SoSy-Lab Software Systems
Software and Computational Systems Lab
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

MASTERARBEIT

Design und Implementierung einer parallelen BDD-Bibliothek

Stephan Holzner

SoSy-Lab Software Systems
Software and Computational Systems Lab
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

MASTERARBEIT

Design und Implementierung einer parallelen BDD-Bibliothek

Stephan Holzner

Aufgabensteller:	Prof. Dr. Dirk Beyer
Betreuer:	Karlheinz Friedberger
Abgabetermin:	15.05.2019

Abstract

The present master thesis investigates the implementation of a competitive and lightweight Binary-Decision-Diagram (BDD) library with modern Java features. Therefore several parallelization and weak reference technologies are used to implement different versions of the BDD framework, which support multithreading and fully automated resource management.
To determine the best variation the differing realizations were evaluated with benchmarks. The most efficient solution was used to compare the developed library Parallel-Java-BDD (PJBDD) with some state-of-the-art frameworks in addition. Finally an integration into CPAchecker has been used to assess live performance in comparison to JavaBDD.
The results approve the competitive performance of PJBDD in small as well as in huge applications. PJBDD also outran the state-of-the-art libraries for several tasks. Furthermore PJBDD supports multiple features to provide easy application.
This work proves the potential of recent Java technologies to develop a lightweight BDD library, that can keep up with heavy optimized C/C++ realizations on modern hardware.

Abstrakt

Die vorliegende Masterarbeit untersucht, ob mit Hilfe neuester Java-Technologien eine Binary-Decision-Diagram (BDD)-Bibliothek implementiert werden kann, die Multithreating unterstützt, entwickler- wie benutzerfreundlich ist und dennoch konkurrenzfähig zu den hoch optimierten Standardbibliotheken ist. Um alternative Ausführungen des BDD-Frameworks mit parallelen Algorithmen und automatisierter Speicherverwaltung zu realisieren, wurden unterschiedlichste Java-Konzepte zur Parallelisierung sowie zur schwachen Referenzierung eingesetzt.
Die umgesetzten Versionen wurden evaluiert und die besten Alternativen aktuellen Standards gegenübergestellt. Im letzten Schritt wurde eine CPAchecker-Integration entwickelt, um Parallel-Java-BDD (PJBDD) und JavaBDD mittels Benchmarking zu vergleichen.
Die Ergebnisse zeigen, dass PJBDD sowohl in kleinen, als auch in großen Anwendungsfällen dank der Parallelisierung konkurrenzfähig ist. Ferner konnten bestimmte Aufgaben mit PJBDD deutlich schneller gelöst werden, als mit aktuellen Standardbibliotheken. Zugleich war es möglich die Anwendung hinsichtlich mehrerer Gesichtspunkte zu vereinfachen.
Die Arbeit beweist, dass aktuelle Java-Konzepte durchaus genutzt werden können, um eine bedienbaren BDD-Bibliothek zu implementieren, die auf moderner Hardware selbst mit hoch optimierten C/C++-Umsetzungen mithalten kann.

Inhaltsverzeichnis

BDD Binary-Decision-Diagram

ROBDD Reduced-Ordered-Binary-Decision-Diagram

PJBDD Parallel-Java-BDD

ZDD Zero-Surpressed-Binary-Decision-Diagram

ITE If-Then-Else

CPA Configurable-Program-Analysis

JMH Java-Microbenchmark-Harness

JVM Java-Virtual-Machine

RSS Resident-Set-Size

NUMA Nonuniform-Memory-Access

JIT Just-In-Time

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Viele komplexe Anwendungsfälle in der Soft- und Hardwareverifikation können durch eine Vielzahl möglicher Systemzustände zu sogenannten Zustandsexplosionen führen. Die Synthese dieser Probleme erfordert nicht nur eine kompakte Darstellung, sondern auch die effiziente Manipulation der Zustände. Für die Hardwareverifikation entwickelten Lee (Lee59) und Akers (Ake78) mit den Binary-Decision-Diagrams (BDDs) eine performante Repräsentation für boolesche Variablen und Operationen.
Neue Verifikationstechniken in der Softwareanalyse profitieren von der Nutzung dieser komprimierten Datenstruktur. Für das Model-Checking stellen Beyer, Apel et al. in (BS12) und (ABF$^+$13) beispielsweise einen hybriden Ansatz vor, der BDDs als Hilfsdatenstruktur zur Abbildung bestimmter Variablen nutzt. Im Model-Checking wird ein System analysiert, in dem alle möglichen Programmzustände automatisiert generiert und hinsichtlich einer vordefinierten Spezifikation geprüft werden.
Allgemein werden BDDs zur Darstellung von Entscheidungsproblemen eingesetzt. Alternative Mittel zur Lösung von Problemen dieser Kategorie sind SMT-Solver, Zero-Surpressed-Binary-Decision-Diagrams (ZDDs) oder Linear-Decision-Diagrams (LDDs). SMT-Solver werden genutzt, um die Erfüllbarkeit von Variablenbeschränkungen unterschiedlichster Datentypen zu prüfen und ZDDs sowie LDDs sind spezielle Arten von BDDs. ZDDs bieten eine effiziente Form zur Repräsentation boolescher Lösungsmengen und LDDs halten Anstelle einer Variable logische Aussagen, die es zu validieren gilt.

Viele BDD-Frameworks ziehen noch keinen Nutzen aus den in den letzten Jahren extrem gestiegenen Rechenressourcen sowie neu aufgekommenen Programmierkonzepten. Meist wird lediglich auf einem einzigen Prozessorkern gerechnet und eine manuelle Speicherbereinigung erfordert.
Die beiden C/C++ BDD-Standardbibliotheken BuDDy[1] und CuDD[2] nutzen beispielsweise lediglich sequentielle Berechnungen der booleschen Operationen und setzen voraus, dass der Anwender dafür Sorge trägt nicht mehr verwendete Knoten zu löschen. Das gleiche gilt für die Java-Versionen JavaBDD[3] und JDD[4]. Die Java-Bibliothek BeeDeeDee[5] bietet zumindest eine threadsichere Möglichkeit zur parallelen Berechnung mehrerer Operationen, jedoch ohne nebenläufige Algorithmen. Das C-Framework Sylvan[6] von Tom van Dijk bietet ein interessantes Konzept zur Parallelisierung der BDD-Manipulation mit manueller Speicherbereinigung.

Im Rahmen dieser Arbeit wird mit Hilfe moderner Java-Technologien eine BDD-Bibliothek

[1] http://buddy.sourceforge.net/manual/main.html Abrufdatum 30.04.2019
[2] http://davidkebo.com/cudd Abrufdatum 30.04.2019
[3] http://javabdd.sourceforge.net Abrufdatum 30.04.2019
[4] https://bitbucket.org/vahidi/jdd/overview Abrufdatum 30.04.2019
[5] https://github.com/JuliaSoft/BeeDeeDee Abrufdatum 30.04.2019
[6] http://trolando.github.io/sylvan Abrufdatum 30.04.2019

entwickelt, die parallele Algorithmen zur BDD-Manipulation implementiert, automatische Speicherverwaltung bietet und mittels moderner Programmierkonzepte eine einfache Anwendung, Anpassung und Erweiterung unterstützt.
Im zweiten Abschnitt werden einige BDD- und Java-spezifische Grundlagen vorgestellt. In Kapitel 3 werden dann der Implementierungsprozess und die verwendeten Technologien beschrieben, bevor die Realisierung der parallelen Bibliothek PJBDD ausführlich dokumentiert wird. Im Analysekapitel 5 werden zunächst die Konfigurationsmöglichkeiten von PJBDD evaluiert und anschließend mit bereits bestehenden Frameworks verglichen, um Vor- und Nachteile herauszuarbeiten. Zum Abschluss wird PJBDD in das Framework CPAchecker integriert und zur Softwareanalyse verwendet.

2 Grundlagen

Dieses Kapitel bespricht die für das Verständnis der Arbeit notwendigen Grundlagen. Dafür werden im ersten Schritt Binary-Decision-Diagrams (BDDs), Zero-Surpressed-Binary-Decision-Diagrams (ZDDs) und zugehörige Konzepte skizziert, ehe der letzte Punkt noch einige spezielle Java-Funktionalitäten vorstellt.

2.1 Binary-Decision-Diagram

Ein BDD ist eine effiziente Datenstruktur zur Darstellung oder Manipulation boolescher Funktionen und Variablen. Jedes BDD besteht aus einem Wurzelknoten *root*, einer boolesche Variable x und genau zwei Nachfolgeknoten, *low* und *high* oder ist ein Blatt. Darüber hinaus ist jeder Nachfolger für sich ein eigenständiges BDD. Dadurch wird ein BDD zu einem gerichteten azyklischen Graphen, dessen Pfade immer in denselben Blattknoten enden. Ein Endpunkt entspricht entweder dem logischen *true* (1) oder dem logischen *false* (0). (Knu09)
Die Kanten zu den beiden Nachfolgeknoten repräsentieren die Variablenbelegung von x. Das heißt, wird x mit *true* belegt kann das BDD auf den *high*-Nachfolger beschränkt werden und die Belegung mit $false$ führt zum *low*-Knoten. Dadurch beschreibt der Wurzelknoten *root* die Funktion $f(x_{root})$:

$$f(x_{root}) = \begin{cases} high_{root} & \text{if } x_{root} = 1; \\ low_{root} & \text{else;} \end{cases} \tag{2.1}$$

Diese Darstellung wird auch als If-Then-Else-Normalform bzw. Shannon-Dekomposition bezeichnet. (And97)
Die Grafiken 2.1 und 2.2 zeigen die BDD-Darstellung der klassischen booleschen Funktionen $f = a \wedge b$ (2.1) und $f = a \vee b$ (2.2).

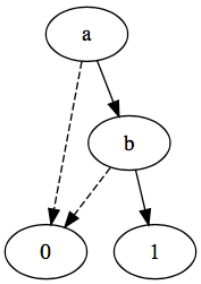


Abbildung 2.1: $f = a \wedge b$

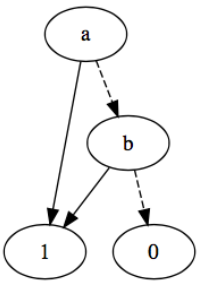


Abbildung 2.2: $f = a \vee b$

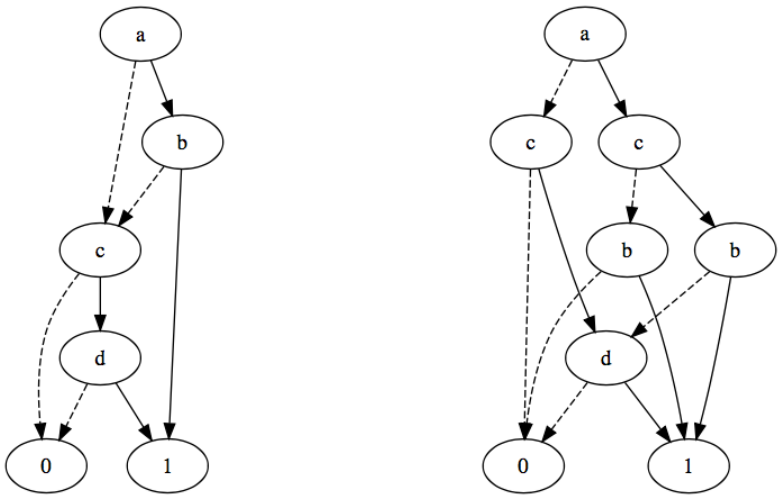
Die low-Kante wird meist gestrichelt dargestellt und die high durchgezogen.
BDDs unterstützen die Darstellung und Manipulation aller boolescher Funktionen sowie

Variablenoperationen.
In der Literatur wie auch in dieser Arbeit werden BDDs meist mit Reduced-Ordered-Binary-Decision-Diagrams (ROBDDs) gleichgesetzt, wodurch die Verwendung zweier Optimierungen impliziert wird: Ein BDD ist genau dann ein ROBDD, wenn alle Knoten reduziert wurden und die Variablenordnung fest ist. Ein reduzierter BDD-Graph muss gemäß Andersen (And97) zwei Grundregeln erfüllen:

1. Zusammenfassung aller isomorphen Knoten, das bedeutet, jeder verbleibende ist einzigartig.

2. Eliminierung aller Subgraphen mit redundanten Kindern, indem eingehende Kanten direkt zu diesen geführt werden.

Diese Reduzierung wird in der Regel mit einer Knotentabelle, auch Uniquetable bezeichnet, gewährleistet.
Ein BDD gilt als geordnet, wenn alle Variablen jedes Pfades von der Wurzel bis zu den Blättern gleich sortiert sind, ohne dass jeder Pfad alle Variablen verwendet. (And97)
Die Ordnung der Variablen hat große Auswirkungen auf die Größe eines BDD. Die beiden Bilder in 2.3 repräsentieren zwei BDDs derselben Funktion $f = (a \wedge b) \vee (c \wedge d)$ mit unterschiedlicher Variablenordnung.



(a) F mit Ordnung $a > b > c > d$ (b) G mit Ordnung: $a > c > b > d$

Abbildung 2.3: BDDs der Funktion $f = (a \wedge b) \vee (c \wedge d)$ (Quelle: eigene Darstellung)

Für das BDD G in 2.3b wurde lediglich die Ordnung der Variablen b und c getauscht mit der Folge, dass die Kardinalität gegenüber F um den Faktor 1,5 steigt ($|G| = 1,5 * |F|$). Das bedeutet, dass mit einer Änderung der Variablensortierung alle bereits existierenden BDDs im selben Kontext angepasst werden müssen.
Ein einfacher Algorithmus dafür ist das Vertauschen benachbarter Variablen. Hierfür beginnt man mit der alten Sortierung und tauscht die differierenden Variablen so oft mit deren Nachbarn, bis die gewünschte neue Ordnung herrscht. Mit jedem dieser Variablenwechsel müssen alle betroffenen Knoten des BDD-Baums verändert werden.

Ein Knoten ist genau dann betroffen, wenn dessen Variablenlevel steigt und das eines seiner Kindknoten sinkt. (Knu09)

Weitere Optimierungsmöglichkeiten bieten ein Cache zur temporären Speicherung bereits berechneter Operationen sowie die Erkennung äquivalenter Funktionsaufrufe. Letzteres kann helfen die Größe des Caches zu reduzieren und dennoch eigentlich identische Berechnungen zu vermeiden. Mit geeigneter Cachegröße und einer guten Variablenordnung können die exponentiellen Operationen fast linear zur BDD-Größe gehalten werden.

BDDs finden unter anderen zur effizienten Lösung kombinatorischer Probleme sowie in der Hard- und Softwareverifikation Anwendung.
Eine besondere Methode zur BDD-Manipulation ist der If-Then-Else (ITE)-Operator.

2.1.1 If-Then-Else-Operator

Der ITE-Operator wird durch die Funktion 2.2 definiert.

$$f(I) \rightarrow T, E = (I \wedge T) \vee (\neg I \wedge E) \tag{2.2}$$

Die Argumente I, T, E repräsentieren je ein BDD.
Mit der Zerlegung nach Shannon kann der Operator rekursiv über alle Kindknoten der drei BDDs aufgebaut werden:

$$ite(F, G, H) = (v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \tag{2.3}$$

Wobei v die kleinste Variable der Knoten F, G, H ist und ein BDD durch das Tripple $(Variable, High, Low)$ beschrieben wird. Diese Rekursion ist solange durchzuführen, bis eine der folgenden Abbruchbedienungen erfüllt ist:

$$iteTerminal(F, G, H) = \begin{cases} G & F = 1 \vee G = H; \\ H & F = 0; \\ F & G = 1 \wedge H = 0; \end{cases} \tag{2.4}$$

Die Laufzeitkomplexität des Algorithmus wird mit $O(|I| * |T| * |E|)$ beschrieben. (And97)
Das Besondere an diesem Operator ist, dass mit Hilfe der logischen Konstanten 0 und 1 alle binären booleschen Funktionen abgebildet werden können. Tabelle 2.1 zeigt die ITE-Darstellung aller booleschen Operationen.

2.1.2 Apply-Algorithmus

Alternativ zum ITE lassen sich alle booleschen Operationen mit dem generalisierten Algorithmus $apply(F, G, OP)$ implementieren, wobei OP der anzuwendenden Operation entspricht und F, G BDD-Darstellungen sind.
Apply nutzt ebenfalls die Shannon-Dekomposition, um den gewünschten Operator rekursiv über alle Kindknoten von F und G zu spannen.

$$apply(F, G, OP) = (v, apply(F_v, G_v, OP), apply(F_{\overline{v}}, G_{\overline{v}}, OP)) \tag{2.5}$$

Bezeichnung	Ausdruck	ITE-Darstellung
$AND(F,G)$	$F \cdot G$	$ITE(F,G,0)$
$F < G$	$\overline{F} \cdot G$	$ITE(F,0,G)$
$XOR(F,G)$	$F \oplus G$	$ITE(F,\overline{G},G)$
$OR(F,G)$	$F + G$	$ITE(F,1,G)$
$NOR(F,G)$	$\overline{F+G}$	$ITE(F,0,\overline{G})$
$XNOR(F,G)$	$\overline{F \oplus G}$	$ITE(F,G,\overline{G})$
$NOT(F)$	$\overline{F}$	$ITE(F,0,1)$
$F \leq G$	$\overline{F} + G$	$ITE(F,G,1)$
$NAND(F,G)$	$\overline{F \cdot G}$	$ITE(F,\overline{G},1)$

Tabelle 2.1: ITE-Darstellung der booleschen Algebra (Quelle: (BRB90))

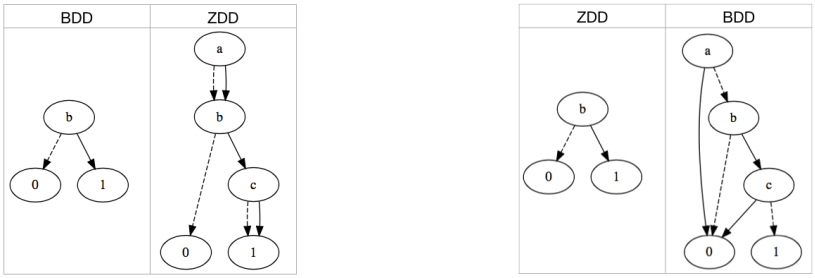
Der Unterschied zur ITE-Umsetzung ist, dass für jeden Operator OP eigene Terminalfälle definiert werden müssen. Vorteil dieser Methode ist eine Reduktion der Laufzeit auf $O(|F| * |G|)$. (And97)
Ein praktischer Vergleich der beiden Algorithmen folgt in Kapitel 5.

2.1.3 Zero-Surpressed-Binary-Decision-Diagram

Zero-Surpressed-Binary-Decision-Diagrams (ZDDs) sind eine besondere Form von BDDs mit dem Hauptunterschied, dass die zweite Reduktionsregeln ersetzt wird. Hier werden stellvertretend für Knoten mit redundanten Kindknoten jene mit $high = 0$ eliminiert.
Es hilft darüber hinaus ZDDs nicht als boolesche Funktion zu sehen, sondern vielmehr als Repräsentation der Lösungsmenge zu verstehen. (Knu09)
Jede BDD-Operation hat ein Pendant für ZDDs, jedoch unterscheiden sich diese teilweise deutlich. Da ZDDs und deren Anwendung im Allgemeinen nicht so geläufig sind wie BDDs, werden die Unterschiede dieser Entscheidungsdiagramme und die ZDD-Operationen in den folgenden Unterabschnitten ausführlich betrachtet.

2.1.4 Anwendungsvergleich BDDs und ZDDs

Aufgrund der divergierenden Reduktionsregeln ergeben sich Differenzen in der Anwendung von BDDs und ZDDs. Dies wird bereits mit der unterschiedlichen Bedeutung der jeweiligen Variablen deutlich. Die beiden Vergleiche in Abbildung 2.4 veranschaulichen



(a) Darstellung der BDD-Variable b (b) Darstellung der ZDD-Variable b

Abbildung 2.4: BDD- und ZDD-Darstellungen der Variable b (Quelle: eigene Darstellung)

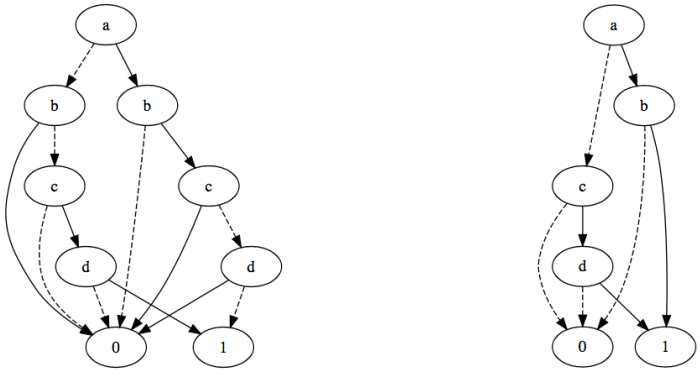
diesen Unterschied für die BDD- und ZDD-Variable b mit Variablenordnung $a > b > c$. Die Abwesenheit einer Variable im BDD bedeutet, dass deren Belegung keine Auswirkung auf das Diagramm hat. Im ZDD hingegen impliziert das Fehlen, dass eine Variablenbelegung mit *true* zum 0-Terminalblatt führt. (Mis01)

Für manche boolesche Funktionen, allen voran solche mit vergleichsweise großen Lösungsmengen, bedeutet das, dass ein BDD-Baum die kompaktere Darstellung bietet. (Knu09) Abbildung 2.5 veranschaulicht diesen Sachverhalt exemplarisch für die Funktion $f = (a \wedge b) \vee (c \wedge d)$ mit Lösungsmenge $F = \{ab\overline{c}\overline{d}, abc\overline{d}, abcd, \overline{a}bcd, \overline{a}\overline{b}cd, a\overline{b}cd\}$:

(a) BDD (b) ZDD

Abbildung 2.5: BDD und ZDD der Funktion $f = (a \wedge b) \vee (c \wedge d)$
(Quelle: eigene Darstellung nach (Mis01))

Für die Darstellung dieser Funktion ist das ZDD deutlich größer als das BDD. Vergleicht man die Lösungsmenge F mit dem ZDD in 2.5b und die Funktion f mit dem BDD in 2.5a, ist außerdem deutlich zu erkennen, dass BDDs Darstellungen von Funktionen sind und ZDDs die der zugehörigen Lösungsmengen. (Knu09)
Im Gegensatz dazu sind ZDDs performanter, wenn die Lösungsmenge vergleichsweise gering ist. Dies zeigt zum Beispiel die Funktion $g = (a \wedge b \wedge \overline{c} \wedge \overline{d}) \vee (\overline{a} \wedge \overline{b} \wedge c \wedge d)$ bzw. die Menge $G = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}cd\}$ in Abbildung 2.6.



(a) BDD (b) ZDD

Abbildung 2.6: BDD und ZDD der Menge $G = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}cd\}$
(Quelle: eigene Darstellung nach (Mis01))

In der Literatur werden Mengen wie G oft als Cover bezeichnet und die einzelnen Bestandteile (z.B. $ab\overline{c}\overline{d}$) als Cubes. Im Folgenden bezeichnen Großbuchstaben Cover, Kleinbuchstaben Cubes und die einzelnen Bestandteile eines Cubes Literale.
Da jede boolesche Funktion durch ihre Lösungsmenge beschrieben werden kann (und vice versa), kann man auch jedes BDD als ZDD darstellen und umgekehrt. Daraus folgt jedes BDD-Konzept besitzt ein anwendbares Pendant für ZDDs, es muss jedoch der oben beschriebener Unterschied der Variablenbedeutung beachtet werden. Auch ist zu bedenken, dass die ZDD-Operationen auf Mengen definiert sind und die Algorithmen von denen der BDD-Methoden abweichen. (Knu09)
Deshalb formuliert Minato in (Min93) eine Reihe an Methoden, die unter Beachtung der Variablenunterschiede äquivalent zu booleschen Funktionen verwendet werden können. Tabelle 2.2 zeigt eine Gegenüberstellung dieser:

Notation	**ZDD-Operation**	**BDD-Operation**	**Ausdruck**
$\emptyset$	Empty()	Zero()	0
$\{\emptyset\}$	Base()	One()	1
F_{v_0}	Subset0(F, v)	Restrict(F, v, 0)	F_{v_0}
F_{v_1}	Subset1(F, v)	Restrict(F, v, 1)	F_{v_1}
$F\&G$	Intersection(F, G)	And (F, G)	$F \wedge G$
$F + G$	Union(F, G)	Or (F, G)	$F \vee G$
$F - G$	Difference(F, G)	And (F, Not(G))	$F \wedge \overline{G}$
$F \oplus G$	$F + G - (F\&G)$	Xor (F, G)	$F \oplus G$

Tabelle 2.2: Gegenüberstellung grundlegender ZDD- und BDD-Operationen (Quelle: eigene Darstellung)

Etwas umständlicher ist die Umsetzung des für BDDs sehr wichtigen NOT-Operators. Denn ZDDs behandeln im Gegensatz zu BDDs abwesende Variablen nicht als Don't Care, sondern als mit false belegt, also müssen sie ebenfalls invertiert werden. Dafür kann die universelle Menge U wie folgt verwendet werden:

$$\overline{F} = U - F \tag{2.6}$$

Wobei $U_{x_1,...,x_n} = f(x_1, ..., x_n) = x_1 \wedge ... \wedge x_n = 1$.
Jedoch bringt das einen gravierenden Nachteil mit sich. Die Repräsentation der universellen Menge ist von den eingeführten Variablen abhängig, sprich ein ZDD-Komplement gilt nur für eine bestimmte Anzahl und Sortierung der Variablen.(Min93)
In diesem Abschnitt wurde die analoge Verwendung von ZDDs zu BDDs gezeigt. Um jedoch das volle Potential der ZDDs auszuschöpfen, werden nachfolgend einige ZDD-spezifische Operationen vorgestellt.

2.1.5 Weitere ZDD-spezifische Operationen

In den beiden Artikeln (Min94) und (OMI98) führen Minato und Okuno et al. einige Methoden ein, die speziell an die ZDD-Reduktion angepasst wurden und ein enormes Optimierungspotential bieten. In (Min94) entwickelt Minato drei performante Mengenoperationen:

$F * G$	Kombination aller Cubes zweier Mengen.
F/G	Schnittmenge der Quotienten der Divisionen von F durch jeden Cube in G.
$F\%G$	Restwert der Division F durch G.

In der folgenden Aufstellung werden diese neuen Operationen detailliert betrachtet.

- **Produkt** $F * G$:
  Erstellt alle möglichen Kombinationen der positiven Literale der Cubes zweier Mengen. Folgendes Beispiel dient zur Veranschaulichung:

$$\begin{aligned} &F = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}c\overline{d}\}, G = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}\overline{c}d\} \\ &F * G = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}c\overline{d}\} * \{ab\overline{c}\overline{d}, \overline{a}\overline{b}\overline{c}d\} \\ &= (ab\overline{c}\overline{d} * ab\overline{c}\overline{d}) + (ab\overline{c}\overline{d} * \overline{a}\overline{b}\overline{c}d) + (\overline{a}\overline{b}c\overline{d} * ab\overline{c}\overline{d}) + (\overline{a}\overline{b}c\overline{d} * \overline{a}\overline{b}\overline{c}d) \\ &= \{ab\overline{c}\overline{d}, ab\overline{c}d, abc\overline{d}, \overline{a}\overline{b}cd\} \end{aligned} \tag{2.7}$$

  Das Produkt kann formal demnach wie folgt formuliert werden:

$$\begin{aligned} &F * G = (\forall f \in F, \forall g \in G) \bigvee (f * g) \\ &f * f = f \\ &f * \overline{f} = f \end{aligned} \tag{2.8}$$

- **Quotient der Division** F/G:
  Bildet die Schnittmenge aller Quotienten der Division einer Menge durch die einzelnen Cubes einer anderen. Bei der Division einer Menge M durch einen Cube c werden alle Cubes aus M extrahiert, die alle positiven Literale von c beinhalten bevor c aus diesen subtrahiert wird. Die Operation soll erneut anhand eines Beispiels verdeutlicht werden:

$$\begin{aligned} &F = \{abc\overline{d}, \overline{a}\overline{b}cd, \overline{a}bc\overline{d}, a\overline{b}\overline{c}d\}, G = \{ab\overline{c}\overline{d}, \overline{a}\overline{b}\overline{c}d\} \\ &F/G = \{abc\overline{d}, \overline{a}\overline{b}cd, \overline{a}bc\overline{d}, a\overline{b}\overline{c}d\}/\{ab\overline{c}\overline{d}, \overline{a}\overline{b}\overline{c}d\} \\ &= (\{abc\overline{d}, \overline{a}\overline{b}cd, \overline{a}bc\overline{d}, a\overline{b}\overline{c}d\}/\{ab\overline{c}\overline{d}\}) \ \& \ (\{abc\overline{d}, \overline{a}\overline{b}cd, \overline{a}bc\overline{d}, a\overline{b}\overline{c}d\}/\{\overline{a}\overline{b}\overline{c}d\}) \\ &= (\{(abc\overline{d} - ab\overline{c}\overline{d})\}) \ \& \ (\{(\overline{a}\overline{b}cd - \overline{a}\overline{b}\overline{c}d), (a\overline{b}\overline{c}d - \overline{a}\overline{b}\overline{c}d)\}) \\ &= \{\overline{a}\overline{b}c\overline{d}\} \ \& \ \{\overline{a}\overline{b}cd, a\overline{b}\overline{c}\overline{d}\} \\ &= \{\overline{a}\overline{b}c\overline{d}\} \end{aligned} \tag{2.9}$$

  Die formale Definition der Division lautet:

$$\begin{aligned} &F/G = (\forall f \in F, \forall g \in G) \bigwedge (f/g) \\ &f/g = \begin{cases} f - g & \text{if } f \succeq g; \\ \emptyset & \text{else;} \end{cases} \end{aligned} \tag{2.10}$$

  Wobei $f \succeq g$ bedeutet, dass der Cube f alle positiven Literale aus g beinhaltet.

- **Modulo der Division** $F\%G$:
  Bildet die Restwertmenge der Division zweier Mengen. Dementsprechend kann der

Restwert wie folgt berechnet werden:

$$F\%G = F - F * (F/G) \tag{2.11}$$

Wenn ZDDs verwendet werden, um die Lösungsmenge sehr großer Entscheidungsprobleme wie dem N-Damenproblem darzustellen, kann die temporäre Kombinationsmenge so groß werden, dass die Berechnung abbricht, obwohl das Endergebnis in den Systemspeicher passen würde.
Um in diesen Fällen die zwischenzeitliche Kombinationsmenge effektiv zu beschränken, stellen Okuno et al. in (OMI98) den Restriction- und Exclusion-Operator vor:

- **Restriction** $F \triangle G$:
  Berechnet eine Teilmenge von F, die mindestens eine Beschränkung aus G erfüllt, wobei G eine Menge möglicher Kombinationen beziehungsweise Cubes darstellt.
  Die formale Definition lautet $F \triangle G \equiv \{f \in F | \exists g \in G, f \succeq g\}$.
  Zur Veranschaulichung dient nachstehendes Beispiel mit $F = \{ab\bar{c}\bar{d}, abc\bar{d}, \bar{a}bcd\}$ und $G = \{abc\bar{d}, \bar{a}bc\bar{d}\}$:

$$\begin{aligned}
&F \triangle G = \{ab\bar{c}\bar{d}, abc\bar{d}, \bar{a}bcd\} \triangle \{abc\bar{d}, \bar{a}bc\bar{d}\} \\
&= (ab\bar{c}\bar{d} \triangle abc\bar{d}) + (abc\bar{d} \triangle abc\bar{d}) + (\bar{a}bcd \triangle abc\bar{d}) + (ab\bar{c}\bar{d} \triangle \bar{a}bc\bar{d}) \\
&+ (abc\bar{d} \triangle \bar{a}bc\bar{d}) + (\bar{a}bcd \triangle \bar{a}bc\bar{d}) \\
&= \cancel{(ab\bar{c}\bar{d} \triangle abc\bar{d})} + (abc\bar{d} \triangle abc\bar{d}) + \cancel{(\bar{a}bcd \triangle abc\bar{d})} + \cancel{(ab\bar{c}\bar{d} \triangle \bar{a}bc\bar{d})} \\
&+ (abc\bar{d} \triangle \bar{a}bc\bar{d}) + (\bar{a}bcd \triangle \bar{a}bc\bar{d}) \\
&= (abc\bar{d} \triangle abc\bar{d}) + (abc\bar{d} \triangle \bar{a}bc\bar{d}) + (\bar{a}bcd \triangle \bar{a}bc\bar{d} \\
&= \{abc\bar{d}, \bar{a}bcd\}
\end{aligned} \tag{2.12}$$

- **Exclusion** $F \bigtriangledown G$:
  Berechnet eine Teilmenge von F aus der alle Elemente, die mindestens eine Beschränkung aus G erfüllen, entfernt werden, wobei G wieder eine Menge möglicher Kombinationen darstellt.
  Formal wird die Exclusion durch $F \bigtriangledown G \equiv \{f \in F | \nexists g \in G, f \succeq g\}$ definiert.
  Zur Veranschaulichung dient erneut ein Beispiel mit $F = \{ab\bar{c}\bar{d}, abc\bar{d}, \bar{a}bcd\}$ und $G = \{abc\bar{d}, \bar{a}bc\bar{d}\}$:

$$\begin{aligned}
&F \bigtriangledown G = \{ab\bar{c}\bar{d}, abc\bar{d}, \bar{a}bcd\} \bigtriangledown \{abc\bar{d}, \bar{a}bc\bar{d}\} \\
&= ((ab\bar{c}\bar{d} \bigtriangledown abc\bar{d}) \ \& \ (ab\bar{c}\bar{d} \bigtriangledown \bar{a}bc\bar{d})) + ((abc\bar{d} \bigtriangledown abc\bar{d}) \ \& \ (abc\bar{d} \bigtriangledown \bar{a}bc)) \\
&+ ((\bar{a}bcd \bigtriangledown abc\bar{d}) \ \& \ (\bar{a}bcd \bigtriangledown \bar{a}bc\bar{d})) \\
&= (ab\bar{c}\bar{d} \ \& \ ab\bar{c}\bar{d}) + (\emptyset \ \ \& \ \emptyset) + (\bar{a}bcd \ \& \ \emptyset) \\
&= \{ab\bar{c}\bar{d}\}
\end{aligned} \tag{2.13}$$

Mit diesen beiden Methoden wurden die wichtigsten ZDD-Operationen vorgestellt. Tabelle 2.3 fasst sie abschließend zusammen.

Op	Methode	Beschreibung
$\emptyset$	Empty()	leere Menge (0)
$\{\emptyset\}$	Base()	leere Familie (1)
U	Universe()	universelle Menge
F_{v_0}	Subset0(F, v)	beschränkt Variable v der Menge F mit 0
F_{v_1}	Subset1(F, v)	beschränkt Variable v der Menge F mit 1
F'	Change(F, v)	tauscht die Kindknoten aller Knoten mit Variable v
$F\&G$	Intersection(F, G)	Schnittmenge von F und G
$F + G$	Union(F, G)	Vereinigung von F und G
$F - G$	Difference(F, G)	Differenz von F und G
$F * G$	Product(F, G)	Kombination alle Teilmengen von F und G
F/G	Division(F, G)	Quotienten der Division von F durch G
$F\%G$	Modulo(F, G)	Restwert der Division F durch G
$F \triangle G$	Restrict(F, G)	Teilmenge von F, die eine Beschränkung von G erfüllt
$F \bigtriangledown G$	Exclude(F, G)	Teilmenge von F, die keine Beschränkung von G erfüllt

Tabelle 2.3: Kompakte Zusammenfassung aller ZDD-Methoden
(Quelle: eigene Darstellung)

Zusätzlich zu diesen theoretischen Grundlagen ist für die Entwicklung einer parallelen Java BDD-/ZDD-Bibliothek die Kenntnis einiger spezifischer Java-Konzepte erforderlich. Diese werden im nächsten Abschnitt behandelt.

2.2 Java Grundlagen

Für das Verständnis dieser Arbeit werden grundlegende Java-Kenntnisse vorausgesetzt. Zusätzlich zu diesen finden einige spezielle Klassen Anwendung, die dieser Abschnitt auflistet und einführt.
Für eine ausführlichere Beschreibung wird auf die Java API Dokumentation von Oracle verwiesen.

- **WeakReferences**[1] sind bestimmte Referenzen mit der Besonderheit, dass sie nicht vom Garbage Collector beachtet werden. Dadurch kann ein Objekt gelöscht werden, obwohl es von WeakReferences referenziert wird.
  Initialisiert man diese mit einer **ReferenceQueue**[2], so wird die WeakReference beim Löschen des referenzierten Objekts in die Queue gelegt. Dadurch können beispielsweise noch Aufräumaktionen auf der Referenz ausgeführt werden. Ein Zugriff auf das bereits gelöschte Objekt ist jedoch nicht mehr möglich.

- Für parallele Programme bietet Java mit dem **ExecutorService**[3]-Framework eine einfache Schnittstelle zur asynchronen Bearbeitung von Jobs. Umsetzungen des

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html
Abrufdatum: 04.05.2019
[2] https://docs.oracle.com/javase/7/docs/api/java/lang/ref/ReferenceQueue.html
Abrufdatum: 04.05.2019
[3] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html
Abrufdatum: 04.05.2019

Interfaces stellen einen Threadpool bereit sowie einen Mechanismus zur Aufgabenverteilung an die Threads. Eine Realisierung ist beispielsweise der **ForkJoinPool**[4], der Mittels eines Work-Stealing-Algorithmus die Arbeitslast gleichmäßig auf alle Worker aufteilt.
Als Rückgabewert jedes eingereichten Jobs erhält man ein **Future**[5]-Objekt, an das das Ergebnis nach Abschluss der Berechnung übergeben wird.

- Die Standardbibliothek bietet mehrere Implementierungen des Future-Interfaces. Mit dem einfachen **FutureTask**[6] können Berechnungen parallel ausgeführt werden. Durch den Befehl Future.*get()* wird das Ergebnis abgerufen, dafür blockiert der Thread solange bis die Aufgabe abgeschlossen ist.
  Die Klasse **CompletableFuture**[7] ist eine Future-Spezialisierung die dem Programmierer zusätzliche Funktionen zur Verkettung mehrerer CompletableFutures bietet. Beispielsweise können zwei asynchron ausgeführte Tasks mit einem dritten verknüpft werden, der die Ergebnisse zusammenführt. Womit oft ein blockierendes Warten voneinander abhängiger Jobs vermieden werden kann.
  Seit Java 7 bietet das **ForkJoin**[8]-Framework mit den **ForkJoinTasks**[9] weitere Future-Implementierungen. Mit task.*fork()* wird ein ForkJoinTask an den ForkJoinPool übergeben und mit task.*join()* können Ergebnisse abgerufen werden. Wichtig ist, dass task.*join()* lediglich den aktuellen ForkJoinTask blockiert und der Worker in der Zwischenzeit andere Jobs bearbeiten kann.
  Das ForkJoin-Framework realisiert eine Parallelisierungsstrategie gemäß dem Divide and Conquer-Prinzip. Rechenintensive Probleme werden zunächst in kleinere unabhängige Teilprobleme gespalten, bevor diese parallel berechnet und die Ergebnisse wieder zusammengeführt werden. Für die Aufgabenverteilung an Workerthreads wird der bereits vorgestellte ForkJoinPool verwendet.

- Die **Google-Guava**[10]-Bibliothek bietet eine Vielzahl mächtiger Collections und anderer interessanter Konzepte, wie zum Beispiel den ListenableFutures (ähnlich CompletableFutures), Caches (ConcurrentMap-Erweiterung) oder der BiMap (Map mit effizienten Value-Key-Zugriffen)

Damit wurden die wichtigsten Grundlagen präsentiert, um eine parallele BDD-Bibliothek mit Hilfe neuester Java-Konzepte umzusetzen. Das nächste Kapitel dokumentiert den Entwicklungsprozess von Parallel-Java-BDD (PJBDD) ausführlich.

[4] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html Abrufdatum: 04.05.2019
[5] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html Abrufdatum: 04.05.2019
[6] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html Abrufdatum: 04.05.2019
[7] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html Abrufdatum: 04.05.2019
[8] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html Abrufdatum: 04.05.2019
[9] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html Abrufdatum 04.05.2019
[10] https://github.com/google/guava/wiki Abrufdatum: 04.05.2019

3 Umsetzung

Nachdem alle notwendigen theoretischen und Java-spezifischen Konzepte erläutert wurden, wird darauf basierend die Implementierung der parallelisierten BDD-Bibliothek PJBDD vorgestellt. PJBDD setzt sich aus drei fundamentalen Bestandteilen zusammen: den BDD-Objekten, einer Knotentabelle und einer Creator-Umgebung. Ein BDD-Objekt repräsentiert einen bestimmten Teilbaum von der spezifischen Wurzel bis zu den Blättern. Die Knotentabelle oder auch Uniquetable dient dazu, bereits erzeugte Graphen zu speichern, um die Kanonizität zu wahren. Sämtliche boolesche Operationen auf einen oder mehreren BDD-Knoten werden über die jeweilige Creator-Umgebung aufgerufen. Im Weiteren wird die Implementierung dieser Komponenten, beginnend mit dem BDD, detailliert beschrieben.

3.1 BDD-Repräsentation

Zur Repräsentation der BDD-Bäume wurde das BDD-Interface eingeführt. Das definierte Interface fordert lediglich Getter für Variable und Kindknoten, boolesche Methoden zur Überprüfung auf Terminalknoten sowie das Überschreiben von equals und hashcode. Jede Implementierung benötigt zusätzlich eine Umsetzung des Factory-Interfaces. Dadurch kann PJBDD mit Dependecy Injection dynamische BDD-Typen instantiieren.
Die Standardimplementierung hält eine int-Variable und je eine Referenz auf die Kindknoten. Die beiden strukturellen Reduktionsregeln (siehe Abschnitt 2.1) werden mit Hilfe der Uniquetable (3.2) und dem makeNode-Algorithmus (3.3.1) außerhalb dieser Komponente sichergestellt.

3.2 Uniquetable-Implementierung

In der Uniquetable werden alle bereits erzeugten und verwendeten BDD-Knoten gespeichert. Vor der Erstellung neuer Knoten wird erst überprüft, ob identische Graphen existieren und wiederverwendet werden können. Außerdem muss der Speicherbedarf so gering wie möglich gehalten werden. Das bedeutet, dass nicht mehr verwendete Einträge zu löschen sind. PJBDD implementiert parallele Algorithmen, die gegebenenfalls zeitgleich Knoten erstellen bzw. suchen. Die Tabelle muss also threadsicher sein.
Daraus ergeben sich folgende Anforderungen an das Konzept:

1. Die Suche bereits existenter Knoten anhand Variable und Kindknoten muss möglichst performant sein. (3.2.1)
2. Ein Eintrag muss gelöscht werden, sobald ein Knoten nicht mehr referenziert wird. (3.2.2)
3. Die Threadsicherheit paralleler Zugriffe muss gewährleistet werden. (3.2.3)

In den kommenden Unterabschnitten wird beginnend mit dem ersten Punkt beschrieben, wie diese Kriterien erfüllt wurden.

3.2.1 Performantes Suchen in der Knotentabelle

Wichtig für die Wahl der zugrunde liegenden Speicherstruktur ist, dass Knoten möglichst performant anhand ihrer Variable und Kindknoten gefunden werden können. Da die Uniquetable in vielen Fällen sehr groß wird und eine Vielzahl an Suchanfragen empfängt, ist eine traversierende Suche mit O(n) unbedingt zu vermeiden. Es müssen also möglichst direkte Zugriffe ermöglicht werden, ohne den tatsächlichen Index zu kennen, weshalb einfache Arrays, Lists, Sets und ähnliche Strukturen nicht verwendet werden können.
Da jede Variablen- und Kindknoten-Kombination einzigartig ist, kann ein Hashwert-Index aus den drei bekannten Parametern gebildet werden. Es existiert jedoch keine injektive Funktion[1] mit drei Eingabewerten $f(i, j, k) \rightarrow x$, deshalb müssen mögliche Kollisionen[2] behandelt werden. Zur Umgehung dieser Hash-Kollisionen wird das Hashbucket-Prinzip implementiert. Ein Hashbucket ist ein Speicher, der Elemente zu einem berechneten Hashwert ablegt. Alle Einträge mit identischem Wert werden im selben Bereich (Bucket) gespeichert.
Die Speicherstruktur erlaubt bei der Suche eines bestimmten Knotens einen direkten Zugriff auf den korrespondierenden Bucket, der anschließend traversiert werden muss. Die Laufzeitkomplexität der Elementsuche liegt also je nach Aussagekraft der Hashfunktion im Best-Case bei O(1) und nur noch im Worst-Case bei O(n). Denn je aussagekräftiger die Funktion ist, desto weniger Elemente liegen in den einzelnen Buckets und desto performanter ist dementsprechend die Suche.
Java bietet mit der HashMap eine native Datenstruktur, die dieses Prinzip implementiert. Werden eindeutige Key-Objekte gewählt, behandelt sie auftretende Hashkollisionen intern. Da jedoch kein eindeutiger int-Key gebildet werden kann, wurden drei alternative Lösungsansätze umgesetzt.

1. **HashDeque**
   Diese Umsetzung basiert auf einer HashMap mit Integer-Keys und Deque-Values. Die Deques dienen als einfache Bucket-Implementierung. Das bedeutet, dass die BDD-Objekte in die Deque gespeichert werden, die zum jeweiligen Hashwert in der Map abgelegt wurden. Kollisionen werden also behandelt, indem alle Knoten mit identischem Hash in der selben Deque gehalten werden und die Objektsuche kann wie folgt realisiert werden:

```
1  BDD lookUp(int var, BDD low, BDD high){
2    int hash = hashNode(var, low, high)
3    Deque bucket = hashDeque.get(hash)
4    for(BDD bdd: bucket){
5      if(matches(bdd, var, low, high)){
6        return bdd
7      }
8    }
9    return null
10 }
```

Listing 3.1: Objektsuche der HashDeque-Implementierung

[1] Eine Funktion ist injektiv, wenn für jede Eingabe eine einzigartige Lösung existiert.
[2] Eine Kollision liegt dann vor, wenn zwei unterschiedliche Eingaben im selben Ergebnis resultieren.

2. **Array-Kombination**
   Die zweite Implementierung basiert auf drei Arrays, deren Größen bei Bedarf verdoppelt werden. Im Ersten werden alle erzeugten BDD-Objekte gespeichert. Das Zweite legt die Indizes der BDD-Objekte des ersten Arrays zum Hashwert der Knoten ab. Das dritte Array dient der Kollisionsbehandlung, wofür zum Index eines Objekts der Index des nächsten BDD mit identischem Hashwert gespeichert wird. Zusammengefasst dient das Erste also als Knoten-, das Zweite als Hash- und das Dritte als Bucket-Tabelle.
   Nachfolgender Programmcode-Auszug zeigt wie die Implementierung eine Suche nach gegebenen Parametern umsetzt.

```
1  BDD lookUp(int var, BDD low, BDD high){
2    int hash = hashNode(var, low, high) % tableSize
3    int index = hashTable[hash]
4    while(index > 1){
5      if(matches(nodeTable[i], var, low, high)){
6        return nodeTable[i]
7      }
8      index = bucketTable[index]
9    }
10   return null
11 }
```

Listing 3.2: Objektsuche der ArrayUniquetable-Implementierung

Zu beachten gilt, dass eine Veränderung der Größe die Neuberechnung aller Hasheinträge erfordert.

3. **HashMap**
   Ein weiterer Ansatz ist, die Kollisionsbehandlung der Java-HashMap zu überlassen. Dafür dienen die BDD-Knoten als Map-Key sowie als Map-Value. Für die Suche kann nun ein temporäres Objekt mit gewünschten Eigenschaften erzeugt und anschließend verwendet werden, um einen identischen Knoten zu finden.
   Falls kein Passender gefunden wird, kann der Temporäre verwendet werden. Dieser Ansatz bietet eine einfache Möglichkeit zur Umsetzung der Suche:

```
1  BDD lookUp(int var, BDD low, BDD high){
2    BDD tmp = factory.create(var, low, high)
3    hashMap.putIfAbsent(tmp, tmp)
4    return hashMap.get(tmp)
5  }
```

Listing 3.3: Objektsuche der HashMap-Implementierung

Um dem daraus resultierenden Overhead durch Initialisierung und direkter Garbage Collection entgegenzuwirken, werden die Hilfsknoten nicht einfach gelöscht, sondern für spätere Wiederverwendung gespeichert. Dementsprechend wird vor der Erzeugung eines neuen Knotens geprüft, ob ein alter wiederverwendet werden kann.

Bedingung für diesen Ansatz ist, dass die equals-Methode der BDD-Implementierung die Objekte anhand der Kindknoten und Variablen vergleicht. Darüber hinaus soll-

te die hashcode-Methode möglichst aussagekräftig überschrieben werden, da sie HashMap-intern zu Hashkollisionen führen kann, die wiederum maßgeblich für die Performanz sind.

In diesem Abschnitt wurden unterschiedliche Hashbucket-Implementierungen vorgestellt und ausgeführt wie diese eine performante Suche anhand gegebener Parameter ermöglichen. In den kommenden beiden Unterpunkten wird die Implementierung der weiteren Anforderungen aus 3.2 fokussiert.

3.2.2 Automatisches Entfernen nicht mehr referenzierter Objekte

Die WeakReference aus der Java-Standardbibliothek bietet eine hervorragende und einfache Lösung für die Anforderung, dass nicht mehr referenzierte Objekte aus der Tabelle entfernt werden müssen. Denn speichert man lediglich WeakReferences auf die erzeugten BDD-Objekte, so werden diese gelöscht, sobald keine normalen Objektreferenzen mehr vorhanden sind. Dafür ist jedoch die Implementierung einer rekursiven BDD-Datenstruktur vonnöten. Das bedeutet jeder Knoten muss echte Referenzen auf seine Nachfolger halten. Ansonsten könnten Knoten gelöscht werden, die als Weg-Knoten dienen und Framework-extern nicht referenziert werden.
Außerdem erfordern die alternativen Implementierungen aus 3.2.1 unterschiedliche Anpassungen an der WeakReference-Klasse.

1. **HashDeque**
   Bei der HashDeque-Lösung kann eine herkömmliche WeakReference, deren referenziertes Objekt gelöscht wurde, keinem Hashwert mehr zugewiesen werden. Also müsste zum Entfernen einer leeren Referenz die gesamte Tabelle inklusive aller Deques traversiert werden, um die entsprechende Referenz zu löschen. Deshalb wurde eine WeakReference-Subklasse eingeführt, deren Objekte den int-Hashwert des Referenten halten, dadurch ist auch für gelöschte Objekte ein direkter Zugriff auf die richtige Deque möglich.

2. **Array-Kombination**
   Da Java Arrays keine gebundenen Objekt-Parameter haben können, ist für dieses Konzept ebenfalls eine modifizierte WeakReference notwendig, die den generischen Typen direkt an BDDs bindet.

3. **HashMap**
   Für den Ansatz ist es erforderlich, dass ein BDD anhand eines anderen mit identischen Parametern gefunden wird. Dafür wurde eine WeakReference-Subklasse mit überschriebenen equals- und hashcode-Methoden erstellt. Objekte dieser Klasse returnieren den Hashwert des referenzierten Objekts und prüfen zusätzlich die Objektgleichheit der Referenten. Dadurch werden WeakReferences neben der Instanz-Gleichheit anhand ihrer Referenten verglichen.

Mit diesen Anpassungen kann für alle Lösungsansätze ein automatisches Entfernen nicht mehr referenzierter Objekte garantiert werden. Um leere Referenzen aus der Tabelle zu

löschen, wird bei der Initialisierung eine ReferenceQueue (siehe Abschnitt 2.2) mitgeben. Darüber hinaus hält jede Implementierung einen Daemon-Thread, der darauf wartet, dass Objekte in diese Queue gelegt werden, um sie anschließend aus der Tabelle zu entfernen. Als nächstes wird dargestellt, wie die Threadsicherheit realisiert wurde.

3.2.3 Threadsicherheit paralleler Zugriffe

Da Algorithmen implementiert wurden, die nebenläufig Lese- und Schreibe-Operationen an den Tabellen durchführen, muss diese threadsicher sein. Der einfachste Ansatz wäre, die Tabelle zu synchronisieren, sprich lediglich sequentielle Anfragen zuzulassen. Aus Performanzgründen wurden jedoch drei alternative Konzepte umgesetzt, die asynchrone Zugriffe möglichst nebenläufig erlauben.

- **Konzept 1 - Nebenläufige Speicherstrukturen**
  Die Java Standardbibliothek stellt einige nebenläufige Speicherstrukturen zur Verfügung. Unter geeigneter Anwendung dient die ConcurrentHashMap beispielsweise als threadsichere Basis für die HashMap Lösung.
  Das HashDeque-Konzept kann ebenfalls mit einer ConcurrentHashMap und zusätzlichen ConcurrentLinkedDeque-Objekten sichere parallele Operationen ermöglichen.

- **Konzept 2 - Read-Write-Monitoring**
  Mit einem sogenannten ReadWriteLock ist eine Realisierung möglich, die beliebig viele parallele Leseoperationen zulässt, jedoch nur eine Schreibaktion. Zudem kann nicht gelesen werden, wenn geschrieben wird, und vice versa.

- **Konzept 3 - Monitoring einzelner Hashsegmente**
  Das dritte Konzept erlaubt beliebig viele gleichzeitige Anfragen auf unterschiedliche Gebiete, jedoch keine auf dasselbe.
  Dafür werden beliebig große Teilbereiche der Tabelle definiert, die je mit einem Monitor synchronisiert werden. Dabei gilt es einen guten Mittelweg für die Größe dieser Segmente zu finden. Denn je größer sie sind, desto größer ist der blockierte Teil der Tabelle wenn ein Monitor gesperrt wird und je kleiner, desto größer ist der Verwaltungsaufwand, sprich desto mehr Monitor-Objekte werden benötigt.

Tabelle 3.1 zeigt eine Auflistung der unterschiedlichen Kombinationen der vorgestellten Sperrkonzepte mit den erarbeiteten Speicherstrukturen.

Nr	**Klassenname**	**Speicherstruktur**	**Sperrkonzept**
1	BDDConcurrentWeakHashDeque	HashDeque	Konzept 1
3	BDDConcurrentWeakHashMap	HashMap	Konzept 1
4	BDDLockOnWriteArray	Array-Kombination	Konzept 2
5	BDDResizingArray	Array-Kombination	Konzept 3

Tabelle 3.1: Implementierte Uniquetable-Kombinationen

3.3 Creator-Umgebung

Wie bereits zu Beginn dieses Kapitels erwähnt, dient die Creator-Komponente dazu boolesche Operationen auf BDDs auszuführen. Dieser Abschnitt dokumentiert also die Umsetzung der Algorithmen sowie deren Parallelisierung. Alle booleschen Methoden wurden je nach Implementierung entweder mit dem ITE- oder dem Apply-Operator realisiert. Eine Auflistung der unterstützen Funktionen ist der Tabelle 2.1 in Kapitel 2.1.1 zu entnehmen. Außerdem setzt diese Umgebung die makeNode-Methode um, die die beiden BDD-Reduktionsregeln mit Hilfe der Uniquetable sicherstellt (siehe 2.1). Die in den kommenden Unterabschnitten vorgestellten Programmcodeauszüge haben keinen Anspruch auf Vollständigkeit, viel mehr liegt der Fokus darauf, das jeweilige Kernkonzept zu verdeutlichen.

3.3.1 Knotenerzeugung

Die Methode *makeNode* wird zur Erzeugung neuer Knoten aufgerufen. Sie überprüft zunächst, ob beide Kindknoten identisch sind, wodurch der Knoten reduziert werden kann, oder ob bereits ein identischer BDD existiert. Erst wenn keines der beiden Kriterien erfüllt ist, erfolgt die Instantiierung eines neuen Knotens. Der Algorithmus wurde wie folgt entwickelt:

```
1 BDD makeNode(BDD low, BDD high, int var){
2   if(low == high){ return low }
3   BDD lookUp = uniqueTable.lookUp(low, high, var)
4   if(lookUp){ return lookUp }
5   return create(low, high, var)
6 }
```

Listing 3.4: Knotenerzeugung der BDD-Umsetzung

Die Methode *create* bezeichnet die Erzeugung eines neuen Knotens, dementsprechend hängt die Performanz hauptsächlich von der Uniquetable Implementierung ab (3.2.1).

3.3.2 Parallelisierung der booleschen Operationen

Wie bereits erwähnt wurden mit dem ITE- und dem Apply-Algorithmus zwei generalisierte Methoden zur Abbildung aller booleschen Operationen realisiert. Die grundsätzliche Idee der Bibliothek ist, die Shannon-Dekomposition dieser Beiden zu nutzen, um mittels Divide and Conquer eine nebenläufige Berechnung durchzuführen. Abbildung 3.1 verdeutlicht das Vorgehen anhand der Funktion $OP(F, G)$.

Für die rekursiven Teilschritte wird je ein neuer Task erzeugt und asynchron berechnet. Anschließend werden die beiden Ergebnisse vom Vatertask zusammengeführt. Task4 und Task5 bilden beispielsweise zwei Rekursionsschritte, die unabhängig von einander ausgeführt und im Anschluss von Task2 fusioniert werden.

Um das bestmögliche Java-Konzept für diese Strategie zu finden, wurden alternative Ansätze implementiert, die im Anschluss anhand des ITE vorgestellt werden.

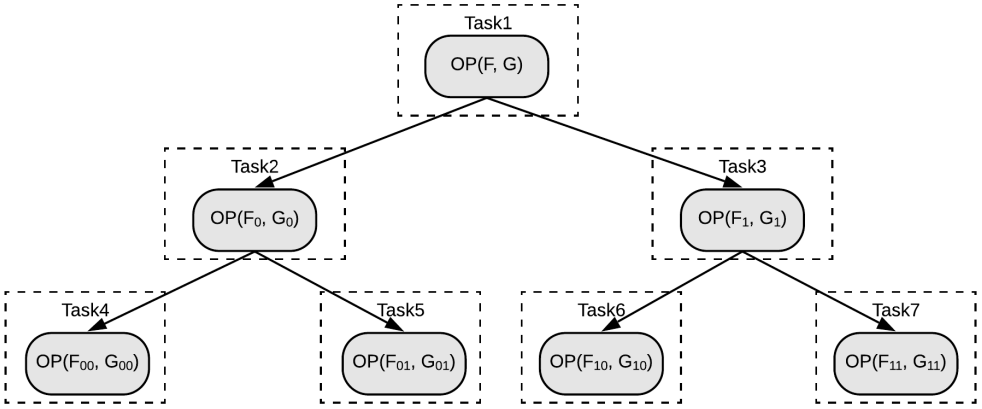


Abbildung 3.1: Aufteilung der Operation $OP(F, G)$ in parallele Tasks (Quelle: eigene Darstellung)

3.3.3 If-Then-Else

Die theoretischen Grundlagen für den ITE-Operator wurden in Abschnitt 2.1.1 geliefert. Dieses Unterkapitel beschreibt nun die tatsächliche Umsetzung. Nachfolgender Auszug zeigt die ITE-Basisimplementierung:

```
1  BDD makeITE(BDD f1, BDD f2, BDD f3){
2    BDD res = terminalCheck(f1, f2, f3)
3    if(res){ return res }
4    BDD res = shannonExpansionIte(f1, f2, f3)
5    cache(f1, f2, f3, res)
6    return res
7  }
8
9  BDD terminalCheck(BDD f1, BDD f2, BDD f3){
10   if(f1 == TRUE || f2 == f3){ return f2 }
11   if(f1 == FALSE){ return f3 }
12   if(f2 == TRUE && f3 == FALSE){ return f1 }
13   BDD res = cacheCheck(f1, f2, f3)
14   if(res){ return res }
15   return null
16 }
17
18 BDD cacheCheck(BDD f1, BDD f2, BDD f3){
19   CacheData data = computedTable.get(f1, f2, f3)
20   if(data){ return data.result }
21   return null
22 }
```

Listing 3.5: Basis-Realisierung der ITE-Methode

Die Methode *terminalCheck* beschreibt die Überprüfung auf eine ITE-Abbruchbedingung und *cacheCheck* implementiert die Optimierung mit Hilfe eines Computation-Caches (sie-

he Grundlagenkapitel 2.1). *shannonExpansionIte* definiert die abstrakte Methode, die die parallelisierte Shannon-Zerteilung realisiert.
Folgender Auszug definiert einige Variablen, die in mehreren Unterabschnitten wiederverwendet werden:

```
1  int topVar = topVar(f1, f2, f3)
2  BDD highF1 = restrict(f1, topVar, true)
3  BDD highF2 = restrict(f2, topVar, true)
4  BDD highF3 = restrict(f3, topVar, true)
5  BDD lowF1 = restrict(f1, topVar, false)
6  BDD lowF2 = restrict(f2, topVar, false)
7  BDD lowF3 = restrict(f3, topVar, false)
8
9  BDD restrict(BDD f, int var, boolean isHigh){
10   return (var(f) != var ? f : (isHigh ? high(f) : low(f)))
11 }
```

Listing 3.6: Definition häufig verwendeter Variablen

Die *restrict*-Methode beschränkt einen bestimmten Knoten auf seinen high- oder low-Nachfolger, wenn seine Variable einer Bestimmten entspricht.
Die alternativen Java-Implementierungen der asynchronen Dekomposition werden in den folgenden Unterpunkten präsentiert.

3.3.3.1 Stream-Parallelisierung

Die erste Realisierung basiert auf den in Java 8 eingeführten parallelen Stream. Dafür werden die beiden Rekursionszweige der ITE-Berechnung auf einen parallelisierten booleschen Stream abgebildet. Im Anschluss werden die Ergebnisse mit Hilfe der nativen Collectors.*toMap*-Funktion in einer Map gespeichert, aus deren Einträge dann das Gesamtergebnis gebildet wird:

```
1  BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2    Map subs = Stream.of(false, true).parallel()
3      .collect(Collectors.toMap(b -> b, b -> {
4        if(b){
5          return makeITE(highF1, highF2, highF3)
6        }else{
7          return makeITE(lowF1, lowF2, lowF3)
8        }
9    }))
10   return makeNode(subs.get(false), subs.get(true), topVar)
11 }
```

Listing 3.7: Stream-Parallelisierung der ITE-Shannon-Dekomposition

Alternative Java-Konzepte zur Parallelisierung bietet das Future-Framework, mit Hilfe dessen weitere Versionen des Algorithmus entwickelt wurden.

3.3.3.2 Future-Parallelisierung

Der Grundgedanke dieser Version ist, für das Ergebnis jedes Rekursionsasts ein eigenes Future-Objekt anzulegen, um anschließend die Ergebnisse der jeweilig Zusammengehörenden zu kombinieren:

```
1 BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2   Future lowFut = threadPool.submit(
3     () -> makeITE(lowF1, lowF2, lowF3))
4   Future highFut = threadPool.submit(
5     () -> makeITE(highF1, highF2, highF3))
6   return makeNode(lowFut.join(), highFut.join(), topVar)
7 }
```

Listing 3.8: Future-Parallelisierung der ITE-Shannon-Dekomposition

Der Nachteil dieser Lösung ist die blockierende Natur des *join()*-Aufrufs, die bewirkt, dass bei unausgeglichener Arbeitslast der beiden Rekursionszweige ein Thread blockierend wartet. Eine Möglichkeit dem Abhilfe zu schaffen ist, die parallelen Rechenzweige asynchron zu verketten. Da das Future-Framework diese Funktionalität nicht unterstützt, wurden dafür CompletableFutures aus der Standardbibliothek und ListenableFutures aus der Google-Guava-Bibliothek eingesetzt.

3.3.3.3 CompletableFuture-Parallelisierung

Ziel der Implementierung mit CompletableFutures ist, die parallelisierten Rekursionsschritte so zu verketten, dass lediglich der Thread wartet, der die Operation ausführt.

```
1  BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2     return asyncExpand(f1, f2, f3).join()
3  }
4
5  CompletableFuture asyncExpand(BDD f1, BDD f2, BDD f3){
6    BDD res = terminalIteCheck(f1, f2, f3)
7    if(res){ return res }
8    CompletableFuture lowFut = expand(lowF1, lowF2, lowF3)
9    CompletableFuture highFut = expand(highF1, highF2, highF3)
10   return highFut.thenCombine(
11     lowFut, (h, l) -> {
12      BDD res = makeNode(l, h, topVar)
13      cache(f1, f2, f3, res)
14      return res
15     })
16 }
17
18 CompletableFuture expand(BDD f1, BDD f2, BDD f3){
19   return CompletableFuture.supplyAsync(() ->
20     asyncExpand(f1, f2, f3)).thenCompose(identity())
21 }
```

Listing 3.9: CompletableFuture-Parallelisierung der ITE-Shannon-Dekomposition

thenCombine() führt eine Operation auf den Ergebnissen zweier CompletableFuture-Objekte aus. Ein weiterer wichtiger Bestandteil dieser Umsetzung ist der Aufruf *thenCompose(identity())*. Dieser bildet lediglich das berechnete CompletableFuture-Objekt auf sich selbst ab und sorgt somit dafür, dass sich der Rückgabetyp der Rekursion nicht weiter verschachtelt.
Diesem Ansatz sehr ähnlich ist das auf ListenableFutures aus der Guava-Bibliothek basierende Konzept.

3.3.3.4 ListenableFuture-Parallelisierung

Analog zur vorherigen Strategie wird die Rückgabe der parallelen Rechenzweige asynchron mit ListenableFutures aus der Guava-Bibliothek kombiniert:

```
1  BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2    return asyncExpand(f1, f2, f3).join()
3  }
4
5  ListenableFuture asyncExpand(BDD f1, BDD f2, BDD f3){
6    BDD res = terminalIteCheck(f1, f2, f3)
7    if(res){
8      return res
9    }
10   ListenableFuture lowFut = Futures.transformAsync(
11     () -> asyncExpand(lowF1, lowF2, lowF3)
12   ListenableFuture highFut = Futures.transformAsync(
13     () -> asyncExpand(highF1, highF2, highF3)
14
15   return Futures.whenAllSucceed(lowFut, highFut)
16    .call(() -> {
17     BDD res = makeNode(lowFut.get(), highFut.get(), topVar)
18     cache(f1, f2, f3, res)
19     return res
20    })
21 }
```

Listing 3.10: ListenableFuture-Parallelisierung der ITE-Shannon-Dekomposition

Mit Futures.*whenAllSucceed()* kann asynchron auf eine beliebige Anzahl an parallelen Rechnungen gewartet werden. Die Methode *call()* führt den mitgegebenen Befehl aus, sobald das Ergebnis des ListenableFutures bekannt ist. Dadurch kann future.*get()* asynchron verwendet werden, da es erst aufgerufen wird, wenn beide Ergebnisse berechnet wurden.
Ein möglicher Schwachpunkt dieser und der vorherigen Implementierungen ist allerdings der Overhead durch die asynchrone Verkettung der Arbeitsschritte. Denn mit jedem Rekursionsschritt werden drei Future-Objekte erzeugt, egal ob tatsächlich parallelisiert wird oder nicht. Selbst ein Terminalergebnis muss in ein Solches gepackt werden.
Eine Reduzierung dieses Overheads konnte mit dem in Java 7 eingeführten ForkJoin-Framework erreicht werden.

3.3.3.5 ForkJoin-Parallelisierung

Mit dem ForkJoin-Framework werden die beiden Rekursionsaufrufe in parallele ForkJoin-Tasks aufgespalten und die Ergebnisse anschließend fusioniert:

```
1 BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2   ForkJoinTask lowTask = forkJoinPool.submit(
3     () -> makeITE(lowF1, lowF2, lowF3))
4   ForkJoinTask highTask = forkJoinPool.submit(
5     () -> makeITE(highF1, highF2, highF3))
6   return makeNode(lowTask.join(), highTask.join(), topVar)
7 }
```

Listing 3.11: ForkJoin-Parallelisierung der ITE-Shannon-Dekomposition

Im ersten Moment scheint diese Implementierung äquivalent zur Future-Umsetzung aus Abschnitt 3.3.3.2. Tatsächlich hat sie aber einen entscheidenden Vorteil, denn der *join*-Aufruf blockiert lediglich den aktuellen Task, der ausführende Thread kann in der Zwischenzeit andere Tasks bearbeiten. Mit dem ForkJoin-Framework können also ebenfalls parallele Tasks asynchron verkettet werden.

3.3.3.6 Parallelisierungskonzepte im Überblick

In den vorherigen Unterabschnitten wurde die Parallelisierung des ITE-Algorithmus mit alternativen Java-Konzepten vorgestellt. Zur Bewertung der Strategien wurde zusätzlich noch eine serielle Version implementiert:

```
1 BDD shannonExpansionIte(BDD f1, BDD f2, BDD f3){
2   int topVar = topVar(f1, f2, f3)
3   BDD low = makeITE(lowF1, lowF2, lowF3)
4   BDD high = makeITE(highF1, highF2, highF3)
5   return makeNode(low, high, var)
6 }
```

Listing 3.12: Serielle Umsetzung der ITE-Shannon-Dekomposition

Tabelle 3.2 zeigt einen zusammenfassenden Überblick der Klassen und das jeweils zur Parallelisierung verwendete Java-Konzept.

Nr	**Klassenname**	**Parallelisierungskonzept**
1	SerialITECreator	keine Parallelisierung
2	StreamITECreator	Stream-Parallelisierung
3	FutureITECreator	Future-Parallelisierung
4	CompletableFutureITECreator	CompletableFuture-Parallelisierung
5	GuavaFutureITECreator	ListenableFuture-Parallelisierung
6	ForkJoinITECreator	ForkJoin-Parallelisierung

Tabelle 3.2: Implementierte Parallelisierungskonzepte

In Kapitel 5 folgt eine ausführliche Analyse der implementierten Konzepte und wie sie

sich in Kombination mit den unterschiedlichen Uniquetable-Typen verhalten.

Zusätzlich zum ITE gibt es mit dem Apply-Algorithmus eine weitere generalisierte Strategie zur Umsetzung der booleschen Methoden, dessen Realisierung im nächsten Punkt skizziert wird.

3.3.4 Apply-Algorithmus

Der Apply-Algorithmus wurde aufgrund der besseren Laufzeitkomplexität ($O(|F| * |G|)$) im Vergleich zum ITE ($O(|F| * |G| * |H|)$) entwickelt.
Die Shannon-Zerlegung kann nahezu analog zu der des ITE umgesetzt werden. Folgender Auszug zeigt das Vorgehen beispielhaft am OR-Operator und der ForkJoin-Parallelisierung:

```
1  BDD makeOr(BDD f1, BDD f2){
2    return apply(f1, f2, OP_OR)
3  }
4
5  BDD apply(BDD f1, BDD f2, ApplyOp op){
6    BDD res = terminalCheck(f1, f2, op)
7    if(res){ return res }
8    ForkJoinTask lowTask = forkJoinPool.submit(
9      () -> apply(lowF1, lowF2, op))
10   ForkJoinTask highTask = forkJoinPool.submit(
11     () -> apply(highF1, highF2, op))
12   res = makeNode(lowTask.join(), highTask.join(), topVar)
13   cache(f1, f2, op, res)
14   return res
15 }
16
17 BDD terminalCheck(BDD f1, BDD f2, ApplyOp op){
18   switch(op){
19     case OP_OR: return orCheck(f1, f2)
20     case OP_AND: return andCheck(f1, f2)
21     ...
22   }
23 }
24
25 BDD orCheck(BDD f1, BDD f2){
26   if(f1 == f2){ return f1 }
27   if(f1 == TRUE || f2 == TRUE){ return TRUE }
28   if(f1 == FALSE || f2 == FALSE){ return FALSE }
29   BDD res = cacheCheck(f1, f2, OP_OR)
30   if(res){ return res }
31   return null
32 }
```

Listing 3.13: ForkJoin-Parallelisierung der Apply-Shannon-Dekomposition

Die Implementierungen der beiden Algorithmen unterscheiden sich hauptsächlich im terminalCheck. Der ITE realisiert eine allgemein gültige Überprüfung, wohingegen der Apply für jede boolesche Operation einen eigene Überprüfungsfall definieren muss. Der Apply-Algorithmus wurde ebenfalls mit unterschiedlichen Java-Konzepten analog zum ITE-Pendant parallelisiert. Umgesetzt wurde eine serielle Version, eine mit Hilfe des ForkJoin-Frameworks und eine auf CompletableFutures basierende Version.
Im nächsten Unterabschnitt wird mit dem Computation-Cache eine weitere wichtige Komponente für BDD-Frameworks entwickelt.

3.3.5 Computation-Cache

Im Grundlagenkapitel wurde der Computation-Cache als mögliche Optimierung vorgestellt. Mit diesem können bereits ausgeführte Operationen zwischengespeichert und Ergebnisse wiederverwendet werden. Zur Speicherung wurde eine CacheData Klasse mit drei Eingabewerten und dem Ergebniswert der Berechnung erstellt. Objekte dieser Klasse werden in der sogenannten computedTable abgespeichert. Um mit Hilfe des Input-Tripels eine effiziente Suche nach einer bestimmten Funktion zu ermöglichen, werden die Einträge zum Hash dieses Tripels abgelegt. Da diese Tabelle ohnehin eine limitierte Größe hat und Einträge nicht dauerhaft gespeichert werden sollen, wird keine Kollisionsbehandlung durchgeführt. Das bedeutet, Elemente mit identischen Hashwerten werden einfach überschrieben. Ein dauerhaftes Sichern würde verhindern, dass nicht verwendete BDDs gelöscht werden, weil der Cache echte Referenzen auf die Objekte hat.
Umgesetzt wurde das Caching wie folgt:

```
1  void cache(BDD f1, BDD f2, BDD f3, BDD result){
2    CacheData data = new CacheData()
3    data.f1 = f1
4    data.f2 = f2
5    data.f3 = f3
6    data.result = result
7    int hash = hash(f1, f2 , f3)
8    computedTable.put(hash, data)
9  }
10
11 BDD checkCache(BDD f1, BDD f2, BDD f3){
12   int hash = hash(f1, f2, f3)
13   CacheData data = computedTable.get(hash)
14   if(data){
15     if(d.f1 == f1 && d.f2 == f2 && d.f3 == f3){
16       return data.res
17     }
18   }
19   return null
20 }
```

Listing 3.14: Realisierung des Cachings von Operationsergebnissen

Der Apply-Cache wurde äquivalent realisiert.
PJBDD implementiert zwei Cache-Versionen mit unterschiedlichen zugrunde liegenden

Speicherstrukturen. Die Erste, der GuavaCache, basiert auf einem LoadingCache aus der Guava-Bibliothek mit fixer Größe und zeitlich begrenzter Verfügbarkeit der Einträge. Der LoadingCache gewährleistet bereits Threadsicherheit paralleler Zugriffe, zudem kann ein Parallelisierungsfaktor bei der Instantiierung mitgeben werden.
Die Zweite, der ArrayCache, nutzt ein normales Array mit fester Größe. Um parallele Zugriffe zu gewähren, wird das in Abschnitt 3.2.3 entwickelte Konzept mit synchronisierten Hashbereichen wiederverwendet. Ein praktischer Vergleich der beiden Alternativen folgt in Kapitel 5.

3.3.6 Reduzierung der asynchronen Tasks

Um zu verhindern, dass für jeden rekursiven Teilschritt ein neuer Task angelegt wird, verfolgen sämtliche Variationen zwei Strategien, um die Erzeugung überflüssiger Jobs zu reduzieren. Die Erste verhindert, dass ein neuer Task für einen Terminalfall oder Cachehit erzeugt wird, indem entsprechende Prüfungen für beide Rekursionszweige bereits vor der tatsächlichen Rekursion durchgeführt werden. Die Zweite beschränkt den Overhead dadurch, dass zunächst geprüft wird, ob genügend Kapazitäten, also Workerthreads zur Verfügung stehen. Dafür wird die in 3.3.3 entwickelte Methode makeITE wie folgt erweitert:

```
1  BDD makeITE(BDD f1, BDD f2, BDD f3){
2    BDD res = terminalCheck(f1, f2, f3)
3    if(res){ return res }
4    BDD low = terminalCheck(lowF1, lowF2, lowF3)
5    BDD high = terminalCheck(highF1, highF2, highF3)
6    if(canFork(topVar) && high == null && low == null){
7      res = shannonExpansionIte(f1, f2, f3)
8    }else{
9      if(low == null){
10       low = makeITE(lowF1, lowF2, lowF3)
11     }
12     if(high == null){
13       high = makeITE(highF1, highF2, highF3)
14     }
15     res = makeNode(low, high, topVar)
16   }
17   cache(f1, f2, f3, res)
18   return res
19 }
```

Listing 3.15: Implementierte Maßnahmen zur Taskreduzierung

Für die Apply Umsetzung wurde ein äquivalentes Konzept eingeführt.

Mit der Änderung der Variablensortierung wird eine weitere BDD-Optimierung unterstützt.

3.3.7 Änderung der Variablensortierung

Wie bereits in Kapitel 2.1 ausgeführt, hat die Variablenordnung gravierende Auswirkung auf die BDD-Darstellung. Bei einer Änderung müssen alle bereits existenten Knoten mit betroffenen Variablen angepasst werden. Hierfür wurde ein einfacher Swap-Algorithmus implementiert, der solange benachbarte Variablen tauscht, bis die existierende mit der neuen Ordnung übereinstimmt:

```
1  BDD setVarOrder(int... newOrder){
2    for(int level = 0; level < newOrder.length; level++){
3      int levelOfVar = level(newOrder[level])
4      while(levelOfVar != level){
5        if(levelOfVar < level){
6          swapLevel(evelOfVar + 1, levelOfVar++)
7        }else{
8          swapLevel(levelOfVar, --levelOfVar)
9  } } } }
10
11 void swapLevel(int levelA, int levelB){
12   int varA = getVar(levelA)
13   int varB = getVar(levelB)
14   setLevel(levelA, varB)
15   setLevel(levelB, varA)
16   swap(varA, varB)
17 }
18
19 void swap(int varA, int varB){
20   for(BDD bdd: uniqueTable){
21     if(var(bdd) == varA){
22       BDD low = l0 = l1 = low(bdd)
23       BDD high = h0 = h1 = high(bdd)
24       if(var(low) == varB || var(high) == varB){
25         if(var(low) == varB){
26           l0 = low(low)
27           l1 = high(low)
28         }
29         if(var(high) == varB){
30           h0 = low(high)
31           h1 = high(high)
32         }
33         setLow(bdd, makeNode(l0, h0, varA))
34         setHigh(bdd, makeNode(l1, h1, varA))
35         setVar(bdd, varB)
36 } } } }
```

Listing 3.16: Einfache Swap-Implementierung zur Änderung der Variablensortierung

Da gleichzeitiges Ausführen einer Umsortierung mit anderen Operationen zu fehlerhaf-

ten Ergebnissen führen kann, wird ein ReadWriteLock zur Koordination verwendet. Eine Neuordnung benötigt den WriteLock, wohingegen alle anderen Interface-Methoden der Creator-Komponente mit einem Read-Zugriff arbeiten. Damit wird sichergestellt, dass beliebig viele andere Operationen gleichzeitig ausgeführt werden können, jedoch nur eine Variablensortierung.
Aktuell unterstützt PJBDD lediglich manuelle Änderungen an der Ordnung. Dynamische Sortierungsstrategien wurden noch nicht umgesetzt.

PJBDD bietet eine objektorientierte Realisierung des BDD-Frameworks mit parallelen Algorithmen. Zusätzlich zu dieser wird eine int-basierte Version integriert.

3.4 int-basierte Ausführung

Um den Overhead aufgrund von Instantiierungen und Garbage Collection zu reduzieren, wurde ein weiteres Konzept entwickelt, das in Anlehnung an andere Bibliotheken keine BDD-Objekte für interne Berechnungen verwendet.
Jeder Knoten wird lediglich von einem primitiven int-Index in der Uniquetable dargestellt, über den die Variable und die Indizes der Kindknoten abrufbar sind. Die Nutzung eines primitiven Datentyps zieht einige gravierende Anpassungen an die eben vorgestellten Konzepte nach sich. Zum Beispiel gibt es nur eine BDD-Wrapperklasse, die als Schnittstelle für externe Anwendungen dient und die lediglich den Index sowie eine Referenz auf die Creator-Umgebung hält. Außerdem können für primitive Datentypen weder Collections noch WeakReferences verwendet werden. Dementsprechend muss die Uniquetable-Umsetzung auf einem Array basieren und einen manuellen Referenzzähler implementieren. Anders als beim Konzept basierend auf einer Array-Kombination aus Abschnitt 3.2.1 genügt ein einziges int-Array, in welchem jedem Knoten eine bestimmte Anzahl an Einträgen zugewiesen wird. So lassen sich alle benötigten Parameter wie folgt abfragen:

```
1 high  = table[index * NODE_SIZE + HIGH]
2 low   = table[index * NODE_SIZE + LOW]
3 var   = table[index * NODE_SIZE + REF_VAR] & VAR_MASK
4 refc  = table[index * NODE_SIZE + REF_VAR] & REF_MASK
5 hash  = table[index * NODE_SIZE + HASH]
6 next  = table[index * NODE_SIZE + NEXT]
```

Listing 3.17: BDD-Parameterabfrage in der int-basierten Version

Um etwas Speicherplatz zu sparen, können sich der Referenzzähler und die Variable via Bitshifting einen Tabelleneintrag teilen, weil diese im Gegensatz zu den anderen Einträgen nicht den vollen int-Bereich nutzen.
Die int-basierte Ausführung enthält eine serielle, eine ForkJoin- sowie eine Completable-Future-basierte Version der Algorithmen. Für externe Anwendung können die IntCreator Klassen analog zu den objektorientierten eingesetzt werden, da dieselben Interfaces implementiert wurden. Für die BDD-Wrapperobjekte werden außerdem WeakReferences eingesetzt, um den zugehörigen Referenzzähler automatisch zu verwalten.
Eine ausführliche Analyse und ein Vergleich dieser Version mit der auf BDD-Objekten basierenden folgt ebenfalls in Kapitel 5.

3.5 ZDD-Umsetzung

Mit den ZDDs unterstützt PJBDD darüber hinaus eine spezielle Form von BDDs, die für bestimmte Anwendungsfälle besser geeignet sind. Trotz der strukturellen Unterschiede aufgrund divergierenden Reduktionsregeln, können die meisten BDD-Komponenten benutzt werden. Zum Beispiel muss lediglich die *makeNode* wie folgt angepasst werden, um sämtliche Uniquetable- und BDD-Implementierungen wiederzuverwenden.

```
1 BDD makeNode(BDD low, BDD high, int var){
2   if (high == ZERO){ return low }
3   BDD lookUp = uniqueTable.lookUp(low, high, var)
4   if(lookUp){ return lookUp }
5   return create(low, high, var)
6 }
```

Listing 3.18: Knotenerzeugung der ZDD-Umsetzung

Die ZDD-spezifischen Methoden implementieren jedoch differierende Algorithmen, weshalb die Creator-Schicht ausgetauscht wird. Dafür wurde ein neues Interface für ZDD-Operationen definiert und die Methoden gemäß Tabelle 2.3 benannt.
Da die ZDD-Methoden teilweise keine vollständige Shannon-Zerteilung unterstützen, können sie nicht durch generelle Konzepte, wie dem Apply- oder ITE-Algorithmus, umgesetzt werden. Die implementierten ZDDCreator-Klassen realisieren daher sämtliche Algorithmen einzeln. Parallelisiert wurde mit Hilfe der ForkJoin-Bibliothek analog zu Abschnitt 3.3.3.5.
Folgender Auszug zeigt die Parallelisierung an der Union-Methode exemplarisch.

```
1  BDD union(BDD f1, BDD f2){
2    if(f1 == FALSE){ return f2 }
3    if(f2 == FALSE){ return f1 }
4    if(f1 == f2){ return f1 }
5    BDD res = cacheCheck(f1, f2, OP_UNION)
6    if(res){ return res }
7    if(level(f1) < level(f2)){
8      res = makeNode(union(low(f1), f2), high(f1), var(f1))
9    }else if(level(f1) == level(f2)){
10     lowTask = forkJoinPool.submit(
11       () -> union(lowF1, lowF2))
12     highTask = forkJoinPool.submit(
13       () -> union(highF1, highF2))
14     res = makeNode(lowTask.join(), highTask.join(), topVar)
15   }else{
16     res = makeNode(union(f1, low(f2)), high(f2), var(f2));
17   }
18   cache(f1, f2, OP_UNION, res)
19   return res
20 }
```

Listing 3.19: Asynchrone Umsetzung der Union-Methode

Entwickelt wurde außerdem eine serielle Version der Algorithmen.

3.6 Weitere Komponenten

Die vorherigen Abschnitte dokumentieren den Entwicklungsprozess der Hauptkomponenten der Bibliothek, wohingegen dieser nun kleinere, aber nicht minder wichtige Details behandelt und vorstellt.

3.6.1 Hashfunktion

Ein wichtiger Bestandteil der Implementierung ist die Hashfunktion. Da in jedem nicht terminierenden Rekursionsschritt jeder Operation eine Suche im Computation-Cache und gegebenenfalls in der Uniquetable durchgeführt wird, hängt die Performanz maßgeblich an dieser Funktion. Das bedeutet, die Berechnung muss sehr performant sein und die Ergebnisse möglichst eindeutig. Aktuell ist folgende Funktion implementiert.

```
1 int hash(int a, int b){
2   return ((a + b) * (a + b + 1) / 2 + a)
3 }
4
5 int hash(int a, int b, int c){
6   return hash(c, hash(a, b))
7 }
```

Listing 3.20: Implementierte Hashfunktion

3.6.2 Threadpool-Management

Zur besseren Kontrolle der Parallelisierung und dem einhergehenden Overhead verwendet die Bibliothek ein eigenes Interface zum Threadpool-Management. Dieses bietet einen Getter für einen ForkJoinPool und einfache Methoden die zur Taskreduzierung verwendet werden. So wird in den parallelen Algorithmen mit der Methode *canFork* überprüft, ob der nächste Rekursionsschritt parallelisiert wird oder nicht. Die Standardimplementierung prüft, ob mindestens ein Thread zur Verfügung steht.

```
1 boolean canFork(int level){
2   return (taskCount < threadCount)
3 }
```

Listing 3.21: Einfache Umsetzung der canFork-Überprüfung

3.6.3 Im- und Exportfunktionen

Zu Visualisierungs- und Testzwecken bietet PJBDD mehrere Im- und Exportfunktionen. Hierfür existiert je ein Interface mit je einer Methode, die die Konvertierung eines Knotens zum bzw. vom gewünschten String-Format durchführt. Darüber hinaus besitzen die Schnittstellen default-Methoden, um das gewünschte Format in eine Datei zu schreiben bzw. von einer Datei zu lesen. Aktuell werden zwei Formate unterstützt, das standardisierte Graph .dot-Format und eine Eigendefinition.
Das .dot-Format wurde gemäß BDD-Standards entwickelt. Das bedeutet, jeder Nicht-Terminalknoten wird durch seine Variable gekennzeichnet und besitzt zwei ausgehende

Kanten. Die low-Kante wird gestrichelt dargestellt und die zum high-Kind durchgezogen. Die beiden Blätter werden mit 0 und 1 betitelt.
Das eigene Format dient lediglich dem einfachen Im- und Export. Es besteht aus einer Header-Zeile, die den Index des Wurzelknotens enthält, gefolgt von der Variablensortierung. Alle weiteren Zeilen repräsentieren den exportierten BDD. Je vier Werte (Index; Variable; low; high) stellen einen Knoten dar und werden durch ein Semikolon getrennt. Diese Kodierung wird bevorzugt für JUnit-Tests verwendet.

3.6.4 JUnit-Tests

Für alle booleschen Operationen wurden Tests zur Terminalfallüberprüfung angelegt. Zudem gibt es einen Testfall, der das N-Damenproblem für $N = 4, 5, 6, 7$ durchspielt.
Um das Ändern der Variablensortierung zu überprüfen, werden zunächst einfache BDDs erstellt bevor die Sortierung geändert und die neue Reihenfolge validiert wird. Außerdem wird das Damenproblem mit $N = 4$ zweimal gebildet, einmal vor der Neuordnung und einmal danach. Anschließend werden beide auf Gleichheit überprüft.
Zur automatisierten Generierung der Testfälle für alle möglichen Creator und Uniquetable-Kombinationen, wird von der abstrakten Testklasse CreatorCombinator geerbt.
Die init-Funktion dieser Klasse erzeugt alle möglichen Kombinationen und ruft für jede die abstrakte Methode test(creator) auf, die den jeweiligen Testfall umsetzen muss. Für die ZDD-Test-Implementierungen wurden die Methoden analog realisiert.

In diesem Hauptabschnitt wurde der Implementierungsprozess der parallelen BDD-/ZDD-Bibliothek in Java (PJBDD) ausführlich beschrieben. Es wurden einzelne Komponenten definiert und unterschiedliche Ausführungen umgesetzt. Im nächsten Kapitel (4) werden diese Entwicklungsergebnisse zusammengefasst und dokumentiert.

4 Dokumentation

Vorherige Abschnitte fokussieren die Implementierung eines asynchronen BDD- und ZDD-Frameworks, das unterschiedliche Java-Konzepte zur Parallelisierung der Operationen einbindet. Dieser Gliederungspunkt bietet eine umfassende Dokumentation der daraus resultierenden Java-Bibliothek PJBDD. Die beiden Interfaces Creator und BDD bilden die Hauptbestandteile der Umsetzungen.

4.1 BDD-Interface und Implementierung

Das BDD-Interface repräsentiert die einzelnen Knoten der BDDs und bietet dem Nutzer folgende Methoden:

- **getVariable():** Gibt die Variable des Knoten-Objekts als int-Wert zurück.
- **getLow():** Liefert den nachfolgenden false-Zweig des Knoten als BDD.
- **getHigh():** Liefert den nachfolgenden true-Zweig des Knoten als BDD.
- **isLeaf():** Liefert einen Wahrheitswert, ob der Knoten entweder der logischen 1 oder 0 entspricht.
- **isTrue():** Liefert einen Wahrheitswert, ob der Knoten der logischen 1 entspricht.
- **isFalse():** Liefert einen Wahrheitswert, ob der Knoten der logischen 0 entspricht.

Um unterschiedliche BDD-Implementierungen dynamisch unterstützen zu können, wird zur Instantiierung der Knoten ein BDD.Factory-Interface zur Verfügung gestellt. Die Aufgabe dieses Bestandteils ist es, Objekte der gewünschten BDD-Unterklasse zu erzeugen oder gegebenenfalls nach einer Variablenumsortierung zu ändern.
Das UML-Diagramm in Abbildung 4.1 visualisiert die Umsetzung des BDD-Interfaces und der korrespondierenden BDD.Factory.

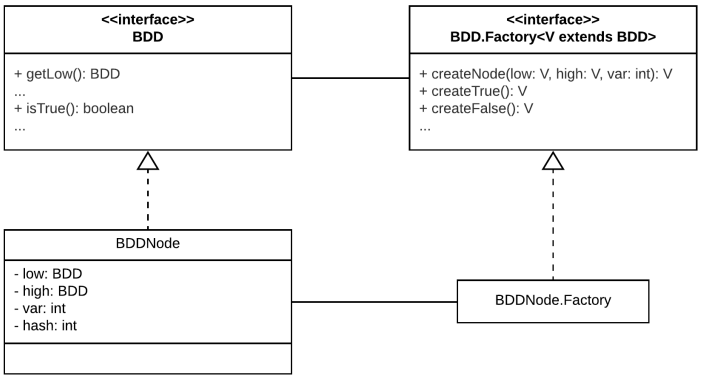


Abbildung 4.1: UML-Diagramm der BDD-Implementierung (Quelle: eigene Darstellung)

Die BDD-Bibliothek bietet mit der Klasse BDDNode eine simple rekursive Umsetzung des Interfaces. Objekte dieser Klasse halten folgende Variablen:

- **var:** Entspricht der BDD-Variable als int-Wert.
- **low:** Entspricht dem false-Zweig.
- **high:** Entspricht dem true-Zweig.
- **hashCode:** Da sich der Hashwerten aus der Variable und den beiden Hashwert der Kindknoten zusammensetzt, wird der Wert lediglich beim Erzeugen der Objekte berechnet und gespeichert. Denn eine Neuberechnung des Wertes würde eine rekursive Neuberechnung für alle Knoten im Baum erfordern.

4.2 Creator-Interface

Das Creator-Interface ermöglicht die Anwendung einer Vielzahl an Operationen zur Darstellung und Manipulation von BDDs. Es bildet das Kernstück der Bibliothek. Implementierungen dieser Klasse müssen die Logik aller booleschen Methoden sowie sämtliche Reduktionsregeln und Optimierungen umsetzen.

4.2.1 Creator-Implementierungen

Alle Implementierungen des Creator-Interfaces beruhen auf drei abstrakten Klassen: Der Klasse AbstractCreator, die alle spezifischen Komponenten hält (siehe Abschnitte 4.2.1.1 und 4.2.1.2), der Klasse BDDCreator, die die grundlegenden Algorithmen realisiert und von AbstractCreator erbt sowie ParallelCreator, der von BDDCreator erbt und die Basisalgorithmen auf die Parallelisierung vorbereitet.
Alle weiteren Creator-Klassen erben von einer dieser Klassen und beschäftigen sich lediglich mit je einer Java-Technologie zur Parallelisierung der Operationen. Folgende Auflistung liefert einen kompakten Überblick der umgesetzten Creator-Klassen, deren Vererbungshierarchie das UML-Diagramm in 4.2 veranschaulicht.

- **CompletableFutureITECreator:** Nutzt CompletableFuture Objekte aus der Java-Standardbibliothek zur parallelen Berechnung des ITE-Algorithmus.
- **FutureITECreator:** Nutzt Future Objekte aus der Java-Standardbibliothek zur parallelen Berechnung des ITE-Algorithmus.
- **GuavaFutureITECreator:** Nutzt ListenableFuture Objekte aus der Google-Guava-Bibliothek zur parallelen Berechnung des ITE-Algorithmus.
- **SerialITECreator:** Nutzt eine serielle Berechnung des ITE-Algorithmus.
- **StreamITECreator:** Nutzt einen parallelen Stream zur asynchronen Berechnung des ITE-Algorithmus.
- **ForkJoinITECreator:** Nutzt die ForkJoin-Bibliothek zur parallelen Berechnung des ITE-Algorithmus.
- **ApplyCreator:** Nutzt die ForkJoin-Java-Standardbibliothek zur parallelen Berechnung des Apply-Algorithmus.
- **CompletableApplyCreator:** Nutzt CompletableFutures zur parallelen Berechnung des Apply-Algorithmus.
- **SerialApplyCreator:** Nutzt eine serielle Berechnung des Apply-Algorithmus.

Wie bereits ausgeführt basiert die Umsetzung auf mehreren Komponenten, die je durch ein Interface beschrieben werden. Diese Aufspaltung wurde eingeführt, um die Implementierung einzelner Funktionen möglichst unabhängig und austauschbar zu halten. In den

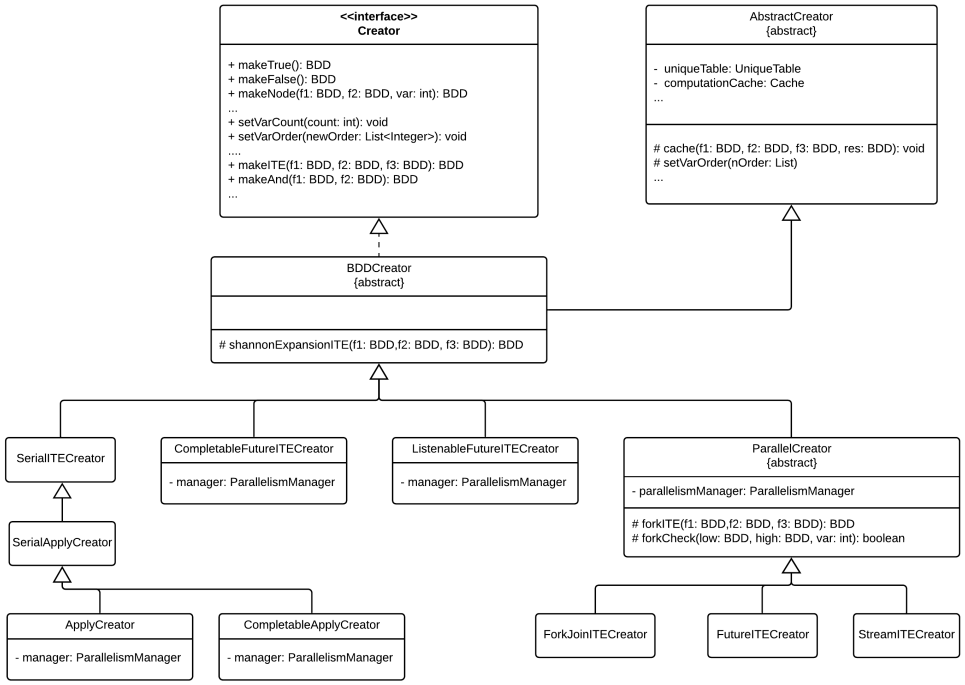


Abbildung 4.2: Vererbungshierarchie der Creator-Implementierungen (Quelle: eigene Darstellung)

nächsten Unterpunkten folgt die Dokumentation dieser Einzelteile, beginnend mit dem Wichtigsten, der Uniquetable.

4.2.1.1 Uniquetable-Interface und Umsetzung

Implementierungen des Uniquetable-Interfaces müssen die Wiederverwendung bereits existenter BDDs, das automatische Löschen nicht mehr verwendeter Knoten und einen threadsicheren parallelen Zugriff gewährleisten. Grafik 4.3 zeigt eine UML-Übersicht der umgesetzten Klassen, deren Unterschiede in folgender Übersicht zusammengefasst werden:

- **BDDConcurrentWeakHashDeque:** Basiert auf einer ConcurrentMap, die mit Hilfe von ConcurrentDeques Hashwert-Kollisionen behandelt.
- **BDDConcurrentWeakHashMap:** Nutzt eine ConcurrentMap, die gespeicherte BDDs sowohl in Key als auch in Value eines Eintrags ablegt.
- **BDDLockOnWriteArray:** Nutzt intern drei Arrays, die bei Bedarf vergrößert werden. Ein Array wird verwendet um die BDD-Einträge abzulegen. Die anderen beiden zur Realisierung des Hashbucket-Prinzips. Zur Gewährleistung threadsicherer und paralleler Zugriffe wird ein ReadWriteLock verwendet.
- **BDDConcurrentArray:** Basiert auf derselben Umsetzung wie das BDDLockOnWriteArray, jedoch wird anstelle eines ReadWriteLocks eine benutzerdefinierte Anzahl an Locks eingesetzt, die je einen bestimmten Teil der Tabelle sperren.

Eine ausführliche Analyse der vorgestellten Versionen folgt in Kapitel 5.

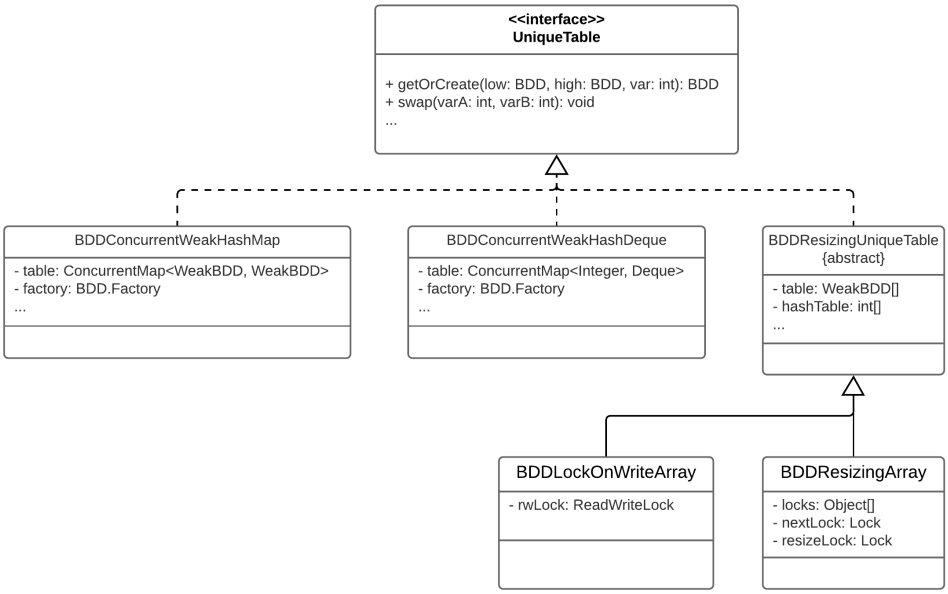


Abbildung 4.3: BDD-Uniquetable-Implementierungen (Quelle: eigene Darstellung)

4.2.1.2 Weitere Komponenten

Im wesentlichen enthält die Creator-Implementierung neben der Uniquetable eine Cache-Komponente, das BDDVariableManager-Interface und für parallele Lösungen den ParallelismManager. Das UML-Diagramm in 4.4 visualisiert die Zusammenhänge der abstrakten Creator-Subklassen und der in diesem Punkt dokumentierten Komponenten.

Der **Cache** ist ein generisches Interface und abstrahiert den Zwischenspeicher bereits berechneter Operationen. Dafür wird der Input zum Resultat gespeichert. Mit den Funktionsargumenten kann überprüft werden, ob bereits ein passendes Ergebnis vorliegt. PJBDD bietet aktuell zwei Umsetzungen:

1. **GuavaCache**: Basiert auf einem Cache aus der Guava-Bibliothek mit einer fixen Größe und einer zeitlich begrenzten Verfügbarkeit der Einträge.
2. **ArrayCache**: Basiert auf einem normalen Array mit fester Größe und einem Array an Monitorobjekten zur Synchronisierung einzelner Array-Bereiche.

Die Komponente **BDDVariableManager** ist für das BDD-Variable-Handling zuständig, sprich für die Verwaltung der Ordnung. Das bedeutet nicht nur, dass das jeweilige Level der Variablen gespeichert wird, sondern auch dass sie das Löschen bereits initialisierter Variablen verhindert. Es ist nicht erforderlich, dass die Komponente threadsicher ist, da alle modifizierenden Methoden synchronisiert sind. Während einer Variablenneusortierung wird beispielsweise die Creator-Umgebung gesperrt.
PJBDD enthält zwei Umsetzungen mit unterschiedlichen zugrunde liegenden Datenstrukturen:

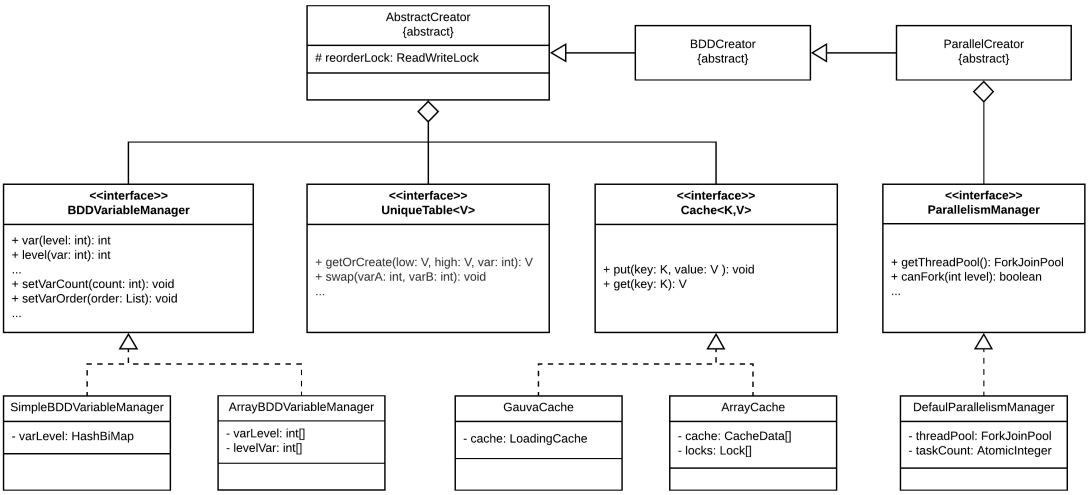


Abbildung 4.4: UML-Diagramm der Creator-Komponenten (Quelle: eigene Darstellung)

1. **SimpleBDDVariableManager**: Nutzt eine HashBiMap, die Variablen zum zugehörigen Level speichert, um sowohl performante Zugriffe auf das Level einer Variable, als auch auf die Variable eines bestimmten Levels zu erlauben.
2. **ArrayBDDVariableManager**: Basiert auf zwei einfachen int-Arrays zur Speicherung der Variablen und deren Level. In einem wird die Variable zum Level-Index gespeichert und im anderen das Level zum Variable-Index.

Ein weiterer wichtiger Bestandteil der nebenläufigen Lösungen ist der **ParallelismManager**. Er dient zur Verwaltung des Worker-Threadpools, der zur Parallelisierung der Operationen verwendet wird. Das heißt, die Komponente muss einen ForkJoinPool halten und entscheiden, ob ein fork zu einem bestimmten Zeitpunkt sinnvoll ist oder ob der aktuelle Thread weiterverwendet werden soll. Diese Maßnahme dient dazu Overhead aufgrund von Overthreating zu vermeiden.
PJBDD prüft vor jedem Rekursionsschritt mit canFork, ob dieser parallelisiert werden soll. Zum Start jedes asynchronen Jobs wird taskSupplied und mit dessen Ende taskDone aufgerufen.
Die Bibliothek bietet mit dem DefaultParallelismManager eine Umsetzung, die einen ForkJoinPool mit frei definierbarer Anzahl an Workerthreads[1] hält. Eingereichte Jobs werden mitgezählt und deren Anzahl bestimmt den Rückgabewert von canFork[2]. Zukünftige Implementierungen bleibt es überlassen vor dem eigentlichen Check zu prüfen, ob der jeweilige Rekursionsschritt in einem vordefinierten Block liegt, um die Anzahl komplexer boolescher Tests zu reduzieren.

Zusätzlich zu der objektorientierten Umsetzung wurde in Kapitel 3 eine Version realisiert, die mit primitiven int-Werten zur BDD-Darstellung rechnet.

[1] standardmäßig wird Runtime.getRuntime().availableProcessors() verwendet.
[2] canFork = (jobCount < threadCount)

4.3 int-basierte Umsetzung

PJBDD realisiert zusätzlich eine alternative Variante des BDD-Frameworks, die zur Berechnung boolescher Operationen intern keine Java-Objekte, sondern lediglich int-Werte verwendet und dennoch objektorientierte Schnittstellen bietet. Die Umsetzung der Algorithmen und die Parallelisierung dieser ist größten Teils äquivalent zur in Abschnitt 4.2.1 vorgestellten Version. Der Hauptunterschied ist, dass ein BDD-Graph nicht länger durch ein BDD-Objekt, sondern durch einen int-Tabellenindex repräsentiert wird. Um automatisiertes Löschen nicht mehr benötigter Knoten zu unterstützen, muss deshalb die Referenzanzahl mitgezählt werden. Die Dokumentation dieser Version erfolgt in den nächsten drei Unterpunkten.

4.3.1 BDD-Implementierung

Grundsätzlich besteht ein BDD intern aus je fünf Einträgen eines int-Arrays:

- **Eintrag 1:** wird mittels Bitshifting in zwei Werte aufgespalten, einer repräsentiert die Anzahl an Referenzen auf diesen Knoten und der andere die BDD-Variable.
- **Eintrag 2** & **3:** speichern den Index des nachfolgenden Low- bzw. High-Kindknotens.
- **Eintrag 4:** bezeichnet den Hashwert eines Knotens und speichert den Index des zugehörigen BDD. Dieser Eintrag wird benötigt, um bereits existierende Knoten mit Hilfe ihrer Kinder und Variablen zu finden.
- **Eintrag 5:** speichert den Index des nächsten Knoten mit identischem Hashwert und dient dazu Hashkollisionen zu behandeln.

Für die externe Nutzung bietet die Bibliothek eine Wrapperklasse, die das BDD-Interface implementiert. Dafür wurde eine innere Klasse der Creator-Implementierung angelegt. Objekte dieser Klasse halten lediglich den Index der Repräsentation in der Tabelle. Zudem erhöht jede Erzeugung eines solchen Objekts die Anzahl an Referenzen des zugehörigen Knotens.

4.3.2 Creator-Implementierung

Die Implementierung des Creator-Interfaces besteht aus zwei Schichten:

1. **IntCreator:** Dient als Schnittstelle und ermöglicht eine objektorientierte Nutzung. Darüber hinaus halten Objekte dieser Klasse WeakReferences auf die erzeugten BDD-Wrapper-Objekte, um bei Garbage Collection die Referenzanzahl anzupassen.

2. **SerialIntCreator, ForkJoinIntCreator und CompletableFutureIntCreator:** Erben von der Klasse IntCreator und setzen die interne Berechnung der booleschen Operationen auf die rein int-basierte Datenstruktur um. Die Klassen ForkJoinIntCreator und CompletableFutureIntCreator implementieren je eine parallele Berechnung des Apply-Algorithmus und SerialIntCreator eine serielle.

Das UML-Diagramm in 4.5 zeigt die Vererbungshierarchie der Umsetzungen. Die beiden parallelisierten Creatoren erweitern die serielle Version lediglich um die Parallelisierungsstrategie analog zu den in Abschnitt 4.2.1 vorgestellten Konzepten. Ansonsten setzen

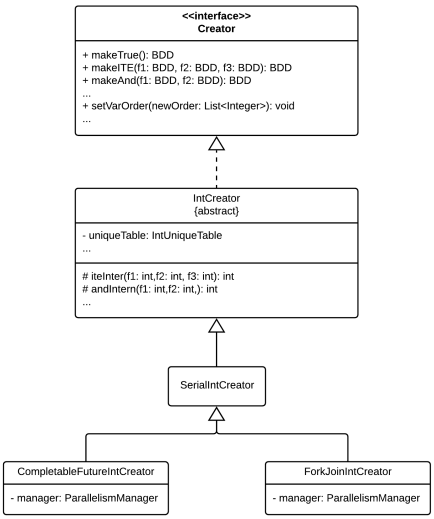


Abbildung 4.5: UML-Diagramm der int-Creator-Implementierungen
(Quelle: eigene Darstellung)

sich die Implementierungen äquivalent zu den objektorientierten Umsetzungen aus unterschiedlichen Komponenten zusammen, wobei jede ein Pendant in der anderen Version hat und derselbe ParallelismManager verwendet wird. Grafik 4.6 zeigt die Komponentenaufteilung der Umsetzung, die in Unterpunkt 4.3.3 knapp präsentiert wird.

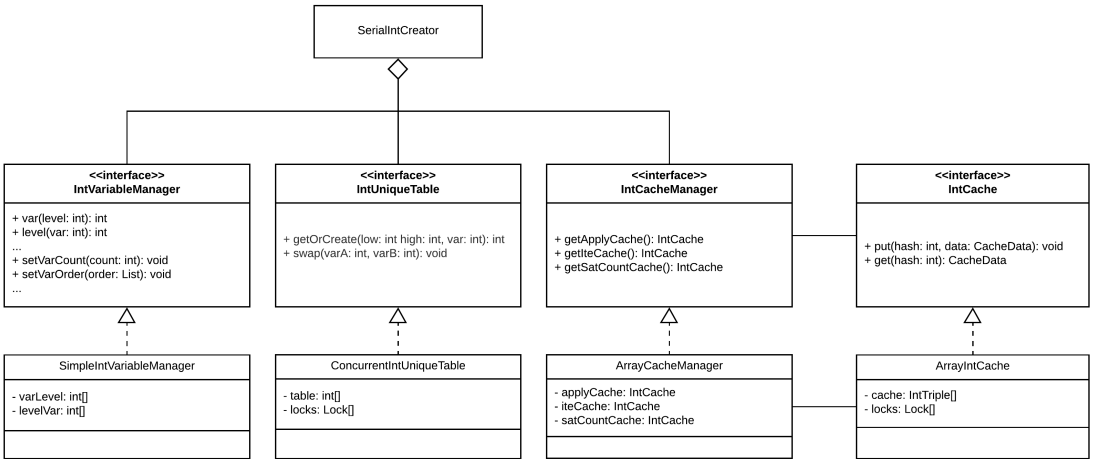


Abbildung 4.6: UML-Diagramm der int-basierten Komponenten
(Quelle: eigene Darstellung)

4.3.3 int-basierte Komponenten

Die wichtigste Komponente bildet wiederum die Uniquetable bzw. das **IntUniquetable-Interface**. Da der Vorteil dieser Implementierung darin liegt, dass für Berechnungen

keine BDD-Objekte erzeugt werden, wird ein eigenes int-basiertes Uniquetable-Interface benötigt. Implementiert wird das Interface mit einem int-Array, in das für jeden Knoten, wie in Abschnitt 4.3.1 beschrieben, fünf Einträge geschrieben werden. Um threadsicheren sowie parallelen Zugriff zu gewährleisten, synchronisiert die Umsetzung ConcurrentIntUniquetable einzelne Hashbereiche.

Der **CacheManager** dient zur Verwaltung der unterschiedlichen Zwischenspeicher für bereits berechnete Operationen. Dieser beinhaltet je einen NOT-, APPLY-, ITE- und SATCOUNT-Cache. Umgesetzt wurden eine Klasse ArrayIntCacheManager, die einfache Arrays und Segmentlocks nutzt.

Die Komponente **IntVariableManager** bildet das Pendant zum **BDDVariableManager**. Die Umsetzung SimpleIntVariableManager basiert auf zwei Arrays, die ebenfalls Variablen und deren Level speichern. Außerdem wird für die Parallelisierung der **ParallelismManager** analog verwendet.

4.4 ZDD-Creator

Das ZDDCreator-Interface definiert die in den ZDD-Grundlagenkapiteln 2.1.4 und 2.1.5 vorgestellten typischen Operationen (siehe Tabelle 2.3). Implementiert wird dieses von den Klassen ZDDConcurrentCreator und ZDDSerialCreator. Betrachtet man das UML-Diagramm der Umsetzungen in 4.7 so fällt auf, dass der ZDDSerialCreator von derselben abstrakten Klasse erbt, wie die BDD-Umgebungen.

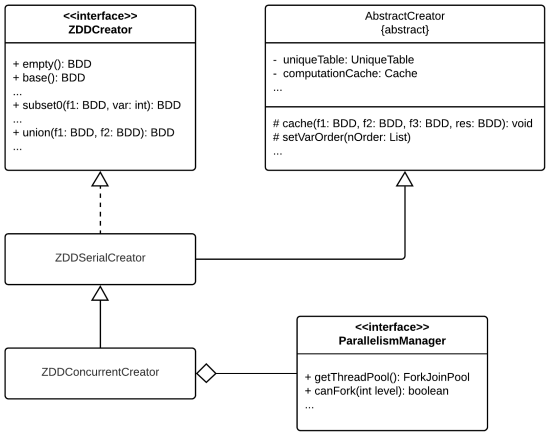


Abbildung 4.7: UML-Diagramm der ZDD-Implementierung (Quelle: eigene Darstellung)

Das bedeutet, dass alle Creator-typischen Komponenten der objektorientierten Umsetzung wiederverwendet werden. Dadurch kann der ZDDConcurrentCreator beispielsweise mit allen existierenden Uniquetable-Implementierungen kombiniert werden.
Die Algorithmen in der Klasse ZDDConcurrentCreator wurden mit Hilfe der ForkJoin-Bibliothek parallelisiert.

4.5 Sonstige Klassen

Neben den bereits vorgestellten Komponenten bietet PJBDD einige weitere Klassen, die nützliche Anwendungsmöglichkeiten mit sich bringen. Folgende Auflistung liefert einen kompakten Überblick über diese und deren Funktionen.

- **HashCodeGenerator**: Diese Klasse stellt statische Methoden zur Hashwertberechnung aus mehreren einzelnen int-Werten zur Verfügung. Eine gute Hashfunktion ist notwendig, um möglichst viele Hashkollisionen zu vermeiden, da diese einen maßgeblichen Einfluss auf die Performanz der Bibliothek haben.
- **Exporter**: Das Exporter-Interface dient dazu, BDD-Objekte in eine bestimmte String-Repräsentation zu konvertieren und gegebenenfalls in eine Datei zu schreiben. Konkret implementiert wurden zwei Exporter, einer erzeugt .dot-Graphen und der andere das in 3.6.3 vorgestellte eigenentwickelte Format.
- **Importer**: Als Gegenstück zum Exporter- dient das Importer-Interface dazu BDD-Objekte aus bestimmten String-Repräsentation zu importieren oder gegebenenfalls direkt aus einer Datei zu lesen. Implementiert wurde das jeweilige Pendant zu den Exporter-Umsetzungen.

4.6 Verwendung der Bibliothek

Um eine einfache Verwendung zu gewährleisten, verfolgt PJBDD das Builder-Pattern. Hierfür können mit der Klasse CreatorBuilder neue Creator-Kombinationen mit beliebig feiner Parameterkonfiguration erstellt oder einfach vordefinierte Standardwerte verwendet werden. Es werden sowohl die BDD-, int- als auch ZDD-Creatoren unterstützt.
Eine genaue Auflistung und Beschreibung der Methoden und der potentiell möglichen Konfigurationen ist der JavaDoc zu entnehmen.
Programmcodeauszug 4.1 zeigt beispielhaft eine einfache Initialisierung mit Standardparametern und eine mit granularer Konfiguration.

```
1 // (1) Instantiierung mit Standardkonfiguration
2 Creator cDefault = CreatorBuilder.newBuilder().build();
3
4 // (2) Instantiierung mit granularer Konfiguration
5 int threads = Runtime.getRuntime().availableProcessors();
6 Creator cCustom = CreatorBuilder.newBuilder()
7   .setVarNum(10).setTableSize(500000).setParallelism(1000)
8   .setNumWorkerThreads(threads).setCacheSize(10000)
9   .useConcurrentResizingArray().makeConcurrentApplyCreator();
```

Listing 4.1: Creator-Instantiierung mittels der Builder-Klasse

Die zweite Einstellung zeigt zudem die Standardwerte.

Damit ist die Dokumentation der implementierten Bibliothek PJBDD abgeschlossen. Im nächsten Hauptabschnitt folgt eine ausführliche Analyse aller diskutierten Komponenten und ein Vergleich mit bestehenden Frameworks.

5 Evaluation

In diesem Passus wird das im Rahmen der Arbeit implementierte nebenläufige BDD-Framework analysiert und mit bisherigen Standardbibliotheken verglichen. Dafür werden in 5.2.1 zunächst die implementierten Komponenten untersucht, anschließend in 5.2.2 den aktuellen Standards gegenübergestellt und zum Abschluss wird die Integration in das Softwareverifikationstool CPAchecker eruiert.
Zur Evaluation der Bibliothek wurden Performanz- und Speicherbedarf-Benchmarks erstellt. Bevor die Ergebnisse der Auswertung präsentiert werden, liefert der folgende Unterabschnitt noch Hintergrundwissen für zuverlässiges Benchmarking sowie mögliche Tücken mit Java.

5.1 Fundiertes Benchmarking in Java

In (BLW17) arbeiten Beyer et al. detailliert aus, welche Kriterien für zuverlässige und reproduzierbare Benchmarks beachtet werden müssen. In dem Paper werden sechs Punkte aufgeführt, die generell zu berücksichtigen sind.

1. **Sorgfältige Messung und Begrenzung der Ressourcen:**
   Dies gilt sowohl für Zeit-, als auch Speicherbedarfsanalysen. Dabei muss beachtet werden, dass die Laufzeiterfassung[1] (Walltime) nicht durch Änderungen der Systemuhr verfälscht wird. Die Messung muss dementsprechend auf einer strikt monotonen Quelle basieren.
   Für eine korrekte Messung der CPU-Zeit ist dafür Sorge zutragen, die CPU-Zeit aller Kindprozesse ebenfalls korrekt zu erfassen.
   Mit Speicherbedarfsanalysen wird der Maximalwert an Bedarf gemessen, dieser entspricht dem minimal benötigten Speicher, um das Programm mit demselben Ergebnis auszuführen. Bei der Erhebung dieser Metrik in Java ist zu bedenken, dass die Größe des Heaps nicht verwendet werden kann, da die des Stacks ebenfalls zu berücksichtigen ist. Des weiteren unterscheidet sich der virtuell allozierte Speicher oft gravierend vom tatsächlichen physischen Bedarf, da Linux diesen erst belegt, sobald der virtuelle beschrieben wird. Die sogenannte Resident-Set-Size (RSS) kann ebenfalls nicht ohne weiteres verwendet werden, weil eventuell ausgelagerter Speicher nicht miteinbezogen wird.
   Außerdem sollte immer eine feste Begrenzung des Speicherbedarfs gesetzt werden. Ansonsten ist dieser vom aktuellen Zustand des Systems abhängig und gegebenenfalls nicht reproduzierbar.

[1] Laufzeit bezeichnet die tatsächliche Zeit, die zur Ausführung benötigt wird.

2. **Freigabe aller Prozesse und Ressourcen:**
   Am Ende jedes Durchlaufs muss sichergestellt werden, dass die Benchmarkumgebung alle Kindprozesse beendet und Ressourcen freigibt. Ansonsten können die Ergebnisse nachfolgender Benchmarks auf derselben Maschine durch blockierte Ressourcen verfälscht werden.

3. **Sorgfältige Zuweisung von Prozessorkernen:**
   Bei der Zuweisung von Prozessorkernen muss bedacht werden, dass jeder Kern zwar die Ausführung eines Threads unabhängig von anderen Kernen ermöglicht, jedoch auf Hardwareebene aufgrund geteilter Ressourcen meistens nicht vollständig unabhängig ist. Eine parallele Ausführung mehrerer Benchmarks auf unterschiedlichen Kernen einer Maschine kann die Performanz damit negativ beeinflussen und sollte vermieden werden, insbesondere bei der Verwendung von Hyperthreating.

4. **Berücksichtigung von Nonuniform-Memory-Access:**
   Mehrkernige Systeme basieren oft auf einer Nonuniform-Memory-Access (NUMA)-Architektur. NUMA bedeutet, dass ein einzelner CPU auf bestimmte Speicherbereiche des Systems direkt zugreifen kann und auf andere Bereiche nur indirekt über andere CPUs. Da dieser indirekte Zugriff zu nicht vorhersehbaren Verzögerungen führen kann, sollte für einzelne Durchläufe immer der Speicher verwendet werden, der den jeweiligen CPUs zugeordnet ist.

5. **Vermeidung von Speicherauslagerung:**
   Auslagerung von Arbeitsspeicher auf Festplattenspeicher muss für Benchmarks unbedingt vermieden werden, da es zu nicht deterministischen Performanzverlusten führen kann. Um sicherzugehen, dass keine Auslagerung während eines Durchlaufs stattgefunden hat, kann ein Warnmechanismus hilfreich sein.

6. **Isolierung einzelner Durchläufe:**
   Einzelne Ausführungen eines Benchmark-Tools sollten unbedingt isoliert werden, um auszuschließen, dass sich mehrere parallele oder sequentielle Durchläufe gegenseitig beeinflussen. Gemeinsam genutzte Dateien können beispielsweise eine Anzahl an Ausführungen so beeinflussen, dass die Ergebnisse von der Reihenfolge abhängig sind.

Diese Liste stellt allgemeingültige Anforderungen an ein zuverlässiges Benchmarking.
Für Java-Anwendungen müssen noch eine Reihe weiterer Tücken bedacht werden, die zu fehlerhaften Ergebnissen führen können. Durch die Just-In-Time (JIT)-Kompilierung der Java-Virtual-Machine (JVM) kann beispielsweise ein einmaliger Funktionsaufruf deutlich langsamer sein, als er es in einem laufenden System tatsächlich wäre. Dadurch können allen voran kleine Benchmarks nicht deterministisch negativ beeinflusst werden.
Ein weiteres Problem im Hinblick auf fundierte Benchmarks ist die Codeoptimierung der JVM. Wird das Ergebnis eines Testlaufs nicht verwendet, kann die sogenannte Dead-Code-Eliminierung möglicherweise den gesamten Testaufruf einsparen, sollte der Compiler erkennen, dass die Ausgabe durch den Aufruf unverändert bleibt.
Um präzisere Ergebnisse zu erhalten, werden üblicherweise die Mittelwerte aus mehrfacher Ausführung der Testszenarien gebildet. Wird dies Mittels einer Schleife realisiert,

können ganze Durchläufe wegoptimiert werden, sollte der JIT-Compiler die identischen Iterationen erkennen, wodurch der Durchschnittswert erheblich verfälscht wird.
Basierend auf diesem Wissen wurden in den folgenden Punkten Benchmarks zur Bewertung der implementierten BDD-Bibliothek erstellt. In 5.2.1 und 5.2.2 werden Arbeitsabläufe getestet, die rein auf Bibliothek-internen Funktionalitäten basieren und in 5.4 wird die Integration als Teil eines bestehenden Programms evaluiert.

5.2 Evaluation der BDD-Bibliothek PJBDD

Im weiteren Verlauf wird das im Rahmen dieser Masterarbeit realisierte BDD-Framework PJBDD mit Hilfe der Bibliothek Java-Microbenchmark-Harness (JMH) analysiert. Diese Evaluation lässt sich in zwei Phasen unterteilen: Die Erste eruiert die bestmöglichen Kombinationen der unterschiedlichen Komponenten, in der Zweiten werden diese Variationen anderen Bibliotheken gegenübergestellt. Zur Bewertung wird jeweils das N-Damenproblem gelöst.

Das **N-Damenproblem** ist ein typisches kombinatorisches Problem, das mit BDDs und ZDDs gelöst werden kann. Die Aufgabenstellung ist, genau N Damen so auf einem Schachbrett der Größe $N \times N$ zu platzieren, dass keine eine andere schlagen kann. Abbildung 5.1 zeigt zur Veranschaulichung eine korrekte Stellung des 5-Damenproblems.

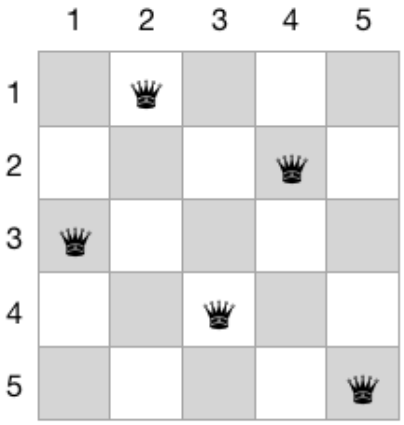


Abbildung 5.1: Eine Stellung des 5-Damenproblems (Quelle: eigene Darstellung)

Bereits für kleine N gibt es mehrere Millionen möglicher Stellungen und die Problemgröße steigt exponentiell. Zum Beispiel gibt es für $N = 5$ bereits 6.375.600 unterschiedliche Stellungen, wenn auch nur 10 gültige Lösungen.
Der Vorteil von BDDs/ZDDs gegenüber anderen Strategien wie zum Beispiel dem Backtracking ist, dass nicht nur eine mögliche Lösung dargestellt wird, sondern alle. Tabelle 5.1 zeigt die Anzahl der Lösungen in Abhängigkeit von N mit $N = [4, ..., 14]$.

N	4	5	6	7	8	9	10	11	12	13	14
Lösungen	2	10	4	40	92	352	724	2.680	14.200	73.712	365.596

Tabelle 5.1: Anzahl an Lösungen für das N-Damenproblem mit $N = 4, ..., 14$ (Quelle: eigene Darstellung)

Die Tabelle zeigt das exponentielle Wachstum des Problems sehr deutlich, weshalb es hervorragend für Benchmarkanalysen geeignet ist.
Die speziell für die beiden Testintervalle entwickelte Testumgebung, generiert mit Hilfe des JMH automatisiert Tests für unterschiedlichste Komponenten- und Parameterkombinationen.
Die Ausführung des Testprogramms erfolgte auf mehreren Rechnern mit Ubuntu 18.04, Intel i7-8700 Prozessor (6 Kerne, Hyperthreating und 64 GB-RAM) sowie der Java Version OpenJDK11. Um aussagekräftige Ergebnisse zu erzielen, wurden die maximale Heap-Größe der JVM auf 16GB beschränkt und keine parallelen Jobs auf einem Rechner erlaubt.
Die Ergebnisse jeder Einstellung setzen sich aus 15 Iterationen zusammen, die ersten fünf dienen zum Aufwärmen der JVM und der Mittelwert der verbleibenden 10 Durchläufe bildet das Resultat. Außerdem bleibt zu erwähnen, dass JMH das mögliche Inlining von identischen Funktionsaufrufen verhindert.

5.2.1 Evaluation der PJBDD-Komponenten

Nachfolgend werden die alternativen Komponentenrealisierungen aus PJBDD evaluiert. Für alle Tests ist der Parallelisierungsfaktor der Tabellen auf 1.000, die initiale Größe der Uniquetable auf 50.000, die Cachegröße auf 10.000 und die anfängliche Variablenanzahl abhängig von N auf $N * N$ festgelegt worden. Tabelle 5.2 zeigt alle Framework-Komponenten, die in diesem Abschnitt analysiert werden.

BDD	
VariableManager	ArrayManager, GuavaManager
Cache	ArrayCache, GuavaCache
Parallelisierung	Stream, Future, CompletableFuture, ListenableFuture, ForkJoin
Uniquetable	RWLockArray, ConcurrentArray, HashMap, HashDeque
Algorithmen	ITE, Apply
Anfragen	Sequentiell, Asynchron
int-Umsetzung	Serial, ForkJoin, CompletableFuture
ZDD	
Parallelisierung	Serial, ForkJoin
Uniquetable	RWLockArray, ConcurrentArray, HashMap, HashDeque
Methoden	nur boolesche Methoden, erweiterte ZDD-Methoden
Parameter	
N-Damenproblem	8, 9, 10, 11, 12
Workerthreads	2, 4, 6, 8, 10, 12

Tabelle 5.2: Zu evaluierende Komponenten

Um die Kombinationsmenge der Komponenten zu reduzieren, werden die Variationen ite-

rativ auf repräsentativen Teilmengen getestet und gegebenenfalls ausgeschlossen.
Im ersten Schritt erfolgt die Bewertung der **VariableManager**- und **Cache**-Alternativen. Dafür wird je eine serielle Creator-Umgebung und eine basierend auf der ForkJoin-Parallelisierung mit vier Workerthreads untersucht, indem jede Kombination ein kleines ($N = 8$) sowie ein großes ($N = 12$) Szenario mit dem Apply-Algorithmus lösen sollte.
Die Visualisierung der Testergebnisse der VariableManager-Vergleiche in Abbildung 5.2 liefert eindeutige Resultate: Der ArrayManager ist für die Szenarien um 25 bis 76% performanter als die Implementierung mit der Guava-HashBiMap.

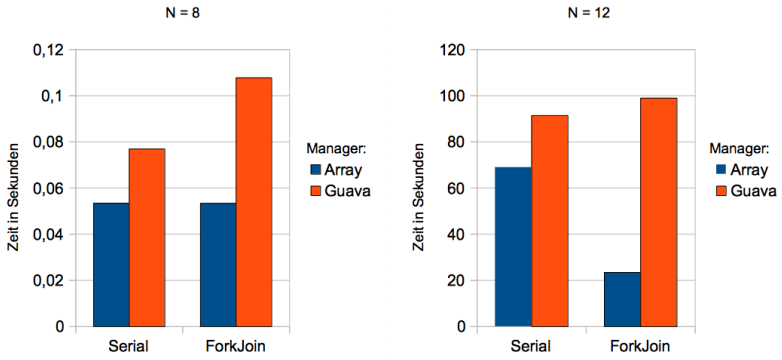


Abbildung 5.2: Vergleich der implementierten VariableManager
(Quelle: eigene Darstellung)

Außerdem zeigt das Diagramm in 5.3, dass der Cache basierend auf einem nebenläufigen Array ebenfalls um 30 bis 60% besser abscheidet, als die Version, die auf dem Guava-LoadingCache fundiert.

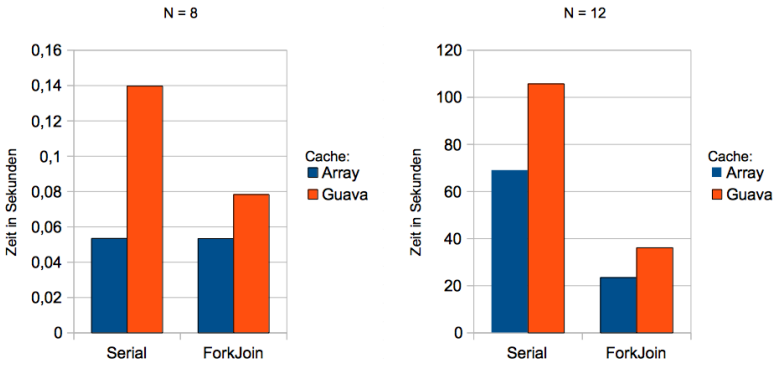
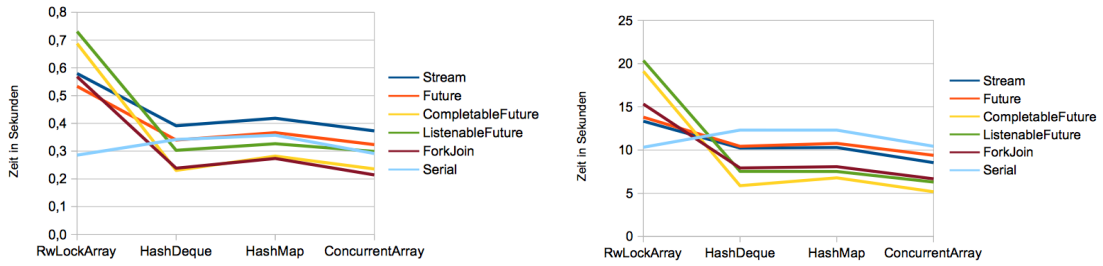


Abbildung 5.3: Direkter Vergleich der implementierten Caches
(Quelle: eigene Darstellung)

Allerdings hat der GuavaCache den Vorteil, dass Elemente nur für eine begrenzte Zeit gespeichert werden. Einträge im ArrayCache hingegen bleiben solange bestehen, bis sie durch Hashkollisionen überschrieben werden.
Aufgrund der Ergebnisse dieser Analysen wurden für die weiteren Tests der ArrayMana-

ger und ArrayCache verwendet.

Nachfolgend werden die **Java-Konzepte zur Parallelisierung** in Kombination mit den **Uniquetable**-Komponenten evaluiert. Dafür lösen in Abbildung 5.4 alle Kombinationen das 9- und 11-Damenproblem je mit Hilfe von sechs Workerthreads.



(a) N = 9 (b) N = 11

Abbildung 5.4: Vergleich aller möglichen Creator und Uniquetable Kombinationen (Quelle: eigene Darstellung)

Die Diagramme in 5.4 zeigen die Schwächen des Sperrkonzepts mit ReadWriteLock-Monitor auf. Es schneidet für die nebenläufigen Algorithmen deutlich schlechter ab, als die anderen Strategien. Die CompletableFuture-Lösung benötigt beispielsweise mit dem RWLockArray 19s für die Darstellung des 11-Damenproblems und mit dem Concurrent-Array nur 5s. Dass das tatsächlich am ReadWriteLock liegt, ist anhand zweier Faktoren festzumachen: Zum Einen sind die Ergebnisse des Konzepts bei seriellen Tests gut, zum Anderen unterscheidet sich das beste, das ConcurrentArray, lediglich im Sperrkonzept. Für weitere Benchmarks bedeutet das, dass das RWLockArray nicht mehr verwendet wird.
Um die oben dargestellten Resultate zu verfeinern, wurde ein weiterer Testlauf für das 12-Damenproblem mit identischem Setup initiiert. Der daraus resultierende Graph in Darstellung 5.5 erlaubt weitere Rückschlüsse.

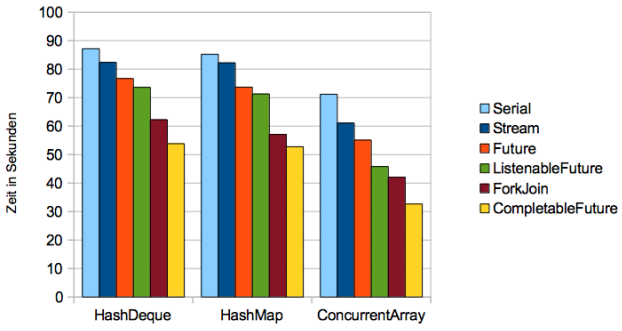


Abbildung 5.5: Vergleich Creator- und Uniquetable-Kombinationen in großen Szenarien (Quelle: eigene Darstellung)

Zeigen sich in kleinen Szenarien noch kaum Laufzeitunterschiede bei den untersuchten Uniquetable-Implementierungen, so stellt sich die Array-Implementierung für größere Testfälle deutlich als am besten geeignet dar. Mit Performanzvorteilen von 16% bei der seriellen und bis zu 38% bei der CompletableFuture-Version bietet sie die performanteste Komponentenalternative.
Gleichzeitig spiegelt sich der Nachteil der Future-Implementierung wieder, denn für größere Szenarien schneidet sie deutlich schlechter ab, als die parallelen Lösungen, die eine asynchrone Taskverknüpfung bieten.
Die schlechtesten Ergebnisse erzielt allerdings die auf dem parallelisierten Java 8 Stream basierende Umsetzung. Mit bis zu 46% längerer Laufzeit liegt sie weit hinter der Spitzengruppe. Die ListenableFutures realisieren dasselbe Konzept wie die CompletableFutures, dennoch liegen die Ergebnisse bei größeren Szenarien ($N = 11, 12$) um 20% - 25% deutlich hinter den CompletableFutures, zudem ist die ListenableFuture-Klasse nicht in der Java-Standardbibliothek enthalten.
Daraus resultierend werden für weitere Analysen lediglich die CompletableFuture- und die ForkJoin-Parallelisierung in Kombination mit der Array-Uniquetable berücksichtigt.

Im nächsten Schritt wird die generalisierte Implementierung der booleschen Methoden bewertet, indem der **ITE**- dem **Apply**-Algorithmus gegenübergestellt wird. Dafür wurde das 9- und 12-Damenproblem jeweils mit diesen Beiden sowie der seriellen, der ForkJoin- und der CompletableFuture-Version mit sechs Workerthreads gelöst. Abbildung 5.6 zeigt den Laufzeitvergleich der Testdurchläufe.

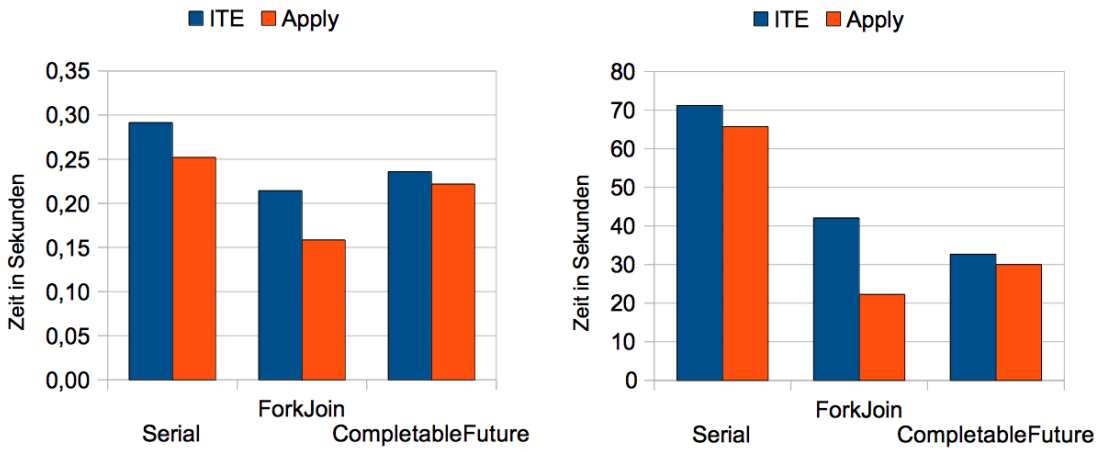


Abbildung 5.6: Vergleich ITE- und Apply-Algorithmus (Quelle: eigene Darstellung)

Die Ergebnisse sind eindeutig, denn sowohl die seriellen als auch die nebenläufigen Apply-Durchläufe waren schneller als analoge mit dem ITE. Besonders sticht die Laufzeitverbesserung um 47% bei der ForkJoin-Parallelisierung in großen Szenarien heraus. Schneidet bei den ITE-Implementierungen noch die CompletableFutures-Alternative besser ab, so stellt die Apply-ForkJoin-Umsetzung die schnellste Lösung dar. Für alle weiteren Analysen hat das zur Folge, dass lediglich Umsetzungen des Apply-Algorithmus getestet werden.

Abschließend wird noch die Auswirkung der Anzahl an **Workerthreads** untersucht. Da-

für wird verglichen, wie die CompletableFuture- und ForkJoin-Versionen mit bis zu 12 Threads arbeiten. Abbildung 5.7 zeigt die benötigte Zeit zur Lösung des N-Damenproblems mit $N = \{7, 8, 9, 10, 11, 12\}$. Die X-Achsen der Graphen beschreiben die Veränderung der Anzahl an Threads und die Y-Achse zeigt die Laufzeit zur Lösung des jeweiligen Problems in Sekunden.

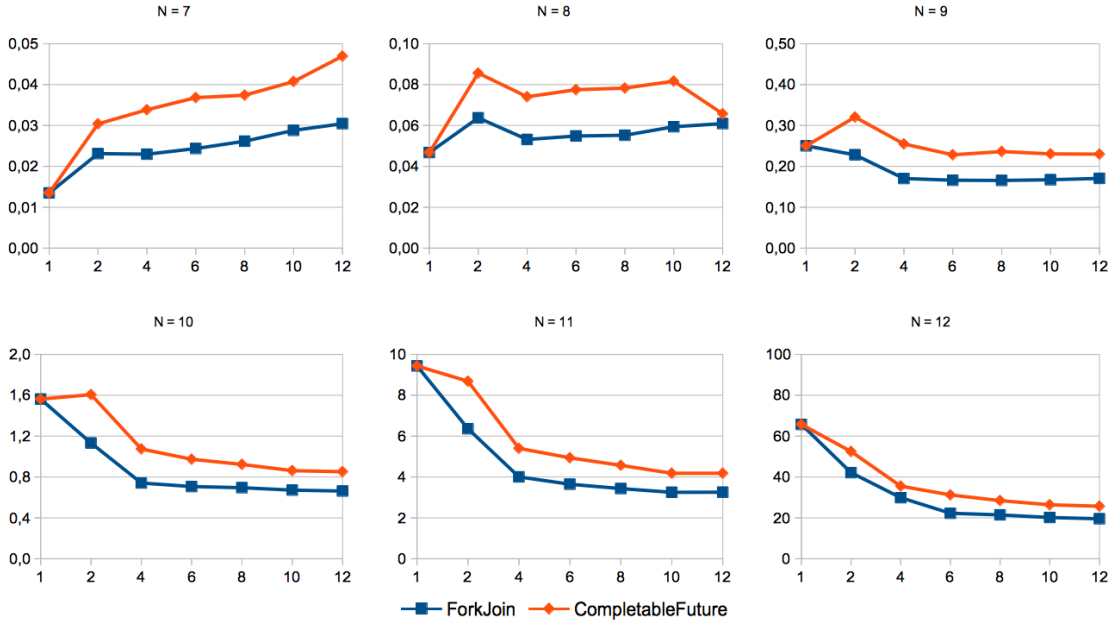


Abbildung 5.7: Laufzeiteinfluss der Anzahl an Threads (Quelle: eigene Darstellung)

Die Diagramme für $N > 9$ zeigen, dass sich die umgesetzte Parallelisierung bei zu kleinen Problemen nicht lohnt bzw. sogar langsamer ist, als die serielle Berechnung. Bei kleineren Problemen bewirkt die zu geringe Rekursionstiefe und somit zu kleine Taskgröße, dass der entstehende Overhead durch Thread-Kontextwechsel sowie Koordination Vorteile durch die parallele Bearbeitung negiert.
Bei sehr großen Anwendungsfällen zeigen sich deutliche Laufzeiteinsparungen. Für $N = 12$ konnte beispielsweise die Laufzeit von ca. 65s auf 19s gesenkt werden. Das entspricht einer Reduktion um 70%.
Darüber hinaus zeigen die Graphen für $N = [7, 8, 9]$, dass eine Bearbeitung mit zu vielen Threads hinsichtlich einer bestimmten Problemgröße ebenfalls zu Laufzeitverlusten führen kann. Denn betrachtet man den Trend, fällt ein Anstieg bei der Bearbeitungszeit mit 10 bzw. 12 Threads auf. Der Divide and Conquer-Natur geschuldet, werden die Probleme in so kleine Teile gespalten, dass die Kontextwechsel und der ansteigende Multithreating-Overhead nicht durch die zusätzlichen Threads aufgewogen werden können.

In Abschnitt 2.1 wird der Einfluss der Variablensortierung auf die Breite eines BDDs beschrieben. Da das N-Damenproblem bisher immer unter optimaler Variablenordnung gelöst wurde, flossen lediglich schmale, tiefe BDD-Bäume in die Untersuchung ein. Deshalb wird als Nächstes das N-Damenproblem mit $N = [7, 8, 9]$ und invertierter Variablenordnung analysiert, um den Wirkungsgrad für breite Bäume zu bewerten. Dafür wurde

es je von der seriellen sowie der nebenläufigen Lösung auf 8 Threads gelöst. Die Laufzeitanalysen in Darstellung 5.8 zeigen deutlich, dass zwar ebenfalls eine Verbesserung erzielt werden kann, jedoch nicht in dem Ausmaß wie für tiefe Bäume.

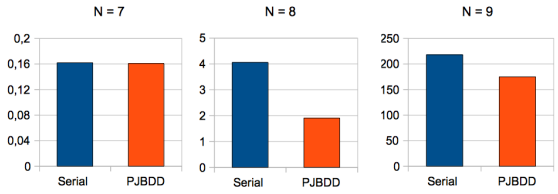


Abbildung 5.8: Laufzeit für das N-Damenproblem mit invertierter Variablenordnung (Quelle: eigene Darstellung)

Betrachtet man zum Beispiel die Grafik für $N = 9$, so besteht der BDD bereits aus mehr Knoten als einer für $N = 12$ bei optimaler Ordnung, dennoch kann die Laufzeit lediglich um 20 % von 218s auf 174s reduziert werden. Insgesamt folgt, dass die Parallelisierung für große Anwendungsfälle gut geeignet ist, insbesondere für tiefe Baumstrukturen.

Folgende Tabelle fasst die beste Komponenten-Kombination resultierend aus vorangehender Evaluation zusammen:

Parallelisierung	Algorithmus	UniqueTable	VariableManager	Cache
ForkJoin	Apply	ConcurrentArray	ArrayManager	ArrayCache

Zusätzlich zu diesen Implementierungen wurde in Kapitel 4.3 eine int-basierte Alternative vorgestellt.

5.2.1.1 Evaluation der int-basierten Umsetzung

Abschnitt 4.3 dokumentiert eine Version der BDD-Bibliothek, deren interne Algorithmen rein mit int-Werten rechnen und die mit Hilfe des ForkJoin-Frameworks und CompletableFutures zwei unterschiedliche Parallelisierungen bietet.
Zur Bewertung der Implementierung lösen diese beiden mit je sechs Threads sowie die serielle Version das 12-Damenproblem. Anschließend werden sie in Abbildung 5.9 ihrem objektorientierten Pendant gegenübergestellt.

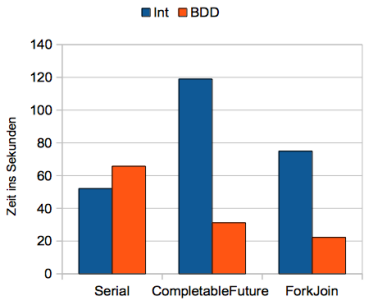


Abbildung 5.9: Analyse der int-basierten Umsetzungen (Quelle: eigene Darstellung)

Ist die serielle Version ca. 20% performanter als das objektorientierte Gegenstück, so zeigt der Vergleich sehr deutlich, dass die Parallelisierungen nicht den gewünschten Effekt erzielen können. Sie sorgen sogar für einen Laufzeitanstieg von 52s (seriell) auf 118s (CompletableFutures) und 75s (ForkJoin) in der Konfrontation zur seriellen Version.
Deshalb wurden die Durchläufe mit Hilfe des Profiling-Tools VisualVM genauer inspiziert. Das Profiling ergab, dass die Auslastung der Worker bei den nebenläufigen int-basierten Versionen im Schnitt lediglich bei 21% lag. Die Berechnungen erfolgten quasi sequentiell auf mehreren Threads. Performanzvorteile von mehrkernigen Lösungen beruhen hingegen darauf, dass das Ergebnis von mehreren Threads gleichzeitig und unabhängig berechnet wird. Zum Vergleich dazu konnten die Objektorientierten trotz analoger Umsetzung eine mittlere Auslastung von circa 70% erzielen.
Diesem Overhead geschuldet bietet die int-basierte Implementierung lediglich im sequentiellen Kontext und für kleine Anwendungsfälle eine Verbesserung. In einer mehrkernigen Umgebung bleibt die im vorigen Abschnitt erarbeitete objektorientierte Kombination die beste Variante.

Bevor diese BDD-Umgebungen mit anderen Bibliotheken verglichen werden, folgt ein kurzer Exkurs der eine Evaluation der in PJBDD enthaltenen ZDD-Realisierung durchführt.

5.2.1.2 Bewertung der ZDD-Implementierung

Im Grundlagenkapitel 2.1.3 wurden ZDDs als alternative BDD-Datenstruktur vorgestellt und in Sektion 4.4 wird die Umsetzung des Frameworks dokumentiert. In diesem Unterabschnitt wird nun die Parallelisierung und die Performanz der in 2.1.5 eingeführten ZDD-spezifischen Methoden evaluiert.
Die ZDD-Umgebungen basieren auf demselben Komponentensystem wie die der BDDs. Es wird angenommen, dass sich die unterschiedlichen Implementierungen für ZDDs analog verhalten. Dementsprechend wurde das bereits erarbeitete Setup wiederverwendet.
Zur Analyse sollen die serielle sowie die parallele Version mit 12 Threads das N-Damenproblem mit und ohne dem Exclude-Operator lösen.
Die Graphen in Abbildung 5.10 zeigen die Ergebnisse der Durchläufe.

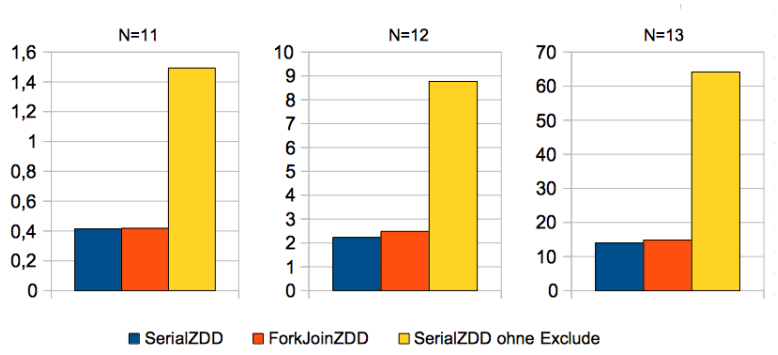


Abbildung 5.10: ZDD-Laufzeit des N-Damenproblems (Quelle: eigene Darstellung)

Die Evaluation belegt den positive Einfluss der ZDD-spezifischen Exclude-Operation. Die Laufzeitreduktion von über 70% in allen Testfällen, für $N = 13$ zum Beispiel von 64s auf

14s, rechtfertigt die zusätzliche Implementierung.
Außerdem zeigt sich, dass die Parallelisierung die analysierten Szenarien eher negativ beeinflusst. Deshalb wird zur Bewertung der nebenläufigen Algorithmen die benötigte Zeit zur Darstellung des 14-Damenproblem in Abbildung 5.11 untersucht.

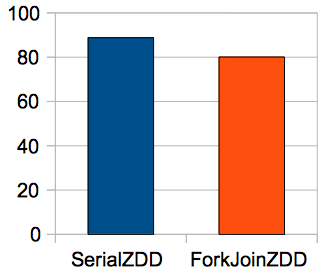


Abbildung 5.11: ZDD-Laufzeit des 14-Damenproblems (Quelle: eigene Darstellung)

Für diesen Anwendungsfall zeigt die parallele Version erste Verbesserungen, jedoch lediglich um 10% von 88s auf 80s. Der geringe Performanzertrag lässt sich anhand der ZDD-Struktur, der BDD-Ergebnisse und der Natur des Damenproblems erklären. Da das selbe Parallelisierungskonzept wie für BDDs angewandt wurde, kann gefolgert werden, dass hier die Vorteile der nebenläufigen Algorithmen ebenfalls an der Größe und insbesondere der Tiefe der Bäume festzumachen sind.
Für jede korrekte Stellung des 14-Damenproblems werden genau 14 Variablen mit 1 und die restlichen 182 mit 0 belegt. Da ZDDs mit 0 belegte Variablen nicht darstellen, bewirkt das, dass alle Pfade des Ergebnisses und somit die Rekursionstiefe genau 14 Level tief sind. Die Lösungswege im BDD erstrecken sich im Vergleich dazu über 196 Level.
Dass trotz der sehr geringen Baumtiefe ein leichter Performanzanstieg zu erkennen ist, lässt sich anhand der Breite von 365.596 Pfaden und der etwas tieferen Teilergebnisse der Berechnung erklären. Die Ergebnisse bestätigen also die in Abschnitt 5.2.1 getroffene Annahme, dass die Parallelisierung vor allem in tiefen Baumstrukturen effizient ist.
Außerdem zeigt die Analyse deutlich, dass ein direkter Vergleich der ZDD- und BDD-Umgebungen hinken würde, da die Datenstrukturen für unterschiedliche Anwendungsfälle ausgelegt sind. Um dies zu veranschaulichen, wurden die Lösungszeiten des N-Damenproblems in Tabelle 5.3 gegenübergestellt.

N	9	10	11	12	13	14
Lösungswege	352	724	2.680	14.200	73.712	365.596
BDD-Tiefe	81	100	121	144	169	196
ZDD-Tiefe	9	10	11	12	13	14
BDD-Zeit in s	0,16	0,69	3,4	19,1	-	-
ZDD-Zeit in s	0,02	0,08	0,4	2,2	14	80

Tabelle 5.3: Lösung des N-Damenproblems mit BDDs und ZDDs
(Quelle: eigene Darstellung)

Die tabellarische Gegenüberstellung belegt, dass ZDDs für Anwendungsfälle mit im Verhältnis zur Größe kleiner Lösungsmenge eine sehr interessante BDD-Alternative bieten.

Mit dieser Evaluation wurde die letzte PJBDD-Komponente ausgewertet. Im nächsten Unterpunkt wird das im Rahmen dieser Arbeit entwickelte Framework mit bereits existierenden BDD-Implementierungen konfrontiert.

5.2.2 Vergleich mit anderen Bibliotheken

In der zweiten Testphase wurde PJBDD den aktuellen C/C++-Standardbibliotheken BuDDy und CuDD sowie den Java-nativen JavaBDD, JDD und BeeDeeDee gegenübergestellt. Mit Sylvan fehlt dabei eine parallele BDD-Bibliothek, da diese aktuell nicht lauffähig scheint.
Für die beiden C/C++ Bibliotheken wurden analoge C und C++ Implementierungen des N-Damenproblems erstellt und für die Java-Frameworks eine Schnittstelle, die es ermöglicht die PJBDD Umsetzung zu verwenden. Zum Vergleich werden zwei PJBDD-Kombinationen getestet, die serielle int-basierte Version und die evaluierte nebenläufig-objektorientierte Umgebung mit 12 Threads. Alle Bibliotheken nutzen eine Cachegröße von 10.000 und $N * N$ initiale Variablen.
Da die Konfiguration der initialen Uniquetableparameter die Frameworks unterschiedlich stark beeinflusst, werden zunächst die besten Einstellungen evaluiert.
Dafür wurde das 12-Damenproblem mit unterschiedlichen Startgrößen gelöst. Abbildung 5.12 visualisiert die Benchmarkergebnisse abhängig von der anfänglichen Größe der Knotentabelle.

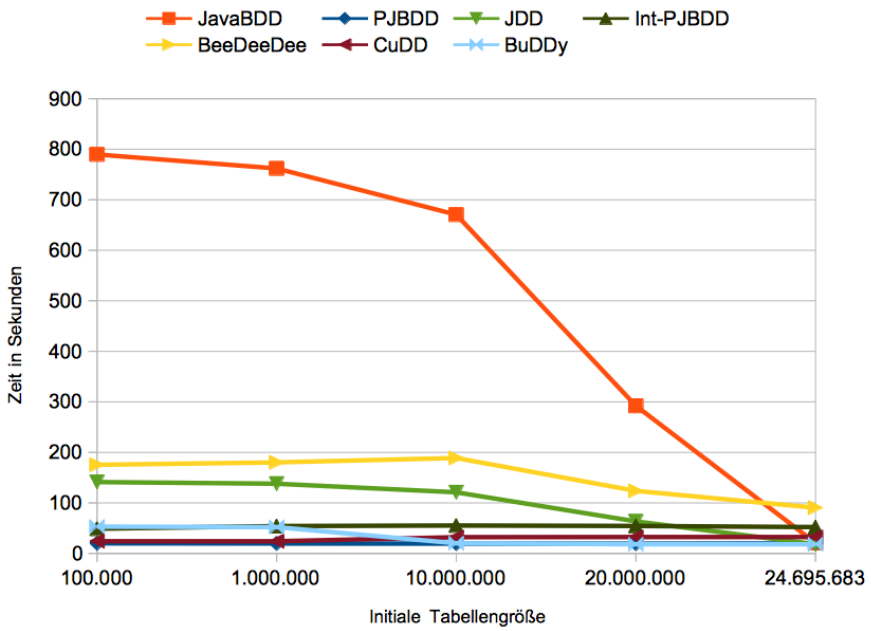


Abbildung 5.12: Lösungszeit für das 12-Damenproblem in Abhängigkeit der initialen Tabellengröße mit Standardkonfiguration (Quelle: eigene Darstellung)

Die Ergebnisse zeigen einen Vorteil von PJBDD, denn schlecht bzw. zu klein gewählte Größen verursachen bei BuDDy, JavaBDD, JDD und BeeDeeDee gravierende Perfor-

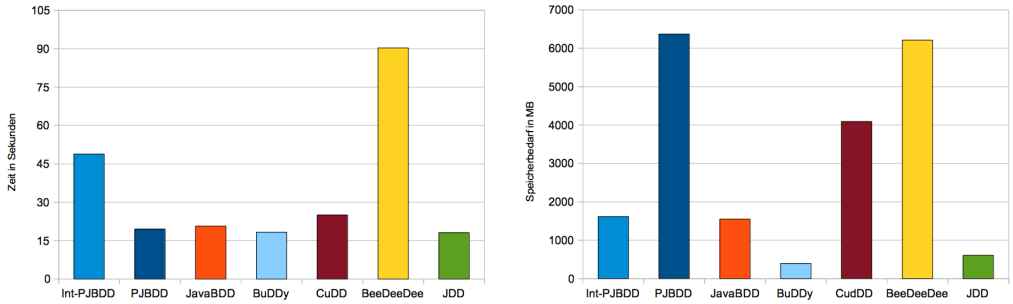
manzverluste. Wählt man zum Beispiel einen anfänglichen Wert von 100.000 benötigt JavaBDD 789s zur Lösung, PJBDD mit 12 Threads jedoch lediglich 19,9s. Selbst mit 20.000.000 initialen Tabellenplätzen rechnet JavaBDD noch 292s, PJBDD 19,1s.
Erst ab einer Größe von 24.695.683 erreicht JavaBDD ebenfalls die 20s Marke. Dieser Wert wurde anhand der Debug-Meldungen von JavaBDD ermittelt, indem der Wert der letzten Tabellenvergrößerung als Startwert verwendet wurde.
Das Problem beruht auf einer statischen Beschränkung der maximalen Vergrößerung. Denn BuDDy, JavaBDD, JDD sowie BeeDeeDee vergrößern die Uniquetable standardmäßig mit konstanten Summanden.
JavaBDD erhöht die Größe je um 50.000. Dadurch erfordert beispielsweise die Ausdehnung von 10.000.000 auf 20.000.000 Plätzen 200 Vergrößerungsoperationen. Im Vergleich dazu benötigt PJBDD für den gleichen Schritt nur eine Operation, da ein Wachstumsfaktor genutzt wird, der die Tabelle standardmäßig verdoppelt.
BuDDy und JavaBDD ermöglichen ebenfalls die Nutzung von Wachstumsfaktoren, indem die Ausdehnungsbeschränkung auf 0 gesetzt wird. Für BeeDeeDee muss explizit ein Faktor definiert werden. JDD hingegen bietet keine Möglichkeit eine konstante Beschränkung zu umgehen.
Der tatsächliche Vergleich der Bibliotheken wird nun basierend auf dieser Analyse mit den jeweils besten Einstellungen durchgeführt. Die Einstellung der initialen Uniquetablegröße kann der Datentabelle 5.4 entnommen werden.

Bibliothek	100.000	1.000.000	10.000.000	20.000.000	24.695.683
PJBDD	19,9	19,5	19,4	19,1	19,6
int-PJBDD	48,8	54,1	55,2	54,4	52,0
JavaBDD	789,6	761,8	670,5	292,1	20,7
JDD	141,2	137,9	120,9	63,4	18,1
BeeDeeDee	175,3	180,1	188,8	123,7	90,4
CuDD	24,1	24	32,1	32,4	32,4
BuDDy	53	52	20	18	18

Tabelle 5.4: Vergleich der BDD-Bibliotheken in Abhängigkeit der initialen Tabellengröße (Quelle: eigene Darstellung)

Gemessen wurde die benötigte Laufzeit und der Spitzenbedarf an Arbeitsspeicher.
Der Speicherbedarf der Java-Bibliotheken wurde mit Hilfe Jens Wilkes Artikel[2] über Arbeitsspeicherbenchmarks in Java ermittelt. Dafür wurde den JMH-Benchmarkdurchläufen ein Profiler angefügt, der die Linux Statistik */proc/[pid]/status* ausließt. Dieser kann neben der Spitzen-RSS auch der ausgelagerte Speicher entnommen werden.
Mit diesen zwei Werten ist eine Rekonstruktion des tatsächlichen Spitzenspeicherbedarfs der Anwendungsfälle möglich. Denn da kein Speicher ausgelagert wurde, kann der RSS verwendet werden. Dazu ist anzumerken, dass diese Rekonstruktion unmöglich wäre, wenn eine Arbeitsspeicherauslagerung erfasst wird. Für die C/C++ Frameworks wurden mit dem Befehl *time -v [command]* dieselben Werte erfasst und analog verarbeitet.
Die beiden Diagramme in 5.13 zeigen die benötige Laufzeit und den Arbeitsspeicherbedarf zur Lösung des 12-Damenproblems auf 12 Kernen.

[2] https://cruftex.net/2017/03/28/The-6-Memory-Metrics-You-Should-Track-in-Your-Java-Benchmarks.html Abrufdatum: 29.04.19



(a) Laufzeitanalyse (b) Analyse des Speicherbedarfs

Abbildung 5.13: Vergleich der BDD-Bibliotheken anhand des 12-Damenproblems (Quelle: eigene Darstellung)

Die Laufzeitanalyse zeigt, dass PJBDD in sequentieller Umgebung für große Szenarien circa um den Faktor zwei langsamer ist, mit Ausnahme von BeeDeeDee. Durch die Parallelisierung kann dieser Nachteil jedoch ausgeglichen werden. Mit 19,1s liegt PJBDD nur knapp hinter BuDDy(18s) und JDD(18,1s) und sogar etwas vor JavaBDD(20,7s) und CuDD(24,s).
Die Speicheranalyse zeigt die Schwäche der objektorientierten PJBDD-Umgebungen deutlich, denn der Arbeitsspeicherbedarf liegt mit 6,3GB weit über dem von JavaBDD(1,5GB), JDD(0,6GB), BuDDy(0,4GB) und der int-basierten PJBDD-Lösung(1,6GB).
Dieser erhöhte Bedarf ist allen voran dem Speicherplatzbedarf von Java-Objekten geschuldet. Denn besteht ein BDD in JavaBDD und den int-basierten Lösungen aus 5 int Werten einer Tabelle ($mem_{intBDD} = 160bit$), so bedarf das umgesetzte BDD-Objekt auf einer 64-bit Maschine 128 für das Objekt an sich, zweimal 32-64 bit für die Kindknoten-Referenzen und zweimal 32 bit für die int-Variable und den hashcode $mem_{BDD_{min}} = 276bit$ und $mem_{BDD_{max}} = 320bit$. Hinzu kommt der Overhead von der Parallelisierung und den WeakReferences.

Ein großer Vorteil von PJBDD ist die Threadsicherheit. Dadurch kann die Umgebung nicht nur einzelne Berechnungen nebenläufig durchführen, sondern auch mehrere Anfragen parallel bearbeiten. Um den Effekt dieser Parallelisierung zu simulieren, zeigt Abbildung 5.14 die Laufzeiten zur 8-maligen Lösung des 12-Damenproblems mit 12 Threads.

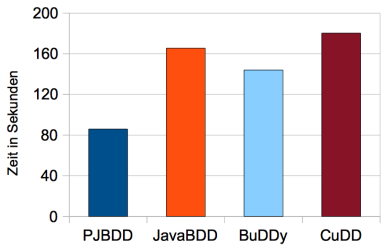


Abbildung 5.14: Auswirkung paralleler Abarbeitung mit 12 Kernen (Quelle: eigene Darstellung)

Es zeigt sich, dass PJBDD die Performanz trotz bereits nebenläufiger Algorithmen noch-

mal um circa 50% gegenüber den anderen Bibliotheken steigern kann. Hier ist zu erwähnen, dass BeeDeeDee in dieser Statistik fehlt, da Java-Exceptions bei parallelen Anfragen aufgetreten sind.

Ein weiterer Vorteil, der in diesen Analysen nicht abgebildet wird, weil er lediglich die Anwendung betrifft ist, dass PJBDD eine automatisierte Speicherfreigabe bietet, wohingegen die Alternativen eine manuelle erfordern. Darüber hinaus ermöglicht PJBDD mittels implementiertem Komponentensystem und Dependency Injection ein einfaches Anpassen einzelner Implementierungsdetails an andere Programme.

Damit ist die Evaluation im Vergleich mit anderen Bibliotheken abgeschlossen. Folgende Tabelle fasst abschließend die erarbeiteten Vor- und Nachteile zusammen, bevor im nächsten Abschnitt der Einsatz in CPAchecker beurteilt wird.

Vorteile	Nachteile
-Threadsicherheit -Automatisierte Speicherverwaltung -Performanzpotential auf Multicore-Rechnern -falsche Konfiguration bietet kein Bottleneck -Dependecy Injection -einfach erweiterbar	-erhöhter Speicherbedarf -benötigt Multithreatingumgebung für konkurrenzfähige Laufzeit in großen Anwendungsfällen

Tabelle 5.5: Vor- und Nachteile von PJBDD

5.3 Integration in CPAchecker

CPAchecker ist ein Framework zur konfigurierbaren Softwareanalyse und Verifikation. Dabei hat CPAchecker den Anspruch die Integration neuer Analysekomponenten einfach zu ermöglichen. Das Grundprinzips des Frameworks ist die Implementierung des CPA-Algorithmus. Der Algorithmus nutzt eine konfigurierbare Programmanalyse (Configurable-Program-Analysis (CPA)), um einen Erreichbarkeitsgraphen zu erstellen, der alle potentiellen Zustände des zu analysierenden Programms darstellt. Kann ein Fehlerzustand erreicht werden, so gilt dieses als fehlerhaft und ansonsten als fehlerfrei. In (BK11) beschreiben Beyer et al. dieses Framework ausführlich.
Für bestimmte Analysen bieten BDDs eine effiziente Repräsentation dieser abstrakten Programmzustände. Deshalb stellen Beyer et al. in (BS12) einen hybriden Ansatz für CPAchecker vor, der es ermöglicht bei der Verifikation nur ausgewählte Variablen mit BDDs zu behandeln. Mit einer Preanalyse wird bestimmt, für welche Variablen eine BDD-basierte oder Explizite-Analyse verwendet werden soll.

Aktuell unterstützt CPAchecker die BDD-Bibliothek JavaBDD, welche wiederum die Nutzung weiterer Frameworks ermöglicht. Teil dieser Arbeit ist es, PJBDD in das Programm CPAchecker zu integrieren und für BDD-basierte Analysen nutzbar zu machen.
Dafür wurde PJBDD analog zu den anderen Bibliotheken mit der Klasse ParallelBDDRegionManager eingebunden, mit dem Unterschied, dass die Schnittstellenmethoden nicht synchronisiert werden und keine Maßnahmen zur Ressourcenfreigabe ergriffen werden müssen. Zudem sind sämtliche Framework-Konfigurationen über das Terminal spezifi-

zierbar.
Eine detaillierte Auflistung der möglichen Konfigurationen ist der CPAchecker PJBDD-Package Dokumentation zu entnehmen.
Nach der Einbindung der im Rahmen dieser Arbeit implementierten Bibliothek wird im nächsten Abschnitt eine Evaluation vorgestellt, für die CPAchecker eingesetzt wurde, um BDD-basierte Analysen mit JavaBDD und PJBDD zu vergleichen.

5.4 Evaluation mit CPAchecker

CPAchecker bietet unterschiedliche Analysetypen zur Softwareverifikation, die unter anderem auf BDDs oder SMTSolver basieren. Zur Bewertung der Integration von PJBDD werden in diesem Unterkapitel zwei der BDD-basierten Typen evaluiert. Dafür wurde die VerifierCloud[3] des SoSy-Lab Lehrstuhls benutzt, um mit unterschiedlichen CPAchecker-Konfigurationen jeweils mehrere tausend Beispielprogramme aus dem Github-Repository sv-benchmarks[4] zu verifizieren. Mit sv-benchmarks stellt SoSy-Lab eine hervorragende Sammlung an Beispielprogrammen für Verifikationstasks zur Verfügung, die von vielen Forschungsgruppen verwendet und regelmäßig erweitert wird.
Das Rechencluster VerifierCloud verfügt über 250 Maschinen, die für die Verifikation von Programmen verwendet werden können. Das Projekt nimmt beliebig viele Verifikationsaufgaben entgegen, denen es mit Hilfe eines Job-Schedulers Rechenressourcen zuteilt. Bei der Taskeingabe können Ressourcenanforderungen, Zeitlimits und CPU-Modelle spezifiziert werden, die der Scheduler bei der Zuweisung berücksichtigt.
Zum gesammelten Benchmarking der einzelnen Verifikationsdurchläufe bietet CPAchecker eine BenchExec[5] Integration. Mit diesem Benchmark-Tool lassen sich nicht nur Ergebnis, CPU-Zeit, Laufzeit, Arbeitsspeicher- und Energiebedarf messen, sondern auch eine Sammlung an Tasks definieren, die direkt an das Cluster übergeben werden kann.
In dem Paper (BLW17), mit dem Beyer et al. die Grundregeln für aussagekräftige Benchmarks definieren, wird BenchExec als zuverlässige Lösung für Linux vorgestellt und dargestellt wie die Bibliothek alle Anforderungen erfüllt. Das Framework nutzt beispielsweise das Cgroups-Feature zur kontrollierten Ressourcenzuweisung sowie für exakte Messungen. Außerdem verwendet BenchExec zur Isolation für jeden Durchlauf einen Container mit neuem Namensraum. Für detailliertere Ausführungen wird an (BLW17) verwiesen.
Für die erste Benchmark-Sammlung werden 3540 C-Programme aus acht Unterordnern des sv-benchmark-Repository ausgewählt. Anschließend werden Testläufe für JavaBDD, PJBDD mit einem, mit vier und mit acht Threads gestartet, um die Programme mit der reinen BDD-Analyse von CPAchecker zu prüfen. Für die Synthese jedes Programms wurde ein Job mit den Beschränkungen: Intel Xeon E3-1230, 8 CPU-Kernen, 20GB Arbeitsspeicher, 16GB maximale Heap-Größe und 300s CPU-Zeit als Timeout an die VerifierCloud übergeben.
Außerdem wurde jede Analyse 10-fach ausgeführt und die Ergebnisse aus den Mittelwerten der Messungen gebildet. In die Ergebnismenge wurden nur die Programme aufgenom-

[3] https://vcloud.sosy-lab.org/
Abrufdatum 04.05.19
[4] https://github.com/sosy-lab/sv-benchmarks
Abrufdatum: 02.05.2019
[5] https://github.com/sosy-lab/benchexec/blob/master/doc/benchexec.md
Abrufdatum: 02.05.2019

men, die erfolgreich von JavaBDD verifiziert werden konnten. Dazu ist anzumerken, dass PJBDD alle mit JavaBDD gelösten Eingaben ebenfalls synthetisieren konnte. Die drei Diagramme in 5.15 zeigen die insgesamt benötigte CPU-Zeit, Laufzeit und Arbeitsspeicher für die 2782 verbleibenden Programme, die validiert werden konnten.

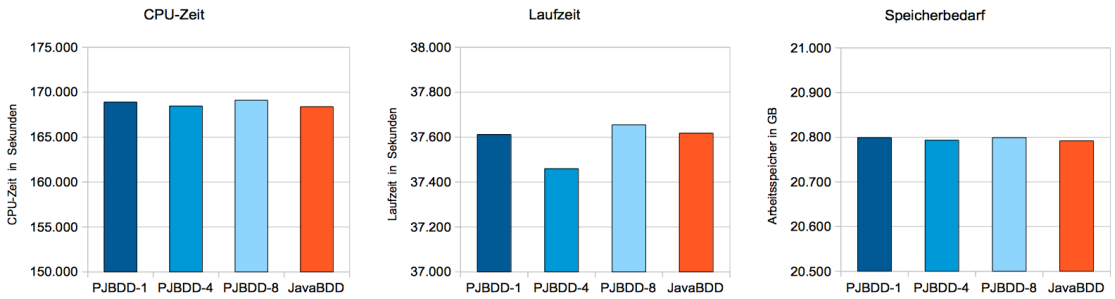


Abbildung 5.15: Ergebnisse der Verifikation von 2782 Programmen mit der reinen BDD-Analyse durch CPAchecker (Quelle: eigene Darstellung)

Die Graphen zeigen, dass die Performanz der Bibliotheken extrem dicht beieinanderliegt. Die geringen Ausschläge sind vor allem wegen der sehr großen Minimalwerte und vergleichsweise kleinen Skalierung sichtbar. Betrachtet man beispielsweise den Laufzeitgraphen so ist zwar zu erkennen, dass PJBDD 158s schneller war als JavaBDD, dies entspricht jedoch lediglich 0,4% der 37617s Gesamtlaufzeit, wodurch die Differenz nahezu keine Aussagekraft hat. Dasselbe gilt für die weiteren Metriken.

Aus dieser Analyse wurde in Abbildung 5.16 ein weiteres Diagramm gebildet. Es beschreibt die Mittelwerte der Abweichungen von den PJBDD- zu den JavaBDD-Synthesen der einzelnen Programme in %.

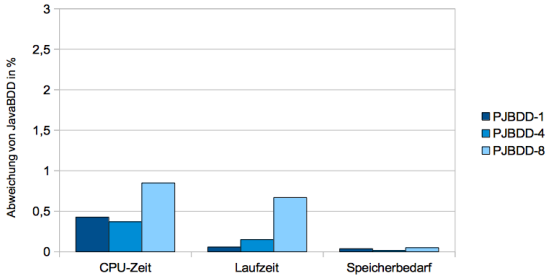


Abbildung 5.16: Durchschnittliche Abweichung von JavaBDD bei der Verifikation mit reiner BDD-Analyse von 2782 Programmen durch CPAchecker (Quelle: eigene Darstellung)

Die Grafik bestätigt die extrem geringen Unterschiede von weniger als 1% der alternativen Durchläufe. Vergleicht man die absoluten Laufzeitwerte von PJBDD mit 4 Threads und JavaBDD, so ist JavaBDD bei der Verifikation von 1373 Programmen um durchschnittlich 0,43s (13,27s auf 13,70s) schneller und PJBDD bei der von 1409 um 0,53s (13,24s auf 13,77).

Diese Evaluation erlaubt den Schluss, dass PJBDD trotz Objektorientierung Anfragen in der Größenordnung der BDD-Analyse genauso performant bearbeitet wie JavaBDD.

Wie bereits erwähnt, bietet CPAchecker hybride Ansätze die einen Teil der zu synthetisierenden Variablen mit BDDs kodieren. Dementsprechend wird als Nächstes der Einsatz von PJBDD in der Value-Analyse die boolesche Variablen, int-Equals- und int-Add-Variablen mit BDDs verifiziert. Int-Equals-Variablen bezeichnen diskrete int Variablen, die in booleschen Vergleichen verwendet werden. Für die Analyse wird jede mögliche Belegung mit einer BDD-Variable kodiert. Als Int-Add werden Variablen aus Rechenoperationen mit linearer Arithmetik oder aus bitweisen Operationen klassifiziert. Variablen dieser Klasse werden als 32-Bitvektoren dargestellt, wobei eine BDD-Variable je ein Bit repräsentiert. (ABF+13)
Ansonsten wurde das Setup aus vorangehender Untersuchung übernommen.
Die drei Diagramme in Abbildung 5.17 zeigen die Gesamtergebnisse der CPU-Zeit, Laufzeit und Speicherbedarf für die Verifizierung von 1836 Programmen.

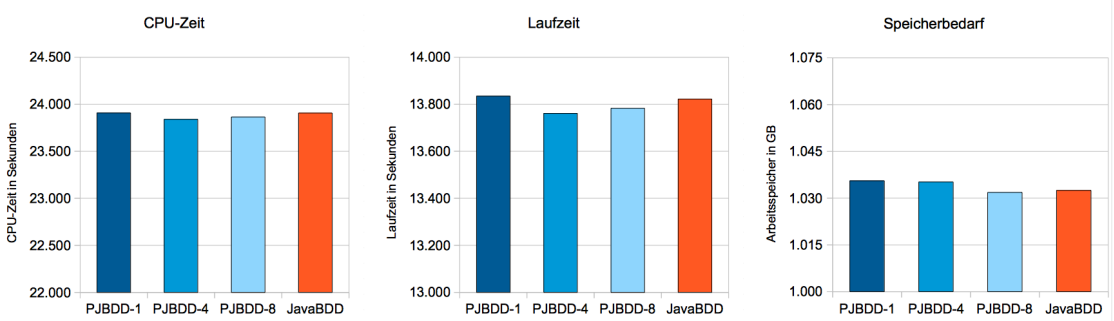


Abbildung 5.17: Ergebnisse der Verifikation von 1836 Programmen mit der Value-Analyse Bool-IntEq-IntAdd durch CPAchecker (Quelle: eigene Darstellung)

Der Vergleich liefert identische Ergebnisse wie die reine BDD-Analyse: Die gemessenen Werte für die unterschiedlichen Metriken unterscheiden sich um weniger als 0,5%. Auch die Mittelwerte der Abweichungen einzelner Programme von PJBDD zu JavaBDD in Darstellung 5.18 bestätigen die Untersuchung.

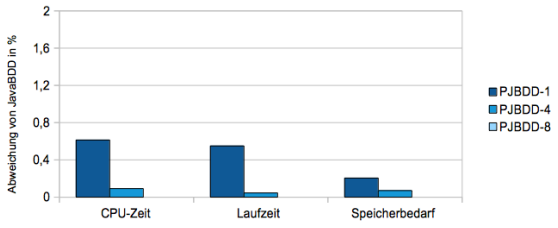


Abbildung 5.18: Durchschnittliche Abweichung von JavaBDD bei der Verifikation mit der Value-Analyse Bool-IntEq-IntAdd von 1836 Programmen durch CPAchecker (Quelle: eigene Darstellung)

Die durchschnittlichen Werte weichen um weniger als 1% von JavaBDD ab.

Diese Evaluation zeigt, dass die BDD-Anfragen der CPA-Analysen zu klein sind, um messbare Unterschiede in Abhängigkeit von den Bibliotheken festzustellen.
Vergleicht man die Einbindung in CPAchecker, zeigen sich die Vorteile von PJBDD. Bei der Integration müssen keine Interface-Funktionen synchronisiert werden, da die Bibliothek bereits threadsicher ist. Vielmehr werden parallele Anfragen unterstützt, die in zukünftig nebenläufigen CPAchecker-Analysen Anwendung finden können.
Außerdem benötigt JavaBDD eine manuelle Ressourcenfreigabe. Die CPAchecker-Schnittstelle nutzt dafür PhantomReferences[6], die auf den BDD-Wrapper-Objekten liegen, um vor jedem Methodenaufruf alle nicht mehr benötigten Objekte freizugeben. PJBDD erledigt das bereits intern, indem nicht mehr verwendete Referenzen im Hintergrund von einem Daemon-Thread aufgeräumt werden und erfordert dadurch diesbezüglich keine weiteren Maßnahmen.

Aus letzteren und vorhergehenden Analysen geht hervor, dass PJBDD Operationen auf vergleichsweise kleinen BDDs genauso performant löst, wie aktuelle Standardbibliotheken und auch für große Operationen dank paralleler Algorithmen konkurrenzfähig ist. Im Vergleich liegt zudem die Performanz der einzig weiteren evaluierten threadsicheren BDD-Bibliothek (BeeDeeDee siehe 5.2.2) weit hinter PJBDD und den hoch-optimierten seriellen Bibliotheken. Darüber hinaus kann durch Threadsicherheit und der nebenläufigen Bearbeitung paralleler Anfragen in entsprechenden Anwendungsfällen ein deutlicher Laufzeitvorteil gewonnen werden.

[6] PhantomReferences verhalten sich ähnlich zu WeakReferences, mit dem Unterschied, dass das referenzierte Objekt erst gelöscht wird, wenn die PhantomReference selbst entfernt wird. Wird eine PhantomReference in die ReferenceQueue eingereiht, ist dennoch kein Zugriff auf das Objekt mehr möglich.

6 Fazit

Die implementierte BDD-Bibliothek PJBDD bietet für viele Anwendungsfälle gute Einsatzmöglichkeiten. Durch Nutzung neuster Java-Konzepte können Schwachstellen anderer Bibliotheken beseitigt werden. So bereinigt PJBDD nicht mehr verwendete Knoten automatisch, wohingegen andere Frameworks eine manuelle Dereferenzierung vom Anwender erfordern, um einen Speicherüberlauf vorzubeugen. Außerdem kann die Implementierung vorteilhaft sein, wenn die Größe der Knotentabelle anfangs nicht genau abzuschätzen ist. Eine zu kleine Tabelle mit der Standardkonfiguration kann in anderen Bibliotheken zu enormen Laufzeitverlusten führen, die PJBDD Laufzeit hingegen ist von dem Startparameter so gut wie unbeeinflusst.
Dem Overhead der Threadsicherheit und den vorher genannten Vorzügen geschuldet, kann die Realisierung mit optimal gewählten Parametern im sequentiellen Kontext zu Performanzverlusten im Vergleich zur Konkurrenz führen. Auf einer mehrkernigen Umgebung optimiert PJBDD allerdings mittels Parallelisierung der BDD-Operationen die Laufzeit und kann dank nebenläufiger Abarbeitung in bestimmten Anwendungen sogar eine schnellere Lösung bieten.
Durch das umgesetzte Komponentensystem konnten mehrere alternative Java-Konzepte miteinander verglichen und die bestmögliche Kombination als Bibliotheken-Standard festgelegt werden. Die Verwendung von Programmierparadigmen wie Dependency Injection oder das Builder Pattern ermöglichen dem Anwender zusätzlich einige Möglichkeiten zu nutzerspezifischen Anpassungen unterschiedlicher Komponenten und eine einfache sowie beliebig granulare Konfiguration.

6.1 Ausblick

In dieser Arbeit wurde eine BDD-Bibliothek entworfen, die die grundlegenden Operationen sowie die Möglichkeit zur manuellen Änderung der Variablensortierung und ein einfaches ZDD-Framework unterstützt. Zukünftigen Arbeiten bleiben noch einige potentielle Optimierungsmöglichkeiten überlassen.
So bietet die Implementierung einer dynamischen Adaption der Variablenordnung zum Beispiel noch Performanzpotential. Darüber hinaus könnte die Realisierung eines threadsicheren Workstacks für weitere Laufzeiteinsparungen sorgen, indem beispielsweise jeder Thread einen eigenen Workstack verwendet. Die Einführung komplementärer BDD-Kanten kann zudem den Speicherbedarf für manche Anwendungsfälle reduzieren. Dafür könnte ein Knoten die Kanten auf seine Kindknoten als Indikator markieren, so dass deren Kanten vertauscht genutzt werden, um Knoten mit komplementären Kindern ebenfalls zu reduzieren.
Die verwendeten Java-nativen Parallelisierungskonzepte bieten für die int-basierte Umsetzung keine brauchbare Lösung, trotz der sequentiellen Vorteile können sie mit den objektorientierten nicht mithalten. Die Implementierung eines Workerthreadpools, der

zum Beispiel das Integer-Autoboxing vermeidet, bietet möglicherweise eine Option das durchaus interessante Konzept konkurrenzfähig zu machen.
Die Analysen haben gezeigt, dass die Parallelisierung kleiner Anfragen keinen Vorteil bietet. Mit Hilfe der ParallelismManager-Schicht könnte ein sinnvoller Mechanismus integriert werden, der verhindert, dass Operationen mit zu geringer Rekursionstiefe nebenläufig berechnet werden.
Ein Nebenprodukt dieser Arbeit ist eine ZDD-Bibliothek, die die meisten spezifischen Operationen unterstützt. Es wurde darüber hinaus gezeigt, dass ZDDs für viele Anwendungsfälle eine hochperformante Datenstruktur sind, da sie für viele boolesche Funktionen die kompaktere Darstellung bieten. Im nächsten Schritt könnte PJBDD die alternative Verwendung von ZDDs unterstützen. Dafür kann beispielsweise mit Hilfe des Grundlagenkapitels 2.1.4 eine BDD-Abstraktionsschicht gebaut werden, die intern auf ZDDs basiert.

Da die meisten BDD-Frameworks kein Multithreating unterstützen, nutzen viele Programme wie CPAchecker aktuell lediglich sequentielle Anfragen an die BDD-Umgebung. Um das volle Potential der threadsicheren Bibliothek auszuschöpfen, bleibt es zukünftigen Versionen überlassen asynchrone Anfragen zu implementieren.

Im Rahmen dieser Masterarbeit wurde mit Hilfe neuster Konzepte eine nutzerfreundliche und objektorientierte BDD-Bibliothek in Java entwickelt, die auf moderner Hardware selbst mit hochoptimierten C/C++-Realisierungen konkurrieren kann.

Literaturverzeichnis

[ABF+13] Apel, Sven ; Beyer, Dirk ; Friedberger, Karlheinz ; Raimondi, Franco ; Rhein, Alexander von: Domain types: Selecting abstractions based on variable usage. In: *arXiv preprint arXiv:1305.6640* (2013)

[Ake78] Akers, Sheldon B.: Binary decision diagrams. In: *IEEE Transactions on computers* (1978), Nr. 6, S. 509–516

[And97] Andersen, Henrik R.: An introduction to binary decision diagrams. In: *Lecture notes, available online, IT University of Copenhagen* (1997), S. 5

[BK11] Beyer, Dirk ; Keremoglu, M E.: CPAchecker: A tool for configurable software verification. In: *International Conference on Computer Aided Verification* Springer, 2011, S. 184–190

[BLW17] Beyer, Dirk ; Löwe, Stefan ; Wendler, Philipp: Reliable benchmarking: Requirements and solutions. In: *International Journal on Software Tools for Technology Transfer* (2017), S. 1–29

[BRB90] Brace, Karl S. ; Rudell, Richard L. ; Bryant, Randal E.: Efficient implementation of a BDD package. In: *27th ACM/IEEE design automation conference* IEEE, 1990, S. 40–45

[BS12] Beyer, Dirk ; Stahlbauer, Andreas: BDD-based software model checking with CPAchecker. In: *International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science* Springer, 2012, S. 1–11

[Knu09] Knuth, Donald E.: *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams, volume 4, fascicle 1.* 2009

[Lee59] Lee, Chang-Yeong: Representation of switching circuits by binary-decision programs. In: *The Bell System Technical Journal* 38 (1959), Nr. 4, S. 985–999

[Min93] Minato, Shin-ichi: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: *30th ACM/IEEE Design Automation Conference* IEEE, 1993, S. 272–277

[Min94] Minato, Shin-ichi: Calculation of unate cube set algebra using zero-suppressed BDDs. In: *31st Design Automation Conference* IEEE, 1994, S. 420–424

[Mis01] Mishchenko, Alan: An introduction to zero-suppressed binary decision diagrams. In: *URL: http://www. ee. pdx. edu/alanmi/research. htm* (2001)

[OMI98] Okuno, Hiroshi G. ; Minato, Shin-ichi ; Isozaki, Hideki: On the properties of combination set operations. In: *Information Processing Letters* 66 (1998), Nr. 4, S. 195–199

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.05.2019

. . . . . .
(Unterschrift des Kandidaten)