Karlsruher Institut für Technologie

Fakultät für **Informatik**

Generierung paralleler Architekturbeschreibungen durch kombinierte statische und dynamische Analysen

Diplomarbeit
von

Jochen Huck

Verantwortlicher Betreuer: Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter: Dipl.-Inform. Korbinian Molitorisz

Bearbeitungszeit: 15. Dezember 2012 – 27. Mai 2013

Karlsruher Institut für Technologie · Institut für Programmstrukturen und Datenorganisation · Geb. 50.34 · Am Fasanengarten 5 · 76131 Karlsruhe

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 27. Mai 2013

Jochen Huck

Inhaltsverzeichnis

1 EINLEITUNG

„The Free Lunch Is Over“ [S05]: Die Anzahl unabhängiger Rechenkerne neuer Prozessorgenerationen steigt stetig, weil die Strukturgrößen von Transistoren dem mooreschen Gesetz [M65] folgend kontinuierlich verkleinert werden können. Seit 2004 nehmen die Taktraten der einzelnen Kerne im Gegensatz zu früher jedoch nicht mehr nennenswert zu. Um die Leistung bestehender Anwendungen dennoch steigern zu können, müssen diese parallelisiert werden.

Die Parallelisierung sequentieller Anwendungen erweist sich jedoch als aufwändig und fehleranfällig [B09]: parallelisierbare Programmteile müssen zunächst durch eine Inspektion der Anwendung identifiziert werden. Dazu ist ein tiefgreifendes Verständnis des zugrundeliegenden Quellcodes nötig. Zudem muss abgeschätzt werden, ob die parallele Ausführung dieser Programmteile tatsächlich zu Laufzeiteinsparungen führt. Selbst mit Kontextwissen und entsprechenden Laufzeitmessungen ist diese Frage nicht leicht zu beantworten. Anschließend müssen die identifizierten Programmteile durch einen Umbau der Anwendung auf eine parallele Architektur abgebildet werden. Im Laufe dieses Prozesses können sich Parallelitätsfehler wie Reihenfolgeverletzungen, Datenwettläufe oder Verklemmungen einschleichen. Aufgrund des Indeterminismus der parallelen Anwendung sind diese Fehler nur schlecht reproduzierbar und schwer aufzufinden. Deshalb muss die Anwendung nach der Parallelisierung auf diese Art von Fehlern getestet werden. Schließlich muss die parallele Anwendung auf die Zielplattform angepasst werden, indem optimale Werte parallelitätsbezogener Parameter gesetzt werden. Im Hinblick auf große Mengen zu parallelisierender Bestandssoftware stellt sich die Frage, ob und wie ein Entwickler bei dem vorgestellten Parallelisierungsprozess systematisch unterstützt werden kann, um Aufwand und Fehleranfälligkeit zu reduzieren.

Ziel dieser Diplomarbeit ist ein Verfahren, welches die automatisierte Identifizierung parallel ausführbarer Programmteile ermöglicht. Vorhandenes Parallelisierungspotential sequentieller Anwendungen soll erkannt und durch die Angabe von Parallelisierungsvorschlägen in Form von Architekturbeschreibungen explizit gemacht werden. Das Verfahren ist ein erster Bestandteil einer Werkzeugkette, welche die automatische Parallelisierung von Bestandssoftware anstrebt: ein sich anschließender Übersetzer generiert aus den erzeugten Parallelisierungsvorschlägen optimierbaren parallelen Quellcode [W13]. Durch automatisch generierte Testfälle [SM+13] und einen zugehörigen Wettlauferkenner wird der generierte Code auf Korrektheit überprüft. Im Anschluss daran passt ein Auto-Tuner parallelitätsbezogene Parameter automatisch auf die Zielplattform an, um optimale Leistungswerte zu erzielen.

In dieser Arbeit wird ein Verfahren vorgestellt, welches zur Erreichung des geschilderten Ziels entwickelt und durch eine prototypische Implementierung umgesetzt wurde. In der Evaluierung konnte gezeigt werden, dass für die sechs verwendeten Fallbeispiele vorhandene Parallelisierungsmöglichkeiten gefunden, und nach der Umsetzung der ausgegebenen Parallelisierungsvorschläge Beschleunigungen von 1,2 bis zu 5,3 auf einem Achtkernsystem erzielt werden können. Das Verfahren schränkt den Suchraum nach Parallelisierungsmöglichkeiten um 95% ein. Dabei befinden sich alle relevanten Parallelisierungsmöglichkeiten in den ausgegebenen Parallelisierungsvorschlägen, was eine Trefferquote von 100% ergibt. Die Präzision von 66% ist noch nicht optimal, weil zu viele der ausgegebenen Vorschläge nicht zu einer Beschleunigung führen. Durch die Erhebung und Berücksichtigung von Laufzeitdaten ist hier jedoch eine erhebliche Verbesserung möglich.

2 GRUNDLAGEN

In diesem Kapitel wird auf grundlegende Sachverhalte eingegangen, die dem Verständnis darauffolgender Kapitel dienlich sind. Zunächst werden verschiedene Arten von Parallelität (Kapitel 2.1) vorgestellt, die sowohl von verwandten Arbeiten, als auch von dieser Arbeit adressiert werden. Im gleichen Kapitel wird die Verbindung zwischen parallelen Entwurfsmustern und den in dieser Arbeit definierten parallelen Architekturmustern aus Kapitel 4.5 erläutert. In Kapitel 2.2 wird im Anschluss anschaulich aber oberflächlich auf den Vorgang der Parallelisierung von Anwendungen eingegangen, für den im Verlauf dieser Arbeit ein Konzept entwickelt wird. Daraufhin wird auf das zentrale Thema der Programmanalyse (Kapitel 2.3) eingegangen und erläutert, welche verschiedenen Ansätze möglich sind um automatisiert und strukturiert Informationen über ein Programm zu gewinnen. Letztlich wird auf grundlegende technische Aspekte eingegangen, die für die Implementierung des Konzepts von Bedeutung sind (Kapitel 2.4).

2.1 Arten von Parallelität und parallele Entwurfsmuster

Parallelität kann auf verschiedenen Abstraktionsebenen umgesetzt werden, wie bereits verwandten Arbeiten [S10] [O13] zu entnehmen ist. Im Folgenden wird auf vier Arten von Parallelität eingegangen:

- **Befehlsebene:** Parallelität auf der Ebene von Maschinenbefehlen kann ausgenutzt werden, wenn zwei Befehle sich nicht beeinflussen (beispielsweise über gemeinsame Speicherzugriffe) und genügend Prozessorressourcen (beispielsweise arithmetisch-logische Einheiten) zur Verfügung stehen, um beide Befehle gleichzeitig auszuführen. Parallelität auf Befehlsebene wird von superskalaren Prozessoren vollständig transparent für den Anwendungsprogrammierer umgesetzt. Sie ist sehr feingranular und bildet damit die „niedrigste“ Form von Parallelität. Übersetzer können bestimmte Optimierungen durchführen, um Parallelität auf Befehlsebene zu begünstigen.
- **Daten- und Schleifenebene:** Parallelität auf der Ebene von Schleifen kann ausgenutzt werden, indem einzelne Iterationen einer Schleife parallel zueinander ausgeführt werden. Dazu werden Ausführungsfäden erzeugt, die einzelne Iterationen durchführen. Die Ausführungsfäden werden auf verfügbare Rechenkerne verteilt. Verschiedene Iterationen einer Schleife müssen dazu unabhängig sein. Diese Art von Parallelität wird auch als Datenparallelität bezeichnet, weil Schleifen oftmals über Elemente einer Datenstruktur iterieren. Die Parallelität kommt dann zustande, weil die gleiche Operation auf verschiedenen Daten parallel ausgeführt wird. Im Gegensatz zu Parallelität auf Befehlsebene ist diese Art von Parallelität meist nicht transparent für den Anwendungsprogrammierer. Es gibt zwar Übersetzer, die Schleifen automatisch parallelisieren, dies gelingt jedoch nur in sehr einfachen Fällen. Deshalb müssen Entwickler meist selbst dafür sorgen, dass Daten- beziehungsweise Schleifenparallelität ausgenutzt wird.
- **Funktions- bzw. Aufgabenebene:** Parallelität auf Funktionsebene kann ausgenutzt werden, indem einzelne Funktionen oder ganze Programmteile, die oft auch als Aufgaben bezeichnet werden, parallel zueinander ausgeführt werden. Die Aufgaben müssen dazu unabhängig sein. Wie die Daten- oder Schleifenparallelität ist sie nicht transparent für den Entwickler und muss daher explizit im Quellcode umgesetzt werden.

- **Fließbandebene:** Bei der hardwareseitigen Fließbandparallelität werden aufeinanderfolgende Maschinenbefehle (Elemente) durch verschiedene Prozessoreinheiten (Stufen) zur gleichen Zeit verarbeitet. Bei der allseits bekannten DLX-Pipeline der gleichnamigen Prozessorarchitektur [HP06] ist es beispielsweise möglich, einen Befehl zu dekodieren (engl. *instruction decode*) während der nächste Befehl bereits geladen wird (engl. *instruction fetch*). Wie die Parallelität auf Befehlsebene, wird diese Art von Parallelität transparent umgesetzt. Fließbandparallelität kann auch softwareseitig umgesetzt werden und bildet dann eine Mischform aus Daten- und Aufgabenparallelität. Sie lässt sich ausnutzen, wenn eine Folge von Verarbeitungsstufen (Aufgaben) von mehreren unabhängigen (Daten-)Elementen durchlaufen wird. Die Aufgaben gleichen dabei Teilen eines Schleifenrumpfs der Schleife, die für die Datenelemente verantwortlich ist. Wie bei Daten- und Aufgabenparallelität muss der Entwickler softwareseitige Fließbandparallelität selbst umsetzen.

Parallelität auf Befehlsebene und hardwareseitige Fließbandparallelität werden transparent durch den Prozessor umgesetzt und durch Optimierungen von Übersetzern begünstigt. Daten-, Aufgaben- und softwareseitige Fließbandparallelität müssen explizit vom Entwickler umgesetzt werden. Zur Umsetzung von Parallelität auf diesen Abstraktionsebenen gibt es etablierte Lösungen, die als parallele Entwurfsmuster (engl. *parallel design patterns*) bezeichnet werden. Zahlreiche parallele Entwurfsmuster und deren Einsatzszenarien wurden in [GH+94] oder [PA+11] formuliert. Beispiele sind *data-parallelism*, *task-parallelism fork-join*, *master-worker* oder *pipeline*. Da die Formulierungen dieser Entwurfsmuster viel Interpretationsspielraum lassen, wurden in dieser Arbeit parallele Architekturmuster definiert, die viele der Konzepte dieser parallelen Entwurfsmuster umsetzen, durch UML-Diagramme jedoch sehr genau festlegen, wie die Komponenten der Muster zu verstehen sind. Um Verwechslungen vorzubeugen wird auf eine Vorstellung der parallelen Entwurfsmuster verzichtet. Ist in Folgenden Kapiteln von Taskgraph oder Fließband die Rede, sind stets die parallelen Architekturmuster aus Kapitel 4.5 gemeint. Sie decken sowohl Daten- und Aufgaben- als auch Fließbandparallelität ab.

2.2 Parallelisierung von Anwendungen

Im Gegensatz zu einer sequentiellen Anwendung besteht eine parallele Anwendung aus mehreren sogenannten Ausführungsfäden (engl. *threads*). Ein Ausführungsfaden kann vom Ausführungsplaner (engl. *scheduler*) des Betriebssystems immer nur einem Rechenkern zugewiesen werden. Bei Mehrkernsystemen ist es möglich, mehrere Ausführungsfäden einer parallelen Anwendung auf verschiedene Rechenkerne zu verteilen und dadurch Parallelität zu erzeugen, die im Idealfall dazu führt, dass sich die Gesamtausführungszeit der parallelen Anwendung verringert.

Ein Anwendungsprogrammierer hat keinen Einfluss auf den *scheduler* des Betriebssystems und damit auch keinen Einfluss auf die Reihenfolge, in welcher der Zustand der Ausführungsfäden geändert wird. Dadurch können beliebige Verschränkungen der Einzelanweisungen nebenläufiger Ausführungsfäden entstehen. Für zwei Fäden F_1 und F_2 mit den Anweisungen (A, B, C) und (X, Y, Z) sind beispielsweise (A, X, B, C, Y, Z) oder (A, X, Y, Z, B, C) als Fadenverschränkungen möglich. Aufgrund dieser Verschränkungen können nichtdeterministische Fehler – sogenannte Reihenfolgevertauschungen – entstehen: liest Y beispielsweise einen Wert, der von B geschrieben wird, so ergibt sich aus der ersten Verschränkung (in der die Reihenfolge B, Y eingehalten wird) ein anderes Ergebnis als die zweite Verschränkung (in der Y schon vor

B ausgeführt wird). Um solche Fehlern vorzubeugen, muss ein Anwendungsprogrammierer sogenannte Synchronisierungsprimitive einsetzten, um die Ausführungsreihenfolge von abhängigen Anweisungen - also solchen, die sich gegenseitig durch das Schreiben und Lesen einer gemeinsamen Speicherstelle beeinflussen - verschiedener Ausführungsfäden sicherzustellen.

Wenn im obigen Beispiel Y erst nach B ausgeführt werden darf, weil B einen Wert liest, den Y scheibt, dann ist Y abhängig von B. Eine wichtige Beobachtung für die automatische Parallelisierung sequentieller Anwendungen ist nun, dass die Reihenfolge abhängiger Anweisungen der sequentiellen Anwendungen auch in parallelen Anwendungen eingehalten werden muss, um die Semantik der Anwendung beizubehalten. Abhängigkeiten zwischen Anweisungen teilen sich in Kontrollfluss- und Datenabhängigkeiten, die im Folgenden definiert werden.

Kontrollflussabhängigkeit: Eine Anweisung A_2 ist kontrollflussabhängig von einer Anweisung A_1, wenn A_1 vor A_2 ausgeführt wird und das Resultat von A_1 darüber entscheidet, ob A_2 ausgeführt wird oder nicht.

Datenabhängigkeit: Eine Anweisung A_2 ist datenabhängig von einer Anweisung A_1, wenn A_1 vor A_2 ausgeführt wird, beide Anweisungen auf die gleiche Speicherstelle s zugreifen und mindestens eine der Anweisungen auf s schreibt. Drei Arten von Datenabhängigkeiten werden unterschieden:

- Read-after-Write (RAW): A_2 liest s nachdem A_1 s geschrieben hat.
- Write-after-Read (WAR): A_2 schreibt s nachdem A_1 s gelesen hat.
- Write-after-Write (WAW): A_2 schreibt s nachdem A_1 s geschrieben hat.

Für die automatische Parallelisierung muss geklärt werden, wie diese Abhängigkeiten aus einer sequentiellen Anwendung ermittelt werden können. Diese Frage wird durch Programmanalysen beantwortet. Wenn diese Abhängigkeiten bekannt sind, ist zu klären, wie die Anweisungen der sequentiellen Anwendung so auf verschiedene Ausführungsfäden aufgeteilt werden können, dass die ursprüngliche Reihenfolge abhängiger Anweisungen trotz möglicher Fadenverschränkungen eingehalten wird. Hier kommen die bereits eingeführten parallelen Entwurfs- beziehungsweise Architekturmuster zum Einsatz.

2.3 Programmanalysen

Eine Programmanalyse ist ein strukturierter Prozess zur Gewinnung bestimmter Informationen über ein Programm. Für diese Arbeit relevante Beispiele für Programmanalysen sind Laufzeitanalysen, die einzelnen Bestandteilen (wie Methoden, Schleifen oder Anweisungen) Laufzeiten zuordnen, insbesondere jedoch Abhängigkeitsanalysen, die Daten- und Kontrollflussabhängigkeiten zwischen Bestandteilen von Programmen ermitteln. Andere Analysen bewerten beispielsweise die Codequalität [AH+08] oder die Sicherheit [WF08] von Programmen. Es existieren zwei grundlegende Herangehensweisen, um Programme zu analysieren: statische und dynamische Analysen. Diese werden im Folgenden kurz diskutiert.

2.3.1 Statische Analysen

Bei der statischen Analyse wird das Programm nicht ausgeführt, sondern nur die Spezifikation des Programms analysiert: meistens also der Quellcode. Statische Analysen sind vor allem aus dem Bereich des Übersetzerbaus bekannt. Ein Beispiel ist die Entfernung von totem Quellcode [KR+94] (engl. *dead code elimination*), worunter Variablen fallen, die zwar deklariert aber nie verwendet werden, oder Anweisungen, die aufgrund von Kontrollflussanweisungen niemals ausgeführt werden. Solche statischen Analysen sind einfach, weil sie intraprozeduraler Natur sind: die Analysen müssen nur den Quellcode innerhalb einer Methode berücksichtigen, um zu dem gewünschten Ergebnis zu gelangen.

Schwieriger sind interprozedurale Analysen (wie beispielsweise Datenabhängigkeitsanalysen), die auch Methodenaufrufe nachverfolgen müssen: aufgrund von Vererbung und dynamischer Bindung ist unter alleiniger Betrachtung des Aufrufs nicht notwendigerweise klar, welche Methode durch einen Methodenaufruf ausgeführt wird. Um zu klären, welche Methode aufgerufen wird, muss der Typ des Objekts eines dynamischen Methodenaufrufs bestimmt werden. Diese Fragestellung kann durch eine Zeigeranalyse (engl. *pointer analysis*) beantwortet werden, die für jeden Zeiger im Programm eine Menge von Objekten (engl. *points-to-set*) berechnet, auf die der Zeiger im Verlauf der Ausführung zeigen kann. Zeigeranalysen können im Allgemeinen jedoch keine exakten Ergebnisse liefern, weil gezeigt werden kann [C03], dass dieses Problem nicht entscheidbar ist. Deshalb wurden Zeigeranalysen entwickelt, deren Komplexität auf Kosten der Präzision der Ergebnisse geringer ausfällt. Die *points-to-sets* dieser Zeigeranalysen stellen eine Überapproximation dar: es wird dafür garantiert, dass alle Objekte enthalten sind, auf die während irgendeiner tatsächlichen Ausführung gezeigt werden kann, das *points-to-set* eines Zeigers kann jedoch auch Objekte enthalten, auf welche der Zeiger während einer tatsächlichen Ausführung niemals zeigen kann. Es gibt viele Möglichkeiten, um die Komplexität von Zeigeranalysen zu reduzieren. Drei dieser Möglichkeiten sind: (i) Fluss-Insensitivität, (ii) Kontext-Insensitivität und (iii) Struktur-Insensitivität. Bei einer fluss-insensitiven Analyse wird der Kontrollfluss (also die Ausführungsreihenfolge von Anweisungen) nicht bei der Berechnung der *points-to-sets* mit einbezogen. Anstatt für jede Stelle (Anweisung) im Programm eine Zuordnung von Zeigern zu Objekten auszugeben, wird also nur eine einzige programmweite Lösung des Problems berechnet. Eine kontext-insensitive Analyse ignoriert den Aufrufkontext von Methoden. Zwei verschiedene Methodenaufrufe zur gleichen Methode werden also nicht getrennt behandelt, selbst wenn durch die beiden Aufrufe wegen unterschiedlichen Argumenten grundverschiedene Objekte angefasst werden. Eine struktur-insensitive Analyse behandelt Elemente von Arrays und Instanzvariablen von Objekten nicht einzeln, sondern als Ganzes. Einen guten Überblick der Probleme von statischen Zeiger-analysen gibt [H01]. Ein tiefgreifendes Verständnis statischer Programmanalysen wird in [FH+05] vermittelt.

Die Präzision einer statischen Datenabhängigkeitsanalyse ist direkt von der Präzision der verwendeten Zeigeranalyse abhängig, weil sie auf deren Ergebnis (den *points-to-sets*) aufbaut: gemäß der Definition sind zwei Anweisungen datenabhängig, wenn beide die gleiche Speicher-stelle (das gleiche Objekt) nutzen und mindestens eine Anweisung schreibend darauf zugreift. Bei der Verwendung einer kontext-insensitiven Zeigeranalyse sind zwei Anweisungen mit Methodenaufrufen zur gleichen Methode automatisch abhängig, selbst wenn sie niemals die gleichen Objekte schreiben, weil die *points-to-sets* für beide Anweisungen gleich sind. Die Verwendung einer struktur-insensitiven Analyse führt dazu, dass zwei Anweisungen, die das

gleiche Array nutzen automatisch abhängig sind, selbst wenn sie niemals das gleiche Element schreiben, weil die *points-to-sets* beide das Array enthalten und nicht das Arrayelement. In beiden Fällen wäre eine automatische Parallelisierung von beispielsweise `MergeSort` (siehe Kapitel 6.2) oder `RayTracer` (siehe Kapitel 6.3) nicht möglich.

2.3.2 Dynamische Analysen

Im Gegensatz zur statischen Analyse wird das Programm bei der dynamischen Analyse ausgeführt, um Informationen über das Programm zu gewinnen. Vor der Ausführung wird das Programm instrumentiert: an gewissen Stellen im Programm werden Instrumentierungsanweisungen eingefügt, um bestimmte Ereignisse zu erfassen, die es ermöglichen, die gewünschte Information während der Ausführung des instrumentieren Programms zu generieren. Die Art und Weise, wie instrumentiert wird ist also stets davon abhängig, welche Information letztlich gewonnen werden soll.

Gegenüber statischen Analysen haben dynamische Analysen den Vorteil, dass Spekulationen darüber, auf welches Objekt ein Zeiger zeigt, wegfallen. Durch eine Instrumentierungsanweisung an entsprechender Stelle kann es einfach abgegriffen werden. In Bezug auf die Ermittlung von Datenabhängigkeiten kann man somit eine dynamische Analyse entwickeln, die im Gegensatz zu statischen Analysen keine Überapproximation liefert: es werden nur Datenabhängigkeiten ermittelt, die während eines tatsächlichen Laufs wirklich auftreten können. Dem steht jedoch entgegen, dass nicht garantiert werden kann, dass die ermittelten Datenabhängigkeiten vollständig sind. Wenn ein Pfad des instrumentierten Programms während der Ausführung nicht besucht wird, werden für diesen Pfad keine Datenabhängigkeiten ermittelt.

2.4 Werkzeuge

Im Folgenden wird kurz auf die grundlegende Struktur von C#-Projekten eingegangen, weil das Konzept prototypisch für die Programmiersprache C# umgesetzt wird. Zudem wird auf die Werkzeuge Roslyn und CCI eingegangen, die zur Verarbeitung von C#-Projekten herangezogen werden.

2.4.1 C#-Projekte

Visual Studio legt Softwareprojekte, die mit C# programmiert sind, in einer sogenannten Arbeitsmappe (engl. *solution*) ab. Eine Arbeitsmappe ist ein Ordner, in dem sich neben Unterordnern für einzelne Teilprojekte insbesondere eine Beschreibungsdatei mit der Dateiendung .sln befindet, die zum Öffnen des Softwareprojekts herangezogen wird. Die Unterordner wiederum enthalten eine Beschreibungsdatei für die Teilprojekte (*.csproj) und den zugehörigen Quellcode (*.cs). Beim Übersetzen einer *solution* wird Zwischencode (IL-Code) erzeugt, der in einer sogenannten *assembly* gebündelt wird. Eine ausführbare *assembly* trägt typischerweise die Dateiendung .exe, eine Bibliothek die Endung .dll. Das Buch „CLR via C#“ [R10] bildet ein umfassendes Nachschalgewerk für C#-Quellcode, „Expert .NET 2.0 IL Assembler“ [L06] für IL-Code. Die offiziellen Standards sind in [E334] und [E335] zu finden.

2.4.2 Roslyn

Roslyn [Ros] ist ein Forschungsprojekt der Firma Microsoft, das momentan in der Vorabversion „Roslyn September 2012 CDT“ verfügbar ist. Das erklärte Ziel des Projekts ist es, die „Blackbox“ des Übersetzers für C#-Projekte (*solutions*) Programmierern zugänglich zu machen. Die Syntax-API von Roslyn stellt Parser zur Erzeugung von abstrakten Syntaxbäumen bereit,

die durch sogenannte SyntaxWalker und SyntaxRewriter besucht und verändert werden können. Die Semantics-API ermöglicht es, beispielsweise Typinformationen von einzelnen Syntaxknoten abzufragen. Die Services-API kann dazu genutzt werden, Softwareprojekte zu Laden und zu Speichern. Die Implementierung des in dieser Arbeit vorgestellten Verfahrens greift auf diese APIs zurück um *solutions* zu instrumentieren und zu annotieren. In Kapitel 5.1 wird darauf eingegangen, wo und wie die Funktionalität von Roslyn konkret in Anspruch genommen wird.

2.4.3 CCI

Common Compiler Infrastructure [Cci] ist ein weiteres Forschungsprojekt der Firma Microsoft, welches auf abstrakter Sicht eine ähnliche Funktionalität wie Roslyn bereitstellt. Während Roslyn sich auf *solutions* und Quellcode bezieht, ist CCI dazu gedacht mit *assemblies* und Zwischencode (IL-Code) zu arbeiten. CCI kann verwendet werden, um *assemblies* zu laden, zu verändern und wieder zurückzuschreiben und wird in dieser Arbeit genutzt um *assemblies* zu instrumentieren. Wie CCI von der Implementierung dieser Arbeit konkret genutzt wird, ist Kapitel 5.2 zu entnehmen.

3 VERWANDTE ARBEITEN

In diesem Kapitel werden verwandte Arbeiten vorgestellt, die sich ebenfalls mit dem Prozess der Parallelisierung von sequentiellen Anwendungen beschäftigen. Angefangen von Arbeiten, die lediglich interessante Abschnitte zur Parallelisierung aufspüren, über Verfahren, die Schätzungen des theoretisch erzielbaren Maximums der Beschleunigung ermitteln, hin zu Konzepten, die eine Überprüfung von benutzerannotierten Parallelisierungsvorschlägen ermöglichen. Arbeiten, die sich ebenfalls mit dem Auffinden und Ausnutzen von Parallelisierungsmöglichkeiten befassen sind im letzten Abschnitt aufgeführt und besitzen die größte Übereinstimmung mit dieser Arbeit.

3.1 Auffinden von vielversprechenden Abschnitten zur Parallelisierung

Für gewöhnlich werden sogenannte HotSpot-Profiler eingesetzt um Stellen in einer Anwendung aufzufinden, die interessant für die Parallelisierung sind, weil sie einen wesentlichen Teil der Gesamtlaufzeit der Anwendung beanspruchen. Traditionell wird nach aufwändigen Funktionen gesucht, neuere Arbeiten [MC+07] fassen auch Schleifen als kleinste Einheit auf.

Es gibt zwei Techniken um aufwändige Stellen zu finden: (i) Instrumentierung und (ii) Stichprobenprüfung (engl. *sampling*). Bei der Instrumentierung werden Ein- und Austritte von Funktionen und Schleifen markiert, um beispielsweise deren Laufzeiten und Aufrufhäufigkeiten bei der Ausführung der instrumentierten Anwendung protokollieren zu können. Dabei ist zu berücksichtigen, dass die Laufzeiten stark durch die Instrumentierungsanweisungen beeinflusst werden können. Deshalb wurden Techniken entwickelt um diese Einflüsse nach Möglichkeit aus den Messungen herauszurechnen. Sampling ist eine Technik, die ohne Instrumentierungsanweisungen auskommt. Die zu vermessende Anwendung wird ausgeführt und nach einer gewissen Zeit (Sampling-Intervall) unterbrochen. Dann wird der Instruktionszeiger der Anwendung ausgelesen und ermittelt, zu welcher Funktion oder Schleife die Instruktion gehört. Für diese Funktion oder Schleife wird daraufhin ein Zähler erhöht. Durch diese Zähler lässt sich nach der Ausführung beispielsweise eine Näherung der relativen Laufzeiten von Funktionen und Schleifen angeben.

Die Frage, ob sich ein Parallelisierungsvorschlag lohnt, also nach dessen Umsetzung tatsächlich zu einem Laufzeitvorteil führt, ist orthogonal zu der Frage, ob ein Parallelisierungsvorschlag korrekt ist, also nach der Umsetzung nicht zu einem Fehlverhalten der Anwendung führt. Unter ausschließlicher Berücksichtigung von Laufzeiten können zwar aufwändige Stellen identifiziert werden - welches parallele Architekturmuster angewendet werden soll und ob es überhaupt eines gibt das zulässig ist, kann im Gegensatz zu dieser Arbeit nicht beantwortet werden. In dieser Arbeit wird die Frage nach lohnenswerten Parallelisierungsvorschlägen an einen sich anschließenden Auto-Tuner delegiert. Die Ergebnisse von HotSpot-Profilern könnten allerdings eingesetzt werden, um nicht lohnenswerte Parallelisierungsvorschläge schon zu einem früheren Zeitpunkt auszufiltern.

3.2 Schätzen des theoretischen Maximums der erzielbaren Beschleunigung

Durch die Ermittlung des kritischen Ausführungspfades einer sequentiellen Anwendung kann unter Berücksichtigung von Abhängigkeiten eine optimistische Schätzung für das theoretische Maximum der erzielbaren Beschleunigung ermittelt werden. Eine frühe Arbeit [K88] führt die sogenannte *critical path analysis (CPA*) ein, um solche Ausführungspfade für Fortran-Anwendungen zu ermitteln. In [HS+09] wird diese Technik auf Java-Anwendungen erweitert und angewandt. Mit [GJ+11] wird eine Erweiterung der Basistechnik, die *hierarchical critical path analysis (HCPA)* vorgestellt, um besser mit Verschachtelungen umgehen zu können.

Bei der Basistechnik *CPA* wird die zu vermessende Anwendung instrumentiert und für jede Variable eine sogenannte Schattenvariable (engl. *shadow variable*) eingeführt. Die Instrumentierung sorgt dafür, dass in einer Schattenvariablen der Zeitpunkt des letzten Schreibzugriffs der zugehörigen Variable vorgehalten wird. Der kritische Ausführungspfad der Anwendung kann dann ermittelt werden, indem für jede Anweisung der früheste mögliche Ausführungszeitpunkt angenommen wird: das Maximum aller Zeitpunkte der Variablen, von denen die Anweisung abhängig ist. Wenn T_{seq} die Laufzeit der sequentiellen Anwendung ist und T_{krit} die Laufzeit des kritischen Pfades, dann ist $S = T_{seq}/T_{krit}$ eine Schätzung des theoretischen Maximums der erzielbaren Beschleunigung.

Im Gegensatz zu dieser Arbeit wird die Ausnutzung des Parallelisierungspotentials in den drei Arbeiten nicht adressiert. Schon auf konzeptioneller Ebene erlaubt es die *CPA* nicht, Rückschlüsse auf konkrete parallel ausführbare Quelltextabschnitte und deren Abbildung auf parallele Architekturen zu ziehen. Auch durch die in [GJ+11] vorgestellte Erweiterung *HCPA* in Kombination mit sogenannten Parallelisierungsplanern (engl. *parallelism planner*) kann diese Informationslücke nicht vollständig schließen. Eine Schätzung des theoretischen Maximums der erzielbaren Beschleunigung könnte jedoch eingesetzt werden, um solche Anwendungen zu identifizieren, für die sich die Erzeugung von Parallelisierungsvorschlägen von dieser Arbeit von vornherein nicht lohnt.

3.3 Automatisches Überprüfen von benutzerannotierten Parallelisierungsvorschlägen

Die kommerziellen Werkzeuge Intel Parallel Studio [IPS] und Prism von CriticalBlue [CBP] bieten dem Benutzer unter anderem die Möglichkeit, einen Parallelisierungsvorschlag durch Annotationen im sequentiellen Quelltext auszudrücken. Durch eine anschließende dynamische Analyse wird überprüft, ob vorhandene Abhängigkeiten durch die vorgeschlagene Parallelisierung verletzt werden. Während sich [IPS] und [CBP] auf Task- und Schleifenparallelität konzentrieren, widmet sich die Arbeit [TC+07] mit dem gleichen Ansatz der Überprüfung von Pipelineparallelität.

Wie in dieser Arbeit, wird die Überprüfung der benutzerannotierten Vorschläge durch eine Instrumentierung der sequentiellen Anwendung möglich. Lese- und Schreiboperationen werden während der Ausführung der instrumentierten Anwendung mitprotokolliert und den zugehörigen Quellcodeabschnitten zugeordnet. Sollte es Abhängigkeiten zwischen Codeabschnitten geben, die vom Benutzer verschiedenen Ausführungsfäden zugeordnet wurden, wird ein Parallelisierungsvorschlag als ungültig markiert. In manchen Fällen erhält der Benutzer auch

einen Handlungshinweis der angibt, wie sich die aufgespürte Abhängigkeit möglicherweise auflösen lässt.

Bei den drei vorgestellten Arbeiten muss Parallelität vom Benutzer selbst angegeben werden, erst dann ist eine Überprüfung der Vorschläge möglich. Im Gegensatz dazu ermöglicht es das Verfahren dieser Arbeit, direkt gültige Parallelisierungsvorschläge ohne Hilfe des Benutzers auszugeben. Das Auffinden von Stellen zur Parallelisierung und die Schätzung des Parallelisierungspotentials werden von [IPS] und [CBP] zwar unterstützt, der Benutzer muss die Schritte jedoch einzeln abarbeiten und den Gesamtprozess mehrmals durchlaufen (siehe [IC11]).

3.4 Auffinden und Ausnutzen von Parallelisierungsmöglichkeiten

Obwohl es möglich ist unabhängige Quelltextabschnitte durch statische Analysen zu ermitteln und diese anschließend zu parallelisieren, scheitern automatisch parallelisierende Übersetzer oft daran, das vorhandene Parallelisierungspotential vollständig auszunutzen [KK+10]. Grund dafür sind die bereits angesprochenen Limitierungen von statischen Analysen. Deshalb setzen [W11] und [H11] statische Abhängigkeitsanalysen in Kombination mit Heuristiken ein, um die Auswirkung von falschpositiven Abhängigkeiten einzuschränken. Während die Heuristik in [H11] auf Kontextinformationen des Quelltextes aufbaut und ermittelte Abhängigkeiten aufgrund der Heuristik anerkennt oder ignoriert, werden Abhängigkeiten in [W11] lediglich durch einen Faktor berücksichtigt, der das geschätzte Parallelisierungspotential senkt. Obwohl beide Arbeiten versuchen Taskparallelität zu erkennen, wird die Ausnutzung der Parallelität durch eine automatische Transformation nur in [H11] durch sogenannte Auto-Futures umgesetzt. Der Ansatz der Auto-Futures wird auch in [KS+12] aufgegriffen.

Der Hauptunterschied zwischen den vorgestellten Arbeiten und dieser Arbeit besteht in der Art der Abhängigkeitsanalyse. Während statische Analysen eine Überapproximation[1] der tatsächlichen Abhängigkeiten liefern und die daraus generierten Parallelisierungsvorschläge stets korrekt[2] sind, ermittelt eine dynamische Analyse nur während des Profillaufs beobachtete Abhängigkeiten. Dadurch kann mehr Parallelisierungspotential entdeckt werden – es können sich dafür aber auch falsche Parallelisierungsvorschläge ergeben. Die Verwendung von Heuristiken führt ebenfalls zu möglichen falschen Parallelisierungsvorschlägen.

Neuere Arbeiten setzen deshalb zunehmend auf dynamische Abhängigkeitsanalysen. Aufgrund des großen Speicherverbrauchs und der erheblichen Verlangsamung der Zielanwendung während des Profillaufs wird aktiv an der Reduzierung des Mehraufwandes geforscht [KK+11‘]. Embla2 [MF+10] ist ein Werkzeug, das durch eine dynamische Abhängigkeitsanalyse Taskparallelität aufdeckt indem es versucht einzelne Anweisungen in separaten Fäden auszuführen. Durch die Ausgabe von möglichen Abspaltungs- und Synchronisierungspunkten sowie kritischen Pfaden wird der Entwickler beim Transformieren in eine parallele Variante unterstützt. Die restlichen Arbeiten versuchen Schleifenparallelität [KK+10] [TW+09] und

[1] Überapproximation: Eine Menge von Abhängigkeiten, die auch falschpositive Abhängigkeiten enthält.

[2] Dies gilt nur solange die analysierte Anwendung keinen Gebrauch von Reflexion oder dynamischem Nachladen von Klassen macht. Ansonsten muss die statische Analyse zumindest in Teilen Ergebnisse von profilbasierten Analysen wie beispielsweise [BS+11] verwenden, wodurch die garantierte Korrektheit verloren geht.

Pipelineparallelität [TF10] [RV+10] zu identifizieren. Während [RV+10] Pipelinestufen durch die Betrachtung des Kontrollflusses und der Abhängigkeiten bildet, wird in [TF10] zusätzlich der Versuch unternommen durch die Berücksichtigung von Laufzeitinformationen balancierte Stufen zu generieren.

Die zuletzt vorgestellten Arbeiten sind mit dieser Arbeit am ehesten vergleichbar. Die dynamische Analyse sowie das Auffinden und Ausnutzen verschiedener Arten von Parallelität sind ihnen gemeinsam. Im Gegensatz zu dieser Arbeit verfolgt jedoch keine der Arbeiten einen Ansatz um sämtliche Arten von Parallelität durch eine Analyse abzudecken. Ein weiterer Unterschied besteht darin, dass die generierten Parallelisierungsvorschläge in dieser Arbeit im sequentiellen Quellcode präsentiert werden, wohingegen die erwähnten Arbeiten auf einer Zwischensprache arbeiten, oder schwerer verständlichen parallelen Quellcode erzeugen. Da die erzeugten Parallelisierungsvorschläge verwandter Arbeiten vom Benutzer überprüft werden müssen, ist eine verständliche Darstellung der Vorschläge aber unabdingbar. Nur eine der vorgestellten Arbeiten erzeugt parallelen Quellcode, der sich anschließend auf die Zielplattform anpassen lässt. Es wurde jedoch bereits gezeigt, dass diese Anpassung einen erheblichen Einfluss auf die erzielbare Beschleunigung besitzt [S10]. Diese Anpassung wird in dieser Arbeit durch die Abbildung auf eine Architekturbeschreibungssprache möglich.

3.5 Zusammenfassung

Tabelle 1 gibt eine Gegenüberstellung zwischen verwandten Arbeiten und dieser Arbeit [DA] wieder. Einige verwandte Arbeiten machen eine Aussage über das Parallelisierungspotential sequentieller Anwendungen. Die Bewertung von Parallelisierungspotential wird nicht direkt von dieser Arbeit adressiert, sondern an den nachgelagerten Auto-Tuner delegiert. Wie in anderen Arbeiten werden Parallelisierungsvorschläge erzeugt und zugehörige Transformationen ausgegeben. Dabei ist diese Arbeit die einzige, die alle der genannten parallelen Architekturmuster behandelt. Wie die meisten neueren Arbeiten setzt diese Arbeit eine dynamische Abhängigkeitsanalyse ein. Unter den verwandten Arbeiten ist diese Arbeit (mit Ausnahme von [TW+09]) die einzige, die eine anschließende Anpassung der parallelen Anwendung auf die Zielplattform adressiert.

	[MC+07]	[K88], [HS+09], [GJ+11]	[IPS], [CBP]	[TC+07]	[W11]	[H11]	[MF+10]	[KK+10]	[TW+09]	[RV+10]	[TF10]	**[DA]**
Pot	✓	✓	✓	×	×	✓	✓	×	×	×	✓	**×**
Vor	✓	×	⅄	⅄	✓	✓	✓	✓	✓	✓	✓	**✓**
Trans	×	×	×	×	×	✓	✓	×	✓	✓	✓	**✓**
Arch	×	×	FJ, MW	P	FJ	FJ	FJ	MW, P	MW	MW, P	P	**FJ, MW, P**
Abh		d	d	d	s+H	s+H	d	d	d	d	d	**d**
Opt	×	×	×	×	×	×	×	×	✓	×	×	**✓**

Tabelle 1: Tabellarische Gegenüberstellung zwischen verwandten Arbeiten und dieser Arbeit [DA]

Die aufgeführten Punkte gestalten sich wie folgt:

Pot: Wird das Parallelisierungspotential der Anwendung bewertet?
ja: ✓, nein: ×

Vor: Werden Parallelisierungsvorschläge generiert?

ja: ✓, nein - nur überprüft: ⅄, nein: ×

Trans: Wird für die Vorschläge eine automatische Transformation des Quellcodes angeboten?
ja: ✓, nein: ×

Arch: Welche parallelen Architekturen werden berücksichtigt?
Fork-Join: FJ, Master-Worker: MW, Pipeline: P, keine: ×

Abh: Wie werden die berücksichtigten Datenabhängigkeiten ermittelt?
dynamisch: d, statisch mit zusätzlicher Heuristik: s+H

Opt: Wird die Optimierung der parallelen Anwendung auf die Zielplattform adressiert?
ja: ✓, nein: ×

4 KONZEPT

Dieses Kapitel stellt ein Verfahren zur Parallelisierung von sequentiellen Anwendungen vor dessen Mehrwert ist, dass parallele Architekturen und Tuning-Parameter basierend auf einem neuartigen Analyseverfahren nach dem *single static multiple dynamic*-Muster (SSMD [SM+13]) erkannt und durch eine Architekturbeschreibugssprache veräußert werden. Damit bildet das Verfahren einen ersten Grundbaustein für die vollständige Automatisierung des Parallelisierungsprozesses für objektorientierte sequentielle Anwendungen.

Nach der Zielsetzung in Kapitel 4.1 wird in Kapitel 4.2 zunächst ein Beispiel angeführt, durch welches ein intuitives Verständnis für das in dieser Arbeit vorgestellte Verfahren vermitteln und dessen Umsetzbarkeit motivieren soll. Im Anschluss daran werden in Kapitel 4.3 Anforderungen an das Verfahren formuliert und die daraus abgeleiteten Lösungen kurz vorgestellt. Daraufhin wird in Kapitel 4.4 ein Überblick über das Parallelisierungsverfahren gegeben und der grundlegende Parallelisierungsansatz des Verfahrens motiviert. In den darauffolgenden Kapiteln 4.5 bis 4.9 wird auf jeden der Verfahrensteile detailliert eingegangen: Zunächst werden parallele Architekturmuster definiert (Kapitel 4.5), die das Verfahren in einer sequentiellen Anwendung erkennen soll. Dann werden Tuning-Parameter dieser Architekturmuster diskutiert (Kapitel 4.6). Es folgt eine Erläuterung Analysen, die zur Ermittlung von Abhängigkeiten eingesetzt werden (Kapitel 4.7), welche wiederum bei der Erkennung der Architekturmuster berücksichtigt werden (Kapitel 4.8). Letztlich wird in Kapitel 4.9 eine Notation vorgestellt, mit der eine sequentielle Anwendung durch Architekturbeschreibungen annotiert werden kann. Den Abschluss bildet eine Zusammenfassung, die in Kapitel 4.10 vorzufinden ist.

4.1 Zielsetzung

Das Ziel des in diesem Abschnitt vorgestellten Verfahrens ist die automatische Generierung von Parallelisierungsvorschlägen in Form von Architekturbeschreibungen zur Unterstützung des Entwicklers beim Auffinden von Parallelisierungsmöglichkeiten für objektorientierte sequentielle Anwendungen. Durch die Ausgabe von Parallelisierungsvorschlägen soll das Verfahren die Aufmerksamkeit des Entwicklers gezielt auf solche Codeabschnitte lenken, die sich gut parallelisieren lassen. Ein Parallelisierungsvorschlag ist eine Beschreibung die angibt, wie ein Codeabschnitt der sequentiellen Anwendung auf einen parallelen Codeabschnitt abgebildet werden kann. In Abbildung 1 ist das Verfahren auf oberster Abstraktionsebene dargestellt: Als Eingabe dient die sequentielle Anwendung, ausgegeben wird die durch parallele Architekturbeschreibungen annotierte Anwendung.

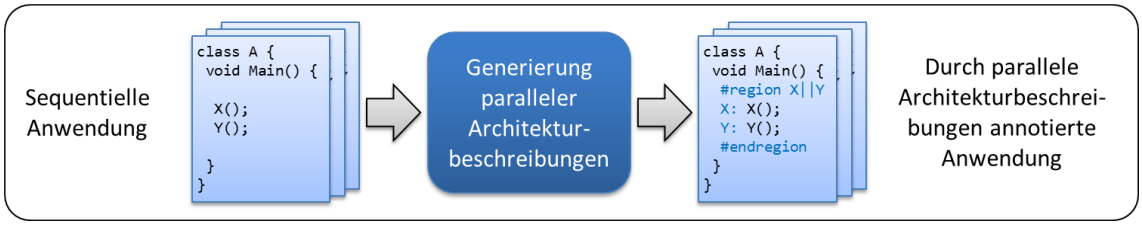

Abbildung 1: Das in dieser Arbeit vorgestellte Verfahren auf oberster Abstraktionsebene.

An das vorgestellte Verfahren schließt sich eine Werkzeugkette zur Weiterverarbeitung der Architekturbeschreibungen an: Ein Übersetzer, der in einer weiteren Forschungsarbeit [W13] umgesetzt wird, ermöglicht die automatische Transformation von annotiertem Quellcode in parallelen Quellcode. Der Übersetzer hält dazu eine Bibliothek mit konkreten Implemen-

tierungen für parallele Architekturmuster vor und bildet die Architekturbeschreibungen, die als Ergebnis der vorliegenden Arbeit ausgegeben werden, darauf ab. Wie [SM+13] zeigen, können fehlerhafte Parallelisierungsvorschläge durch automatisch generierte Testfälle und einen Wettlauferkenner (engl. *race detector*) erkannt und ausgesondert werden. Ein Auto-Tuner kann im Anschluss daran eine Anpassung auf die Zielplattform vornehmen und optimale Werte für Tuning-Parameter festlegen. Somit steht diese Arbeit im Kontext eines Parallelisierungsprozesses, ist in diesen eingebettet und stellt einen ersten wichtigen Schritt zur automatischen Parallelisierung von Bestandssoftware dar.

4.2 Motivierendes Beispiel

Bevor die Anforderungen dieser Arbeit formuliert werden und im Anschluss auf das Konzept eingegangen wird, soll anhand eines konkreten Beispiels motiviert werden, wie die automatische Generierung paralleler Architekturbeschreibungen ermöglicht wird. Als Beispiel dient die reale Anwendung `VideoProcessing` zur Bearbeitung eines Videostroms, die einer Studienarbeit [D08] entstammt und später für die Evaluation nochmals im Detail aufgegriffen wird. Ein Ausschnitt des Quellcodes der Anwendung ist in Abbildung 2 dargestellt.

```
1 public void ProcessImages() {
2   foreach (Bitmap bmp in _inputStream) {
3    var tmp_bmp = bmp;
4    tmp_bmp = doCrop(tmp_bmp);
5    tmp_bmp = doHistogramEqualization(tmp_bmp);
6    ...
7    ConsumeBmp(tmp_bmp);
8   }
9 }
```

Abbildung 2: Ein Ausschnitt des Quellcodes der Anwendung `VideoProcessing`.

Die dargestellte Schleife durchläuft die Einzelbilder des Videostroms der Reihe nach und wendet eine Folge von Bildbearbeitungsfiltern auf jedes Einzelbild an, bevor das bearbeitete Einzelbild in einen Ausgabevideostrom zurückgeschrieben wird. Der vielleicht naheliegende Parallelisierungsansatz – eine parallele `foreach`-Schleife – würde zu einem Fehler führen, selbst wenn man Zeile 7 durch eine Sperre schützt: Beim Zurückschreiben der Einzelbilder könnte die ursprüngliche Reihenfolge der Einzelbilder missachtet werden, wodurch das resultierende Video vollkommen entartet wird. Der Grund dafür, dass diese Parallelisierung fehlschlägt ist in der Abhängigkeit der Schleifeniterationen begründet: Betrachtet man jede Schleifeniteration als Ganzes, dann ist eine solche Iteration stets von der vorhergehenden datenabhängig, weil beide in den Ausgabevideostrom schreiben.

Um die dargestellte Schleife dennoch parallelisieren zu können, wird in dieser Arbeit ein feingranularer Ansatz gewählt, der Iterationen nicht als Ganzes betrachtet, sondern einzelne Anweisungen einer Iteration als kleinste Einheit auffasst. Die Grundidee entstammt der hardwareseitigen Fließbandverarbeitung, die auch in Software umgesetzt werden kann: Jede Anweisung der Schleife wird als Fließbandstufe aufgefasst, Variablen (wie das Einzelbild `tmp_bmp`) die von den Anweisungen gelesen und geschrieben werden als Datenelemente, die durch die Stufen fließen. Über Puffer werden die Datenelemente von einer Stufe zur nächsten weitergereicht. Jeder Fließbandstufe wird dabei eine eigene ausführende Einheit – also ein Ausführungsfaden – zugewiesen. Ein einfaches Fließband, mit welchem die Schleife aus Abbildung 2 parallelisiert werden kann, ist in Abbildung 3 dargestellt.

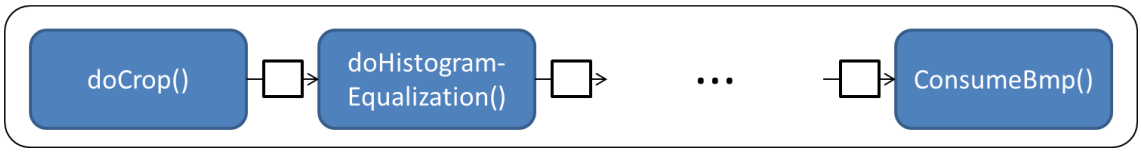

Abbildung 3: Einfaches Fließband zur Parallelisierung der Schleife des Beispiels aus Abbildung 2.

Mit diesem Fließband ist folgende Ausführung möglich: das erste Einzelbild `b1` wird in der Stufe `doCrop()` bearbeitet. Sobald diese Bearbeitung abgeschlossen ist, wird das bearbeitete Einzelbild `b1'` in den Ausgabepuffer der Stufe gelegt, der gleichzeitig den Eingabepuffer der nächsten Stufe `doHistogramEqualization()` darstellt. Jetzt kann die erste Stufe schon das nächste Einzelbild `b2` bearbeiten, während die zweite Stufe gleichzeitig das erste Einzelbild `b1'` weitermodifiziert. Zusätzlich wird hier eine weitere (nicht abgebildete) Stufe benötigt, welche die Einzelbilder der Reihe nach in das Fließband hineingibt, die im Folgenden Erzeuger-Stufe genannt wird. In diesem einfachen Beispiel ist die Erzeuger-Stufe mit der `foreach`-Schleife (ohne deren Schleifenrumpf) gleichzusetzen. Durch die gleichzeitige Bearbeitung von Einzelbildern in verschiedenen Stufen wird bei einem Fließband Parallelität erzeugt, und zudem die Reihenfolge der Einzelbilder beibehalten. Es kann daher nicht zu dem Fehler kommen, den die parallele `foreach`-Schleife erzeugen würde.

Das in Abbildung 3 dargestellte Fließband kann allerdings noch verbessert werden: Es fällt auf, dass nur die Stufe `ConsumeBmp()`, die für das Zurückschreiben der Einzelbilder in den Ausgabevideostrom verantwortlich ist, die Reihenfolge der Bilder beachten muss, solange dafür gesorgt wird, dass die Reihenfolge vor der `ConsumeBmp()`-Stufe wieder hergestellt wird. Es können also durchaus mehrere Einzelbilder gleichzeitig von der ersten beziehungsweise zweiten Stufe bearbeitet werden, weil diese Stufen im Gegensatz zu `ConsumeBmp()` keinen internen Zustand halten. Eine zustandsfreie Stufe kann repliziert – das heißt durch mehrere Ausführungsfäden parallel ausgeführt – werden. Dies ist in Abbildung 4 durch mehrere Instanzen der gleichen Stufe verdeutlicht. Eine replizierte Stufe besitzt einen impliziten Parameter: die Anzahl der Ausführungsfäden, der ihr zugeordnet wird. Die optimale Wahl dieses Parameters ist nicht unbedingt trivial und kann die Laufzeit des Fließbandes erheblich beeinflussen. In dieser Arbeit werden weitere solche Parameter identifiziert und als implizite Tuning-Parameter ausgegeben, die anschließend von einem Auto-Tuner auf die Zielplattform angepasst werden können.

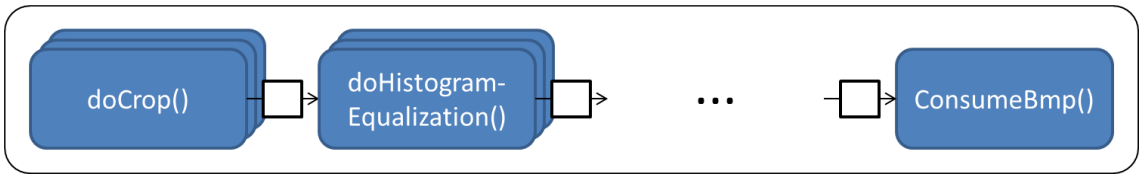

Abbildung 4: Verbessertes Fließband zur Parallelisierung der Schleife des Beispiels.

Das Konzept dieser Arbeit ermöglicht es, ein solches Fließband automatisch auszugeben. Wie in der Evaluierung gezeigt wird, kann somit eine quasi „kostenlose“ Beschleunigung von bis zu 5,3 auf einem Rechner mit acht Prozessorkernen erzielt werden. Auch replizierbare Stufen werden dabei erkannt und entsprechend markiert.

Die Ausgabe eines solchen Fließbandes findet dabei nicht direkt als paralleler Quelltext, sondern in Form von annotiertem Quelltext statt, wie er in Abbildung 5 dargestellt ist. Die Anweisungen, die als Fließbandstufe aufgefasst werden, bekommen Bezeichner (`S1` bis `S9`), die in

der sogenannten Architekturbeschreibung zur Beschreibung des Fließbandes aufgegriffen werden. Replizierbare Fließbandstufen sind dort durch ein Plus gekennzeichnet.

```
1  public void ProcessImages() {
2   #region S1+ => S2+ => ... => S9
3    foreach (Bitmap bmp in _inputStream) {
4     var tmp_bmp = bmp;
5     S1: tmp_bmp = doCrop(tmp_bmp);
6     S2: tmp_bmp = doHistogramEqualization(tmp_bmp);
7     ...
8     S9: ConsumeBmp(tmp_bmp);
9    }
10  #endregion
11 }
```

Abbildung 5: Der mit dem Fließband aus Abbildung 4 annotierte Quellcodeausschnitt des Beispiels.

Im Allgemeinen gestaltet sich die Erkennung und Ausgabe eines Fließbandes weitaus komplexer, als im vorgestellten Beispiel: Zwischen Fließbandstufen kann es iterationsübergreifende Abhängigkeiten geben, was ein Zusammenfassen dieser Stufen nötig macht. Außerdem soll das Konzept auch für andere Schleifentypen (`for`- und `while`-Schleifen) funktionieren, für die eine Abbildung auf eine Erzeuger-Stufe nicht immer so einfach ist, wie in diesem Beispiel.

Um iterationsübergreifende Abhängigkeiten zwischen Anweisungen einer Schleife bei der Abbildung auf ein Fließband berücksichtigen zu können, müssen diese zunächst ermittelt werden. Dieser Vorgang selbst ist nicht trivial. Der Abhängigkeitsanalyse ist daher ein eigenes Kapitel gewidmet (Kapitel 4.7). Wie diese Analyse konkret funktioniert, greift für ein motivierendes Beispiel zu weit. Konzeptionell ist die Abhängigkeitsanalyse jedoch wie folgt aufgebaut: Der Quelltext wird instrumentiert, sodass vor und nach jeder Anweisung eine Eintritts- und eine Austrittsanweisung steht, um zu signalisieren, wann eine Anweisung „aktiv“ ist. Wenn während der „aktiven“ Phase einer Anweisung ein Speicherzugriff stattfindet (vor dem ebenfalls eine entsprechende Anweisung angebracht wird, um diesen Vorgang mitzuprotokollieren), kann dieser Speicherzugriff mit der Anweisung in Verbindung gebracht werden. Dadurch lassen sich Datenabhängigkeiten zwischen Anweisungen ableiten. Im obigen Beispiel wird beispielsweise erkannt, dass die Anweisung `ConsumeBmp()` einer Iteration abhängig ist von derselben Anweisung der vorherigen Iteration. Damit wurde eine iterationsübergreifende Abhängigkeit einer möglichen Fließbandstufe zu sich selbst identifiziert, die dazu führt, dass diese Stufe als nicht replizierbar markiert wird. Wie genau Abhängigkeiten bei der Abbildung von Schleifen auf Fließbänder zu berücksichtigen sind wird ebenfalls in einem eigenen Kapitel erläutert (Kapitel 4.8).

Neben der Möglichkeit, Schleifen durch Fließbänder zu parallelisieren, wird in dieser Arbeit auch ein Ansatz beschrieben um Sequenzen durch Taskgraphen zu parallelisieren, wodurch zum Beispiel bei der Anwendung `MergeSort` (die ebenfalls in der Evaluierung im Detail diskutiert wird) Beschleunigungen von bis zu 3,7 erzielt werden.

4.3 Anforderungen

Das Verfahren soll einer Reihe von Anforderungen genügen, die im Folgenden mitsamt den daraus abgeleiteten Lösungen kurz präsentiert werden. Auf jede Lösung wird in den Kapiteln 4.4 bis 4.9 im Detail eingegangen.

1. **Anforderung: Flexible Parallelisierungsstrategien**

   Die Identifizierung von Parallelisierungsmöglichkeiten soll sich an wiederkehrenden und möglichst allgemeinen Parallelisierungsstrategien orientieren.

   Lösung: Parallele Architekturmuster

   Erläuterung: Verschiedene Parallelisierungsstrategien werden in Form konfigurierbarer paralleler Architekturmuster vordefiniert: Taskgraph und Fließband. Diese werden in Kapitel 4.5 definiert und erläutert. Parallele Architekturmuster bestehen aus nebenläufig ausführbaren Komponenten und steuern deren Nebenläufigkeit. Die nebenläufigen Komponenten wiederum besitzen Platzhalter für sequentielle Codeabschnitte. Parallelisierungsmöglichkeiten werden identifiziert, indem in der sequentiellen Anwendung nach Codeabschnitten gesucht wird, die sich auf eine Konfiguration eines parallelen Architekturmusters abbilden lassen. Eine Konfiguration eines parallelen Architekturmuster lässt sich durch die Festlegung seiner Parameterwerte angeben. Zu diesen Parametern gehören die Anzahl der nebenläufigen Komponenten und deren Platzhalter. Ein paralleles Architekturmuster kann aber insbesondere auch weitere Parameter definierten, die dessen Verhalten und Laufzeit beeinflussen. Eine mögliche Konfiguration eines Fließbands wäre also beispielsweise ein dreistufiges Fließband, wobei zu jeder Stufe der sequentielle Codeabschnitt angegeben und die dritte Stufe als replizierbar gekennzeichnet ist: `A => B => C+`.

   Verwandte Arbeiten (siehe dazu Kapitel 3.4) diskutieren entweder Aufgaben- [MF+10] [W11] [H11], Schleifen- [KK+10] [TW+09] oder Fließbandparallelität [RV+10] [TF10]. Diese Parallelisierungsansätze können durch die parallelen Architekturmuster Taskgraph und Fließband abgedeckt werden. Die Ausdrucksmächtigkeit eines Taskgraphen ist zudem mächtiger, als die oft verwendete Notation sequentieller und paralleler Sektionen, die beispielsweise von OpenMP angeboten wird (siehe dazu Kapitel 4.9.1). Durch Taskgraphen und Fließbänder kann sowohl Aufgaben-, Daten als auch Fließbandparallelität (siehe Kapitel 2.1) ausgenutzt werden.

2. **Anforderung: Leichtes Verständnis**

   Ein Parallelisierungsvorschlag soll für einen Entwickler gut zu verstehen sein. Aus dem Parallelisierungsvorschlag soll intuitiv hervorgehen, welche Bestandteile des sequentiellen Codeabschnitts nebenläufig ausgeführt werden. Von Implementierungsdetails soll möglichst abstrahiert werden.

   Lösung: Architekturbeschreibungen und Codepartitionierungen

   Erläuterung: Die zweite Anforderung wird durch die explizite Trennung von sequentiellem Code, der die Funktionalität der Anwendung beschreibt, und parallelem Code, der die Ausführungsreihenfolge der sequentiellen Bestandteile beschreibt, erreicht. Eine Architekturbeschreibung gibt eine Konfiguration eines parallelen Architekturmusters

an, wie sie beispielhaft in 1. aufgeführt wurde. Eine Codepartitionierung zerlegt einen sequentiellen Codeabschnitt in mehrere Bestandteile und ordnet sie den Komponenten des Architekturmusters zu: im Beispiel wird `A`, `B` und `C` dadurch Quellcode zugeordnet. Gemeinsam bilden Architekturbeschreibung und Codepartitionierung einen Parallelisierungsvorschlag, der einen sequentiellen Codeabschnitt auf einen parallelen abbildet.

Bestehende Arbeiten konnten bereits zeigen, dass Architekturbeschreibungen sinnvoll von Entwicklern eingesetzt werden können um Parallelität kompakt und verständlich auszudrücken [O13]. Im Gegensatz zu verwandten Arbeiten, die direkt parallelen Quell- oder Zwischencode ausgeben, können Implementierungsdetails hinter den parallelen Architekturmustern verborgen werden, was das Verständnis und die Lesbarkeit der erzeugten Parallelisierungsvorschläge erleichtert. Wichtiger ist jedoch, dass die Implementierung der Architekturmuster ausgetauscht werden kann, ohne die Analyse erneut durchführen zu müssen.

3. **Anforderung: Optimierbarkeit**

   Eine Architekturbeschreibung soll keine komplette Konfiguration, sondern nur die Werte der Parameter eines parallelen Architekturmusters angeben, deren Festlegung zur Erhaltung der Semantik der sequentiellen Anwendung nötig ist. Dies ist dann der Fall, wenn der Wert eines Parameters Einfluss auf die Ausführungsreihenfolge der abhängigen sequentiellen Bestandteile hat. Alle anderen Parameterwerte sollen optimal auf die Zielplattform angepasst werden können.

   Lösung: Tuning-Parameter

   Erläuterung: Die Parameter paralleler Architekturmuster werden darauf untersucht, ob deren Wert einen Einfluss auf die Semantik hat. Parameter, die stets einen Einfluss darauf besitzen sind die maximale Anzahl nebenläufiger Komponenten, deren Verknüpfung und deren Platzhalter. Diese Parameter werden in jeder Architekturbeschreibung angegeben. Parameter, die nie einen Einfluss auf die Semantik, wohl jedoch auf die Laufzeit des Architekturmusters haben sind beispielsweise Puffergrößen oder die Zusammenfassung mehrerer nebenläufiger Komponenten zu einer. Diese Parameter werden bewusst offen gelassen und implizit als Tuning-Parameter veräußert, um später von einem Auto-Tuner an die Zielplattform angepasst werden zu können. Für Parameter, die unter bestimmten Bedingungen einen Einfluss auf die Semantik haben, wird eine Notation definiert um deren Wertebereich in der Architekturbeschreibung einschränken zu können. Eine Nichteinschränkung impliziert wiederum einen Tuning-Parameter. Diese Arbeit erkennt Tuning-Parameter also automatisch und unterscheidet zwischen semantikerhaltenden und semantikverfälschenden Parametern. Diese werden durch die gewählte Architekturbeschreibung veräußert um zu ermöglichen, dass ein Übersetzer entsprechende Transformationen automatisch durchführen und ein Auto-Tuner Werte für diese Parameter festlegen kann. Dies stellt einen der Mehrwerte dieser Arbeit dar.

   In bestehenden Arbeiten konnte bereits gezeigt werden, dass Tuning-Parameter und anschließendes Auto-Tuning wichtig sind, um mögliche Laufzeitverbesserungen paralleler Anwendungen voll auszuschöpfen [S10]. Diese Tatsache wird von verwand-

ten Arbeiten weitestgehend ignoriert. Nur eine Arbeit [TW+09] adressiert die anschließende Optimierung der parallelen Anwendung durch maschinelles Lernen.

4. **Anforderung: Keine Vorenthaltung**

   Dem Entwickler sollen keine Parallelisierungsvorschläge vorenthalten werden, die zu einer fehlerfreien parallelen Anwendung führen können. Falschpositive Abhängigkeiten sind unerwünscht.

   Lösung: Optimistische Analysen

   Erläuterung: Reihenfolgevertauschungen kausal abhängiger Bearbeitungsschritte führen zum Fehlverhalten der parallelen Anwendung. Die in Kapitel 4.8 vorgestellten Algorithmen zur Erzeugung von Abbildungen sequentieller Codeabschnitte auf parallele Architekturmuster berücksichtigen deshalb Abhängigkeiten zwischen Bestandteilen sequentieller Codeabschnitte. Diese Abhängigkeiten werden durch Programmanalysen automatisch ermittelt. Programmanalysen zur exakten Berechnung aller Abhängigkeiten existieren nur für praxisuntaugliche Spezialfälle, wie etwa Programme ohne dynamisch allokierten Speicher [C03]. Deshalb muss auf pessimistische oder optimistische Analysen zurückgegriffen werden, die entweder potentiell zu viele oder zu wenige Abhängigkeiten ermitteln. Automatisch parallelisierende Übersetzer verwenden pessimistische Analysen, die garantieren, dass die ermittelten Abhängigkeiten vollständig sind. Pessimistische Analysen berechnen jedoch auch (falschpositive) Abhängigkeiten, die während eines Programmlaufs nie auftreten können. Wie [KK+10] belegt, führt dies dazu, dass eine Vielzahl möglicher Parallelisierungsvorschläge unnötigerweise abgelehnt wird. Im Gegensatz zu pessimistischen Analysen ermitteln optimistische Analysen nur Abhängigkeiten, die während eines Programmlaufs tatsächlich auftreten können, treffen jedoch keine Aussage darüber, ob die ermittelten Abhängigkeiten vollständig sind. Unter dem Vorbehalt der Korrektheit können dadurch mehr Parallelisierungsvorschläge ausgegeben werden. In dieser Arbeit werden pessimistische (statische) und optimistische (dynamische) Abhängigkeitsanalysen deshalb bewusst nach dem SSMD-Muster (engl. *single static multiple dynamic*) [SM+13] kombiniert: auf eine unvollständige statische Analyse, die keine falschpositiven Abhängigkeiten erzeugt, folgen mehrere Durchläufe dynamischer Analysen (mit verschiedenen Eingabedaten für die sequentielle Anwendung) um möglichst alle Datenabhängigkeiten zu ermitteln.

   Dynamische Analysen werden auch von einigen verwandten Arbeiten untersucht [KK+10‘] und eingesetzt [MF+10] [TF10] [KK+10] [TW+09]. Im Gegensatz zu dieser Arbeit sind die untersuchten Abhängigkeitsanalysen nicht auf objektorientierten speicherverwalteten Quellcode ausgelegt. Zudem wird die Berücksichtigung von Abhängigkeiten aus verschiedenen Analysen von den verwandten Arbeiten nicht explizit unterstützt.

4.4 Parallelisierungsverfahren

Die erste große Herausforderung bei der Parallelisierung einer sequentiellen Anwendung ist die Identifizierung von nebenläufig ausführbaren Bearbeitungsschritten. Weil sequentielle Anwendungen unter der Annahme eines sequentiellen Ausführungsmodells entwickelt wurden gibt es neben kausal abhängigen Bearbeitungsschritten, deren Reihenfolge eingehalten werden muss, auch kausal unabhängige Bearbeitungsschritte, deren Reihenfolge willkürlich festgelegt wurde. Die Parallelisierung einer sequentiellen Anwendung besteht darin, kausal unabhängige Bearbeitungsschritte zu finden (durch die Abhängigkeitsanalyse in Kapitel 4.7), ihre willkürlich festgelegte Reihenfolge aufzuheben und sie stattdessen nebenläufig auszuführen. Die zweite große Herausforderung ist die Transformation der Bearbeitungsschritte in eine parallele Form. Darauf wird in Kapitel 4.8 eingegangen. Dieser Gesamtprozess wird durch das Parallelisierungsverfahren aus Abbildung 6 umgesetzt.

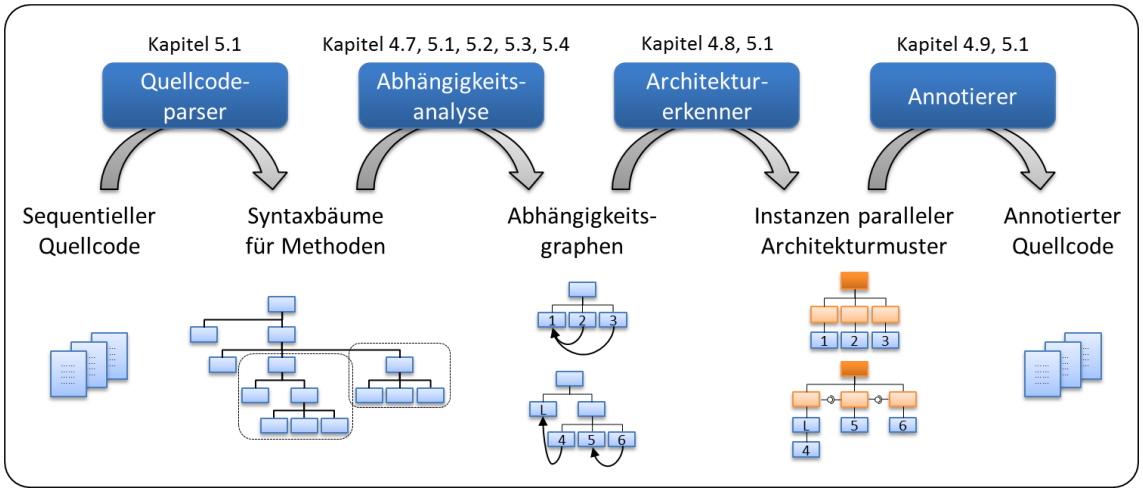

Abbildung 6: Das Parallelisierungsverfahren im Gesamtüberblick: dargestellt sind die vier Teilkomponenten und die ausgetauschten Datenformate.

Das Parallelisierungsverfahren dieser Arbeit ist ein Quellcode-zu-Quellcode-Verfahren, welches prototypisch in Kapitel 1 für die Programmiersprache C# umgesetzt ist: sequentieller Quellcode wird als Eingabe erwartet, die Ausgabe bilden Parallelisierungsvorschläge in Form von annotiertem Quellcode. Das Verfahren gliedert sich in vier Teilkomponenten:

- **Quellcodeparser:** Der Quellcodeparser überführt den sequentiellen Quellcode in Syntaxbäume. Syntaxbäume sind eine interne Darstellung des Quellcodes, auf der die weiteren Komponenten basieren. Zusammenhängende Quellcodeabschnitte werden im Syntaxbaum als Syntaxknoten aufgefasst. Als Quellcodeparser für die Sprache C# wird eine externe Komponente eingesetzt, auf die erst im Implementierungskapitel 5.1 näher eingegangen wird.
- **Abhängigkeitsanalyse:** Die Abhängigkeitsanalyse isoliert Teilbäume des Syntaxbaums, für die ein Parallelisierungsansatz verfolgt wird und identifiziert Abhängigkeiten zwischen den Syntaxknoten der Teilbäume. Sie generiert Abhängigkeitsgraphen, also um Abhängigkeiten ergänzte Teilbäume, die vom Architekturerkenner als Eingabe erwartet werden. In dieser Teilkomponente kommt die kombinierte statische und dynamische Analyse zum Einsatz. Die Erläuterungen zur Abhängigkeitsanalyse finden sich in Kapitel 4.7 wieder.

- **Architekturerkenner:** Im Architekturerkenner sind Algorithmen verankert, die Abhängigkeitsgraphen darauf untersuchen, ob und wie sie auf eine Konfiguration eines parallelen Architekturmusters abgebildet werden können. In Kapitel 4.8 werden diese Algorithmen vorgestellt und motiviert.
- **Annotierer:** Der Annotierer erhält die generierten Konfigurationen paralleler Architekturmuster und bildet sie über eine Annotationssprache, die Architekturbeschreibungen spezifiziert, wieder zurück auf Quellcode ab. Wie diese Abbildung konkret aussieht, wird in Kapitel 4.9 erläutert.

4.4.1 Parallelisierungsansatz

Der in dieser Arbeit verfolgte Parallelisierungsansatz stützt sich auf drei Programmierparadigmen, die von Universalprogrammiersprachen (engl. *general-purpose programming languages*) wie C++, C# oder Java verfolgt werden: prozedurale, imperative und strukturierte Programmierung. Ein prozedurales Programm ist durch Methoden in Lösungsansätze überschaubarer Teilprobleme gegliedert. Ein imperatives Programm legt fest, welche Bearbeitungsschritte (was) in welcher Reihenfolge (wie) durchzuführen sind. Zur Festlegung der Reihenfolge von Bearbeitungsschritten verwendet ein strukturiertes Programm vornehmlich Kontrollstrukturen (Sequenz, Schleife, Verzweigung) und verzichtet nach Möglichkeit auf beliebige Sprünge [D68].

Die Ursache für die willkürliche Festlegung der Reihenfolge kausal unabhängiger Bearbeitungsschritte sind also die Kontrollstrukturen Sequenz, Schleife und Verzweigung. Der Parallelisierungsansatz dieser Arbeit ist daher der Versuch, Instanzen dieser Strukturen auf dazu äquivalente Instanzen paralleler Codeabschnitte abzubilden, ohne dabei die Funktionalität der Methoden – also die Lösungsansätze überschaubarer Teilprobleme – nach außen hin zu verändern.

Die syntaktische Entsprechung des intuitiven Begriffs Bearbeitungsschritt ist im Kontext von Programmiersprachen die Anweisung. Eine Programmiersprache definiert verschiedene Ausprägungen von Anweisungen. Eine Anweisung ist insbesondere ein Kompositum: sie kann also wiederum aus Anweisungen zusammengesetzt sein. Anweisungen (blau) einer typischen Universalprogrammiersprache können wie in Abbildung 7 dargestellt klassifiziert werden. Die Kontrollstrukturen Sequenz, Schleife und Verzweigung finden sich dort als abstrakte Syntaxknoten wieder. Die Grammatik einer Programmiersprache kann als abstrakter Syntaxbaum, ein Programm als Instanz eines solchen Syntaxbaums angegeben werden.

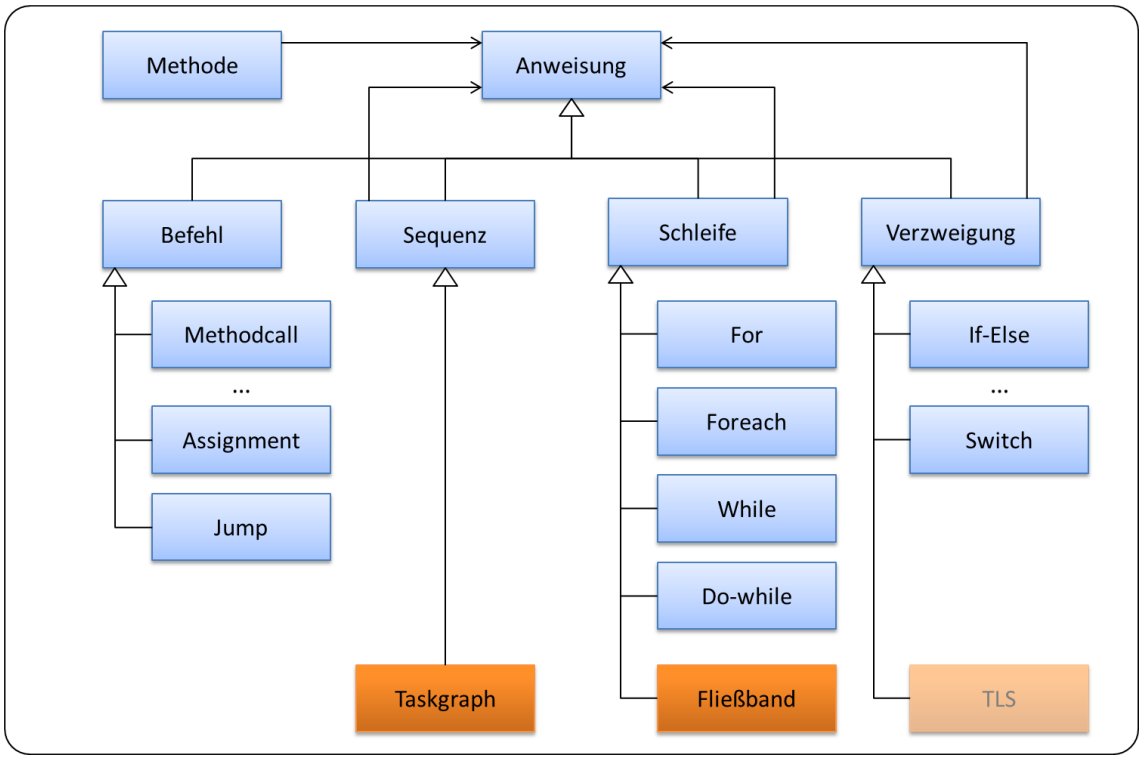

Abbildung 7: Klassifikationsschema von Syntaxknoten einer Programmiersprache: sequentielle Anweisungen (blau) und die Erweiterung um parallele Architekturmuster (orange). TLS wird in dieser Arbeit ausgeklammert.

Diese Arbeit erweitert die dargestellte Vererbungshierarchie um spezielle parallelitätsbezogene Anweisungen. Dabei wird das Programmierparadigma der strukturierten Programmierung konsequent für parallelitätsbezogene Anweisungen fortgeführt: anstatt beliebige Befehle zur Erzeugung und Synchronisation von Ausführungsfäden zu verwenden, werden parallele Kontrollstrukturen, sogenannte parallele Architekturmuster (orange), definiert: Taskgraph und Fließband. Parallele Architekturmuster sind eine Arte Meta-Muster, von dem konkrete Konfigurationen angegeben werden können, die weitläufig als *master-worker* oder *pipeline* bezeichnet werden (siehe Kapitel 2.1). Parallele Architekturmuster sind wiederum zusammengesetzte Anweisungen, können sich also insbesondere selbst enthalten, und werden in 4.5 im Detail beschrieben.

Auf Grundlage der dargestellten Vererbungshierarchie kann der Parallelisierungsansatz weiter konkretisiert werden: Im Syntaxbaum der sequentiellen Anwendung wird versucht, Instanzen von Sequenzen und Schleifen auf dazu äquivalente Konfigurationen von Taskgraphen und Fließbändern abzubilden. Die Abhängigkeitsanalyse (siehe Abbildung 6) isoliert also Sequenzen und Schleifen des Syntaxbaums und identifiziert Abhängigkeiten zwischen den enthaltenen Anweisungen. Im Architekturerkenner sind Algorithmen verankert die spezifizieren, wie Sequenzen auf Taskgraphen und Schleifen auf Fließbänder abzubilden sind. Die Überführung der erkannten Architekturen in die technische Ausgabeform erfolgt mittels einer Architekturbeschreibungssprache, auf die in Kapitel 4.9 eingegangen wird.

Wieso Befehle und Verzweigungen im Gegensatz zu Sequenzen und Schleifen nicht adressiert werden, wird im Folgenden erläutert:

- **Parallelisierung von Befehlen:** Ein Befehl ist eine atomare Anweisung, die auf Quellcodeebene in keine weiteren Anweisungen zerfällt. Ein Befehl ist also die kleinstmögliche Einheit und kann deshalb nicht auf Quellcodeebene parallelisiert werden. Allerdings können Übersetzer und Prozessoren Parallelität auszunutzen (siehe Kapitel 2.1), nachdem ein Befehl durch die Übersetzung in mehrere Zwischencode- oder Maschinenbefehle zerfallen ist.
- **Parallelisierung von Verzweigungen:** Eine Verzweigung besteht aus einer Bedingung und einer Menge von Zweigen, die selbst eine Sequenz von Anweisungen sind, und führt in Abhängigkeit einer Bedingung einen bestimmten Zweig aus. Parallelität kann durch die spekulative nebenläufige Vorausberechnung (engl. *thread level speculation, TLS*) [SC+00] aller Zweige eingeführt werden: noch bevor die Bedingung ausgewertet wird, wird die Ausführung sämtlicher Zweige angestoßen. Diese Art von Parallelität hat jedoch einen entscheidenden Nachteil: wenn der tatsächlich auszuführende Zweig nach der Auswertung der Bedingung schließlich feststeht, müssen die Zustandsänderungen der restlichen Zweige rückgängig gemacht werden. Spekulative Parallelität ist deshalb nur sinnvoll einsetzbar, wenn der Aufwand für das Zurückrollen der Zustandsänderungen sehr gering ist. Aus diesem Grund wird spekulative Parallelität in dieser Arbeit nicht weiter betrachtet.
- **Parallelisierung von Sequenzen:** Eine Sequenz besteht aus einer Folge von Elementen (Anweisungen) und führt diese zeitlich aufeinanderfolgend aus. Parallelität kann durch die zeitliche Überlappung einzelner Anweisungen eingeführt werden. Sequenzen werden in dieser Arbeit durch die Abbildung auf konfigurierbare Taskgraphen parallelisiert.
- **Parallelisierung von Schleifen:** Eine Schleife besteht aus einer Laufbedingung und einem Schleifenrumpf, der selbst eine Sequenz von Anweisungen ist. Die Schleife wird für mehrere Iterationen ausgeführt, bis die Laufbedingung zu falsch ausgewertet wird. Parallelität kann durch teilweise oder vollständige zeitliche Überlappung der Iterationen eingeführt werden. Schleifen werden in dieser Arbeit durch die Abbildung auf konfigurierbare Fließbänder parallelisiert.

4.5 Parallele Architekturmuster

Parallele Architekturmuster (orange) bieten eine vordefinierte Lösung für eine Klasse paralleler Probleme, indem sie eine bestimmte Parallelisierungsstrategie definieren [MS+04]. Ein paralleles Architekturmuster ist eine konfigurierbare Schablone, die ausführbare Komponenten (hellorange) und deren Kommunikationsbeziehung definiert. Eine ausführbare Komponente besitzt Platzhalter für Anweisungen (blau) und kann von einem Ausführungsfaden ausgeführt werden. Durch die Definition der Kommunikationsbeziehungen spricht ein paralleles Architekturmuster bestimmte Reihenfolgegarantien für die Anweisungen der Komponenten aus. Eine Konfiguration eines parallelen Architekturmusters kann durch die Festlegung seiner Parameter angegeben werden. In Abhängigkeit davon, ob ein Parameter die Reihenfolgegarantie für die Anweisungen der Komponenten beeinflusst muss er in der Architekturbeschreibung festgelegt werden oder kann alternativ implizit als Tuning-Parameter veräußert werden. Diese abstrakte Beschreibung paralleler Architekturmuster wird durch die folgenden Kapitel 4.5.1 und 4.5.2 konkretisiert, in denen das Taskgraph-Muster und das Fließband-Muster eingeführt

werden. Nach welchen Kriterien entschieden wird, wann eine Sequenz durch eine Konfiguration eines Taskgraphen und eine Schleife durch eine Konfiguration eines Fließbandes ersetzt werden, wird in Kapitel 4.8 erläutert, welches die Architekturerkennung diskutiert.

4.5.1 Taskgraph-Muster

Das Taskgraph-Muster beschreibt die in dieser Arbeit verwendete Parallelisierungsstrategie für eine Sequenz. Welche Kriterien zur der Entscheidung führen, ob eine Sequenz durch eine Konfiguration eines Taskgraphen ersetzt wird, und welche Konfiguration dies ist, wird in Kapitel 4.8.1 definiert.
Wenn – wie in einer Sequenz – eine Menge von Anweisungen aufeinanderfolgend ausgeführt werden soll, bietet der Taskgraph die Möglichkeit die Ausführung jeder Anweisung anzustoßen, sobald alle benötigten Eingabedaten vorliegen. Diese Art von Parallelität wird durch die ausführbare Komponente Task ermöglicht. An dieser Stelle sei noch einmal erwähnt, dass eine Anweisung ein Kompositum ist und selbst wiederum Anweisungen enthalten kann.

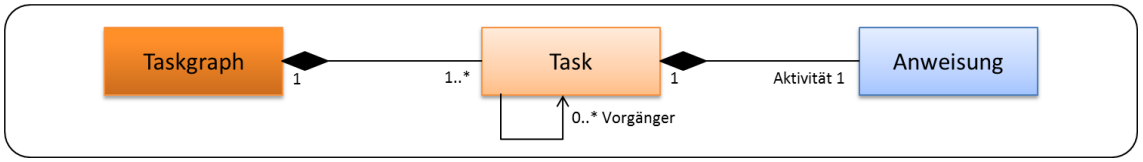

Abbildung 8: Klassendiagramm zur Veranschaulichung der Beziehungen zwischen Taskgraph, Task und Anweisung.

Ein Taskgraph ist ein paralleles Architekturmuster und setzt sich aus mehreren Tasks zusammen (Abbildung 8). Ein Task ist eine ausführbare Komponente, besitzt einen Platzhalter für eine Aktivität und kann von anderen Tasks, seinen Vorgängern, abhängig sein. Der Platzhalter Aktivität ist eine Menge zusammenhängender Anweisungen, die als Sequenz interpretiert wird.

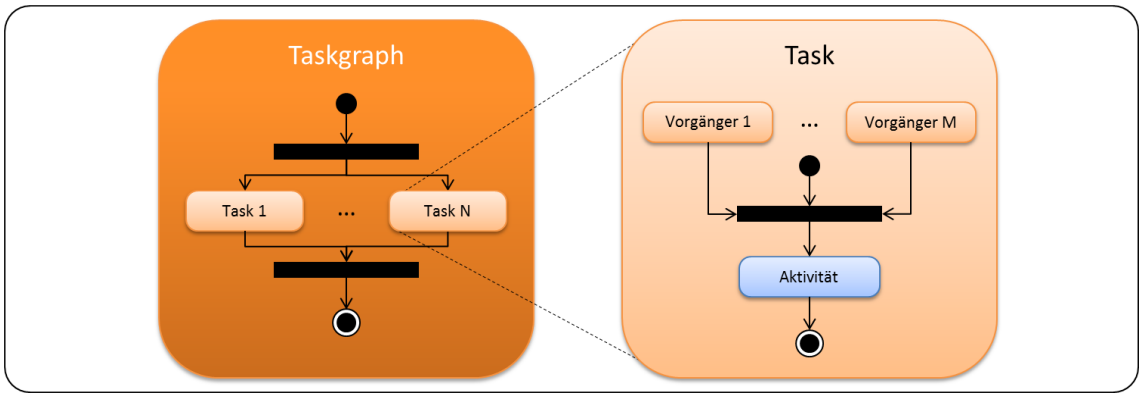

Abbildung 9: Aktivitätsdiagramme für das parallele Architekturmuster Taskgraph und die ausführbare Komponente Task.

Ein Taskgraph stößt die Ausführung der ihm zugeordneten Tasks nebenläufig an (engl. *fork*) und wartet, bis alle Tasks beendet sind (engl. *join*) (Abbildung 9). Ein Task kann von einem Faden (engl. *thread)* ausgeführt werden. Er wartet zunächst bis alle Vorgänger beendet sind und führt erst dann die ihm zugeordnete Aktivität (die Anweisungen) aus.

Ein Taskgraph definiert damit für jeden enthaltenen Task folgende Parameter:

- Aktivität des Tasks
- Vorgänger des Tasks
- Faden der den Task ausführt

Um eine konkrete Konfiguration eins Taskgraphen anzugeben müssen diese Parameter festgelegt werden. In Kapitel 4.6.1 wird untersucht, welche der Parameter in eine Architekturbeschreibung aufgenommen werden müssen und welche als implizite Tuning-Parameter veräußert werden können.

4.5.2 Fließband-Muster

Das Fließband-Muster beschreibt die in dieser Arbeit verwendete Parallelisierungsstrategie für eine Schleife. Die Kriterien zur Entscheidung darüber, ob eine Schleife durch eine Konfiguration eines Fließbandes ersetzt wird, und welche Konfiguration dies ist, werden in Kapitel 4.8.2 formuliert.

Wenn – wie in einer Schleife – auf eine Menge von Elementen mehrere aufeinanderfolgende Anweisungen angewendet werden sollen, bietet ein Fließband die Möglichkeit unterschiedliche Anweisungen auf unterschiedliche Elemente zur gleichen Zeit anzuwenden. Diese Art von Parallelität wird durch eine Reihe hintereinandergeschalteter nebenläufiger Komponenten, den Fließbandstufen, ermöglicht: die Elemente werden nacheinander von Stufe zu Stufe gereicht, wobei jede Stufe eine Anweisung anwendet. Eine spezielle Form des Fließbands (oft als *master-worker* bezeichnet) kann dazu verwendet werden, alle Iterationen einer Schleife parallel auszuführen.

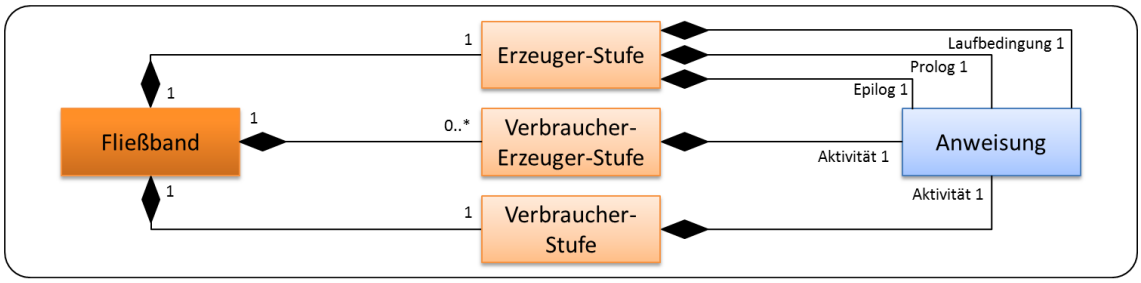

Abbildung 10: Klassendiagramm zur Veranschaulichung der Beziehungen zwischen Fließband, Erzeuger-Stufe, Erzeuger-Verbraucher-Stufe, Verbraucher-Stufe und Anweisung.

Ein Fließband besteht aus verschiedenen Fließbandstufen (Abbildung 10): einer Erzeuger-Stufe, beliebig vielen Verbraucher-Erzeuger-Stufen und einer Verbraucher-Stufe. Fließbandstufen besitzen Eingabe- und Ausgabeschnittstellen zum Empfangen und Senden von Elementen. Das Fließband gibt an, welche der Schnittstellen der Fließbandstufen miteinander verbunden sind. Der Austausch von Elementen zwischen verbundenen Ein- und Ausgabeschnittstellen wird über synchronisierte größenbeschränkte Puffer realisiert, die der Eingabeschnittstelle zugeordnet sind. Synchronisierte größenbeschränkte Puffer stellen sicher, dass es beim Einfügen und Entnehmen von Elementen nicht zu einem Wettlauf kommt, einem leeren Puffer keine Elemente entnommen werden und einem vollen Puffer keine weiteren Elemente hinzugefügt werden können.

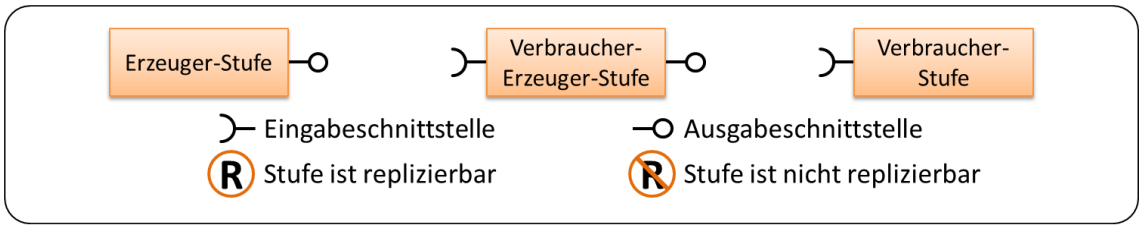

Abbildung 11: Eingabe- und Ausgabeschnittstellen der verschiedenen Fließbandstufen und Symbolik für replizierbare und nicht replizierbare Stufen.

Eine Erzeuger-Stufe besitzt Platzhalter für die Anweisungssequenzen Laufbedingung, Prolog und Epilog und eine Ausgabeschnittstelle (Abbildung 10, Abbildung 11). Eine Verbraucher-Erzeuger-Stufe besitzt einen Platzhalter für eine Aktivität, eine Eingabeschnittstelle und eine Ausgabeschnittstelle. Eine Verbraucher-Stufe ist analog zur Erzeuger-Verbraucher-Stufe aufgebaut, besitzt jedoch keine Ausgabeschnittstelle. Die Platzhalter Prolog, Epilog und Aktivität der Stufen sind Anweisungen oder Verkettungen von Anweisungen, die als Sequenz interpretiert werden. Während eine Erzeuger-Stufe nur durch einen Faden ausgeführt werden kann, ist es prinzipiell möglich eine Verbraucher-Erzeuger- oder Verbraucher-Stufe zu replizieren und mehrere Instanzen der Stufe zu erzeugen, wobei jeder Instanz ein eigener Faden zugeordnet ist. Dadurch kann zusätzlich zur Nebenläufigkeit zwischen verschiedenen Stufen auch Nebenläufigkeit zwischen verschiedenen Instanzen der gleichen Stufe erzeugt werden. Das angesprochene *master-worker*-Muster kann beispielsweise durch ein Fließband mit einer Erzeuger-Stufe und einer replizierten Verbraucher-Stufe umgesetzt werden.

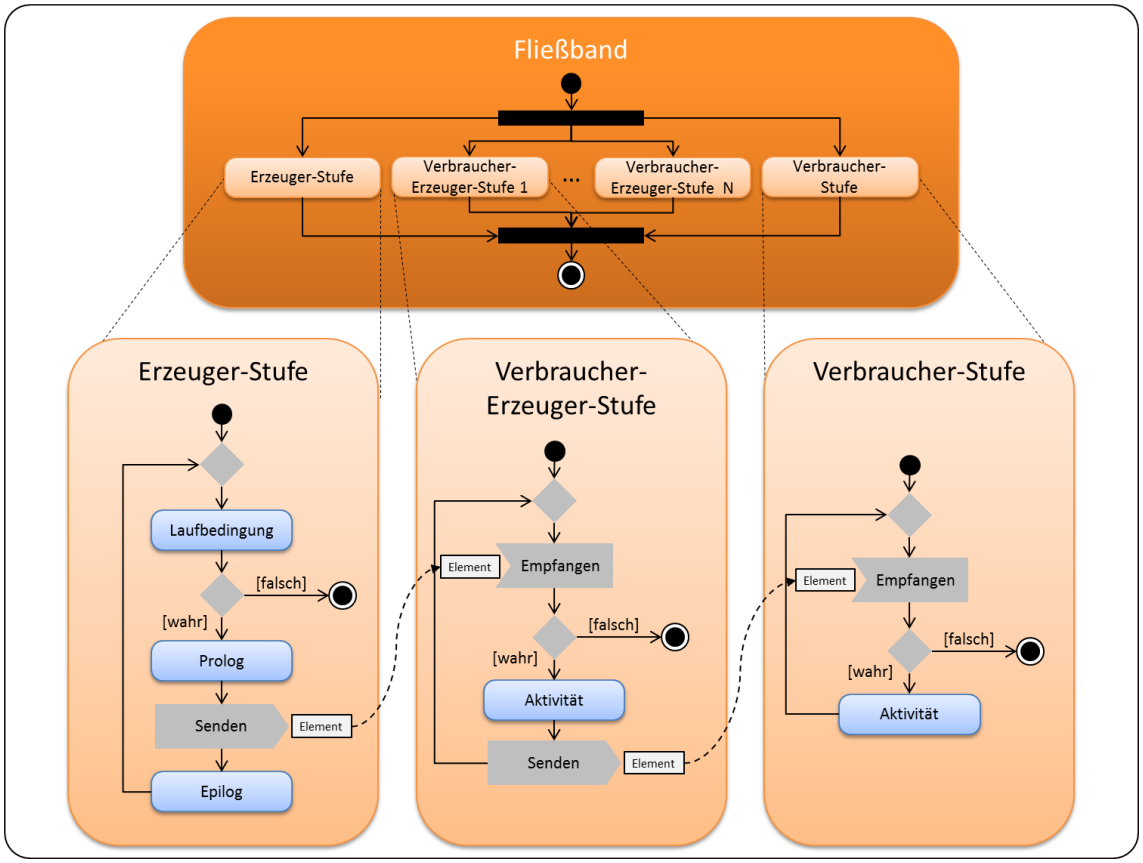

Abbildung 12: Aktivitätsdiagramme für das parallele Architekturmuster Fließband und die ausführbaren Komponenten Erzeuger-Stufe, Verbraucher-Erzeuger-Stufe und Verbraucher-Stufe.

Ein Fließband stößt die Ausführung der ihm zugeordneten Fließbandstufen an (engl. *fork*) und wartet, bis alle Stufen beendet sind (engl. *join*) (Abbildung 12).

Eine Erzeuger-Stufe dient zur Erzeugung eines Stroms aus Elementen, die durch das Fließband hindurchgereicht werden und verhält sich während der Ausführung wie eine `while`-Schleife (die abbricht, wenn die Laufbedingung falsch wird), in deren Schleifenrumpf zunächst der Prolog, dann das Senden und anschließend der Epilog ausgeführt wird. Beim Senden wird der Zustand der aktuellen Schleifeniteration in Form eines Elements über die Ausgabeschnittstelle verschickt. Die Platzhalter Prolog und Epilog bieten die Möglichkeit, vor oder nach dem Senden bestimmte Anweisungen in der Erzeuger-Stufe durchzuführen. Die Bedeutung dieser Platzhalter kann anschaulich bei der Parallelisierung einer `while`-Schleife hervorgehoben werden: `while(i < 10) { A(i); i++; }`. Die Erzeuger-Stufe soll zehn Aufgabenpakete (`i = 0` bis `i = 9`) an eine nachgelagerte Verbraucher-Stufe mit der Aktivität `A(i);` versenden. Die Erzeuger-Stufe bekommt dazu die Laufbedingung `i < 10`. Zur Inkrementierung der Laufvariablen `i`, wird die Anweisung `i++;` dem Epilog zugeordnet, der genau für solche Fälle eingeführt wurde.

Eine Verbraucher-Erzeuger-Stufe wartet zunächst auf den Empfang eines Elements über die Eingabeschnittstelle, stellt daraufhin den Zustand der Schleifeniteration wieder her und führt dann die Anweisungen der Aktivität aus. Anschließend wird der neue Zustand wieder in Form eines Elements über die Ausgabeschnittstelle weiterversendet. Dieser Vorgang wiederholt sich, bis der Strom aus Elementen beendet ist.
Eine Verbraucher-Stufe verhält sich analog zu einer Verbraucher-Erzeuger-Stufe, unterlässt jedoch das Weiterversenden von Elementen.

Ein Fließband definiert für jede enthaltene Stufe folgende Parameter:

- Laufbedingung, Prolog und Epilog der Erzeuger-Stufe
- Aktivität der Verbraucher-Erzeuger- oder Verbraucher-Stufe
- Nachfolgende Stufe
- Anzahl der Instanzen der Verbraucher-Erzeuger- oder Verbraucher-Stufe
- Für jede Instanz einer Stufe: Faden der die Stufe ausführt
- Für jede Instanz einer Stufe: Größe des Puffers der Eingabeschnittstelle

Um eine konkrete Konfiguration eins Fließbandes anzugeben müssen diese Parameter festgelegt werden. In Kapitel 4.6.2 wird untersucht, welche der Parameter in eine Architekturbeschreibung aufgenommen werden müssen und welche als implizite Tuning-Parameter veräußert werden können.

4.6 Tuning-Parameter paralleler Architekturmuster

Parameter paralleler Architekturmuster, deren Wert keinen Einfluss auf die Semantik hat, können implizit als Tuning-Parameter veräußert werden um die Anforderung der Optimierbarkeit (Kapitel 4.3) zu erfüllen. Ein Parameter hat Einfluss auf die Semantik, wenn er die Ausführungsreihenfolge von abhängigen sequentiellen Codeabschnitten bedingt. Die vor-

gestellten parallelen Architekturmuster Taskgraph und Fließband werden in den Kapiteln 4.6.1 und 4.6.2 auf solche Tuning-Parameter untersucht.

Unabhängig von der Art des parallelen Architekturmusters existieren zwei Tuning-Parameter, die für jeden Vorschlag implizit definiert sind: Der erste Tuning-Parameter `runParallel` ist die Entscheidung darüber, ob der zugrundeliegende Codeabschnitt parallel im Sinne der Architekturbeschreibung oder doch sequentiell ausgeführt wird. Der Grund für diesen Parameter ist folgender: Eine Architekturbeschreibung enthält nur die Information, dass eine parallele Architektur möglich ist, nicht jedoch ob sie auch eine Verringerung der Laufzeit nach sich zieht. Es ist unter Umständen sinnvoller, die ursprüngliche sequentielle Variante der parallelen vorzuziehen. Der zweite Tuning-Parameter, der für jede Methode eingeführt wird, die durch einen Parallelisierungsvorschlag transformiert wird ist `maxRecursionDepth`. Dieser Parameter gibt an, bis zu welcher maximalen Rekursionstiefe der Methode die parallele Codealternative ausgeführt wird und ab welcher Tiefe auf die ursprüngliche sequentielle Variante umgeschaltet wird. Dieser Parameter ist wichtig, wenn eine rekursive Methode durch einen Vorschlag parallelisiert wird, da sich die Parallelität ab einer gewissen Rekursionstiefe möglicherweise nicht mehr lohnt.

4.6.1 Taskgraph

Die Parameter eines Taskgraphen werden im Folgenden der Reihe nach darauf untersucht, ob sie sich als Tuning-Parameter eignen:

- Aktivität des Tasks: Weil sich ein Task über seine Aktivität definiert (Komposition), muss die Liste der Anweisungen der Aktivität explizit angegeben werden.
- Vorgänger des Tasks: Die Liste der Vorgänger eines Tasks hat Einfluss darauf, auf welche anderen Tasks ein Task wartet, bevor er mit der Ausführung seiner Aktivität beginnt. Die Liste der Vorgänger wird dazu genutzt, die Ausführungsreihenfolge von Tasks mit abhängigen Aktivitäten sicherzustellen und scheidet damit als Tuning-Parameter aus.
- Faden der den Task ausführt: Während die maximal mögliche Nebenläufigkeit erzielt wird, wenn jeder Task durch einen eigenen Faden ausgeführt wird, kann es sinnvoll sein Tasks zusammenzufassen und nur durch einen gemeinsamen Faden auszuführen. Durch das Zusammenfassen von Tasks kann der Mehraufwand (engl. *overhead*) der Erzeugung von Fäden eingespart werden.

Für je zwei Tasks eines Taskgraphen ergibt sich damit ein Tuning-Parameter (`taskFusion`), der angibt, ob diese Tasks in einem Faden zusammengefasst werden sollen oder nicht. Dabei ist zu beachten, dass Tasks in beliebiger Reihenfolge zusammengefasst werden können, falls über die Vorgängerbeziehungen keine Reihenfolge definiert ist. Ansonsten ist die Zusammenfassung in der definierten Reihenfolge vorzunehmen.

Beispiel: Ein Taskgraph mit drei Tasks ohne Vorgänger (der später durch die Architekturbeschreibung `T1 || T2 || T3`) angegeben wird, hat folgende Tuning-Parameter: `runParallel` (gibt an, ob die parallele Variante als Taskgraph oder ursprüngliche sequentielle Variante ausgeführt wird), `maxRecursionDepth` (gibt an, bis zu welcher rekursionstiefe der Methode, in der sich der Taskgraph befindet, die parallele Variante als Taskgraph ausgeführt wird und ab welcher Tiefe die ursprüngliche sequentielle Variante), `taskFusion(T1, T2)`

(gibt an, ob die Tasks `T1` und `T2` zusammengefasst und von einem Faden ausgeführt werden sollen), `taskFusion(T1, T3)` und `taskFusion(T2, T3)`.

4.6.2 Fließband

Die Parameter eines Fließbandes werden im Folgenden der Reihe nach darauf untersucht, ob sie sich als Tuning-Parameter eignen:

- Laufbedingung der Stufe, Prolog der Stufe, Epilog der Stufe und Aktivität der Stufe: Weil sich eine Stufe über diese Platzhalter definiert (Kompositum), müssen sie explizit angegeben werden.
- Nachfolgende Stufe: Über die Verknüpfungsreihenfolge von Stufen wird offensichtlich auch die Ausführungsreihenfolge der Anweisungen ihrer Platzhalter festgelegt. Weil aufeinanderfolgende Stufen über die ausgetauschten Elemente voneinander abhängig sind, darf dieser Parameter nicht frei gewählt werden und scheidet damit als Tuning-Parameter aus.
- Faden der die Stufe ausführt: Während die maximal mögliche Nebenläufigkeit erzielt wird, wenn jede Stufe und jede Instanz einer Stufe von einem eigenen Faden ausgeführt wird, kann es sinnvoll sein, Stufen zusammenzufassen und durch einen gemeinsamen Faden auszuführen. Durch das zusammenfassen von Stufen kann der Mehraufwand der Erzeugung von Fäden und der Mehraufwand der Kommunikation über Puffer eingespart werden.
- Anzahl der Instanzen einer Stufe: Während die maximal mögliche Nebenläufigkeit erzielt wird, wenn die Anzahl der Instanzen maximal ist, kann die Ausführung der Aktivität der Stufe für zwei aufeinanderfolgende Elemente abhängig sein. Dann ist die Stufe nicht replizierbar und die Anzahl der Instanzen fest auf eins zu setzen. Um Flüchtigkeitsfehler zu vermeiden wird eine Stufe standardmäßig als nicht replizierbar angenommen. Die Replizierbarkeit einer Stufe muss durch die Architekturbeschreibung explizit signalisiert werden. Für eine replizierbare Stufe kann es wiederum sinnvoll sein, die Anzahl der Instanzen zu beschränken, um wie schon zuvor erläutert den Mehraufwand durch die Erzeugung von Fäden und die Kommunikation über Puffer zu reduzieren.
- Größe des Puffers der Eingabeschnittstelle: Unabhängig von der Größe des Puffers werden die Elemente, die durch ein Fließband strömen, in der gleichen Reihenfolge abgearbeitet. Die Größe kann damit frei gewählt werden. Die Puffergröße kann die Laufzeit des Fließbandes maßgeblich beeinflussen, weil sie eine Stellschraube für die Obergrenze des von den Elementen belegten Speichers darstellt.

Für je zwei benachbarte Stufen eines Fließbandes ergibt sich damit ein Tuning-Parameter (`stageFusion`), der angibt, ob diese Stufen in einem Faden zusammengefasst werden sollen oder nicht. Dabei ist zu beachten, dass benachbarte Stufen in ihrer definierten Verknüpfungsreihenfolge zusammengefasst werden. Jede replizierbare Stufe besitzt einen Tuning-Parameter (`numThreads`), die die konkrete Anzahl an zu erzeugenden Instanzen angibt. Jede Stufe besitzt zudem einen Tuning-Parameter (`bufferSize`) für die Größe des Puffers der Eingabeschnittstelle.

Beispiel: Ein Fließband mit einer impliziten Erzeuger-Stufe, einer Verbraucher-Erzeuger-Stufe und einer replizierbaren Verbraucher-Stufe (das später durch die Architekturbeschreibung `S1 => S2+`) angegeben wird, hat folgende Tuning-Parameter: `runParallel` (gibt an, ob die parallele Variante als Fließband oder ursprüngliche sequentielle Variante ausgeführt wird), `maxRecursionDepth` (gibt an, bis zu welcher rekursionstiefe der Methode, in der sich das Fließband befindet, die parallele Variante als Fließband ausgeführt wird und ab welcher Tiefe die ursprüngliche sequentielle Variante), `stageFusion(S0, S1)` (gibt an, ob die implizite Erzeuger-Stufe `S0` und die Stufe `S1` zusammengefasst und von einem Faden ausgeführt werden sollen), `stageFusion(S1, S2), bufferSize(S0, S1)` (gibt an, auf welche Anzahl an Elementen der Puffer zwischen `S0` und `S1` maximal anwachsen darf), `bufferSize(S1, S2)` und `numThreads(S2)` (gibt an, mit wie vielen Ausführungsfäden die Stufe S2 ausgeführt werden soll).

4.7 Abhängigkeitsanalyse

Nachdem die parallelen Architekturmuster vorgestellt wurden, die in dieser Arbeit verarbeitet werden sollen, wird in diesem Kapitel darauf eingegangen, wie Abhängigkeiten zwischen Anweisungen identifiziert werden, die bei der Erkennung der Architekturen berücksichtigt werden müssen. Zur Veranschaulichung werden die in diesem Kapitel eingeführten Analysen im Kapitel 4.7.6 auf ein alltagsnahes Beispiel angewandt, das in Kapitel 4.7.5 eingeführt wird.

Kontrollfluss- und Datenabhängigkeiten sind die Ursache für die kausale Abhängigkeit von Anweisungen und müssen daher bei der Abbildung von sequentiellen Codeabschnitten auf parallele Architekturmuster berücksichtigt werden. Deren Definition scheint an dieser Stellt noch einmal angebracht:

Kontrollflussabhängigkeit: Eine Anweisung A_2 ist kontrollflussabhängig von einer Anweisung A_1, wenn A_1 vor A_2 ausgeführt wird und das Resultat von A_1 darüber entscheidet, ob A_2 ausgeführt wird oder nicht.

Datenabhängigkeit: Eine Anweisung A_2 ist datenabhängig von einer Anweisung A_1, wenn A_1 vor A_2 ausgeführt wird, beide Anweisungen auf die gleiche Speicherstelle s zugreifen und mindestens eine der Anweisungen auf s schreibt. Drei Arten von Datenabhängigkeiten werden unterschieden:

- Read-after-Write (RAW): A_2 liest s nachdem A_1 s geschrieben hat.
- Write-after-Read (WAR): A_2 schreibt s nachdem A_1 s gelesen hat.
- Write-after-Write (WAW): A_2 schreibt s nachdem A_1 s geschrieben hat.

Die Aufgabe der Identifizierung dieser Abhängigkeiten kommt der Abhängigkeitsanalyse zu. Sequenzen und Schleifen werden zu Beginn der Abhängigkeitsanalyse im Syntaxbaum lokalisiert und im weiteren Verlauf um Kontrollfluss- und Datenabhängigkeiten ergänzt. Abhängigkeiten sind nur zwischen denjenigen Anweisungen interessant, die durch den Architekturerkenner auf Nebenläufigkeit untersucht werden. Für Sequenzen sind dies Abhängigkeiten zwischen den Anweisungen der Sequenz. Für Schleifen sind dies Abhängigkeiten zwischen der Laufbedingung der Schleife und den Anweisungen den Schleifenrumpfs.

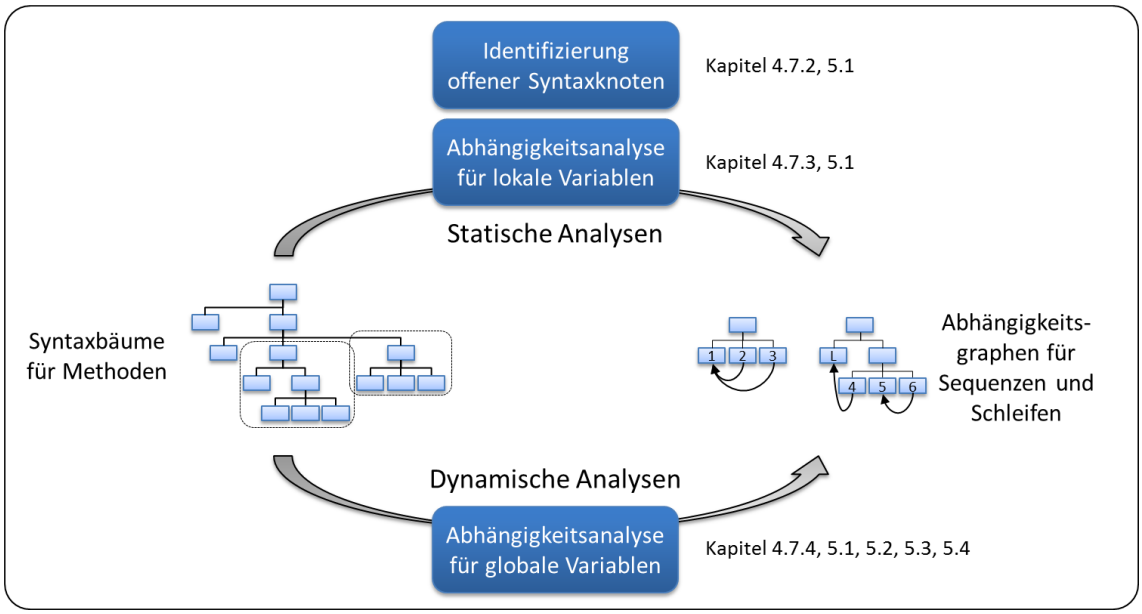

Abbildung 13: Die Abhängigkeitsanalyse im Gesamtüberblick: dargestellt sind die drei Teilanalysen.

Die Abhängigkeitsanalyse besteht aus zwei statischen und einer dynamischen Analyse (Abbildung 13):

- **Identifizierung offener Syntaxknoten:** Diese Analyse markiert unter den relevanten Anweisungen diejenigen, die für Kontrollflussabhängigkeiten verantwortlich sind: offene Syntaxknoten. Details sind in Kapitel 4.7.2 vorzufinden.
- **Abhängigkeitsanalyse für lokale Variablen:** Diese Analyse ermittelt Datenabhängigkeiten zwischen den relevanten Anweisungen, die durch Zugriffe auf lokale Variablen zustande kommen. Die zugehörigen Algorithmen werden in Kapitel 4.7.3 vorgestellt.
- **Abhängigkeitsanalyse für globale Variablen:** Diese Analyse ermittelt Datenabhängigkeiten zwischen den relevanten Anweisungen, die durch Zugriffe auf Feldelemente, Klassen- und Instanzvariablen zustande kommen. Auf den konzeptionellen Teil der Analyse wird in Kapitel 4.7.4 eingegangen. Wie die einzelnen Bestandteile Instrumentierung (Kapitel 5.1 und 5.2), Protokollierung und Nachverarbeitung (Kapitel 5.3) umgesetzt wurden wird in den Implementierungskapiteln klarer. Auf das Zusammenspiel der drei Komponenten wird anhand eines abstrakten Beispiels in Kapitel 5.4 eingegangen.

4.7.1 Auftrennung der Datenabhängigkeitsanalyse

Die Auftrennung der Datenabhängigkeitsanalyse in eine statische Analyse für lokale Variablen und eine dynamische Analyse für globale Variablen, soll im Folgenden motiviert werden: Der Basisalgorithmus (siehe Abbildung 14) zur Ermittlung für die Datenabhängigkeiten einer Anweisung leitet sich direkt aus der Definition für Datenabhängigkeiten ab.

```
 1 Seien LW(s) und LR(s) für alle s aus S global definiert und korrekt belegt.
 2
 3 ErmittleDatenabhängigkeiten(A, R(A), W(A)) {
 4   Für alle lokalen Variablen r aus R(A) {
 5     Wenn LW(r) nicht leer ist {
 6       A ist RAW-abhängig von LW(r)
 7     }
 8     LR(r) = LR(r) vereinigt A
 9   }
10   Für alle lokalen Variablen w aus W(A) {
11     Wenn LR(w) nicht leer ist {
12       Für alle Anweisungen LR aus LR(w) {
13         A ist WAR-abhängig von LR
14       }
15     }
16     LR(w) = leer
17     Wenn LW(w) nicht leer ist {
18       A ist WAW-abhängig von LW(w)
19     }
20     LW(w) = A
21   }
22 }
```

Abbildung 14: Basisalgorithmus zur Identifizierung von Datenabhängigkeiten einer Anweisung A bei gegebenen Lese- und Schreibmengen R(A) und W(A).

Zur Durchführung des Algorithmus aus Abbildung 14 ist S die Menge aller Speicherstellen, $R(A) \subseteq S$ die Menge an Speicherstellen, die von der Anweisung A gelesen werden, die $W(A) \subseteq S$ die Menge von Speicherstellen, die von der Anweisung A geschrieben werden. Zudem ist LW(s) die Anweisung, welche die Speicherzelle $s \in S$ zuletzt geschrieben hat und LR(s) die Menge von Anweisungen, welche die Speicherstelle $s \in S$ seit dem letzten Schreibzugriff gelesen hat.

Die Ermittlung von Datenabhängigkeiten anhand des Quellcodes ist nicht trivial, weil Anweisungen im Allgemeinen nicht direkt, sondern über zwei Indirektionen auf Speicherstellen zugreifen:

- Anweisungen können über Objektzeiger auf die Instanzvariablen von Objekten zugreifen. Die Zuordnung von Bezeichner im Quellcode und Speicherstelle zur Laufzeit ist daher nicht eindeutig. Lese- und Schreibmengen lassen sich nicht direkt aus dem Quellcode ablesen. Beispiel: Seien `a` und `b` Zeiger auf Instanzen der gleichen Klasse, die eine Instanzvariable `x` deklariert. Für die Anweisungen `a.x++` und `b.x++` ist dann nicht aus dem Quellcode ersichtlich, ob beide auf die gleiche Speicherstelle schreiben (falls `a` und `b` auf die gleiche Instanz zeigen) und damit abhängig sind, oder nicht.
- Anweisungen können dynamische Methodenaufrufe enthalten, deren Ziel über Objektzeiger bestimmt wird. Die Zuordnung von Methodenaufruf im Quellcode und Ziel des Aufrufs zur Laufzeit ist daher nicht eindeutig. Für Anweisungen mit Methodenaufrufen ist dadurch nicht klar, welche Anweisungen durch den Methodenaufruf ausgeführt werden. Es ist insbesondere ungewiss, welche Anweisung auf eine Anweisung mit Methodenaufruf folgt. Beispiel: Sei `B` eine Klasse, die von `A` erbt und deren Methode `foo()` überschreibt. Sei `a` ein Zeiger auf Instanzen der Klasse `A`, dann nicht aus dem Quellcode ersichtlich, ob `a.foo()` die Methode `foo()` der Klasse `A` oder die Methode `foo()` der Klasse `B` ausführt, wenn nicht klar ist, ob `a` auf eine Instanz von `A` oder von `B` zeigt.

Statische Analysen ermitteln Datenabhängigkeiten durch symbolische Ausführung von Anweisungen und verwenden Zeigeranalysen um mit den genannten Indirektionen umzugehen (siehe Kapitel 2.3.1). Statische Analysen können jedoch nur eine Überapproximation der tatsächlichen Datenabhängigkeiten liefern [C03]: es können falschpositive Datenabhängigkeiten ermittelt werden, die während der tatsächlichen Ausführung niemals auftreten können. Im Verlauf der statischen Analyse hat eine falschpositive Datenabhängigkeit gravierende Folgen: sie kann eine Vielzahl von weiteren falschpositiven Datenabhängigkeiten nach sich ziehen. Falschpositive Datenabhängigkeiten verhindern mögliche Parallelität [KK+10] und sind daher nicht mit der vierten Anforderung an das Konzept (siehe Kapitel 4.2) zu vereinen. Bei Datenabhängigkeiten durch lokale Variablen spielen die genannten Indirektionen keine Rolle. Falschpositive Abhängigkeiten bleiben überschaubar und pflanzen sich nicht über Methodengrenzen hinweg fort. Für diese Datenabhängigkeiten wird daher eine statische Analyse eingesetzt.

Dynamische Analysen führen die Anwendung tatsächlich aus, protokollieren Speicherzugriffe und leiten daraus Datenabhängigkeiten ab. Sie liefern nur Datenabhängigkeiten, die während der Ausführung tatsächlich auftreten und führen daher nicht zu falschpositiven Abhängigkeiten. Dynamische Analysen können jedoch nur eine Unterapproximation der tatsächlichen Datenabhängigkeiten liefern: es können Datenabhängigkeiten fehlen, die während der Ausführung mit anderen Eingaben für die Anwendung auftreten können. Dieser Umstand führt dazu, dass generierte Parallelisierungsvorschläge mit anderen Eingaben zu Fehlverhalten führen können. Dieses Problem wird durch die nachgelagerte Werkzeugkette angegangen: Fehlerbehaftete Parallelisierungsvorschläge können durch automatisch generierte Tests [SM+13] und einen Wettlauferkenner identifiziert und ausgesondert werden. Um die unnötige Ablehnung von Parallelisierungsvorschlägen durch die Überapproximation von Abhängigkeiten zu vermeiden wird für globale Variablen (im Kontext von C# also Feldelementen, Klassen- und Instanzvariablen) daher eine dynamische Analyse eingesetzt.

4.7.2 Identifizierung offener Syntaxknoten

Syntaxknoten, also (aufgrund der Grammatik der Programmiersprache) zusammenhängende Codeabschnitte, können in zwei Klassen eingeteilt werden: offene und abgeschlossene Syntaxknoten. Ein offener Syntaxknoten ist ein Knoten, der einen Sprungbefehl enthält, dessen Sprungziel außerhalb des Syntaxknoten liegt. Für einen offenen Knoten wird zudem angegeben, durch welchen Sprungbefehl er offen ist. Ein abgeschlossener Syntaxknoten enthält entweder keinen Sprungbefehl, oder für alle enthaltenen Sprungbefehle auch die zugeordneten Sprungziele. Der Algorithmus zur Identifizierung offener Syntaxknoten ist in Abbildung 15 angegeben.

```
 1 IstOffen(Syntaxknoten K) {
 2   Für alle durch Abstieg von K erreichbaren Sprungbefehle S {
 3     Wenn S eine return-Anweisung ist {
 4       Wenn es beim Aufstieg von S nach K keine Methodendeklaration gibt {
 5         K ist offen durch return
 6       }
 7     }
 8     Wenn S eine break Anweisung ist {
 9       Wenn es beim Aufstieg von S nach K keine Schleifendeklaration und
10       keine switch-Verzweigung gibt {
11         K ist offen durch break
12       }
13     }
14     Wenn S eine continue-Anweisung ist {
15       Wenn es beim Aufstieg von S nach K keine Schleifendeklaration gibt {
16         K ist offen durch continue
17       }
18     }
19   }
20   K ist abgeschlossen
21 }
```

Abbildung 15: Algorithmus zur Identifizierung offener Syntaxknoten.

Die Sprungziele der Sprungbefehle `return`, `break` und `continue` sind implizit über den Syntaxbaum definiert: `return` springt aus der zugeordneten Methode, `break` aus der zugeordneten Schleife oder der zugeordneten `switch`-Verzeweigung und `continue` zur nächsten Iteration der zugeordneten Schleife. Die zugeordnete Methode/Schleife/Verzweigung wird über das Aufsteigen im Syntaxbaum vom Syntaxknoten des Sprungbefehls bis zum ersten Vorkommen einer Methode/Schleife/Verzweigung gefunden.

```
1 if(…) { continue; }
2 if(…) { A; break; }
3 for(…) { A; if(…) { return; } B; }
```

```
4 for(…) { A; B; C; }
5 for(…) { A; if(…) break; B; }
6 var a = delegate() { A; return; }
```

Abbildung 16: Die Zeilen 1 bis 6 stellen jeweils einen Syntaxknoten dar. Links: Beispiele für offene Syntaxknoten, rechts: Beispiele für abgeschlossene Syntaxknoten.

Beispiele für offene und abgeschlossene Syntaxknoten sind in Abbildung 16 gegeben, *A*, *B* und *C* sind Anweisungen beliebiger Ausprägung. Eine Zeile entspricht einem Syntaxknoten: 1 ist offen durch `continue`, 2 ist offen durch `break` und 3 durch `return`. Die Syntaxknoten 4, 5 und 6 sind abgeschlossen, 5 und 6 enthalten aber wiederum offene Syntaxknoten (`if(…) break;` und `return;`)

Offene Syntaxknoten müssen aus zwei Gründen berücksichtigt werden:

- Offene Syntaxknoten sind die Ursache von Kontrollabhängigkeiten. In einer Sequenz von Syntaxknoten sind alle Syntaxknoten, die auf einen offenen Knoten folgen vom offenen Knoten kontrollabhängig, weil der offene Knoten darüber entscheidet, ob die folgenden Anweisungen ausgeführt oder übersprungen werden. In Abbildung 16, Zeile 3 ist die Anweisung *B* kontrollflussabhängig von `if(…) { return; }`. Eine Missachtung dieser Abhängigkeit würde dazu führen, dass `A` und `B` parallel ausgeführt werden können, was jedoch die Semantik der Anwendung ändert: `B` darf überhaupt erst ausgeführt werden, wenn sicher ist, dass nicht durch `return` aus der Methode gesprungen wird.

- Die Aktivität von Tasks und Fließbandstufen darf keine offenen Syntaxknoten enthalten. Die Begründung dazu folgt: Bei der Transformation in parallelen Quellcode wird die Aktivität von Fließbandstufen von einer neue Schleife umschlossen (Abbildung 12), zudem wird diese neue Schleife, wie auch die Aktivität eines Tasks von einer neuen Methode umschlossen. Da offene Syntaxknoten Sprungbefehle, aber nicht deren implizit zugeordnetes Sprungziel enthalten, würde das implizit zugeordnete Sprungziel dadurch verändert: offene Syntaxknoten durch `return` würden aus der neuen Methode springen, nicht aber aus der ursprünglichen. Entsprechende Überlegungen gelten für offene Syntaxknoten durch `break` und `continue`. Eine Ausnahme bilden Prolog und Epilog der Erzeuger-Stufe: sie dürfen offene Syntaxknoten durch `break` oder `continue` enthalten, weil diese Syntaxknoten bei der Abbildung von Schleifen auf Fließbänder berücksichtigt werden und die Erzeuger-Stufe dabei einen Ersatz für die ursprüngliche Schleife darstellt.

4.7.3 Abhängigkeitsanalyse für lokale Variablen

Datenabhängigkeiten durch Zugriffe auf lokale Variablen werden in dieser Arbeit statisch bestimmt, weil keine der zwei genannten Indirektionen (4.7.1) berücksichtigt werden müssen: lokale Variablen sind nicht über Objektzeiger zugreifbar, wodurch die Zuordnung von lokalen Variablen im Quelltext und Speicherstellen zur Laufzeit eindeutig ist. Die Lese- und Schreibmengen einer Anweisung können direkt aus dem Syntaxbaum abgelesen werden. Zudem sind lokale Variablen nur innerhalb einer Methode gültig: Methodenaufrufe müssen nicht verfolgt werden, weil die sich dahinter verbergenden Anweisungen nicht auf die lokalen Variablen zugreifen können. Zur Veranschaulichung wird der im Folgenden vorgestellte Algorithmus in Kapitel 4.7.6 in Auszügen auf das alltagsnahe Beispiel angewandt.

Datenabhängigkeiten durch Zugriffe auf lokale Variablen für eine isolierte Sequenz $A_{1->N}$ = (A_1, …, A_N) mit den aufeinanderfolgenden Anweisungen A_1 bis A_N können ermittelt werden, indem die Anweisungen symbolisch nacheinander ausgeführt werden und dabei der Basisalgorithmus zur Ermittlung von Datenabhängigkeiten aus Abbildung 14 durchgeführt wird.

```
 1 Für alle lokalen Variablen v aus V {
 2   LW(v) = leer
 3   LR(v) = leer
 4 }
 5
 6 Für alle Anweisungen A aus A1->N {
 7   R(A) = alle lokalen Variablen, die von der Anweisung A (oder ihren Kindern
 8          im Syntaxbaum) gelesen werden.
 9   W(A) = alle lokalen Variablen, die von der Anweisung A (oder ihren Kindern
10          im Syntaxbaum) geschrieben werden.
11   ErmittleDatenabhängigkeiten(A, R(A), W(A));
12 }
```

Abbildung 17: Algorithmus zur Bestimmung von Datenabhängigkeiten durch Zugriffe auf lokale Variablen für eine Sequenz A_{1-N}.

Für eine Schleife (L, $A_{1->N}$) mit der Laufbedingung L und dem Schleifenrumpf $A_{1->N}$ = (A_1, …, A_N) aus den aufeinanderfolgenden Anweisungen A_1 bis A_N kann der Algorithmus aus Abbildung 17 erweitert werden: Da eine Anweisung in jeder Schleifeniteration die gleichen lokalen Variablen liest und schreibt, kann die Definition der Lese- und Schreibmengen beibehalten werden. Es werden zwei Schleifeniterationen simuliert, indem (L, $A_{1->N}$) und (L, $A_{1->N}$) symbolisch nacheinander ausgeführt werden, ohne LW(v) und LR(v) dazwischen

zurückzusetzen. Wegen gleichbleibender Lese- und Schreibmengen führt die Simulation einer dritten Schleifeniteration zu keinen neuen Abhängigkeiten. Um Abhängigkeiten über Iterationsgrenzen hinweg zu erkennen, werden LW(v) und LR(v) erweitert: zusätzlich zur Anweisung wird die Iteration gespeichert. Vor der Ausgabe einer Abhängigkeit zwischen zwei Anweisungen wird überprüft, ob beide Anweisungen zur gleichen Iteration gehören oder nicht. Falls nicht, wird die Abhängigkeit aus iterationsübergreifend markiert.

Durch die Definition der Lese- und Schreibmengen R(A) und W(A) können die vorgestellten Algorithmen zu Überapproximationen führen, wenn eine Anweisung bedingte Anweisungen enthält. Diese Überapproximationen werden in dieser Arbeit jedoch bewusst in Kauf genommen.

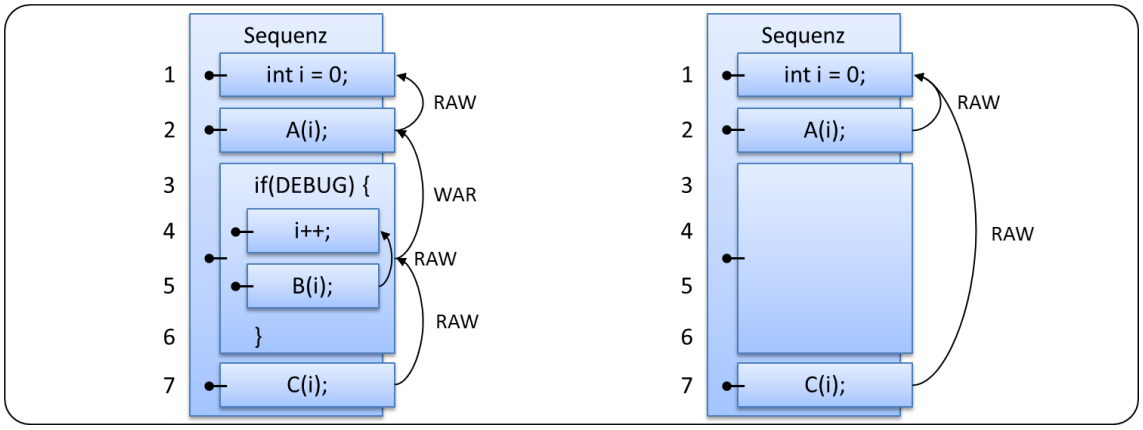

Abbildung 18: Links: resultierende Abhängigkeiten durch lokale Variablen. Rechts: tatsächliche Abhängigkeiten, wenn DEBUG immer zu falsch ausgewertet wird.

Ein Beispiel für eine Überapproximation ist in Abbildung 18 gegeben: Links sind die resultierenden Abhängigkeiten des Algorithmus aus Abbildung 17 angegeben, rechts die tatsächlichen, wenn `DEBUG` zur Laufzeit immer zu falsch ausgewertet wird. Dann werden die Zeilen 4 und 5 nie ausgeführt: Die Zeilen 2 und 7 könnten parallel abgearbeitet werden. Die rechts dargestellte RAW-Abhängigkeit von der Zeile 7 zur Zeile 1 ist links nicht explizit dargestellt. Sie ist jedoch implizit über die transitive Hülle definiert: weil 7 von (3-6) abhängig ist, (3-6) von 2 und 2 von 1, ist auch 7 von 1 abhängig.

4.7.4 Abhängigkeitsanalyse für globale Variablen

Datenabhängigkeiten durch Zugriffe auf Feldelemente, Klassen- und Instanzvariablen werden in dieser Arbeit dynamisch bestimmt, weil Überapproximationen durch statische Analysen an dieser Stelle nicht akzeptabel sind. Der Grundgedanke dynamischer Analysen ist einleuchtend: weil die Anwendung tatsächlich ausgeführt wird, stellen die zwei genannten Indirektionen (Kapitel 4.7.1) kein Problem dar: zur Laufzeit ist ein Bezeichner genau einer Speicherstelle zugeordnet. Auch die Ziele von Methodenaufrufen sind eindeutig.

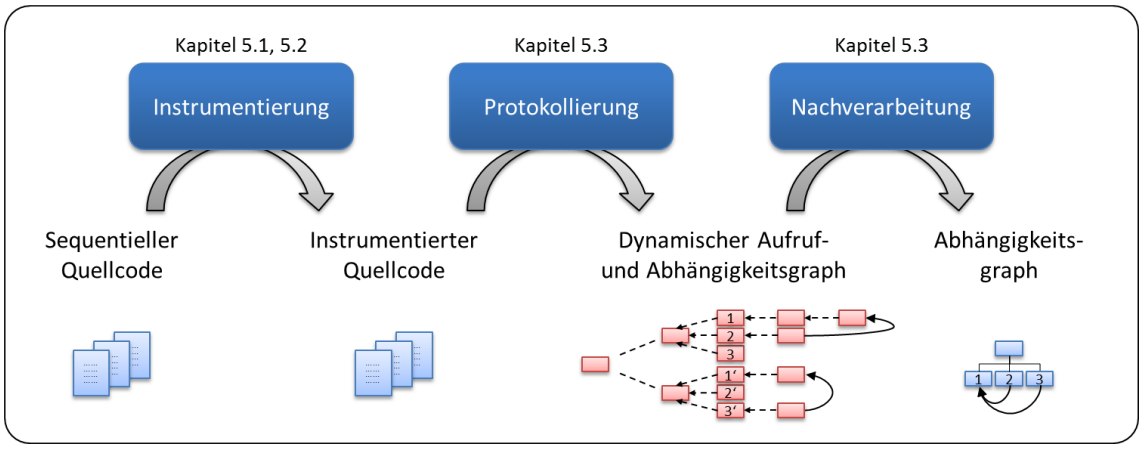

Abbildung 19: Die Abhängigkeitsanalyse für globale Variablen im Gesamtüberblick: dargestellt sind die drei Phasen und die ausgetauschten Datenformate.

Um Datenabhängigkeiten dynamisch zu ermitteln wird der Basisalgorithmus aus Abbildung 14 in abgewandelter Form während der tatsächlichen Ausführung der Anwendung im Hintergrund mit ausgeführt. Die Abhängigkeitsanalyse für globale Variablen besteht aus drei Phasen (Abbildung 19): Instrumentierung, Protokollierung und Nachverarbeitung. Im Folgenden wird nur konzeptionell auf die drei Phasen eingegangen, Details finden sich in den Implementierungskapiteln 5.1, 5.2 und 5.3. Das Zusammenspiel von Instrumentierung, Protokollierung und Nachverarbeitung wird anhand eines Beispiels in Kapitel 5.4 durchexerziert.

- **Instrumentierung:** Bei der Instrumentierung werden der sequentiellen Anwendung Signalisierungsanweisungen hinzugefügt, auf die während der Protokollierung reagiert wird. Die Instrumentierung erfolgt nach zwei Regeln.
  Die erste Regel wird durch die Instrumentierung von Quellcode realisiert, deren Umsetzung ist in Kapitel 5.1 vorzufinden. Die zweite Regel wird durch die Instrumentierung von Zwischencode umgesetzt, welche in Kapitel 5.2 beschrieben ist. Zur Auftrennung der Instrumentierung in zwei Teilabschnitte wird in Kapitel 5.2.1 Stellung genommen.
  - Jede Anweisung A, die Datenabhängigkeiten durch Zugriffe auf Klassen- oder Instanzvariablen verursachen kann, wird von einer Eintrittsanweisung `Enter(A)` und von einer Austrittsanweisung `Leave(A)` umschlossen, wobei für `A` ein Bezeichner in Form einer eindeutigen Zahl (später `SyntaxNodeId`) eingeführt wird. `Enter(A)` signalisiert, dass mit der Ausführung der Anweisung A begonnen wird, `Leave(A)` signalisiert, dass die Ausführung der Anweisung A abgeschlossen ist. Zudem wird zur Erkennung von Abhängigkeiten zwischen Iterationen einer Schleife S dem Schleifenrumpf von S die Iterationsanweisung `EnterIteration(S)` vorangestellt. Die Anweisungen `Enter()`, `Leave()` und `EnterIteration()` werden als Abgrenzungsanweisungen bezeichnet, weil sie den Beginn und das Ende von Anweisungen im Quellcode signalisieren.
  - Vor jedem Zugriff auf eine Klassen- oder Instanzvariable oder ein Feldelement V wird bei lesendem Zugriff die Anweisung `Load(V)` und bei schreibendem Zugriff `Store(V)` hinzugefügt. Die Anweisung `Load(V)` signalisiert, dass alle sich in Ausführung befindenden Anweisungen auf V lesend zugreifen, `Store(V)` signalisiert einen schreibenden Zugriff. V ist als Argument so

anzugeben, dass die zugrundeliegende Speicherstelle eindeutig zu identifizieren ist. Die Anweisungen `Load()` und `Store()` werden als Zugriffsanweisungen bezeichnet, weil sie den Zugriff auf eine Speicherstelle signalisieren.

- **Protokollierung:** Bei der Protokollierung wird die instrumentierte Anwendung mit Eingabedaten für die sequentielle Anwendung ausgeführt. Dabei wird auf die Signalisierungsanweisungen reagiert und ein dynamischer Aufruf- und Abhängigkeitsgraph aufgebaut.

  Die Reaktionen auf die Anweisungen werden in diesem Abschnitt nur kurz angerissen. Eine genauere Beschreibung ist Kapitel 5.3 zu entnehmen. Die Zusammenhänge werden Anhand eines Beispiels in Kapitel 5.4 klarer.

  - Auf das Signal `Enter(A)` wird eine neuer Laufzeitrepräsentant L_A für die Anweisung A erzeugt und auf dem sogenannten Aufrufstapel abgelegt. Befindet sich bereits ein Laufzeitrepräsentant L_B einer anderen Anweisung B auf dem Stapel, wird L_B als Vorgänger von L_A markiert.
  - Auf das Signal `Leave(A)` werden alle Laufzeitrepräsentanten bis zum nächsten Laufzeitrepräsentanten L_A des Knoten A und anschließend auch L_A vom Aufrufstapel entfernt.
  - Auf das Signal `EnterIteration(S)` werden alle Laufzeitrepräsentanten bis zum nächsten Laufzeitrepräsentanten L_S der Schleife S vom Aufrufstapel entfernt. Daraufhin wird ein neuer Laufzeitrepräsentant L_I auf den Stapel gelegt, der eine Iteration I der Schleife S repräsentiert.
  - Auf das Signal `Load(V)` wird entsprechend des Basisalgorithmus aus Abbildung 14 reagiert: es wird überprüft, ob LW(V) leer ist. Falls nicht, wird für den obersten Laufzeitrepräsentant L_A des Aktivitätsstapels eine RAW-Datenabhängigkeit von LW(V) ausgegeben. Zudem wird die Menge LR(V) um L_A ergänzt.
  - Auf das Signal `Store(V)` wird ebenfalls entsprechend des Basisalgorithmus aus Abbildung 14 reagiert: es wird überprüft, ob LR(V) leer ist. Falls nicht, wird für den obersten Laufzeitrepräsentant L_A für jeden Laufzeitrepräsentant aus LR(V) eine WAR-Datenabhängigkeit ausgegeben. LR(V) wird daraufhin geleert. Zudem wird überprüft, ob LW(V) leer ist. Falls nicht, wird für L_A eine WAW-Datenabhängigkeit von LW(V) ausgegeben. Zudem wird LW(V) = L_A gesetzt.

- **Nachverarbeitung:** Bei der Nachverarbeitung werden Abhängigkeiten des dynamischen Aufruf- und Abhängigkeitsgraphen zurück auf den Syntaxbaum projiziert. Erst durch diesen Schritt entstehen die Abhängigkeitsgraphen, die vom Architekturerkenner benötigt werden. Für eine RAW-/WAR-/WAW-Abhängigkeit des Laufzeitrepräsentanten L_B vom Laufzeitrepräsentanten L_A ergibt sich die resultierende Abhängigkeit im Syntaxbaum wie folgt.

  Wie bei der Protokollierung sind Details zur Nachverarbeitung sind in den Kapiteln 5.3 und 5.4 zu finden.

- Finde den ersten gemeinsamen Vorgänger L_V von L_A und L_B. Seien $P_1 = (L_A, L_{A1}, \ldots, L_{AN}, L_V)$ und $P_2 = (L_B, L_{B1}, \ldots, L_{BM}, L_V)$ die Pfade der Laufzeitrepräsentanten, die vom Abstieg von L_A und L_B nach L_V besucht wurden.
- Dann ist der Syntaxknoten B_M RAW-/WAR-/WAW-abhängig vom Syntaxknoten A_N falls V keine Schleife ist.
- Falls V eine Schleife ist, sind A_N und B_M Laufzeitrepräsentanten für eine Iteration der Schleife V. Damit ist B_{M-1} iterationsübergreifend RAW-/WAR-/WAW-abhängig vom Syntaxknoten A_{N-1}.

4.7.5 Alltagsnahes Beispiel

In diesem Abschnitt wird ein alltagsnahes Beispiel vorgestellt, das für die folgenden Kapitel (in 4.7.6, 4.8.1, 4.8.2, 4.9.1 und 4.9.2) zur Demonstration von Teilen des Konzepts herangezogen wird. Es ersetzt das motivierende Beispiel aus Kapitel 4.2, weil durch die Struktur des alltagsnahen Beispiels komplexere Sachverhalte veranschaulicht werden können. Es kann beispielsweise gezeigt werden, wieso für die Erzeuger-Stufe eines Fließbands die Platzhalter Prolog und Epilog definiert wurden und wie diese belegt werden.

Wir betrachten folgenden vereinfachten Vorgang zur Zubereitung eines Tellers Spaghetti Bolognese:

1. Portion Hackfleisch aus dem Kühlschrank holen
2. Falls das Hackfleisch Pferdefleisch enthalten sollte: Abbruch
3. Zwiebeln schneiden
4. Hackfleisch und Zwiebeln anbraten
5. Tomatenmark dazugeben
6. Wasser erhitzen
7. Nudeln kochen

Um auch die Parallelisierung von Schleifen durch das vorgestellte Beispiel abdecken zu können, wird das Beispiel erweitert: Stellen wir uns vor, dass ein Restaurant ständig neue Bestellungen für einen Teller Spaghetti Bolognese bekommt. Dann wird der obige Vorgang mehrmals wiederholt. Um die Qualität hoch zu halten, werden die Schritte Zwiebeln schneiden und Anbraten in Abhängigkeit der Qualität des letzten Tellers angepasst: waren zu viele Zwiebeln auf dem letzten Teller, werden diesmal weniger Zwiebeln geschnitten. War das Hackfleisch durch zu langes Anbraten nichtmehr saftig genug, wird diesmal kürzer angebraten.

Eine mögliche Modellierung dieses Vorgangs in Form von Quellcode ist in Abbildung 20 dargestellt.

```
 1 while(i < BestellteTeller) {
 2  Hackfleisch = AusKühlschrankHolen();
 3  if(EnthältPferd(Hackfleisch)) { break; }
 4  Zwiebeln = ZwiebelnSchneiden();
 5  Gebratenes = Braten(Zwiebeln, Hackfleisch);
 6  Soße = Übergießen(Gebratenes, Tomatenmark);
 7  HeißesWasser = WasserErhitzen();
 8  Nudeln = Kochen(HeißesWasser);
 9  i++;
10 }
```

Abbildung 20: Eine mögliche Modellierung des alltagsnahen Beispiels.

4.7.6 Anwendung auf das alltagsnahe Beispiel

Die vorgestellte Abhängigkeitsanalyse wird im Folgenden auf das alltagsnahe Beispiel (Kapitel 4.7.5) angewandt. Zunächst werden zwei Teilbäume des Syntaxbaums erkannt, für die ein Parallelisierungsansatz verfolgt wird: eine Sequenz und eine Schleife. Es ist dabei zunächst irrelevant, dass die Sequenz ein Kind der Schleife ist. Das Ergebnis der Abhängigkeitsanalyse ist in Abbildung 21 dargestellt.

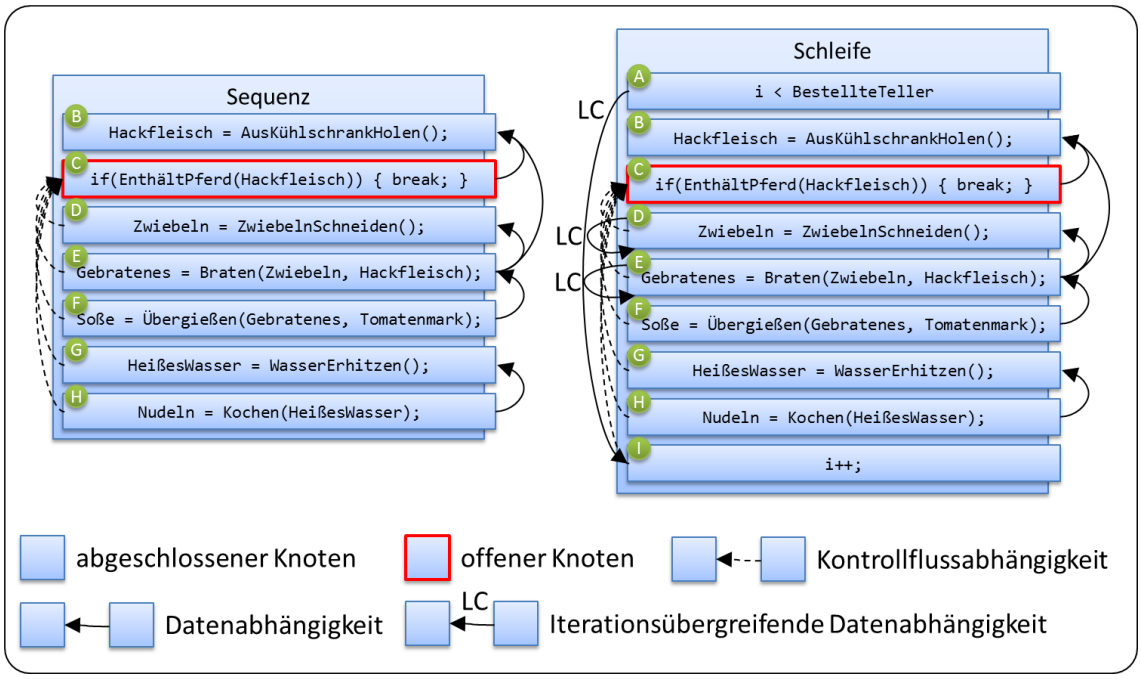

Abbildung 21: Ergebnis der Abhängigkeitsanalyse, bei der Anwendung auf das alltagsnahe Beispiel. Links: Abhängigkeitsgraph für die Sequenz, rechts: Abhängigkeitsgraph für die Schleife.

Die Anweisung `C` wird als offener Syntaxknoten identifiziert, weil sie ein `break` aber nicht die zugehörige Schleife enthält. Alle Anweisungen, die auf die offene Anweisung folgen sind von ihr kontrollflussabhängig, was durch die gestrichelten Pfeile signalisiert wird.

Für die ersten drei Anweisungen der Sequenz (`B`, `C` und `D`) verläuft die Analyse der Datenabhängigkeiten über Zugriffe auf lokale Variablen wie folgt:

1. `B` verübt einen Schreibzugriff auf die lokale Variable `Hackfleisch`. Weil LR(`Hackfleisch`) und LW(`Hackfleisch`) beide leer sind wird keine Abhängigkeit erzeugt. Es wird LW(`Hackfleisch`) = `B` gesetzt.

2. `C` verübt einen Lesezugriff auf `Hackfleisch`. Weil LW(`Hackfleisch`) = `B` nicht leer ist, wird eine RAW-abhängigkeit von `C` zu `B` ausgegeben. Es wird LR(`Hackfleisch`) = `C` gesetzt.
3. `D` verübt einen Schreibzugriff auf die lokale Variable `Zwiebeln`. Weil LR(`Zwiebeln`) und LW(`Zwiebeln`) beide leer sind, wird keine Abhängigkeit ausgegeben. Es wird LW(`Zwiebeln`) = `D` gesetzt.
4. Im weiteren Verlauf werden die übrigen Abhängigkeiten ausgegeben.

Für die Schleife werden mit einer Ausnahme die gleichen Datenabhängigkeiten über Zugriffe auf lokale Variablen ermittelt: Nach der Simulation der ersten Iteration ist LW(`i`) = `I`, weil `I` (in der ersten Iteration) den letzten Schreibzugriff auf `i` durchführt. In der zweiten Iteration verübt `A` einen Lesezugriff auf `i`. Weil LW(`i`) = `I` und der Schreibzugriff in einer anderen Iteration stattfand als der Lesezugriff, wird eine iterationsübergreifende Abhängigkeit von `A` zu `I` ausgegeben.

Bei der dynamischen Abhängigkeitsanalyse werden die zwei verbleibenden Abhängigkeiten (die iterationsübergreifenden Abhängigkeiten von `D` und `E` zu sich selbst) der Schleife ermittelt. Diese iterationsübergreifenden Datenabhängigkeiten lassen sich nicht durch den Quellcodeausschnitt des Beispiels erklären, weil er dazu nicht vollständig genug ist. Die Datenabhängigkeiten können aber anhand der Beschreibung des Vorgangs erklärt werden: `ZwiebelnSchneiden()` und `Braten()` werden in jeder Iteration in Abhängigkeit der letzten Iteration angepasst um die Qualität des Essens hoch zu halten. Im Quellcode passiert in der Methode `ZwiebelnSchneiden()` also mehr, als man dem Quellcodeausschnitt ansieht. In einer globalen Variablen wird gespeichert, wie viele Zwiebeln dem Essen hinzugefügt wurden. In der nächsten Iteration wird genau diese Variable wieder ausgelesen, um die Anzahl der Zwiebeln korrigieren zu können.

4.8 Architekturerkenner

Die grundlegende Idee des Parallelisierungsansatzes wurde bereits in Kapitel 4.4.1 erläutert: Sequenzen sollen auf Taskgraphen und Schleifen auf Fließbänder abgebildet werden, um die willkürlich festgelegte Reihenfolge kausal unabhängiger Anweisungen aufzuheben. Gleichzeitig muss Reihenfolge kausal abhängiger Anweisungen beibehalten werden, um kein Fehlverhalten der parallelen Anwendung zu provozieren. Diese Abhängigkeiten werden durch die Abhängigkeitsanalyse aus Kapitel 4.7 ermittelt und in Abhängigkeitsgraphen abgelegt. In diesem Abschnitt wird nun geklärt, wie Abhängigkeiten bei der Abbildung von Sequenzen und Schleifen auf Taskgraphen und Fließbänder berücksichtigt werden und wieso eine Nichtberücksichtigung zum Fehlverhalten führen würde. Aus den Algorithmen folgt zudem die konkrete Konfiguration der Taskgraphen und Fließbänder, die als gleichwertiger Ersatz für Sequenzen und Schleifen vorgeschlagen werden.

4.8.1 Abbildungsalgorithmus von Sequenz auf Taskgraph

Für die Abbildung von einer Sequenz auf einen Taskgraph wird folgendermaßen vorgegangen:

1. Eine Sequenz, die offene Syntaxknoten enthält wird zunächst in mehrere Teilsequenzen ohne offene Syntaxknoten aufgeteilt. Teilsequenzen, die nur eine Anweisung enthalten, werden nicht weiter betrachtet.

Erläuterung: Anweisungen, die vor einem offenen Knoten stehen, dürfen nicht hinter den offenen Knoten verschoben werden, weil ihre Ausführung sonst ungewollt durch den offenen Knoten bedingt wird. Knoten, die auf einen offenen Knoten folgen, dürfen nicht vor den offenen Knoten verschoben werden, da ihre Ausführung sonst nicht mehr durch den offenen Knoten bedingt wird. Deshalb können Knoten vor offenen Knoten nicht nebenläufig zu Knoten nach offenen Knoten ausgeführt werden. Zudem dürfen offene Syntaxknoten nicht auf Tasks abgebildet werden, weil sonst der Kontrollfluss verändert wird (siehe dazu Kapitel 4.7.2). Aus dieser Überlegung folgt die Zerlegung in Teilsequenzen, die einzeln auf Taskgraphen abgebildet werden. Für Teilsequenzen mit nur einer Anweisung kann ein Taskgraph nicht zu Parallelität führen: diese Teilsequenzen werden deshalb nicht parallelisiert.

2. Für jede Sequenz ohne offene Syntaxknoten wird ein Taskgraph erzeugt. Es werden so viele Tasks erzeugt, wie es Anweisungen gibt. Als Aktivität wird jedem Task eine Anweisung zugeordnet. Einem Task T_A mit der Aktivität A wird jeder Task V_B mit der Aktivität B als Vorgänger hinzugefügt, wenn gilt: A ist datenabhängig von B. Auf die Angabe redundanter Vorgänger wird verzichtet.

   Erläuterung: Für eine gute Skalierbarkeit werden möglichst viele Tasks angestrebt, da jeder Task von einem Ausführungsfaden ausgeführt werden kann. Für eine hohe Nebenläufigkeit werden einem Task nur diejenigen Vorgänger zugeordnet, die nötig sind: die Reihenfolge unabhängiger Anweisungen soll nicht willkürlich festgelegt werden. Zur Wahrung der Korrektheit werden einem Task diejenigen Tasks als Vorgänger zugeordnet, deren Reihenfolge aufgrund von Datenabhängigkeiten eingehalten werden muss. Die Angabe eines Vorgängers ist redundant, wenn der Task aufgrund der bereits ausgewiesenen Vorgängerbeziehungen schon auf den Vorgänger wartet.

Anwendung auf das alltagsnahe Beispiel

Das Ergebnis der Anwendung des beschriebenen Algorithmus auf die Sequenz des alltagsnahen Beispiels ist in Abbildung 22 dargestellt. Auf das Zustandekommen des zugrundeliegenden Abhängigkeitsgraphen wurde bereits in (Kapitel 4.7.6) eingegangen. Dort ist auch die Legende des Abhängigkeitsgraphen zu finden.

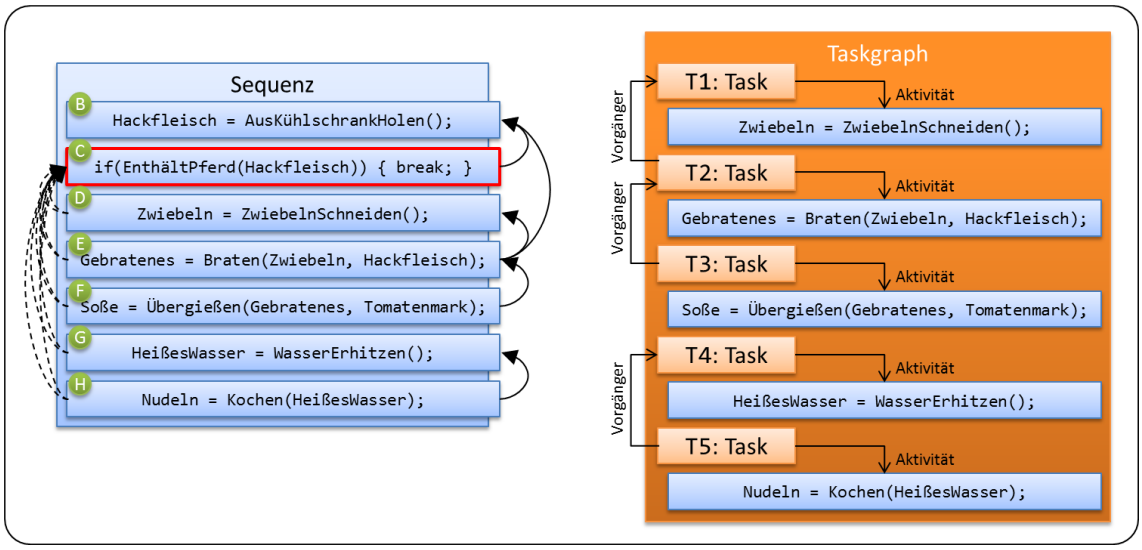

Abbildung 22: Ergebnis der Architekturerkennung bei der Anwendung auf die Sequenz des alltagsnahen Beispiels. Links: zuvor generierter Abhängigkeitsgraph, rechts: resultierender Taskgraph.

Zunächst wird die Sequenz in zwei Teilsequenzen aufgeteilt, da `C` ein offener Syntaxknoten ist. Die erste Teilsequenz enthält nur eine Anweisung und wird nicht weiter betrachtet. Die zweite Teilsequenz enthält mehrere Anweisungen und kann auf einen Taskgraphen abgebildet werden. Für jede Anweisung der zweiten Teilsequenz wird ein Task angenommen. Die Vorgänger der Tasks werden entsprechend den Abhängigkeiten gewählt.

Zurück auf die Realität übertragen bedeutet dies, dass insgesamt fünf Köche in den Vorgang einbezogen werden können, die jeweils eine Aufgabe (einen Task) übernehmen. Sobald sichergestellt ist, dass das Hackfleisch kein Pferdefleisch enthält, können die Köche mit ihren Aufgaben beginnen. Der Koch mit der Aufgabe `T1` muss auf keinen anderen Koch warten und kann sofort beginnen, Zwiebeln zu schneiden. Der Koch mit der Aufgabe `T2` muss zunächst in Wartestellung bleiben und kann erst mit dem Anbraten beginnen, wenn sein Vorgänger (der Koch mit der Aufgabe `T1`) mit dem Schneiden der Zwiebeln fertig ist. Auch die Köche mit den Aufgaben `T3` und `T5` müssen zunächst warten. Der Koch `T4` kann allerdings ebenfalls sofort loslegen. Es fällt auf, dass Köche, die sowieso nie gleichzeitig arbeiten können (beispielsweise `T1`, `T2` und `T3`), durch das Zusammenfassen von Aufgaben eingespart werden können. Auf diese Optimierung wird in Kapitel 4.9.1 eingegangen.

4.8.2 Abbildungsalgorithmus von Schleife auf Fließband

Die Abbildung von einer Schleife auf ein Fließband teilt sich in zwei Abschnitte: die Abbildung auf eine implizite Erzeuger-Stufe und die Abbildung verbleibender Anweisungen auf weitere Stufen. Der Grundgedanke ist folgender: die Erzeuger-Stufe soll einen Strom von Elementen erzeugen und dabei die gleiche Anzahl an Iterationen durchlaufen, wie die ursprüngliche Schleife. Alle Anweisungen der ursprünglichen Schleife, die nicht der Erzeuger-Stufe zugeordnet werden, sollen in nachgeschaltete Fließbandstufen ausgelagert werden.

Für die Abbildung einer Schleife auf ein Fließband wird folgendermaßen vorgegangen:

1. Enthält der Schleifenrumpf offene Syntaxknoten durch `return`, kann die Schleife nicht auf ein Fließband abgebildet werden.

Erläuterung: Offene Syntaxknoten durch `return` sind weder in der Erzeuger-Stufe noch in den restlichen Fließbandstufen erlaubt, weil sonst der Kontrollfluss verändert wird (siehe dazu Kapitel 4.7.2).

2. Als Laufbedingung der Erzeuger-Stufe wird die Laufbedingung der zugrundeliegenden Schleife gewählt. Den Platzhaltern Prolog und Epilog der Erzeuger-Stufe sind alle Anweisungen des Schleifenrumpfs zuzuordnen, die Einfluss auf die Anzahl der Iterationen haben. Diese Anweisungen werden mit PE (Prolog/Epilog) bezeichnet. Initial wird PE mit allen Anweisungen belegt, von denen die Laufbedingung (iterationsübergreifend) datenabhängig ist. Außerdem werden alle offenen Syntaxknoten des Schleifenrumpfs zu PE hinzugefügt. Die Menge PE wird nun sukzessive erweitert: Ist eine Anweisung aus PE kontrollfluss- oder datenabhängig von einer weiteren Anweisung A, die nicht in PE enthalten ist, wird A der Menge PE hinzugefügt. Dieser Vorgang wird solange wiederholt, bis sich keine Änderungen mehr ergeben. Umfasst PE alle Anweisungen des Schleifenrumpfs der zugrundeliegenden Schleife, wird die Abbildung auf ein Fließband abgebrochen, ansonsten werden die restlichen Anweisungen mit R bezeichnet.

   Erläuterung: Enthält der Schleifenrumpf keine offenen Anweisungen, wird die Schleife durch eine zu falsch ausgewertete Laufbedingung beendet (wie typischerweise bei `for`- und `foreach`-Schleifen). Während einer Iteration wird die Laufbedingung von mindestens einer Anweisung A aktualisiert (außer im irrelevanten Fall einer Endlosschleife). Die Laufbedingung ist dann iterationsübergreifend datenabhängig von A. Die Reihenfolge von A und der Laufbedingung darf nicht vertauscht werden: A kann nicht in nachgeschaltete Fließbandstufen ausgelagert werden und wird deshalb in die Erzeuger-Stufe übernommen. Enthält der Schleifenrumpf offene Anweisungen durch `break` oder `continue`, wird die Anzahl der Iterationen nicht nur durch die Laufbedingung, sondern auch durch die offenen Anweisungen bestimmt. Auch diese Anweisungen müssen daher in die Erzeuger-Stufe übernommen werden. Gibt es Anweisungen im Schleifenrumpf, von denen Anweisungen der Erzeuger-Stufe abhängig sind, muss denen Reihenfolge ebenfalls eingehalten werden. Daher rührt die sukzessive Erweiterung der Menge PE.

3. Die Anweisungen aus PE müssen auf Prolog und Epilog der Erzeuger-Stufe verteilt werden. Die ursprüngliche Reihenfolge der Anweisungen aus PE wird dabei beibehalten. Die Anweisungen A_{PE} aus PE werden nun nacheinander zugeordnet: A_{PE} wird dem Prolog zugeordnet, falls es eine Anweisung A_R aus R gibt, die iterationsintern kontrollfluss- oder datenabhängig von A_{PE} ist. Sonst werden A_{PE} und alle darauffolgenden Anweisungen aus PE dem Epilog zugeordnet. Gibt es Anweisungen A_R aus R, die iterationsintern abhängig von einer Anweisung A_E im Epilog sind, müssen diese Anweisungen ebenfalls in den Epilog übernommen werden.

   Erläuterung: Durch die Zuordnung von Prolog und Epilog wird nur entschieden, nach welcher Anweisung der Zustand der Schleifeniteration als Element verpackt und über die Ausgabeschnittstelle zur ersten nachgeschalteten Fließbandstufe versendet wird. Die Reihenfolge der Anweisungen aus PE wird beibehalten, weil diese Anweisungen voneinander abhängig sind (siehe Schritt 2) und deren Reihenfolge deshalb nicht vertauscht werden darf. Eine Anweisung A_R aus R, die iterationsintern kontrollfluss- oder datenabhängig von einer Anweisung A_{PE} aus PE ist, wobei A_{PE} dem Epilog zugeordnet wurde, müsste auch in den Epilog verschoben werden. Diese Anweisung

könnte daher nichtmehr in eine andere Fließbandstufe ausgelagert werden. Wenn die erste Anweisung A_{PE} aus PE dem Epilog zugeordnet wird, müssen die restlichen Anweisungen aus PE auch in den Epilog um die Reihenfolge dieser Anweisungen einzuhalten.

4. Für jede verbleibende Anweisung A_R aus R wird eine Fließbandstufe mit der Aktivität A_R angenommen: bei N verbleibenden Anweisungen also N-1 Verbraucher-Erzeuger-Stufen und eine Verbraucher-Stufe, die gemäß Abbildung 12 stets die letzte Stufe eines Fließbandes darstellt. Die Stufen werden der Reihenfolge der Anweisungen nach verknüpft. Solange es Stufen gibt, die iterationsübergreifend abhängig sind, werden sie zu einer Stufe zusammengefasst, indem deren Aktivitäten als Sequenz vereint werden. Ist eine (zusammengefasste) Stufe zu sich selbst iterationsübergreifend abhängig, wird sie als nicht replizierbar markiert, ansonsten als replizierbar.

   Erläuterung: Für eine gute Skalierbarkeit werden möglichst viele Stufen angestrebt, da jede Stufe von einem Ausführungsfaden ausgeführt werden kann. Für eine hohe Nebenläufigkeit wird jede Stufe zunächst als replizierbar angenommen. Die Verknüpfung der Stufen erfolgt in der Reihenfolge der Anweisungen im Schleifenrumpf, weil die iterationsinternen Abhängigkeiten dadurch automatisch erhalten werden: iterationsinterne Abhängigkeiten verlaufen immer von einer Anweisung zu einer anderen Anweisung im Schleifenrumpf, die über ihr steht. Bei iterationsübergreifenden Abhängigkeiten zwischen Anweisungen verläuft die Abhängigkeit immer in die andere Richtung: von einer Anweisung zu einer Anweisung die unter ihr steht. Eine iterationsübergreifende Abhängigkeit geht damit immer von einer Stufe S_i zu einer nachfolgenden Stufe S_j mit $i < j$. Für zwei aufeinanderfolgende Elemente e_1 und e_2 ist damit $S_i(e_2)$ abhängig von $S_j(e_1)$. Ein Fließband bietet aber keine Möglichkeit um sicherzustellen, dass $S_i(e_2)$ erst ausgeführt wird, wenn $S_j(e_1)$ abgeschlossen ist. Die Reihenfolge dieser Bearbeitungsschritte wird sichergestellt, indem die Stufen zusammengefasst werden. Für eine Stufe S, die zu sich selbst iterationsübergreifend abhängig ist gilt die gleiche Überlegung: hier ist $S(e_2)$ abhängig von $S(e_1)$. Die Reihenfolge dieser Bearbeitungsschritte wird sichergestellt, indem die Stufe als nicht replizierbar markiert wird.

Anwendung auf das alltagsnahe Beispiel

Das Ergebnis der Anwendung des beschriebenen Algorithmus auf die Schleife des alltagsnahen Beispiels ist in Abbildung 23 dargestellt. Auf das Zustandekommen des zugrundeliegenden Abhängigkeitsgraphen wurde bereits in (Kapitel 4.7.6) eingegangen. Dort ist auch die Legende des Abhängigkeitsgraphen zu finden.

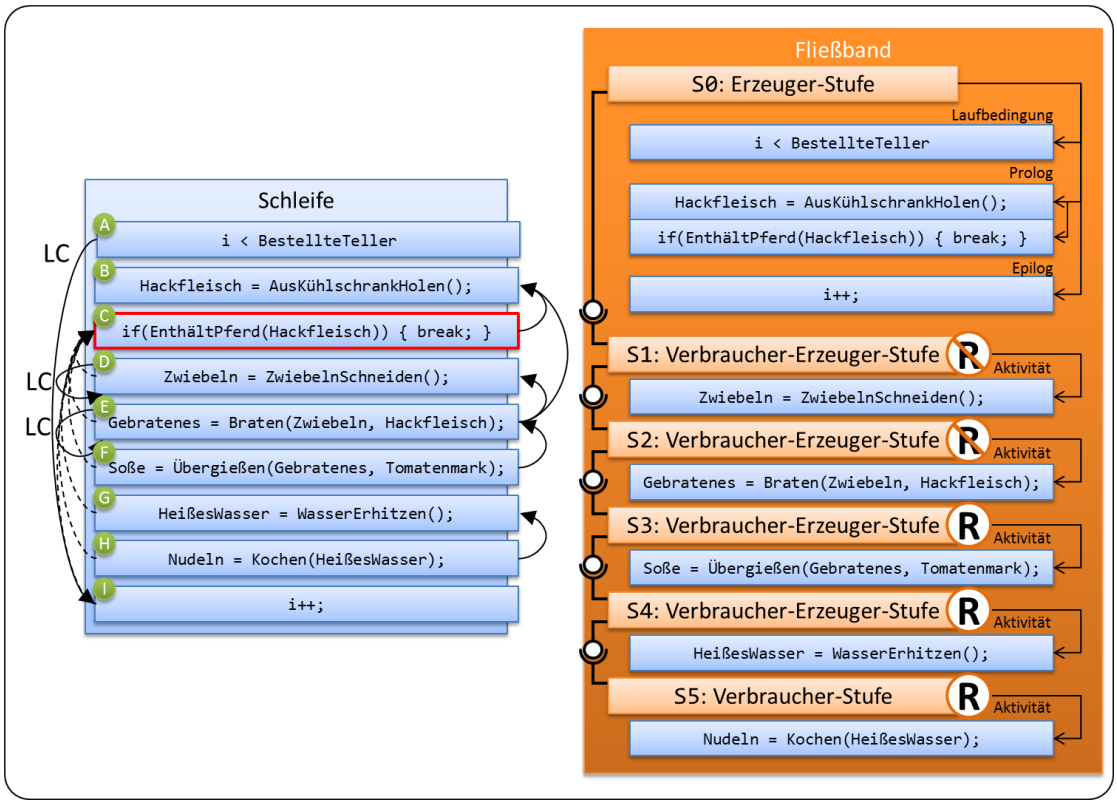

Abbildung 23: Ergebnis der Architekturerkennung bei der Anwendung auf die Schleife des alltagsnahen Beispiels. Links: zuvor generierter Abhängigkeitsgraph, rechts: resultierendes Fließband.

Eine Abbildung auf ein Fließband ist nach dem ersten Schritt prinzipiell möglich, weil der Schleifenrumpf keine offene Anweisung durch `return` enthält. Im zweiten Schritt besteht die Menge PE zunächst aus der Anweisung `I` weil die Laufbedingung `A` iterationsübergreifend von ihr abhängig ist. Die offene Anweisung `C` wird PE ebenfalls hinzugefügt. PE wird nun sukzessive erweitert: `B` wird PE hinzugefügt, weil die offene Anweisung `C` (die bereits zu PE gehört) abhängig von ihr ist. Danach ergeben sich für PE keine weiteren Änderungen mehr. Der Reihenfolge im Schleifenrumpf folgend wird nun zunächst `B` und anschließend `C` dem Prolog der Erzeuger-Stufe hinzugefügt, weil es Anweisung aus R gibt, die iterationsintern von ihnen abhängig sind. Die Anweisung `I` wird dem Epilog zugeordnet. Für die verbleibenden Anweisungen `D`, `E`, `F`, `G` und `H` wird nun je eine Stufe angenommen. Da es keine iterationsübergreifenden Abhängigkeiten zwischen Stufen gibt, müssen keine Stufen zusammengefasst werden. Aufgrund der iterationsübergreifenden Abhängigkeiten zu sich selbst, `F`, `G` und `H` im Gegensatz zu `D` und `E` als replizierbar markiert.

Zurück auf die Realität übertragen bedeutet dies, dass mehrere Teller leckerer Spaghetti Bolognese durch eine fließbandartige Verarbeitung hergestellt werden können. Jede Aufgabe (Stufe) kann von einem Koch besetzt werden. Replizierbare Stufen sogar von mehreren Köchen, die sich die Aufgabe teilen und somit die Bearbeitungszeit der Stufe verkürzen. Ein „Überprüfer“-Koch (`S0`) ist dafür verantwortlich das Hackfleisch auf Pferdefleisch zu untersuchen und den Gesamtvorgang zur Not abzubrechen. Ein „Zwiebelschneider“-Koch (`S1`) wartet auf das OK des „Überprüfer“-Kochs und beginnt dann damit, Zwiebeln zu schneiden und deren Menge entsprechend der zuvor gefertigten Tellers anzupassen. Ein weiterer „Zwiebelschneider“-Koch ist nicht sinnvoll, weil es sonst möglich wäre, dass zwei Teller mit der nicht

angepassten Zwiebelmenge gefertigt werden. Die Zwiebeln werden dem „Anbrat“-Koch weitergegeben. Sollte der „Überprüfer“-Koch mittlerweile schon das OK für eine weitere Portion Hackfleisch gegeben haben, kann der „Zwiebelschneider“-Koch schon jetzt mit einer weiteren Portion Zwiebeln beginnen.

4.9 Architekturbeschreibungen und Codepartitionierungen

Ein von dieser Arbeit erzeugter Parallelisierungsvorschlag besteht aus annotiertem sequentiellem Quellcode, der alle Informationen zur automatischen Transformation in parallelen Quellcode enthält. Auf die automatische Transformation selbst wird in dieser Arbeit nicht eingegangen, weil sie von einer weiteren Arbeit umgesetzt wird [W13]. Dieser Arbeit ist auch die formale Definition der verwendeten Architekturbeschreibungssprache TADL zu entnehmen.

Ein Parallelisierungsvorschlag erstreckt sich über umschließende Marker (`#region` und `#endregion`) auf einen sequentiellen Codeabschnitt und besteht aus einer Architekturbeschreibung und einer zugehörigen Codepartitionierung. Er bildet damit eine Abbildungsvorschrift von einem sequentiellen auf einen parallelen Codeabschnitt. Eine Architekturbeschreibung (TADL-Ausdruck) folgt direkt auf den `#region`-Marker und gibt eine Konfiguration eines parallelen Architekturmusters an, der eindeutige Bezeichner für die Platzhalter der ausführbaren Komponenten vergibt. Durch eine Architekturbeschreibung wird also angegeben, wie verschiedene Komponenten miteinander interagieren, ohne jedoch auf deren Funktionalität einzugehen. Dies geschieht erst durch die zugehörige Codepartitionierung: Sie zerlegt einen sequentiellen Codeabschnitt in Bestandteile und ordnet diesen Bestandteilen ebenfalls eindeutige, zur Architekturbeschreibung passende, Bezeichner zu. Bezeichner einzeiliger Anweisungen werden der Zeile vorangestellt und durch einen Doppelpunkt von der Anweisung getrennt. Mehrzeilige Anweisungen werden über umschließende Marker (`#region` und `#endregion`) abgegrenzt, wobei der Bezeichner direkt auf den `#region`-Marker folgt.

4.9.1 Notation für Taskgraphen

Die Architekturbeschreibung für einen Taskgraphen gibt an, welche Tasks es gibt und welche Vorgänger ein Task besitzt: ein Task wird durch einen eindeutigen Bezeichner, dessen Vorgänger darauffolgend in Klammern angegeben. Die Klammern entfallen für Tasks, die keine Vorgänger besitzen. Tasks werden durch Kommata getrennt. Die Codepartitionierung gibt an, welchem Task welche Aktivität zugeordnet ist.

Zur Veranschaulichung der Notation wird der Taskgraph (Kapitel 4.7.6, Abbildung 22) des alltagsnahen Beispiels herangezogen und dessen Quellcode (Kapitel 4.7.5, Abbildung 20) annotiert dargestellt (Abbildung 24).

```
 1 while(i < BestellteTeller) {
 2  Hackfleisch = AusKühlschrankHolen();
 3  if(EnthältPferd(Hackfleisch)) { break; }
 4  #region T1, T2(T1), T3(T2), T4, T5(T4)
 5   T1: Zwiebeln = ZwiebelnSchneiden();
 6   T2: Gebratenes = Braten(Zwiebeln, Hackfleisch);
 7   T3: Soße = Übergießen(Gebratenes, Tomatenmark);
 8   T4: HeißesWasser = WasserErhitzen();
 9   T5: Nudeln = Kochen(HeißesWasser);
10  #endregion
11  i++;
12 }
```

Abbildung 24: Ergebnis der Annotation bei der Anwendung auf die Sequenz des alltagsnahen Beispiels. Schwarz: ursprünglicher Quellcode, blau: hinzugefügte Annotationen.

Die vorgestellte Notation zur Angabe von Taskgraphen ist geeignet um beliebige Abhängigkeitsbeziehungen zwischen Tasks auszudrücken. Es geht jedoch nicht intuitiv hervor, welche Tasks nacheinander und welche Tasks nebenläufig zueinander ausgeführt werden. Ineinander geschachtelte sequentielle und parallele Sektionen sind dazu besser geeignet:

- `T1;T2;T3` ist eine sequentielle Sektion aus den Tasks `T1`, `T2` und `T3`, führt diese zeitlich aufeinanderfolgend (`T1`, dann `T2`, dann `T3`) aus und ist abgeschlossen, wenn die letzte Anweisung abgeschlossen ist. Die sequentielle Sektion `T1;T2;T3` kann in einen dazu äquivalenten Taskgraphen überführt werden: `T1, T2(T1), T3(T1, T2)`.
- `T1||T2||T3` ist eine parallele Sektion aus den Tasks `T1`, `T2` und `T3`, führt diese nebenläufig aus und ist abgeschlossen, wenn die Ausführung aller Anweisungen abgeschlossen ist. Die parallele Sektion `T1||T2||T3` kann in einen dazu äquivalenten Taskgraphen überführt werden: `T1, T2, T3`.

Ein gieriger Algorithmus (engl. *greedy*) zur Ableitung eines Ausdrucks mit verschachtelten sequentiellen und parallelen Sektionen aus einem allgemeinen Abhängigkeitsgraph ist in Abbildung 25 gegeben.

```
 1 Sei (T1, …, TN) die ursprüngliche Reihenfolge der Tasks T1 bis TN.
 2 Sei eine Ebene ein Behälter mit einem Index, der mehrere Tasks aufnehmen kann.
 3
 4 Füge T1 der Ebene mit dem Index 0 hinzu.
 5 Für alle weiteren Tasks T aus (T2, …, TN) {
 6  Wenn T keinen anderen Task als Vorgänger besitzt {
 7   Füge T der Ebene mit dem Index 0 hinzu.
 8  } Ansonsten {
 9   Sei max das Maximum über alle Indizes der Ebenen, denen die Vorgänger von T
10   hinzugefügt wurden.
11   Füge T der Ebene mit dem Index max+1 hinzu.
12   Falls T von einem anderen Task aus einer übergeordneten Ebene unabhängig ist,
13   ist der erzeugte Ausdruck nicht optimal.
14  }
15 }
16
17 Wandle jede Ebene in eine parallele Sektion um. Vereine alle parallelen
18 Sektionen in einer sequentiellen Sektion in der Reihenfolge ihrer Indizes.
```

Abbildung 25: Greedy-Algorithmus zur Erzeugung eines Ausdrucks aus geschachtelten sequentiellen und parallelen Sektionen.

Der vorgestellte Greedy-Algorithmus erzeugt für den Taskgraph-TADL-Ausdruck des alltagsnahen Beispiels aus Abbildung 24 den in Abbildung 26 dargestellten Ausdruck. Hier wird auch ersichtlich, dass nur zwei der fünf Köche tatsächlich benötigt werden.

```
 1 while(i < BestellteTeller) {
 2  Hackfleisch = AusKühlschrankHolen();
 3  if(EnthältPferd(Hackfleisch)) { break; }
 4  #region (T1 ; T2 ; T3) || (T4 ; T5)
 5   T1: Zwiebeln = ZwiebelnSchneiden();
 6   T2: Gebratenes = Braten(Zwiebeln, Hackfleisch);
 7   T3: Soße = Übergießen(Gebratenes, Tomatenmark);
 8   T4: HeißesWasser = WasserErhitzen();
 9   T5: Nudeln = Kochen(HeißesWasser);
10  #endregion
11  i++;
12 }
```

Abbildung 26: Ergebnis des Greedy-Algorithmus bei der Anwendung auf den Taskgraph-Ausdruck des alltagsnahen Beispiels.

Der vorgestellte Greedy-Algorithmus erzeugt nicht notwendigerweise optimale Ausdrücke: mögliche Parallelität kann dabei verloren gehen. Aufgrund der besseren Lesbarkeit wird die Angabe verschachtelter Sektionen angestrebt, solange dadurch keine mögliche Parallelität verloren geht. Denn: ineinander verschachtelte Sektionen sind zwar besser lesbar als Taskgraphen, besitzen jedoch den entscheidenden Nachteil, dass ihre Ausdrucksmächtigkeit im Vergleich zu Taskgraphen geringer ist. Zur Verdeutlichung sei der Taskgraph `T1,T2(T1), T3(T1),T4(T2,T3),T5(T3),T6(T4,T5)` aus Abbildung 27 gegeben.

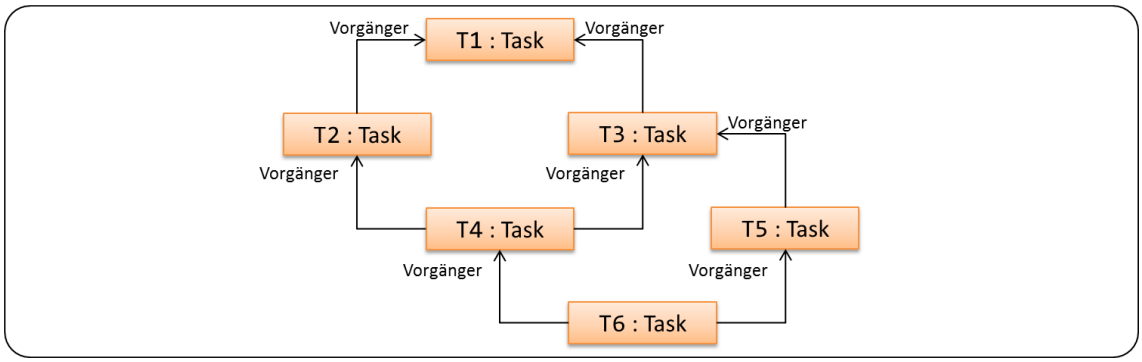

Abbildung 27: Ein Taskgraph, zu dem es keinen äquivalenten Ausdruck in Form verschachtelter Sektionen gibt.

`T2` und `T3`, `T4` und `T5` sowie `T2` und `T5` dürfen gleichzeitig ausgeführt werden. Es ist aber nicht möglich, die parallelen Sektionen `T2||T3`, `T4||T5` sowie `T2||T5` in einem Ausdruck aus verschachtelten Sektionen unterzubringen, ohne Abhängigkeiten zu verletzen. Durch die Architekturbeschreibungen `T1;(T2||T3);(T4||T5);T6` oder `T1;(T2||T5);T4;T6` kann nur ein Teil der möglichen Parallelität ausgedrückt werden.

Weil die Taskgraph-Notation mächtiger ist als die Notation durch parallele und sequentielle Sektionen, sollte die Architekturbeschreibungssprache TADL entsprechend erweitert werden.

4.9.2 Notation für Fließbänder

Die Erzeuger-Stufe eines Fließbands ist implizit und wird nicht explizit in der Architekturbeschreibung angegeben. Die Architekturbeschreibung für ein Fließband gibt an, welche Stufen es gibt, welche Stufen replizierbar sind und wie die Stufen miteinander verknüpft sind: eine Stufe wird durch einen eindeutigen Bezeichner angegeben, die Replizierbarkeit durch ein darauffolgendes Plus (`+`) und die Verknüpfung zweier Stufen durch einen gerichteten Pfeil (`=>`) von der erzeugenden zur verbrauchenden Stufe. Die Codepartitionierung gibt an, welcher Stufe

welche Aktivität zugeordnet ist. Die Platzhalter der Erzeuger-Stufe werden wie folgt angegeben: Die Laufbedingung der Erzeuger-Stufe stimmt aufgrund des Abbildungsalgorithmus (Kapitel 4.8.2) mit der Laufbedingung der Schleife überein und muss nicht explizit angegeben werden. Anweisungen des Schleifenrumpfs, die dem Prolog/Epilog zugeordnet sind, werden durch den Bezeichner Prolog/Epilog gekennzeichnet.

Zur Veranschaulichung der Notation wird das Fließband (Kapitel 4.8.2, Abbildung 23) des alltagsnahen Beispiels herangezogen und dessen Quellcode (Kapitel 4.7.5, Abbildung 20) annotiert dargestellt (Abbildung 28).

```
1  #region S1 => S2 => S3+ => S4+ => S5+
2   while(i < BestellteTeller) {
3    Prolog: Hackfleisch = AusKühlschrankHolen();
4    Prolog: if(EnthältPferd(Hackfleisch)) { break; }
5    S1: Zwiebeln = ZwiebelnSchneiden();
6    S2: Gebratenes = Braten(Zwiebeln, Hackfleisch);
7    S3: Soße = Übergießen(Gebratenes, Tomatenmark);
8    S4: HeißesWasser = WasserErhitzen();
9    S5: Nudeln = Kochen(HeißesWasser);
10   Epilog: i++;
11  }
12 #endregion
```

Abbildung 28: Ergebnis der Annotation bei der Anwendung auf die Schleife des alltagsnahen Beispiels. Schwarz: ursprünglicher Quellcode, blau: hinzugefügte Annotationen.

4.10 Zusammenfassung

In diesem Kapitel wurde ein Konzept vorgestellt, welches dazu eingesetzt werden kann, automatisch Parallelisierungsvorschläge für sequentielle Anwendungen zu erzeugen. Die Vorschläge werden in Form von Architekturbeschreibungen ausgegeben, welche zudem Tuning-Parameter definieren, deren Werte zur anschließenden Optimierung der parallelen Anwendung variiert werden können. Der Mehrwert dieses Verfahrens ist, dass sich aus sequentiellem Quellcode, der nicht für Mehrkernarchitekturen geschrieben wurde, vollautomatisch Parallelisierungskandidaten generieren lassen, die in Form konkreter Parallelisierungsvorschläge mit Tuning-Parametern, die zur Optimierung der Laufzeit genutzt werden können, ausgegeben werden.

In Kapitel 4.5 wurden zunächst die parallelen Architekturmuster Taskgraph und Fließband vorgestellt, nach deren Einsatzmöglichkeiten in der sequentiellen Anwendung gesucht wird. Ein Taskgraph kann eingesetzt werden um eine Sequenz zu parallelisieren, ein Fließband für eine Schleife. Ein Parallelisierungsvorschlag, beziehungsweise dessen Architekturbeschreibung, gibt konkrete Konfigurationen eines Taskgraphs oder Fließbands für einen bestimmten Codeabschnitt der sequentiellen Anwendung an. Dabei können bestimmte Parameter der Architekturmuster offen gelassen und als Tuning-Parameter veräußert werden. Dies wurde in Kapitel 4.6 untersucht. Im Anschluss daran wurde aufgezeigt, wie Abhängigkeiten zwischen Anweisungen der sequentiellen Anwendung ermittelt werden, die bei der Untersuchung auf Einsatzmöglichkeiten paralleler Architekturmuster berücksichtigt werden müssen. Weil sich manche Abhängigkeiten nur schwer oder überhaupt nicht statisch bestimmen lassen, wurde die Abhängigkeitsanalyse (Kapitel 4.7) in einen statischen und einen dynamischen Teil aufgeteilt. Wie Sequenzen und Schleifen unter Berücksichtigung dieser Abhängigkeiten auf semantisch äquivalente Konfigurationen von Taskgraphen und Fließbändern abgebildet werden können,

wurde in Kapitel 4.8 beschrieben. Zur Veräußerung der Parallelisierungsvorschläge wird die Architekturbeschreibungssprache TADL herangezogen, wie in Kapitel 4.9 aufgezeigt wurde.

Das vorgestellte Konzept lässt sich in eine Werkzeugkette eingliedern, welche die ausgegebenen Parallelisierungsvorschläge automatisch in parallelen Quellcode transformiert, der im Anschluss ebenfalls automatisch auf Korrektheit getestet werden kann. Durch die Tuning-Parameter ist schließlich ist eine Optimierung der parallelen Anwendung auf der Zielplattform möglich.

5 IMPLEMENTIERUNG

Im letzten Kapitel wurde ein Verfahren zur automatischen Erkennung von parallelen Architekturmustern vorgestellt, das in einer kombinierten statischen und dynamischen Analyse nach Mustern und Tuning-Parametern sucht und diese in Form von Architekturbeschreibungen veräußert. In diesem Kapitel wird aufgezeigt, wie das vorgestellte Verfahren zur automatischen Generierung von Parallelisierungsvorschlägen für sequentielle Anwendungen umgesetzt wurde.

Zunächst werden die drei in Abbildung 29 dargestellten Pakete in den Kapiteln 5.1 bis 5.3 dokumentiert. Im Kapitel 5.4 wird das Zusammenspiel von Instrumentierung, Protokollierung und Nachverarbeitung eingegangen, um die dynamische Abhängigkeitsanalyse (aus Kapitel 4.7.4) an einem Beispiel zu veranschaulichen.

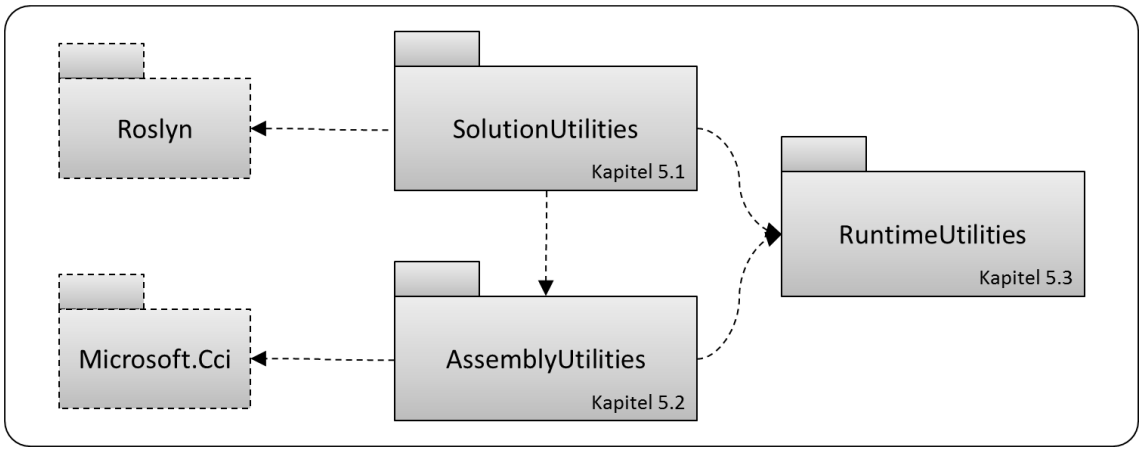

Abbildung 29: Paketdiagramm der Implementierung. Die eigenen Pakete `SolutionUtilities`, `AssemblyUtilities` und `RuntimeUtilities` und die externen Komponenten (gestrichelt) `Roslyn` und `Microsoft.Cci`.

Aufgrund der Abhängigkeiten zu den externen Komponenten `Roslyn` und `Microsoft.Cci`, die in Kapitel 2.3 vorgestellt wurden, zerfällt die Implementierung auf Paketebene in drei eigene Komponenten (Abbildung 29):

- **`SolutionUtilities`:** Das Paket `SolutionUtilies` steht im Zentrum der Implementierung. Den Einstiegspunkt zur Durchführung des implementierten Verfahrens bildet die Klasse `SolutionProcessor` mit den Phasen I, II und III, auf die im Kapitel 5.1.1 im Detail eingegangen wird. Das Paket enthält sämtliche Klassen zur Erledigung von Aufgaben, die sich auf Quellcode oder Syntaxbäume beziehen: Laden und Speichern von Softwareprojekten in Quellcodeform (engl. *solution*), Parsen von Quellcode, Durchführen statischer Abhängigkeitsanalysen (Kapitel 4.7.2 und 4.7.3), Teilinstrumentieren mit Abgrenzungsanweisungen (`Enter()`, `Leave()`, `EnterIteration()`) für die dynamische Abhängigkeitsanalyse (Kapitel 4.7.4), Erkennen von Architekturen (Kapitel 4.8), sowie Annotieren (Kapitel 4.9) zur Ausgabe von Parallelisierungsvorschlägen. Zur Durchführung dieser Aufgaben wird auf die API von `Roslyn` zurückgegriffen.
- **`AssemblyUtilities`:** Das Paket `AssemblyUtilities` ist eine Bibliothek, auf die von der Klasse `SolutionProcessor` zur Instrumentierung von Softwareprojekten in übersetzter Ausführungsform (engl. *assemblies*) zurückgegriffen wird. Die Klassen aus `AssemblyUtilities` können dazu verwendet werden, *assemblies* mit den Zugriffs-

anweisungen (siehe Kapitel 4.7.4) `Load()` und `Store()` auszustatten. Dazu wird auf die API von `Microsoft.Cci` zurückgegriffen. Wieso die Instrumentierung auf zwei Ebenen (Quell- und Zwischencode) stattfindet, wurde in Kapitel 4.7.4 noch nicht motiviert und wird deshalb in Kapitel 5.2.1 erläutert.

- **`RuntimeUtilities`:** Das Paket `RuntimeUtilities` stellt Klassen bereit, auf die bei der dynamischen Abhängigkeitsanalyse während der Protokollierung (Kapitel 4.7.4, Abbildung 19) zugegriffen wird. Die Klasse `Logger` bietet Methoden an, um auf Signalisierungsanweisungen, die der Anwendung durch die Instrumentierung hinzugefügt wurden, entsprechend zu reagieren und dynamische Datenabhängigkeiten aufzuzeichnen.

5.1 Das Paket SolutionUtilities

Das Paket `SolutionUtilities` enthält die zentrale Klasse `SolutionProcessor`, die den Prozess zur automatischen Generierung von Parallelisierungsvorschlägen ermöglicht, der in Kapitel 4.4 in Abbildung 6 dargestellt ist. Neben dieser Klasse wird im Folgenden ein Einblick in weitere Klassen gegeben, die in erheblichem Maße zur Implementierung beitragen. Klassen und Methoden, die für das Grundverständnis der Implementierung weniger wichtig sind, werden nicht thematisiert. Sie können der Dokumentation auf der beigefügten CD im Anhang entnommen werden.

5.1.1 Der Gesamtprozess in drei Phasen

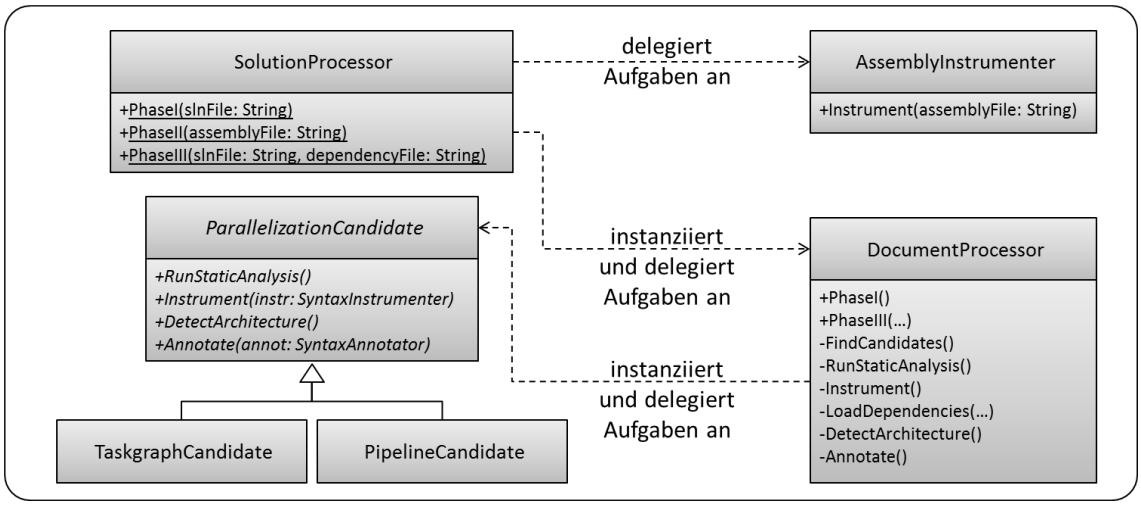

Abbildung 30: `SolutionProcessor` bildet den Einstiegspunkt um das implementierte Verfahren für eine sequentielle Anwendung durchzuführen. Aufgaben werden an `AssemblyInstrumenter` und `DocumentProcessor` delegiert. `DocumentProcessor` delegiert Aufgaben wiederum an die Parallelisierungskandidaten `TaskgraphCandidate` und `PipelineCandidate`.

`SolutionProcessor`: Der Gesamtprozess von einer sequentiellen Anwendung hin zu einer mit Architekturbeschreibungen annotierten Anwendung wird in drei aufeinanderfolgenden Phasen umgesetzt, die von der Klasse `SolutionProcessor` implementiert werden. Durch Phase I und II wird eine instrumentierte *assembly* erzeugt, die für die dynamische Abhängigkeitsanalyse (Kapitel 4.7.4) benötigt wird. Nach der Ausführung dieser *assembly* liegen die dynamisch ermittelten Datenabhängigkeiten in der Datei `DynamicDependencies.xml` vor. In Phase III werden die dynamischen Abhängigkeiten geladen, die statische Abhängigkeitsanalyse

(Kapitel 4.7.2 und 4.7.3) ausgeführt, die Architekturerkennung (Kapitel 4.8) angestoßen und schließlich die Annotierung mit Architekturbeschreibungen (Kapitel 4.9) vollzogen.

`SolutionProcessor.PhaseI():` Als Eingabe wird der Pfad zur *solution* (*.sln-Datei) der sequentiellen Anwendung erwartet. Der gesamte Inhalt des Ordners, der diese Datei enthält wird daraufhin in einen neuen Ordner (dessen Name die Zeichenkette _instr nachgestellt wird) kopiert. Anschließend wird die Kopie der *solution* geöffnet. Für jedes enthaltene Dokument (*.cs-Datei) wird ein `DocumentProcessor` erzeugt, an den die Durchführung der Phase für dieses Dokument delegiert wird. Nachdem die Phase I für alle Dokumente durchgeführt wurde, liegt die *solution* „teilinstrumentiert" vor. Die *solution* wird als „teilinstrumentiert" bezeichnet, weil nur die Abgrenzungsanweisungen `Enter()`, `EnterIteration()` und `Leave()` hinzugefügt werden, die zur Erfassung des Kontrollflusses dienen. Die verbleibenden Zugriffsanweisungen `Load()` und `Store()`, die zur Erfassung von Speicherzugriffen benötigt werden, kommen erst in der Phase II hinzu. An dieser Stelle sei noch einmal angemerkt, dass die Auftrennung der Instrumentierung in zwei Teile in Kapitel 5.2.1 erläutert wird. Jedem Dokument wird zudem eine `using`-Direktive für das Paket `RuntimeUtilities` vorangestellt, damit die Aufrufe zu den Instrumentierungsanweisungen der instrumentierten *assemblies* referenziert sind. Außerdem besitzt jedes Projekt der teilinstrumentierten *solution* einen neuen Verweis auf die *assembly* `RuntimeUtilities.dll`. Die teilinstrumentierte *solution* muss anschließend geöffnet und per Hand ergänzt werden: Es muss sichergestellt werden, dass die Anweisung `Logger.Save();` ausgeführt wird, bevor sich die Anwendung beendet. Anschließend ist die *solution* zu übersetzen.

`SolutionProcessor.PhaseII():` Als Eingabe wird der Pfad zur ausführbaren *assembly* (*.exe-Datei) der teilinstrumentierten *solution* erwartet. Die Instrumentierung der *assembly* mit den verbleibenden Zugriffsanweisungen `Load()` und `Store()` wird an die Klasse `AssemblyInstrumenter` des Pakets `AssemblyUtilities` delegiert. Die vollständig instrumentierte *assembly* (erkennbar durch die nachgestellte Zeichenkette _instr) wird ausgegeben. Die Ausführung der instrumentierten *assembly* muss nun per Hand angestoßen werden, um die Protokollierung der dynamischen Abhängigkeiten zu starten. Anschließend ist auf die Terminierung der Protokollierung zu warten, die letztlich die Datei `Dynamic-Dependencies.xml` ausgibt.

`SolutionProcessor.PhaseIII():` Als Eingabe wird der Pfad zur originalen *solution* (*.sln-Datei) der sequentiellen Anwendung und der Pfad zur Datei `Dynamic-Dependencies.xml` erwartet. Wie zuvor wird die *solution* in einen neuen Ordner (mit nachgestelltem _annot) kopiert und geöffnet. Dann wird die Durchführung der Phase III für jedes Dokument wieder an den zugehörigen `DocumentProcessor` delegiert. Dieser sucht nach Parallelisierungskandidaten, ermittelt statische Abhängigkeiten, lädt die dynamischen Abhängigkeiten, erkennt mögliche parallele Architekturen und annotiert diese. Es resultiert die mit Architekturbeschreibungen annotierte Anwendung.

`DocumentProcessor:` Im Folgenden wird auf die Klasse `DocumentProcessor` eingegangen, an die `SolutionProcessor` die eigentliche Arbeit der Phasen I und III delegiert. Die Phase II gibt es in der Klasse `DocumentProcessor` deshalb nicht, weil sich diese Phase nicht auf Quellcode, sondern übersetzte *assemblies* bezieht. Die Phasen I und III der Klasse `DocumentProcessor` gestalten sich wie folgt:

`DocumentProcessor.PhaseI():` In Phase I werden drei Schritte auf dem Syntaxbaum des zugrundeliegenden Dokuments ausgeführt, die schließlich zu einem teilinstrumentierten Dokument führen:

- Zunächst wird nach Sequenzen und Schleifen gesucht (`FindCandidates()`), die sich als Syntaxknoten der Ausprägung `BlockSyntax`, `WhileStatementSyntax` (für `while`-Schleifen), `ForStatementSyntax` (für `for`-Schleifen) und `ForEachStatementSyntax` (für `foreach`-Schleifen) im Syntaxbaum wiederfinden. Für jeden dieser Syntaxknoten wird eine neue Stellvertreter-Instanz (`TaskgraphCandidate` oder `PipelineCandidate`) erzeugt, die von der abstrakten Klasse `ParallelizationCandidate` erbt und deren abstrakte Methoden implementiert.
- Es folgt ein optionaler Schritt (`RunStaticAnalysis()`). Dabei wird die Durchführung der statischen Abhängigkeitsanalyse an jeden einzelnen Parallelisierungskandidaten delegiert. Dieser Schritt ist optional, weil er nur zur Optimierung des nächsten Schritts dient: in gewissen Konstellationen (wenn aufgrund der statisch ermittelten Abhängigkeiten schon klar ist, dass kein Parallelisierungsvorschlag ausgegeben wird) können Instrumentierungsanweisungen eingespart werden. Die ermittelten Abhängigkeiten werden nicht persistiert, sondern in Phase III einfach erneut erhoben.
- Jetzt wird die Teilinstrumentierung (`Instrument()`) angestoßen. Zunächst werden alle Schleifen in eine normalisierte Form gebracht um bei der anschließenden Instrumentierung auch die Laufbedingung der Schleife durch Abgrenzungsanweisungen umschließen zu können: falls die Laufbedingung auf globale Variablen zugreifen kann, wird vor der Schleife eine neue boolesche Variable deklariert und mit der Laufbedingung initialisiert. Die Laufbedingung der Schleife wird durch die boolesche Variable ersetzt. Am Ende des Schleifenrumpfs wird die boolesche Variable durch die Laufbedingung aktualisiert. Beispiel: Sei `g` eine globale Variable und die Schleife `while(g < 42) { A(g); g++; }` gegeben. Dann wird nach der Normalisierung daraus: `bool tmp = g < 42; while(tmp) { A(g); g++; tmp = g < 42; }`. Durch diese Normalisierung ist es jetzt auch möglich `tmp = g < 42` durch `Enter()` und `Leave()` zu umschließen, und den Lesezugriff auf `g` der Laufbedingung der Schleife zuzuordnen. Für diese Normalisierung wird die Klasse `SyntaxNormalizer` herangezogen. Die eigentliche Instrumentierung wird anschließend an die Parallelisierungskandidaten delegiert, welche die Klasse `SyntaxInstrumenter` verwenden, um dem Dokument Abgrenzungsanweisungen (Kapitel 4.7.4) hinzuzufügen.

`DocumentProcessor.PhaseIII():` In Phase III werden fünf Schritte auf dem Syntaxbaum des zugrundeliegenden Dokuments ausgeführt, die schließlich zu einem mit Architekturbeschreibungen annotierten Dokument führen:

- Wie bereits in Phase I werden zunächst die Parallelisierungskandidaten instanziiert (`FindCandidates()`).
- Es erfolgt die statische Abhängigkeitsanalyse (`RunStaticAnalysis()`), die diesmal nicht optional ist. Jeder Parallelisierungskandidat führt die Analyse für seinen Geltungsbereich aus und speichert die ermittelten Abhängigkeiten in seinem Abhängigkeitsgraph, der von der Klasse `NodeDependenceGraph` implementiert wird.

`TaskgraphCandidate` und `PipelineCandidate` implementieren dazu in der Methode `RunStaticAnalysis()` die Algorithmen aus 4.7.2 und 4.7.3.

- Im Anschluss wird jeder Parallelisierungskandidat aufgefordert, seinen Abhängigkeitsgraph um dynamisch ermittelte Abhängigkeiten zu ergänzen (`LoadDependencies()`). Dazu existiert eine zentrale Instanz der Klasse `DynamicDependencyProvider` welche die Datei `DynamicDependencies.xml` verwaltet.
- Daraufhin wird jeder Parallelisierungskandidat angewiesen, den jeweiligen Abbildungsalgorithmus unter Berücksichtigung der ermittelten Abhängigkeiten auszuführen (`DetectArchitecture()`). `DetectArchitecture()` implementiert für die Klasse `TaskgraphCandidate` den Algorithmus aus Kapitel 4.8.1, für `PipelineCandidate` den Algorithmus aus Kapitel 4.8.2.
- In einem letzten Schritt wird die Annotierung (Kapitel 4.9) jedes Parallelisierungskandidaten eingeleitet (`Annotate()`), wobei die Klasse `SyntaxAnnotator` die Funktionalität zum Hinzufügen von Annotierungen bereitstellt.

Außer den vorgestellten Methoden zur Erledigung der Aufgaben der Phasen, stellt die Klasse `DocumentProcessor` noch die Methode `StoreDependencies()` bereit, die Abhängigkeitsgraphen aller im Dokument enthaltenen Parallelisierungskandidaten in einer Datei pro Dokument persistieren kann.

5.1.2 Grundlegendes: Laden, Verändern und Speichern von Softwareprojekten, Dokumenten und Syntaxbäumen

Das Paket `Roslyn.Services` stellt Klassen zur Verwaltung von *solutions* bereit: `Solution`, `Projekt` und `Document` sind Repräsentanten für *solutions*, Projekte und Quellcodedateien. Eine *solution* enthält Projekte, ein Projekt Dokumente. Die von `Roslyn` bereitgestellten Klassen sind unveränderbar (engl. *immutable*): Das Ändern eines Dokuments erzeugt ein neues Dokument, dadurch ein neues Projekt und damit eine neue *solution*. Um eine Reihe von Änderungen auf ein Dokument anzuwenden, muss jede Änderung also stets auf die neue Instanz der *solution* angewandt werden.

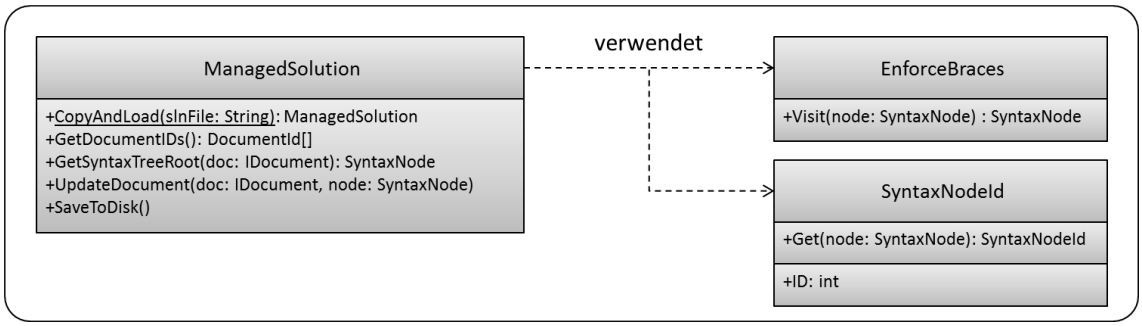

Abbildung 31: `ManagedSolution` ist einen Stellvertreter für eine *solution* und sichert über die Verwendung von `EnforceBraces` und `SyntaxNodeId` zu, dass jedes Dokument keine einzeiligen Blöcke verwendet und jeder Syntaxnoten einen (über Instanzen des gleichen Syntaxknoten) eindeutigen Bezeichner besitzt.

`ManagedSolution`: Um den typischen Lebenszyklus (Laden, mehrmals Verändern, Speichern) einer *solution* zu unterstützen und Flüchtigkeitsfehler durch das Verwenden einer alten *solution*-Instanz zu vermeiden wurde die Klasse `ManagedSolution` (Abbildung 31) eingeführt. Sie macht Dokumente nur in Form eindeutiger Bezeichner (`DocumentID`) verfügbar

(`GetDocumentIDs()`) und gibt nie Instanzen von `Document` nach außen weiter. Das Parsen des Quellcodes eines Dokuments sowie der Zugriff auf den resultierenden Syntaxbaum werden über `GetSyntaxTreeRoot()` ermöglicht. Um einen veränderten Syntaxbaum in ein Dokument und damit die *solution* einzubringen, muss `UpdateDocument()` aufgerufen werden. `SaveToDisk()` schreibt eine veränderte *solution* zurück ins Dateisystem. Die Klasse `ManagedSolution` stellt zudem folgendes sicher:

- Beim Laden (`CopyAndLoad()`) einer *solution* wird automatisch eine Kopie der Original-*solution* angelegt und schließlich die Kopie geladen. So kann die ursprüngliche *solution* nicht versehentlich verändert werden.
- Die *solution* wird vor der Verfügbarmachung eines Dokuments normalisiert. Jedes Dokument wird von `EnforceBraces` (eine Ableitung von `SyntaxRewriter`) besucht und gegebenenfalls umgeschrieben. `EnforceBraces` stellt sicher, dass beispielsweise einzeilige Schleifenrümpfe ohne umschließende Klammern mit Klammern umschlossen werden. Diese Zusicherung wird bei der Instrumentierung benötigt.
- Vor der Verfügbarmachung eines Dokuments wird jeder Syntaxknoten mit einer eindeutigen `SyntaxNodeID` versehen. Die Erläuterung zu diesem Schritt folgt nach dieser Aufzählung.

Das Paket `Roslyn.Compilers` stellt Klassen zur Repräsentation von Syntaxknoten (`SyntaxNode`), sowie Klassen zum Besuchen und Verändern von Syntaxknoten (`SyntaxRewriter`) bereit. Syntaxknoten sind wie *solutions* und Dokumente unveränderbar, weil sie eine Fassade der zugrundeliegenden veränderbaren Struktur darstellen. Auf die veränderbare Struktur ermöglicht `Roslyn` jedoch keinen Zugriff. Auf eine Änderung eines (veränderbaren) Syntaxknoten wird eine neue Fassaden-Instanz des Knoten zurückgegeben. Wegen der Referenzierung von Eltern- und Kindknoten geschieht dies gleich für jeden Knoten des kompletten Teilbaums. Für den vorgesehenen Anwendungsfall der Instrumentierung und der Annotierung ist dieses Verhalten sehr ungünstig: während der Architekturerkennung werden Informationen über die (unveränderten) Syntaxknoten gespeichert. Sobald sich ein Syntaxknoten während der Annotierung ändert, müssen diese Informationenen von den alten Knoten auf die neuen Knoten übertragen werden. Weil es sich bei den neuen Knoten um neue Instanzen handelt ist diese Übertragung nicht über einen Referenzvergleich möglich.

`SyntaxNodeId`: Abhilfe schafft die Klasse `SyntaxNodeId`, mit der jedem Syntaxknoten eine eindeutige Kennung zugeordnet wird. Solange kein Syntaxknoten verändert wird, kann mit Referenzvergleichen zwischen Syntaxknoten gearbeitet werden. Sobald eine Veränderung vorliegt, ist das Testen auf Gleichheit zweier Syntaxknoten nur noch über die Referenzgleichheit ihrer `SyntaxNodeId` möglich. Weil das Hinzufügen einer `SyntaxNodeId` zu einem Knoten ebenfalls eine Veränderung darstellt, muss sichergestellt sein, dass jeder Knoten mit einer `SyntaxNodeId` ausgestattet ist, bevor ein erster Vergleich über `SyntaxNodeId` ansteht. Dies wird durch die Klasse `ManagedSolution` sichergestellt. Die Klasse `SyntaxNodeId` dient zudem einem weiteren Zweck: für Teilinstrumentierung in der Phase I werden *solution*-weit eindeutige Bezeichner für Anweisungen (Syntaxknoten) benötigt. Die `SyntaxNodeId` eines Syntaxknoten kann als Argument für die Abgrenzungsanweisungen `Enter()`, `EnterIteration()` und `Leave()` herangezogen werden.

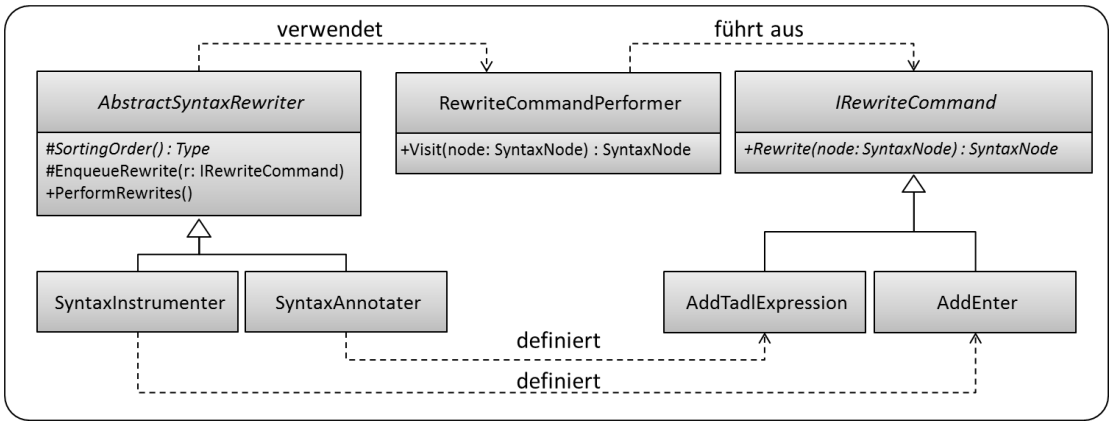

Abbildung 32: `AbstractSyntaxRewriter` ermöglicht es, Veränderungen am Syntaxbaum durch Veränderungsbefehle (`IRewriteCommand`) umzusetzen. Die konkreten Ausprägungen definieren konkrete Veränderungsbefehle und werden von den Parallelisierungskandidaten für die Instrumentierung und die Annotierung von Syntaxbäumen herangezogen.

`AbstractSyntaxRewriter`: Um die Verwendung von `SyntaxNodeIds` transparent zu halten und die Komplexität der Veränderung von Syntaxknoten für die Instrumentierung und die Annotierung zu verringern, wird die abstrakte Klasse `AbstractSyntaxRewriter` eingeführt. Sie stellt die Möglichkeit zur Verfügung, Veränderungen von Syntaxknoten in Form von Veränderungsbefehlen (`IRewriteCommand`) auszudrücken, die durch das Entwurfsmuster Befehl umgesetzt sind. Ein konkreter Veränderungsbefehl für eine Instanz eines Syntaxknoten kann über die Methode `EnqueueRewrite()` in eine Warteschlage hinzugefügt werden. Die Ausführung der Veränderungsbefehle wird erst durch die Methode `PerformRewrites()` angestoßen. Falls sich bei der Ausführung der Veränderungsbefehle mehrere Befehle für einen Syntaxknoten in der Warteschlage befinden, kann den Befehlen zudem eine typabhängige Ausführungsreihenfolge zugeordnet werden. Diese Ausführungsreihenfolge ist über die abstrakte Methode `SortingOrder()` festzulegen. Die Ausführung der Veränderungsbefehle wird durch die von `SyntaxRewriter` abgeleitete Klasse `RewriteCommandPerformer` umgesetzt, welche die Methode `Visit()` überschreibt und dabei auf `SyntaxNodeId` zurückgreift um auf Gleichheit von Knoten zu testen. Nach der Ausführung der Befehle wird der veränderte Syntaxbaum automatisch zurück in das Dokument und die zugrundeliegende `ManagedSolution` geschrieben.

Die Klassen zur Instrumentierung (`SyntaxInstrumenter`) und Annotierung (`SyntaxAnnotater`), sowie die Klasse `SyntaxNormalizer`, nehmen die Funktionalität von `AbstractSyntaxRewriter` in Anspruch, indem sie von der Klasse erben, konkrete Veränderungsbefehle definieren und die typabhängige Ausführungsreihenfolge dieser Befehle festlegen. Die Logik zur Erzeugung von Veränderungsbefehlen wird von den Parallelisierungskandidaten `TaskgraphCandidate` und `PipelineCandidate` implementiert.

5.1.3 Parallelisierungskandidaten (`ParallelizationCandidate`)

Die abstrakte Klasse `ParallelizationCandidate` implementiert die Methoden `IsOpen()`, `IsOpenThroughReturn()`, `IsOpenThroughBreak()` und `IsOpenThroughContinue()` zur Identifizierung offener Syntaxknoten (Kapitel 4.7.2). Diese Methoden sind unabhängig von den konkreten Kandidaten zur Parallelisierung und werden von `TaskgraphCandidate` und `PipelineCandidate` wiederverwendet. Zudem definiert die Klasse `ParallelizationCandidate` im Wesentlichen vier Methoden, die von einer konkreten Ausprägung eines

Kandidaten implementiert werden müssen: `RunStaticAnalysis()` (statische Abhängigkeitsanalyse, Kapitel 4.7.3), `Instrument()` (Instrumentierung für die dynamische Abhängigkeitsanalyse, Kapitel 4.7.4), `DetectArchitecture()` (Architekturerkennung, Kapitel 4.8) und `Annotate()` (Annotierung, Kapitel 4.9). Zudem implementiert jeder konkrete Parallelisierungskandidat die statische Methode `Create()`, welche Instanzen erzeugt und zurückgibt.

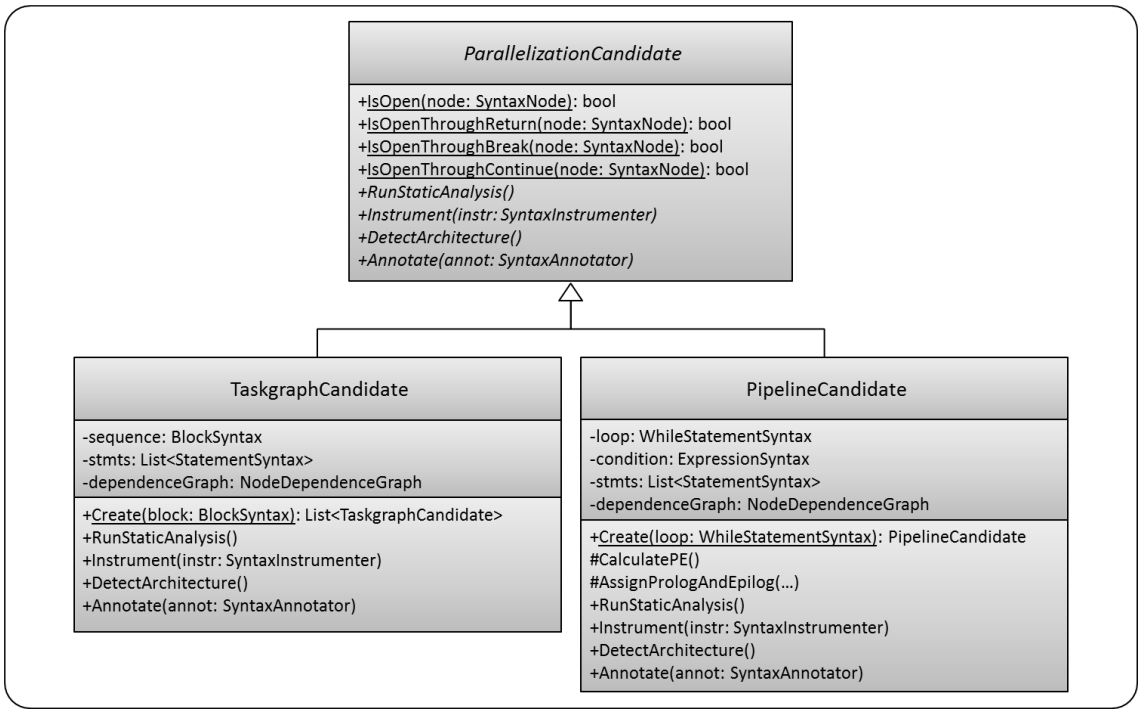

Abbildung 33: `ParallelizationCandidate` ist eine abstrakte Klasse für welche die zwei Ausprägungen `TaskgraphCandidate` und `PipelineCandidate` implementiert wurden.

`TaskgraphCandidate`: Eine Instanz der Klasse `TaskgraphCandidate` ist Repräsentant für eine Teilsequenz ohne offene Syntaxknoten, die durch einen Taskgraphen parallelisiert werden soll. Sie kapselt daher einen Syntaxknoten `sequence` vom Typ `BlockSyntax`, der die Sequenz repräsentiert und eine Liste von Anweisungen `stmts`, welche die Anweisungen der Teilsequenz repräsentiert. Um Abhängigkeiten zwischen den Anweisungen der Teilsequenz abzulegen, ist einem Taskgraph-Kandidaten ein Abhängigkeitsgraph `dependenceGraph` der Klasse `NodeDependenceGraph` zugeordnet.

Die statische Methode `Create()` leitet aus einer Sequenz mehrere Teilsequenzen ohne offene Syntaxknoten ab und gibt sie als Liste von Instanzen der Klasse `TaskgraphCandidate` zurück. Sie implementiert damit den 1. Schritt des Abbildungsalgorithmus aus Kapitel 4.8.1.

Die Methode `RunStaticAnalysis()` implementiert die Abhängigkeitsanalyse für lokale Variablen, wie sie in Kapitel 4.7.3, Abbildung 14 vorgestellt wurde. Lese- und Schreibmengen werden durch die Datenflussanalyse (`DataFlowAnalysis`) von `Roslyn` zur Verfügung gestellt. Lokale Variablen und Methodenparameter werden durch die Klasse `Symbol` repräsentiert. Die letzte schreibende Anweisung LW(V) und die letzten lesenden Anweisungen LR(V) einer Variable V werden jeweils in einem `Dictionary` hinterlegt, welches als Schlüssel die Variable V entgegennimmt und LW(V) beziehungsweise LR(V) als Wert zurückgibt.

Die Methode `Instrument()` sorgt dafür, dass jeder Anweisung der Teilsequenz die Anweisung `Logger.Enter(ID)` voran- und die Anweisung `Logger.Leave(ID)` nachgestellt wird. Das Argument `ID` ist dabei die eindeutige `SyntaxNodeId`, die der Anweisung beim

Laden der *solution* (`ManagedSolution`) zugewiesen wurde. Die Methode verwendet dazu eine `SyntaxInstrumenter`-Instanz, die dazu die Methoden `AddEnterBefore()` und `AddLeaveAfter()` bereitstellt, welche Instanzen entsprechender Veränderungsbefehle erzeugen und in die Warteschlange (auf die bei der Beschreibung der abstrakten Klasse `AbstractSyntaxRewriter` eingegangen wurde) einfügen. Falls zum Zeitpunkt der Instrumentierung schon einige Abhängigkeiten bekannt sind, können die umschließenden Anweisungen einer Anweisung A eingespart werden, falls es zwischen A und allen anderen Anweisungen der Teilsequenz schon eine Abhängigkeit gibt.

Die Methode `DetectArchitekture()` implementiert den 2. Schritt des Abbildungsalgorithmus aus Kapitel 4.8.1. Aus Gründen der Übersichtlichkeit wurde die Logik dazu in eine private Klasse `TaskGraph` ausgelagert. Falls die Abbildung auf einen Taskgraph möglich ist, wird eine Instanz dieser Klasse angelegt. Diese Instanz definiert Bezeichner für alle enthaltenen Tasks und ermöglicht die Ausgabe des TADL-Ausdrucks.

Die Methode `Annotate()` verwendet die erzeugte Instanz des Taskgraphen (`TaskGraph`) und sorgt dafür, dass der zugrundeliegende Quellcode annotiert wird. Der umzuformende Codeabschnitt wird durch `#region`- und `#endregion`-Marker umschlossen. Der TADL-Ausdruck dem `#region`-Marker nachgestellt. Die Aktivität jedes Tasks wird durch umschließende `#region`- und `#endregion`-Marker gekennzeichnet und mit dem eindeutigen Bezeichner des Tasks versehen. Dazu wird eine `SyntaxAnnotater`-Instanz herangezogen, welche die Methoden `AddRegionTadlBefore()` und `AddRegionLabelBefore()` bereitstellt, welche wiederum Instanzen von Veränderungsbefehlen in die Warteschlange einreihen.

`PipelineCandidate`: Eine Instanz der Klasse `PipelineCandidate` ist Repräsentant für eine Schleife, die durch ein Fließband parallelisiert werden soll. Sie kapselt daher einen Syntaxknoten `loop` vom Typ `WhileStatementSyntax` (für `while`-Schleifen), der die Schleife repräsentiert, einen Knoten `condition` vom Typ `ExpressionSyntax`, der die Laufbedingung repräsentiert und eine Liste von Anweisungen `stmts`, die die Anweisungen des Schleifenrumpfs repräsentieren. Um Abhängigkeiten zwischen der Laufbedingung und den Anweisungen des Schleifenrumpfs abzulegen, ist einem Fließband-Kandidaten ein Abhängigkeitsgraph `dependenceGraph` der Klasse `NodeDependenceGraph` zugeordnet.

Die statische Methode `Create()` gibt eine neue Instanz eines Fließband-Kandidaten für eine Schleife zurück und belegt die Variablen `loop`, `condition` und `stmts` für den konkreten Schleifentyp (`while`-, `for`- oder `foreach`-Schleife).

Die Methode `RunStaticAnalysis()` ermittelt Kontrollflussabhängigkeiten die durch offene Syntaxknoten zustande kommen und implementiert die Abhängigkeitsanalyse für lokale Variablen, wie sie in Kapitel 4.7.3 vorgestellt wurde: es werden zwei Iterationen des Schleifenrumpfs simuliert. Lese- und Schreibmengen, sowie LW(V) und LR(V) sind wie in der Klasse `TaskgraphCandidate` implementiert, wobei LW(V) und LR(V) zu jedem letzten Schreibenden oder Lesenden zusätzlich die Iteration vorhalten, in welcher der Zugriff stattfand. Dadurch können iterationsübergreifende Abhängigkeiten ermittelt werden.

Die Methode `Instrument()` sorgt dafür, dass jeder Anweisung des Schleifenrumpfs die Anweisung `Logger.Enter(ID)` voran- und die Anweisung `Logger.Leave(ID)` nachgestellt wird. Das Argument `ID` ist wieder die eindeutige `SyntaxNodeId`, die der Anweisung zugewiesen wurde. Zusätzlich wird die Schleife `loop` selbst von `Logger.Enter(ID)` und `Logger.Leave(ID)` umschlossen und der ersten Anweisung des Schleifenrumpfs die An-

weisung `Logger.EnterIteration(ID)` vorangestellt. Dabei ist `ID` die `SyntaxNodeId` der Schleife `loop`. Die Funktionalität zum Hinzufügen dieser Anweisungen stellt wiederum eine `SyntaxInstrumenter`-Instanz zur Verfügung.

Die Methode `DetectArchitekture()` implementiert die vier Schritte des Abbildungsalgorithmus aus Kapitel 4.8.2. Die Methode `CalculatePE()` ermittelt die Menge PE der Anweisungen, die aufgrund von Abhängigkeiten zur Laufbedingung der Erzeuger-Stufe zugeordnet werden müssen. Die Methode `AssignPrologAndEpilog()` weist die Anweisungen aus PE dem Prolog oder dem Epilog der Erzeuger-Stufe zu. Die restliche Logik (zusammenfassen iterationsübergreifend abhängiger Stufen, Entscheidung über Replizierbarkeit) wurde in die private Klasse `Pipeline` ausgelagert. Falls die Abbildung auf ein Fließband möglich ist, wird eine Instanz dieser Klasse angelegt. Die Instanz definiert Bezeichner für alle enthaltenen Stufen und ermöglicht die Ausgabe des TADL-Ausdrucks.

Die Methode `Annotate()` verwendet die erzeugte Instanz des Fließbandes (`Pipeline`) und sorgt dafür, dass der zugrundeliegende Quellcode annotiert wird. Der umzuformende Codeabschnitt wird durch `#region`- und `#endregion`-Marker umschlossen. Der TADL-Ausdruck dem `#region`-Marker nachgestellt. Prolog und Epilog der Erzeuger-Stufe werden gekennzeichnet. Die Aktivität jeder weiteren Stufe wird durch umschließende `#region`- und `#endregion`-Marker gekennzeichnet und mit dem eindeutigen Bezeichner der Stufe versehen. Eine `SyntaxAnnotater`-Instanz stellt die benötigten Methoden zum Hinzufügen dieser Annotationen bereit.

5.1.4 Abhängigkeitsgraph (`NodeDependenceGraph`)

Eine Anweisung kann auf zwei Arten angegeben werden: durch eine Instanz eines Syntaxknoten oder die eindeutige `SyntaxNodeId`. Deshalb gibt es auch zwei Implementierungen des Abhängigkeitsgraphen: `IdDependenceGraph` codiert Abhängigkeiten zwischen `SyntaxNodeIds`, `NodeDependenceGraph` kapselt einen `IdDependenceGraph` und macht Abhängigkeiten zwischen den zugehörigen Instanzen von Syntaxknoten verfügbar.

Eine Abhängigkeit zwischen zwei Knoten kann verschiedene Eigenschaften besitzen, die im Aufzählungstyp (engl. *enumeration*) `DependencyKind` abgelegt sind:

- `None`: keine Abhängigkeit
- `LoopCarried`: iterationsübergreifende Abhängigkeit
- `RAW`: Lese-nach-Schreib-Abhängigkeit (engl. *read-after-write*)
- `WAR`: Schreib-nach-Lese-Abhängigkeit (engl. *write-after-read*)
- `WAW`: Schreib-nach-Schreib-Abhängigkeit (engl. *write-after-write*)
- `ControlFlow`: Kontrollflussabhängigkeit
- `Transitiv`: transitive Abhängigkeit (durch Berechnung der transitiven Hülle)

Jeder Parallelisierungskandidat besitzt seinen eigenen Abhängigkeitsgraph. Abhängigkeiten werden immer nur zwischen bestimmten Anweisungen ermittelt. Bei Taskgraph-Kandidaten sind das die Anweisungen der Teilsequenz (`stmts`), bei Schleifen die Laufbedingung (`condition`) und die Anweisungen des Schleifenrumpfs (`stmts`). Diese Anweisungen werden im Feld `SyntaxNodeIds` (bei `IdDependenceGraph`) vorgehalten. Abhängigkeiten werden dann in der zweidimensionalen Abhängigkeitsmatrix `dependenceFromTo` vom Typ

`DependencyKind` gespeichert: `dependenceFromTo[1, 3] = RAW` bedeutet, dass die Anweisung `SyntaxNodeIds[1]` von der Anweisung `SyntaxNodeIds[3]` RAW-abhängig ist.

Ist `dependencyProvider` die Instanz der Klasse `DynamicDependencyProvider`, welche die Datei `DynamicDependencies.xml` verwaltet. Dann können einem Abhängigkeitsgraphen `graph` dynamisch ermittelte Abhängigkeiten hinzugefügt werden, indem die Methode `dependencyProvider.LoadDependencies(graph)` aufgerufen wird.

5.2 Das Paket AssemblyUtilities

Auf die Funktionalität des Pakets `AssemblyUtilities` wird in der Phase II der Klasse `SolutionProcessor` zurückgegriffen um eine teilinstrumentierte *solution* mit den Zugriffsanweisungen `Load()` und `Store()` auszustatten. Wieso die Instrumentierung mit diesen Anweisungen erst auf IL-Ebene stattfindet, wird in Kapitel 5.2.1 geklärt. Zur Erinnerung: `Load()` und `Store()` sollen vor jedem lesenden beziehungsweise schreibenden Zugriff auf Klassen- und Instanzvariablen, sowie Feldelementen hinzugefügt werden (Kapitel 4.7.4). Das Argument dieser Anweisungen ist so zu wählen, dass die zugrundeliegende Speicherstelle der gelesenen oder geschriebenen Variablen eindeutig identifiziert werden kann. Konkret werden daher die folgenden sechs Zugriffsanweisungen unterschieden:

- `LoadStaticField(int staticFieldId)`: Eine Klassenvariable wird gelesen. Als eindeutiger Bezeichner zur Identifizierung der zugrundeliegenden Speicherstelle wird eine ID verwendet, die jede statische Variable eindeutig macht.
- `StoreStaticField(int staticFieldId)`: Wie `LoadStaticField()`, nur schreibender Zugriff.
- `LoadField(Object object, int fieldId)`: Eine Instanzvariable wird gelesen. Als eindeutiger Bezeichner zur Identifizierung der zugrundeliegenden Speicherstelle wird das Objekt (die Instanz) und eine ID und verwendet, die jede Instanzvariable eindeutig macht.
- `StoreField(Object object, int fieldId)`: Wie `LoadField()`, nur schreibender Zugriff.
- `LoadArrayElement(Object object, int idx)`: Ein Feldelement wird gelesen. Als eindeutiger Bezeichner zur Identifizierung der zugrundeliegenden Speicherstelle wird das Objekt (die Instanz) und der Feldindex verwendet.
- `StoreArrayElement(Object object, int idx)`: Wie `LoadArrayElement()`, nur schreibender Zugriff.

Für die Generierung eindeutiger IDs ist die Klasse `UniqueIdProvider` verantwortlich. Sie bietet die Methoden `GetStaticFieldId()` und `GetFieldId()` an, die als Argument einen Repräsentanten einer Klassen- beziehungsweise Instanzvariable vom Typ `IFieldReference` erwarten.

`AssemblyInstrumenter`: Die Instrumentierung wird über die Methode `Instrument()` der Klasse `AssemblyInstrumenter` angestoßen. Daraufhin wird die angegebene *assembly* geöffnet und eine veränderbare Metadatenstruktur aufgebaut, die die *assembly* repräsentiert. Anschließend wird eine Instanz von `UserModuleRewriter` erzeugt, die intern eine Instanz von `MemoryAccessInstrumenter` verwendet. Die aufgebaute Metadatenstruktur wird nun

vom `UserModuleRewriter` besucht, der wiederum eine Instanz von `MemoryAccess-Instrumenter` verwendet.

`MemoryAccessInstrumenter`: Weil `MemoryAccessInstrumenter` `EmitMethodBody()` und `EmitOperation()` der von `Microsoft.Cci` angebotenen Klasse `ILRewriter` überschreibt, können Methodenrümpfe und Anweisungen auf Zwischensprachenebene (engl. *intermediate language*) instrumentiert werden. Für das Verständnis des Folgenden ist zu beachten, dass die Zwischensprache stapelbasiert ist: Variablen werden zwischen Anweisungen ausgetauscht, indem sie auf den Stapel gelegt und wieder davon entfernt werden. In der Methode `EmitMethodBody()` wird dafür gesorgt, dass die Deklaration `.maxstack` angepasst wird: dies ist nötig, weil bei der Instrumentierung möglicherweise neue lokale Variablen eingeführt werden, für die Speicher auf dem Stapel reserviert werden muss. Weil maximal zwei neue lokale Variablen eingeführt werden, genügt es `.maxstack` für jeden Methodenrumpf um zwei zu erhöhen.

In der Methode `EmitOperation(IOperation op)` erfolgt die eigentliche Instrumentierung: in Abhängigkeit der IL-Anweisung `op` werden verschiedene Instrumentierungsanweisungen ausgegeben. Die sechs Fälle sind den folgenden Tabellen dargestellt: auf die fett dargestellte IL-Anweisung folgt eine kurze Erläuterung zu deren Funktion, die beispielsweise auch dem .NET-Standard [ECMA-335] entnommen werden kann. Ausgegeben werden die unter Ausgabe aufgeführten IL-Anweisungen, denen in Form von Kommentaren eine Erklärung hinzugefügt ist (`[RU]` steht für `[RuntimeUtilities]`).

Anweisung	**`ldsfld <type> <staticField>`**
Erläuterung	Der Wert vom Typ `<type>` des statischen Felds `<staticField>` wird auf den Stapel gelegt. Sei `<staticFieldId>` die eindeutige ID des statischen Felds, die über den Aufruf `GetStaticFieldId(<staticField>)` an `UniqueIdProvider` zurückgegeben wird.

Ausgabe	`// <staticFieldId> auf den Stapel legen (für Logger-Aufruf)`
	`ldc.i4 <staticFieldId>`
	`// Logger aufrufen (konsumiert <staticFieldId>)`
	`call void [RU]RuntimeUtilities.Logger::LoadStaticField(int32)`
	`// Originalanweisung ausführen`
	`ldsfld <type> <staticField>`

Anweisung	**`stsfld <type> <staticField>`**
Erläuterung	Das oberste Element des Stapels wird vom Stapel genommen und als `value` bezeichnet. Der Wert des statischen Felds `<staticField>` wird auf `value` gesetzt. Sei `<staticFieldId>` wiederum die eindeutige ID des statischen Felds.

Ausgabe	`// <staticFieldId> auf den Stapel legen (für Logger-Aufruf)`
	`ldc.i4 <staticFieldId>`
	`// Logger aufrufen (konsumiert <staticFieldId>)`
	`call void [RU]RuntimeUtilities.Logger::LoadStaticField(int32)`
	`// Originalanweisung ausführen`
	`ldsfld <type> <staticField>`

Anweisung	**`ldfld <type> <field>`**
Erläuterung	Das oberste Element des Stapels wird vom Stapel genommen und als `object` bezeichnet. Der Wert des Felds `object.<field>` wird auf den Stapel gelegt.

	Sei `<fieldId>` die eindeutige ID des Feldes, die über den Aufruf `GetFieldId(<field>)` an `UniqueIdProvider` zurückgegeben wird.

Ausgabe	`// <object> duplizieren (für Logger-Aufruf)`
	`dup`
	`// <fieldId> auf den Stapel legen (für Logger-Aufruf)`
	`ldc.i4 <fieldId>`
	`// Logger aufrufen (konsumiert <object> und <fieldId>)`
	`call void [RU]RuntimeUtilities.Logger::LoadField(object, int32)`
	`// Originalanweisung ausführen`
	`ldfld <type> <field>`

Anweisung	**`stfld <type> <field>`**
Erläuterung	Das oberste Element wird vom Stapel genommen und als `<value>` bezeichnet. Das nächste Element wird vom Stapel genommen und als `<object>` bezeichnet. Der Wert des Felds `<object>.<field>` wird auf `<value>` gesetzt. Sei `<fieldId>` wiederum die eindeutige ID des Felds. Um den Objektzeiger `<object>` duplizieren zu können, muss zunächst der Wert `<value>` vom Stapel genommen und in einer neu eingeführten lokalen Variable zwischengespeichert werden. Diese Variable sei über `<slot>` zugreifbar.

Ausgabe	`// <value> vom Stapel nehmen und in <slot> zwischenspeichern`
	`stloc.s <slot>`
	`// <object> duplizieren (für Logger-Aufruf)`
	`dup`
	`// <fieldId> auf den Stapel legen (für Logger-Aufruf)`
	`ldc.i4 <fieldId>`
	`// Logger aufrufen (konsumiert <object> und <fieldId>)`
	`call void [RU]RuntimeUtilities.Logger::StoreField(object, int32)`
	`// <value> zurück auf den Stapel legen`
	`ldloc.s <slot>`
	`// Originalanweisung ausführen`
	`stfld <type> <field>`

Anweisung	**`ldelem`**
Erläuterung	Das oberste Element wird vom Stapel genommen und als `<index>` bezeichnet. Das nächste Element wird vom Stapel genommen und als `<array>` bezeichnet. Der Wert `<array>[<index>]` wird auf den Stapel gelegt. Um den Arrayzeiger `<array>` duplizieren zu können, muss zunächst der Index `<index>` vom Stapel genommen und in einer neuen lokalen Variable zwischengespeichert werden. Diese Variable sei über `<slot>` zugreifbar.

Ausgabe	`// <index> vom Stapel nehmen und in <slot> zwischenspeichern`
	`stloc.s <slot>`
	`// <array> duplizieren (für Logger-Aufruf)`
	`dup`
	`// <index> auf den Stapel legen (für Logger-Aurfuf)`
	`ldloc.s <slot>`
	`// Logger aufrufen (konsumiert <array> und <index>)`
	`call void [RU]RuntimeUtilities.Logger::LoadArrayElement(object, int32)`
	`// <index> zurück auf den Stapel legen`
	`ldloc.s <slot>`
	`// Originalanweisung ausführen`
	`ldelem`

Anweisung	**stelem**
Erläuterung	Das oberste Element wird vom Stapel genommen und als <value> bezeichnet. Das nächste Element wird vom Stapel genommen und als <index> bezeichnet. Schließlich wird das nächste Element vom Stapel genommen und als <array> bezeichnet. Das Feldelement <array>[<index>] wird auf <value> gesetzt. Um den Arrayzeiger <array> duplizieren zu können, müssen zunächst der Wert <value> und dann der Index <index> vom Stapel genommen und in neuen lokalen Variablen zwischengespeichert werden. Diese Variablen seien über <slot1> (für <value>) und <slot2> (für <index>) zugreifbar.

Ausgabe	`// <value> vom Stapel nehmen und in <slot1> zwischenspeichern`
	`stloc.s <slot1>`
	`// <index> vom Stapel nehmen und in <slot2> zwischenspeichern`
	`stloc.s <slot2>`
	`// <array> duplizieren (für Logger-Aufruf)`
	`dup`
	`// <index> auf den Stapel legen (für Logger-Aufruf)`
	`ldloc.s <slot2>`
	`// Logger aufrufen (konsumiert <array> und <index>)`
	`call void [RU]RuntimeUtilities.Logger::StoreArrayElement(object, int32)`
	`// <index> zurück auf den Stapel legen`
	`ldloc.s <slot2>`
	`// <value> zurück auf den Stapel legen`
	`ldloc.s <slot1>`
	`// Originalanweisung ausführen`
	`stelem`

Notiz: Zugriffe auf Klassen- und Instanzvariablen, die als `readonly` markiert sind, müssen nicht instrumentiert werden: durch den Übersetzer ist sichergestellt, dass der einzige Schreibzugriff auf diese Variablen im Konstruktor der Klasse stattfindet.

5.2.1 Warum IL- anstatt Quellcode-Instrumentierung?

In diesem Abschnitt soll geklärt werden, wieso die Zugriffsanweisungen `Load()` und `Store()` im Gegensatz zu den Abgrenzungsanweisungen `Enter()`, `Leave()` und `EnterIteration()` auf Basis von IL-Code und nicht auf Basis von Quellcode instrumentiert werden.
Eine sequentielle Anwendung die als *solution* vorliegt, besteht im Allgemeinen aus Quellcode und referenzierten *assemblies* für die kein Quellcode vorliegt. Die Anweisungen `Load()` und `Store()` müssen vor allen Speicherzugriffen auf Klassen- und Instanzvariablen und Feldelementen hinzugefügt werden, weil Anweisungen im Quelltext auch aufgrund von Speicherzugriffen in referenzierten *assemblies* abhängig sein können. Es genügt also nicht, nur diejenigen Speicherzugriffe zu instrumentieren, die im Quellcode ersichtlich sind.
Jetzt stellt sich möglicherweise die Frage, wieso die Abgrenzungsanweisungen nicht ebenfalls auf Basis von IL-Code eingefügt werden. Dies ist durch das Konzept begründet: Parallelisierungsvorschläge sollen nur für Quellcode ausgegeben werden. Daher genügt es auch, nur zwischen Quellcodeanweisungen Datenabhängigkeiten zu ermitteln. Die Abgrenzungsanweisungen dienen dazu, den Eintritt in und den Austritt aus einer Quellcodeanweisung zu signalisieren. Der Ein- und Austritt aus IL-Anweisungen ist irrelevant.

Notiz: Aus technischen Gründen wird momentan nur die vom Benutzer in Phase II angegebene *assembly* instrumentiert. Weitere *assemblies*, die von dieser *assembly* referenziert werden (wie beispielsweise die Standardbibliotheken `mscorlib.dll` und `system.dll`), müssten ebenfalls

instrumentiert werden, um auch dort Speicherzugriffe und damit Abhängigkeiten protokollieren zu können.

5.3 Das Paket RuntimeUtilities

Auf die Funktionalität des Pakets `RuntimeUtilities` wird während der Protokollierung bei der dynamischen Abhängigkeitsanalyse (und damit zur Laufzeit) zurückgegriffen. Die instrumentierte ausführbare sequentielle Anwendung (die nach Phase II vorliegt) besitzt Signalisierungsanweisungen, die auf öffentliche statische Methoden der Klasse `Logger` verweisen. Die Methoden der Klasse `Logger` reagieren entsprechend der bereits in Kapitel 4.7.4 (Protokollierung) angedeuteten Regeln, um einen dynamischen Aufruf- und Abhängigkeitsgraphen aufzubauen. In einem Nachverarbeitungsschritt (der in der Klasse `DependencyExtractor` implementiert ist) werden aus diesem Graphen dann Abhängigkeiten zwischen Syntaxknoten abgeleitet.

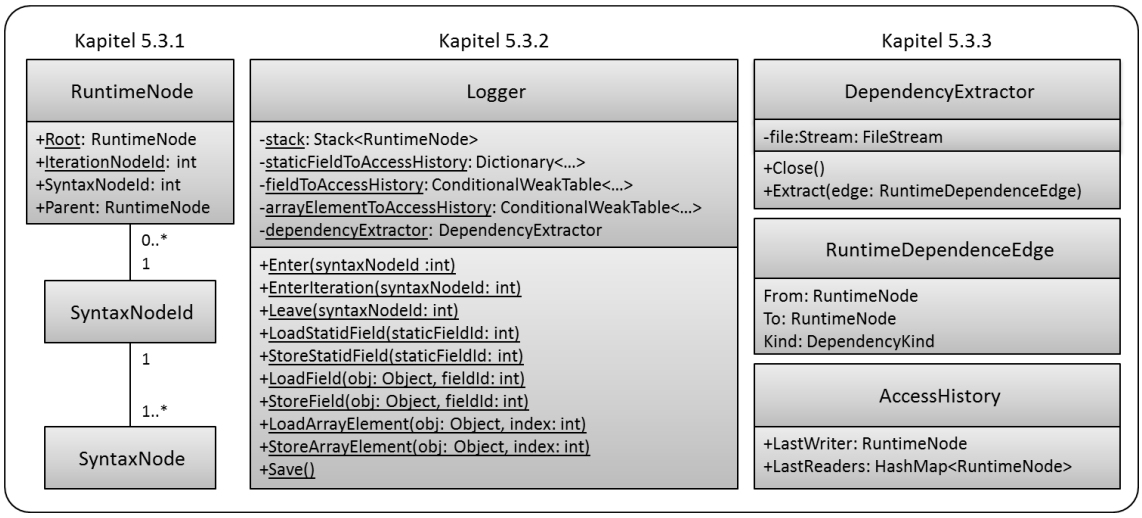

Abbildung 34: `Logger` ist die zentrale Klasse, in der die Methoden implementiert sind, die durch die Singalisierungsanweisungen aufgerufen werden. Durch diese Methoden wird der dynamische Aufruf- und Abhängigkeitsgraph aufgebaut, dessen Knoten Instanzen von `RuntimeNode` sind. Durch die Klasse `DependencyExtractor` wird die Datei `DynamicDependencies.xml` erzeugt.

5.3.1 Laufzeitrepräsentanten (`RuntimeNode`)

`RuntimeNode`: Knoten des dynamischen Aufruf- und Abhängigkeitsgraphen sind Laufzeitrepräsentanten von Syntaxknoten, und vom Typ `RuntimeNode`. Eine Instanz von `RuntimeNode` ist über dessen Feld `SyntaxNodeId` einem Syntaxknoten zuzuordnen. Für einen Syntaxknoten kann es während der Protokollierung aber mehrere Laufzeitrepräsentanten geben. `RuntimeNode` besitzt zudem das Feld `Parent`, welches verwendet wird, um Aufrufbeziehungen zu codieren: ist `r.Parent = t`, dann ist `t` „Aufrufer“ von `r`. Letztlich besitzt die Klasse `RuntimeNode` ein statisches Feld, welches den ausgezeichneten Wurzelknoten `RootNode` darstellt. Zu `RootNode` gibt es keinen zugehörigen Syntaxknoten.

Notiz: Der Wurzelknoten `RootNode` wird zur Optimierung verwendet: er wird zu Beginn auf den Stapel `stack` gelegt und nie heruntergenommen. Dadurch muss bei Operationen auf dem Stapel nicht ständig überprüft werden, ob der Stapel gerade leer ist.

An dieser Stelle ist zu erwähnen, dass die Richtung der Verzeigerung zwischen Instanzen von `RuntimeNode` mit Bedacht gewählt wurde, um das Aufräumen (engl. *garbage collection*) von `RuntimeNode`-Instanzen von Seiten der automatischen Speicherverwaltung zu ermöglichen: Bei einer Verzeigerung in die andere Richtung wäre eine Instanz von `RuntimeNode` stets über den Wurzelknoten erreichbar. Weil dieser immer auf dem Stapel liegt, könnte keine Instanz von `RuntimeNode` aufgeräumt werden. Mit der gewählten Verzeigerung ist das Aufräumen möglich, sobald: (i) die Instanz nicht mehr auf dem Stapel liegt, (ii) kein Zugriffstabelleneintrag auf die Instanz zeigt, (iii) keine Laufzeitabhängigkeit auf die Instanz zeigt.

5.3.2 Protokollierung (`Logger`)

`Logger`: Der statische Konstruktor der Klasse Logger erzeugt eine Instanz von `DependencyExtractor`, welche die Datei `DynamicDependencies.xml` verwaltet und dort Abhängigkeiten zwischen Syntaxknoten einträgt. Um diese Datei ordnungsgemäß zu schließen, muss `Logger.Save()` aufgerufen werden, wodurch wiederum die Methode `Close()` der Klasse `DependencyExtractor` getriggert wird. Deshalb muss der Aufruf `Logger.Save()` nach der Phase I (siehe Kapitel 5.1.1) per Hand an allen Stellen, die zur Beendigung der Anwendung führen, eingefügt werden.

Um den dynamischen Aufruf- und Abhängigkeitsgraphen aufbauen zu können, verwaltet die Klasse `Logger` vier wichtige private Datenstrukturen:

- **`stack`:** Der Aufrufstapel dient dazu, die Aufrufhierarchie abbilden zu können. Auf dem Stapel befinden sich alle Laufzeitrepräsentanten, die momentan „aktiv“ sind, sich also in Ausführung befinden. `Enter()`, `Leave()` und `EnterIteration()` sorgen dafür, dass Laufzeitrepräsentanten auf den Aufrufstapel gelegt oder von ihm heruntergenommen werden.
- **`staticFieldToAccessHistory`:** Die Zugriffstabelle dient dazu, LW(V) und LR(V) für alle Klassenvariablen vorzuhalten. Als Datenstruktur wird ein `Dictionary` verwendet. Als Schlüssel wird der eindeutige Bezeichner (vgl. Kapitel 5.2) von V (`staticFieldId`) erwartet, woraufhin als Wert eine Instanz der Klasse `AccessHistory` zurückgegeben wird, die den letzten Schreibenden `LastWriter` und die letzten Lesenden `LastReaders` vorhält.
- **`fieldToAccessHistory`:** Die Zugriffstabelle dient dazu, LW(V) und LR(V) für alle Instanzvariablen V vorzuhalten. Als Datenstruktur wird kein `Dictionary`, sondern ein `ConditionalWeakTable` verwendet. Als Schlüssel wird der erste Teil des Bezeichners von V (`object`) erwartet, woraufhin als Wert eine Instanz eines `Dictionary` zurückgegeben wird. Dieses wiederum erwartet den zweiten Teil des Bezeichners von V (`fieldId`) und gibt daraufhin eine Instanz der Klasse `AccessHistory` zurück.
- **`arrayElementToAccessHistory`:** Die Zugriffstabelle dient dazu, LW(V) und LR(V) für alle Feldelemente V vorzuhalten. Es wird ebenfalls `ConditionalWeakTable` verwendet, welche als Schlüssel den ersten Teil des Bezeichners von V (`object`) erwartet. Zurückgegeben wird ein `Dictionary`, welches den zweiten Teil des Bezeichners (`idx`) als Schlüssel erwartet und wiederum eine Instanz der Klasse `AssessHistory` zurückgibt.

Notiz: Auf die Verwendung der Datenstruktur `ConditionalWeakTable` wird deshalb explizit hingewiesen, weil ohne diese Datenstruktur kein Aufräumen von Objekten von Seiten der automatischen Speicherverwaltung mehr möglich wäre.

Im Folgenden wird die Funktionsweise der Methoden der Klasse `Logger` eingegangen, die auf die Signalisierungsanweisungen reagieren, die der Anwendung während der Instrumentierung hinzugefügt wurden. Von den `Load()`- und `Store()`-Methoden wird nur die Methode `LoadField()` exemplarisch vorgestellt:

- **`Enter(int syntaxNodeId):`** `Enter()` erzeugt beim Betreten eine Instanz `r` von `RuntimeNode` mit `r.SyntaxNodeId = syntaxNodeId`. Zudem wird dafür gesorgt, dass das Feld `r.Parent` auf denjenigen Laufzeitrepräsentanten gesetzt wird, der beim Betreten von `Enter()` an oberster Stelle auf dem Aufrufstapel `stack` liegt. Solange ein Laufzeitrepräsentant auf dem Aufrufstapel liegt, können ihm gemeldete Speicherzugriffe (über `Load()` und `Store()`) zugerechnet werden.
- **`Leave(int syntaxNodeId):`** `Leave(syntaxNodeId)` bildet das Gegenstück zu `Enter(syntaxNodeId)`. Diese Aufrufe wurden bei der Instrumentierung paarweise eingefügt: sie umschließen eine Anweisung, die sich durch `syntaxNodeId` auszeichnet. `Leave()` sorgt dafür, dass der Aufrufstapel `stack` wieder genauso aussieht, wie vor dem zugehörigen `Enter()`-Aufruf. Deshalb sucht `Leave()` nach dem obersten Laufzeitrepräsentanten `r` mit `r.SyntaxNodeId = syntaxNodeId` und entfernt alle darüber liegenden Laufzeitrepräsentanten und schließlich auch `r`. Es kann vorkommen, dass `r` nicht ganz oben auf dem Stapel liegt, wenn durch Kontrollflussanweisungen wie `return` oder `break` nicht alle `Leave()`-Anweisungen anderer Anweisungen ausgeführt wurden.
- **`EnterIteration(int syntaxNodeId):`** Der Aufruf `EnterIteration()` wurde zu Beginn des Schleifenrumpfs der Schleife eingefügt, deren `SyntaxNodeId` mit dem übergebenen Parameter `syntaxNodeId` übereinstimmt. Wie `Leave()` räumt `EnterIteration()` zunächst den Aufrufstapel auf: an oberster Stelle des Aufrufstapels liegt dann ein Laufzeitrepräsentant der Schleife. Die aufzuräumenden Laufzeitrepräsentanten können von vorherigen Iterationen oder durch Kontrollflussanweisungen verursacht werden. Dann wird ein Iterationsknoten (eine Instanz von `RuntimeNode` mit `SyntaxNodeId = RuntimeNode.IterationNodeId = -2` und `Parent = stack.Peek()`) erzeugt und auf den Stapel gelegt. Dieser Iterationsknoten wird benötigt um später iterationsübergreifende Abhängigkeiten erkennen zu können.
- **`LoadField(Object object, int fieldId):`** Der Aufruf `LoadField(object, fieldId)` wurde vor einem lesenden Zugriff auf ein Feld (`fieldId`) des Objekts `object` hinzugefügt. Beim Betreten von `LoadField()` wird der oberste Laufzeitrepräsentant `r = stack.Peek()` des Aufrufstapels für den Zugriff verantwortlich gemacht. Durch die Zugriffstabelle `fieldToAccessHistory` wird `LastWriter` und `LastReaders` verfügbar gemacht. Falls `LastWriter` nicht `null` ist, wird eine RAW-Laufzeitabhängigkeit (vom Typ `RuntimeDependenceEdge`) vom Laufzeitrepräsentanten `r` zu `LastWriter` erzeugt. Anschließend wird `LastReaders` um `r` ergänzt. Dann wird die Methode `Extract()` der `DependencyExtractor`-Instanz aufgerufen, um aus der Laufzeitabhängigkeit eine Abhängigkeit zwischen

Syntaxknoten abzuleiten und diese in der Datei `DynamicDependencies.xml` zu persistieren.

Notiz: Die Datenstruktur `RuntimeDependenceEdge` wurde geschaffen um die Protokollierung von der Nachverarbeitung zu entkoppeln. Es ist denkbar, Instanzen von `Runtime-DependeneEdge` in einen Puffer zu schreiben, anstatt direkt die Methode `Extract()` aufzurufen. Für die Nachverarbeitung müsste dazu zu Beginn der Protokollierung ein eigener Ausführungsfaden gestartet werden, der dem Puffer ständig Elemente entnimmt und die Methode `Extract()` für diese Elemente ausführt. Protokollierung und Nachverarbeitung würden dann durch das Erzeuger-Verbraucher-Muster parallel arbeiten.

5.3.3 Nachverarbeitung (DependencyExtractor)

`DependencyExtractor`: Laufzeitabhängigkeiten (vom Typ `RuntimeDependenceEdge`) besitzen drei Eigenschaften: `From`, `To` und `Kind`. `From` ist der Laufzeitrepräsentant, der von `To` `Kind`-abhängig (siehe 5.1.4) ist. Um eine Laufzeitabhängigkeit auf eine Abhängigkeit zwischen Syntaxknoten abzubilden genügt es nicht, das Offensichtliche zu tun: `From.SyntaxNodeId` als `Kind`-abhängig von `To.SyntaxNodeId` zu erklären. Aus diesem Grund wurden die Verzeigerung von Laufzeitrepräsentanten sowie die Erzeugung von Iterationsknoten eingeführt. Der Methode `Extract()` kommt die Aufgabe zu, über die Verzeigerung und die Iterationsknoten eine Laufzeitabhängigkeit in eine Abhängigkeit zwischen Syntaxknoten zu überführen und anschließend in die Datei `DynamicDependencies.xml` auszugeben.

`Extract(RuntimeDependencyEdge edge)`: Die Methode `Extract()` implementiert den bereits in Kapitel 4.7.4 (Nachverarbeitung) beschriebenen Nachverarbeitungsschritt. Dazu muss zunächst der erste gemeinsame Vorgänger `v` von `edge.From` und `edge.To` gefunden werden. Weil die Menge aller verzeigerten Instanzen von `RuntimeNode` einen umgekehrten Baum darstellt, können zwei Laufzeitrepräsentanten `currentFrom` und `currentTo` nur dann einen gemeinsamen Vorgänger haben, wenn sie die gleiche Entfernung zum Wurzelknoten `RootNode` besitzen. Diese Entfernung kann durch das Aufsteigen im Baum (über die Eigenschaft `Parent`) gemessen werden und ist in der privaten Methode `Depth()` implementiert. Weil sich `edge.From` und `edge.To` nicht notwendigerweise in der gleichen Entfernung vom Wurzelknoten befinden, wird von dem weiter entfernten Knoten solange aufgestiegen, bis die Entfernung beider Knoten übereinstimmt. Diese Knoten heißen dann `currentFrom` und `currentTo`. Anschließend kann der gemeinsame Vorgänger gefunden werden, indem von diesen Knoten Schritt für Schritt gleichzeitig aufgestiegen wird, bis `currentFrom.Parent` gleich `currentTo.Parent` ist. Dann sind zwei Fälle zu unterscheiden: (i) handelt es sich bei `currentFrom` und `currentTo` nicht um Iterationsknoten, so ist eine `edge.Kind`-Abhängigkeit vom Syntaxknoten `currentFrom.SyntaxNodeId` zu `currentTo.SyntaxNodeId` auszugeben. (ii) handelt es sich um Iterationsknoten, so ist eine iterationsübergreifende `edge.Kind`-Abhängigkeit auszugeben. Allerdings nicht zwischen den Syntaxknoten, der Laufzeitrepräsentanten `currentFrom` und `currentTo`, sondern zwischen denen der Laufzeitrepräsentanten, die von `currentFrom` und `currentTo` vor dem letzten Aufsteigen belegt wurden (`lastFrom` und `lastTo`).

Die Ausgabe einer `WAR`-Abhängigkeit vom Knoten `63` zum Knoten `53` erfolgt in der Notation aus Abbildung 35. Abhängigkeiten werden der Datei `DynamicDependencies.xml` textuell hinzugefügt.

```
1 <Edge from="63" to="53" kind="WAR"/>
```

Abbildung 35: Die Notation einer `WAR`-Abhängigkeit vom Knoten `63` zum Knoten `53`.

Notiz: Um Redundanz in der Datei `DynamicDependencies.xml` zu vermeiden, werden Abhängigkeiten zwischen Syntaxknoten in einem Zwischenspeicher gehalten und nur dann in die Datei geschrieben, wenn sie nicht bereits zuvor in die Datei geschrieben wurden.

5.4 Zusammenspiel von Instrumentierung, Protokollierung und Nachverarbeitung

In diesem Abschnitt geht es darum, die dynamische Abhängigkeitsanalyse anhand eines konkreten Beispiels vollständig zu beleuchten. In diesem Kapitel wird daher nicht der gesamte Parallelisierungsprozess abgedeckt. Um das Zusammenspiel von Instrumentierung, Protokollierung und Nachverarbeitung zu beleuchten, wird weder das motivierende Beispiel aus Kapitel 4.2, noch das alltagsnahe Beispiel aus Kapitel 4.7.5 herangezogen, weil sich an beiden Beispielen nicht so viele interessante Aspekte demonstrieren lassen. Stattdessen ist in Abbildung 36 ein abstraktes Quellcodebeispiel dargestellt, an dem sich mehrere interessante Aspekte verdeutlichen lassen, die in Abbildung 38 durch grüne Punkte markiert sind.

```
 1 class E {
 2  static readonly E e = new E();
 3  int field;
 4
 5  public static void Main() {
 6   int i = 0;
 7   while (i < 4) {
 8    A();
 9    if (i == 1) { break; }
10    B();
11    i++;
12   }
13   C();
14  }
```

```
15
16  static void A() {
17   int i = e.field;
18  }
19
20  static void B() {
21   e.field = 17;
22  }
23
24  static void C() {
25   e.field = 42;
26  }
27 }
28
```

Abbildung 36: Abstraktes Quellcodebeispiel, ohne höhere Bedeutung.

Den Methoden und Variablen des abstrakten Quellcodebeispiels aus Abbildung 36 kommt keine höhere Bedeutung zu. Es ist absichtlich abstrakt gehalten, um mit möglichst wenig Quellcode dennoch interessante Aspekte beleuchten zu können. Die lokale Variable `i` wird schon von der statischen Analyse berücksichtigt und spielt für die dynamische Analyse keine Rolle mehr. Außer der Klassenvariablen `e`, die nicht für Abhängigkeiten sorgt, weil sie nur einmal durch den statischen Konstruktor der Klasse `E` geschrieben und anschließend nur noch gelesen wird, gibt es nur die Instanzvariable `e.field`, die bei der dynamischen Abhängigkeitsanalyse für Abhängigkeiten sorgt. Die Methode `A()` greift lesend auf `e.field` zu, `B()` und `C()` schreibend. Bei der Ausführung der `Main()`-Methode wird zunächst eine erste Iteration der `while`-Schleife ausgeführt. Die zweite Iteration wird schon vorzeitig abgebrochen, weil die Bedingung `i==1` wahr wird und deshalb die Anweisung `break` ausgeführt wird.

In Abbildung 37 ist das abstrakte Quellcodebeispiel nach der Instrumentierung dargestellt. Durch Phase I hinzugefügte Abgrenzungsanweisungen sind blau hervorgehoben. Abgrenzungsanweisungen, die im Folgenden nicht zum Verständnis der Protokollierung beitragen, wurden aus Gründen der Übersichtlichkeit entfernt. Jeder Anweisung wurde durch Phase I eine ein-

deutige `SyntaxNodeId` zugeordnet: so besitzt die `while`-Schleife beispielsweise die ID `48`, was durch die umschließenden Anweisungen `Logger.Enter(48)` und `Logger.Leave(48)` deutlich wird. Der Aufruf `A()` besitzt die ID `53`, `B()` die ID `63` und `C()` die ID `70`. Insbesondere zwischen diesen Anweisungen werden später Abhängigkeiten ausgegeben, weil sie auf die gemeinsame Variable `e.field` zugreifen.

Zugriffsanweisungen, die in Phase II eigentlich nur dem IL-Code und nicht dem Quellcode hinzugefügt werden, sind grün markiert. Dem Feld `field` wird von der Klasse `AssemblyUtilities.UniqueIdProvider` eine eindeutige ID zugewiesen: die `fieldId 0`. Zugriffe auf das Feld `e.field` werden wie bereits in Kapitel 5.2 beschrieben instrumentiert, Zugriffe auf das statische Feld `e` nicht, weil `e` mit `readonly` markiert ist.

```
 1 class E {
 2  static readonly E e = new E();
 3  int field; /* fieldId = 0 */
 4
 5  public static void Main() {
 6   int i = 0;
 7   Logger.Enter(48);
 8   while (i < 4) {
 9    Logger.EnterIteration(48);
10    Logger.Enter(53);
11    A();
12    Logger.Leave(53);
13    Logger.Enter(57);
14    if (i == 1) { break; }
15    Logger.Leave(57);
16    Logger.Enter(63);
17    B();
18    Logger.Leave(63);
19    i++;
20   }
21   Logger.Leave(48);
22
```

```
23   Logger.Enter(70);
24   C();
25   Logger.Leave(70);
26   Logger.Save();
27  }
28
29  static void A() {
30   Logger.Enter(78);
31   Logger.LoadField(e, 0);
32   int i = e.field;
33   Logger.Leave(78);
34  }
35
36  static void B() {
37   Logger.StoreField(e, 0);
38   e.field = 17;
39  }
40  static void C() {
41   Logger.StoreField(e, 0);
42   e.field = 42;
43  }
44 }
```

Abbildung 37: Abstraktes Quellcodebeispiel nach der Instrumentierung. Blau: Abgrenzungsanweisungen aus Phase I, grün: Zugriffsanweisungen aus Phase II.

Die Protokollierung von Abhängigkeiten geschieht während der Ausführung des instrumentierten Beispiels und ist im zeitlichen Verlauf von links nach rechts in Abbildung 38 dargestellt. Verfolgt man den Kontrollfluss durch die `Main()`-Methode (also zunächst eine Iteration der `while`-Schleife und danach eine zweite Iteration die frühzeitig durch `break` abgebrochen wird), ergibt sich die dort zuunterst dargestellte Folge der Signalisierungsanweisungen.

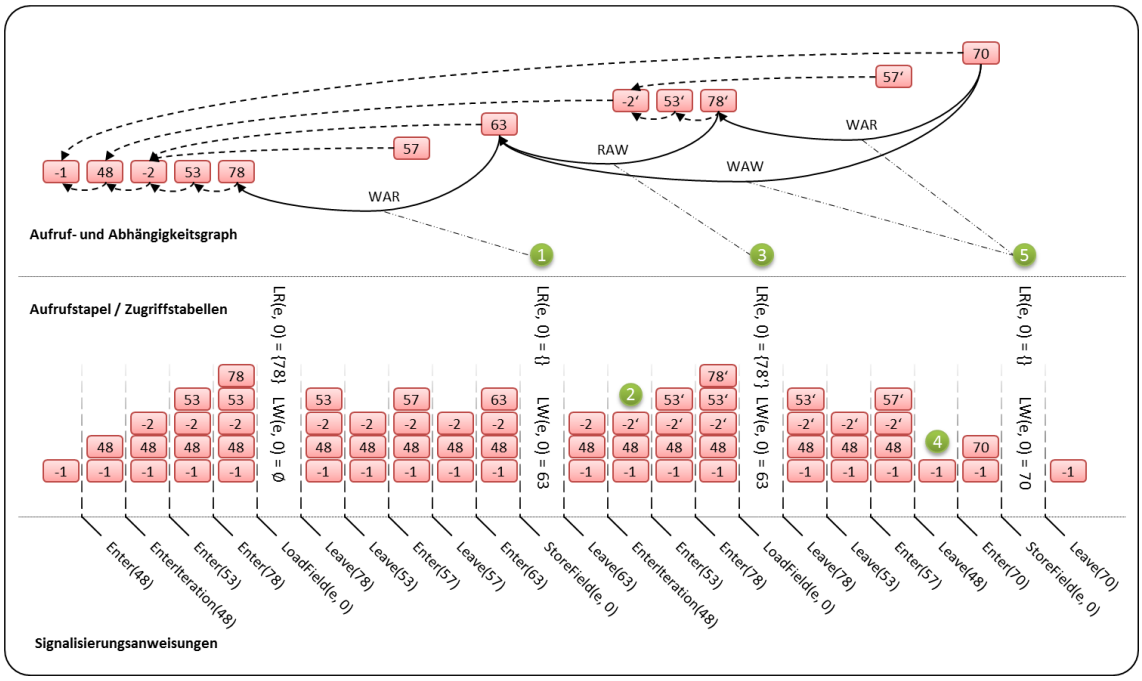

Abbildung 38: Zeitlicher Verlauf der Protokollierung. Unten: die Folge von Signalisierungsanweisungen, mitte: der Zustand des Aufrufstapels oder der Zugriffstabellen, oben: der Zustand des dynamischen Aufruf- und Abhängigkeitsgraphen.

Der zeitliche Verlauf der Protokollierung ist in Abbildung 38 dargestellt und entwickelt sich von links nach rechts. Unten ist die Folge der Signalisierungsanweisungen zu sehen, die sich während der Ausführung des instrumentierten Beispiels ergibt. In der Mitte ist je nach Art der Signalisierungsanweisung entweder der Zustand des Aufrufstapels (bei Abgrenzungsanweisungen) oder der Zugriffstabellen (bei Zugriffsanweisungen) dargestellt. Laufzeitrepräsentanten (Instanzen von `RuntimeNode`) sind rot dargestellt, die Nummer steht für deren `SyntaxNodeId` (`-1` ist die ID des eindeutigen Wurzelknoten `RootNode` und `-2` die ID für Iterationsknoten). Ein nachgestelltes Hochkomma (‘) dient zur Unterscheidung zwei verschiedener Laufzeitrepräsentanten mit gleicher `SyntaxNodeId`. Oben ist der dynamische Aufruf- und Abhängigkeitsgraph dargestellt: gestrichelte Pfeile stellen die Aufrufbeziehung (die Eigenschaft `Parent`) dar, durchgezogene Pfeile sind Laufzeitabhängigkeiten.

In Abbildung 38 ist zu sehen, wie durch `Enter()` Laufzeitrepräsentanten erzeugt und auf den Stapel gelegt werden. Mit der Erzeugung eines Laufzeitrepräsentanten geht die Verknüpfung mit einem bestehenden Laufzeitrepräsentanten im dynamischen Aufruf- und Abhängigkeitsgraph einher, die durch einen gestrichelten Pfeil dargestellt wird: ein neu erzeugter Laufzeitrepräsentant zeigt immer auf denjenigen, der zuvor auf dem Stapel lag (`Parent`). Man erkennt zudem, wie `Leave()` dazu führt, dass Laufzeitrepräsentanten vom Stapel entfernt werden. Es wird ersichtlich, dass durch `Load()`- oder `Store()`-Anweisungen die Zugriffstabellen LR und LW aktualisiert werden (im Beispiel für die Variable `e.field`, deren eindeutiger Bezeichner sich aus `e` und der `fieldId 0` zusammensetzt). Zudem fällt auf, dass dieser Vorgang in mehreren Fällen dazu führt, dass Laufzeitabhängigkeiten erzeugt werden, auf die im Folgenden eingegangen wird. Fünf interessante Zeitpunkte sind gesondert durch grüne Punkte markiert:

1. `StoreField(e, 0)`: Wegen des Schreibzugriffs und weil die Variable `e.field` zuvor schon durch den Laufzeitrepräsentanten `78` gelesen wurde (`LR(e, 0) = {78}`) wird eine WAR-Laufzeitabhängigkeit ausgegeben.
2. `EnterIteration(48)`: Weil sich über dem Laufzeitrepräsentanten noch ein Iterationsknoten (`-2`) befindet, wird dieser zunächst entfernt und erst dann ein neuer Iterationsknoten auf den Stapel gelegt. An dieser Stelle zeigt sich, dass `EnterIteration()` mehr tun muss, als nur einen neuen Iterationsknoten auf den Stapel zu legen.
3. `LoadField(e, 0)`: Wegen des Lesezugriffs und weil die Variable `e.field` zuvor schon durch den Laufzeitrepräsentanten `63` geschrieben wurde (`LW(e, 0) = {63}`) wird eine RAW-Laufzeitabhängigkeit ausgegeben. Zu diesem Zeitpunkt ist noch nicht klar, dass es sich um eine iterationsübergreifende Abhängigkeit handelt. Dies stellt sich erst während der Nachverarbeitung heraus.
4. `Leave(48)`: Weil sich über dem Laufzeitrepräsentanten `48` noch weitere Laufzeitrepräsentanten auf dem Stapel befinden, werden diese zunächst entfernt. Schuld daran ist die `break`-Anweisung, welche die Schleife frühzeitig abbricht und dazu führt, dass einige `Leave()`-Anweisungen übersprungen werden. An dieser Stelle zeigt sich also, dass Leave() mehr tun muss, als den obersten Laufzeitrepräsentanten vom Stapel zu nehmen.
5. `StoreField(e, 0)`: Wegen des Schreibzugriffs und `LW(e, 0) = 63` wird eine WAW-Abhängigkeit und wegen `LR(e, 0) = {78‘}` eine WAR-Abhängigkeit ausgegeben.

Die ausgegebenen Laufzeitabhängigkeiten führen während der Nachverarbeitung zu den in Abbildung 39 dargestellten Abhängigkeiten zwischen Syntaxknoten.

Abbildung 39: Durch die Nachverarbeitung werden aus Laufzeitabhängigkeiten Abhängigkeiten zwischen Syntaxknoten. Oben: Laufzeitabhängigkeiten, unten: Abhängigkeiten zwischen Syntaxknoten.

Der erste gemeinsame Vorgänger (gestrichelte Pfeile im dynamischen Aufruf- und Abhängigkeitsgraphen) der Laufzeitrepräsentanten `78` und `63` ist der Iterationsknoten `-2`. Auf dem Pfad von `78` und `63` zum gemeinsamen Vorgänger `-2` sind `53` und 63 die letzten besuchten Knoten. Die erste Laufzeitabhängigkeit ist also auf eine Abhängigkeit vom Syntaxknoten `63` zum Syntaxknoten `53` abzubilden. Bei der zweiten Abhängigkeit wird eine iterationsübergreifende (`LC`) Abhängigkeit festgestellt: Der erste gemeinsame Vorgänger von `63` und `78‘` ist ein Knoten einer Schleife (`48`), die letzten zwei besuchten Knoten auf dem Pfad zu `48` sind zwei verschiedene Iterationsknoten (`-2` und `-2‘`). Die verbleibenden zwei Abhängigkeiten werden analog behandelt.

Die ermittelten Abhängigkeiten zwischen Syntaxknoten werden in der Datei `Dynamic-Dependencies.xml` abgelegt, welche in Phase III benutzt wird, um dynamisch ermittelte Datenabhängigkeiten einzulesen. Der Inhalt dieser Datei ist für die in Abbildung 39 darge-

stellten Abhängigkeiten in Abbildung 40 abgebildet. Die Phase III, während der die Architekturerkennung durchgeführt wird, wird in diesem Beispiel nicht noch einmal behandelt, weil sie bereits im zugehörigen Kapitel 4.8 durch Beispiele beleuchtet wurde.

```
1 <DependenceEdges>
2  <Edge from="63" to="53" kind="WAR"/>
3  <Edge from="53" to="63" kind="LoopCarried, RAW"/>
4  <Edge from="70" to="48" kind="WAR"/>
5  <Edge from="70" to="48" kind="WAW"/>
6 </DependenceEdges>
```

Abbildung 40: Die resultierende Datei `DynamicDependencies.xml`, die zum Einlesen der dynamischen Datenabhängigkeiten in Phase III herangezogen wird.

6 EVALUIERUNG

In diesem Kapitel wird das in Kapitel 1 präsentierte Verfahren mit Hilfe der in Kapitel 1 vorgestellten Implementierung anhand von sechs Fallbeispielen evaluiert. Tabelle 2 gibt einen ersten Überblick der Fallbeispiele wieder: für jedes Projekt wird aufgezeigt, aus wie vielen Zeilen, Methoden und Klassen es sich zusammensetzt. Diese Zahlen wurden mit dem Werkzeug `SourceMonitor` [SM] ermittelt und geben die Größe der Fallbeispiele nur aus Sicht des Quellcodes wieder. Referenzierte Bibliotheken sind nicht aufgeführt, können die Komplexität eines Projekts jedoch stark erhören. `VideoProcessing` ist eines der Fallbeispiele, welches beispielsweise die Bibliothek `AForge` [AF] verwendet, die ihrerseits aus über 130.000 Zeilen Quellcode besteht. In Tabelle 2 ist zudem aufgeführt, wie groß die Anzahl der enthaltenen Sequenzen und Schleifen ist, die ein Entwickler überprüfen muss, wenn er das Projekt ohne die Hilfe des in dieser Arbeit vorgestellten Verfahrens parallelisieren will. Diese Zahlen wurden mit `Roslyn` [Ros] ermittelt, indem nach den entsprechenden Typen von Syntaxknoten gesucht wurde.

Projekt	Zeilen	Methoden	Klassen	Sequenzen	Schleifen
`MergeSort`	94	6	1	14	5
`RayTracer`	522	37	15	58	5
`DesktopSearch`	315	22	8	40	7
`CompGeo`	1.058	73	11	149	20
`VideoProcessing`	110	12	2	14	2
`PowerCollections`	24.481	1.252	79	3.354	287

Tabelle 2: Überblick der Fallbeispiele: Sequenzen und Schleifen sind die Stellen im Projekt, die von einem Entwickler inspiziert werden müssen, um das Projekt zu parallelisieren.

Die Fallbeispiele werden in den Kapiteln 6.2 bis 6.7 im Detail vorgestellt. Für die Fallbeispiele wird in den jeweiligen Unterkapiteln auf folgende Aspekte eingegangen:

- **Kategorisierung der Vorschläge:** Die erzeugten Parallelisierungsvorschläge werden begutachtet und in drei Kategorien eingeordnet: (i) korrekt und leistungssteigernd, (ii) korrekt aber nicht leistungssteigernd und (iii) nicht korrekt. Korrekt bedeutet, dass der Vorschlag nicht zum Fehlverhalten der Anwendung führt, leistungssteigernd bedeutet, dass die Umsetzung des Parallelisierungsvorschlags zu einem Laufzeitvorteil führt.
- **Leistungsmessungen:** Parallelisierungsvorschläge, die als korrekt und leistungssteigernd kategorisiert wurden, werden manuell parallelisiert um anschließend deren Laufzeit gemessen um daraus die erzielbare Beschleunigung abzuleiten. Auf die manuelle Parallelisierung muss zurückgegriffen werden, weil der Übersetzer [W13] zum Zeitpunkt der Evaluierung noch nicht zur Verfügung steht. Ein Prototyp der Bibliothek mit Implementierungen für die parallelen Architekturmuster (im Folgenden auch Laufzeitbibliothek genannt) wird bei der manuellen Parallelisierung jedoch schon eingesetzt.
- **Protokollierungsmehraufwand:** Es wird aufgezeigt, wie sich Laufzeit und Speicherverbrauch der instrumentierten Anwendung im Vergleich zur ursprünglichen Variante verhalten um ein Gefühl für diese Werte zu vermitteln.

Die Zusammenfassung und Diskussion der Ergebnisse folgt in Kapitel 6.9.

6.1 Durchführung und Versuchsumgebung

Weil die prototypische Implementierung momentan noch keine referenzierten *assemblies* instrumentiert (siehe Kapitel 5.2.1), können Abhängigkeiten übersehen werden, die zu inkorrekten Parallelisierungsvorschlägen führen, welche nicht erzeugt würden, wenn referenzierte *assemblies* instrumentiert wären. Dieser Effekt kann vermieden werden, indem der Quellcode referenzierter Klassen in die *solution* eingebunden wird. Des Weiteren wird die Instrumentierung von Zugriffen auf Arrays mit Referenztyp momentan nicht unterstützt. Anstatt der IL-Anweisungen `ldelem` und `stelem` (siehe 5.2) wird bei Arrays mit Referenztyp eine Kombination aus den IL-Anweisungen `ldelema` und `stobj` verwendet, vor denen momentan keine Instrumentierungsanweisungen eingefügt werden. Um zu verhindern, dass Abhängigkeiten über solche Zugriffe übersehen werden, können dem Quellcode die benötigten Zugriffsanweisungen per Hand hinzugefügt werden. Um die gesonderte Behandlung dieser Fälle überflüssig zu machen, muss die Implementierung der IL-Instrumentierung erweitert werden. Das Konzept der Arbeit bleibt davon unberührt.

Bei der Begutachtung der Parallelisierungsvorschläge wird versucht, Vorschläge zu finden, welche aufgrund der soeben beschriebenen fehlenden Abhängigkeiten zustande kommen, und diese als Pseudovorschläge zu identifizieren. Anschließend wird darauf eingegangen, wie die *solution* überarbeitet wurde, um die Ermittlung fehlender Abhängigkeiten zu ermöglichen. Schließlich wird aufgezeigt, wie sich diese Abhängigkeiten auf den Pseudovorschlag auswirken. Es sei noch einmal darauf hingewiesen, dass Pseudovorschläge und deren Behandlung lediglich ein Artefakt der prototypischen Implementierung sind und damit keinen Einfluss auf das Konzept dieser Arbeit besitzen.

Falls Pseudovorschläge identifiziert und daraufhin Veränderungen an einer *solution* vorgenommen wurden, wird bei der Messung des Mehraufwands der Protokollierung stets auf die abgeänderte Variante zurückgegriffen, weil diese Variante potentiell einen größeren Mehraufwand nach sich zieht.

In den Tabellen zu den Leistungsmessungen der folgenden Unterkapitel ist jeweils die Summe der Laufzeiten aus zehn Testläufen aufgetragen um mögliche Schwankungen zu kompensieren. Die Messungen wurden auf folgendem System durchgeführt:

- Prozessor: AMD FX 8120h, 8 Prozessorkerne, je 3,1 GHz Taktfrequenz
- Hauptplatine: Acer Predator G3120
- Hauptspeicher: 8 GB (4x2GB DDR3, PC3-10700, Dual-Channel)
- Festplatte: Samsung SSD 830
- Betriebssystem: Windows 7 x64

Wie in einem Mikrobenchmark gezeigt werden konnte [W13], kann mit diesem System trotz der acht Prozessorkerne nur eine Beschleunigung von etwa 4,5 erzielt werden, sobald Fließkommazahlen verwendet werden. Grund dafür ist die Prozessorarchitektur: je zwei Prozessorkerne teilen sich eine Fließkommaeinheit, auf die konkurrierend zugegriffen wird. Dieser Umstand mag eine Erklärung für die relativ gering ausfallenden Beschleunigungen von beispielsweile `RayTracer` liefern. Wie eine Untersuchung zeigt, beeinflusst die verwendete

Laufzeitbibliothek die erzielbaren Beschleunigungen negativ. Sie befindet sich aktuell in Entwicklung und liegt in prototypischer Form vor.

Aus den aufgeführten Gründen und deshalb, weil bei den Leistungsmessungen nie der gesamte Tuning-Parameterraum ausgeschöpft wurde, sind die gemessenen Beschleunigungen nur als untere Schranke der möglichen erzielbaren Beschleunigungen zu verstehen. Für `MergeSort` und `VideoProcessing` konnte diese These bestätigt werden, weil für diese Anwendungen bereits in [O13] beziehungsweise [D08] höhere Beschleunigungswerte berichtet wurden, wobei sich die dort eingesetzte Parallelität vollständig mit dem jeweiligen von dieser Arbeit ausgegebenen Parallelisierungsvorschlag deckt.

6.2 MergeSort

Sortieren durch Mischen basiert auf dem Teile-und-Herrsche-Prinzip (engl. *divide and conquer*). Das zu sortierende Array wird rekursiv in kleinere Teilarrays zerlegt und sortiert, sobald die Teilarrays klein genug sind. Anschließend werden die sortierten Teilarrays sukzessive zusammengeführt.

Als Eingabe erwartet `MergeSort` die Größe des zu sortierenden Arrays. Danach wird ein Array der gewünschten Größe angelegt, mit zufälligen Ganzzahlen initialisiert und schließlich sortiert.

`MergeSort` lässt sich parallelisieren, indem die Sortierung des linken und des rechten Teilarrays nebenläufig angestoßen wird [O13]. Diese Parallelisierungsmöglichkeit wird durch das Werkzeug aufgefunden und als Parallelisierungsvorschlag ausgegeben, wie im Folgenden Kapitel erläutert wird.

6.2.1 Kategorisierung der Vorschläge

Für `MergeSort` werden zwei Parallelisierungsvorschläge ausgegeben, wobei der erste als korrekt und leistungssteigernd und der zweite als Pseudovorschlag identifiziert wird. Nach der Abänderung der *solution* (Hinzufügen des Quellcodes der Klasse `Random`) fällt der Pseudovorschlag weg. Tabelle 3 gibt eine Übersicht der generierten Vorschläge wieder: Vorschläge gibt die Anzahl der ausgegebenen Vorschläge vor der Abänderung des Projekts wieder, Pseudovorschläge die Anzahl der identifizierten Pseudovorschläge, Wegfallende Pseudovorschläge die Anzahl der Pseudovorschläge, die durch die Abänderung des Projekts wegfallen. Die dann verbleibenden Vorschläge sind entweder korrekt und leistungssteigernd (K & L), korrekt aber nicht leistungssteigernd (K & ¬L) oder inkorrekt (¬K).

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
2	1	1	1	0	0

Tabelle 3: Übersicht der generierten Vorschläge für `MergeSort`.

Nach der Umsetzung des korrekten und leistungssteigernden Parallelisierungsvorschlags sind Beschleunigungen von bis zu 3,7 möglich (siehe Kapitel 6.2.2).

- Der erste Vorschlag bezieht sich auf die Methode `Sort()`, in der die rekursive Aufteilung in Teilarrays stattfindet (Abbildung 41): der Vorschlag besagt, dass `T2` und `T3` parallel zueinander abgearbeitet werden können. Dass `T1`, `T2` und `T3` abhängig von `T0` sind, wird schon während der statischen Analyse klar: die Abhängigkeit wird durch

die lokale Variable `mid` verursacht. Dass `T3` allerdings von `T1` und `T2` abhängig ist, wird erst durch die dynamische Analyse erfasst: `T3` greift auf Elemente des Arrays zu, die zuvor durch `T1` und `T2` geschrieben wurden. `T1` und `T2` greifen stets auf getrennte Bereiche des Arrays zu. Wie in Kapitel 6.2.2 gezeigt wird, lässt sich durch die Umsetzung dieses Vorschlags auch eine Beschleunigung erzielen. Der Parallelisierungsvorschlag ist damit korrekt und leistungssteigernd.

```
 1 private void Sort(int left, int right) {
 2  if (right > left) {
 3   #region (T0) ; (T1 || T2) ; (T3)
 4    T0: int mid = (right + left) / 2;
 5    T1: Sort(left, mid);
 6    T2: Sort(mid + 1, right);
 7    T3: Merge(left, mid + 1, right);
 8   #endregion
 9  }
10 }
```

Abbildung 41: Korrekter und leistungssteigernder Vorschlag für die Methode `Sort()`.

- Der zweite Vorschlag bezieht sich auf die Methode `Initialize()`, in der das zu sortierende Array mit der gewünschten Größe initialisiert und mit zufälligen Ganzzahlen gefüllt wird (Abbildung 42): der Vorschlag besagt, dass die `for`-Schleife durch ein Fließband mit einer impliziten Erzeuger-Stufe und einer replizierbaren Verbraucher-Stufe `S1` parallelisiert werden kann, weil keine iterationsübergreifende Abhängigkeit von `S1` zu sich selbst festgestellt wird. Eine solche Abhängigkeit existiert aber! Die Anweisung `r.Next(100)` ändert den internen Zustand des Pseudozufallsgenerators `r`. Diese Abhängigkeit wird während der dynamischen Abhängigkeitsanalyse nicht entdeckt, weil die Klasse `Random` der *assembly* `mscorlib.dll` nicht instrumentiert wird. Um die Erkennung dieser Abhängigkeit zu ermöglichen, wurde der *solution* der Quellcode[3] der Klasse `Random` hinzugefügt. Unter Berücksichtigung dieser Abhängigkeit, wird `S1` als nicht replizierbar markiert. Dies führt wiederum dazu, dass der Vorschlag nicht ausgegeben wird, weil keine Beschleunigung zu erwarten ist. Der Vorschlag aus Abbildung 42 ist damit ein Pseudovorschlag, der im Weiteren ausgeklammert wird.

```
 1 private int[] Initialize(int size) {
 2  Random r = new Random();
 3  int[] array = new int[size];
 4  #region S1+
 5   for (int i = 0; i < size; i++) {
 6    S1: array[i] = r.Next(100);
 7   }
 8  #endregion
 9  return array;
10 }
```

Abbildung 42: Pseudovorschlag für die Methode `Initialize()` der zustande kommt, weil die Klasse `Random` momentan nicht automatisch instrumentiert wird.

[3] https://github.com/mono/mono/blob/master/mcs/class/corlib/System/Random.cs

6.2.2 Leistungsmessungen

Für die Messung der erzielbaren Beschleunigung durch den Parallelisierungsvorschlag aus Abbildung 41 wurde die Methode `Sort()` manuell parallelisiert. Dabei wurde der Tuning-Parameter `maxRecursionDepth` eingeführt: er gibt an, bis zu welcher Rekursionstiefe eine parallele Ausführung von `T0` und `T1` angestoßen wird. Sollte die Rekursionstiefe den Wert von `maxRecursionDepth` überschreiten, wird die ursprüngliche sequentielle Variante ausgeführt. Die angestrebte automatische Codetransformation [W13] sorgt dafür, dass dieser Tuning-Parameter für jede Methode eingeführt wird, die aufgrund eines Parallelisierungsvorschlags transformiert wird.

`MergeSort` ist ein gutes Beispiel um zu zeigen, dass der Tuning-Parameter `maxRecursionDepth` (kurz `mRD`) entscheidenden Einfluss auf die erzielbare Beschleunigung hat. Bei den Laufzeitmessungen (siehe Tabelle 4) wurde der Parameter daher mit verschiedenen Werten belegt, um diesen Sachverhalt zu veranschaulichen. Der optimale Wert für `mRD` liegt für die durchgeführten Messungen in etwa bei 6 und führt zu Beschleunigungen von bis zu 3,7. Für wesentlich kleinere Eingaben könnte der Wert aber auch geringer ausfallen. Der Fall `mRD` = 32 mit einer 10-fachen Verlangsamung zeigt deutlich, dass die Steuerung der maximalen Rekursionstiefe nötig ist.

Der von dieser Arbeit vorgeschlagene Parallelisierungsansatz ist identisch mit dem in [O13] verfolgten Ansatz. Dort werden sogar Beschleunigungen von bis zu 4,8 berichtet. Dies zeigt, dass die Beschleunigungen dieser Arbeit lediglich eine untere Schranke darstellen, weil höhere Beschleunigungen erzielt werden können, ohne den Ort und die Art der Parallelisierung zu ändern.

Arraygröße	Sequentiell (s/10)	Tuning-Parameter `mRD`	Parallel (s/10)	Speedup
1.000.000	1,54	1	0,93	1,7
		2	0,65	2,4
		3	0,63	2,4
		4	0,61	2,5
		5	0,59	2,6
		6	0,59	2,6
		32	13,73	0,1
10.000.000	18,73	1	10,76	1,7
		2	7,29	2,6
		3	6,28	3,0
		4	6,32	3,0
		5	6,17	3,0
		6	5,99	3,1
		32	166,53	0,1
50.000.000	106,32	1	61,67	1,7
		2	40,23	2,6
		3	31,18	3,4
		4	31,64	3,4
		5	30,39	3,5

		6	28,76	3,7
		32	942,43	0,1

Tabelle 4: Laufzeitmessungen für die parallelisierte Variante von `MergeSort` für verschiedene Eingabegrößen und variierende Werte für den Tuning-Parameter `maxRecursionDepth` (`mRD`).

6.2.3 Protokollierungsmehraufwand

Während der Protokollierung der dynamischen Abhängigkeiten ist der instrumentierte `MergeSort`-Algorithmus bis zu 692-mal langsamer und verbraucht bis zu 65-mal mehr Speicher als die Originalimplementierung. Für größere Eingaben sind vor allem für den Speicherverbrauch noch größere Faktoren zu erwarten, weil für jedes Element des Eingabearrays der letzte Schreibende und die letzten Lesenden vorgehalten werden müssen.

	Original		Instrumentiert		Faktor	
Arraygröße	Zeit	Speicher	Zeit	Speicher	Zeit	Speicher
100.000	0,03 s	2,5 MB	13,18 s	91 MB	440	36
200.000	0,05 s	3 MB	30,44 s	184 MB	609	61
300.000	0,07 s	4 MB	48,41 s	259 MB	692	65

Tabelle 5: Gegenüberstellung der Laufzeit und des Speicherverbrauchs von instrumentierter und ursprünglicher `MergeSort`-Implementierung.

Die Menge der dynamisch ermittelten Datenabhängigkeiten ist für alle Eingabegrößen gleich.

6.3 RayTracer

Strahlenverfolgung (engl. *ray tracing*) ist eine Technik, die zur Darstellung von 3D-Szenen eingesetzt wird. Eine 3D-Szene besteht aus dreidimensionalen Objekten, die im Raum angeordnet sind. Ein Beobachter nimmt selbst eine Position und eine Blickrichtung im Raum ein. Zur Darstellung der Szene werden vom Beobachter Strahlen ausgesandt, und deren Schnittpunkte mit den Objekten im Raum berechnet. Dadurch kann ein Bild generiert werden, das die 3D-Szene aus Sicht des Beobachters darstellt.

Als Eingabe erwartet `RayTracer` die Größe des zu berechnenden Bildes. Danach werden Objekte erzeugt und im Raum angeordnet. Anschließend wird das Bild durch das Aussenden von Strahlen berechnet.

`RayTracer` lässt sich parallelisieren, indem das Aussenden mehrerer Strahlen nebenläufig angestoßen wird. Diese Parallelisierungsmöglichkeit wird vom Werkzeug aufgefunden und als Parallelisierungsvorschlag ausgegeben. Zudem findet das Werkzeug weitere Vorschläge, die allerdings nicht zu einer Beschleunigung führen, wie im Folgenden Kapitel erläutert wird.

6.3.1 Kategorisierung der Vorschläge

Für `RayTracer` werden sieben Parallelisierungsvorschläge ausgegeben, wobei ein Vorschlag als korrekt und leistungssteigernd identifiziert wird. Fünf Vorschläge sind korrekt, führen aber nicht zu einer Beschleunigung. Der verbleibende Vorschlag stellt sich als Pseudovorschlag heraus. Nach der Abänderung der *solution* (Hinzufügen von fehlenden Zugriffsanweisungen) wird der Pseudovorschlag zu einem Vorschlag, der nicht zu einer Beschleunigung führt. Tabelle 6 gibt eine Übersicht der generierten Vorschläge wieder. Die Spalten sind wie in Kapitel 6.2.1, Tabelle 3 gewählt.

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
7	1	0	1	6	0

Tabelle 6: Übersicht der generierten Vorschläge für `RayTracer`.

Die Umsetzung des korrekten und leistungssteigernden Vorschlags führt zu Beschleunigungen von bis zu 2,4 (siehe Kapitel 6.3.2).

- Zwei Vorschläge beziehen sich auf die Methode `Render()` der Klasse `RayTracer` (Abbildung 43). Die Methode enthält eine doppelte `for`-Schleife: Eine Iteration der inneren Schleife sendet einen Strahl aus um die Farbe des Bildpunktes (`x`, `y`) zu berechnen (`S2`) und setzt diese Farbe dann im Zielbild (`S3`). Eine Iteration der äußeren Schleife führt diesen Vorgang für eine ganze Zeile `y` aus. Die innere Schleife kann durch ein Fließband mit einer impliziten Erzeuger-Stufe, einer Verbraucher-Erzeuger-Stufe `S2` und einer Verbraucher-Stufe `S3` parallelisiert werden, wobei sowohl `S2` als auch `S3` replizierbar sind, weil es keine iterationsübergreifenden Abhängigkeiten gibt. Die äußere Schleife kann ebenfalls durch ein (davon unabhängiges) Fließband parallelisiert werden. Auch die Stufe `S1` ist aufgrund fehlender iterationsübergreifender Abhängigkeiten replizierbar. Wie in Kapitel 6.3.2 gezeigt wird, lässt sich durch die Parallelisierung der äußeren Schleife eine Beschleunigung erzielen. Die Parallelität der inneren Schleife hingegen ist zu feinkörnig und führt zu Laufzeiteinbußen. Ein Auto-Tuner würde diesen Vorschlag daher aussondern und dafür sorgen, dass stattdessen die ursprüngliche sequentielle Version der inneren Schleife zur Ausführung kommt.

```
 1 internal void Render(Scene scene) {
 2  #region S1+
 3   for (int y = 0; y < screenHeight; y++) {
 4    S1:
 5     #region S2+ => S3+
 6      for (int x = 0; x < screenWidth; x++) {
 7       S2: Color color = TraceRay(new Ray() { … }, scene, 0);
 8       S3: setPixel(x, y, color.ToDrawingColor());
 9      }
10     #endregion
11   }
12  #endregion
13 }
```

Abbildung 43: Korrekter und leistungssteigernder Parallelisierungsvorschlag für die äußere Schleife der Methode `Render()`.

- Zwei weitere Vorschläge beziehen sich auf die Methode `GetNaturalColor()` der Klasse `RayTracer` (Abbildung 44). Die `foreach`-Schleife kann durch ein mehrstufiges Fließband parallelisiert werden, die bis auf `S5` als replizierbar markiert sind. Dass die Stufe `S5` nicht replizierbar ist, wird schon durch die Abhängigkeitsanalyse lokaler Variablen klar: `ret` sorgt für eine iterationsübergreifende Abhängigkeit. Der nächste Vorschlag gibt an, dass die Anweisungen des `if`-Blocks teilweise parallel ausgeführt werden können: `T0` mit `T2` und `T1` mit `T3`. Um aufzuzeigen, dass in diesem Beispiel mögliche Parallelität verloren geht, weil TADL momentan nur parallele und sequentielle Sektionen zulässt, wurde der Taskgraph-Ausdruck als Kommentar mit angegeben: Laut Taskgraph-Ausdruck ist beispielsweise auch die parallele Ausführung von `T1` und `T2` möglich. Beide Vorschläge wurden überprüft und haben sich als korrekt

herausgestellt. Allerdings sind beide Vorschläge nicht leistungssteigernd, da die Methode `GetNaturalColor()` selbst nur wenige Mikrosekunden Laufzeit verbraucht.

```
 1 private Color GetNaturalColor(…) {
 2  Color ret = Color.Make(0, 0, 0);
 3  #region S1+ => S2+ => S3+ => S4+ => S5
 4   foreach (Light light in scene.Lights) {
 5    S1: Vector ldis = Vector.Minus(light.Pos, pos);
 6    S2: Vector livec = Vector.Norm(ldis);
 7    S3: double neatIsect = TestRay(new Ray() { … }, scene);
 8    S4: bool isInShadow = !((neatIsect > Vector.Mag(ldis)) || (…));
 9    S5: if (!isInShadow) {
10     /* T0(), T1(T0), T2(), T3(T2), T4(T1, T3) */
11     #region (T0 || T2) ; (T1 || T3) ; (T4)
12      T0: double illum = Vector.Dot(livec, norm);
13      T1: Color lcolor = illum > 0 ? Color.Times(…) : …;
14      T2: double specular = Vector.Dot(livec, Vector.Norm(rd));
15      T3: Color scolor = specular > 0 ? Color.Times(…) : …;
16      T4: ret = Color.Plus(ret, Color.Plus(Color.Times(…), …);
17     #endregion
18    }
19   }
20  #endregion
21  return ret;
22 }
```

Abbildung 44: Korrekte, aber nicht leistungssteigernde Parallelisierungsvorschläge für die Methode `GetNaturalColor()`.

- Drei Vorschläge beziehen sich auf die Methode `RayTracerForm_Load()` der Klasse `RayTracerForm` (Abbildung 45). Die Parallelisierungsvorschläge für die `for`-Schleifen sind korrekt, führen aber nicht zu einer Leistungssteigerung. Der Taskgraph-Vorschlag ist ein Pseudovorschlag: `T3` liest Elemente von `bitmapArray`, die durch die Strahlenverfolgung von `T2` geschrieben werden. Diese Abhängigkeit sollte von der dynamischen Abhängigkeitsanalyse erkannt werden, wird aber übersehen, weil Zugriffe auf Arrays mit Referenztyp momentan noch nicht durch die Implementierung abgedeckt werden. Für die Evaluierung genügt es, die Instrumentierungsanweisungen per Hand im Quellcode zu verankern: Vor dem Zugriff in Zeile 6 muss eine `Store()`-Zugriffsanweisung[4] und in Zeile 14 eine `Load()`-Zugriffsanweisung[5] eingefügt werden. Dann wird bei der dynamischen Abhängigkeitsanalyse erkannt, dass `T3` von `T2` abhängig ist und die Architekturbeschreibung ändert sich zu `(T0 || T1 || T4 || T5) ; (T2) ; (T3)`. Dieser Vorschlag wiederum führt nicht zu einer Beschleunigung und wird deshalb nicht weiterverfolgt.

[4] Logger.StoreArrayElement(bitmapArray, (y * imageHight) + x);
[5] Logger.LoadArrayElement(bitmapArray, (y * imageHight) + x);

```
 1 private void RayTracerForm_Load(object sender, EventArgs e) {
 2  #region (T0 || T1 || T3 || T4 || T5) ; (T2)
 3   T0: this.Show();
 4   T1: RayTracer rayTracer = new RayTracer(imageWidth, imageHight,
 5      (int x, int y, System.Drawing.Color color) =>
 6      { bitmapArray[(y * imageHight) + x] = color; });
 7   T2: rayTracer.Render(rayTracer.DefaultScene);
 8   T3:
 9    #region S1+
10    for (int x = 0; x < imageWidth; x++) {
11     S1:
12      #region S2+
13       for (int y = 0; y < imageWidth; y++) {
14        S2: bitmap.SetPixel(x, y, bitmapArray[(y * imageHight) + x]);
15       }
16      #endregion
17     }
18    #endregion
19   T4: pictureBox.Refresh();
20   T5: pictureBox.Invalidate();
21  #endregion
22 }
```

Abbildung 45: Pseudovorschlag für die Methode `RayTracerForm_Load()` der zustande kommt, weil Zugriffe auf Arrays mit Referenztyp momentan nicht automatisch instrumentiert werden.

6.3.2 Leistungsmessungen

Um die erzielbare Beschleunigung zu messen, wurde die Methode `Render()` in zwei Versionen nach den Vorschlägen aus Abbildung 43 manuell parallelisiert: in der ersten Version (V1) wurde nur der erste Vorschlag umgesetzt und damit die äußere `for`-Schleife durch ein Fließband mit einer replizierbaren Stufe `S1` parallelisiert, in der zweiten Version (V2) wurden beide Vorschläge umgesetzt: die innere `for`-Schleife wurde also zusätzlich durch ein Fließband mit einer kombinierten replizierbaren Stufe `S23` parallelisiert. Messungen der Version V1 finden sich in Tabelle 7, in Tabelle 8 sind die Messungen für V2 dargestellt. Es zeigt sich, dass die Parallelisierung der inneren Schleife nicht lohnenswert ist, weil V2 stets mehr Laufzeit benötigt als V1: Mit V1 können Beschleunigungen von bis zu 2,4 erzielt werden, die Werte von V2 bleiben mit bis zu 1,6 weit hinter den Erwartungen zurück. Bei den Messungen wurde für jede replizierbare Stufe der Tuning-Parameter `numThreads` (Anzahl der Stufenreplikate) variiert. Der optimale Wert liegt wie zu erwarten bei etwa 8.

Obwohl man bei `RayTracer` eine nahezu lineare Beschleunigung von etwa 8 erwartet, wird auf dem Achtkernsystem nur eine Beschleunigung von 2,4 erzielt, wohingegen auf einem weiteren Testsystem (Intel Core2Quad Q6600, 4 Prozessorkerne, je 2,4GHz) eine Beschleunigung von 2,9 möglich war (bei 1000x1000 und `nT(S1)` = 4). Dieser Umstand ist zum einen auf das Achtkernsystem zurückzuführen, welches für Fließkommaberechnungen nur 4 unabhängige Fließkommaeinheiten bereitstellt [W13]. Ein weiterer Grund mag die noch unausgereifte Implementierung der Laufzeitbibliothek sein: (i) Anstatt die Iterationen der `for`-Schleife durch eine Gebietszerlegung in grobgranulare Pakete mit mehreren Durchläufen zu zerlegen und auf die Instanzen der replizierbaren Stufe S1 zu verteilen, wird momentan für jede Iteration ein Arbeitspaket generiert. Das sorgt für viel Mehraufwand, der zu einer geringeren Beschleunigung führt als erwartet. Um dies zu vermeiden müsste die Laufzeitbibliothek um einen Tuning-Parameter erweitert werden, der das Einstellen der Paketgröße ermöglicht. (ii) Zudem teilen

sich mehrere Instanzen einer replizierbaren Stufe momentan den gleichen Eingabepuffer. Weil das Einreihen und Entnehmen von Elementen des Puffers durch eine Sperre geschützt ist, kommt es dort zu einem Wettbewerb um diese Sperre, was wiederum die erzielbare Beschleunigung nach unten drückt. Die Laufzeitbibliothek ist zur Vermeidung dessen so zu erweitern, dass die Anzahl von Puffern einer replizierbaren Stufe einstellbar ist.

Der Tuning-Parameter `maxRecursionDepth` wird auch in dieser Methode eingeführt, hat jedoch keine Auswirkung auf die Laufzeit, weil die Methode `Render()` nicht rekursiv aufgerufen wird.

Bildgröße	Sequentiell (s/10)	Tuning-Parameter `nT(S1)`	Parallel (s/10)	Speedup
600x600	27,64	2	17,20	1,6
		4	12,36	2,2
		8	12,82	2,2
		16	15,19	1,8
800x800	49,92	2	29,07	1,7
		4	20,40	2,4
		8	20,89	2,4
		16	24,83	2,0
1.000x1.000	78,42	2	46,28	1,7
		4	32,13	2,4
		8	32,68	2,4
		16	39,10	2,0

Tabelle 7: Laufzeitmessungen für die parallelisierte Variante V1 von `RayTracer` für verschiedene Eingabegrößen und variierende Werte für den Tuning-Parameter `numThreads` der replizierbaren Stufe `S1` (`nT(S1)`).

Bildgröße	Sequentiell (s/10)	Tuning-Parameter `nT(S1), nT(S23)`	Parallel (s/10)	Speedup
600x600	27,64	2, 1	23,22	1,2
		2, 2	17,54	1,6
		2, 4	18,08	1,5
		4, 1	17,87	1,5
		4, 2	18,35	1,5
		4, 4	23,08	1,2
800x800	49,92	2, 1	41,62	1,2
		2, 2	31,44	1,6
		2, 4	31,32	1,6
		4, 1	31,24	1,6
		4, 2	31,28	1,6
		4, 4	38,58	1,3

Tabelle 8: Laufzeitmessungen für die parallelisierte Variante V2 von `RayTracer` für verschiedene Eingabegrößen und variierende Werte für den Tuning-Parameter `numThreads` der replizierbaren Stufe `S1` (`nT(S1)`) und der kombinierten replizierbaren Stufe `S23` (`nT(S23)`).

6.3.3 Protokollierungsmehraufwand

Die Verlangsamung durch die Instrumentierung fällt bei `RayTracer` geringer aus, als bei `MergeSort`: die instrumentierte Anwendung ist bis zu 29-mal langsamer und verbraucht bis zu 80-mal mehr Speicher. Der beachtliche Unterschied zwischen einem Verlangsamungsfaktor von 692 bei `MergeSort` und lediglich 29 bei `RayTracer` lässt sich vor Allem durch die geringe Verschachtelungstiefe von `RayTracer` erklären: Die Hauptarbeit findet hier in einer doppelten `for`-Schleife statt, während die Hautarbeit bei `MergeSort` durch rekursive Aufrufe zustande kommt, wodurch sehr viele `Enter()`- und `Leave()`-Anweisungen ausgeführt und viele Instanzen von `RuntimeNode` erzeugt werden.

	Original		Instrumentiert		Faktor	
Bildgröße	Zeit	Speicher	Zeit	Speicher	Zeit	Speicher
200x200	0,49 s	7 MB	8,16 s	137 MB	17	20
400x400	1,73 s	9,5 MB	32,38 s	465 MB	19	49
600x600	3,84 s	13 MB	110,20 s	1043 MB	29	80

Tabelle 9: Gegenüberstellung der Laufzeit und des Speicherverbrauchs von instrumentierter und ursprünglicher `RayTracer`-Implementierung.

Wie schon bei `MergeSort` ist die Menge der dynamisch ermittelten Datenabhängigkeiten für alle Eingabegrößen gleich.

6.4 DesktopSearch

Die Desktopsuche [TM10] ist eine Anwendung zum Durchsuchen von Textdateien nach Schlüsselworten. Um Suchanfragen schnell beantworten zu können, wird ein invertierter Index aufgebaut, der einem Schlüsselwort alle Textdateien zuordnet, welche das Schlüsselwort enthalten. Des Weiteren wird hinterlegt, an welcher Position sich das Wort innerhalb der Datei befindet.

Als Eingabe erwartet `DesktopSearch` ein Verzeichnis, welches rekursiv nach Textdateien durchsucht wird. Daraufhin werden die Textdateien geöffnet, und die enthaltenen Wörter einem invertierten Index hinzugefügt. Anschließend sind Suchanfragen auf der erzeugten Indexdatenstruktur möglich. Für die Evaluierung wird jedoch nur die Indexgenerierung betrachtet.

`DesktopSearch` lässt sich durch ein dreistufiges Fließband parallelisierten: die erste Stufe ermittelt Worte und zugehörige Positionen einer Textdatei, die zweite Stufe pflegt diese in eine Indexstruktur ein und die dritte Stufe vereint den Index mit dem bereits vorhandenen Index vorheriger Textdateien [S10]. Die ersten zwei Stufen können zudem repliziert werden und dadurch mehrere Textdateien nebenläufig abarbeiten. Diese Parallelisierungsmöglichkeit wird vom Werkzeug aufgefunden und als Parallelisierungsvorschlag ausgegeben. Das Werkzeug findet einen weiteren Vorschlag, der allerdings nicht zu einer Beschleunigung führt.

6.4.1 Kategorisierung der Parallelisierungsvorschläge

Für `DesktopSearch` werden vier Parallelisierungsvorschläge ausgegeben, wobei jeder Vorschlag zunächst als Pseudovorschlag identifiziert wird. Nach der Abänderung der *solution* (Hinzufügen des Quellcodes der Klassen `List`, `HashSet`, `Dictionary` und `StringBuilder`) fallen zwei der Pseudovorschläge weg. Die verbleibenden zwei werden unter der Berücksichtigung der neu erfassten Abhängigkeiten angepasst. Beide Vorschläge sind anschließend

korrekt, einer davon führt zudem zu einer Beschleunigung. Tabelle 10Tabelle 6 gibt eine Übersicht der generierten Vorschläge wieder. Die Spalten sind wie in Kapitel 6.2.1, Tabelle 3 gewählt.

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
4	4	2	1	1	0

Tabelle 10: Übersicht der generierten Vorschläge für `DesktopSearch`.

Mit dem korrekten und leistungssteigernden Vorschlag sind Beschleunigungen von bis zu 2,0 zu erreichen.

- Zwei Vorschläge beziehen sich auf die Methode `GenerateIndex()` der Klasse `SequentialIndexing` (Abbildung 46), wobei sich beide Vorschläge zunächst als Pseudovorschläge herausstellen: Zum einen ist `T1` von `T0` abhängig, weil `T1` von der Liste `filenames` liest, die von `T0` geschrieben wird. Zum anderen besitzt die Stufe `S7` (Zeile 33) aufgrund der Zusammenführung der Indizes eine iterationsübergreifende Abhängigkeit zu sich selbst. Nachdem für die Klassen `List`, `HashSet` und `Dictionary` Quellcode[6] bereitgestellt wurde, fällt der erste Vorschlag weg, der zweite wird korrigiert und die Stufe `S7` als nicht replizierbar markiert: `S1+ => S2+ => S3+ => S4+ => S5+ => S6+ => S7 => S8+`. Dieser Vorschlag ist korrekt und führt zudem zu einer Beschleunigung.

[6] Der Quellcode stammt von http://powercollections.codeplex.com/. List, HashSet und Dictionary wurden durch die semantisch äquivalenten Implementierungen BigList, Set und OrderedDictionary ersetzt.

```
 1 public override bool GenerateIndex(string directory) {
 2  …
 3  #region T0 || T1
 4   T0: ScanForFiles(filenames, directory);
 5   T1:
 6    #region S1+ => S2+ => S3+ => S4+ => S5+ => S6+ => S7+ => S8+
 7     for (int i = 0; i < filenames.Count; i++) {
 8      S1: var fs = File.OpenRead(filenames[i]);
 9      S2: var bytesToRead = (int)fs.Length;
10      S3: var bytesOfFile = bytesToRead;
11      S4: var terms = new HashSet<TermPositionTuple>();
12      S5:
13       while (bytesToRead > 0) {
14        byte[] buffer = new byte[BUFFER_SIZE];
15        var bytesRead = fs.Read(buffer, 0, BUFFER_SIZE);
16        if (bytesRead < 1) { break; }
17        int c = 0;
18        var term = new char[MAX_TERM_LENGTH];
19        for (int b = 0; b < bytesRead; b++) {
20         var currentChar = (char)buffer[b];
21         if (IS_LETTER[currentChar] && c < MAX_TERM_LENGTH) {
22          term[c++] = currentChar;
23         } else {
24          if (c >= 3) {
25           terms.Add(new TermPositionTuple(new String(term, 0, c), …));
26           c = 0;
27          }
28         }
29        }
30        bytesToRead -= bytesRead;
31       }
32      S6: var index = Index.Create(terms, filenames[i]);
33      S7: multiIndex = multiIndex.Merge(index);
34      S8: fs.Close();
35     }
36   #endregion
37  #endregion
38  …
39 }
```

Abbildung 46: Pseudovorschläge für die Methode `GenerateIndex()`, die durch das Hinzufügen des Quellcodes von `List`, `HashSet` und `Dictionary` korrigiert werden.

- Ein Pseudovorschlag bezieht sich auf die Methode `Create()` der Klasse `Index`. Hier wird zunächst ein fünfstufiges Fließband mit fünf replizierbaren Stufen vorgeschlagen, was jedoch nicht korrekt ist. Nach dem Hinzufügen des Quellcodes der Klassen `List`, `HashSet` und `Dictionary` werden übersehene Abhängigkeiten berücksichtigt, was dazu führt, dass die Stufen `S2`, `S4` und `S5` als nicht replizierbar markiert werden. Dieser Vorschlag ist korrekt, führt aber nicht zu einer Leistungssteigerung.
- Ein Pseudovorschlag, der nach der Abänderung der *solution* wegfällt, bezieht sich auf die Methode `ToString()` der Klasse `SequentialIndexing`. Es wird vorgeschlagen, dass aufeinanderfolge Anweisungen mit Aufrufen zu `StringBuilder.Append()` parallel ausgeführt werden. Dass diese Anweisungen aber abhängig voneinander sind wird übersehen, weil die Klasse `StringBuilder` nicht instrumentiert wird. Dem kann wie bereits zuvor durch das Bereitstellen des Quellcodes von `StringBuilder` vorgebäugt werden.

6.4.2 Leistungsmessungen

Für die Leistungsmessungen wurde die Methode `GenerateIndex()` durch ein Fließband mit drei Stufen `S15`, `S6` und `S78` parallelisiert. Die Stufe `S16` fasst dabei die Stufen `S1` bis `S5` aus Abbildung 46 zusammen, `S78` die Stufen `S7` und `S8`. Entsprechend sind `S15` und `S6` als replizierbar und `S78` nicht als replizierbar markiert (weil `S7` nicht replizierbar ist). Durch das Zusammenlegen der Stufen wurde der Tuning-Parameter `stageFusion` für jedes Paar aus Stufen explizit festgelegt und kann während der Messungen nicht variiert werden. Diese Variation lässt die Laufzeitbibliothek momentan noch nicht zu. Variiert wird der Tuning-Parameter `numThreads` der replizierbaren Stufen `S15` und `S6` (`nT(S15)`, `nT(S6)`). Eine optimale Konfiguration der Parameter liegt in etwa bei `nT(S15)` = `nT(S6)` = 8. Sie führt zu einer Beschleunigung von bis zu 2,0. [TM10] berichten von Beschleunigungen von bis zu 4,7 auf einem Vierkernsystem, erfahren jedoch bei einem Achtkernsystem auch nur eine Beschleunigung von 2,1 ohne auf die Gründe einzugehen. Eine „kostenlose“ Beschleunigung von 2,0, wie sie durch dieses Verfahren möglich wird, ist daher durchaus akzeptabel.

Die Größen der Puffer zwischen den Fließbandstufen wurden bei den Messungen zwar auch variiert, hatten jedoch einen Einfluss auf die Laufzeiten. Auch der Tuning-Parameter `maxRecursionDepth` ist irrelevant, weil `GenerateIndex()` nicht rekursiv aufgerufen wird.

Ordner	Sequentiell (s/10)	Tuning-Parameter `nT(S15)`, `nT(S6)`	Parallel (s/10)	Speedup
TxtFiles32L	64,29	1, 1	54,26	1,2
		1, 2	54,34	1,2
		1, 4	57,46	1,2
		2, 1	55,18	1,2
		2, 2	47,27	1,4
		2, 4	47,39	1,4
		4, 1	54,34	1,2
		4, 2	47,27	1,4
		4, 4	43,26	1,5
		6, 6	42,92	1,5
		8, 8	42,60	1,5
TxtFiles64L	149,37	1, 1	118,84	1,3
		1, 2	118,95	1,3
		1, 4	121,42	1,2
		2, 1	123,77	1,2
		2, 2	101,76	1,5
		2, 4	101,35	1,5
		4, 1	122,84	1,2
		4, 2	99,25	1,5
		4, 4	91,49	1,6
		6, 6	90,00	1,7
		8, 8	89,56	1,7

TxtFiles798M	290	5, 5	146,24	2,0
		6, 6	150,2	1,9
		8, 8	151,92	1,9

Tabelle 11: Laufzeitmessungen für die parallelisierte Variante von `DesktopSearch` für verschiedene Eingabegrößen und variierende Werte für die Tuning-Parameter `numThreads` der replizierbaren Stufe `S15` und `S6` (`nT(S15)`, `nT(S6)`).

6.4.3 Protokollierungsmehraufwand

Die bis zu 363-fache Verlangsamung bei `DesktopSeach` liegt zwischen der von `MergeSort` und `Raytracer`. Im Gegensatz zu den zwei anderen Fallbeispielen nimmt der zusätzliche bis zu 31-fache Speicherverbrauch nicht so extrem mit der wachsenden Eingabegröße zu.

	Original		Instrumentiert		Faktor	
Ordner	Zeit	Speicher	Zeit	Speicher	Zeit	Speicher
TxtFiles8S	0,07 s	15 MB	16,55 s	232 MB	236	15
TxtFiles16S	0,17 s	16 MB	30,56 s	482 MB	180	30
TxtFiles32S	0,30 s	31 MB	108,96 s	969 MB	363	31

Tabelle 12: Gegenüberstellung der Laufzeit und des Speicherverbrauchs von instrumentierter und ursprünglicher `DesktopSearch`-Implementierung.

Auch in diesem Fallbeispiel ist die Menge der dynamisch ermittelten Abhängigkeiten unabhängig von der Eingabe.

6.5 CompGeo

`CompGeo` [CG] ist eine Bibliothek, die Algorithmen und Datenstrukturen zur Lösung geometrischer Problemstellungen bereitstellt. Einer dieser Algorithmen dient zur Berechnung der konvexen Hülle einer Punktmenge. Im Zweidimensionalen zeichnet sich die konvexe Hülle dadurch aus, dass alle Punkte der Punktmenge von ihr umschlossen werden und keine Verbindungslinie zweier Punkte außerhalb der Grenzen der konvexen Hülle liegt.

Weil `CompGeo` eine Bibliothek und damit nicht direkt ausführbar ist, wurde sie um eine `Main()`-Methode ergänzt. Als Eingabe erwartet die `Main()`-Methode von `CompGeo` die Größe der anzulegenden Punktmenge. Danach wird eine Liste der gewünschten Größe angelegt, mit zufällig im Raum verteilten Punkten gefüllt und schließlich die konvexe Hülle der Punktmenge berechnet. Dazu wird die Punktmenge zunächst nach y-Koordinaten sortiert. Daraufhin wird die obere Hälfte der konvexen Hülle berechnet und dann die untere.

Klassen der Bibliothek, die aufgrund der Wahl der `Main()`-Methode nicht zur Ausführung kommen, werden für die Evaluierung ausgeklammert.

6.5.1 Kategorisierung der Parallelisierungsvorschläge

Für `CompGeo` werden fünf Parallelisierungsvorschläge ausgegeben, wobei alle zunächst als Pseudovorschläge identifiziert werden. Nach der Abänderung der *solution* (Hinzufügen des Quellcodes der Klassen `Random` und `List` wie bereits bei `MergeSort` bzw. `DesktopSearch`) fallen vier Pseudovorschläge weg, der verbleibende Vorschlag ist korrekt, und führt zu einer Beschleunigung von bis zu 1,2. Tabelle 13 gibt eine Übersicht der generierten Vorschläge wieder. Die Spalten sind wie in Kapitel 6.2.1, Tabelle 3 gewählt.

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
5	5	4	1	0	0

Tabelle 13: Übersicht der generierten Vorschläge für `CompGeo`.

Auf die Pseudovorschläge wird im Folgenden nicht mehr eingegangen. Stattdessen wird nur der korrekte und leistungssteigernde Vorschlag nach der Abänderung der *solution* diskutiert.

- Der korrekte und leistungssteigernde Vorschlag bezieht sich auf die Methode `GetConvexHull2D()` der Klasse `ConvexHull`. Es wird ein Taskgraph in Form paralleler und sequentieller Sektionen vorgeschlagen, der zunächst jedoch nicht zu einer Beschleunigung führt. Parallelität lohnt sich nur für die Sortierung der Punktmenge (`T1`) und für die zwei `for`-Schleifen, also die Tasks `T5` und `T7`, da die restlichen Tasks schlichtweg zu wenig Laufzeit verbrauchen.

  Der Architekturbeschreibung aus Zeile 4 ist zu entnehmen, dass `T1`, `T5` und `T7` jeweils nacheinander und nicht parallel auszuführen sind. Betrachtet man jedoch den Taskgraph-Ausdruck, der in Zeile 2 und 3 als Kommentar beigefügt ist, wird klar, dass `T5` und `T7` gar nicht abhängig sind und parallel ausgeführt werden können. Hier liefert der Greedy-Algorithmus zur Erzeugung paralleler und sequentieller Sektionen (Kapitel 4.9.1) also kein optimales Ergebnis und bildet damit einen weiteren Ansatzpunkt für kommende Verbesserungen, solange Taskgraph-Ausdrücke noch nicht in die Architekturbeschreibungssprache TADL aufgenommen wurden. Für die Leistungsmessungen wurde der folgende, aus der Taskgraph-Notation ermittelte, Ausdruck umgesetzt: `(T0;T1);((T2;T3;T4;T5)||(T6;T7;T8;T9));(T10;T11)`. Nach der Sortierung der Punktmenge kann die Berechnung der oberen Hälfte der konvexen Hülle also parallel zur Berechnung der unteren Hälfte stattfinden.

```
 1 public static List<Point2D> GetConvexHull2D(List<Point2D> points) {
 2  /* T0(), T1(T0), T2(), T3(T1, T2), T4(T3), T5(T4), T6(T1),
 3     T7(T6), T8(T7), T9(T8), T10(T2), T11(T3, T5, T6, T9, T10) */
 4  #region (T0||T2);(T1||T10);(T3||T6);(T4||T7);(T5||T8);(T9);(T11)
 5   T0: var vComparer = new Point2DComparer { ComparisonType = … };
 6   T1: points.Sort(vComparer.Compare);
 7   T2: List<Point2D> upper = new List<Point2D>();
 8   T3: upper.Add(points[0]);
 9   T4: upper.Add(points[1]);
10   T5:
11    for (int i = 2; i < points.Count; i++) {
12     upper.Add(points[i]);
13      while (upper.Count > 2 && MathUtility.CrossProduct(…) > 0) {
14       upper.RemoveAt(upper.Count - 2);
15      }
16     }
17   T6: var lower = new List<Point2D> { points[points.Count - 1], … };
18   T7:
19    for (int i = points.Count - 3; i >= 0; i--) {
20     lower.Add(points[i]);
21      while (lower.Count > 2 && MathUtility.CrossProduct(…) > 0) {
22       lower.RemoveAt(lower.Count - 2);
23      }
24     }
25   T8: lower.RemoveAt(0);
26   T9: lower.RemoveAt(lower.Count - 1);
27   T10: var output = upper;
28   T11: output.AddRange(lower);
29  #endregion
30  return output;
31 }
```

Abbildung 47: Korrekter und leistungssteigernder Vorschlag für die Methode `GetConvexHull2D()` nach der Abänderung der *solution*.

6.5.2 Leistungsmessungen

Um die erzielbare Beschleunigung zu messen, wurde die Methode `GetConvexHull2D()` wie bereits in Kapitel 6.5.1 beschrieben parallelisiert: im Wesentlichen wird die Berechnung der oberen Hälfte der konvexen Hülle parallel zur Berechnung der unteren ausgeführt, nachdem die Sortierung der Punktmenge abgeschlossen ist. Die Laufzeitmessungen zeigen, dass sich eine Beschleunigung von bis zu 1,2 erzielen lässt. Im Idealfall kann hier eine Beschleunigung von maximal 2 erwartet werden, weil maximal zwei Aufgaben nebenläufig ausgeführt werden. Sie fällt geringer aus, weil die Sortierung der Punktmenge einen wesentlichen sequentiellen nicht parallelisierbaren Anteil an der Gesamtlaufzeit besitzt. Den Messungen ist zu entnehmen, dass die Beschleunigung mit wachsender Eingabegröße sinkt. Das ist jedoch zu erwarten: der Aufwand der Sortierung der Punktmenge in Abhängigkeit der Eingabegröße N ist O(N log N), wobei der Aufwand der beiden `for`-Schleifen nur linear mit O(N) wächst. Weil nur die beiden Schleifen parallel und die Sortierung sequentiell durchgeführt werden, skaliert die Beschleunigung daher nicht mit der Eingabe.

Der Tuning-Parameter `maxRecursionDepth` wurde (wie bereits bei `RayTracer` und `DesktopSearch`) nicht variiert, weil die Methode `GetConvecHull2D()` nicht rekursiv aufgerufen wird und er damit keinen Einfluss auf die Laufzeit nimmt.

Listengröße	Sequentiell (s/10)	Parallel (s/10)	Speedup
1.000.000	8,06	6,87	1,2
2.500.000	23,04	20,22	1,1
5.000.000	51,37	45,77	1,1

Tabelle 14: Laufzeitmessungen für die parallelisierte Variante von `CompGeo` für verschiedene Eingabegrößen.

6.5.3 Protokollierungsmehraufwand

Die bis zu 1080-fache Verlangsamung stellt den bisher größten beobachteten Faktor dar. Im Gegensatz dazu fällt der 49-fache Speicherfaktor durchschnittlich aus. Grund für die erhebliche Verlangsamung ist die Sortierung der Punktmenge, welche in der Klasse `BigList` durch eine Art iteratives Sortieren durch Mischen implementiert ist, weshalb der Faktor in einer ähnlichen Größenordnung erwartet wird wie bei `MergeSort`. Im Vergleich zu `MergeSort` fällt der Verlangsamungsfaktor noch einmal größer aus, weil es sich bei `BigList` einerseits um eine dynamische Datenstruktur und nicht um ein Array handelt und zudem noch die Berechnung der konvexen Hülle zur Verlangsamung beiträgt.

	Original		Instrumentiert		Faktor	
Listengröße	Zeit	Speicher	Zeit	Speicher	Zeit	Speicher
25000	0,02 s	12 MB	4,56 s	63 MB	228	5
50000	0,04 s	13 MB	31,12 s	348 MB	778	27
100000	0,08 s	15 MB	86,36 s	734 MB	1080	49

Tabelle 15: Gegenüberstellung der Laufzeit und des Speicherverbrauchs von instrumentierter und ursprünglicher `CompGeo`-Implementierung.

Es bleibt zu erwähnen, dass auch in diesem Fallbeispiel die Menge der dynamisch ermittelten Abhängigkeiten unabhängig von der Eingabe war.

6.6 VideoProcessing

`VideoProcessing` ist eine Anwendung, die im Rahmen einer Studienarbeit [D08] entstand und dort zur Evaluierung von parametrisierbaren Fließbändern verwendet wurde. Die Anwendung wendet sieben Bildfilter nacheinander auf die Einzelbilder eines Videostroms an, wodurch ein neuer Videostrom erzeugt wird.

Weil die Klassen zur Kodierung und Dekodierung von Videos aufgrund von auftretenden Ausnahmen (engl. *exceptions*) nicht verwendet werden konnten, wurde die Anwendung so abgeändert, dass Einzelbilder aus einem Verzeichnis ausgelesen und wieder dorthin zurückgeschrieben werden. An der Semantik der Anwendung oder den Abhängigkeiten zwischen Anweisungen ändert diese Abänderung nichts. Als Eingabe erwartet `VideoProcessing` daher ein Verzeichnis, das rekursiv nach Bildern durchsucht wird, die als Videostrom interpretiert, durch Bildfilter verändert und anschließend wieder zurückgeschrieben werden.

`VideoProcessing` lässt sich durch ein achtstufiges Fließband parallelisieren, wie bereits in [D08] und [O13] gezeigt wurde. Jeder Bildfilter kann dabei in eine eigene replizierbaren Stufe ausgelagert werden. Das zurückschreiben in den Videostrom kann ebenfalls in einer Fließbandstufe stattfinden. Weil jedoch die Reihenfolge der Einzelbilder erhalten bleiben muss, ist diese

Stufe nicht replizierbar. Diese Parallelisierungsmöglichkeit wird aufgefunden und als Parallelisierungsvorschlag ausgegeben.

Der Protokollierungsmehraufwand dieser Anwendung wurde nicht mehr gemessen, weil bereits durch die Messungen der vorherigen Fallbeispiele ein Gefühl für die zu erwarteten Faktoren für Zeit und Speicher vermittelt werden konnte.

6.6.1 Kategorisierung der Parallelisierungsvorschläge

Für `VideoProcessing` wird genau ein Vorschlag ausgegeben, der sich mit dem erwarteten Vorschlag deckt und daher korrekt und leistungssteigernd ist. Er führt zu einer Beschleunigung von bis zu 5,3. Tabelle 16 Tabelle 6gibt eine Übersicht der generierten Vorschläge wieder. Die Spalten sind wie in Kapitel 6.2.1, Tabelle 3 gewählt.

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
1	0	0	1	0	0

Tabelle 16: Übersicht der generierten Vorschläge für `VideoProcessing`.

- Der Vorschlag bezieht sich auf die Methode `ProcessImages()` und ist in Abbildung 48 dargestellt. Es wird ein neunstufiges Fließband vorgeschlagen, das bis auf die Stufe `S9` nur aus replizierbaren Stufen besteht. Es ist leicht zu erkennen, dass die erste Stufe `S1` nichts weiter tut, als eine Variable zu deklarieren und zuzuweisen, weshalb sich die Auslagerung dieser Anweisung in eine Fließbandstufe niemals lohnen wird. Ein Auto-Tuner würde daher den Tuning-Parameter `stageFusion` für die Stufen `S1` und `S2` auf wahr setzen, um eine Zusammenlegung der Stufen zu veranlassen. Um einem Auto-Tuner Arbeit abzunehmen und dessen Suchraum sinnvoll einzuschränken, wäre es denkbar solche offensichtlichen Fälle durch eine statische Analyse des Quellcodes zu erkennen und daraufhin einen Vorbelegungsvorschlag für den Tuning-Parameter zu generieren.

```
 1 public void ProcessImages() {
 2  #region S1+ => S2+ => S3+ => S4+ => S5+ => S6+ => S7+ => S8+ => S9
 3   foreach (Bitmap bmp in _inputStream) {
 4    S1: var tmp_bmp = bmp;
 5    S2: tmp_bmp = doCrop(tmp_bmp);
 6    S3: tmp_bmp = doHistogramEqualization(tmp_bmp);
 7    S4: tmp_bmp = doOilPainting(tmp_bmp);
 8    S5: tmp_bmp = doResize(tmp_bmp);
 9    S6: tmp_bmp = doSharpen(tmp_bmp);
10    S7: tmp_bmp = doSepia(tmp_bmp);
11    S8: tmp_bmp = ConvertColorsTo32bppArgb(tmp_bmp);
12    S9: ConsumeBmp(tmp_bmp);
13   }
14  #endregion
15 }
```

Abbildung 48: Korrekter und leistungssteigernder Parallelisierungsvorschlag für die Methode `ProcessImages()`.

6.6.2 Leistungsmessungen

Für die Leistungsmessungen wurde die Methode `ProcessImages()` durch ein Fließband mit den Stufen `S18` und `S9` parallelisiert: `S18` fasst dabei die replizierbaren Stufen `S1` bis `S8` zu

einer einzigen replizierbaren Stufe zusammen. Der Tuning-Parameter `stageFusion` für die Stufen `S1` bis S8 wurde dadurch fest auf wahr gesetzt, weil die momentane Implementierung der Laufzeitbibliothek bisher keine Variation des Parameters zulässt, wie bereits erwähnt wurde. Der Tuning-Parameter `numThreads` der Stufe `S18` (`nT(S18)`) wurde bei den Laufzeitmessungen variiert. In der Arbeit [D08] wurde nach der manuellen Optimierung der Fließbandparameter mit dieser Konfiguration des Fließbandes bei `nT(S18)` = 8 eine Beschleunigung von 6,2 erzielt. Während den Testmessungen dieser Arbeit konnte hingegen nur eine Beschleunigung von 5,3 erzielt werden, obwohl Ort und Art der Parallelisierung gleich sind. Trotz unterschiedlicher Eingabedaten und Testrechner, lässt dieses Ergebnis darauf schließen, dass die Laufzeitbibliothek noch weiter optimiert werden kann, was der gleichartigen Vermutung aus Kapitel 6.3.2 zusätzlich Gewicht verleiht.

Ordner	Sequentiell (s/10)	Tuning-Parameter `nT(S18)`	Parallel (s/10)	Speedup
Images8	25,68	2	13,03	1,97
		4	8,16	3,15
		8	5,94	4,3
		10	5,90	4,3
Images64	208,84	2	106,31	2,0
		4	61,35	3,4
		8	39,69	5,3
		10	39,58	5,3

Tabelle 17: Laufzeitmessungen für die parallelisierte Variante von `VideoProcessing` für variierende Werte für den Tuning-Parameter `numThreads` der Stufe `S18` (`NT(S18)`).

6.7 PowerCollections

`PowerCollections` [PC] ist eine quelloffene Bibliothek, die generische Datenstrukturen bereitstellt, welche nicht in der Standardbibliothek von .NET vorzufinden sind. Zu diesen Datenstrukturen gehören unter anderem `MultiDictionary` (lässt im Gegensatz zu `Dictionary` mehrere Werte pro Schlüssel zu), `Bag` (lässt im Gegensatz zu `Set` Duplikate zu), `OrderedBag` (hält Elemente im Gegensatz zu `Bag` sortiert vor) und viele weitere.

Wie `CompGeo` ist `PowerCollections` eine Bibliothek und damit nicht direkt ausführbar. Allerdings stehen für `PowerCollections` 644 Testfälle (ca. weitere 26.400 Zeilen Quellcode) in Form von Unit-Tests zur Verfügung, die zur Ausführung herangezogen werden können. Weil der Aufruf `Logger.Save()` nicht wie sonst am Ende der `Main()`-Methode hinzugefügt werden kann (siehe Kapitel 5.1.1), wird dieser Aufruf in einen eigenen Testfall eingebettet. Dieser Testfall muss angestoßen werden, nachdem alle anderen Testfälle ausgeführt wurden.

Aufgrund der Größe von `PowerCollections` werden die ausgegebenen Parallelisierungsvorschläge nur stichprobenartig ausgewertet und diskutiert. Laufzeitmessungen und die Bemessung des Mehraufwands während der Protokollierung wurden für dieses Projekt nicht durchgeführt.

6.7.1 Kategorisierung der Parallelisierungsvorschläge

Für `PowerCollections` werden insgesamt 172 Parallelisierungsvorschläge ausgegeben. Die Auswertung der Vorschläge erfolgt daher nur stichprobenartig. Zur Auswertung wurde die

Klasse `Algorithms` herangezogen, die selbst 23 Vorschläge enthält. Von diesen Vorschlägen wurden insgesamt 10 als Pseudovorschläge identifiziert, bei denen Abhängigkeiten übersehen werden, weil die Klassen `Array`, `List`, `Dictionary`, `StringBuilder` und `Random` nicht instrumentiert werden. Wie bereits zuvor gezeigt wurde, werden diese Abhängigkeiten berücksichtigt, wenn der *solution* der Quellcode dieser Klassen hinzugefügt wird. Nach der Abänderung der *solution* fallen 8 der 10 Pseudovorschläge weg, zwei werden unter Berücksichtigung der Abhängigkeiten korrigiert, führen aber nicht zu einer Beschleunigung. Von den verbleibenden 13 Vorschlägen sind weitere sieben korrekt, jedoch ebenfalls nicht leistungssteigernd. Die restlichen sechs Vorschläge werden im Folgenden kurz diskutiert: drei sind korrekt und leistungssteigernd, einer weder eindeutig korrekt noch inkorrekt und zwei inkorrekt. Tabelle 18 gibt eine Übersicht der generierten Vorschläge wieder. Die Spalten sind wie in Kapitel 6.2.1, Tabelle 3 gewählt.

Vorschläge	Pseudovorschläge	Wegfallende Pseudovorschläge	K & L	K & ¬L	¬K
23	10	8	4	9	2

Tabelle 18: Übersicht der generierten und untersuchten Vorschläge für `PowerCollections`.

- Zwei korrekte und leistungssteigernde Vorschläge beziehen sich jeweils auf eine Variante der überladenen generische Methode `ReplaceInPlace<T>()`, die über eine Liste iteriert und Elemente ersetzt, die eine bestimmte Ersetzungsbedingung erfüllen (Abbildung 49). Es wird vorgeschlagen, die zugrundeliegende `for`-Schleife durch ein Fließband mit einer replizierbaren Stufe `S1` zu ersetzen. Weil sich die Größe der Liste beim Iterieren über die Liste nicht ändert und in jeder Iteration auf ein anderes Listenelement zugegriffen wird, sind beide Vorschläge korrekt. Ab einer gewissen Mindestanzahl von Listenelementen oder einer entsprechend aufwändigen Überprüfung der Ersetzungsbedingung führen beide Vorschläge auch zu einer Beschleunigung.

```
 1 public static void ReplaceInPlace<T>(IList<T> list, T itemFind, …) {
 2  …
 3  int listCount = list.Count;
 4  #region S1+
 5   for (int index = 0; index < listCount; ++index) {
 6    S1:
 7     if (equalityComparer.Equals(list[index], itemFind)) {
 8      list[index] = replaceWith;
 9     }
10   }
11  #endregion
12 }
```

Abbildung 49: Korrekter und leistungssteigernder Parallelisierungsvorschlag für die Methode `ReplaceInPlace<T>()`.

- Ein weiterer korrekter und leistungssteigernder Vorschlag bezieht sich auf die generische Methode `SortInPlace<T>()`, die eine iterative Variante des QuickSort-Algorithmus implementiert (Abbildung 50). Bei der Suche nach zwei Elementen `item_i` und `item_j` wird zweimal nacheinander über die zu sortierende Liste iteriert. Es wird vorgeschlagen, diese Elemente parallel zueinander zu suchen. Weil in beiden Schleifen auf getrennte Variablen geschrieben wird, ist der Vorschlag korrekt. Für aufwändige Vergleiche und große Listen ist dieser Vorschlag zudem leistungssteigernd.

```
 1 public static void SortInPlace<T>(IList<T> list, IComparer<T> comparer) {
 2  …
 3  #region (T0 || T1)
 4   T0:
 5    do {
 6     ++i;
 7     item_i = list[i];
 8    } while (comparer.Compare(item_i, partition) < 0);
 9   T1:
10    do {
11     --j;
12     item_j = list[j];
13    } while (comparer.Compare(item_j, partition) > 0);
14  #endregion
15  …
16 }
```

Abbildung 50: Korrekter und leistungssteigernder Parallelisierungsvorschlag für die Methode `SortInPlace<T>()`.

- Ein inkorrekter Vorschlag bezieht sich auf die Methode `IndicesOfMany<T>()`, welche die Indizes einer Menge von Elementen zurückgibt. Aufgrund des Schlüsselworts `yield` ist der Vorschlag inkorrekt: `yield return index` sorgt dafür, dass einem implizit angelegten Enumerations-Objekt der Wert `index` hinzugefügt wird. Die parallele Ausführung der Schleife würde daher zu einem Wettlauf führen. Diese Sprachbesonderheit wurde beim Entwurf des Konzepts nicht berücksichtigt. Inkorrekte Vorschläge dieser Art können in Zukunft aber vermieden werden, indem Syntaxknoten mit `yield` als offene Syntaxknoten markiert werden. Dadurch werden solche Syntaxknoten weder als Tasks, noch als Stufen eines Fließbands angenommen (siehe Kapitel 4.8).

```
 1 public static IEnumerable<int> IndicesOfMany<T>(IList<T> list, …) {
 2  …
 3  int index = 0;
 4  foreach (T x in list) {
 5   #region S1+
 6    foreach (T y in itemsToLookFor) {
 7     S1:
 8      if (predicate(x, y)) {
 9       yield return index;
10      }
11     }
12   #endregion
13   ++index;
14  }
15 }
```

Abbildung 51: Inkorrekter Parallelisierungsvorschlag aufgrund des Schlüsselworts `yield`.

- Der andere inkorrekte Vorschlag bezieht sich auf die generische Methode `SearchForSubsequence<T>`, die eine Liste nach einer Teilsequenz durchsucht. Aufgrund des Schlüsselworts `goto` ist der Vorschlag inkorrekt: es ist unklar, wie der Quellcode der Zeilen 7 bis 9 in eine Fließbandstufe ausgelagert werden soll, weil `goto NOMATCH` dann vom Sprungziel `NOMATCH` getrennt wird. Auch hier ist es möglich Vorschläge dieser Art in Zukunft zu unterbinden, wenn Syntaxknoten mit `goto` als offene Syntaxknoten markiert werden, wenn deren Sprungziel außerhalb des Syntaxknoten liegt.

```
 1 public static int SearchForSubsequence<T>(IList<T> list, …) {
 2  …
 3  for (int start = 0; start <= listCount - patternCount; ++start) {
 4   #region S1+
 5    for (int count = 0; count < patternCount; ++count) {
 6     S1:
 7      if (!predicate(list[start + count], patternArray[count])) {
 8       goto NOMATCH;
 9      }
10     }
11   #endregion
12   return start;
13   NOMATCH: ; /* no match found at start. */
14  }
15  return -1;
16 }
```

Abbildung 52: Inkorrekter Parallelisierungsvorschlag aufgrund des Schlüsselworts `goto`.

- Der letzte Vorschlag, den es zu diskutieren gilt, bezieht sich auf die generische Methode `ForEach<T>()` aus Abbildung 53. Es wird vorgeschlagen, die `foreach`-Schleife zu parallelisieren, die über alle Elemente iteriert und darauf die Aktion `a` ausführt. Weil während der Unit-Tests keine Aktion übergeben wurde, die Abhängigkeiten verursacht, ist dieser Vorschlag als korrekt (und für aufwändige Aktionen oder eine große Anzahl an Elementen als leistungssteigernd) zu werten. Allerdings handelt es sich bei `PowerCollections` um eine Bibliothek: ihr Verwendungszweck ist nicht auf die Unit-Tests beschränkt. Insbesondere ist es natürlich möglich, für `a` eine Aktion zu übergeben, die für Abhängigkeiten zwischen den Iterationen sorgt. Dann wäre der Vorschlag inkorrekt. Dieser Vorschlag ist daher ein gutes Beispiel um die Eingabeabhängigkeit dynamischer Analysen zu demonstrieren.

```
1 public static void ForEach<T>(IEnumerable<T> collection, Action<T> a) {
2  …
3  #region S1+
4   foreach (T item in collection) {
5    S1: a(item);
6   }
7  #endregion
8 }
```

Abbildung 53: Eingabeabhängiger Vorschlag für die Methode `ForEach<T>()`.

6.8 Erweiterungsmöglichkeiten des Konzepts

Im Folgenden soll ein Grenzfall demonstriert werden, für welchen kein Parallelisierungsvorschlag ausgegeben wird, obwohl Parallelität möglich ist und auch zu einer Beschleunigung führt: In der `for`-Schleife aus Abbildung 54 wird die Summe aller Werte des Arrays `numbers` berechnet. Als Akkumulationsvariable wird `sum` verwendet. Da `sum` in jeder Iteration zunächst gelesen und dann geschrieben wird, stellt die statische Abhängigkeitsanalyse eine iterationsübergreifende Abhängigkeit von Zeile 4 zu sich selbst fest und gibt damit keinen Parallelisierungsvorschlag für die Schleife aus. Mit dem Kontextwissen, dass die Addition eine assoziative Operation ist, lässt sich die Schleife jedoch parallelisieren: das Intervall [0, N] wird in mehrere Teilintervalle aufgeteilt, die Summe für jedes Teilintervall parallel berechnet und schließlich aufaddiert.

```
1 int CalculateSum(int[] numbers) {
2  int sum = 0;
3  for(int i = 0; i < numbers.Length; i++) {
4   sum = sum + numbers[i];
5  }
6  return sum;
7 }
```

Abbildung 54: Obwohl eine die Parallelisierung von `CalcualteSum()` möglich ist, wird kein Parallelisierungsvorschlag ausgegeben.

Um solche Fälle abzudecken, müssten assoziative Operationen erkannt und die daraus folgenden Abhängigkeiten ignoriert werden. Dies stellt einen möglichen Ansatzpunkt für zukünftige Arbeiten dar.

6.9 Zusammenfassung

In der Evaluierung wurde untersucht, welche Parallelisierungsvorschläge für die Fallbeispiele ausgegeben werden. Die ausgegebenen Vorschläge wurden per Hand darauf untersucht, ob sie dadurch zustande kommen, weil Abhängigkeiten aufgrund von technischen Unzulänglichkeiten der Implementierung übersehen wurden. Diese Vorschläge wurden als Pseudovorschläge identifiziert. Um das Konzept evaluieren zu können, wurden die Projekte angepasst um die technischen Unzulänglichkeiten der Implementierung zu umgehen: den Projekten wurde Quellcode von referenzierten Klassen, die nur als *assembly* vorliegen, hinzugefügt. Dadurch konnten die zuvor übersehenen Abhängigkeiten entdeckt und vom Werkzeug berücksichtigt werden. Die Pseudovorschläge fallen daraufhin weg, oder werden vom Werkzeug unter der Berücksichtigung der übersehenen Abhängigkeiten angepasst.

Tabelle 19 zeigt, dass sich der Suchraum, den ein Programmierer zur Parallelisierung einer Anwendung erkunden muss, durch das vorgestellte Konzept drastisch einschränken lässt. Für jedes der Fallbeispiele ist zunächst angegeben, wie viele Sequenzen und Schleifen es enthält (siehe dazu auch Tabelle 2). In der nächsten Spalte ist angegeben, wie viele Vorschläge in erster Instanz für jedes der Fallbeispiele ausgegeben werden. Daraufhin wird aufgezeigt, wie viele dieser Vorschläge wegfallen, sobald ein Projekt abgeändert wurde um Pseudovorschläge zu unterbinden. In der letzten Spalte ist schließlich die relative Suchraumreduktion angegeben: wie viel Prozent der Sequenzen und Schleifen scheiden nachdem Wegfallen von Pseudovorschlägen für die Parallelisierung aus. Im Durchschnitt über alle Fallbeispiele lässt sich mit dem vorgestellten Konzept eine relative Suchraumeinschränkung ca. 95% erzielen.

Projekt	Sequenzen und Schleifen	Vorschläge	Wegfallende Pseudovorschläge	Surchraum-reduktion
`MergeSort`	19	2	1	95 %
`RayTracer`	63	7	1	90 %
`DesktopSearch`	47	4	2	96 %
`CompGeo`	169	4	3	98 %

VideoProcessing	16	1	0	94 %
PowerCollections	3.641	172	60[7]	97 %

Tabelle 19: Veranschaulichung der erzielten relativen Suchraumeinschränkung

Der Suchraum wird aber nicht nur eingeschränkt: die erwarteten Parallelisierungsmöglichkeiten befinden sich auch unter den ausgegebenen Parallelisierungsmöglichkeiten. Diese Aussage konnte für die fünf kleineren Fallbeispiele belegt werden, für PowerCollections ist das nur eine begründete Annahme. Die Trefferquote (engl. *recall*) des vorgestellten Konzepts ist für die Fallbeispiele damit 100%. In Kapitel 6.8 wurde aber auch ein Beispiel aufgezeigt, für das kein Parallelisierungsvorschlag ausgegeben wird, obwohl Parallelität möglich ist. Unter der Betrachtung weiterer Fallbeispiele wird die 100%-ige Trefferquote also nicht zu halten sein.

Um die Qualität der ausgegebenen Parallelisierungsvorschläge zu beurteilen, wurde nach der Aussonderung von Pseudovorschlägen für jeden Vorschlag manuell überprüft, ob er korrekt ist und ob er zu Leistungssteigerungen führt. Die Ergebnisse dieser Untersuchung fasst Tabelle 20 zusammen. In der Spalte „Vorschläge“ sind die ausgegebenen Vorschläge ohne wegfallende Pseudovorschläge angegeben. Sie ergibt sich also aus der Differenz der Spalten drei und vier der Tabelle 19. Korrekte und leistungssteigernde Vorschläge sind unter K & L angegeben, korrekte aber nicht leistungssteigernde unter K & ¬L, inkorrekte unter ¬K. Außer den korrekten und leistungssteigernden Parallelisierungsvorschlägen, die sich sinnvoll für die Parallelisierung einer Anwendung darstellen (K & L), werden weitere Vorschläge ausgegeben. Diese Vorschläge sollen bei einer Parallelisierung nicht umgesetzt werden, weil sie entweder nicht zu einer Beschleunigung, sondern zu einer Verlangsamung (K & ¬L) oder gar einem Fehlverhalten (¬K) der Anwendung führen. Das Verhältnis zwischen „guten“ und „schlechten“ Vorschlägen wird Präzision (engl. *precision*) genannt und ist in der letzten Spalte dargestellt. Im Durchschnitt über alle Fallbeispiele erreicht das vorgestellte Konzept eine Präzision von 66%, etwa jeder dritte ausgegebene Vorschlag sollte also nicht in der Ergebnismenge sein.

Projekt	Vorschläge	K & L	K & ¬L	¬K	Präzision
MergeSort	1	1	0	0	100 %
RayTracer	6	1	5	0	17 %
DesktopSearch	2	1	1	0	50 %
CompGeo	1	1	0	0	100 %
VideoProcessing	1	1	0	0	100 %
PowerCollections[8]	15	4	9	2	27 %
	$\sum = 26$	$\sum = 9$	$\sum = 15$	$\sum = 2$	Ø = 66%

Tabelle 20: Veranschaulichung erzielten Präzision

Durch eine Aussonderung von Parallelisierungsvorschlägen, die nicht zu Beschleunigungen führen, könnte die Präzision drastisch erhöht werden. Eine manuelle Überprüfung ergab, dass 5 der insgesamt 15 K & ¬L Vorschläge aussortiert werden können, wenn lediglich Informationen

[7] Für PowerCollections ist in dieser Spalte nur ein Schätzwert angegeben, der aus einer Hochrechnung der ausgewerteten Stichprobe ermittelt wurde: von 23 ausgewerteten Vorschlägen waren 8 Pseudovorschläge, die nach der Abänderung des Projekts wegfallen. Dadurch ergibt sich der angegebene Wert von 172*8/23 ≈ 60.

[8] Für PowerColldections sind die Zahlen in dieser Zeile nur für die 23 tatsächlich ausgewerteten Vorschläge aufgetragen.

über die Laufzeit einzelner Methoden vorliegen: Ein Vorschlag wird ausgesondert, wenn die Laufzeit der Methode in der er sich befindet, zu gering ist. Wenn zusätzlich Laufzeitinformationen über einzelne Schleifen und Anweisungen bekannt sind, lassen sich alle der 15 Vorschläge aussortieren.

Dass die als korrekt und leistungssteigernd identifizierten Vorschläge tatsächlich zu einer Beschleunigung führen, wurde durch die manuelle Parallelisierung und anschließende Leistungsmessungen untermauert. Eine Übersicht der erzielten Beschleunigungen unter den besten getesteten Tuning-Parameter-Konfigurationen ist in Abbildung 55 dargestellt.

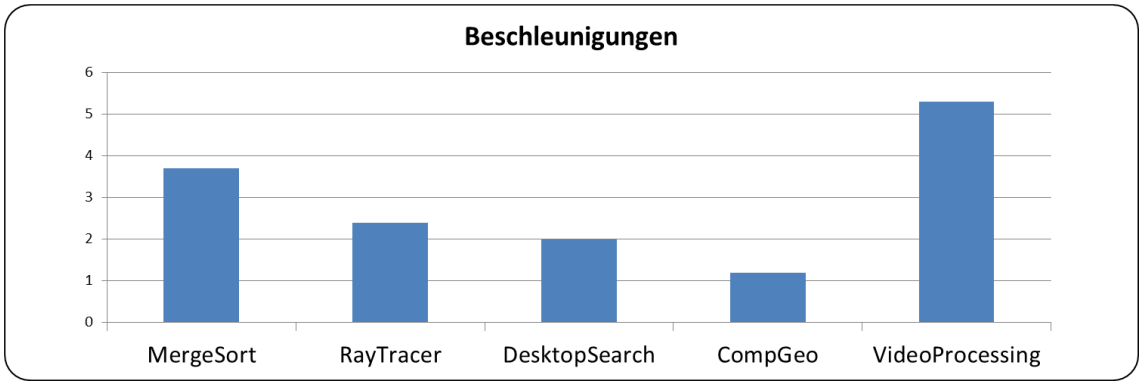

Abbildung 55: Übersicht der erzielten Beschleunigungen der Fallbeispiele.

Es zeigt sich, dass Beschleunigungen von 1,2 bei `CompGeo` und bis zu 5,3 bei `VideoProcessing` möglich sind. Wie in den jeweiligen Abschnitten zur Leistungsmessung der Projekte diskutiert wurde, sind dargestellten Beschleunigungen als untere Schranke zu verstehen: in anderen Arbeiten wurden für `MergeSort` [O13] und `VideoProcessing` [D08] bereits höhere Beschleunigungen berichtet, obwohl sich die dort umgesetzte Parallelität vollständig mit jeweiligen den Parallelisierungsvorschlägen dieser Arbeit deckt: Ort und Architektur der Parallelisierung sind identisch. Des Weiteren konnte gezeigt werden, dass die erzeugten Tuning-Parameter einen wesentlichen Einfluss auf die erzielbare Beschleunigung besitzen und damit einen wichtigen Mehrwert dieser Arbeit darstellen.

Aus der Evaluierung ergeben sich im Wesentlichen fünf Anknüpfungspunkte für weiterführende Arbeiten:

- **Verhinderung von Pseudovorschlägen:** Um Pseudovorschläge in Zukunft nicht mehr behandeln zu müssen, ist die Implementierung zu erweitern. Dazu ist es nötig, die Instrumentierung von *assemblies* auf referenzierte *assemblies* auszuweiten. Interessant ist an dieser Stelle, wie die Instrumentierung der Standardbibliothek und insbesondere der *assembly* `mscorlib.dll` gelingt. Die Instrumentierung von `mscorlib.dll` ist deshalb nicht trivial, weil bei der Initialisierung (engl. *bootstrapping*) der Laufzeitumgebung von .NET möglicherweise eine bestimmte Ladereihenfolge von Klassen angenommen wird, die durch die Instrumentierung beeinflusst werden kann. Dieses Problem ist auch für Java bekannt und kann weitestgehend wie in [MB+11] gelöst werden. Des Weiteren ist die Instrumentierung von *assemblies* so zu erweitern, dass auch Zugriffe auf Arrays mit Referenztyp durch Zugriffsanweisungen instrumentiert werden. Anstatt den IL-Anweisungen `ldelem` und `stelem` wird bei diesen Arrays eine Kombination der Anweisungen `ldelema` und `stobj` verwendet: `ldelema` konsumiert einen Array-Zeiger und einen Index und legt die Adresse des Arrayelements auf den

Stapel, `stobj` konsumiert diesen Zeiger und zusätzliche Objekt, das daraufhin im Array gespeichert wird.

- **Aussondern von nicht leistungssteigernden Vorschlägen**: Um Parallelisierungsvorschläge auszusondern, die nicht zu einer Beschleunigung führen, können Laufzeitinformationen von Methoden, Schleifen oder gar einzelnen Anweisungen herangezogen werden. Werden gewisse Schwellwerte unterschritten, wird ein Vorschlag ausgesondert. Laufzeitinformationen lassen sich durch externe Werkzeuge wie beispielsweise HotSpot-Profiler (Kapitel 3.1) generieren.
- **Verhinderung von inkorrekten Vorschlägen:** Bei der Evaluierung traten zwei inkorrekte Vorschläge auf, die sich dadurch erklären lassen, dass bestimmte Sprachkonstrukte beim Entwurf des Konzepts noch nicht berücksichtigt wurden. Für zwei dieser Sprachkonstrukte (`yield return` und `goto`) wurde bereits in Kapitel 6.7.1 darauf eingegangen, wie das Konzept zur Berücksichtigung dieser Konstrukte zu erweitern ist. In zukünftigen Arbeiten ist zu untersuchen, welche Sprachkonstrukte noch zu unterstützen sind, und wie sich diese in das vorhandene Konzept integrieren lassen.
- **Identifizieren assoziativer und kommutativer Operationen:** Durch die Identifizierung assoziativer und kommutativer Operationen kann mehr Parallelisierungspotential ausgeschöpft werden als bisher: Abhängigkeiten dieser Operationen können bei der Abbildung von Sequenzen und Schleifen auf Taskgraphen und Fließbänder ausgeklammert werden, was beispielhaft in Kapitel 6.8 aufgezeigt wurde. Das Wissen um die Assoziativität oder Kommutativität einer Operation ist jedoch stark kontextbasiert: ohne die Kenntnis des konkreten Anwendungsfalls ist es beispielsweise nicht möglich zu entscheiden, ob die Reihenfolge von Elementen einer Liste für das Ergebnis eine Rolle spielt, oder ob die Liste nur als Menge verwendet wird und ein Hinzufügen vom Elementen zur Liste daher als assoziative Operation betrachtet werden kann.
- **Einführung einer Taskgraph-Notation/Verbesserung des Greedy-Algorithmus:** Bereits in Kapitel 4.9.1 wurde darauf eingegangen, dass die Taskgraph-Notationen mächtiger ist, als die Notation aus parallelen und sequentiellen Sektionen, weshalb eine Erweiterung von TADL vorgeschlagen wurde. In der Evaluierung hat sich gezeigt, dass diese Erweiterung tatsächlich Vorteile bringt: der Ausdruck paralleler und sequentieller Sektionen in Kapitel 6.5.1 führt nicht zu einer Beschleunigung, der zugrundeliegende Taskgraph-Ausdruck aber sehr wohl. Im besagten Beispiel würde es allerdings auch genügen, einen verbesserten Greedy-Algorithmus (siehe dazu Kapitel 4.9.1) zur Erzeugung eines Ausdrucks aus parallelen und sequentiellen Sektionen einzusetzen.

7 ZUSAMMENFASSUNG UND AUSBLICK

Ziel dieser Diplomarbeit war die automatische Identifizierung von Parallelisierungsmöglichkeiten sequentieller Anwendungen. Parallelisierungsvorschläge sollten automatisch generiert und in Form von Architekturbeschreibungen ausgegeben werden. Mit dem vorgestellten Konzept und dessen prototypischen Implementierung konnte in der Evaluierung gezeigt werden, dass dieses Ziel erreicht wurde: Für jedes der evaluierten Fallbeispiele wurde diejenige Stelle der Anwendung gefunden, die auch ein Entwickler zur Parallelisierung heranziehen würde. Es wurde jedoch nicht nur die konkrete Stelle gefunden, sondern zusätzlich exakt beschrieben, wie die Parallelisierung durch einen Taskgraphen oder ein Fließband vorzunehmen ist. Damit liefert diese Arbeit ein Verfahren, welches sequentielle Anwendungen vollautomatisch parallelisiert und zudem Tuning-Parameter zur Optimierung der parallelen Anwendung identifiziert. Die von dieser Arbeit vorgeschlagene Parallelisierung für die Fallbeispiele `MergeSort`, `Desktop-Search` und `VideoPipeline` ist „kostenlos“ und identisch mit derjenigen, die von „Parallelisierungsexperten“ nach einer gründlichen und zeitaufwändigen Analyse der jeweiligen Anwendung umgesetzt wurde.

Für die Evaluierung wurde die Parallelisierung zwar manuell durchgeführt, die automatische Codetransformation wird jedoch bereits von einer Folgearbeit angestrebt [W13]. Durch Laufzeitmessungen der manuell parallelisierten Anwendungen konnte nicht nur gezeigt werden, dass die Parallelisierungsvorschläge zu einer Beschleunigung von bis zu 5,3 auf einem Achtkernsystem führen, sondern auch, dass eine optimale Wahl der Werte der von diesem Verfahren automatisch zutage geförderten Tuning-Parameter einen entscheidenden Einfluss auf die erzielbare Beschleunigung besitzt und Tuning-Parameter daher einen wichtigen Mehrwert dieser Arbeit darstellen. Die gemessenen Beschleunigungen stellen allenfalls untere Schranken dar: im Fall von `VideoProcessing` wurde mit dem Verfahren dieser Arbeit eine Beschleunigung von bis zu 5,3 erzielt, während mit einer identischen manuellen Parallelisierung sogar Beschleunigungen von bis zu 6,2 berichtet wurden [D08]. Mit einer ausgereiften Laufzeitbibliothek sind daher noch größere Beschleunigungen zu erwarten.

In der Evaluierung wurde auch darauf eingegangen, dass momentan auch Parallelisierungsvorschläge ausgegeben werden, die zwar korrekt sind, jedoch nicht zu einer Beschleunigung führen, weil die zugrundeliegende Parallelität zu feingranular ist: die Laufzeitanteile der nebenläufigen Abschnitte sind also zu gering, sodass der Mehraufwand zur Erzeugung der Parallelität den durch die Parallelisierung möglichen Laufzeitvorteil überwiegt. Es wurde zudem gezeigt, wie die Umsetzung solcher Vorschläge die erzielbare Beschleunigung nach unten drückt und Laufzeitgewinne anderer Parallelisierungsvorschläge somit zunichtemacht. Die Aussortierung dieser Vorschläge kommt momentan einem sich anschließenden Auto-Tuner zu, der aufgrund der Tuning-Parameter die Möglichkeit besitzt, auf die ursprüngliche sequentielle Variante zurückzugreifen. Für zukünftige Arbeiten ist jedoch auch ein Ansatz denkbar, der von vornherein Laufzeitdaten über Methoden, Schleifen und Anweisungen miteinbezieht und dadurch Vorschläge aussondert, die nicht zu einer Beschleunigung führen. Mit einer solchen Erweiterung ließe sich die Präzision der automatisch generierten Vorschläge von aktuell 66% auf nahezu 100% erhöhen, weil praktisch alle Vorschläge ausgesondert werden könnten, die nicht zu einer Beschleunigung führen.

8 LITERATURVERZEICHNIS

[AF] AForge
[Online] http://code.google.com/p/aforge/

[AH+08] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh
"Using Static Analysis to Find Bugs"
IEEE Software (2008) Vol. 25, Nr. 5, 22-29

[B09] C. Breshears
"The Art of Concurrency"
O'Reilly (2009) ISBN-978-0-569-52153-0

[BS+11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, M. Mezini
"Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders"
International Conference on Software Engineering ICSE (2011), 241-250

[C03] V. T. Chakaravarthy
"New results on the computability and complexity of points–to analysis"
SIGPLAN Notices (2003) Vol. 38, Nr. 1, 115-125

[CBP] CriticalBlue Prism
[Online] http://www.criticalblue.com/criticalblue_products/prism.shtml

[Cci] Microsoft Common Compiler Infrastructure (CCI)
[Online] http://ccimetadata.codeplex.com/

[D08] M. Dempe
"Sprachkonzepte für Fließbandverarbeitung in C#"
Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2008)

[CG] CompGeo
[Online] http://compgeo.codeplex.com/

[D68] E. W. Dijkstra
"Go To Statement Considered Harmful"
Communications of the ACM (1968) Vol. 11, Nr. 3, 147-148

[E334] Standard ECMA-334, C# Language Specification
[Online] www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[E335] Standard ECMA-335, Common Language Infrastructure
[Online] www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

[FH+05] N. Flemming, R. N. Hanne, H. Chris
"Principles of Program Analysis"
Springer (2005) ISBN-3-540-65410-0

[GH+94] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides
"Design Patterns: Elements of Reusable Object-Oriented Software"
Addison-Wesley Professional (1994) ISBN-978-0201633610

[GJ+11] S. Garcia, D. Jeon, C. Louie, M.B. Taylor
"Kremlin: Rethinking and Rebooting gprof for the Multicore Age"
Conference on Programming language Design and Implementation PLDI (2011) 458-469

[H01] M. Hind
"Pointer analysis: haven't we solved this problem yet?"
Workshop on Program Analysis for Software Tools and Engineering PASTE (2001) 54-61

[H11] J. Huck
"Automatisierte Parallelisierung mit Auto-Futures"
Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2011)

[HP06] J. L. Hennessy, D. A. Patterson
"Computer Architecture: A Quantitative Approach"
Morgan Kaufmann (2006) ISBN-978-0123704900

[HS08] M. Herlihy, N. Shavit
"The Art of Multiprocessor Programming"
Morgan Kaufmann (2008) ISBN-978-0123705914

[HS+09] C. Hammacher, K. Streit, S. Hack, A. Zeller
"Profiling Java Programs for Parallelism"
Workshop on Multicore Software Engineering IWMSE (2009) 49-55

[IC11] Intel Corporation
"Intel(R) Parallel Advisor 2011 Getting Started Tutorial"
Document Number: 323355-003US.

[IPS] Intel Corporation Intel Parallel Studio
[Online] http://software.intel.com/en-us/intel-parallel-studio-home

[K88] M. Kumar
"Measuring Parallelism in Computation-Intensive Scientific Engineering Applications"
Transactions on Computers (1988) Vol. 37, Nr. 9, 1088-1098

[KK+10] M. Kim, H. Kim, C. Luk
"Prospector: A Dynamic Data-Dependence Profiler To Help Parallel Programming"
Workshop on Hot Topics in Parallelism HotPar (2010)

[KK+10‘] M. Kim, K. Kim, C. Luk
"SD3: A Scalable Approach to Dynamic Data-Dependence Profiling"
International Symposium on Microarchitecture MICRO (2010), 535-546

[KS+12] K. Molitorisz, J. Schimmel, F. Otto
"Automatic Parallelization Using AutoFutures"
International Conference on Multicore Software Engineering, Performance, and Tools MSEPT (2012) 78-81

[KR+94] J. Knoop, O. Rüthing, B. Steffen
"Partial dead code elimination"
Conference on Programming Language Design and Implementation PLDI (1994) 147-158

[L06] S. Lidin
"Expert .NET 2.0 IL Assembler"
Apress (2006) ISBN-978-1590596463

[M65] G. Moore
"Cramming more Components onto Integrated Circuits"
Electronics Magazine (1965) Vol. 38, Nr. 8, 114-117

[MB+11] P. Moret, W. Binder, E. Tander
"Polymorphic Bytecode Instrumentation"
International Conference on Aspect-Oriented Software Development (2011), 129-140

[MC+07] T. Moseley, D.A. Connors, D. Grunewald, R. Peri
"Identifying Potential Parallelism via Loop-centric Profling"
International Conference on Computing Frontiers CF (2007) 143-152

[MF+10] J. Mak, K.F. Faxén, S. Janson, A. Mycroft
"Estimating and Exploiting Potential Parallelism by Source-level Dependence Profling"
International Conference on Parallel Processing EuroPar (2010) 26-37

[MS+04] T. G. Mattson, B. A. Sanders, B. L. Massingill
"Patterns for Parallel Programming"
Addison-Wesley Professional (2004) ISBN-978-0321228116

[O13] F. Otto
"Objektorientierte Stromprogrammierung"
Dissertation, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2013)

[PA+11] V. Pankratius, A. Adl-Tabatabai, W. F. Tichy
"Fundamentals of Multicore Software Development"
CRC Press (2011) ISBN-978-1439812730

[PC] PowerCollections
[Online] http://powercollections.codeplex.com/

[R10] J. Richter
"CLR via C#"
Microsoft Press (2010) ISBN-978-0735627048

[Ros] Microsoft Roslyn
[Online] msdn.microsoft.com/en-us/vstudio/roslyn.aspx

[RV+10] S. Rul, H. Vandierendonck, K.D. Bosschere
"A profile-based tool for finding pipeline parallelism in sequential programs"
Parallel Computing (2010) Vol. 36, Nr. 9, 531-551

[S05] H. Sutter
"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software"
Dr. Dobb's Journal (2005) Vol. 30, Nr. 3, 16-23

[S10] C. Schäfer
"Automatisierte Performanzoptimierung Paralleler Architekturen"
Dissertation, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2010)

[SC+00] J. G. Steffan, C. B. Colohan, A. Zhai, T. C. Mowry
"A scalable approach to thread-level speculation"
International Symposium on Computer Architecture ISCA (2000) 1-12

[SM] SourceMonitor
[Online] http://www.campwoodsw.com/sourcemonitor.html

[SM+13] J. Schimmel, K. Molitorisz, A. Jannesari, W. F. Tichy
"Automatic Generation of Parallel Unit Tests"
Workshop on Automation of Software Test AST (2013)

[TC+07] W. Thies, V. Chandrasekhar, S. Amarasinghe
"A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs"
International Symposium on Microarchitecture (2007) 356-369

[TF10] G. Tournavitis, B. Franke
"Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profling Information"
International Conference on Parallel Architectures and Compilation Techniques PACT (2010) 377-388

[TM10] W. F. Tichy, D. J. Meder
"Parallelizing an Index Generator for Desktop Search"
Technischer Bericht, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2010) 2010-9

[TW+09] G. Tournavitis, Z. Wang, B. Franke
"Towards a Holistic Approach to Auto-Parallelization"
Conference on Programming Language Design and Implementation PLDI (2009) 177-187

[W11] A. Wilhelm
"Analyse des Parallelisierungspotentials sequenzieller Programme durch Kombination von statischer und dynamischer Analyse"
Bachelorarbeit, Universität Passau (2011)

[W13] S. Wagner
"Automatische Parallelisierung sequentieller Programme mittels Architekturbeschreibungen"
Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Karlsruher Institut für Technologie (2013) bisher unveröffentlicht

[WF08] M. S. Ware, C. J. Fox
"Securing Java code: heuristics and an evaluation of static analysis tools"
Workshop on Static Analysis SAW (2008) 12-21

ANHÄNGE

A. Abkürzungsverzeichnis

Abkürzung	Langbezeichnung und/oder Begriffserklärung
CCI	Common Compiler Infrastructure
.NET	Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen der Firma Microsoft.
RAW	Read-after-Write
TADL	Tuneable Architecture Desctiption Language
WAR	Write-after-Read
WAW	Write-after-Write

B. Abbildungsverzeichnis