflikt-Klauseln bis zu einer Länge von 3		
Gesamtlaufzeit [s]	1282,12	3242,62
Gesamtaufwand Kommunikation [s]	162,07	1658,35
Gesamtaufwand „SAT-Solving“ [s]	1120,05	1584,27
Im Mittel vom Master weitergeleitete Klauseln	127,43	733,56

Tabelle 7.7: Einfluss des Austauschs von Konflikt-Klauseln auf den Kommunikationsaufwand im parallelen Betrieb mit sechs Clients

von Konflikt-Klauseln den eigentlichen Suchprozess beschleunigt. Ohne den Austausch von Konflikt-Klauseln benötigt der von sechs Clients durchgeführte Suchprozess 2588,74 Sekunden, während dieser Wert durch den Austausch von Konflikt-Klauseln bis zu einer Länge von 3 auf 1584,27 Sekunden fällt. Insgesamt überwiegt im zweiten Szenario aber dennoch der massive Zuwachs der Kommunikationszeit, der zu einem Anstieg der Gesamtlaufzeit von 2821,24 Sekunden auf 3242,62 Sekunden führt.

Die für die Instanzen der Kategorie *Random Unsat* bezüglich des Austauschs von Konflikt-Klauseln erzielten negativen Resultate sind in der Architektur des Multiprozessorsystems begründet, die an diesem Punkt keine effizientere Realisierung erlaubt. Wie in den Abschnitten 7.2.3 und 7.3.2 ausführlich dargelegt, ist die Kommunikation zwischen Master und Clients sehr zeitintensiv und nicht auf den Austausch größerer Datenmengen ausgelegt. Die beiden folgenden Kapitel werden aber zeigen, dass der Austausch von Informationen über die zu lösende Probleminstanz zwischen den verschiedenen sequentiellen SAT-Prozeduren, geeignete Kommunikationskanäle und Datenstrukturen vorausgesetzt, eine gute Möglichkeit ist, das Leistungsvermögen eines parallelen SAT-Algorithmus zu steigern.

Als Fazit der durchgeführten Experimente kann festgehalten werden, dass PIChaff im Vergleich zum sequentiellen Betriebsmodus mit lediglich einem Client sehr gut skaliert und über alle getesteten Probleminstanzen gemittelt eine lineare Beschleunigung erreicht. Durch den Austausch von Konflikt-Klauseln kann in Abhängigkeit von der jeweiligen Problemklasse und der Anzahl der eingesetzten Clients insbesondere bei den *Graph Colouring* Instanzen ein zusätzlicher Performance-Gewinn von bis zu 13% erzielt werden.

Kapitel 8

MiraXT

Mit der Entwicklung von MiraXT und dessen Erweiterung PaMiraXT, die im Mittelpunkt des Kapitels 9 steht, wird die hardwarenahe Programmierung verlassen. Beide Verfahren richten den Fokus auf „klassische“ Rechner beziehungsweise Rechnernetzwerke und sind mit C/C++ in einer Hochsprache entwickelt worden. Während in PIChaff neben dem eigentlichen SAT-Algorithmus zusätzlich noch Methoden zur Kommunikation zwischen den Mikroprozessoren auf unterster Hardware-Ebene etabliert werden mussten, konnte in diesem Bereich bei MiraXT/PaMiraXT auf existierende Funktionsbibliotheken beziehungsweise Software-Pakete wie PThread [22] und MPICH [48] zurückgegriffen werden.

Abbildung 8.1 zeigt das für MiraXT gewählte Design. Bedingt durch das umgesetzte Thread-Konzept wird an die Hardware-Umgebung die Anforderung gestellt, dass alle zur Verfügung stehenden Prozessoren beziehungsweise CPU-Kerne an einen gemeinsamen Speicher angebunden sind, was auf Dual-/Multi-Core Prozessoren und Mehrprozessorsysteme zutrifft. Analog zu beispielsweise MiniSat2 wird auch in MiraXT dem eigentlichen Suchprozess eine *Preprocessing*-Einheit vorgeschaltet, um die zu lösende Probleminstanz vor der Übergabe an die Threads nach Möglichkeit so zu vereinfachen, dass die daraus resultierende Beschleunigung der Suche nach einer erfüllenden Belegung den Zeitaufwand der Vorverarbeitung dominiert. Die dabei umgesetzte Funktionalität ist von algorithmischen Details abgesehen von MiniSat2 übernommen worden, so dass für weiterführende Informationen auf Abschnitt 4.2 verwiesen sei. Die modifizierte CNF-Formel wird im Anschluss an das Preprocessing an den SAT-Kern von MiraXT übergeben (*SAT Solving Unit*), der in Abbildung 8.1 beispielhaft für den parallelen Betriebsmodus mit vier parallel agierenden Threads dargestellt ist.

Die *SAT Solving Unit* arbeitet nach dem in Abschnitt 5.1 erläuterten Prinzip, bei dem die zu lösende CNF-Formel unter den Threads in disjunkte Teilprobleme aufgeteilt wird, die dann parallel bearbeitet werden. Während der Initialisierungsphase wird zunächst lediglich ein Thread gestartet, der das gesamte Problem löst. Die anderen Threads bleiben vorerst inaktiv und stellen Anfragen nach zu lösenden Teilproblemen, woraufhin der Suchraum solange aufgeteilt wird, bis alle Threads über ein initiales Teilproblem verfügen. Zur Abwicklung derartiger Anfragen kommt in MiraXT statt eines aktiven Master-Prozesses (wie er beispielsweise in PIChaff genutzt wird) lediglich eine passive, als *Master Control Object*

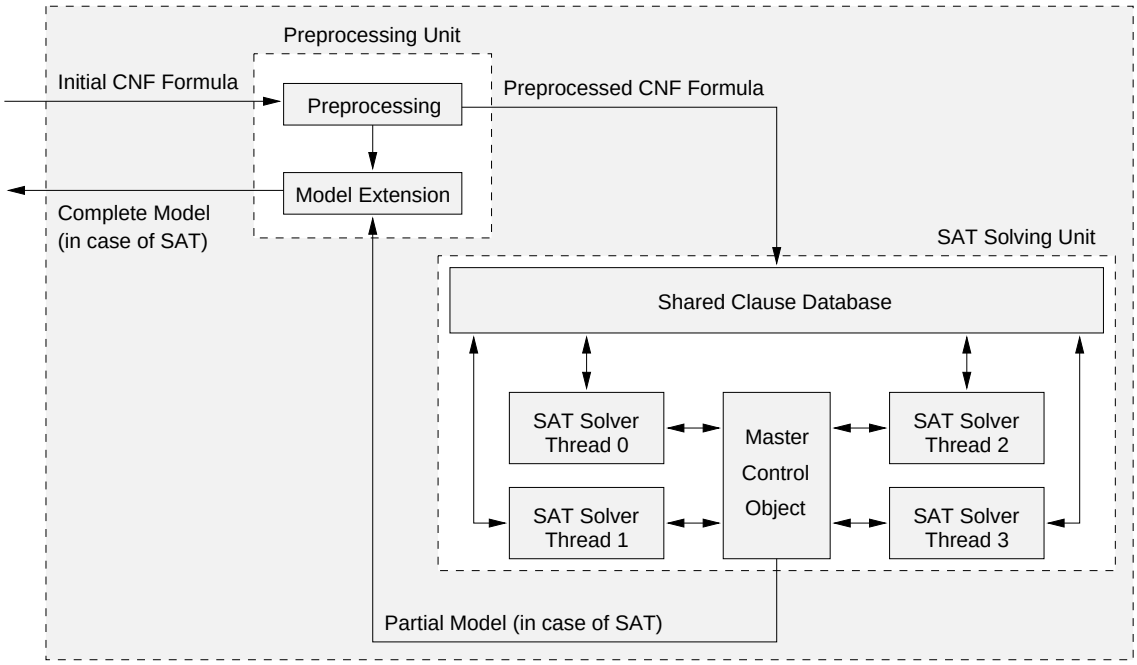


Abbildung 8.1: Design MiraXT

bezeichnete Datenstruktur zum Einsatz, die der Kommunikation zwischen den Threads dient. Tritt nun beispielsweise die Situation ein, dass ein Thread sein Teilproblem abgearbeitet hat, ohne allerdings eine erfüllende Belegung gefunden zu haben, wird durch das Setzen einer speziellen Variablen eine entsprechende Anfrage an die verbliebenen Threads im Master Control Object hinterlegt. Der erste aktive Thread, der diese Meldung liest, gibt daraufhin einen noch nicht analysierten Bereich seines eigenen Teilproblems an das Master Control Object ab, von dort wird es schlussendlich vom inaktiven Thread entgegengenommen und bearbeitet. Auch für den Fall, dass ein Thread eine erfüllende Belegung ermittelt hat, wird im Master Control Object eine entsprechende Variable gesetzt, die den anderen Threads signalisiert, dass die Suche beendet werden kann.

Wie Abbildung 8.1 illustriert, kommt in MiraXT lediglich eine einzige Klauseldatenbank zum Einsatz, die von allen Threads gemeinsam genutzt wird: die *Shared Clause Database*. Neben den Klauseln der initialen CNF-Formel enthält diese Klauseldatenbank alle Konflikt-Klauseln, die von den verschiedenen Threads während des Suchprozesses hergeleitet wurden. Durch diese Konzeption bietet sich den Threads die Möglichkeit, unmittelbar auf Konflikt-Klauseln anderer Threads zuzugreifen und somit von deren Wissen profitieren zu können. Hierbei bewerten die Threads in regelmäßigen Abständen die verfügbaren Konflikt-Klauseln und beziehen all diejenigen in ihren eigenen Suchprozess mit ein, die besonders geeignet sind, den Suchraum des von ihnen aktuell bearbeiteten Teilproblems

einzuschränken.

Im vorliegenden Kapitel wird in den Abschnitten 8.1 und 8.2 ein Überblick über die Realisierung der *Shared Clause Database* und des *Master Control Objects* gegeben, bevor in Abschnitt 8.3 die von den Threads ausgeführte sequentielle SAT-Prozedur vorgestellt wird. Abschließend wird in Abschnitt 8.4 das Leistungsvermögen von MiraXT ermittelt und anderen modernen SAT-Algorithmen gegenübergestellt.

8.1 Klauseldatenbank

Wie zuvor erwähnt, wird in MiraXT zur Abspeicherung der Klauseln lediglich eine Datenstruktur verwendet, die von den an der Suche nach einer erfüllenden Belegung beteiligten Threads gemeinsam genutzt wird. Abbildung 8.2 stellt schematisch das Design der *Shared Clause Database* zusammen mit der Anbindung des Threads mit der ID-Nummer 0 dar. Alle weiteren Threads sind nach dem gleichen Prinzip an die Klauseldatenbank angebunden, der Übersichtlichkeit halber aber nicht in der Abbildung aufgeführt.

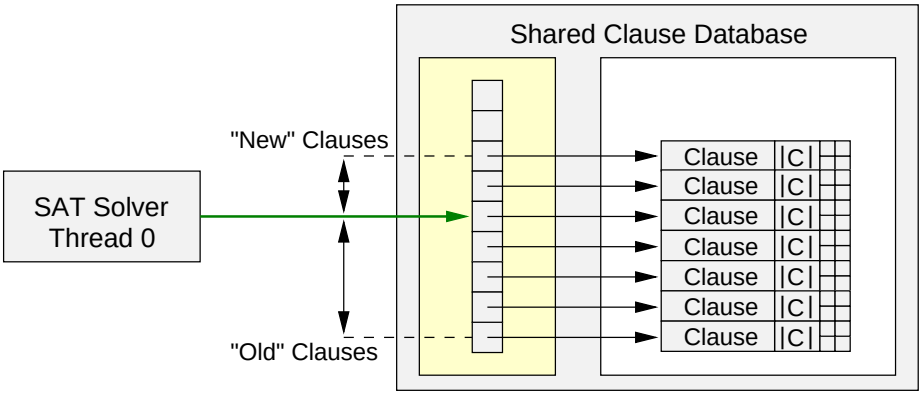


Abbildung 8.2: Shared Clause Database

Im Kern besteht die *Shared Clause Database* aus einer Liste von Zeigern (gelb unterlegt), die ihrerseits jeweils auf die Position innerhalb des Speichers verweisen, an dem die entsprechende Klausel abgelegt ist. Die Klauseln enthalten in Form eines *Arrays* nacheinander die einzelnen Literale sowie in einem separaten Feld die Länge der Klausel ($|C|$), die für die Threads zur Durchführung der BCP-Funktion und der Entscheidung, welche Klauseln in den eigenen Kontext eingebunden werden sollen, von Bedeutung ist. Weiterhin wird mit jeder Klausel und getrennt für jeden Thread ein so genanntes *Clause Deletion Flag* assoziiert, das angibt, ob diese Klausel aktuell vom entsprechenden Thread genutzt wird oder nicht. In Abbildung 8.2 sind diese Flags mittels kleiner Quadrate am Ende der Klauseln symbolisiert, hier ein Szenario mit vier Threads.

Jeder Thread besitzt als Anbindung an die Shared Clause Database einen speziellen Positionszeiger (in der Abbildung grün dargestellt), der dazu dient, bereits bekannte von neu in die Datenbank aufgenommenen Klauseln unterscheiden zu können. Wie in Abschnitt 8.3.3 erläutert wird, prüfen die Threads nach jedem Aufruf der Konflikt-Analyse, in deren Verlauf die hergeleitete Konflikt-Klausel in der Shared Clause Database abgelegt wird, welche Klauseln außer der eigenen Konflikt-Klausel noch neu zur Datenbank hinzugekommen sind. Da Konflikt-Klauseln immer ans Ende der bisherigen Liste mit Verweisen auf Klauseln angehängt werden, sind dies alle Klauseln, die sich zwischen dem grün dargestellten Positionszeiger des jeweiligen Threads und dem Ende der Zeigerliste befinden. Diese werden der Reihe nach analysiert und für den eigenen Suchprozess berücksichtigt oder verworfen. Abschließend wird der Positionszeiger auf das Ende der Liste gesetzt. Dies bedeutet, dass durch das stetige „nach oben“ Bewegen des Positionszeigers der Threads (zum Ende der Liste) eine Klausel nur ein einziges Mal analysiert wird. Fällt diese Analyse negativ aus und die Klausel wird verworfen, kann sie zu einem späteren Zeitpunkt nicht erneut begutachtet und gegebenenfalls doch noch in den Kontext des jeweiligen Threads einbezogen werden (beispielsweise bei der Bearbeitung eines neuen Teilproblems). Einerseits wird dadurch das vorhandene Wissen eventuell nicht optimal genutzt, andererseits begünstigt das gewählte Konzept eine schnelle Bewertung der unter Umständen Millionen von Konflikt-Klauseln, die nicht stets wieder von vorne beginnen muss. Weiterhin ermöglicht dieser Ansatz, den Speicherbedarf von MiraXT in vertretbaren Grenzen zu halten. Jede Klausel, die von keinem Thread mehr genutzt wird, das heißt, bei der von allen Threads das mit diesen jeweils assoziierte *Clause Deletion Flag* gesetzt wurde, wird in periodischen Abständen aus der Datenbank entfernt.

Als Implementierung wurde für die *Shared Clause Database* ein C++-Objekt gewählt. Neben Konstruktor und Destruktor bietet das Objekt noch drei weitere Routinen, mit denen Konflikt-Klauseln in die Datenbank eingetragen, Klauseln zur Begutachtung entgegengenommen und nicht mehr genutzte Klauseln gelöscht werden können. Die beiden letztgenannten Funktionen werden im Detail in den nachfolgenden Abschnitten erläutert, an dieser Stelle soll als Abschluss das Einfügen von Klauseln an einem Beispiel vorgestellt werden. Abbildung 8.3 illustriert das Vorgehen.

Eine von einem Thread während der Konflikt-Analyse per Resolution erzeugte Konflikt-Klausel wird im ersten Schritt an die Integrationsroutine der Shared Clause Database weitergegeben. Diese nimmt die Klausel entgegen, bestimmt die Länge und kopiert diese zusammen mit den Literalen der Klausel in einen neu angelegten Speicherbereich (in Abbildung 8.3 blau hervorgehoben). Weiterhin wird für jeden Thread das *Clause Deletion Flag* für speziell diese Klausel angelegt. Zur Durchführung dieser Operationen ist kein exklusiver Zugriff auf die Datenbank erforderlich. Das heißt, selbst wenn zwei oder mehr Threads zeitgleich eine Konflikt-Klausel in die Shared Clause Database eintragen und somit entsprechend viele Instanzen der Integrationsroutine parallel aufgerufen werden, besteht keine

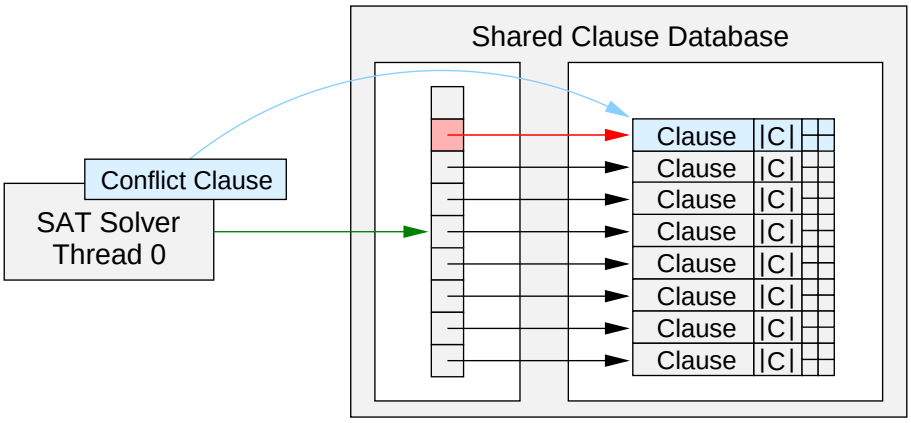


Abbildung 8.3: Einfügen einer Klausel in die *Shared Clause Database*

Gefahr, dass die Threads sich gegenseitig die abzuspeichernden Klauseln überschreiben, da für jede Klausel ein eigener Speicherblock neu angelegt wird.

Erst im nächsten Schritt (in der Abbildung rot markiert) ist ein exklusiver Zugriff auf die Datenbank notwendig, der dadurch erreicht wird, dass die erste einen solchen Schreibzugriff anfordernde Integrationsroutine die Shared Clause Database kurzfristig sperrt. In diesem Schritt wird der Verweis auf die neu angelegte Klausel generiert. Hier ist ein so genannter *Lock*, ein Sperren der Datenbank für andere Funktionen, notwendig, um zu gewährleisten, dass ein Verweis auf eine neue Klausel nicht von einer weiteren Instanz der Integrationsroutine überschrieben wird und dadurch die Klausel, auf die eigentlich verwiesen werden soll, verloren geht.

Es sei betont, dass ein Sperren der Shared Clause Database, was eventuell für Wartezeiten anderer Threads verantwortlich ist, neben dem in Abschnitt 8.3.4 diskutierten, sehr selten durchgeführten Löschen von Konflikt-Klauseln nur beim Einfügen einer neuen Klausel notwendig ist. Alle anderen Zugriffe wie etwa das Evaluieren neuer Konflikt-Klauseln oder die Bestimmung von Watched Literals während der BCP-Phase kommen gänzlich ohne Locks aus. Zudem muss während des Einfügens einer Konflikt-Klausel in die Datenbank die Sperrung nur sehr kurz und insbesondere nicht während des eigentlichen Kopiervorgangs der Klausel aufrecht gehalten werden. Wie die experimentellen Ergebnisse in Abschnitt 8.4 zeigen werden, konnte durch diese Konzeption verhindert werden, dass sich die Zugriffe auf die Klauseldatenbank und die damit verbundenen Wartezeiten zum „Flaschenhals“ des Algorithmus entwickeln und das Leistungsvermögen von MiraXT nachteilig beeinflussen.

8.2 Master Control Object

Das in MiraXT integrierte *Master Control Object* übernimmt analog zum Master-Prozess von PIChaff die Aufgabe, für eine reibungslose Weitergabe von Statusinformationen und Teilproblemen zwischen den Threads zu sorgen. Es handelt sich dabei allerdings nicht um einen separaten und aktiven Prozess, sondern lediglich um eine „passive“ Datenstruktur. Abbildung 8.4 gibt einen Überblick über das Master Control Object und die von diesem verwalteten Statusvariablen und Datenstrukturen.

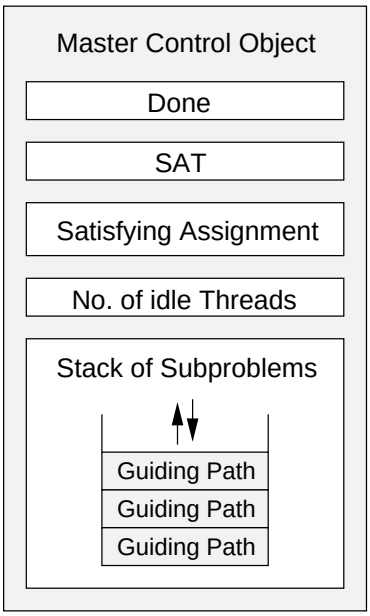


Abbildung 8.4: Master Control Object

Die Realisierung erfolgte wie bei der Klauseldatenbank in Form eines C++-Objekts und bietet neben Konstruktor und Destruktor diverse Routinen, mit denen die Threads auf die einzelnen Felder sowohl lesend als auch schreibend zugreifen können. Jeder schreibende Zugriff wurde aus Gründen der Datenkonsistenz mit einem Lock versehen, um insbesondere die Eintragungen in die beiden unteren Felder des Master Control Objects, die der Weitergabe von Teilproblemen dienen, zu schützen. Anders als bei der Shared Clause Database sind die schreibenden Zugriffe auf das Master Control Objekt deutlich seltener, so dass durch die Vergabe exklusiver Schreibrechte keine Performance-Nachteile entstehen.

Das Feld *Done* ist für die Steuerung des parallel durchgeführten Suchprozesses von zentraler Bedeutung und wird von den an der Suche nach einer erfüllenden Belegung beteiligten Threads in regelmäßigen Abständen gelesen. Es signalisiert, wann die von den Threads

ausgeführte sequentielle SAT-Prozedur gestoppt werden kann. Der Abbruch der Suche wird beispielsweise dann eingeleitet, wenn ein vom Benutzer vorgegebenes Zeitlimit überschritten wurde. Der erste diesen Sachverhalt feststellende Thread setzt die Variable *Done*, woraufhin alle Threads ihre Suche stoppen und die Ausführung von MiraXT mit einer entsprechenden Meldung beendet wird.

Gelingt es einem Thread, ein Modell für die gegebene CNF-Formel zu bestimmen, wird zunächst ebenfalls die Variable *Done* gesetzt, da ein Weiterführen des Suchprozesses nicht mehr erforderlich ist. Zudem wird die Variable *SAT* gesetzt sowie die erfüllende Belegung in *Satisfying Assignment* gespeichert. Die beiden letzten Operationen sind erforderlich, um nach Beendigung der Threads, wodurch alle von den Threads genutzten Variablen und Datenstrukturen gelöscht werden, dennoch den Zugriff auf das gefundene Modell zu haben und das zentrale Hauptprogramm von MiraXT mit einer entsprechenden Ausgabe beenden zu können.

Die beiden verbleibenden Datenfelder, *No. of idle Threads* und *Stack of Subproblems*, dienen den Threads zum Austausch von Teilproblemen, was wie folgt realisiert wurde: Immer wenn einer der Threads in einen inaktiven Zustand (*idle*) übergeht, das heißt, das aktuelle Teilproblem gelöst hat, ohne eine erfüllende Belegung gefunden zu haben, erhöht dieser Thread einen in *No. of idle Threads* gespeicherten Zähler und geht in einen „Wartezustand“ über. Der Zählerwert wird während der Initialisierungsphase mit 0 initialisiert, so dass jeder Zählerstand größer 0 eine entsprechende Anzahl inaktiver Threads repräsentiert. Die aktiven Threads lesen in regelmäßigen Abständen diesen Wert, was im Fall von *No. of idle Threads* > 0 als Aufforderung zur Aufteilung des eigenen Suchbereichs interpretiert wird. In derartigen Fällen stoppt ein aktiver Thread kurzfristig seinen Suchprozess und übergibt seinen derzeitigen Decision Stack an eine entsprechende mit dem Master Control Object assoziierte Funktion. Diese spaltet analog zu Abschnitt 5.1 einen noch unbearbeiteten Bereich des erhaltenen Decision Stacks beziehungsweise des dadurch spezifizierten Teilproblems ab, speichert das neue Teilproblem in *Stack of Subproblems* und modifiziert den ursprünglichen Decision Stack dahingehend, dass ein Wechsel des aktiven Threads in den soeben abgetrennten Bereich nicht möglich ist. Der sich im Wartezustand befindende inaktive Thread reagiert auf den neuen Eintrag in *Stack of Subproblems* und nimmt das dort zwischengespeicherte Teilproblem entgegen, woraufhin dieses aus der Liste wieder entfernt wird. Zudem wird der Zähler der inaktiven Threads um eins dekrementiert.

Die Entscheidung, welcher Thread ein Teilproblem abgibt, hängt davon ab, welcher aktive Thread zuerst den entsprechenden Eintrag im Master Control Object liest und wird somit zufällig getroffen. Liegen zu einer Anfrage eines inaktiven Threads Antworten von mehreren Threads vor, wird vom Master Control Object lediglich eine akzeptiert, alle anderen werden abgewiesen. Die Auslegung des Datenfelds *Stack of Subproblems* auf mehr als einen Eintrag beruht auf der Tatsache, dass gleichzeitig mehrere Threads inaktiv sein können und dementsprechend viele Teilprobleme im Master Control Object zwischengespeichert

werden müssen, bevor diese von den inaktiven Threads entgegengenommen werden.

Um eine korrekte Terminierung von MiraXT gewährleisten zu können, verlassen inaktive Threads den Wartezustand nicht nur nach dem Erhalt eines neuen Teilproblems, sondern auch wenn die Variable *Done* gesetzt wurde. Dies ist immer dann der Fall, wenn das vorgegebene Zeitlimit überschritten wurde oder eine erfüllende Belegung ermittelt werden konnte. Der jeweilige diesen Sachverhalt feststellende Thread setzt die Variable *Done* und signalisiert so den inaktiven Threads das Ende des Suchprozesses. Darüber hinaus wird auch dann die Variable *Done* gesetzt, wenn von insgesamt t Threads bereits $t-1$ Threads inaktiv sind und der letzte aktive Thread ebenfalls sein aktuelles Teilproblem gelöst hat, aber dabei auch keine erfüllende Belegung ermitteln konnte. Ohne eine Sonderfallbehandlung würde auch dieser Thread den Wert von *No. of idle Threads* um eins inkrementieren und in den Wartezustand übergehen, woraufhin alle Threads auf ein neues Teilproblem warten würden, was mangels aktiver Threads nicht mehr generiert werden kann. MiraXT würde in diesen Situationen nicht terminieren. Daher testet jeder SAT-Thread vor dem Inkrementieren des Zählers *No. of idle Threads* dessen aktuellen Wert. Beträgt dieser $t-1$, so ist klar, dass alle anderen Threads ebenfalls ihre Suche beendet haben und die CNF-Formel unerfüllbar ist. Der letzte SAT-Thread geht daher nicht in den Wartezustand über, sondern setzt die Variable *Done*, um den bereits wartenden Threads das Ende der Suche zu signalisieren.

8.3 SAT-Prozeduren der Threads

Nach einem Überblick über die *Shared Clause Database* und das *Master Control Object* werden in diesem Abschnitt nun die Eckpunkte der von den Threads ausgeführten sequentiellen SAT-Prozedur diskutiert.

8.3.1 Entscheidungsheuristik

Im Kern basiert die in MiraXT implementierte Entscheidungsheuristik auf der in PIChaff verwendeten VSIDS-Variante, die in Abschnitt 7.2.1 erklärt wurde. Nur der Parameter, mit dem die Aktivitäten der Variablen periodisch um einen konstanten Faktor geteilt werden, sowie der zeitliche Abstand zwischen zwei derartigen „Normalisierungen“ sind in MiraXT, bedingt durch die Anwendung auf sehr viel größere Probleminstanzen als bei PIChaff, anders gewählt worden. Zusätzlich wird nach jeweils 8192 Aufrufen der Entscheidungsheuristik eine Decision Variable samt Belegung zufällig bestimmt.

8.3.2 Boolean Constraint Propagation

Analog zu PIChaff wird auch in MiraXT das Prinzip der *Watched Literals* übernommen, mit dem gewährleistet werden kann, dass während der *Boolean Constraint Propagation* Phase immer nur die Klauseln untersucht werden, bei denen auch die Chance besteht,

Implikationen oder Konflikte bestimmen zu können. Zur effizienten Umsetzung ist dazu die *Watched Literals Reference List* (WLRL) eingeführt worden, die als eine Art Klauseldatenbank „im Kleinen“ aufgefasst werden kann. Abbildung 8.5 zeigt die schematische Struktur der *Watched Literals Reference List* am Beispiel einer Klausel bestehend aus drei Literalen (blau markiert), einer Klausel mit vier Literalen (rot markiert) sowie einer grün dargestellten Klausel, die alle Klauseln vertritt, die aus fünf oder mehr Literalen bestehen. Binäre Klauseln mit genau zwei Literalen sowie Unit Clauses werden in MiraXT separat behandelt und seien für den Moment außer Acht gelassen. Ebenso seien an dieser Stelle die mit jeder Variable assoziierten Listen, die angeben, in welchen Klauseln die Variable ein Watched Literal darstellt, vernachlässigt. Wie sich im weiteren Verlauf zeigen wird, werden bei MiraXT für jede Variable nicht nur die üblichen zwei sondern insgesamt acht Listen mitgeführt.

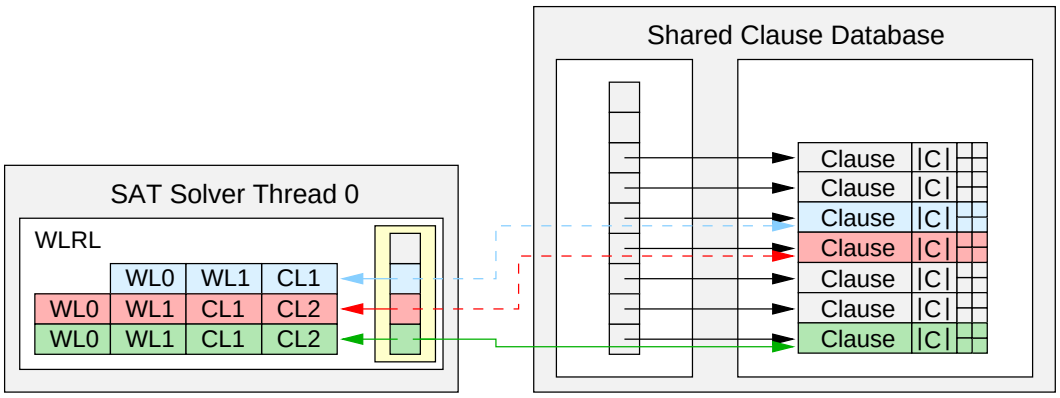


Abbildung 8.5: Watched Literals Reference List

Die Watched Literals Reference List besteht wie auch die Shared Clause Database im Wesentlichen aus einer Liste von Zeigern, die in Abbildung 8.5 gelb unterlegt ist. Im Fall der WLRL zeigen die Elemente dieser Liste zum Einen auf Klauseln innerhalb der Klauseldatenbank, zum Anderen auf die dafür lokal gespeicherten Watched Literals (WL0 beziehungsweise WL1) und je nach Länge der Klausel auf ein oder zwei zusätzliche Literale, die so genannten *Cache Literals* (CL1 beziehungsweise CL2). Innerhalb der Watched Literals Reference List wird für jede von dem entsprechenden Thread aktuell genutzten Klausel ein derartiger Eintrag angelegt, so dass jeder Thread „komprimierte“ Kopien aller von ihm berücksichtigten Klauseln in seiner WLRL-Struktur und damit in seinem privaten Speicherbereich verwaltet.

Offensichtlich sind Klauseln mit lediglich drei oder vier Literalen bei dieser Konzeption vollständig in der Watched Literals Reference List gespeichert, wobei die zwei Watched Literals stets am Anfang des entsprechenden WLRL-Elements abgelegt sind. Wird im Verlauf

der Abarbeitung der sequentiellen SAT-Prozedur für eine derartige Klausel ein Watched Literal falsch belegt, genügen die Daten des mit der Klausel assoziierten WLRL-Eintrags während der BCP-Operation, um einen Nachfolger zu bestimmen: es wird geprüft, ob eines der beiden Cache Literals als neues Watched Literal in Frage kommt. Ist dies der Fall, werden die beiden Einträge vertauscht, andernfalls liegt je nach Belegung des zweiten Watched Literals entweder eine Implikation oder ein Konflikt vor.[1] Zur Evaluierung während der BCP-Operation sind die Verweise auf den Speicherort der kompletten Klausel innerhalb der Klauseldatenbank nicht notwendig und daher lediglich gestrichelt dargestellt. Die Verweise werden bei Klauseln mit drei oder vier Literalen nur während der Konflikt-Analyse benötigt und um den Threads die Möglichkeit zu geben, die entsprechenden Klauseln durch ein Setzen des *Clause Deletion Flags* als „nicht mehr genutzt“ markieren zu können.

Etwas komplizierter gestaltet sich die Situation bei Klauseln mit mindestens fünf Literalen, bei denen der WLRL-Eintrag ebenfalls aus vier Literalen der Klausel besteht. Auch hier wird im Fall eines falsch belegten Watched Literals zunächst geprüft, ob entweder CL1 oder CL2 als Nachfolger in Frage kommt. Ist dies möglich, so werden wie zuvor die beiden Einträge vertauscht. Ist dies nicht der Fall, kann nicht unmittelbar auf eine Implikation oder einen Konflikt geschlossen werden, da die aktuell betrachtete Klausel nicht nur aus den Watched Literals und den beiden Cache Literals besteht, sondern zumindest noch aus einem weiteren Literal. In dieser Situation wird mit Hilfe des Zeigers, der auf den physikalischen Speicherort der Klausel verweist, untersucht, ob eines der restlichen Literale der Klausel als Watched Literal in Frage kommt. Im positiven Fall wird es an die entsprechende Position verschoben (entweder WL0 oder WL1) und das ehemalige Watched Literal überschreibt eines der beiden Cache Literals. Im negativen Fall liegt in Abhängigkeit des Status des zweiten Watched Literals entweder eine Implikation oder ein Konflikt vor. Die Evaluierung von Klauseln bestehend aus fünf oder mehr Literalen kann folglich nicht allein auf Basis der WLRL-Daten erfolgen, sondern zieht gegebenenfalls eine Analyse der gesamten Klausel nach sich.

Auf den ersten Blick erscheint die Art der Auswertung von Klauseln der Länge fünf und mehr nicht optimal zu sein, da, falls eine Analyse der gesamten Klausel erforderlich ist, auch die vier Literale erneut begutachtet werden, die bereits im WLRL-Eintrag gespeichert sind. Andererseits haben die in [71] für eine Variante von MiraXT mit lediglich einem Cache Literal pro Klausel durchgeführten Experimente gezeigt, dass in etwa 84% aller Klauselevaluierungen während der Boolean Constraint Propagation die Einträge der Watched Literals Reference List ausreichen, um ein neues Watched Literal zu bestimmen. Das bedeutet, dass nur in 16% der Fälle ein Zugriff auf die gesamte Klausel, einhergehend mit einem erhöhten Aufwand, nötig ist. Durch den Einsatz von bis zu zwei Cache Literals

[1] Analog zu PIChaff wird auch in MiraXT vor der Suche nach einem Nachfolger für ein falsch belegtes Watched Literal zunächst der „Status“ des zweiten Watched Literals überprüft. Erfüllt dieses Watched Literal die entsprechende Klausel, wird auf die Suche nach einem Nachfolger für das falsch belegte Watched Literal verzichtet.

pro Klausel sinkt dieser Anteil noch weiter.

Es sei angemerkt, dass die Idee der Cache Literals zu einem gewissen Teil dem mit der *Shared Clause Database* verfolgten Konzept widerspricht, welches zum Ziel hat, jede Klausel nur einmal gespeichert zu halten. Das gewählte Design bietet aber bei der Ausführung auf Mehrprozessorsystemen wie dem in Abschnitt 8.4 eingesetzten *Dual-Core AMD Opteron 280 Doppelprozessorsystem* erhebliche Vorteile. Dort stellt Abbildung 8.9 in vereinfachter Form die Architektur der genannten Hardware-Plattform dar, bei der zwei Prozessoren in unterschiedlichen Gehäusen und Fassungen auf einer Hauptplatine untergebracht und jeweils direkt an einen 2 GB großen Hauptspeicher angeschlossen sind. Jedem Prozessor ist es möglich, auf den Speicher des anderen Prozessors zuzugreifen, allerdings sind derartige Operationen erheblich langsamer als Zugriffe auf den lokal angeschlossenen Speicher. Im parallelen Betriebsmodus ist MiraXT so ausgelegt, dass die auf den *AMD Opteron* Prozessoren ausgeführten Threads ihre komplette *Watched Literals Reference List* immer im lokal an den jeweiligen Prozessor angeschlossenen Speicher ablegen, während sich die Klauselmenge üblicherweise auf beide Hauptspeicherblöcke erstreckt. Solange nun die WLRL-Einträge in der Mehrheit aller Evaluierungen genügen, um ein neues Watched Literal bestimmen zu können, sind nur Zugriffe auf den lokalen und damit schnellen Speicher erforderlich, unabhängig davon, in welchem Speicher-Modul die eigentliche Klausel physikalisch gespeichert ist. Das bedeutet, dass nach dem Anlegen des WLRL-Eintrags, was einen Zugriff auf die Klausel und den gegebenenfalls „weiter entfernten“ Speicher notwendig macht, die Threads im Allgemeinen mit lokalen Speicherzugriffen auskommen, was die ungleich schnellere Variante ist.

Wie zu Beginn angedeutet, werden binäre Klauseln mit zwei Literalen in MiraXT nicht in Form von WLRL-Elementen repräsentiert, sondern separat gehalten. Dies ist im Bereich der SAT-Algorithmen ein gängiges Vorgehen und wird unter anderem auch von Siege und PicoSAT umgesetzt. Abbildung 8.6 zeigt das Grundprinzip am Beispiel von drei binären Klauseln.

Für jede Variable und deren Vorkommen als positives beziehungsweise negatives Literal in binären Klauseln wird jeweils eine Liste mitgeführt, die angibt, welche Implikationen sich bei einer entsprechenden Zuweisung an die Variable ergeben. Im skizzierten Beispiel führt dies dazu, dass für das Literal $\neg x_1$ eine Liste mit den drei Einträgen x_7, $\neg x_2$ und x_9 mitgeführt wird, da aus der Zuweisung $x_1 = 0$ anhand der drei Klauseln $(x_1 \vee x_7)$, $(x_1 \vee \neg x_2)$ und $(x_1 \vee x_9)$ die Implikationen $x_7 = 1$, $x_2 = 0$ und $x_9 = 1$ gefolgert werden können. Umgekehrt ist in den zu $\neg x_7$, x_2 und $\neg x_9$ korrespondierenden Listen jeweils der Eintrag x_1 enthalten, da sowohl $x_7 = 0$, $x_2 = 1$ als auch $x_9 = 0$ die Zuweisung $x_1 = 1$ erzwingen. Diese Methodik hat den Vorteil, dass alle sich aus binären Klauseln ergebenden Implikationen direkt verarbeitet werden können. Die in der Abbildung zwischen je zwei Implikationen angegebenen Verweise auf die eigentliche Klausel dienen während der Konflikt-Analyse dazu, den Auslöser einer Implikation bestimmen und somit die korrekte

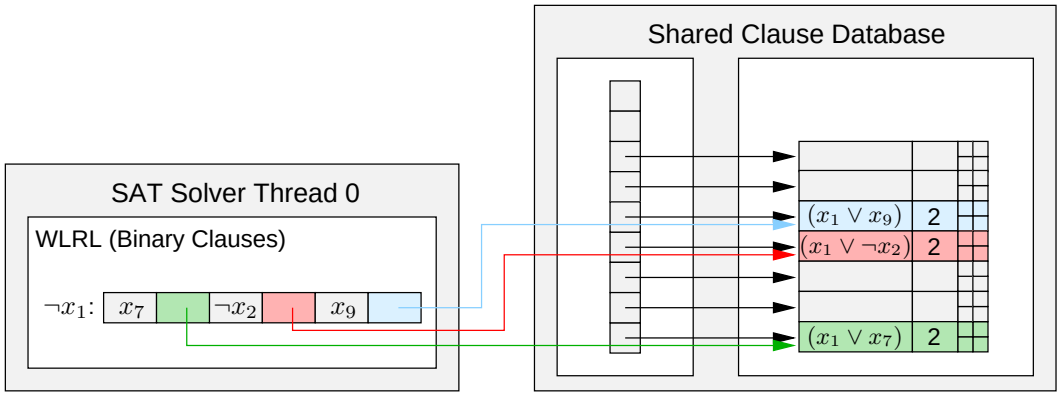


Abbildung 8.6: WLRL-Einträge binärer Klauseln

Herleitung der Konflikt-Klausel gewährleisten zu können.

Ebenso wie binäre Klauseln werden auch Unit Clauses gesondert behandelt, allerdings werden von den Threads keinerlei Verweise auf derartige Klauseln angelegt. Stattdessen zieht jede Unit Clause sofort eine Backtrack-Operation auf Decision Level 0 nach sich, auf dem dann die sich aus der Unit Clause ergebende Implikation unabänderlich verankert wird.

Aufbauend auf der Datenstruktur der *Watched Literals Reference List* und der Unterteilung in verschiedene Gruppen von Klauseln werden in MiraXT für jede Variable insgesamt acht Listen (in Kapitel 7 als WL-Listen bezeichnet) bereitgestellt, die angeben, in welchen Klauseln mit zwei, drei, vier oder mehr als vier Literalen eine Variable in Form eines positiven beziehungsweise negativen Literals derzeit ein Watched Literal darstellt. Die eigentliche BCP-Funktion operiert nun so, dass nach jeder Variablenzuweisung zunächst die sich aus binären Klauseln ergebenden Auswirkungen auf den Suchprozess bestimmt werden, bevor die Menge der potenziell eine Implikation oder einen Konflikt auslösenden Klauseln mit drei, vier und mehr als vier Literalen analysiert wird. Bei einer Änderung eines Watched Literals wird dabei der Verweis auf die entsprechende Klausel aus der WL-Liste des ehemaligen Watched Literals entfernt und der WL-Liste des neuen Watched Literals angehängt.

Zur Beschleunigung der BCP-Funktion wurde in MiraXT neben dem in Abschnitt 4.4.2 diskutierten *Early Conflict Detection Based BCP* noch das so genannte *Implication Queue Sorting* integriert, bei dem die Kernidee darin besteht, die Einträge der Implication Queue so anzuordnen, dass möglichst schnell auf etwaige Konflikte geschlossen werden kann. An dieser Stelle sei für weitere Informationen auf [70] verwiesen.

8.3.3 Konflikt-Analyse und Non-Chronological Backtracking

Die in MiraXT durchgeführte Konflikt-Analyse ist konzeptuell identisch zu den entsprechenden Routinen anderer moderner SAT-Algorithmen, die allesamt auf dem 1UIP-Prinzip aufbauen. Abgesehen von Implementierungs-Details gibt es im sequentiellen Betriebsmodus keinen Unterschied zu anderen Verfahren. Allerdings ist das Vorgehen im parallelen Szenario dahingehend erweitert worden, dass an dieser Stelle im Algorithmus die Threads die von anderen Threads bereitgestellten Konflikt-Klauseln bewerten und gegebenenfalls in den eigenen Suchprozess einbeziehen. Daher wird im Anschluss an die Konflikt-Analyse und das Einfügen der eigenen Konflikt-Klausel in die Shared Clause Database nicht sofort ein Backtracking vorgenommen. Stattdessen wird zuerst geprüft, welche Klauseln anderer Threads seit der letzten diesbezüglichen Anfrage der Shared Clause Database hinzugefügt wurden und welche davon für das aktuelle Teilproblem von Nutzen sein könnten. Wie in Abschnitt 8.1 erklärt, wird für diesen Zweck der mit jedem Thread assoziierte Positionszeiger verwendet, der angibt, welche Konflikt-Klauseln für den anfragenden Thread neue Informationen darstellen. Nacheinander werden alle neu in die Datenbank aufgenommenen Konflikt-Klauseln der Reihe nach analysiert und entweder in den eigenen Kontext integriert oder verworfen, wobei im zweiten Fall dann direkt das entsprechende *Clause Deletion Flag* gesetzt wird.

In der aktuellen Variante von MiraXT werden von den Threads alle Klauseln übernommen, die aus maximal 10 Literalen bestehen oder zum aktuellen Zeitpunkt des Suchprozesses eine Implikation oder einen Konflikt auslösen. Die Integration in den Kontext eines Threads bedeutet, dass für jede hinzugenommene Klausel zunächst ein neuer WLRL-Eintrag erzeugt wird. Danach werden, analog zu den für PIChaff in Abschnitt 7.2.6 gemachten Ausführungen, je nach Status der Klausel gegebenenfalls erforderliche Operationen eingeleitet und beispielsweise ein Backtracking vorgenommen, um einen durch eine neue Klausel hervorgerufenen Konflikt aufzulösen.

Da der Reihe nach alle für den anfragenden Thread neuen Klauseln der Shared Clause Database begutachtet werden, hat dieses Vorgehen unter Umständen zur Folge, dass mehrere Backtrack-Operationen hintereinander ausgeführt werden, bei denen, hervorgerufen durch entsprechende Klauseln, auf immer niedrigere Decision Level zurückgesprungen wird, bevor der Suchprozess schlussendlich fortgesetzt wird.

8.3.4 Löschen von Konflikt-Klauseln

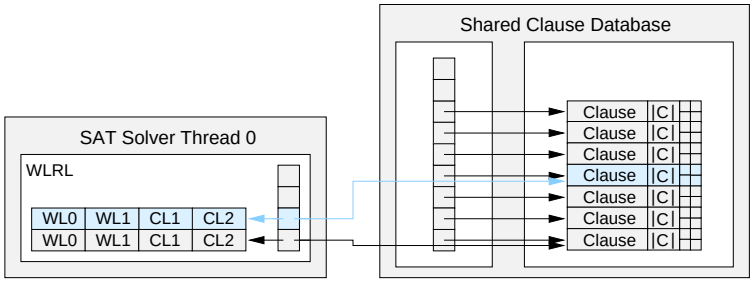
Wie in Abschnitt 4.6 erläutert, hat das Löschen von Konflikt-Klauseln unter anderem zum Ziel, zu verhindern, dass sich die Durchführung der Boolean Constraint Propagation aufgrund einer fortwährend wachsenden Zahl an zu analysierenden Klauseln stetig verlangsamt. Die bisherigen Abschnitte lassen bereits erahnen, dass das Löschen von Konflikt-Klauseln im Fall von MiraXT auf zwei Ebenen durchgeführt werden muss: zum Einen

auf der Ebene der Threads, die Konflikt-Klauseln aus ihrem eigenen Kontext entfernen, ohne dass dadurch auch zwangsläufig die entsprechenden Klauseln aus der *Shared Clause Database* entfernt werden, da sie gegebenenfalls noch von anderen Threads verwendet werden; zum Anderen auf der Ebene der Klauseldatenbank, aus der periodisch alle Konflikt-Klauseln entfernt werden, bei denen alle Threads das *Clause Deletion Flag* gesetzt haben und die somit von keinem Thread mehr berücksichtigt werden. Letzteres reduziert den Speicherverbrauch von MiraXT.

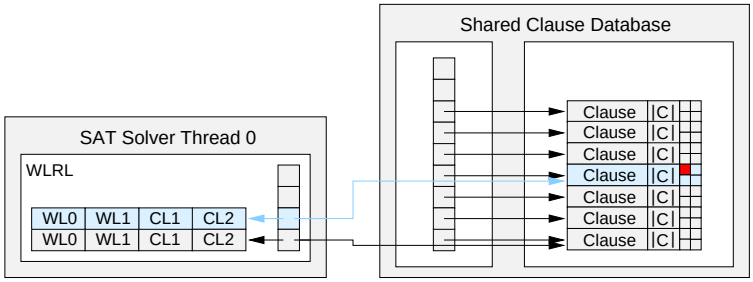
Auf der Ebene der Threads folgt das Entfernen von Konflikt-Klauseln einem Konzept, das als Kombination der Strategien von BerkMin und Grasp aufgefasst werden kann. Vereinfacht ausgedrückt werden in gewissen Abständen alle Konflikt-Klauseln aus der Menge der aktuell betrachteten Klauseln eines Threads entfernt, die aus sehr vielen Literalen bestehen (und daher den Suchraum nur sehr wenig einschränken) und zudem wenig Einfluss auf den bisherigen Verlauf der Suche hatten. Um das zweite Auswahlkriterium umzusetzen, werden nicht nur für die Variablen, sondern auch für die Klauseln Aktivitätszähler eingesetzt, deren Wert immer dann inkrementiert wird, wenn die entsprechende Klausel für einen während der Konflikt-Analyse durchgeführten Resolutions-Schritt herangezogen wurde. Das hat zur Folge, dass Klauseln genau dann eine hohe Aktivität aufweisen, wenn sie oft an Konflikten beteiligt sind und auf diesem Weg dazu beitragen, das verbleibende Restproblem weiter einzuschränken. Analog zu den Aktivitäten der Literale werden die Zähler der Klauseln periodisch durch einen konstanten Faktor geteilt.

Das Löschen von Konflikt-Klauseln wurde so realisiert, dass nach einer festgelegten Anzahl von in den eigenen Kontext aufgenommenen Konflikt-Klauseln jeder Thread einige von diesen wieder löscht, das heißt, die entsprechenden WLRL-Einträge entfernt und das jeweilige *Clause Deletion Flag* setzt. Dazu wird zunächst eine Liste all der Konflikt-Klauseln angelegt, die von dem jeweiligen Thread derzeit berücksichtigt werden, aus mehr als 10 Literalen bestehen und zugleich aktuell keine Implikation auslösen (ansonsten besteht die Gefahr, dass im Konflikt-Fall der Grund für eine Implikation gelöscht wurde und die Konflikt-Analyse nicht korrekt durchgeführt werden kann). Im zweiten Schritt werden die 50% inaktivsten Konflikt-Klauseln dieser Liste bestimmt und gelöscht.

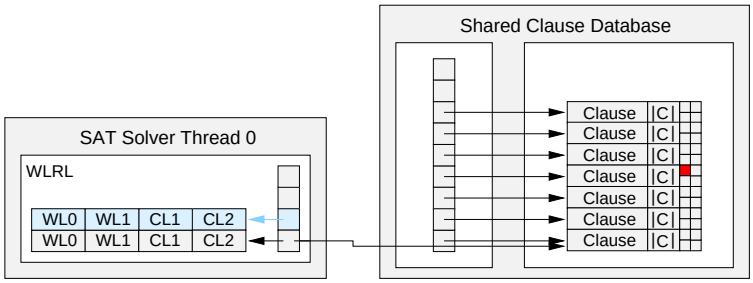
Für jede vom Löschen betroffene Klausel wird dabei die in Abbildung 8.7 skizzierte Prozedur durchgeführt. Es sei angenommen, dass der Thread mit der ID-Nummer 0 sich entscheidet, die blau unterlegte Konflikt-Klausel aus seinem Kontext zu entfernen. Zuerst wird das *Clause Deletion Flag* der Klausel gesetzt (symbolisiert durch ein rotes Kästchen). Danach wird der Verweis auf die Klausel entfernt, gefolgt vom Löschen des entsprechenden WLRL-Eintrags. Da neue Elemente stets an das Ende der WLRL-Liste angehängt werden, weist die *Watched Literals Reference List* des Threads nach mehreren Lösch-Operationen üblicherweise eine Reihe von Leerstellen auf, die periodisch durch ein Aufrücken der nachfolgenden Einträge beseitigt werden. Alle Operationen sind ohne exklusive Schreibrechte auf die globale Datenbank durchführbar, da nur lokale Daten betroffen sind beziehungs-



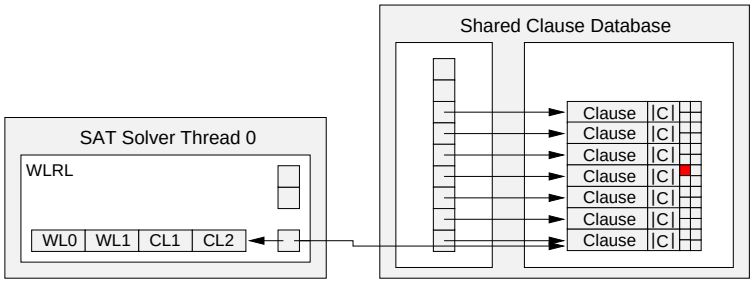
(a) Ausgangsbasis: Löschen der blau markierten Klausel



(b) 1. Schritt: *Clause Deletion Flag* setzen



(c) 2. Schritt: Verweis auf die Klausel entfernen

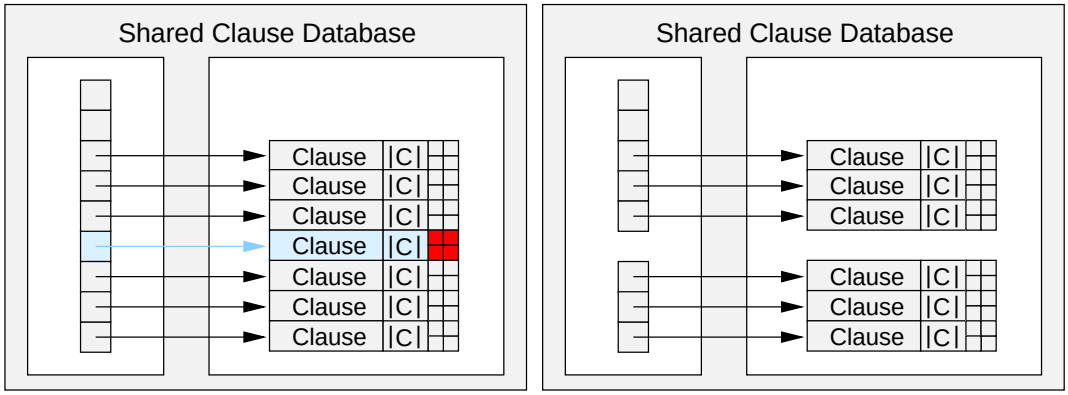


(d) 3: Schritt: WLRL-Eintrag entfernen

Abbildung 8.7: Löschen einer Konflikt-Klausel aus der Menge der von einem Thread berücksichtigen Klauseln

weise das *Clause Deletion Flag*, das den Status „gelöscht“ signalisiert, nur von dem damit assoziierten Thread genutzt wird und somit keine Gefahr von Daten-Inkonsistenzen besteht.

In wesentlich größeren Abständen als beim Entfernen von Konflikt-Klauseln auf der Ebene der Threads wird auch die globale Datenbank von nicht mehr benötigten Einträgen, also Klauseln, bei denen alle Threads das Flag für den Status „gelöscht“ gesetzt haben, befreit. Abbildung 8.8 stellt dies exemplarisch für eine Klausel, die von keinem der vier an der Suche nach einer erfüllenden Belegung beteiligten Threads mehr berücksichtigt wird, dar: alle vier *Clause Deletion Flags* sind gesetzt (rote Kästchen). Die Durchführung derartiger Lösch-Operationen erfordert eine komplette Sperrung der Klauseldatenbank, weshalb dies nur sehr selten durchgeführt wird. Das Problem liegt dabei nicht im eigentlichen Entfernen der Klauseln begründet, sondern in der Tatsache, dass auch hier die entstandenen Leerstellen durch ein Aufrücken der nachfolgenden Klauseln entfernt werden. Ein zeitgleicher Zugriff eines Threads auf die Datenbank, unabhängig ob lesend oder schreibend, führt in dieser Situation zwangsläufig zu Fehlern.



(a) Vor dem Löschen einer Klausel (b) Nach dem Löschen einer Klausel

Abbildung 8.8: Löschen einer Konflikt-Klausel aus der *Shared Clause Database*

8.3.5 Neustarts

Wie am Ende des vierten Kapitels angedeutet wurde, handelt es sich bei dem Konzept der Neustarts um ein probates Mittel, einen SAT-Algorithmus aus Bereichen des durch die CNF-Formel aufgespannten Suchraums, die aller Voraussicht nach nicht zu einer erfüllenden Belegung führen, herauszuführen. Vereinfacht ausgedrückt beruht die Durchführung von Neustarts auf dem Argument, dass mit der Dauer, die ein SAT-Algorithmus erfolglos nach einer erfüllenden Belegung sucht, die Wahrscheinlichkeit steigt, dass Entscheidungen auf den ersten Entscheidungsebenen bereits „ungünstig“ gewählt wurden und das Bestim-

men eines Modells verhindern.

Bei einem Neustart werden Zuweisungen auf früheren Entscheidungsebenen geändert, indem der Suchprozess an der aktuellen Stelle gestoppt, mit Ausnahme der Zuweisungen auf Decision Level 0 die komplette Variablenbelegung zurückgenommen und die Suche nach einer erfüllenden Belegung auf Decision Level 1 neu gestartet wird. Nicht geändert werden bei diesem Vorgang die Aktivitäten der einzelnen Variablen, somit besteht eine gute Chance, dass die auf Decision Level 1 gewählte Entscheidungsvariable nicht identisch ist mit der ehemaligen Decision Variable des ersten Levels.

In MiraXT wird sowohl im sequentiellen als auch im parallelen Betriebsmodus von Neustarts Gebrauch gemacht, wobei das Intervall zwischen zwei Neustarts kontinuierlich ansteigt, um identische Folgen von Variablenzuweisungen zwischen zwei derartigen Operationen zu verhindern. Die Anwendung im parallelen Szenario ist möglich, da die von den verschiedenen Threads bearbeiteten Teilprobleme beziehungsweise die Zuweisungen des entsprechenden Guiding Path stets auf Decision Level 0 verankert sind. Das bedeutet, dass es einem Thread selbst bei einem Neustart nicht möglich ist, das ihm ursprünglich zugewiesene Teilproblem zu verlassen und einen Bereich zu analysieren, der von einem anderen Thread bearbeitet wird. Andernfalls wäre eine Partitionierung des Suchraums in disjunkte Teilbereiche nicht gewährleistet.

8.4 Experimentelle Ergebnisse

In diesem Abschnitt werden die von MiraXT im sequentiellen Betrieb mit einem Thread und im parallelen Betrieb mit zwei und vier Threads erzielten Ergebnisse dargestellt. Insgesamt wurden für die Durchführung der Experimente drei Rechnerkonfigurationen herangezogen, die jeweils mit einer Debian-Distribution von Linux und aktuellen Versionen von *gcc* und *g++* ausgestattet sind:

– *Dual-Core AMD Opteron 280 Doppelprozessorsystem*: Abbildung 8.9 deutet die Konfiguration dieser Hardware-Plattform an. Zum Einsatz kommen zwei Prozessoren vom Typ *Dual-Core AMD Opteron 280* [5], die bei 2,4 GHz Taktfrequenz betrieben werden. Jeder Prozessor ist über den integrierten *RAM-Controller* direkt mit 2 GB Hauptspeicher verbunden und besteht aus zwei CPU-Kernen, die über einen jeweils eigenen L2-Cache mit 1 MB Kapazität verfügen. Mittels so genannter *Hyper Transport Links* in Kombination mit einem *Switch* (ähnlich der Switch-Matrix des Multiprozessorsystems) ist es einem Prozessor möglich, auf den Speicher des jeweils anderen Prozessors zuzugreifen. Selbstverständlich ist ein solcher Zugriff zeitaufwendiger als ein Speicherzugriff auf den eigenen „lokalen“ Speicher. Durch eine geeignete Programmierung kann gewährleistet werden, dass die Threads nach Möglichkeit nur den Speicher verwenden, der an den Prozessor angeschlossen ist, auf dem auch der entsprechende Thread ausgeführt wird.

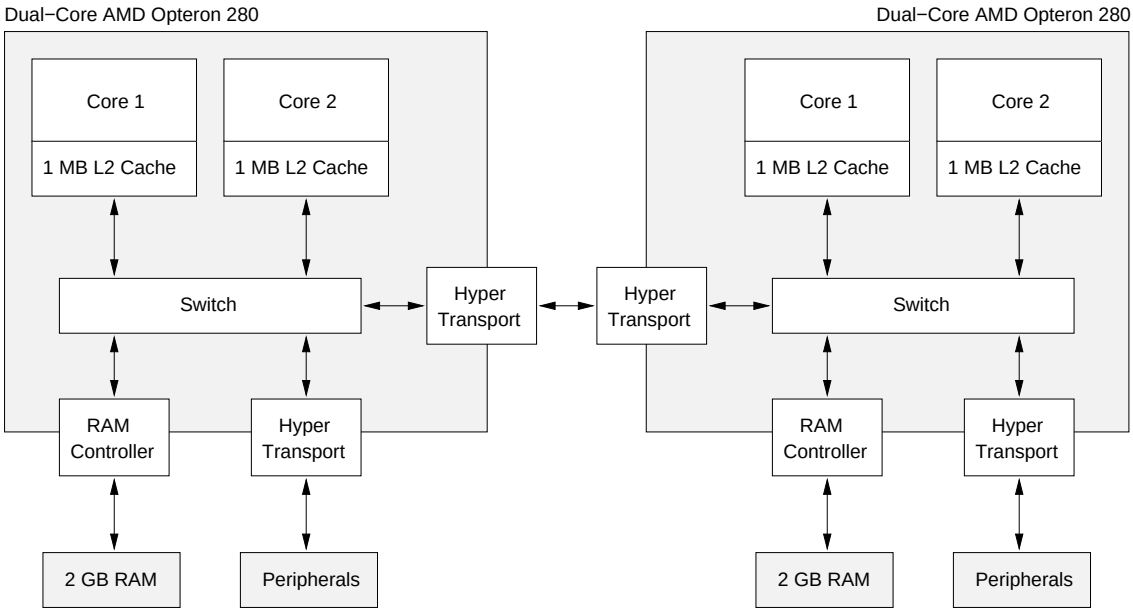


Abbildung 8.9: Dual-Core AMD Opteron 280 Doppelprozessorsystem [4, 6]

– *Intel Core 2 Duo T7200*: Bei dieser Konfiguration handelt es sich um ein System, das mittlerweile sowohl im privaten als auch im beruflichen Umfeld weit verbreitet ist. Den Kern bildet ein *Intel Core 2 Duo T7200* Prozessor [57], der mit 2,0 GHz Taktfrequenz betrieben wird, zwei CPU-Kerne beinhaltet und über insgesamt 4 MB L2-Cache verfügt. In der hier genutzten Variante verfügt der entsprechende Rechner über 2 GB Hauptspeicher. In Abbildung 8.10 ist die Anbindung von Speicher und Peripherie an den Prozessor schematisch dargestellt.

  Eine Besonderheit gegenüber dem *Dual-Core AMD Opteron 280* Prozessor liegt in der Anbindung des Cache-Speichers an die beiden CPU-Kerne. Im Gegensatz zum AMD-Prozessor, bei dem jeder CPU-Kern über einen eigenen L2-Cache mit 1 MB Kapazität verfügt, handelt es sich beim L2-Cache des *Intel Core 2 Duo T7200* Prozessors um einen so genannten *Intel Advanced Smart Cache* [116], der von beiden CPU-Kernen gleichermaßen genutzt werden kann und dynamisch unter diesen aufgeteilt wird.

– *Intel Pentium 4 HTT*: Der *Intel Pentium 4* Prozessor mit *Hyper-Threading Technology* (HTT) kann als ein Vorläufer der heutigen Dual-Core Technik angesehen werden. Auch bei diesem Prozessortyp werden dem Benutzer zwei CPU-Kerne suggeriert, allerdings sind diese im Gegensatz zu den anderen hier eingesetzten Prozessoren nicht vollständig doppelt vorhanden. Stattdessen sind lediglich all diejenigen Komponenten in zweifacher Ausführung auf dem Chip integriert, die für den so genannten *Architecture State* notwendig sind und den aktuellen Zustand einer CPU eindeu-

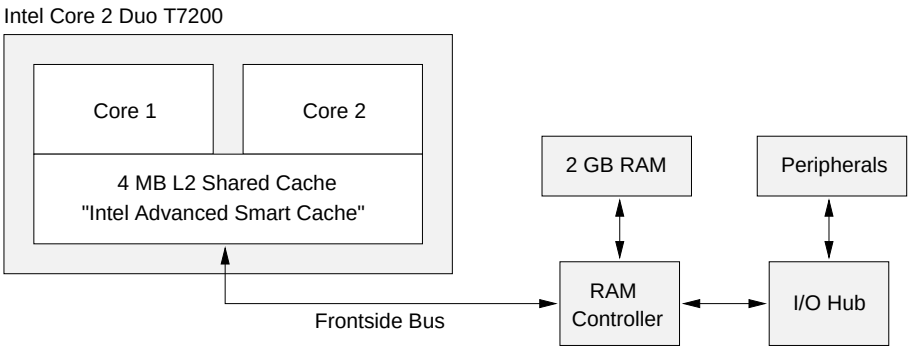


Abbildung 8.10: Intel Core 2 Duo T7200 [55, 58]

tig beschreiben [16, 81]. Zu diesen Komponenten, die weniger als 5% der gesamten Schaltung ausmachen, gehören beispielsweise die *Control Register*, die *General Purpose Register* und der *Interrupt Controller*. Alle anderen Einheiten sind nur einfach vorhanden und müssen von den zwei CPU-Kernen geteilt werden. Man spricht daher auch von zwei *logischen* CPU-Kernen. Das für die Durchführung der Experimente eingesetzte Modell wird bei einer Taktfrequenz von 3,2 GHz betrieben, verfügt über 1 MB L2-Cache und kann auf 1 GB Hauptspeicher zugreifen [54].

Als CNF-Formeln wurden 347 Probleminstanzen aus zwei Kategorien ausgewählt, die in den beiden letzten Jahren auch im Rahmen der *SAT Competition* [110] verwendet wurden und als anerkannt schwierig gelten:

- *SAT Race 2006*: 100 zumeist aus dem industriellen Umfeld stammende Instanzen, die für den Wettbewerb des Jahres 2006 ausgewählt wurden. In Übereinstimmung mit den Regeln der *SAT Competition 2006* beträgt das Zeitlimit, das für die Bearbeitung der einzelnen Instanzen zur Verfügung steht, jeweils 900 Sekunden. Erfüllbare und unerfüllbare CNF-Formeln halten sich in etwa die Waage.

- *SAT 2007 Industrial*: 247 Instanzen, die für den Wettbewerb des Jahres 2007 in der Kategorie *Industrial* eingesetzt wurden, somit unter anderem auch Problemstellungen aus den in Kapitel 3 angedeuteten Anwendungsgebieten enthalten. Pro Instanz dieser Problemklasse beträgt das Zeitlimit analog zur finalen Evaluierungsrunde der *SAT Competition 2007* 10000 Sekunden. Wiederum sind erfüllbare und unerfüllbare CNF-Formeln in etwa gleich stark vertreten.

Für eine aussagekräftige Analyse des Potenzials von MiraXT wurden alle Probleminstanzen ebenfalls mit den aktuellsten Versionen von MiniSat2 [111], PicoSAT [112] und RSat [113] gelöst. Alle drei sequentiellen SAT-Algorithmen gehören zu den leistungsstärksten Vertretern der *SAT Competition 2007*. Man beachte in diesem Zusammenhang, dass die

eingesetzte Variante von RSat nicht identisch ist mit der Version, die bei der *SAT Competition 2007* zum Einsatz kam. Bei letzterer wurde in Form eines Skripts die aus SatELite [31] bekannte Preprocessing-Einheit zur Vorverarbeitung der jeweiligen Probleminstanz vorgeschaltet. Zwar ist diese Variante auch auf der Homepage der Autoren verfügbar, lässt sich aber auf keinem der hier genutzten Hardware-Systeme fehlerfrei übersetzen. Daher bezieht sich RSat im Folgenden immer auf den ursprünglichen SAT-Algorithmus ohne Vorverarbeitung. Weiterhin wurde auf vergleichende Messungen mit anderen parallelen SAT-Algorithmen verzichtet, da es zu MiraXT derzeit keine ähnlich leistungsfähigen Alternativen gibt, ein Punkt, der im Übrigen auch auf die in Kapitel 5 vorgestellten parallelen SAT-Algorithmen zutrifft.

Bei den CNF-Formeln der Kategorie *SAT Race 2006* geben alle im Folgenden für MiraXT im parallelen Modus mit mehreren Threads angegebenen Werte den Mittelwert dreier Durchläufe wieder. Analog zu PIChaff wird das Vorgehen der einzelnen Threads im parallelen Betrieb von MiraXT maßgeblich dadurch beeinflusst, wann welche Konflikt-Klauseln zur Verfügung stehen beziehungsweise welcher Thread ein Teilproblem an welchen inaktiven Thread abgibt. Die maximale Abweichung der drei Durchläufe zu den nachfolgend angegebenen Mittelwerten variiert je nach Rechnerkonfiguration und bewegt sich in der Größenordnung von 10–15%. Aufgrund des hohen Zeitbedarfs wurde bei den *SAT 2007 Industrial* Instanzen auf die Durchführung mehrerer Läufe verzichtet.

8.4.1 AMD Opteron 280 Doppelprozessorsystem

In Tabelle 8.1 sind die von RSat, MiniSat2, PicoSAT und MiraXT bei der Lösung der *SAT Race 2006* Instanzen erzielten Ergebnisse aufgeführt. Es zeigt sich, dass gegenüber PicoSAT bereits die sequentielle Variante von MiraXT einen Laufzeitvorteil von 15% bietet. Zudem ist diese Variante von MiraXT in der Lage, eine Probleminstanz mehr zu lösen als PicoSAT, das in dieser Versuchsreihe schnellste der drei Verfahren RSat, MiniSat2 und PicoSAT. Durch den Einsatz von zwei Threads gewinnt MiraXT weiter an Performance (17% schneller als die sequentielle Version von MiraXT; 34% schneller als PicoSAT) und kann zudem mit im Durchschnitt 80,67 Probleminstanzen weitere CNF-Formeln innerhalb des vorgegebenen Zeitlimits erfolgreich bearbeiten. Der Kommawert ergibt sich aus der Tatsache, dass einige der Probleminstanzen im parallelen Betriebsmodus eine Laufzeit von annähernd 900 Sekunden benötigen und daher aufgrund der zuvor angedeuteten Abweichungen zwischen verschiedenen Durchläufen nicht immer gelöst werden können. Mit vier Threads steigert sich die Leistung noch einmal, gegenüber PicoSAT ist MiraXT 71% schneller und löst mit 85 Instanzen mit Abstand die meisten Probleme aller miteinander verglichenen SAT-Algorithmen.

Die in der letzten Spalte der Tabelle angegebene Anzahl an Konflikt-Klauseln, die von allen Threads von MiraXT im Mittel in der Sekunde generiert werden (abgekürzt durch #CC/s), kann als Maß für die verrichtete Arbeit angesehen werden. Neben der Laufzeit

SAT-Algorithmus	Gesamtlaufzeit [s]	Gelöste Instanzen	#CC/s
RSat	40209,10	68	—
MiniSat2	41631,15	70	—
PicoSAT	37033,29	77	—
MiraXT, 1 Thread	32264,09	78	4149,88
MiraXT, 2 Threads	27651,93	80,67	7136,50
MiraXT, 4 Threads	**21674,50**	**85**	13118,85

Tabelle 8.1: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT und MiraXT. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	11353,62	1,00
MiraXT, 2 Threads	6902,29	1,64
MiraXT, 4 Threads	4418,97	2,57

Tabelle 8.2: Beschleunigung von MiraXT durch den Einsatz von zwei und vier Threads. Problemklasse: *SAT Race 2006* (beschränkt auf die Instanzen, die von allen Konfigurationen von MiraXT gelöst werden konnten), Zeitlimit: 900 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

und der Anzahl der gelösten Instanzen ist dies ein weiteres Indiz für eine effiziente Implementierung. Durch den Einsatz eines zweiten Threads erhöht sich die Anzahl der im Mittel pro Sekunde generierten Konflikt-Klauseln um etwa 72%, selbst beim Übergang von zwei auf vier Threads steigt der Wert um beachtliche 84%. Diese Zahlen belegen zweifelsfrei, dass sich die gemeinsame Klauseldatenbank und damit die einzige Datenstruktur, bei der vermehrt *Locks* zur Vergabe von exklusiven Schreibrechten benötigt werden, im parallelen Modus von MiraXT nicht zum „Flaschenhals“ entwickelt.

Bezüglich der in Tabelle 8.1 erzielten Ergebnisse gibt Tabelle 8.2 die Beschleunigung wieder, die sich durch den Einsatz eines zweiten und vierten Threads im Vergleich zum sequentiellen Modus von MiraXT einstellt. Zur Ermittlung des korrekten Speedup-Wertes ist die angegebene Summe der Laufzeiten auf diejenigen Instanzen beschränkt worden, die von allen Konfigurationen von MiraXT innerhalb des Zeitlimits von 900 Sekunden gelöst werden konnten. Insgesamt wurde ein Speedup-Faktor von 1,64 (zwei Threads) beziehungsweise 2,57 (vier Threads) erreicht. Vor dem Hintergrund des geringen Zeitlimits ist das ein sehr guter Wert, der in dieser Größenordnung in [71] auch für andere Probleminstanzen ermit-

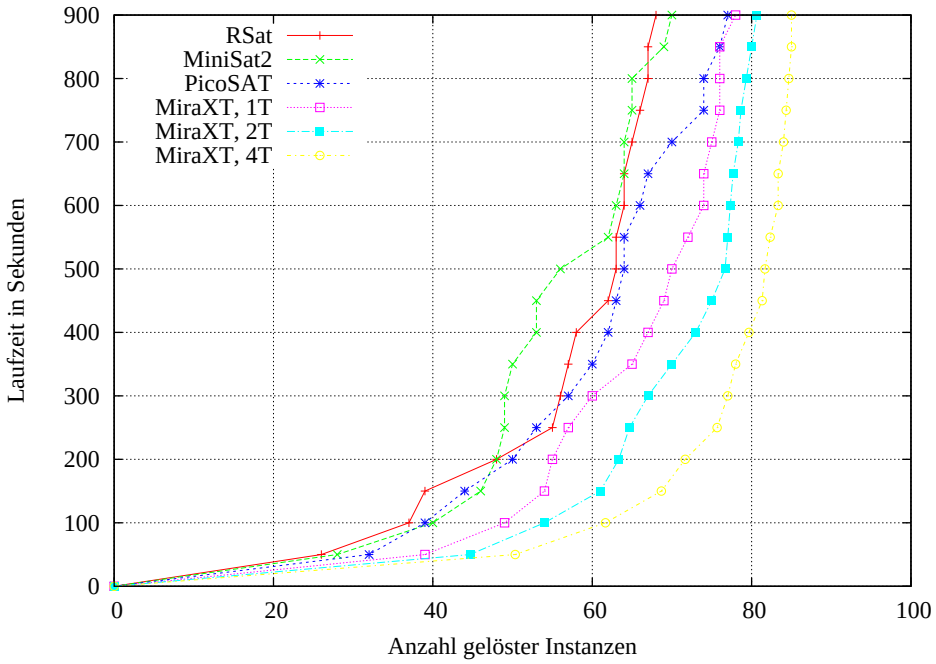


Abbildung 8.11: Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

telt werden konnte. Wie sich bei der Kategorie *SAT 2007 Industrial* zeigen wird, kann MiraXT erst bei schwierigeren Problemstellungen und einem deutlich höheren Zeitlimit die Vorteile des parallelen Betriebsmodus voll ausspielen. Zur Verdeutlichung sei die von MiraXT mit einem Thread benötigte Gesamtlaufzeit von 32264,09 Sekunden zum Lösen aller 100 CNF-Formeln der Problemklasse *SAT Race 2006* angeführt. Zieht man für jede der 22 nicht gelösten Instanzen jeweils 900 Sekunden ab, verbleibt als Laufzeit ein Wert von 12464,09 Sekunden, welche die sequentielle Variante von MiraXT zum Lösen der 78 restlichen Probleme benötigt hat. Das heißt, dass jede der 78 Instanzen im Mittel innerhalb von knapp 160 Sekunden erfolgreich bearbeitet werden konnte, ein Wert, der offensichtlich nicht viel Spielraum für eine lineare Beschleunigung insbesondere beim Einsatz von vier Threads lässt. Abbildung 8.11 stellt die Anzahl der von den verschiedenen Algorithmen gelösten Instanzen der Laufzeit gegenüber und bestätigt die erhaltenen Resultate.

Tabelle 8.3 zeigt den Einfluss der Vorverarbeitung und des Austauschs von Konflikt-Klauseln auf das Laufzeitverhalten von MiraXT beim Lösen der Probleme der Kategorie *SAT Race 2006* beispielhaft für die Variante mit vier Threads. Die Abkürzungen *PP* und

MiraXT-Konfiguration	Gesamtlaufzeit [s]	Gelöste Instanzen
4 Threads, ohne PP	28827,26	79
4 Threads, ohne TKS	26248,80	81
4 Threads, mit PP und TKS	21674,50	85

Tabelle 8.3: Experimentelle Ergebnisse verschiedener Konfigurationen von MiraXT. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

TKS stehen stellvertretend für *Preprocessing* beziehungsweise *Thread Knowledge Sharing* (bezeichnet den Austausch von Konflikt-Klauseln zwischen den Threads).

Deutlich zu erkennen ist, dass ein Deaktivieren der Preprocessing-Einheit für erhebliche Nachteile sorgt: die Laufzeit erhöht sich um 33% (28827,26 gegenüber 21674,50 Sekunden), zudem können sechs Instanzen weniger innerhalb von 900 Sekunden gelöst werden. Ebenso ist der Austausch von Konflikt-Klauseln zwischen den Threads wichtig, um das Potenzial von MiraXT maximal auszuschöpfen. Ohne diesen Austausch, jeder Thread bezieht dann nur die von ihm selbst generierten Konflikt-Klauseln in den eigenen Suchprozess ein, sinkt die Performance von MiraXT in Bezug zur Laufzeit um 21%, ebenfalls sinkt die Anzahl erfolgreich gelöster CNF-Formeln um 4. Ausgehend von diesen Ergebnissen werden nachfolgend nur noch Konfigurationen von MiraXT betrachtet, bei denen sowohl die Vorverarbeitung der Probleminstanz als auch der Austausch von Konflikt-Klauseln aktiviert ist.

Die für die Problemklasse *SAT 2007 Industrial* von RSat, MiniSat2, PicoSAT und MiraXT erzielten Ergebnisse sind in Tabelle 8.4 angegeben. Dazu passend ist in Abbildung 8.12 die Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Bezug zur jeweils benötigten Laufzeit dargestellt. Es sei darauf hingewiesen, dass in dieser Versuchsreihe aufgrund des hohen Zeitaufwands auch die parallele Variante von MiraXT nur einmal ausgeführt wurde.

Die Resultate sind beeindruckend, unabhängig von der Konfiguration setzt sich MiraXT von den anderen SAT-Algorithmen ab und ist gegenüber PicoSAT, dem schnellsten der drei anderen Verfahren um 11% (mit einem Thread), 33% (mit zwei Threads) beziehungsweise um 42% (mit vier Threads) schneller. Weiterhin ist es MiraXT möglich, signifikant mehr Probleminstanzen zu lösen, wobei es im parallelen Modus mit zwei und vier Threads sein gesamtes Potenzial ausspielen und insgesamt 177 beziehungsweise 180 CNF-Formeln lösen kann. Das ungleich höhere Zeitlimit von 10000 Sekunden gegenüber 900 Sekunden bei der Problemklasse *SAT Race 2006* macht sich auch beim Speedup bemerkbar: zwei Threads sind um den Faktor 2,47 schneller als die sequentielle Variante von MiraXT, bei

SAT-Algorithmus	Gesamtlaufzeit [s]	Gelöste Instanzen
RSat	1293297,72	136
MiniSat2	1199874,43	143
PicoSAT	1110696,47	152
MiraXT, 1 Thread	999154,42	162
MiraXT, 2 Threads	835851,65	177
MiraXT, 4 Threads	**783190,38**	**180**

Tabelle 8.4: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT und MiraXT. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	132618,80	1,00
MiraXT, 2 Threads	53734,05	2,47
MiraXT, 4 Threads	37769,97	3,51

Tabelle 8.5: Beschleunigung von MiraXT durch den Einsatz von zwei und vier Threads. Problemklasse: *SAT 2007 Industrial* (beschränkt auf die Instanzen, die von allen Konfigurationen von MiraXT gelöst werden konnten), Zeitlimit: 10000 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

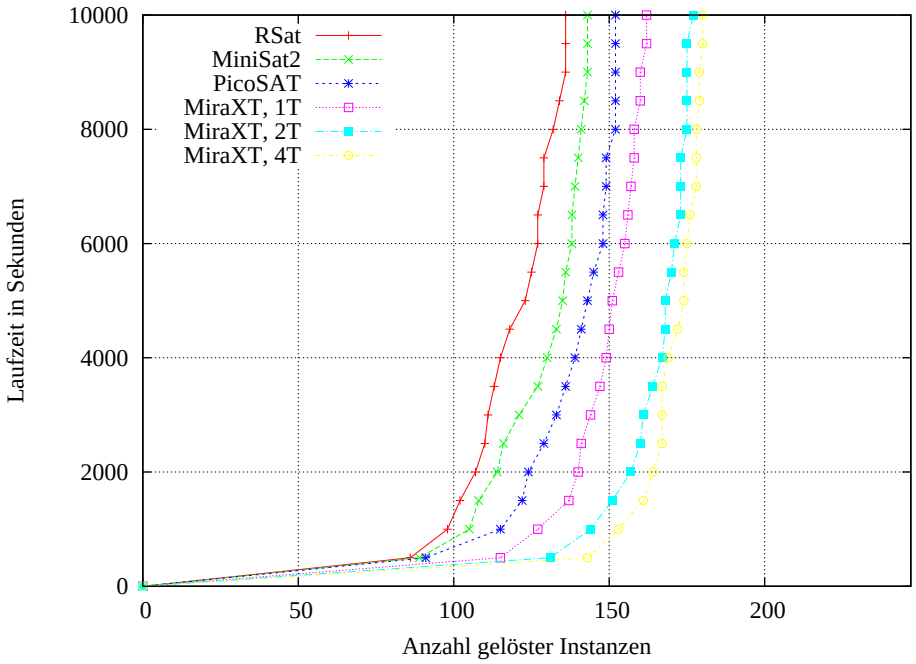


Abbildung 8.12: Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattform: *AMD Opteron 280 Doppelprozessorsystem*.

vier Threads erreicht die Beschleunigung einen Wert von 3,51. Hier kann von einem linearen Speedup gesprochen werden kann (siehe Tabelle 8.5). In der Variante mit vier Threads kann die leicht nachlassende Beschleunigung dadurch begründet werden, dass auf jedem CPU-Kern der beiden *Dual-Core AMD Opteron 280* Prozessoren ein Thread ausgeführt wird, was zur Folge hat, dass sich je zwei Threads den lokal an den entsprechenden Prozessor angeschlossenen Speicher teilen (mit allen damit verbundenen Wartezeiten beim Zugriff auf den Speicher). Bei der MiraXT-Variante mit zwei Threads werden diese vom Betriebssystem auf die beiden *AMD Opteron* Prozessoren verteilt, so dass jeder Thread, abgesehen von Zugriffen des jeweils anderen Threads über die *Hyper Transport Links*, nahezu alleinigen Zugriff auf seinen lokalen Speicher besitzt.

8.4.2 Intel Core 2 Duo T7200

Die zweite Versuchsreihe wurde auf einem Laptop, ausgestattet mit einem *Intel Core 2 Duo T7200* Prozessor, der über 2 GB Hauptspeicher verfügt, durchgeführt. Tabelle 8.6 stellt die Resultate von RSat, MiniSat2, PicoSAT und MiraXT für die Problemklasse *SAT*

SAT-Algorithmus	Gesamtlaufzeit [s]	Gelöste Instanzen	#CC/s
RSat	30488,02	81	—
MiniSat2	35815,64	71	—
PicoSAT	29740,01	81	—
MiraXT, 1 Thread	27715,77	81	6955,30
MiraXT, 2 Threads	**24604,99**	**81,33**	10275,65

Tabelle 8.6: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT und MiraXT. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	9486,39	1,00
MiraXT, 2 Threads	6402,79	1,48

Tabelle 8.7: Beschleunigung von MiraXT durch den Einsatz von zwei Threads. Problemklasse: *SAT Race 2006* (beschränkt auf die Instanzen, die von beiden Konfigurationen von MiraXT gelöst werden konnten), Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

Race 2006 gegenüber. Zunächst fällt auf, dass es sich bei diesem Prozessortyp um den schnellsten Prozessor innerhalb der drei getesteten Hardware-Systeme handelt. Trotz der ungleich geringeren Taktfrequenz im Vergleich zu den AMD-Prozessoren sind alle auf einem *Intel Core 2 Duo T7200* Prozessor ausgeführten sequentiellen SAT-Algorithmen deutlich schneller (der *Intel Pentium 4 HTT* Prozessor sei an dieser Stelle aufgrund der älteren Bauart von einem Vergleich ausgenommen).

Auch bei dieser Hardware-Konfiguration benötigen die beiden MiraXT-Varianten die geringste Laufzeit und sind 7% (mit einem Thread) beziehungsweise 21% (mit zwei Threads) schneller als PicoSAT, der leistungsstärkste der drei anderen SAT-Algorithmen. Zugleich wird aber deutlich, dass der Performance-Gewinn durch den zweiten Thread, ungeachtet des ebenfalls guten Wertes die Anzahl der pro Sekunde generierten Konflikt-Klauseln betreffend, nicht so groß ausfällt wie beim AMD-Doppelprozessorsystem, was auch Abbildung 8.13 belegt. Aus diesem Grund fällt die Beschleunigung, die durch den Einsatz eines zweiten Threads erzielt wird, mit einem Faktor von 1,48 geringer aus als beim AMD-System (siehe Tabelle 8.7).

Die im Vergleich zum *AMD Opteron 280 Doppelprozessorsystem* geringfügig schlechtere

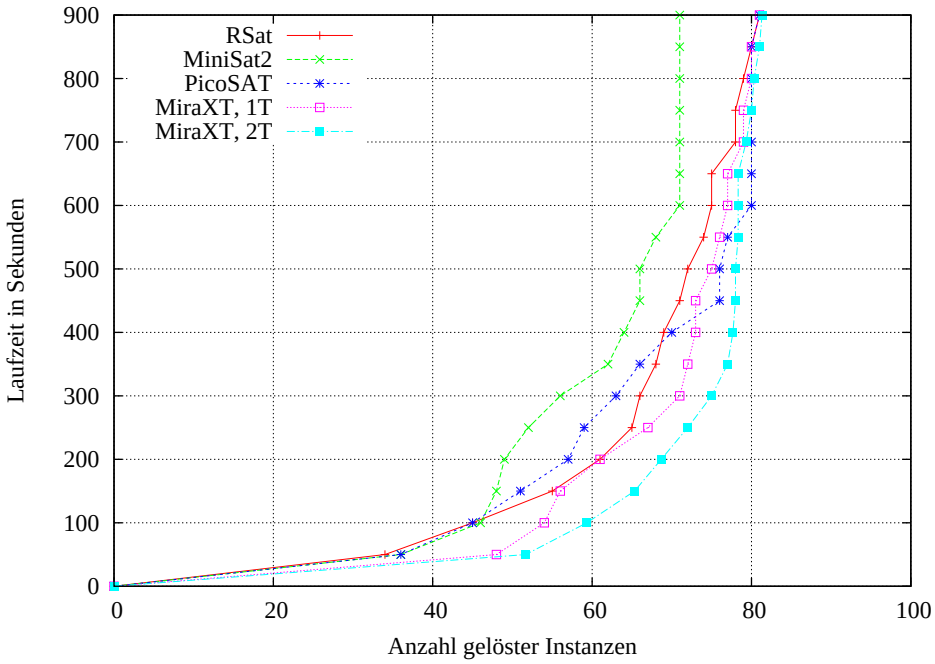


Abbildung 8.13: Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

Skalierung beim Übergang von einem auf zwei Threads lässt sich darauf zurückführen, dass die Threads auf den beiden CPU-Kernen des *Intel Core 2 Duo T7200* Prozessors ausgeführt werden und sich Adress- und Datenleitungen beim Zugriff auf den Speicher teilen müssen. Dies führt zwangsläufig zu Wartezeiten. Beim *AMD Opteron 280 Doppelprozessorsystem* werden bei der Variante mit zwei Threads diese hingegen auf die beiden zur Verfügung stehenden Prozessoren verteilt und besitzen somit nahezu exklusiven Zugriff auf den lokal an den jeweiligen Prozessor angebundenen Speicher.

Auch auf diesem Rechner wurden die Instanzen der Kategorie *SAT 2007 Industrial* von allen betrachteten SAT-Algorithmen bearbeitet. Die dabei ermittelten Resultate sind in Tabelle 8.8 aufgelistet. Erneut ist MiraXT sowohl mit einem als auch mit zwei Threads den anderen SAT-Verfahren überlegen und kann mit 168 beziehungsweise 174 Instanzen deutlich mehr Problemstellungen erfolgreich bearbeiten als die drei restlichen Verfahren (siehe auch Abbildung 8.14). Die in Tabelle 8.9 mit einem Wert von 2,07 angegebene Beschleunigung belegt, dass MiraXT bei schwierigen Instanzen auch bei der in dieser Versuchsreihe eingesetzten Hardware-Plattform in der Lage ist, einen linearen Speedup zu erzielen.

SAT-Algorithmus	Gesamtlaufzeit [s]	Gelöste Instanzen
RSat	1123558,97	150
MiniSat2	1174159,73	145
PicoSAT	1028860,82	158
MiraXT, 1 Thread	907601,00	168
MiraXT, 2 Threads	**797674,91**	**174**

Tabelle 8.8: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT und MiraXT. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	111255,83	1,00
MiraXT, 2 Threads	53629,50	2,07

Tabelle 8.9: Beschleunigung von MiraXT durch den Einsatz von zwei Threads. Problemklasse: *SAT 2007 Industrial* (beschränkt auf die Instanzen, die von beiden Konfigurationen von MiraXT gelöst werden konnten), Zeitlimit: 10000 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

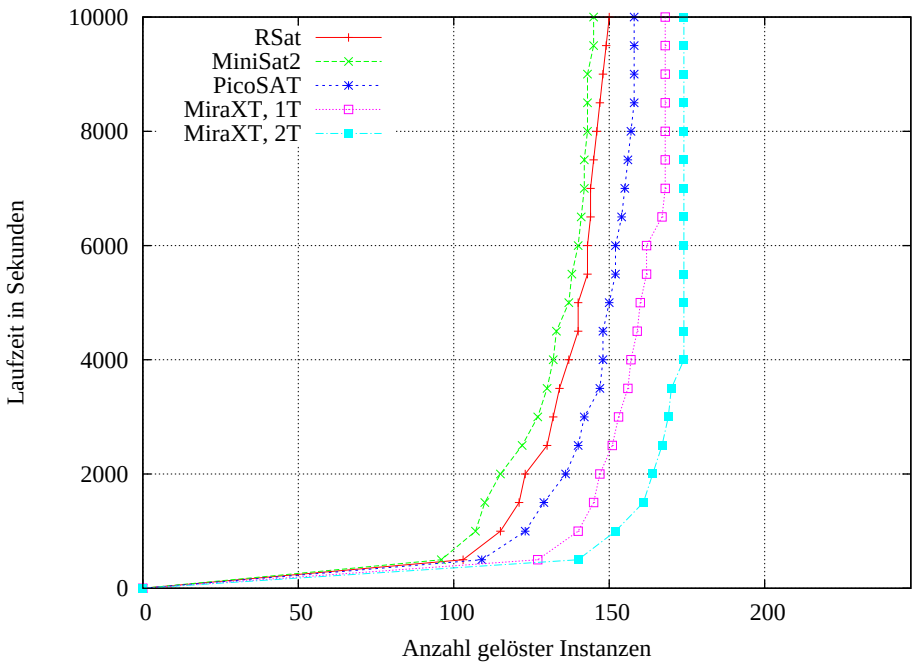


Abbildung 8.14: Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattform: *Intel Core 2 Duo T7200*.

8.4.3 Intel Pentium 4 HTT

Wie zuvor angedeutet, handelt es sich bei einem *Intel Pentium 4* Prozessor mit *Hyper-Threading Technology* (HTT) nicht um einen Dual-Core Prozessor im eigentlichen Sinne, denn statt zweier „vollständig“ ausgestatteter CPU-Kerne sind lediglich alle Register, die für das Speichern des CPU-Zustands benötigt werden, doppelt vorhanden. Alle anderen Komponenten werden von den beiden *logischen* CPU-Kernen gemeinsam genutzt. Dennoch wurden auch für diese Hardware-Konfiguration Experimente durchgeführt, die demonstrieren, dass selbst bei dieser Prozessor-Architektur ein paralleler SAT-Algorithmus Vorteile gegenüber einem sequentiellen Verfahren bietet. Erwartungsgemäß fällt die Laufzeiteinsparung im Vergleich zu den beiden anderen Systemen aber wesentlich geringer aus. Eine ähnliche Versuchsreihe wurde in [94] für eine Vorgängerversion von MiraXT dokumentiert, welche die im Folgenden diskutierten Resultate bestätigt.

Die Ergebnisse der verschiedenen SAT-Algorithmen für die Problemklasse *SAT Race 2006* sind Tabelle 8.10 zu entnehmen. Wie auch bei den anderen Rechnerkonfigurationen setzt MiraXT unter Ausnutzung aller Hardware-Ressourcen (mit zwei Threads) in Bezug zur

SAT-Algorithmus	Gesamtlaufzeit [s]	Gelöste Instanzen	#CC/s
RSat	37392,15	74	—
MiniSat2	42276,01	70	—
PicoSAT	39208,32	76	—
MiraXT, 1 Thread	35257,40	75	3579,32
MiraXT, 2 Threads	**34307,25**	**76,67**	3588,33

Tabelle 8.10: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT und MiraXT. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Pentium 4 HTT*.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	10261,43	1,00
MiraXT, 2 Threads	8945,19	1,15

Tabelle 8.11: Beschleunigung von MiraXT durch den Einsatz von zwei Threads. Problemklasse: *SAT Race 2006* (beschränkt auf die Instanzen, die von beiden Konfigurationen von MiraXT gelöst werden konnten), Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Pentium 4 HTT*.

benötigten Laufzeit den Maßstab. Bezüglich der gelösten Instanzen sind mit Ausnahme von MiniSat2 alle Verfahren etwa auf dem gleichen Niveau. Abbildung 8.15 verdeutlicht dies. Der Speedup, der bei MiraXT mit zwei Threads gegenüber der sequentiellen Variante erreicht werden kann, fällt mit 15% weitaus geringer aus, als dies bei den anderen Hardware-Umgebungen der Fall ist (siehe Tabelle 8.11). Es bestätigt sich, dass die Performance von MiraXT im parallelen Modus dadurch, dass sich die zwei logischen CPU-Kerne alle relevanten Komponenten des *Intel Pentium 4 HTT* Prozessors teilen müssen, negativ beeinflusst wird. Ein starkes Indiz hierfür ist die in Tabelle 8.10 in der letzten Spalte (#CC/s) angegebene Anzahl an Konflikt-Klauseln, die im Mittel pro Sekunde generiert werden: die Werte der beiden MiraXT-Varianten sind fast identisch.

Aufgrund der eingeschränkten Kapazität des Hauptspeichers von 1 GB und der begrenzten Rechenleistung des *Intel Pentium 4 HTT* Prozessors wurde auf die Bearbeitung der *SAT 2007 Industrial* Probleminstanzen verzichtet.

8.4.4 Abschlussbemerkung

Als Fazit der durchgeführten Experimente kann festgehalten werden, dass MiraXT bereits in der sequentiellen Variante mit nur einem Thread auf allen drei Hardware-Konfigurationen

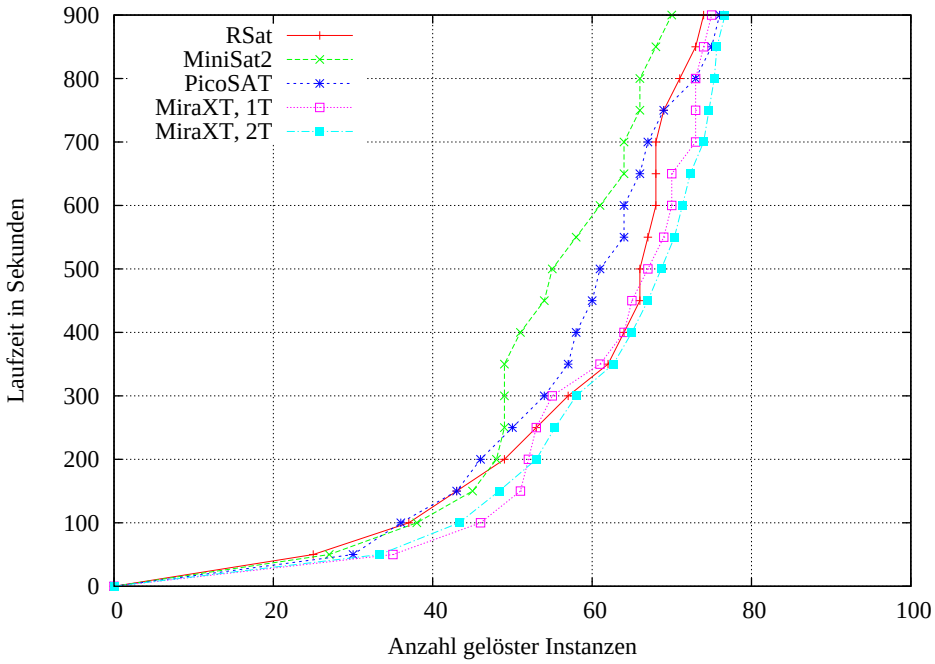


Abbildung 8.15: Anzahl der von RSat, MiniSat2, PicoSAT und MiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattform: *Intel Pentium 4 HTT*.

schneller ist als RSat, MiniSat2 und PicoSAT. Zusätzlich zur Reduktion der Laufzeit erhöht sich mit steigender Anzahl an parallel agierenden Threads die Zahl der gelösten Instanzen. Die Ergebnisse belegen, dass das Konzept, die Klauselmenge in Form einer einzigen, von allen Threads gemeinsam genutzten Datenstruktur zu realisieren, die Performance von MiraXT nicht negativ beeinflusst. Sie wirkt sich im Gegenteil positiv aus, da jeder Thread zu jedem Zeitpunkt den Zugriff auf alle verfügbaren und für sein Teilproblem relevanten Konflikt-Klauseln hat und anhand seines aktuellen Status entscheiden kann, welche er davon berücksichtigt, auch wenn diese ursprünglich von einem anderen Thread hergeleitet wurden.

Bei den verwendeten Hardware-Plattformen handelt es sich ausnahmslos um Systeme, die zumindest in ähnlicher Ausprägung an vielen Standorten, sei es im Umfeld einer Hochschule oder im Bereich der Industrie, eingesetzt werden. MiraXT dürfte daher aufgrund seines Leistungsvermögens für eine Vielzahl potenzieller Anwender von Interesse sein, bietet es doch insbesondere die Möglichkeit, alle Prozessoren beziehungsweise CPU-Kerne eines Rechners gewinnbringend in den Suchprozess einzubinden.

Abschließend sei erwähnt, dass die hier erzielten Ergebnisse nicht mit denen der *SAT Competition 2007* vergleichbar sind, bei der MiraXT mit zwei Threads in der Kategorie der industriellen Problemstellungen den fünften Platz von insgesamt 44 Wettbewerbern belegt hat. Zum Einen ist die hier beschriebene Variante gegenüber der Anfang 2007 eingereichten Version erheblich weiterentwickelt worden. Zum Anderen war die Art der Zeitmessung während der *SAT Competition 2007* von den Organisatoren unglücklich gewählt. Unter allen eingereichten Verfahren gab es mit MiraXT lediglich einen parallelen SAT-Algorithmus. Um eine gewisse Vergleichbarkeit zu den sequentiellen Ansätzen zu gewährleisten, wurde entschieden, im Fall von MiraXT die CPU-Zeiten beider Threads, die auf zwei Prozessoren eines *Intel Xeon Doppelprozessorsystems* ausgeführt wurden, zu addieren und diesen Wert als die benötigte Laufzeit anzunehmen. Die tatsächlich benötigte „Real-Zeit“ entspricht aber lediglich dem Maximum der beiden CPU-Zeiten, da die Threads parallel auf zwei verschiedenen Prozessoren agieren. Das bedeutet, dass die in [110] angegebenen Zeiten für MiraXT künstliche Werte darstellen, die keineswegs der Zeit vom Starten des Algorithmus bis zum Ende des Suchprozesses und der Ausgabe des Ergebnisses am Bildschirm entsprechen. Vor diesem Hintergrund ist der fünfte Platz als Erfolg zu werten, da aufgrund der Art der Zeitmessung eine bessere Platzierung nicht zu erreichen war. Mit dem in dieser Arbeit genutzten Setup, das die tatsächlich verstrichene Zeit zwischen Starten und Stoppen des Algorithmus misst, hätte sich MiraXT aller Wahrscheinlichkeit nach an die Spitze des Feldes der *SAT Competition 2007* gesetzt.

Kapitel 9

PaMiraXT

Im vorangegangenen Kapitel wurde mit MiraXT ein leistungsstarker, threadbasierter SAT-Algorithmus vorgestellt. Die experimentellen Ergebnisse haben eindrucksvoll belegt, dass bereits im sequentiellen Modus mit lediglich einem Thread MiraXT anderen *State-of-the-Art* Verfahren wie RSat, MiniSat2 und PicoSAT überlegen ist. MiraXT eignet sich somit einerseits für den Einsatz auf Rechnern mit so genannten *Single-Core* Prozessoren, die nur über eine CPU verfügen. Andererseits bietet MiraXT den entscheidenden Vorteil, dass bei Rechnern mit Dual- oder Multi-Core Prozessoren und auch Mehrprozessorsystemen die zusätzlich vorhandenen CPU-Kerne beziehungsweise Prozessoren, die bei sequentiellen SAT-Verfahren ansonsten ungenutzt bleiben, effizient mit in den Suchprozess eingebunden werden.

Da MiraXT auf einem Thread-Konzept aufbaut, ist eine Voraussetzung für den parallelen Betriebsmodus, dass alle Prozessoren, auf denen die einzelnen Threads ausgeführt werden sollen, auf einen gemeinsamen Speicher zugreifen können. Betrachtet man in diesem Zusammenhang herkömmliche Rechnernetzwerke, bei denen die einzelnen Rechner (Knoten des Netzwerks) über eine Ethernet-Verbindung miteinander verknüpft sind, so ist diese Bedingung nicht erfüllt. Jeder Knoten verwaltet seinen eigenen Speicher, der von den anderen Rechnern aus nicht zugreifbar ist. In einem derartigen Szenario ist MiraXT nur auf einem Knoten des Netzwerks ausführbar, ohne dass Prozessoren anderer Rechner in die Suche nach einer erfüllenden Belegung einbezogen werden könnten.

Mit PaMiraXT, einer Erweiterung von MiraXT, wird diese Einschränkung aufgehoben, indem MiraXT übergeordnet auf einer zweiten Ebene ein Master/Client-Modell realisiert wird. Das Grundprinzip lässt sich wie folgt skizzieren: auf jedem Rechner des Netzwerks, das für die Ausführung von PaMiraXT verwendet werden soll, wird MiraXT gestartet. Die Anzahl der auf dem entsprechenden Knoten verwendeten Threads richtet sich nach der Anzahl der „lokal“ auf diesem Rechner vorhandenen Prozessoren, MiraXT wird folglich entweder in der sequentiellen Variante mit nur einem Thread oder in der parallelen Variante mit mehreren parallel agierenden Threads gestartet. Insbesondere kann pro Netzwerkknoten eine unterschiedliche Konfiguration gewählt werden, so dass sich PaMiraXT optimal an die gegebene Hardware-Umgebung, üblicherweise bestehend aus Rechnern mit Single-Core und vermehrt auch Multi-Core Prozessoren, anpassen lässt. Die so gestarteten

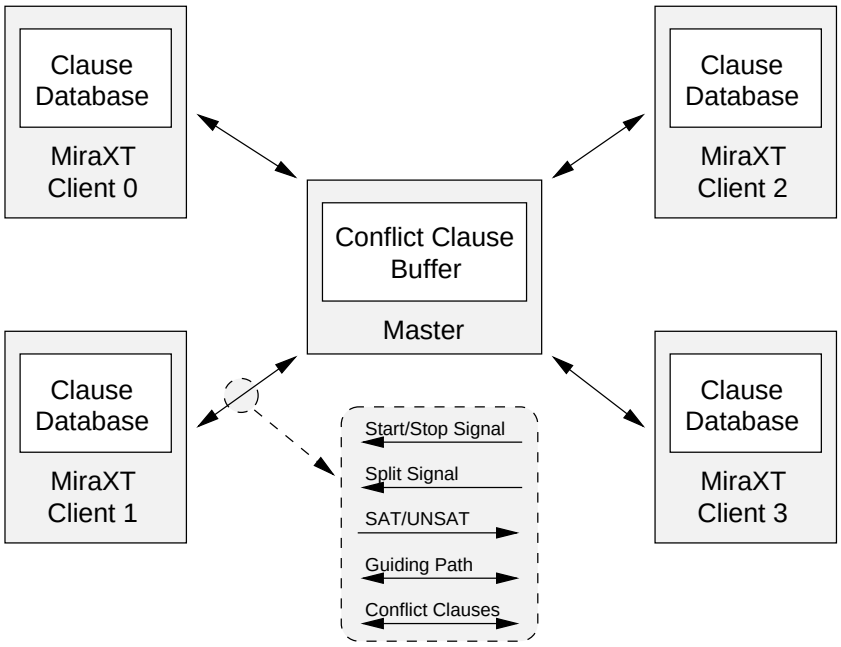


Abbildung 9.1: Design PaMiraXT

„Kopien“ von MiraXT werden als Clients eines Master/Client-Modells aufgefasst und unter der Regie eines separaten Master-Prozesses zu PaMiraXT zusammengefasst. Abbildung 9.1 illustriert das Konzept am Beispiel von vier Clients und dem sich als Mittler zwischen den Clients befindenden Master. Abgesehen von der Zuordnung der verschiedenen Funktionseinheiten zu expliziten Hardware-Komponenten und dem Fehlen der Switch-Matrix, sind auf dieser abstrakten Ebene deutliche Analogien zu PIChaff erkennbar. Die Kommunikation zwischen den Threads eines einzelnen Clients erfolgt, wie im vorherigen Kapitel erläutert, über die *Shared Clause Database* und das *Master Control Object*. Dem gegenüber wird die Kommunikation zwischen Master- und Client-Prozessen über den Austausch von Nachrichten, das so genannte *Message Passing*, abgewickelt.

Die weiteren Abschnitte dieses Kapitels sind wie folgt gegliedert: in Abschnitt 9.1 werden die vom Master zu leistenden Aufgaben diskutiert. Die für den Master-Prozess umgesetzte Funktionalität orientiert sich dabei stark an den während der Entwicklung von PIChaff gemachten Erfahrungen, die auf einer konzeptuellen Ebene weitgehend unverändert in das PaMiraXT-Szenario übertragen wurden. Danach wird in Abschnitt 9.2 über die Änderungen berichtet, den an sich eigenständigen SAT-Algorithmus MiraXT so zu erweitern, dass dieser als Client in PaMiraXT eingesetzt werden kann. Im Kern beruhen die vorgenommenen Modifikationen auf der Idee, zusätzlich zu den Threads, welche die sequentiellen SAT-Prozeduren abarbeiten (im Folgenden zur besseren Unterscheidung als *SAT-Threads*

bezeichnet), mit dem so genannten *MPI-Thread* einen weiteren Thread einzuführen. Die Aufgabe dieses speziellen Threads besteht darin, den Kontakt zum Master zu halten und so für einen reibungslosen Austausch von Statusmeldungen, Teilproblemen und generierten Konflikt-Klauseln auf der Ebene der Clients zu sorgen. Die Namensgebung ergibt sich dabei aus der Tatsache, dass die Kommunikation zwischen Master und Clients mit MPICH [48] umgesetzt wurde, einer Implementierung des so genannten *Message Passing Interface Standards* [105]. Abschließend wird in Abschnitt 9.3 das Potenzial von PaMiraXT in verschiedenen Konfigurationen mit bis zu drei Clients und insgesamt acht SAT-Threads anhand der aus Kapitel 8 bekannten Problemklassen *SAT Race 2006* und *SAT 2007 Industrial* evaluiert.

9.1 Master-Prozess

Der Master-Prozess hat wie der Kommunikationsprozessor in PIChaff die Aufgabe, für ein reibungsloses und effizientes Zusammenspiel der verschiedenen Clients zu sorgen. Im Einzelnen gehören folgende Punkte zum Aufgabenspektrum des Masters:

- Starten und Stoppen der Clients,
- Steuern des Austauschs von Teilproblemen zwischen den Clients und
- Weiterleiten von Konflikt-Klauseln auf der Client-Ebene.

Die Initialisierungsphase von PaMiraXT gestaltet sich etwas komplexer als ähnliche Routinen anderer paralleler SAT-Algorithmen, die ebenfalls auf einem Master/Client-Modell aufbauen. Die verschiedenen Verfahren eint, dass den Clients zum Aufbau der initialen Klauseldatenbank während der Startphase entweder vom Master die zu lösende CNF-Formel weitergeleitet wird oder diese die Formel selbständig aus einer Datei einlesen. Die letztgenannte Option bedeutet einerseits, dass alle Clients Zugriff auf die entsprechende CNF-Datei haben müssen und andererseits, dass der Master-Prozess den Clients vor der Zuweisung von ersten Teilproblemen entsprechend Zeit zum Einlesen und Initialisieren der Klauseldatenbank geben muss. In PaMiraXT wird die zweite Variante verfolgt, die für den Master entstehende Pause gleich zu Beginn der Abarbeitung fällt unter Umständen deutlich höher aus als bei anderen parallelen SAT-Algorithmen, da jeder Client zunächst für seine lokal gehaltene initiale Klauselmenge das *Preprocessing* durchführt. Solange die Clients auf unterschiedlichen Rechnern gestartet werden, entsteht im Vergleich zu einem vom Master zentral durchgeführten Preprocessing kein Performance-Nachteil, zumal die in MiraXT implementierte Funktionalität rein deterministisch vorgeht und somit auf allen Clients zum gleichen Resultat führt.

Der eigentliche Start der Suche nach einer erfüllenden Belegung erfolgt also erst nach einer entsprechenden Rückmeldung der Clients, die über diesen Weg das Ende des Preprocessings

und ihre Bereitschaft zum Bearbeiten von Teilproblemen signalisieren. Wie gehabt wird im letzten Schritt der Startphase durch den Master lediglich ein Client mit einem leeren Guiding Path gestartet, was bewirkt, dass der komplette durch die zu lösende CNF-Formel aufgespannte Suchraum unter den SAT-Threads des soeben gestarteten Clients aufgeteilt wird. Alle anderen Clients stellen zunächst eine Anfrage nach einem Teilproblem an den Master, so dass direkt im Anschluss an das Startsignal die gegebene Probleminstanz auf der Ebene der Clients solange in disjunkte Bereiche aufgeteilt wird, bis jeder Client über ein initiales Teilproblem verfügt.

Der Austausch von Teilproblemen lässt sich dabei folgendermaßen skizzieren: immer wenn während der Suche nach einer erfüllenden Belegung die Situation eintritt, dass alle SAT-Threads eines Clients inaktiv geworden sind, initiiert der entsprechende MPI-Thread eine Anfrage an den Master, indem er als Lösung für sein aktuelles Teilproblem unerfüllbar (*UNSAT* in Abbildung 9.1) zurückgibt. Solange zumindest noch ein Client aktiv ist, verfährt der Master wie folgt: unter allen aktiven und für einen Austausch eines Teilproblems in Frage kommenden Clients wird, analog zu PIChaff, derjenige Client kontaktiert und ausgewählt, dessen Restproblem heuristisch bestimmt aller Wahrscheinlichkeit nach die meiste Laufzeit benötigen wird. Das erhaltene Teilproblem wird dann durch den Master an den inaktiven Client beziehungsweise dessen MPI-Thread weitergeleitet, der dieses wiederum „seinen“ SAT-Threads zur Verfügung stellt. Dadurch, dass der Austausch von Teilproblemen nach Kontaktierung eines aktiven Clients und der Aufforderung zur Aufteilung des Suchbereichs über den Master und nicht direkt zwischen den Clients erfolgt, bietet sich auf diesem Weg dem Master die Chance, für jeden Client das von diesem bearbeitete Teilproblem, kodiert durch einen Guiding Path, zu speichern. Dieses Wissen wird vom Master genutzt, um zu bewerten und einzuschätzen, welcher Client ein schwieriges Teilproblem bearbeitet (kurzer Guiding Path, wenige Variablen vorgegeben) und daher vorrangig zum Aufteilen des Suchraums aufgefordert werden sollte.

Ist bei einer Anfrage eines Clients nach einem Teilproblem kein anderer Client mehr aktiv, also alle Clients in einen wartenden Zustand übergegangen, ist das gestellte Problem unerfüllbar: in keinem der von den Clients analysierten Bereiche der CNF-Formel, die zusammen das Gesamtproblem ergeben, konnte eine erfüllende Belegung ermittelt werden. Als Folge dessen stoppt der Master zunächst alle Clients (*Stop Signal*, siehe Abbildung 9.1), bevor PaMiraXT mit einer entsprechenden Ausgabe beendet wird. Gleiches gilt für den Fall, dass ein Client beziehungsweise einer der SAT-Threads ein Modell gefunden hat; auch in dieser Situation werden erst die Clients gestoppt, dann das Resultat weitergegeben und schlussendlich PaMiraXT beendet.

Neben dem (lokalen) Austausch von Konflikt-Klauseln zwischen SAT-Threads eines einzelnen Clients ist in PaMiraXT auch der Austausch von Konflikt-Klauseln zwischen unterschiedlichen Clients möglich. An dieser Stelle wird ebenfalls auf den Erfahrungen bei der Entwicklung von PIChaff aufgebaut, indem der Master-Prozess zunächst alle von den Cli-

ents als prinzipiell hilfreich eingestuften Konflikt-Klauseln entgegennimmt, diese nach ihrer Bedeutung für die spezifischen Teilprobleme der restlichen Clients bewertet und entsprechend weiterreicht. Das „Filtern“ der erhaltenen Konflikt-Klauseln richtet sich wiederum danach, ob eine bestimmte Konflikt-Klausel bereits durch den das aktuelle Teilproblem eines Clients spezifizierenden Guiding Path erfüllt ist oder nicht. Nur im zweiten Fall wird die Klausel an den entsprechenden Client, genauer an den MPI-Thread des Clients, weitergeleitet und von diesem in die *Shared Clause Database* eingetragen, so dass die davon betroffenen SAT-Threads unmittelbar von diesem Wissen profitieren können.

Im Gegensatz zu PIChaff ist der Transfer von Konflikt-Klauseln allerdings variabel gestaltet worden: je nach Konfiguration wird entweder jede Klausel als eigenständige Nachricht gesendet oder es werden mehrere Konflikt-Klauseln zu einer Nachricht zusammengefasst. Ersteres hat den Vorteil, dass relevante Konflikt-Klauseln entsprechend früh an den anderen Standorten (den Clients und damit den jeweiligen SAT-Threads) verfügbar sind, bedeutet aber auch einen erhöhten Kommunikationsaufwand. Durch ein Zusammenfassen mehrerer Klauseln kann dieser Aufwand reduziert werden, geht aber zugleich mit dem Nachteil „verspätet“ eintreffender Konflikt-Klauseln einher. Als Vorgriff auf Abschnitt 9.3 sei angedeutet, dass sich eine Paketgröße von 50 Klauseln pro Nachricht in diesem Zusammenhang als guter Kompromiss erwiesen hat.

Im Falle einer Konfiguration von PaMiraXT, die ein Zusammenfassen mehrerer Konflikt-Klauseln zu einem Datenpaket vorsieht, hat dies sowohl Auswirkungen auf die Übergabe der Konflikt-Klauseln von einem Client an den Master als auch auf die Weiterleitung der Daten vom Master an die restlichen Clients. Auf Seiten der Clients werden in einem entsprechenden Puffer solange die ein vorgegebenes Auswahlkriterium erfüllenden Konflikt-Klauseln abgelegt, bis das festgesetzte Limit (in diesem Fall Pakete zu je 50 Klauseln) erreicht ist und die Klauseln gesammelt an den Master transferiert werden. Auf Seiten des Masters erfolgt dann, getrennt nach Clients, ebenfalls eine Pufferung der Konflikt-Klauseln. Abbildung 9.2 zeigt den hierfür vom Master verwendeten *Conflict Clause Buffer* schematisch für ein Szenario mit vier Clients. Bedingt durch den filternden Zwischenschritt des Masters, bei dem alle Konflikt-Klauseln von einer Weiterleitung an diejenigen Clients ausgeschlossen werden, bei denen die Klauseln bereits durch den jeweiligen Guiding Path erfüllt sind, ist klar, dass die „Füllstände“ der einzelnen Puffer unterschiedlich sind. Ist bei einem Puffer das Limit erreicht, werden alle darin gespeicherten Konflikt-Klauseln in Form einer einzigen Nachricht an den entsprechenden Client transferiert.

Wie bereits erwähnt, wurde die Realisierung der Kommunikation zwischen Master- und Client-Prozessen mit Hilfe von *MPI Chameleon* (MPICH) [48], einer Implementierung des *Message Passing Interface Standards* [105], umgesetzt. Dieser Standard legt Richtlinien und Schnittstellen bezüglich der Funktionalität einzelner Routinen fest, mit denen eine reibungslose Kommunikation auf der Basis eines Austauschs von Nachrichten gewährleistet ist. Das Prinzip von MPICH beruht darauf, dass auf allen Rechnern des Netzwerks, die für

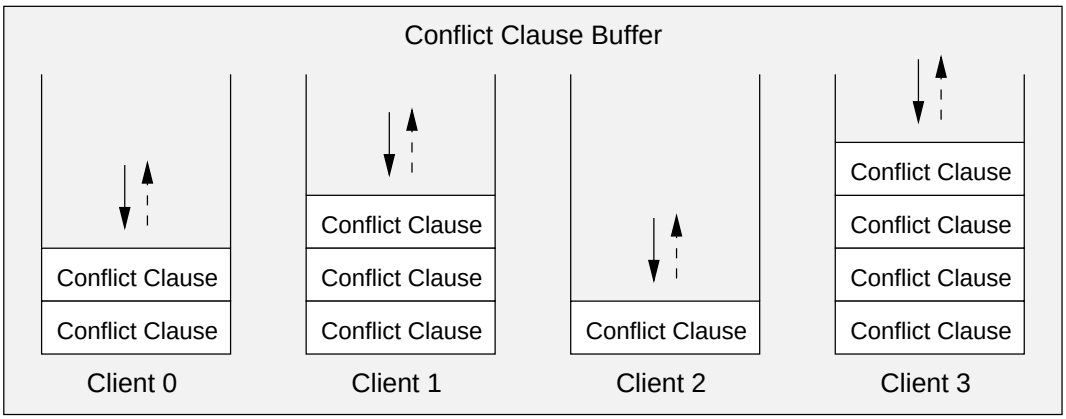


Abbildung 9.2: Conflict Clause Buffer

eine parallele Anwendung genutzt werden sollen, zunächst ein spezielles MPICH-Programm ausgeführt wird, das wiederum die auf den jeweiligen Rechnern auszuführenden Anwenderprogramme startet und während der Abarbeitung im Hintergrund die Kommunikation auf Hardware-Ebene zwischen diesen steuert. Prinzipiell kommen für die Realisierung von PaMiraXT auch andere Funktionsbibliotheken oder Software-Pakete in Frage. Exemplarisch seien LAM [21] und, obgleich mit deutlich größerem Funktionsumfang als benötigt, PVM [43] genannt. Da die Kommunikation zwischen Master- und Client-Prozessen im Wesentlichen mit Standardfunktionen zum Senden und Empfangen von Nachrichten umgesetzt wurde, sollte eine Portierung problemlos möglich sein und die Performance von PaMiraXT nur marginal beeinflussen. MPICH wurde aufgrund der sehr guten Dokumentation und der zahlreichen Beispielprogramme ausgewählt [47].

Neben den obligatorischen Befehlen zum Senden und Empfangen von Nachrichten ist für den Master insbesondere das MPICH-Kommando `MPI_Probe` von Wichtigkeit, da mit diesem überprüft werden kann, ob neue Nachrichten für den Master bereitstehen (unabhängig von welchem Absender und mit welchem Inhalt). Der Vorteil dieses Befehls liegt darin, dass der entsprechende Prozess, solange keine Nachricht vorliegt, bis zum Eintreffen der nächsten Nachricht in einen „schlafenden“ Zustand versetzt wird. Da der Master-Prozess nach der Initialisierungsphase lediglich dann aktiv werden muss, wenn einer oder mehrere Clients Anfragen nach Teilproblemen stellen oder Konflikt-Klauseln übermitteln, bietet sich `MPI_Probe` an, um den Master in der Zwischenzeit inaktiv werden zu lassen. Insbesondere wenn der Master zusammen mit einem Client auf einem Rechner ausgeführt wird (Standardkonfiguration der in Abschnitt 9.3 durchgeführten Experimente), ist dies ein geeignetes Mittel, um die CPU-Auslastung durch den Master zu verringern und so die Performance des mit dem Master auf einem Rechner ausgeführten Clients möglichst wenig zu beeinflussen.

9.2 Client-Prozesse

In weiten Teilen ist die in PaMiraXT für die Clients eingesetzte Variante von MiraXT identisch mit der in Kapitel 8 vorgestellten und als eigenständiges Programm genutzten MiraXT-Version. Jeder Client liest zunächst die originale CNF-Formel ein, führt das Preprocessing durch und initialisiert im Anschluss daran mit der modifizierten Klauselmenge seine *Shared Clause Database*. Auch sind die SAT-Threads, welche die eigentliche Suche nach einer erfüllenden Belegung durchführen, unverändert. Einzig zur Kommunikation mit dem Master-Prozess wurde ein zusätzlicher Thread, der so genannte *MPI-Thread*, eingeführt. In Abbildung 9.3 ist die Konzeption für ein Beispiel mit vier SAT-Threads und einem MPI-Thread dargestellt.

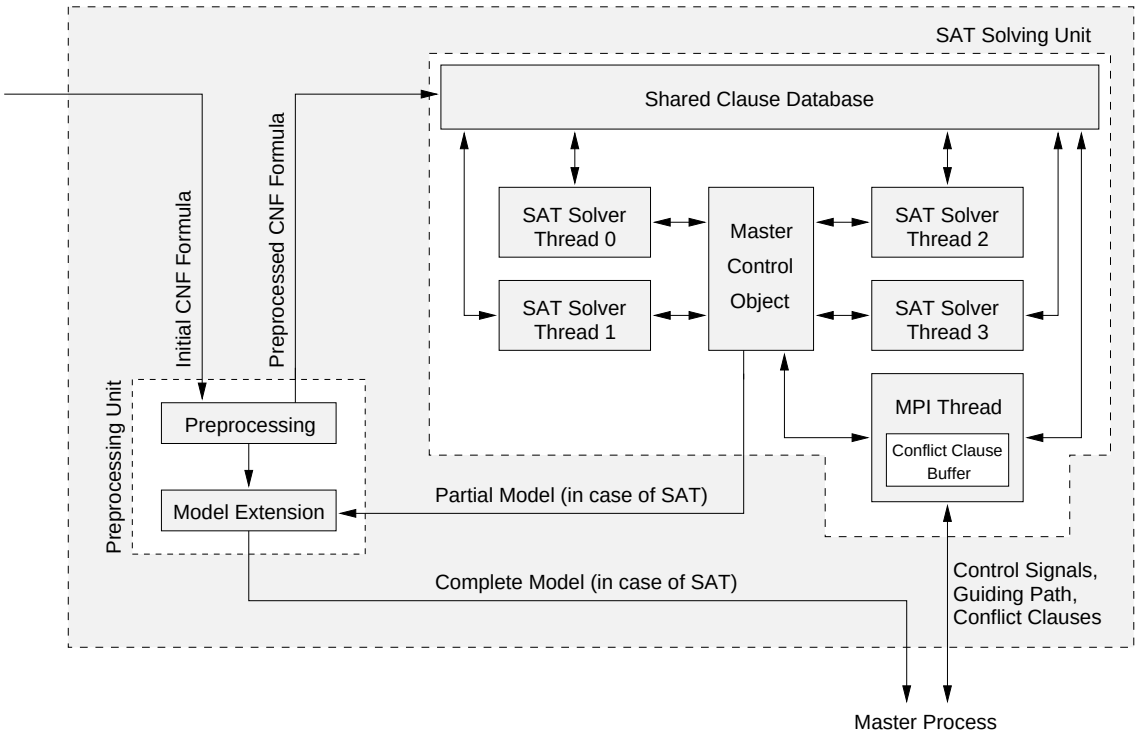


Abbildung 9.3: Design der in PaMiraXT eingesetzten Variante von MiraXT

Zunächst fällt auf, dass der MPI-Thread ebenso wie die SAT-Threads einen Zugang zur *Shared Clause Database* und zum *Master Control Object* besitzt. Letzteres ist insbesondere für den Austausch von Teilproblemen zwischen verschiedenen Clients von Bedeutung und folgendermaßen geregelt: in periodischen Abständen prüft der MPI-Thread den Status der im *Master Control Object* abgelegten Variablen *No. of idle Threads*. Ist der dort gespeicherte Wert gleich mit der Anzahl der SAT-Threads, bedeutet dies, dass alle SAT-Threads

des von diesem MPI-Thread gesteuerten Clients inaktiv sind. Als Folge dessen wird der Master-Prozess durch den MPI-Thread kontaktiert und eine Anfrage nach einem neuen Teilproblem gestellt. Ein vom Master daraufhin erhaltenes Teilproblem eines anderen Clients wird vom MPI-Thread durch einen Eintrag in das Datenfeld *Stack of Subproblems* des Master Control Objects (siehe Abbildung 8.4) an die SAT-Threads weitergeleitet, so dass diese das Problem entgegennehmen und untereinander aufteilen können. In ähnlicher Art und Weise wird verfahren, wenn der MPI-Thread eine Anfrage nach einem Teilproblem vom Master-Prozess erhält. Ebenfalls über das *Master Control Object* wird die Anfrage an die SAT-Threads weitergeleitet, indem sich der MPI-Thread selbst als einen inaktiven Thread in *No. of idle Threads* einträgt, das heißt den dortigen Zählerstand inkrementiert. Sobald ein Teilproblem durch einen „seiner“ SAT-Threads bereitgestellt wurde und der MPI-Thread den exklusiven Zugriff auf das Feld *Stack of Subproblems* erhalten hat, nimmt er den entsprechenden Guiding Path aus der Liste heraus, dekrementiert den Zähler *No. of idle Threads* wieder und leitet das Teilproblem an den Master weiter. Während der Master bei der Weitergabe von Teilproblemen versucht, zuerst den größten der von den Clients analysierten Bereiche aufzuteilen, geschieht die Partitionierung auf der Ebene der SAT-Threads weiterhin zufällig. Dies ist unabhängig davon, ob das abgetrennte Teilproblem von einem anderen SAT-Thread aufgegriffen oder, wie in dem hier skizzierten Szenario, durch den MPI-Thread an den Master übermittelt wird.

Ist es dem Master nach einer Anfrage eines Clients nicht möglich, diesen mit einem neuen Teilproblem zu versorgen, heißt das, dass alle anderen Clients sich ebenfalls im Wartezustand befinden und bis dato keiner eine erfüllende Belegung ermitteln konnte. Die gestellte CNF-Formel ist folglich unerfüllbar. Den MPI-Threads aller Clients wird dies mitgeteilt, woraufhin sie das Flag *Done* ihres *Master Control Objects* setzen. Zu jenem Zeitpunkt befinden sich alle SAT-Threads sämtlicher Clients im Wartezustand. Wie in Abschnitt 8.2 erläutert, reagieren die SAT-Threads in dieser Phase lediglich auf Veränderungen innerhalb des Datenfelds *Stack of Subproblems* sowie auf Veränderungen der Variablen *Done*, so dass über diesen Weg bei einer unerfüllbaren Instanz eine korrekte Terminierung aller Clients und damit die Terminierung von PaMiraXT gewährleistet ist. Gleiches gilt für den Fall, dass ein SAT-Thread eines Clients eine erfüllende Belegung bestimmen konnte, auch in dieser Situation stoppen die MPI-Threads nach dem Erhalt einer dahingehenden Nachricht des Masters die von ihnen verwalteten SAT-Threads durch ein Setzen der Variablen *Done*.

Der Zugang des MPI-Threads zur *Shared Clause Database* wird dazu genutzt, den Austausch von Konflikt-Klauseln nicht nur auf den Austausch zwischen den SAT-Threads eines einzelnen Clients zu beschränken, sondern auf die Client-Ebene zu erweitern. Analog zu den SAT-Threads verfügt auch der MPI-Thread über einen Positionszeiger, der es ihm erlaubt, neue Konflikt-Klauseln von bereits bekannten Klauseln zu unterscheiden. Alle neuen Klauseln werden der Reihe nach durch den MPI-Thread bezüglich ihrer Länge bewertet und, sofern ein festgelegtes Limit nicht überschritten wird, als relevant für andere Clients ein-

gestuft und an den Master-Prozess weitergeleitet. Wie im vorherigen Abschnitt erläutert, kann diese Übertragung entweder für jede Klausel einzeln oder als Block mehrerer Klauseln erfolgen. Im zweiten Fall speichert der MPI-Thread die Klauseln solange in seinem *Conflict Clause Buffer* zwischen (siehe Abbildung 9.3), bis die gewünschte Anzahl an Klauseln erreicht wurde, die daraufhin in Form einer einzigen Nachricht an den Master gesendet werden. Jede hierbei vom MPI-Thread analysierte Konflikt-Klausel wird, unabhängig ob weitergegeben oder nicht, durch das Setzen des mit dem MPI-Thread assoziierten *Clause Deletion Flags* als „vom MPI-Thread gelöscht“ markiert, so dass ein Entfernen dieser Klauseln aus der Shared Clause Database einzig von den SAT-Threads und deren Zugriffen auf die Klauseln abhängt.

An dieser Stelle sei nochmals darauf hingewiesen, dass auch bezüglich der *Shared Clause Database* alle Threads eines Clients gleich behandelt werden. Eine Klausel kann folglich erst dann aus der globalen Klauselmenge entfernt werden, wenn sowohl alle SAT-Threads als auch der MPI-Thread signalisiert haben, diese nicht mehr zu berücksichtigen. Im Umkehrschluss kann dadurch garantiert werden, dass keine Klausel gelöscht wird, die der MPI-Thread noch nicht bewertet und gegebenenfalls an den Master weitergeleitet hat. Vom Master entgegengenommene Klauseln, seien es einzelne Klauseln oder ein Paket mehrerer Klauseln, werden auf dem gleichen Weg in die Klauseldatenbank eingetragen, wie dies die SAT-Threads für ihre hergeleiteten Konflikt-Klauseln vornehmen. Die Klauseln werden der Reihe nach an die Integrationsroutine der Shared Clause Database übergeben, wo sie von dieser zur Klauselmenge hinzugefügt werden, so dass sie unmittelbar danach allen SAT-Threads des jeweiligen Clients zur Verfügung stehen. In diesem Kontext ist es wichtig, dass ein MPI-Thread vom Master erhaltene Konflikt-Klauseln nicht erneut als relevant einstuft und wieder an den Master zurückgibt. Die Folge wäre ein erhöhter Kommunikationsaufwand ohne jeglichen Informationsgewinn. Dazu wird mit jeder Klausel neben der Länge und den Clause Deletion Flags noch die ID desjenigen Threads gespeichert, der die Klausel zur Klauseldatenbank hinzugefügt hat. So ist es dem MPI-Thread möglich, in seinen Betrachtungen all diejenigen Klauseln von einer Weitergabe auszuschließen, die von ihm selbst in die Shared Clause Database eingetragen wurden.

Abschließend sei angemerkt, dass der MPI-Thread zur Reduzierung der durch ihn verursachten CPU-Auslastung sporadisch in einen „schlafenden“ Zustand versetzt wird. Dies erhöht zwar unter Umständen die Reaktionszeit des MPI-Threads, bringt aber Vorteile mit sich, wenn der MPI-Thread zusammen mit einem SAT-Thread auf einem CPU-Kern eines Prozessors ausgeführt wird.

9.3 Experimentelle Ergebnisse

Für die Durchführung der nachfolgend diskutierten Experimente wurden für den Austausch von Konflikt-Klauseln zwischen den MiraXT-Clients folgende Parameter verwendet: vom

MPI-Thread des entsprechenden Clients werden nur Konflikt-Klauseln an den Master-Prozess weitergegeben, die aus maximal drei Literalen bestehen. Zudem wird nicht jede Konflikt-Klausel als eigene Nachricht verschickt, sondern je 50 Konflikt-Klauseln zusammengefasst. Der Master wiederum wählt, wie in Abschnitt 9.1 beschrieben, für die einzelnen Clients nur diejenigen Konflikt-Klauseln aus, die nicht bereits durch den Guiding Path, der das jeweilige von den Clients aktuell bearbeitete Teilproblem spezifiziert, erfüllt sind. Zur Durchführung des Informationsaustauschs nutzt der Master für jeden Client einen eigenen Zwischenspeicher innerhalb seines *Conflict Clause Buffers* (siehe Abbildung 9.2) und verschickt ebenfalls Datenpakete zu je 50 Konflikt-Klauseln in Form einer einzigen Nachricht.

Die Entscheidung, nur relativ kurze Konflikt-Klauseln zu berücksichtigen und in einem mit 50 Elementen relativ großen Paket weiterzuleiten, beruht darauf, dass das Verhältnis zwischen dem Nutzen von „externen“ Klauseln und dem zur Weiterleitung erforderlichen Aufwand berücksichtigt werden muss. Werden auch Konflikt-Klauseln mit mehr Literalen getauscht, steigt die Anzahl der potenziellen Kandidaten, was zu einem erhöhten Kommunikations- und Integrationsaufwand sowohl auf Seiten des MPI- als auch auf Seiten der SAT-Threads führt. Das gleiche Argument gilt für ein Verschicken der Klauseln in kleineren Paketen, beides beeinflusst die Laufzeit negativ. Andererseits sind für die SAT-Threads die Konflikt-Klauseln bei kleineren Paketen oder einem weniger restriktiven Auswahlkriterium gegebenenfalls früher beziehungsweise überhaupt verfügbar, was eine effizientere Navigation des Suchprozesses und damit eine Reduzierung der Laufzeit bedeuten könnte.

In diesem Zusammenhang muss berücksichtigt werden, dass die durchgeführten Experimente von Konfigurationen von PaMiraXT ausgehen, bei denen exakt so viele SAT-Threads gestartet werden wie dem jeweiligen Client auf dem Rechner, auf dem er ausgeführt wird, an Prozessoren oder CPU-Kernen zur Verfügung stehen. Das bedeutet auch, dass der MPI-Thread zusammen mit einem der SAT-Threads auf einem CPU-Kern ausgeführt wird, was nur dann den entsprechenden SAT-Thread nicht verlangsamt, wenn die Aufgaben des MPI-Threads auf ein Minimum beschränkt sind. Weiterhin wird auch der Master zusammen mit einem der Clients auf einem Rechner gestartet, so dass an dieser Stelle der Kommunikationsaufwand ebenfalls nicht außer Acht gelassen werden darf. All diese Randbedingungen führten schlussendlich zu den gewählten Einstellungen.

Für die Durchführung der Experimente wurden wiederum die insgesamt 347 CNF-Formeln der beiden Problemklassen *SAT Race 2006* und *SAT 2007 Industrial* gewählt, die bereits im vorherigen Kapitel zum Einsatz kamen. Das Zeitlimit wurde mit 900 Sekunden (*SAT Race 2006*) beziehungsweise 10000 Sekunden (*SAT 2007 Industrial*) identisch zu Abschnitt 8.4 angesetzt. Als Hardware-Plattformen standen neben dem *Dual-Core AMD Opteron 280 Doppelprozessorsystem* noch zwei weitere Doppelprozessorsysteme zur Verfügung. Beide werden im Folgenden als *AMD Opteron 250 Doppelprozessorsystem* bezeichnet. Abbildung 9.4 zeigt schematisch die Architektur dieser Rechner, die nahezu identisch mit dem *Dual-*

Core AMD Opteron 280 Doppelprozessorsystem sind. Auf der Hauptplatine befinden sich ebenfalls zwei Prozessoren, die jeweils lokal an 2 GB Hauptspeicher angeschlossen und per *Hyper Transport Links* miteinander verbunden sind. Die *AMD Opteron 250* Prozessoren selbst sind nahezu identisch mit der *AMD Opteron 280*-Reihe, verfügen über je 1 MB L2-Cache und werden bei 2,4 GHz Taktfrequenz betrieben. Allerdings handelt es sich um so genannte *Single-Core* Prozessoren, die gegenüber den *AMD Opteron 280* Prozessoren lediglich mit einem CPU-Kern ausgestattet sind. Alle Rechner sind per Ethernet-Verbindung miteinander verknüpft, so dass für die Ausführung von PaMiraXT maximal sechs Prozessoren mit insgesamt acht CPU-Kernen zur Verfügung stehen.

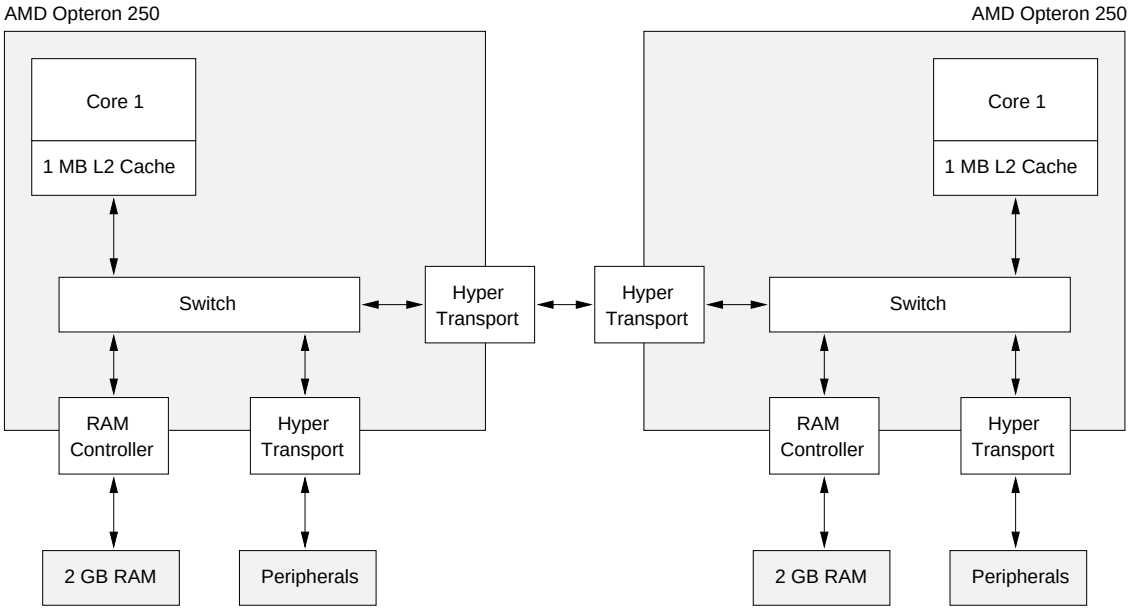


Abbildung 9.4: AMD Opteron 250 Doppelprozessorsystem [4, 6]

Tabelle 9.1 stellt die von verschiedenen Konfigurationen von PaMiraXT beim Lösen aller 100 Instanzen der Problemklasse *SAT Race 2006* erzielten Ergebnisse den Resultaten von RSat, MiniSat2, PicoSAT und MiraXT aus Kapitel 8 gegenüber. Analog zu MiraXT spiegeln die Gesamtlaufzeiten von PaMiraXT jeweils den Mittelwert dreier Durchläufe wider.

Bei der PaMiraXT-Konfiguration mit zwei Clients und vier SAT-Threads werden analog zur schnellsten der MiraXT-Varianten insgesamt vier SAT-Threads eingesetzt, allerdings verteilt auf zwei Clients. Wie die Ergebnisse sowohl bei der Laufzeit als auch bezüglich der gelösten Instanzen zeigen, fällt diese Version von PaMiraXT gegenüber MiraXT mit vier Threads deutlich ab. Bei einer identischen Anzahl an sequentiellen SAT-Prozeduren demonstriert dieses Resultat die Vorteile eines rein threadbasierten parallelen SAT-Algorithmus

SAT-Algorithmus	Gesamtlaufzeit	Gelöste Instanzen	Hardware-Plattform
RSat	40209,10s	68	AMD Opteron 280 2P
MiniSat2	41631,15s	70	AMD Opteron 280 2P
PicoSAT	37033,29s	77	AMD Opteron 280 2P
MiraXT 1 Thread	32264,09s	78	AMD Opteron 280 2P
MiraXT 2 Threads	27651,93s	80,67	AMD Opteron 280 2P
MiraXT 4 Threads	21674,50s	85	AMD Opteron 280 2P
PaMiraXT 2 Clients, 4 Threads	23949,76s	81,67	2× AMD Opteron 250 2P
PaMiraXT 3 Clients, 8 Threads kein CKS	22130,88s	84	2× AMD Opteron 250 2P 1× AMD Opteron 280 2P
PaMiraXT 3 Clients, 8 Threads	**21274,41s**	**85,67**	2× AMD Opteron 250 2P 1× AMD Opteron 280 2P

Tabelle 9.1: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT, MiraXT und PaMiraXT. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattformen: *Dual-Core AMD Opteron 280 Doppelprozessorsystem* (AMD Opteron 280 2P) und *AMD Opteron 250 Doppelprozessorsystem* (AMD Opteron 250 2P). Das Kürzel *CKS* steht stellvertretend für *Client Knowledge Sharing*, was den Austausch von Konflikt-Klauseln auf der Ebene der Clients von PaMiraXT bezeichnet.

wie MiraXT, bei dem die Threads jederzeit Zugriff auf alle für sie potenziell relevanten Konflikt-Klauseln haben. Dem gegenüber werden bei PaMiraXT nur kurze Klauseln zwischen den beiden Clients ausgetauscht, so dass gegebenenfalls wichtige, aber zu lange Klauseln von der Weitergabe ausgeschlossen werden. Ebenso macht sich die relativ langsame Kommunikation per *Message Passing* zwischen den beiden Clients negativ bemerkbar.

Die beiden letzten Zeilen von Tabelle 9.1 fassen die Ergebnisse für eine Konfiguration von PaMiraXT zusammen, bei der sowohl die beiden *AMD Opteron 250 Doppelprozessorsysteme* als auch das *Dual-Core AMD Opteron 280 Doppelprozessorsystem* genutzt wurden. Bei dieser Variante werden, verteilt auf drei Clients, insgesamt acht SAT-Threads (vier auf dem *Dual-Core AMD Opteron 280 Doppelprozessorsystem* und je zwei auf den beiden *AMD Opteron 250 Doppelprozessorsystemen*) gestartet. Die aufgeführten Werte unterstreichen, dass neben dem Austausch von Konflikt-Klauseln zwischen den SAT-Threads eines einzelnen Clients, was aufgrund der Erfahrungen des vorherigen Kapitels als Standard angesehen wird, auch die Weitergabe von Klauseln auf der Ebene der Clients von Vorteil ist. PaMiraXT ist dann in der Lage, sich mit einer Gesamtlaufzeit von 21274,41 Sekunden und durchschnittlich 85,67 erfolgreich bearbeiteten Probleminstanzen an die Spitze aller getesteten SAT-Algorithmen zu setzen.

Im Vergleich zur MiraXT-Variante mit vier Threads, die auf dem *Dual-Core AMD Opteron 280 Doppelprozessorsystem* ausgeführt wurde, zeigt sich aber auch, dass beide Konfigurationen von PaMiraXT mit drei Clients und acht SAT-Threads bei dieser Problemklasse nicht oder nur wenig von den vier zusätzlichen SAT-Threads profitieren. Sowohl die in Abbildung 9.5 in Bezug zur Laufzeit dargestellte Anzahl der von RSat, MiniSat2, PicoSAT, MiraXT und PaMiraXT jeweils gelösten Instanzen als auch die in Tabelle 9.2 angegebenen Speedup-Werte der diversen MiraXT- und PaMiraXT-Varianten bestätigen dies.

Wie in Abschnitt 8.4.1 dargelegt wurde, können die von MiraXT im sequentiellen Betriebsmodus innerhalb des Zeitlimits von 900 Sekunden erfolgreich bearbeiteten *SAT Race 2006* Instanzen im Mittel in etwa 160 Sekunden gelöst werden. Dieser relativ geringe Zeitrahmen hat bei PaMiraXT zur Folge, dass die für die Kommunikation zwischen den drei Clients und dem Master-Prozess benötigte Laufzeit einen nicht unerheblichen Teil der Gesamtlaufzeit stellt. Vereinfacht ausgedrückt ist bei der Problemklasse *SAT Race 2006* das Verhältnis zwischen dem Leistungszuwachs, hervorgerufen durch zusätzliche, auf weitere Clients verteilte SAT-Threads, und dem benötigten Aufwand, diese in den Suchprozess einzubinden, nicht optimal.

Das Bild wandelt sich beim Übergang zur Problemklasse *SAT 2007 Industrial*, bei der die Mehrzahl der Instanzen sehr viel schwerer zu lösen ist und einen höheren Zeitaufwand erfordert, so dass der Kommunikationsaufwand eine nur untergeordnete Rolle spielt. Tabelle 9.3 enthält die jeweiligen Ergebnisse. Wie in Kapitel 8 wurde auch hier aufgrund des hohen Zeitlimits von 10000 Sekunden auf die Durchführung mehrerer Läufe verzichtet und

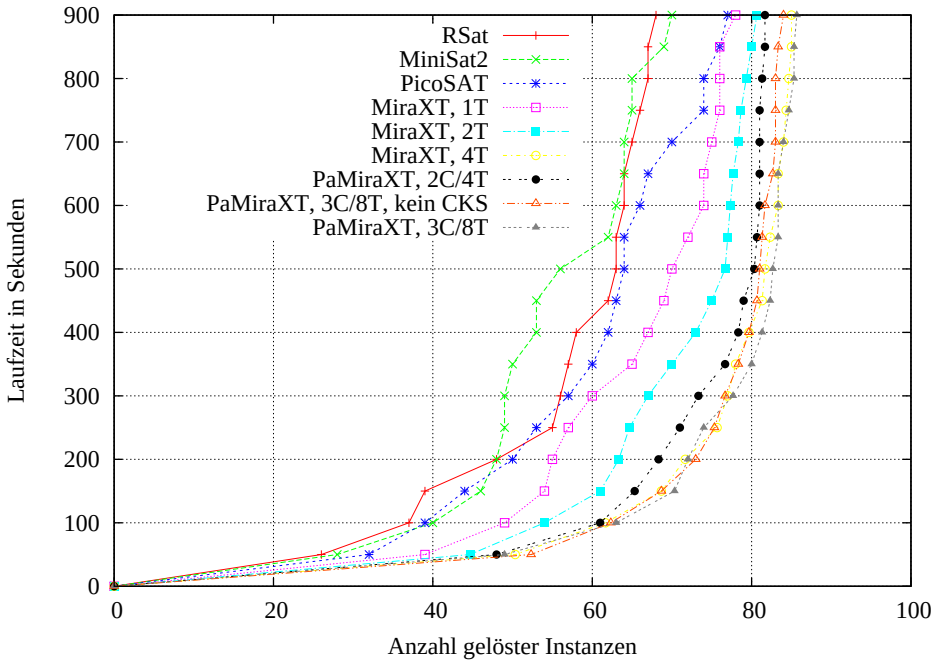


Abbildung 9.5: Anzahl der von RSat, MiniSat2, PicoSAT, MiraXT und PaMiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT Race 2006*, Zeitlimit: 900 Sekunden, Hardware-Plattformen: siehe Tabelle 9.1. Das Kürzel *CKS* steht stellvertretend für *Client Knowledge Sharing*, was den Austausch von Konflikt-Klauseln auf der Ebene der Clients von PaMiraXT bezeichnet.

PaMiraXT zudem nur in der leistungsstärksten Variante mit drei Clients, acht Threads und dem aktivierten Austausch von Konflikt-Klauseln zwischen den Clients gestartet.

Wie deutlich zu erkennen ist, hebt sich PaMiraXT bei dieser Gruppe von Problemstellungen von den restlichen SAT-Algorithmen ab und kann gegenüber der MiraXT-Variante mit vier Threads weitere sechs Instanzen erfolgreich innerhalb von 10000 Sekunden lösen. Tabelle 9.4 vergleicht die für MiraXT und PaMiraXT ermittelten Beschleunigungen mit dem sequentiellen Betriebsmodus von MiraXT, jeweils beschränkt auf die CNF-Formeln, die von allen Varianten von MiraXT beziehungsweise PaMiraXT gelöst werden konnten. PaMiraXT liegt mit einem Speedup von 5,62 zwar nicht ganz im linearen Bereich, erzielt aber dennoch eine sehr gute Beschleunigung und hat bei der Kategorie *SAT 2007 Industrial*, wie Abbildung 9.6 belegt, stets die führende Position inne.

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	11353,62	1,00
MiraXT, 2 Threads	6902,29	1,64
MiraXT, 4 Threads	4418,97	2,57
PaMiraXT, 2 Clients, 4 Threads	4887,16	2,32
PaMiraXT, 3 Clients, 8 Threads, kein CKS	4134,58	2,75
PaMiraXT, 3 Clients, 8 Threads	4332,08	2,62

Tabelle 9.2: Beschleunigung verschiedener MiraXT- und PaMiraXT-Konfigurationen im Vergleich zum sequentiellen Modus. Problemklasse: *SAT Race 2006* (beschränkt auf die Instanzen, die von allen Konfigurationen von MiraXT und PaMiraXT gelöst werden konnten), Zeitlimit: 900 Sekunden, Hardware-Plattformen: siehe Tabelle 9.1. Das Kürzel *CKS* steht stellvertretend für *Client Knowledge Sharing*, was den Austausch von Konflikt-Klauseln auf der Ebene der Clients von PaMiraXT bezeichnet.

SAT-Algorithmus	Gesamtlaufzeit	Gelöste Instanzen	Hardware-Plattform
RSat	1293297,72s	136	AMD Opteron 280 2P
MiniSat2	1199874,43s	143	AMD Opteron 280 2P
PicoSAT	1110696,47s	152	AMD Opteron 280 2P
MiraXT 1 Thread	999154,42s	162	AMD Opteron 280 2P
MiraXT 2 Threads	835851,65s	177	AMD Opteron 280 2P
MiraXT 4 Threads	783190,38s	180	AMD Opteron 280 2P
PaMiraXT 3 Clients, 8 Threads	**711278,80s**	**186**	2× AMD Opteron 250 2P 1× AMD Opteron 280 2P

Tabelle 9.3: Experimentelle Ergebnisse von RSat, MiniSat2, PicoSAT, MiraXT und PaMiraXT. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattformen: *Dual-Core AMD Opteron 280 Doppelprozessorsystem* (AMD Opteron 280 2P) und *AMD Opteron 250 Doppelprozessorsystem* (AMD Opteron 250 2P).

SAT-Algorithmus	Gesamtlaufzeit [s]	Speedup
MiraXT, 1 Thread	132618,80	1,00
MiraXT, 2 Threads	53734,05	2,47
MiraXT, 4 Threads	37769,97	3,51
PaMiraXT, 3 Clients, 8 Threads	23584,20	5,62

Tabelle 9.4: Beschleunigung von MiraXT und PaMiraXT im Vergleich zum sequentiellen Modus. Problemklasse: *SAT 2007 Industrial* (beschränkt auf die Instanzen, die von allen Konfigurationen von MiraXT/PaMiraXT gelöst werden konnten), Zeitlimit: 10000 Sekunden, Hardware-Plattformen: siehe Tabelle 9.3.

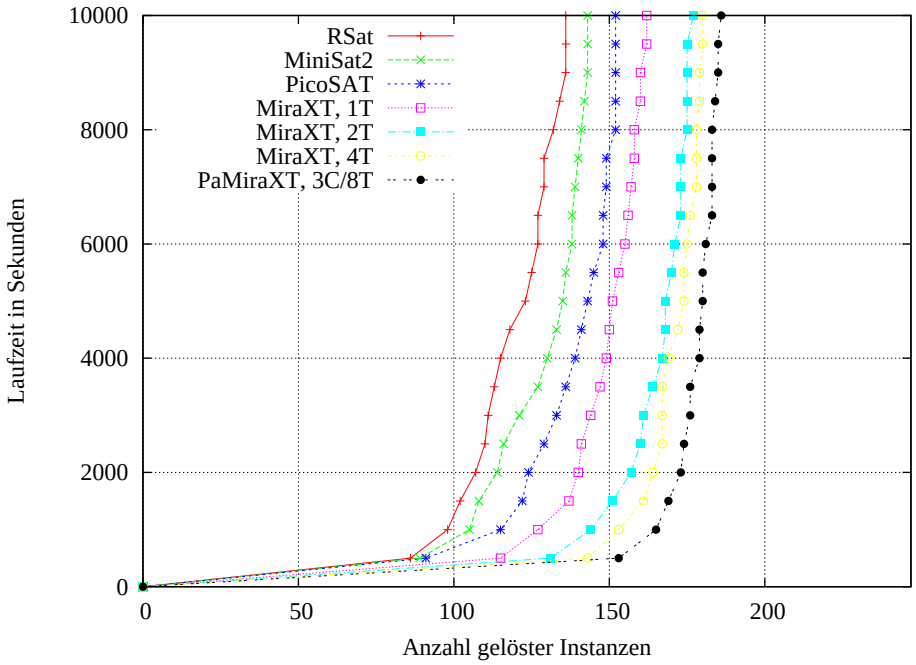


Abbildung 9.6: Anzahl der von RSat, MiniSat2, PicoSAT, MiraXT und PaMiraXT gelösten Instanzen in Abhängigkeit von der Laufzeit. Problemklasse: *SAT 2007 Industrial*, Zeitlimit: 10000 Sekunden, Hardware-Plattformen: siehe Tabelle 9.3.

Kapitel 10

Zusammenfassung

Mit der vorliegenden Arbeit wird die Entwicklung effizienter, paralleler und an die jeweilige Hardware-Umgebung speziell angepasster SAT-Algorithmen vorgestellt. Die drei entwickelten Verfahren sind PIChaff, MiraXT und PaMiraXT.

Den Anfang macht mit PIChaff ein Verfahren, das an ein am Lehrstuhl für Rechnerarchitektur entwickeltes Multiprozessorsystem angepasst wurde. Die drei Hauptkomponenten dieser Hardware-Plattform bestehen aus einer ISA-Steckkarte, einem Kommunikationsprozessor sowie den so genannten Recheneinheiten, auf denen die eigentliche SAT-Prozedur von PIChaff ausgeführt wird. Die Steckkarte fungiert dabei als Trägerboard und bietet Platz für bis zu neun Recheneinheiten. Der Kommunikationsprozessor, der in PIChaff als Master-Prozess eines Master/Client-Modells eingesetzt wird, ist einerseits für den Datentransfer zwischen den Recheneinheiten verantwortlich, indem er die Kommunikationskanäle einer Switch-Matrix so konfiguriert, dass die seriellen Schnittstellen der beiden in Kontakt tretenden Recheneinheiten direkt miteinander verbunden sind. Andererseits wird die Kommunikation mit dem Rechner, in den das Trägerboard eingesteckt ist, ebenfalls über den Kommunikationsprozessor abgewickelt.

Bedingt durch den mit 64 kWord sehr limitierten Speicher der Recheneinheiten, in dem die eigentliche SAT-Prozedur aber auch alle zum Lösen einer CNF-Formel benötigten Daten abgelegt sind, galt es insbesondere, eine möglichst kompakte SAT-Prozedur zu entwickeln, um mehr Speicherzellen für die zu bearbeitenden Probleminstanzen zur Verfügung zu stellen. Erreicht wurde dies auf der einen Seite durch eine vollständig in Maschinensprache vorgenommene Implementierung für die Recheneinheiten. Auf der anderen Seite sind einige Routinen, wie in Abschnitt 7.2.2 exemplarisch für die *Boolean Constraint Propagation* (BCP) skizziert, dahingehend abgeändert worden, dass sie mit möglichst wenig Speicher auskommen. Dennoch beinhaltet PIChaff mit einer Adaption der *Variable State Independent Decaying Sum* Entscheidungsheuristik, einer BCP-Funktion auf Basis von *Watched Literals* sowie einer Routine zur Konflikt-Analyse gemäß des *1UIP*-Prinzips alle elementaren Eckpfeiler moderner SAT-Algorithmen. Das umgesetzte Master/Client-Modell ermöglicht es den Recheneinheiten (Clients), gesteuert durch den als Master agierenden Kommunikationsprozessor, untereinander sowohl Teilprobleme als auch hergeleitete und potenziell hilfreiche Konflikt-Klauseln auszutauschen.

Die in Abschnitt 7.5 dargestellten experimentellen Ergebnisse belegen ein erfolgreiches Vorgehen. Sowohl mit drei, sechs als auch neun Clients konnte im Mittel eine im Vergleich zum sequentiellen Modus von PIChaff lineare Beschleunigung des Suchprozesses erzielt werden, wobei sich durch den Austausch von Konflikt-Klauseln in Abhängigkeit von der Problemklasse und der Anzahl der eingesetzten Clients ein zusätzlicher Performance-Gewinn von bis zu 13% eingestellt hat.

Der zweite in dieser Arbeit entwickelte parallele SAT-Algorithmus, MiraXT, richtet den Fokus auf Rechner mit aktuellen Dual-/Multi-Core Prozessoren und so genannte Mehrprozessorsysteme. Der Vorteil derartiger Hardware-Plattformen liegt darin, dass alle Prozessoren beziehungsweise CPU-Kerne Zugriff auf einen gemeinsamen Speicher haben, was in MiraXT zur Abwicklung jeglicher Kommunikation zwischen den einzelnen Threads genutzt wird. Neben den Threads, auf denen die eigentliche SAT-Prozedur ausgeführt wird, bilden die *Shared Clause Database* und das *Master Control Object* das Grundgerüst von MiraXT. Im Unterschied zu anderen parallelen und ebenfalls auf einem Thread-Konzept aufbauenden SAT-Verfahren zeichnet MiraXT aus, dass alle Threads auf einer einzigen Klauseldatenbank operieren, in der sämtliche während des Suchprozesses von den Threads hergeleiteten Konflikt-Klauseln abgelegt werden. Durch dieses Vorgehen eröffnet sich den Threads die Möglichkeit, zu jedem Zeitpunkt auf alle verfügbaren Konflikt-Klauseln zugreifen zu können, unabhängig von welchem Thread diese ursprünglich stammen, was einen besonders effizienten Austausch von Informationen über die zu lösende Probleminstanz zwischen den SAT-Prozeduren erlaubt. Ein ausgeklügeltes System von *Locks* zur Vergabe von exklusiven Schreibrechten gewährleistet dabei die Datenkonsistenz der *Shared Clause Database* bei zugleich minimalen Wartezeiten für die einzelnen Threads beim Einfügen neuer Klauseln.

Anders als in PIChaff wird in MiraXT kein Master/Client-Modell mit einem separaten, aktiven Master-Prozess umgesetzt. Stattdessen wird mit dem *Master Control Object* lediglich eine „passive“ Datenstruktur verwendet, mit der die Threads durch das Editieren entsprechender Speicherbereiche Statusinformationen und noch unbearbeitete Teilprobleme untereinander austauschen können.

Die in Abschnitt 8.4 dokumentierten experimentellen Ergebnisse belegen eindrucksvoll das Potenzial von MiraXT. Sowohl bei der Problemklasse *SAT Race 2006* als auch der Problemklasse *SAT 2007 Industrial*, beides Kollektionen anerkannt schwieriger Probleminstanzen, ist MiraXT bereits im sequentiellen Modus schneller als die zum Vergleich herangezogenen SAT-Algorithmen RSat, MiniSat2 und PicoSAT. Alle drei Verfahren gehören zu den aktuell leistungsstärksten sequentiellen SAT-Algorithmen. Durch den Einsatz von bis zu vier Threads konnte empirisch gezeigt werden, dass zusätzlich zur weiteren Reduktion der benötigten Gesamtlaufzeit zum Lösen der Problemstellungen auch die Zahl der innerhalb eines festgelegten Zeitlimits erfolgreich bearbeiteten CNF-Formeln erhöht wird. Die im

parallelen Betrieb gegenüber der sequentiellen MiraXT-Variante erzielte Beschleunigung variiert je nach Hardware-Plattform und Problemklasse und reicht von 2,07 (zwei Threads, *SAT 2007 Industrial*, *Intel Core 2 Duo T7200* Rechner) bis hin zu 3,51 bei vier Threads auf einem Doppelprozessorsystem mit zwei *Dual-Core AMD Opteron 280* Prozessoren (*SAT 2007 Industrial*).

Mit PaMiraXT, der finalen Entwicklung der vorliegenden Arbeit, wird MiraXT dahingehend erweitert, dass es auf Rechnernetzwerken eingesetzt werden kann, bei denen die einzelnen Rechner zwar per Ethernet-Verbindung miteinander verknüpft sind, aber nicht über einen gemeinsamen Speicherbereich verfügen. MiraXT wäre in dieser Situation nur auf einem einzelnen Knoten eines derartigen Netzwerks ausführbar. PaMiraXT überwindet diese Einschränkung, indem eine zweite Ebene der Parallelität eingeführt wird, bei der die auf verschiedenen Rechnern gestarteten „Kopien“ von MiraXT als Clients eines Master/Client-Modells aufgefasst und unter der Regie eines separaten Master-Prozesses zu PaMiraXT zusammengeführt werden. Für jeden Netzwerkknoten und damit für jede MiraXT-Kopie kann eine unterschiedliche Anzahl an Threads gewählt werden, so dass sich PaMiraXT optimal an die gegebene Hardware-Plattform anpassen lässt.

Der Master-Prozess ist wie sein Pendant in PIChaff sowohl für das Starten und Stoppen der Clients als auch das Weiterleiten von Teilproblemen und Konflikt-Klauseln auf der Ebene der MiraXT-Clients verantwortlich. Die Kommunikation zwischen Master und Clients erfolgt über den Austausch von Nachrichten, das so genannte *Message Passing*, da die einzelnen Prozesse auf verschiedenen, lediglich per Ethernet-Verbindung verknüpften Rechner gestartet werden. Zur Umsetzung wurde auf das Software-Paket MPICH [48] zurückgegriffen.

Die für PaMiraXT mit drei Clients und insgesamt acht Threads durchgeführten Experimente belegen, dass bei den besonders schweren und zeitintensiven Probleminstanzen der Problemklasse *SAT 2007 Industrial* die Zahl der innerhalb von 10000 Sekunden gelösten Instanzen gegenüber der besten MiraXT-Variante (vier Threads) um weitere sechs Instanzen erhöht werden konnte. In Anbetracht des sehr hohen Zeitlimits ist dies durchaus damit gleichzusetzen, dass die entsprechenden CNF-Formeln nur durch den Einsatz von PaMiraXT überhaupt erfolgreich gelöst werden können.

Insbesondere MiraXT und PaMiraXT bieten eine vielversprechende Ausgangsbasis für weiterführende Arbeiten. Exemplarisch seien als Ausblick die beiden folgenden Punkte genannt:

– Das aktuelle Design von MiraXT sieht vor, dass zunächst sequentiell das *Preprocessing* der gegebenen CNF-Formel durchgeführt wird, bevor das verbleibende Restproblem anschließend von mehreren Threads gemeinsam bearbeitet wird. Dieses Konzept ist immer dann von Nachteil, wenn die für die Vorverarbeitung benötigte Laufzeit

die für den nachfolgenden Suchprozess aufgewendete Zeit dominiert, was einen signifikanten Performance-Gewinn durch den Einsatz mehrerer Threads während der eigentlichen Suche unmöglich macht. In diesem Zusammenhang wäre es wünschenswert, die Preprocessing-Einheit ebenfalls zu parallelisieren und auf diesem Weg das Leistungsvermögen von MiraXT zu erhöhen.

– Für PaMira, eine Vorgängerversion von PaMiraXT, wurde untersucht, welcher Effekt sich einstellt, wenn die am Suchprozess beteiligten, sequentiellen SAT-Prozeduren auf unterschiedliche Entscheidungsheuristiken zurückgreifen. In diesem Zusammenhang konnte in [95] empirisch gezeigt werden, dass eine Konfiguration von PaMira mit vier Clients, die im Rahmen der Entscheidungsheuristik unterschiedliche Strategien verwenden, etwa 70% schneller ist als eine Konfiguration, bei der die vier Clients die gleiche Strategie einsetzen. Die Ergebnisse legen nahe, diese Idee auch auf MiraXT/PaMiraXT zu übertragen und neben der Entscheidungsheuristik gegebenenfalls auch andere Komponenten eines SAT-Algorithmus wie beispielsweise das Löschen von Konflikt-Klauseln in dieses Konzept aufzunehmen.

Im Hinblick auf zukünftige Multi-Core Prozessoren mit mehreren Dutzend CPU-Kernen [115] werden parallele Algorithmen nicht nur beim Lösen von SAT-Problemen, sondern auch beim Bearbeiten dazu eng verwandter Fragestellungen wie etwa aus dem Bereich *Satisfiability Modulo Theories* [41] zunehmend an Bedeutung gewinnen. Nur bei parallelen Ansätzen ist eine optimale Ausnutzung der zur Verfügung stehenden Hardware-Ressourcen gewährleistet. Mit den in dieser Arbeit entwickelten Verfahren wurde ein wichtiger Beitrag in diese Richtung geleistet.

Literaturverzeichnis

[1] E. Abraham, T. Schubert, B. Becker, M. Fränzle, and C. Herde. Parallel SAT Solving in Bounded Model Checking. In *5th International Workshop on Parallel and Distributed Methods in Verification*, pages 301–315, 2006.

[2] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[3] T. Albrecht. Test und Einbindung der Switch-Matrix in das Multiprozessorsystem BEA. Studienarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 2001.

[4] AMD, Inc. *The AMD Opteron Processor for Servers and Workstations*, 2005.

[5] AMD, Inc. *AMD Opteron Processor Product Data Sheet*, 2006.

[6] AMD, Inc. *Next Generation AMD Opteron Processor with Direct Connect Architecture: 2P Server and Workstation Comparison*, 2006.

[7] AMD, Inc. *Next Generation AMD Opteron Processor with Direct Connect Architecture: 4P Server Comparison*, 2006.

[8] Atmel Corporation. *ATF1500A(L) Datasheet*, 1999.

[9] Atmel Corporation. *Using the ATF1500(A) CPLD*, 1999.

[10] R.J. Bayardo and R.C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *14th National Conference on Artificial Intelligence*, pages 203–2008, 1997.

[11] C.L. Berman and L.H. Trevillyan. Functional Comparison of Logic Designs for VLSI Circuits. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 456–459, 1989.

[12] A. Biere. The Evolution from LIMMAT to NANOSAT. Technical Report 444, ETH Zürich, 2004.

[13] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[14] P. Biermann, R. Drechsler, and B. Becker. Modularity as Key Element in Modern System Design – A Case Study for Industrial Application of Parallel Processing. In *European Design & Test Conference User Forum*, pages 15–20, 1997.

[15] W. Blochinger, C. Sinz, and W. Küchlin. A Universal Parallel SAT Checking Kernel. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1720–1725, 2003.

[16] D. Boggs, A. Bakhta, J. Hawkins, D.T. Marr, J.A. Miller, P. Roussel, R. Singhal, B. Toll, and K.S. Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1):1–17, 2004.

[17] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *IEEE/ACM Design Automation Conference*, pages 40–45, 1990.

[18] D. Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design*, pages 534–537, 1993.

[19] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[20] J.R. Burch and V. Singhal. Tight Integration of Combinational Verification Methods. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 570–576, 1998.

[21] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium*, pages 379–386, 1994.

[22] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[23] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[24] Compuware Corporation. *NuMega DriverAgent - User's Guide*, 1999.

[25] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[26] Cypress Semiconductor Corporation. *Datasheet for CY7C136 2K×8 Dual-Port Static RAM*, 2005.

[27] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, 1962.

[28] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[29] S. Disch and C. Scholl. Combinational Equivalence Checking Using Incremental SAT Solving, Output Ordering, and Resets. In *12th Asia and South Pacific Design Automation Conference*, pages 938–943, 2007.

[30] R. Drechsler, N. Drechsler, E. Mackensen, T. Schubert, and B. Becker. Design Reuse by Modularity: A Scalable Dynamical (Re)Configurable Multiprocessor System. In *26th Euromicro Conference*, volume 1, pages 425–431, 2000.

[31] N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *8th International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[32] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[33] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[34] S. Eggersgluss, G. Fey, and R. Drechsler. SAT-based ATPG for Path Delay Faults in Sequential Circuits. In *IEEE International Symposium on Circuits and Systems*, pages 3671–3674, 2007.

[35] P. Engelke, I. Polian, M. Renovell, and B. Becker. Automatic Test Pattern Generation for Resistive Bridging Faults. *Electronic Testing: Theory and Applications*, 22(1):61–69, 2006.

[36] K. Erk and L. Priese. *Theoretische Informatik – Eine umfassende Einführung*. Springer-Verlag, 2002.

[37] C. Faller. Die Programmierung des CPLDs PZ5128-S10BB1 in einem Multiprozessorsystem. Studienarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 1999.

[38] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2005.

[39] Freescale Semiconductor, Inc. *Integrated Processor with DMA Product Brief*, 1992.

[40] Freescale Semiconductor, Inc. *MC68340 Integrated Processor with DMA User's Manual*, 1992.

[41] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *JSAT Special Issue on SAT/CP Integration*, 1:209–236, 2007.

[42] Z. Fu, Y. Mahajan, and S. Malik. New Features of the SAT'04 Versions of zChaff. In *SAT Competition 2004 – Solver Description*, 2004.

[43] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: a User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[44] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe*, pages 142–149, 2002.

[45] C.P. Gomes and B. Selman. Algorithm Portfolio Design: Theory vs. Practice. In *13th Conference on Uncertainty in Artificial Intelligence*, pages 190–197, 1997.

[46] J.F. Groote and H. Zantema. Resolution and Binary Decision Diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.

[47] W. Gropp and E.L. Lusk. User's Guide for MPICH, a Portable Implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, 1996.

[48] W. Gropp, E.L. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[49] M. Herbstritt, T. Kmieciak, and B. Becker. On the Impact of Structural Circuit Partitioning on SAT-based Combinational Circuit Verification. In *5th International Workshop on Microprocessor Test and Verification*, pages 50–55, 2004.

[50] J.N. Hooker. Solving the Incremental Satisfiability Problem. *Journal of Logic Programming*, 15(1–2):177–186, 1993.

[51] H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. *Frontiers in Artificial Intelligence and Applications*, 63:283–292, 2000.

[52] I-Cube, Inc. *The FPID Family Datasheet*, 1994.

[53] IBM Corporation. *IBM System p5 590 and 595 Technical Overview and Introduction*, 2006.

[54] Intel Corporation. *Intel Pentium 4 Processor Supporting Hyper-Threading Technology Datasheet*, 2004.

[55] Intel Corporation. *Intel Core 2 Duo Mobile Processor Product Brief*, 2006.

[56] Intel Corporation. *Quad-Core Intel Xeon Processor 5300 Series Product Brief*, 2006.

[57] Intel Corporation. *Intel Core 2 Duo Mobile Processor for Intel Centrino Duo Mobile Processor Technology Datasheet*, 2007.

[58] Intel Corporation. *Mobile Intel 965 Express Chipset Family Datasheet*, 2007.

[59] M. Jonas. Inbetriebnahme eines Mikrocontroller-Systems mit ISA-Schnittstelle, Zugriff auf Hardware unter Windows und Entwicklung einer Arbeitsumgebung. Diplomarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 2000.

[60] B. Jurkowiak, C.M. Li, and G. Utard. Parallelizing Satz using Dynamic Workload Balancing. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, pages 205–211, 2001.

[61] H.A. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic Restart Policies. In *18th AAAI National Conference on Artificial Intelligence*, pages 674–681, 2002.

[62] H.A. Kautz and B. Selman. Planning as Satisfiability. In *10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[63] H.A. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *13th AAAI National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[64] H. Kleine Büning and T. Lettmann. *Aussagenlogik: Deduktion und Algorithmen.* B.G. Teubner, 1994.

[65] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *IEEE/ACM Design Automation Conference*, pages 263–268, 1997.

[66] T. Larrabee. Test Pattern Generation using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, 1992.

[67] R. Lawrence. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University, 2004.

[68] D. Le Berre. Exploiting the Real Power of Unit Propagation Lookahead. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[69] M. Lewis, T. Schubert, and B. Becker. Early Conflict Detection Based BCP for SAT Solving. In *7th International Conference on Theory and Applications of Satisfiability Testing*, pages 29–36, 2004.

[70] M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, pages 437–443, 2005.

[71] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *12th Asia and South Pacific Design Automation Conference*, pages 926–931, 2007.

[72] C.M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *15th International Joint Conference on Artificial Intelligence*, pages 366–371, 1997.

[73] C. Liu, A. Kuehlmann, and M.W. Moskewicz. CAMA: A Multi-Valued Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 326–333, 2003.

[74] I. Lynce and J.P. Marques-Silva. Efficient Data Structures for Backtrack Search SAT Solvers. In *5th International Symposium on Theory and Applications of Satisfiability Testing*, pages 308–315, 2002.

[75] I. Lynce and J.P. Marques-Silva. An Overview of Backtrack Search Satisfiability Algorithms. *Annals of Mathematics and Artificial Intelligence*, 37(3):307–326, 2003.

[76] I. Lynce and J.P. Marques-Silva. Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *15th IEEE International Conference on Tools with Artificial Intelligence*, pages 105–110, 2003.

[77] J.P. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *9th Portuguese Conference on Artificial Intelligence*, pages 62–74, 1999.

[78] J.P. Marques-Silva and K.A. Sakallah. Robust Search Algorithms for Test Pattern Generation. In *27th International Symposium on Fault-Tolerant Computing*, pages 152–161, 1997.

[79] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[80] J.P. Marques-Silva and K.A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *IEEE/ACM Design Automation Conference*, pages 675–680, 2000.

[81] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[82] Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. In *IEEE/ACM Design Automation Conference*, pages 629–634, 1996.

[83] C. Meinel and M. Mundhenk. *Mathematische Grundlagen der Informatik*. B.G. Teubner, 2002.

[84] Microchip Technology, Inc. *PIC17C4X Datasheet*, 1996.

[85] Microchip Technology, Inc. *MPLAB IDE User's Guide*, 2006.

[86] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *IEEE/ACM Design Automation Conference*, pages 530–535, 2001.

[87] V. Paruthi and A. Kuehlmann. Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation. In *IEEE International Conference on Computer Design*, pages 459–464, 2000.

[88] T. Schubert. Konfiguration eines Multiprozessorsystems und Integration von genetischen Algorithmen. Diplomarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 1999.

[89] T. Schubert and B. Becker. A Distributed SAT Solver for Microcontrollers. In *7th Workshop on Parallel Systems and Algorithms*, pages 338–347, 2004.

[90] T. Schubert and B. Becker. Parallel SAT Solving with Microcontrollers. In *2nd Asian Applied Computing Conference*, pages 59–67, 2004.

[91] T. Schubert and B. Becker. PICHAFF^2 – A Hierarchical Parallel SAT Solver. In *5th International Workshop on Microprocessor Test and Verification*, pages 56–61, 2004.

[92] T. Schubert and B. Becker. Knowledge Sharing in a Microcontroller based Parallel SAT Solver. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1049–1055, 2005.

[93] T. Schubert and B. Becker. Lemma Exchange in a Microcontroller based Parallel SAT Solver. In *IEEE Symposium on VLSI*, pages 142–147, 2005.

[94] T. Schubert, M. Lewis, and B. Becker. Accelerating Boolean SAT Engines Using Hyper-Threading Technology. In *3rd Asian Applied Computing Conference*, pages 58–62, 2005.

[95] T. Schubert, M. Lewis, and B. Becker. PaMira – a Parallel SAT Solver with Knowledge Sharing. In *6th International Workshop on Microprocessor Test and Verification*, pages 29–34, 2005.

[96] T. Schubert, E. Mackensen, N. Drechsler, R. Drechsler, and B. Becker. Specialized Hardware for Implementation of Evolutionary Algorithms. In *Genetic and Evolutionary Computing Conference*, page 369, 2000.

[97] U. Schöning. *Logik für Informatiker.* Spektrum Akademischer Verlag, 1995.

[98] U. Schöning. *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag, 1995.

[99] B. Selman, H.A. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:521–532, 1996.

[100] B. Selman, H.J. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[101] M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. In *Formal Methods in Computer Aided Design*, pages 82–99, 1998.

[102] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel. PASSAT: Efficient SAT-based Test Pattern Generation for Industrial Circuits. In *IEEE Symposium on VLSI*, pages 212–217, 2005.

[103] O. Shtrichman. Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70, 2001.

[104] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT – Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[105] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995.

[106] F. Somenzi. Efficient Manipulation of Decision Diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[107] P. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Combinational Test Generation using Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[108] S. Subbarayan and D. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances. In *7th International Conference on Theory and Applications of Satisfiability Testing*, pages 276–291, 2004.

[109] Sun Microsystems, Inc. *The UltraSPARC T1 Processor – Power Efficient Throughput Computing*, 2005.

[110] The International SAT Competitions Web Page. http://www.satcompetition.org/.

[111] The MiniSat Web Page. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/.

[112] The PicoSAT Web Page. http://fmv.jku.at/picosat/.

[113] The RSat Web Page. http://reasoning.cs.ucla.edu/rsat/.

[114] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Automation of Reasoning 2: Classical Papers on Computational Logic*, pages 466–483, 1983.

[115] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65 CMOS. In *IEEE International Solid-State Circuits Conference*, pages 98–100, 2007.

[116] O. Wechsler. Inside Intel Core Microarchitecture: Setting New Standards for Energy-Efficient Performance. *Technology@Intel Magazine*, pages 1–11, 2006.

[117] I. Wegener. *Theoretische Informatik – Eine algorithmenorientierte Einführung*. B.G. Teubner, 1993.

[118] I. Wegener. *Kompendium Theoretische Informatik – Eine Ideensammlung*. B.G. Teubner, 1996.

[119] Xilinx, Inc. *CoolRunner XPLA3 CPLD*, 2005.

[120] H. Zhang. SATO: An Efficient Propositional Prover. In *International Conference on Automated Deduction*, pages 272–275, 1997.

[121] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[122] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam Method. *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.

[123] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, 2001.

[124] L. Zhang and S. Malik. Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 287–298, 2003.

[125] L. Zhang and S. Malik. Validating SAT Solvers using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Design, Automation, and Test in Europe*, pages 10880–10885, 2003.

Abbildungsverzeichnis

Tabellenverzeichnis