Karlsruhe Institute of Technology

Objektorientierte Stromprogrammierung

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Frank Otto

aus Detmold

Tag der mündlichen Prüfung:	10. 01. 2013
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Michael Philippsen

Zusammenfassung

Multikernrechner machen Parallelverarbeitung zum Standard. Die meisten Programmiersprachen berücksichtigen Parallelität jedoch in einem nicht ausreichenden Maß – die Entwicklung und Optimierung paralleler Anwendungen gilt daher noch immer als schwierig und fehleranfällig.

Die vorliegende Arbeit entwickelt Konzepte, welche Stromprogrammierung in objektorientierten Sprachen ermöglichen. Stromprogrammierung erlaubt es, verschiedene Arten von Parallelität kompakt zu implementieren und zu optimieren, wobei von Fäden und expliziter Synchronisierung abstrahiert wird. Die meisten Stromsprachen sind jedoch auf die Domäne der Grafik- und Signalverarbeitung spezialisiert. Diese Arbeit verfolgt daher das Ziel, Stromprogrammierung in einer universellen Programmiersprache zugänglich zu machen und Optimierungen unabhängig vom Anwendungstyp durchführen zu können.

Zunächst wird ein Sprachentwurf vorgestellt, der die Verwendung von Filtern und Stromgraphen im objektorientierten Kontext erlaubt. Es werden Programmanalysen entwickelt, welche zustandsbehaftete Filter und kritische Abschnitte beim Übersetzen identifizieren und so zur Programmkorrektheit beitragen können. Des Weiteren wird eine Übersetzertechnik präsentiert, welche Stromprogramme in eine optimierbare Form überführt und sinnvoll vorkonfiguriert. Darauf aufbauend wird ein Verfahren für Laufzeit-Tuning konzipiert, welches die Leistung von Stromprogrammen im Produktivbetrieb automatisch misst und optimiert, ohne dass ein Zutun des Programmierers erforderlich ist.

Die Wirksamkeit der vorgestellten Konzepte wird anhand von Fallstudien belegt. Hierzu werden Anwendungen unterschiedlicher Art, Größe und Komplexität untersucht und dabei stromorientierte mit nicht stromorientierten Implementierungen verglichen. Die Ergebnisse zeigen, dass die objektorientierte Stromprogrammierung Codeumfang und Fehleranfälligkeit signifikant reduzieren kann. Laufzeit-Tuning erweist sich zudem als geeignete Methode, um die Leistung von Stromprogrammen unabhängig vom Anwendungstyp zu optimieren.

Abstract

With multicore chips, parallelism becomes mainstream. However, most programming languages address parallelism at an abstraction level that is too low – thus, implementing and optimizing parallel applications is still difficult and error-prone.

This thesis presents concepts enabling stream programming in object-oriented languages. Stream programming is capable of easily exploiting parallelism and performing optimizations while abstracting from threads and explicit synchronization. However, most stream languages are specialized on the graphics and signal processing domains. Therefore, our work aims for encompassing stream concepts in a fully-featured, general-purpose programming language, with optimization techniques that are independent of the type of the application.

First, we present a language design for using filters and stream graphs in an object-oriented context. We develop program analyses for detecting stateful filters and critical sections at compile time in order to support correctness. In addition, we present compiler techniques for translating stream programs into a tunable representation and making informed decisions on pre-configurations. On that basis, we develop a method for online-tuning stream programs. Performance measurement and optimization becomes available at run-time and without programmer interaction.

We evaluate our concepts with applications of different types, sizes, and complexities, comparing stream-based and non-stream-based implementations. Results show that object-oriented stream programming can significantly reduce code size and the risk of errors. In addition, online-tuning turns out a useful method for optimizing the performance of stream programs independent of the application type.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die vorliegende Dissertation stellt ein Programmiermodell vor, welches die Entwicklung und Optimierung von Software für Multikernrechner vereinfacht. Das Programmiermodell ermöglicht es, in objektorientierten Sprachen stromorientiert zu programmieren und Stromprogramme automatisch im Produktivbetrieb zu optimieren.

1.1 Motivation

Die Stagnierung von Prozessortaktfrequenzen hat einen fundamentalen Umbruch in der Softwaretechnik eingeleitet [99, 98]. Multikernprozessoren integrieren immer mehr Rechenkerne auf einem Chip. Für die Softwaretechnik bedeutet dies, dass Leistungssteigerungen nicht mehr automatisch durch höhere Taktraten der nächsten Prozessorgeneration erreicht werden; vielmehr muss performanzkritische Software aller Art nunmehr parallel programmiert werden, um die verfügbaren Rechenkerne möglichst gut auszunutzen.

Ein Schwerpunkt der Parallelverarbeitung lag bisher in den Bereichen des Hochleistungsrechnens und der Datenbanken. Inzwischen sind Multikernprozessoren in fast allen Notebooks und Arbeitsplatzrechnern zu finden. Parallele Programmierung wird somit zum Standard und stellt Entwickler vor neue Herausforderungen:

- Parallelität erhöht die Komplexität eines Programms. Mit der Komplexität sinkt die Codeverständlichkeit und steigt das Fehlerrisiko. Insbesondere Synchronisierungsfehler wie Wettläufe oder Verklemmungen sind schwer zu lokalisieren.

- Parallelität ist maschinenabhängig. Die Performanz eines parallelen Programms ist von Rechner zu Rechner unterschiedlich. Programme, die für einen bestimmten Rechnertyp entwickelt und optimiert wurden, zeigen möglicherweise ein schlechtes Laufzeitverhalten auf anderen Rechnertypen. Parallele Anwendungen müssen dahingehend konfigurierbar sein, dass sie schnell an neue Hardwarearchitekturen angepasst werden können.

1.2 Problemstellung

Die Programmiersprache ist die Schnittstelle zwischen Mensch und Maschine, mit der Verfahren zur Lösung eines Problems formal spezifiziert werden, so dass sie von einem Rechner ausgeführt werden können. Der Großteil der heute verbreiteten Programmiersprachen wurde ursprünglich für sequenzielles Programmieren konzipiert. Parallelverarbeitung wird meist durch die Verwendung von Fäden ermöglicht, wobei deren Erzeugung und Koordination durch den Programmierer erfolgen muss; dies ist bekanntermaßen ein nichttriviales und fehleranfälliges Unterfangen. Um korrekt zu parallelisieren, muss der Programmierer Datenabhängigkeiten sowie Codeabschnitte, welche auf gemeinsame Variablen zugreifen, kennen. Erschwerend kommt hinzu, dass der parallele Code für unterschiedliche Hardwarearchitekturen unterschiedlich konfiguriert bzw. optimiert werden muss, um gute Programmleistungen zu erzielen.

Es besteht somit eine dringende Notwendigkeit an parallelen Programmiersprachen bzw. Spracherweiterungen, welche die Implementierung und Optimierung paralleler Anwendungen vereinfachen. Hierbei stellen sich grundsätzliche Fragen:

- Welche Eigenschaften sollte eine Programmiersprache oder Spracherweiterung besitzen, um die Entwicklung von Software für Multikernsysteme zu vereinfachen?
- In welcher Form sollen Programmierer mit Aspekten der Parallelverarbeitung und deren Optimierung konfrontiert werden?
- Inwieweit können ein geeignetes Programmiermodell und Übersetzertechniken dazu beitragen, das Risiko von Synchronisierungsfehlern zu reduzieren?
- Welcher Abstraktionsgrad ist für eine Programmiersprache im Spannungsfeld zwischen maschinenspezifischen Faktoren und Portabilität sinnvoll?

Viele Problemstellungen der Softwaretechnik können durch Abstraktion und Automatisierung vereinfacht werden. Diese Prinzipien lassen sich auch auf parallele Programmiersprachen übertragen:

- Die Programmiersprache sollte von den Schwierigkeiten, die mit der Implementierung, Optimierung und Synchronisierung paralleler Applikationen verbunden sind, *abstrahieren*.
- Die Programmiersprache sollte geeignete Konzepte bereitstellen, die die Implementierung *automatisiert optimierbarer* Programme ermöglichen.

1.2.1 Stromorientierte Programmierung

Stromorientierte Programmierung stellt einen viel versprechenden Ansatz dar, um durch Abstraktion das Fehlerrisiko bei der parallelen Programmierung zu senken sowie das Problem der Performanzoptimierung zu automatisieren. Das stromorientierte Programmiermodell basiert auf drei grundlegenden Konzepten, Datenströmen, Filtern und Stromgraphen. Ein Datenstrom ist eine beliebig lange Folge von Datenelementen, die von Filtern verarbeitet werden. Filter sind Programmkomponenten, welche Eingabe- in Ausgabeströme umwandeln. Ein Stromgraph definiert die Verbindungen zwischen Filtern und beschreibt damit die Datenflüsse explizit.

Datenströme, Filter und Stromgraphen erlauben es, von Ausführungsfäden und expliziter Synchronisierung zu abstrahieren und somit den Entwickler zu entlasten. Parallelität wird erzeugt, indem Filter nebenläufig ausgeführt werden. Fließband-, Aufgaben- und Datenparallelität werden durch Stromgraphen explizit erfasst; Übersetzer können dieses Wissen zielgerichtet für Optimierungen verwenden. Die Übersetzung und Optimierung ist dabei meist auf spezielle Prozessortypen abgestimmt. Bisher wird stromorientierte Programmierung für konkrete Problemstellungen der Grafik- und Signalverarbeitung erfolgreich eingesetzt. Filter arbeiten hierbei meist mit primitiven Datentypen, dürfen keine Seiteneffekte verursachen und bieten keine Möglichkeit zur objektorientierten Programmierung.

Die Vorzüge der Stromprogrammierung sind jedoch auch für Problemstellungen abseits der Grafik- und Signalverarbeitung von Interesse. Viele Multikern-Anwendungen aus unterschiedlichen Bereichen nutzen Fließband-, Aufgaben- und Datenparallelität. Fallstudien zur Parallelisierung realistischer Applikationen [85, 84] unterstreichen die Notwendigkeit verschiedener Arten von Parallelität auf verschiedenen Ebenen in automatisiert optimierbarer Form. Für solche Anwendungen sind die Mächtigkeit und Flexibilität objektorientierter Konzepte oftmals essentiell. Somit ist es sinnvoll, die Vorzüge der Stromprogrammierung und der Objektorientierung zu kombinieren.

1.3 Ziel

Die vorliegende Arbeit stellt Konzepte vor, welche die stromorientierte Programmierung in objektorientierten Sprachen zugänglich machen und somit die Entwicklung korrekter, performanter Anwendungen für Multikernsysteme vereinfachen. Hieraus ergeben sich folgende Teilziele:

1. Es soll eine Spracherweiterung definiert werden, welche die Verwendung von Datenströmen, Filtern und Stromgraphen in objektorientierten Sprachen erlaubt und somit von Ausführungsfäden und expliziter Synchronisierung abstrahiert. Parallele Entwurfsmuster sollen damit kompakt implementiert werden können.

2. Zustandsbehaftete Filter und kritische Abschnitte sollen beim Übersetzen identifiziert und entsprechend behandelt werden. Die frühzeitige Identifikation soll dazu beitragen, dass Fehlerrisiko zu reduzieren.

3. Um von den Schwierigkeiten der Performanzoptimierung zu abstrahieren, soll eine Methodik entwickelt werden, mit der objektorientierte Stromprogramme in eine automatisch optimierbare Form überführt werden können. Hierbei gilt es, das in der stromorientierten Darstellung inhärente Wissen über die parallele Struktur eines Programms auszunutzen.

4. Es soll ein Verfahren entwickelt werden, welches die Leistung von Stromprogrammen automatisch bestimmt und optimiert. Dieses Verfahren soll während der Programmausführung zum Einsatz kommen, um Portabilität und Flexibilität erreichen.

Die Konzepte und Lösungsansätze, welche zum Erreichen der Teilziele beitragen, bilden das Gesamtkonzept der *objektorientierten Stromprogrammierung*.

1.4 Thesen

Entsprechend der Zielsetzung der Arbeit formulieren wir folgende Thesen:

- **These 1.** Objektorientierte Stromprogrammierung erlaubt es, diverse Arten von Parallelität für Multikernrechner kompakt zu implementieren. Eine Beschränkung auf die Domäne der Grafik- und Signalverarbeitung ist nicht notwendig. Zentrale parallele Entwurfsmuster können mithilfe weniger Sprachkonstrukte präzise ausgedrückt werden.

- **These 2.** Zustandsbehaftete Filter und kritische Abschnitte können mithilfe von Programmanalysen identifiziert werden.

- **These 3.** Es ist möglich, beim Übersetzen eines Stromprogramms bestimmte Klassen leistungsrelevanter Programmparameter und -informationen zu extrahieren. Programme können so in eine geeignete optimierbare Form überführt werden und deren Parameter sinnvoll vorkonfiguriert werden.

- **These 4.** Der Einsatz von Laufzeit-Tuning erlaubt es, die Performanz von Stromprogrammen im Produktivbetrieb zu optimieren.

Die Thesen werden in Kapitel 7 anhand von Fallstudien belegt.

1.5 Beitrag

Die Arbeit soll einen Beitrag auf dem Gebiet der Programmiermodelle für Multikernsysteme leisten. Es wird gezeigt, dass stromorientierte Programmierung im objektorientierten Kontext die Implementierung und Optimierung paralleler Anwendungen vereinfachen kann.

Ein Beitrag ist die Definition einer entsprechenden Spracherweiterung für objektorientierte Sprachen. Mit dieser Spracherweiterung ist es möglich, stromorientiert zu programmieren und zusätzlich die Vorzüge der Objektorientierung zu nutzen. In diesem Zusammenhang werden Programmanalysen für Übersetzer vorgestellt, um zustandsbehaftete Filter und gemeinsame Variablen frühzeitig in der Entwicklungsphase zu erkennen.

Ein weiterer Beitrag ist die Konzeption eines Verfahrens zur automatischen Leistungsbestimmung und -optimierung eines Programms im Produktivbetrieb. Automatische Performanzoptimierung (Auto-Tuning) wird somit zum integralen Bestandteil des Programmiermodells, ohne dass ein Einwirken des Programmierers notwendig ist. Das Verfahren ist dabei nicht speziell an unsere Sprache gebunden, sondern lässt sich allgemein auf Stromsprachen anwenden.

1.6 Aufbau der Arbeit

Dieses Kapitel beschrieb den Problemkontext, Ziele, Thesen und Beitrag der Arbeit. Kapitel 2 behandelt zunächst Begriffe und Konzepte, welche die Grundlage der entwickelten Lösungsansätze bilden, bevor verwandte Arbeiten in Kapitel 3 diskutiert werden. Die Lösungsansätze und Konzepte der vorliegenden Arbeit sind Gegenstand der Kapitel 4 und 5; eine Beschreibung ihrer prototypischen Implementierung folgt in Kapitel 6. Kapitel 7 evaluiert die Konzepte anhand von Fallstudien, diskutiert Ergebnisse und Erkenntnisse und stellt einen Bezug zu den Zielen und Thesen der Arbeit her. Kapitel 8 fasst die Arbeit schließlich zusammen und skizziert Forschungsfragen, die sich an diese Arbeit anschließen.

Kapitel 2

Grundlagen

In Kapitel 1 wurde das Ziel definiert, Stromverarbeitung und Objektorientierung zu einem Programmiermodell zu verbinden, welches die Entwicklung korrekter, performanter Anwendungen für Multikernsysteme vereinfacht. Dieses Kapitel legt die Grundlagen für die Lösungsansätze, die in den Kapiteln 4 und 5 vorgestellt werden.

- Abschnitt 2.1 thematisiert zentrale Begriffe und Modelle der Stromverarbeitung, welche für unseren Sprachentwurf von Bedeutung sind.
- Ein wichtiges Ziel des Sprachentwurfs ist es, die Implementierung von Parallelität zu vereinfachen. Zentrale Aspekte der parallelen Programmierung werden in Abschnitt 2.2 besprochen.
- Die vorliegende Arbeit verfolgt den Ansatz, Filterzustände und kritische Zugriffe auf gemeinsame Variablen mithilfe von Programmanalysen zu identifizieren. Die hierfür benötigten Grundbegriffe und Techniken sind Gegenstand von Abschnitt 2.3.
- Ein Beitrag der Arbeit liegt in der Konzeption von Methoden zur automatischen Leistungsbestimmung und -optimierung (Auto-Tuning) von Stromprogrammen. Abschnitt 2.4 behandelt die erforderlichen Grundlagen.

2.1 Stromverarbeitende Systeme

Der Sprachentwurf, der in Kapitel 4 vorgestellt wird, ermöglicht Entwicklern die Verwendung stromorientierter Programmierkonzepte in objektorientierten Sprachen. Dieser Abschnitt erläutert daher Grundbegriffe und Modelle stromverarbeitender Systeme, beschreibt deren Parallelisierungspotenzial und definiert das Parallelisierungsproblem für diesen Kontext. Wir verwenden und erweitern hierzu Definitionen und Formalisierungen von Stephens [97].

2.1.1 Grundbegriffe

Die grundlegende Datenstruktur stromverarbeitender Systeme ist der Datenstrom. Datenströme werden durch Filter verarbeitet. Filter können zu einem Stromgraphen verbunden werden. Miteinander verbundene Filter können Datenströme über Kanäle empfangen und versenden.

Definition 1 *Ein Datenstrom s entspricht einer Liste $(a_1, a_2, ...)$. Die Elemente a_i heißen Stromelemente. s ist vom Typ T, wenn alle Stromelemente a_i vom Typ T sind.*

Definition 2 *Ein Filter f transformiert eine Menge von m Eingabeströmen in eine Menge von n Ausgabeströmen. f lässt sich also beschreiben durch eine Verarbeitungsfunktion*

$$\phi : \mathbb{A}^m \to \mathbb{A}^n,$$

wobei $\mathbb{A}$ die Menge aller Stromelemente sei. $m, n \in \mathbb{N} \cup \{0\}$ heißen Eingangs- bzw. Ausgangsgrad von f. Im Falle $m = 0$ und $n > 0$ heißt f Quelle; im Falle $m > 0$ und $n = 0$ heißt f Senke.

Definition 3 *Jeder Filter f besitzt einen Eingabetyp T_{in} und einen Ausgabetyp T_{out}, welche die Typen der Eingabe- bzw. Ausgabeströme festlegen. Daraus folgt, dass (1) alle Stromelemente, die von f empfangen werden, den Typ T_{in} oder einen Subtyp besitzen müssen und (2) alle Stromelemente, die von f versendet werden, den Typ T_{out} oder einen Subtyp besitzen müssen.*

Definition 4 *Ein Stromgraph ist ein zusammenhängender, gerichteter Graph $G = (V, E)$, wobei V eine Menge von Filtern und $E \subset V \times V$ eine Menge von Verbindungen zwischen zwei Filtern ist. Eine Verbindung $(f_1, f_2) \in E$ entspricht einem Kanal, über den von f_1 produzierte Stromelemente an f_2 übergeben werden. Eine Verbindung (f_1, f_2) ist genau dann zulässig, wenn der Ausgabetyp $T_{out}^{(1)}$ von f_1 gleich dem Eingabetyp $T_{in}^{(2)}$ von f_2 ist oder $T_{out}^{(1)}$ ein Subtyp von $T_{in}^{(2)}$ ist. Der Kanaltyp für eine zulässige Verbindung (f_1, f_2) ist durch $T_{in}^{(2)}$ gegeben.*

Definition 5 *Ein Filter bzw. stromverarbeitendes System verhält sich deterministisch, wenn für denselben Eingabestrom stets derselbe Ausgabestrom produziert wird.*

2.1.2 Berechnungsmodelle

Das Verhalten stromverarbeitender Systeme lässt sich mithilfe von Berechnungsmodellen (engl. *models of computation*) beschreiben. Im Folgenden werden drei zentrale Modelle erläutert und voneinander abgegrenzt; viele heutige Stromsprachen basieren auf diesen Modellen. Anschließend skizzieren wir das Berechnungsmodell der vorliegenden Arbeit.

2.1.2.1 Prozessnetzwerke nach Kahn

In einem Prozessnetzwerk nach Kahn (Kahn Process Network, KPN) [51] entspricht jeder Filter einem in sich sequentiellen, deterministischem Prozess. Die Kanäle zur Verbindung von Filtern sind gepuffert und von beliebig großer Kapazität. Die Zugriffe auf einen Kanal, d.h. das Einfügen und Entnehmen eines Elements, können blockieren. Verklemmungen können somit nicht ausgeschlossen werden. Da jeder Filter deterministisch ist, gilt diese Eigenschaft auch für das gesamte Netzwerk. Ein prominentes Beispiel, welches das Konzept von Prozessnetzwerken implementiert, sind UNIX-Fließbänder.

2.1.2.2 Synchroner Datenfluss

Das Modell des synchronen Datenflusses (Synchronous Dataflow, SDF) [59] ist eine restriktive Form des Prozessnetzwerks nach Kahn. Filter arbeiten hierbei periodisch, d.h. sie definieren einen Verarbeitungsschritt, welcher x Elemente aus dem Eingabestrom entnimmt und y Elemente in den Ausgabestrom legt. Durch sukzessive Ausführung des Verarbeitungsschritts wird der gesamte Datenstrom verarbeitet. x und y heißen Datenraten; diese werden als konstant und bekannt vorausgesetzt. Mithilfe der Datenraten können bereits vor der Programmausführung feste Ablaufpläne berechnet werden, welche blockierende Zugriffe auf einen Kanal vermeiden und somit daraus resultierende Verklemmungen ausschließen. Die benötigten Kanalkapazitäten können ebenfalls vorhergesagt und entsprechend eingestellt werden. Die Sprache StreamIt [107] basiert auf dem Prinzip des synchronen Datenflusses.

2.1.2.3 Kommunizierende Sequentielle Prozesse

Das Modell der kommunizierenden sequentiellen Prozesse (Communicating Sequential Processes, CSP) [46] unterscheidet sich von Prozessnetzwerken bezüglich Kanaleigenschaften und Determinismus. Die Kanäle sind in diesem Modell ungepuffert; Stromelemente werden daher nach dem Rendezvous-Prinzip von Filter zu Filter übergeben. Kanalzugriffe erfolgen somit über blockierende Sende- und Empfangsoperationen. Des Weiteren können CSP-Systeme indeterministisch arbeiten: ein Filter kann Elemente von mehreren anderen Filtern in undefinierter Reihenfolge empfangen, sodass das Gesamtsystem bei wiederholter Ausführung mit gleicher Eingabe verschiedene Ausgaben produzieren kann. Ein Beispiel einer Programmiersprache, die auf dem Modell der kommunizierenden sequentiellen Prozessen basiert, ist Occam [74].

2.1.2.4 Berechnungsmodell der vorliegenden Arbeit

Die vorliegende Arbeit verwendet ein eigenes Berechnungsmodell, welches zwischen Prozessnetzwerken und synchronem Datenfluss einzuordnen ist. Wie im Modell des synchronen Datenflusses kann ein Filter einen Verarbeitungsschritt definieren, der periodisch für jedes Stromelement ausgeführt wird. Aufgrund der möglichen objektorientierten Dynamik (z.B. durch dynamisches Binden) können statische Datenraten jedoch nicht als bekannt vorausgesetzt werden. Somit werden Ablaufpläne und Kanalkapazitäten nicht statisch, sondern dynamisch ermittelt bzw. angepasst. Um Verklemmungen zu vermeiden, werden Filter, deren Kanalzugriff blockieren würde, vom Ablaufplaner pausiert und stattdessen andere wartende Filter ausgeführt. Die Stromverarbeitung ist somit nicht zwangsläufig deterministisch.

2.1.3 Definition des Parallelisierungsproblems

Jeder Filter stellt eine potenziell nebenläufige Aktivität dar. Das Parallelisierungsproblem der Stromverarbeitung besteht demnach darin, die Filter eines gegebenen Stromprogramms den vorhandenen Ressourcen, d.h. Prozessorkernen oder Ausführungsfäden, zuzuordnen. Gesucht ist eine Zuordnung, die die Leistung des Programms bezüglich eines bestimmten Leistungskriteriums optimiert. Ein häufig verwendetes Leistungskriterium ist die Ausführungszeit bzw. die Beschleunigung gegenüber der sequentiellen Ausführung. Je nach Anwendungskontext können jedoch auch andere Kriterien von Interesse sein, z.B. der Speicherverbrauch oder die Energieeffizienz in eingebetteten Systemen.

In diesem Zusammenhang kann es sinnvoll sein, den Stromgraphen zuvor aufzubereiten. Beispielsweise können Teile des Stromgraphen (Filtergruppen) zu einem Filter zusammengefasst oder filterinterne bzw. geschachtelte Parallelität berücksichtigt werden.

Die vorliegende Arbeit betrachtet das Parallelisierungsproblem als Optimierungsproblem. Ein Stromprogramm wird hierfür zunächst in eine parametrisierte, optimierbare Form überführt und mithilfe von Heuristiken vorkonfiguriert. Dieser Schritt entspricht der Aufbereitung bzw. Vorkonfiguration des Stromgraphen. Die Programmausführung, d.h. die Zuordnung von Filtern zu Ressourcen, geschieht mithilfe eines dynamischen Ablaufplaners sowie Methoden der suchbasierten Performanzoptimierung (Auto-Tuning) im Produktivbetrieb.

2.2 Parallelität

Stromprogrammierung ist eine Form der parallelen Programmierung. Dieser Abschnitt behandelt daher Grundbegriffe und -konzepte der Parallelverarbeitung, welche sowohl für die parallele Programmierung im Allgemeinen als auch für die Stromprogrammierung im Besonderen relevant sind. Thematisiert werden die verschiedenen Arten von Parallelität, das Erzeugen

von Parallelität, parallele Entwurfsmuster sowie die Problematik gemeinsamer Variablen. Die Betrachtung erfolgt aus der Perspektive von Multikernsystemen mit gemeinsamem Speicher. Allgemeinere Zusammenfassungen der Grundlagen und Konzepte der parallelen Programmierung haben beispielswiese Gleim [38] oder Rauber und Rünger [89] vorgelegt.

2.2.1 Arten von Parallelität

Parallelverarbeitung kann auf verschiedene Arten und auf unterschiedlichen Abstraktionsebenen erfolgen.

- **Befehlsebene.** Parallelität ist bereits auf der Ebene von Maschinenbefehlen möglich. Eine Folge von Befehlen ist genau dann parallel ausgeführbar, wenn das Ergebnis stets dasselbe ist wie bei sequenzieller Abarbeitung. Dies erfordert, dass die Befehle untereinander keine Datenabhängigkeiten aufweisen. Eine Datenabhängigkeit liegt vor, wenn zwei Befehle auf dasselbe Register zugreifen und mindestens ein Zugriff schreibt. Superskalare Prozessoren verwenden dynamische Befehlsablaufplaner in Hardware, um voneinander unabhängige Befehle auf verschiedene Funktionseinheiten zu verteilen. Darüber hinaus können Übersetzer Befehlsumordnungen durchführen, um ein möglichst hohes Maß an Parallelität auf Befehlsebene zu erlauben.

- **Daten- und Schleifenebene.** Datenparallelität lässt sich in Programmen ausnutzen, wenn dieselbe Operation auf verschiedene Elemente einer Datenstruktur angewendet wird. Dies kann beispielsweise in Schleifen der Fall sein. Parallelverarbeitung ist möglich, wenn die Elemente der Datenstruktur unabhängig voneinander bearbeitet werden können. Eine Reihe von Programmiersprachen bieten datenparallele Operatoren, z.B. parallele Schleifen oder Arrayoperationen.

- **Funktions- bzw. Aufgabenebene.** Parallelität auf Funktions- bzw. Aufgabenebene kann als die grobkörnigste Parallelitätsform angesehen werden. Ihre Bezugsgrößen sind ganze Funktionen, Berechnungen oder Programmteile. Der Einfachheit halber fassen wir diese Artefakte unter dem Begriff der Aufgabe zusammen. Einem ähnlichen Prinzip wie bei der Parallelität auf Befehlsebene folgend, kann Aufgabenparallelität dann ausgenutzt werden, wenn ein Programm mehrere voneinander unabhängige Aufgaben definiert, die auf verschiedene Prozessoren verteilt und parallel ausgeführt werden können. Aufgabengraphen sind eine häufig verwendete Struktur, um Abhängigkeiten zwischen Aufgaben zu erfassen und so einen Ablaufplan erstellen zu können.

- **Fließbandebene.** Softwareseitige Fließbandparallelität kann als Spezialform des Prinzips der Aufgabenparallelität aufgefasst werden. Fließbandparallelität lässt sich für eine Folge $(s_1, ..., s_n)$ von Verarbeitungsstufen (Aufgaben) ausnutzen, welche von Elementen $e_1, ..., e_m$ durchlaufen wird. Bezogen auf ein einzelnes Element e_i sind die Verarbeitungsstufen voneinander abhängig, für unterschiedliche Elemente sind die Stufen jedoch unabhängig. Während ein Element e_i von Stufe s_j verarbeitet wird, kann das nächste Element e_{i+1} bereits von der vorangehenden Stufe s_{j-1} verarbeitet werden.

Parallelität auf Befehlsebene kann zwar durch Übersetzer automatisch erzeugt werden, reicht aber nicht mehr aus, um die Performanz eines Programms signifikant zu steigern. Fallstudien mit realistischen, komplexeren Anwendungen haben gezeigt, dass Parallelität auf einer höheren Abstraktionsebene betrachtet werden muss, wobei oftmals die Ausschöpfung verschiedener Arten von Parallelität notwendig ist [84, 85]. Die Stromprogrammierung besitzt hierfür großes Potenzial, da sie die Implementierung von Aufgaben-, Daten- und Fließbandparallelität deutlich vereinfachen kann. Dieses Potenzial auch abseits feingranularer Algorithmen der Grafik- und Signalverarbeitung zu nutzen, ist ein Ziel der vorliegenden Arbeit.

2.2.2 Erzeugen von Parallelität

In einer parallelen Programmiersprache kann Parallelität explizit oder implizit repräsentiert sein bzw. erzeugt werden. Hier sei bereits angemerkt, dass die Trennlinie zwischen diesen beiden Möglichkeiten unscharf verläuft und auch Mischformen existieren. Der wesentliche Unterschied kann derart formuliert werden, dass explizite Parallelität vom Programmierer erzeugt und verwaltet wird, während implizite Parallelität erst durch den Übersetzer „entsteht“. Nach dieser Definition wird Parallelität in Stromsprachen implizit erzeugt, da Filter von Fäden und der damit verbundenen Synchronisierung abstrahieren.

2.2.2.1 Fäden

Fäden (engl. *threads*) kapseln nebenläufige Aufgaben und sind in vielen Programmiersprachen das zentrale Konzept zum Erzeugen von Parallelität. Ein Faden entspricht einem ausführbaren Instruktionsstrom und besitzt einen eigenen Befehlszeiger, Keller sowie Registerkopien. Jeder Faden ist einem Prozess zugeordnet, wobei ein Prozess mehrere Fäden „enthalten“ kann. Alle Fäden eines Prozesses teilen sich denselben Adressraum sowie dasselbe Code- und Datensegment.

Fäden werden durch entsprechende Anweisungen im Programmcode explizit erzeugt, gestartet und zeitlich aufeinander abgestimmt. Letztgenannter Aspekt wird mit dem Begriff Synchronisierung bezeichnet.

2.2.2.2 Ausführungsdienst

Ausführungsdienste (engl. *executor service*) stellen eine häufig verwendete Möglichkeit dar, Aufgaben und Fäden voneinander zu trennen. Ein Ausführungsdienst besteht aus einem Aufgabenvorrat (engl. *task pool*) und einer Menge von Ausführungsfäden. Statt für jede Aufgabe einen eigenen Faden zu erzeugen, werden erzeugte Aufgaben im Aufgabenvorrat abgelegt und nach und nach von den Ausführungsfäden entnommen und ausgeführt.

Ausführungsdienste eignen sich besonders für stark aufgabenparallele Programme, die aus vielen, voneinander unabhängigen Aufgaben bestehen. Aus Sicht des Programmierers wird von Fäden abstrahiert. Zudem ist die Anzahl der Ausführungsfäden von Beginn an begrenzt, da nicht für jede Aufgabe ein neuer Faden erzeugt wird.

2.2.3 Ablaufplanung

Ein Ablaufplaner (engl. *scheduler*) steuert die Zuordnung von Prozessen und Fäden zu Ressourcen. Ablaufplaner existieren nicht nur auf Betriebssystemebene, sondern können auch Teil einer Laufzeitumgebung oder eines separaten Programms sein. Ziel der Ablaufplanung ist es, die Ausführung der Aufgaben bezüglich bestimmter Kriterien zu optimieren. Häufig zu berücksichtigende Kriterien sind eine möglichst gute Auslastung der Ressourcen (Effizienz) oder Fairness, d.h. keine Aufgabe soll zu lang auf ihre Ausführung warten müssen.

Grundsätzlich zu unterscheiden sind statische und dynamische Ablaufplaner:

- Statische Ablaufplaner setzen voraus, dass Menge und Komplexität der Aufgaben vorab bekannt ist und erstellen mithilfe dieser Informationen einen Ablaufplan.
- Dynamische Ablaufplaner setzen das Wissen über Aufgabenmenge und -komplexität nicht voraus, d.h. der Ablaufplan muss zur Laufzeit erstellt und ggf. angepasst werden.

In Zusammenhang mit der Ablaufplanung ist der Begriff der *Lastverteilung* (engl. *load balancing*) zu nennen. Während die Ablaufplanung über zeitliche Zuordnungen entscheidet, bestimmt die Lastverteilungsstrategie die örtliche Verteilung von Aufgaben (Last) auf verfügbare Ressourcen. Lastverteilung kann, ähnlich wie die Ablaufplanung, auf unterschiedlichen Ebenen stattfinden. So muss z.B. entschieden werden, welcher Faden welchem Prozessor zugeordnet wird. Bei der Verwendung von Ausführungsdiensten müssen Aufgaben Fäden zugewiesen werden. In Stromprogrammen mit datenparallelen Abschnitten, beschrieben durch Filterinstanzen $f_1, ..., f_n$, muss entschieden werden, welche Filterinstanz f_i ein Stromelement verarbeiten soll. Analog zur Ablaufplanung lassen sich statische und dynamische Lastverteilungsstrategien unterscheiden, welche vorab oder zur Laufzeit die Aufgaben- bzw. Lastverteilung festlegen.

Eine wichtige Strategie zur Lastverteilung stellt *Workstealing* [18] dar. Workstealing ist ein dynamisches Verfahren, welches sowohl die Kommunikationskosten als auch die Lastverteilung optimiert. Eine Ressource kann von einer anderen Ressource Aufgaben „stehlen“ (engl. *steal*), sofern momentan keine eigenen Aufgaben ausgeführt werden können. Kommunikationskosten entstehen durch Datenaustausch zwischen verschiedenen Ressourcen, z.B. Prozessorkernen. Das Workstealing-Verfahren versucht daher, aus Kommunikationssicht „verwandte“ Aufgaben derselben Ressource zuzuordnen.

In der klassischen Stromprogrammierung werden zur Ausführung i.d.R. statische Ablaufpläne berechnet, welche die Filter eines Programms den verfügbaren Fäden bzw. Prozessorkernen zuordnen. Dabei werden Restriktionen in Stromsprachen gefordert und ausgenutzt, z.B. eine konstante Anzahl von Filtern und zuvor festgelegte Datenaustauschraten, durch die die Aufgabengrößen charakterisiert sind. Aufgrund der Dynamik der Objektorientierung wird in der vorliegenden Arbeit ein dynamischer Ablaufplaner eingesetzt.

2.2.4 Parallele Entwurfsmuster

Parallele Entwurfsmuster sind spezielle Formen von Entwurfsmustern (engl. *design patterns*) [37]. Sie sind ein geeignetes Mittel zur Beschreibung, Implementierung und Optimierung von parallelen Strukturen in Programmen. Sie bieten etablierte Lösungen für häufige Parallelisierungsprobleme und können somit die Produktivität und die Codeverständlichkeit erhöhen. Für die Anwendung von Optimierungstechniken sind parallele Muster besonders interessant, da sie Strukturwissen kapseln und implizit leistungsrelevante Parameter definieren. Dieses Wissen lässt sich bei der Optimierung ausnutzen.

Im Folgenden stellen wir zentrale parallele Muster bzw. Musterkategorien vor, die beim Entwurf und der Implementierung paralleler Anwendungen häufig zum Einsatz kommen und daher als besonders wichtig angesehen werden können. Zudem deckt die Auswahl die in Abschnitt 2.2.1 beschriebenen Arten von Parallelität ab. In Kapitel 4 wird gezeigt, wie sich diese Muster mithilfe von Konzepten der Stromprogrammierung implementieren lassen.

Mit dieser Auswahl ist kein Anspruch auf Vollständigkeit gegeben. Eine umfassendere Darstellung paralleler Entwurfs- und Implementierungsmuster findet sich beispielsweise in Mattson et. al. [66].

2.2.4.1 Fließband

Kontext. Eine Folge von Datenelementen wird in aufeinander folgenden Schritten verarbeitet, wobei das Ergebnis eines Schritts als Eingabe des nächsten Schritts fungiert.

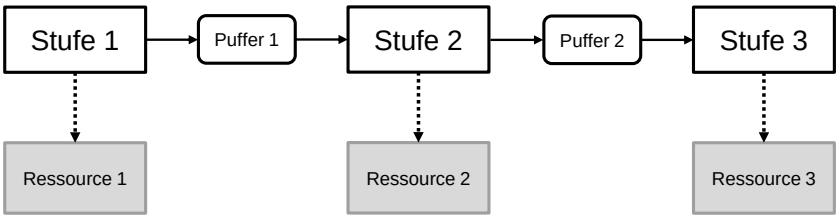


Abbildung 2.1: Fließbandmuster

Lösung. Für diese Verarbeitungsstruktur kann Fließbandparallelität ausgenutzt werden, indem jeder Verarbeitungsschritt als Fließbandstufe umgesetzt wird. Die Fließbandstufen sind durch Kanäle miteinander verbunden. Jede Fließbandstufe wird einer Ressource (Faden, Kern) zugeordnet. Während eine Stufe s_i ein Datum d_1 bearbeitet, kann die Vorgängerstufe s_{i-1} bereits Datum d_2 verarbeiten. Abbildung 2.1 veranschaulicht das Fließbandmuster.

Varianten. Ein lineares Fließband besitzt ausschließlich Stufen mit maximal einer direkten Vorgänger- und maximal einer direkten Nachfolgerstufe. Dagegen existieren in nichtlinearen Fließbändern Stufen mit mehr als einer direkten Vorgänger- oder Nachfolgerstufe. Fließbänder mit Rückkopplung können Stufen enthalten, die ihre Ergebnisse nicht nur an ihren direkten Nachfolger, sondern auch an eine Vorgängerstufe übergeben können.

2.2.4.2 Erzeuger-Verbraucher

Kontext. Eine Aufgabe A_1 produziert Daten, die von einer anderen Aufgabe A_2 bearbeitet werden.

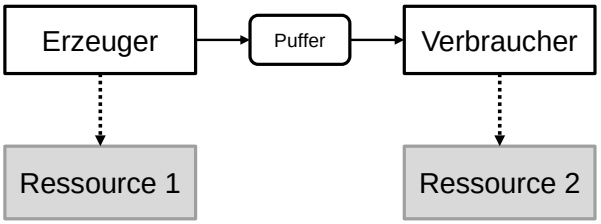


Abbildung 2.2: Erzeuger-Verbraucher-Muster

Lösung. Beide Aufgaben werden je einer Ressource zugewiesen. Der Datenaustausch von A_1 nach A_2 erfolgt über einen Puffer. Die Pufferkapazität kann begrenzt sein, um einen Überlauf

zu verhindern, wenn A_1 schneller produziert als A_2 konsumiert. Das Erzeuger-Verbraucher-Muster kann somit als Sonderfall des Fließbandmusters betrachtet werden. Abbildung 2.2 veranschaulicht das Erzeuger-Verbraucher-Muster.

Varianten. Es können mehrere Erzeuger- oder Verbraucher-Instanzen erzeugt werden, um die Verarbeitungsgeschwindigkeit zu erhöhen oder die Arbeitslast zu balancieren.

2.2.4.3 Auftraggeber-Auftragnehmer

Kontext. Eine Menge von Aufgaben, die im Voraus möglicherweise nicht bekannt sind, soll bearbeitet werden.

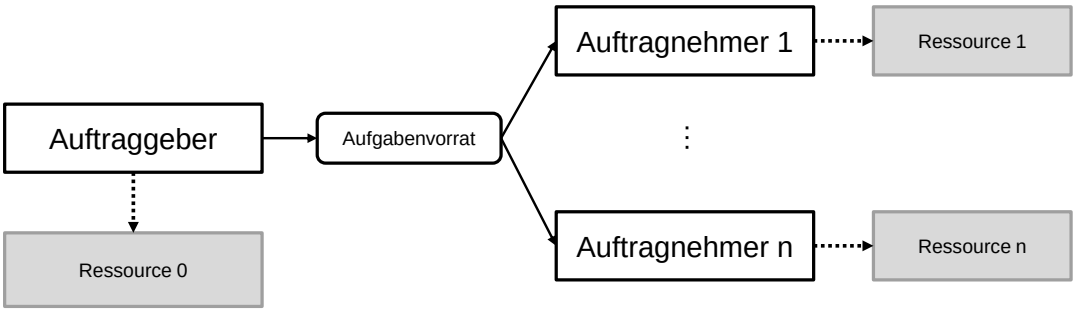


Abbildung 2.3: Auftraggeber-Auftragnehmer-Muster

Lösung. Ein Auftraggeber (engl. *master*) erzeugt Aufgaben (engl. *tasks*), die von Auftragnehmern (engl. *worker*) bearbeitet werden. Die Aufgaben werden in einem Aufgabenvorrat (engl. *task pool*) abgelegt und nach und nach von den Auftragnehmern abgeholt und bearbeitet. Der Auftraggeber wartet, bis alle Teilaufgaben bearbeitet sind, oder bearbeitet selbst einige Aufgaben. Sowohl der Auftraggeber als auch die Auftragnehmer werden je einer Ressource zugeordnet. Abbildung 2.3 veranschaulicht das Auftraggeber-Auftragnehmer-Muster.

Varianten. Der Auftraggeber kann, nachdem er alle Teilaufgaben erzeugt hat, selbst als Auftragnehmer fungieren und Teilaufgaben bearbeiten. Dies entspräche dem Fork-Join-Muster. Darüber hinaus können, ähnlich wie im Fließbandmuster, Rückkopplungen definiert werden. Eine weitere Variante ist das hierarchische oder rekursive Auftraggeber-Auftragnehmer-Muster. Hier können die Auftragnehmer wiederum als Auftraggeber fungieren, indem sie Teilaufgaben erneut aufteilen und an weitere Arbeitnehmer delegieren.

2.2.4.4 Teile und Herrsche

Kontext. Ein gegebenes Problem lässt sich rekursiv in kleinere, voneinander unabhängige Teilprobleme zerlegen. Die entstehenden Teillösungen werden anschließend zur Lösung des ursprünglichen Problems zusammengeführt.

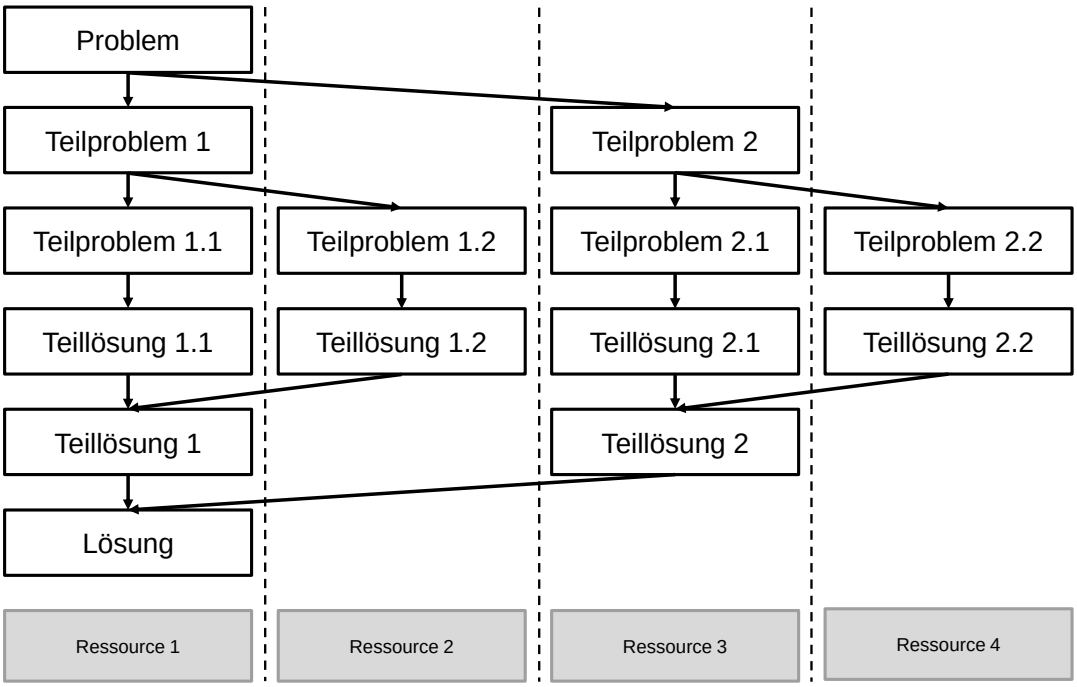


Abbildung 2.4: Teile-und-Herrsche-Muster

Lösung. Im sequentiellen Teile-und-Herrsche-Verfahren werden die Teilprobleme einer Rekursionsebene nacheinander gelöst. Im parallelen Fall werden Teilprobleme von verschiedenen Ressourcen gleichzeitig bearbeitet und deren Teillösungen zusammengeführt. Wird die Problemgröße ab einer bestimmten Rekursionsebene so „klein“, dass die Kosten der Parallelität deren Geschwindigkeitsvorteil dominieren, sollten die Teilprobleme sequentiell statt parallel gelöst werden. Abbildung 2.4 veranschaulicht das Teile-und-Herrsche-Muster.

Varianten. Das klassische Teile-und-Herrsche-Verfahren sieht vor, dass in einem Rekursionsschritt ein Problem in zwei Teilprobleme zerlegt wird. Es können jedoch auch mehrere Teilprobleme in einem Schritt erzeugt werden. Darüber hinaus ist das Prinzip von Teile-und-Herrsche verwandt mit dem der Datenzerlegung.

2.2.4.5 Datenzerlegung

Kontext. Eine große Datenmenge soll nach einer bestimmten Vorschrift verarbeitet werden.

Lösung. Die Datenmenge wird in Partitionen zerlegt. Die Partitionen werden verschiedenen Ressourcen zugeordnet. Abbildung 2.5 veranschaulicht das Prinzip der Datenzerlegung.

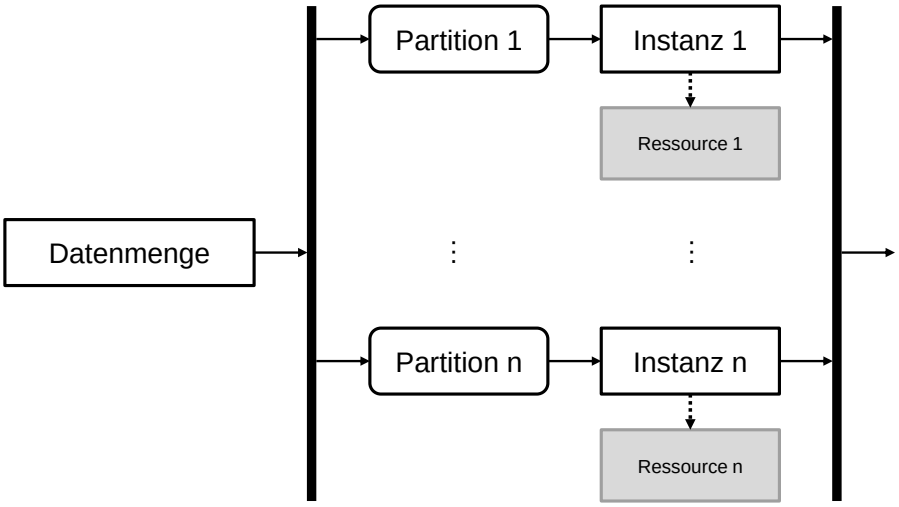


Abbildung 2.5: Prinzip der Datenzerlegung

Varianten. Das Datenzerlegungs-Muster ist prinzipiell ein Sammelbegriff für verschiedene datenparallele Muster. Diese unterscheiden sich im Wesentlichen durch die Art, nach der Daten zerlegt werden. Häufig verwendet werden iterative, rekursive und geometrische Zerlegungsstrategien. Beispielmuster aus diesen Kategorien sind nach Mattson et. al. [66] die iterative Reduktion, datenzentrische Teile-und-Herrsche-Verfahren oder das Wavefront-Muster. Der in Kapitel 3 besprochene MapReduce-Ansatz ist als iteratives Verfahren zu betrachten.

2.2.5 Gemeinsame Variablen und kritische Abschnitte

Gemeinsame Variablen werden, im Unterschied zu exklusiven Variablen, von verschiedenen Fäden geschrieben oder gelesen. Zugriffe auf gemeinsame Variablen müssen daher koordiniert bzw. synchronisiert werden, um Zugriffskonflikte zu vermeiden. Zugriffskonflikte bzw. daraus resultierende Wettlaufsituationen (engl. *race conditions*) können zu falschen Variablenbelegungen und somit ungültigen Programmzuständen führen.

Definition 6 *Ein Zugriffskonflikt liegt vor, wenn mindestens zwei Fäden zeitgleich auf dieselbe gemeinsame Variable zugreifen und mindestens ein Zugriff die Variable schreibt. Programmabschnitte, in denen Zugriffskonflikte bzw. Wettlaufsituationen verursacht werden können, heißen kritische Abschnitte.*

Um Zugriffskonflikte zu verhindern, muss sichergestellt werden, dass kritische Abschnitte atomar bzw. jederzeit von maximal einem Faden ausgeführt werden. Dies lässt sich mithilfe von Sperren, atomaren Anweisungen oder atomaren Abschnitten erreichen.

Klassische stromorientierte Modelle schließen die Verwendung gemeinsamer Variablen innerhalb von Filtern aus, um deren Unabhängigkeit und somit eine aggressive Parallelisierung zu erreichen. Diese Restriktion ist für einige spezielle Programme hinnehmbar, im allgemeinen Fall aber nicht sinnvoll bzw. durchsetzbar.

Das Programmiermodell dieser Arbeit erlaubt dagegen die Verwendung gemeinsamer Variablen. Dabei verfolgen wir den Ansatz, kritische Abschnitte mithilfe von Programmanalysen zu erkennen und in Form von Übersetzerwarnungen zu melden.

2.2.5.1 Sperren

Eine Sperre (engl. *lock*) ist ein Programmierkonstrukt, mit dem das Prinzip des gegenseitigen Ausschlusses erreicht wird. Eine Sperre s ist mit einer Ressource bzw. mit einem kritischen Abschnitt assoziiert. Ist ein kritischer Abschnitt noch nicht gesperrt, kann ein Faden t_1 ihn betreten, indem er zuvor die Sperre s erwirbt. Unmittelbar danach ist der Abschnitt gesperrt. Ein Faden t_2, der nun ebenfalls diesen Abschnitt ausführen will, muss warten, bis die Sperre s wieder freigegeben ist. Dies ist der Fall, wenn t_1 den kritischen Abschnitt verlässt. In der objektorientierten Programmierung werden Sperren mit Objekten verknüpft und entweder explizit (durch lock/unlock-Operationen) oder implizit (z.B. in Java mithilfe von `synchronized`-Blöcken) gesetzt bzw. freigegeben. Das Sperrkonzept lässt sich verfeinern durch die Differenzierung zwischen Lese- und Schreibsperren. In jedem Fall ist auf das korrekte Setzen und Freigeben von Sperren zu achten, da z.B. bei fehlender Sperrfreigabe nachfolgende Fäden „verhungern“ können oder bei der Verwendung mehrerer Sperren zyklische Warteabhängigkeiten zwischen Fäden und somit Verklemmungen (engl. *deadlocks*) auftreten können.

2.2.5.2 Atomare Anweisungen

Eine atomare Anweisung besteht aus einer Operation bzw. Operationsfolge, die nicht unterbrochen werden darf und somit stets als Ganzes auszuführen ist. Greift eine atomare Anweisung dabei auf gemeinsame Variablen zu, ist durch die Ununterbrechbarkeit gewährleistet, dass ungünstige Fadenverschränkungen und damit verbundene Wettlaufsituationen nicht auftreten können. Eine atomare Anweisung, die einen kritischen Abschnitt darstellt, muss somit nicht durch Sperren geschützt werden.

Atomare Anweisungen werden durch spezielle Maschineninstruktionen bereitgestellt. Beispiele sind atomare Lese-Schreib-Operationen, Testen und Schreiben (engl. *test-and-set*) oder Vergleichen und Tauschen (engl. *compare-and-swap*).

2.2.5.3 Atomare Abschnitte

Das Konzept atomarer Abschnitte abstrahiert von der Sperrproblematik, indem die Idee atomarer Anweisungen auf Programmabschnitte übertragen wird. Hier gibt der Programmierer lediglich durch ein entsprechendes Schlüsselwort (i.d.R. `atomic`) an, dass ein Abschnitt kritisch ist. Die Behandlung solcher Abschnitte erfolgt nach dem Vorbild von Datenbanktransaktionen und garantiert Atomizität, Konsistenz und Isolation. Atomare Abschnitte sind ein elementarer Aspekt im Kontext von softwareseitigem transaktionalem Speicher (engl. *software transactional memory*).

Es wird vorausgesetzt, dass ein atomarer Abschnitt nur reversible Seiteneffekte verursacht, also solche, die rückgängig gemacht werden können. Eine entsprechende Laufzeitumgebung erlaubt es nun mehreren Fäden, atomare Abschnitte nebenläufig auszuführen. Wird ein Konflikt zwischen einer laufenden Transaktion und einer anderen festgestellt, wird eine abgebrochen (d.h. die Seiteneffekte werden rückgängig gemacht) und deren Ausführung zu einem späteren Zeitpunkt wiederholt. Falls kein Konflikt auftritt, werden die Ergebnisse der Transaktion „sichtbar“ gemacht (engl. *commit*) und andere Transaktionen können damit weiterarbeiten.

Ein Nachteil atomarer Abschnitte liegt in dem Verwaltungsaufwand, der insbesondere für „lange“ Transaktionen mit vielen Seiteneffekten entsteht. Es ist die Aufgabe des Programmierers, atomare Abschnitte möglichst kostengünstig zu positionieren und zu gestalten.

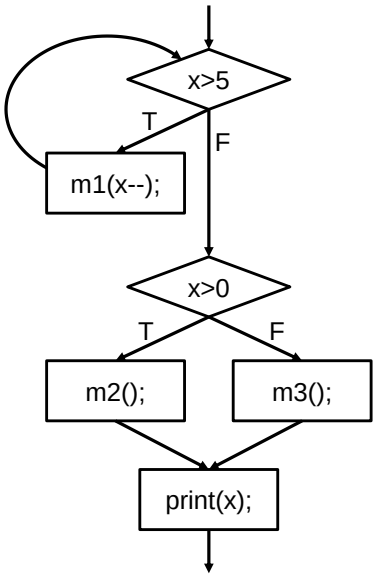
2.3 Programmanalyse

Dieser Abschnitt behandelt Grundbegriffe und Techniken der Programmanalyse. Programmanalysen erlauben es, Programme auf bestimmte Eigenschaften zu prüfen, ohne den Code ausführen zu müssen. Diese Verfahren können somit innerhalb eines Übersetzers oder auch als eigenständiges Werkzeug eingesetzt werden, um z.B. Code zu optimieren oder bestimmte Fehlertypen frühzeitig zu identifizieren. Basierend auf den hier besprochenen Grundlagen stellen wir in Kapitel 4 ein Konzept vor, welches objektorientierte Stromprogramme auf zustandsbehaftete Filter und kritische Abschnitte analysiert und somit zur Programmkorrektheit beitragen kann.

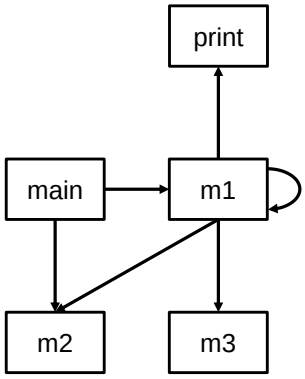
In dieser Arbeit gehen wir von „reinen“ objektorientierten Stromprogrammen aus: wir setzen voraus, dass Parallelität ausschließlich aus der Verwendung strombasierter Sprachkonstrukte, also Filtern und deren Kombination, resultiert. Nicht stromorientierte Parallelität wie explizite Fäden oder asynchrone Methodenaufrufe bzw. Futures schließen wir aus.

```
void main() {
  m1(5);
  m2();
}

void m1(int x) {
  while (x>5)
    m1(x--);
  if (x>0)
    m2();
  else
    m3();
  print(x);
}

void m2() { ... }
void m3() { ... }
```

(a) Beispielprogramm



(b) Kontrollflussgraph für $m1$



(c) Aufrufgraph des Programms

Abbildung 2.6: Kontrollflussgraph und Aufrufgraph

2.3.1 Kontrollfluss- und Aufrufgraph

Viele Programmanalysen basieren auf Kontrollfluss- und Aufrufgraphen. Ein Kontrollflussgraph (engl. *control flow graph*) ist eine Abstraktion der internen Ablaufstruktur einer Methode.

Definition 7 *Der Kontrollflussgraph einer Methode m ist ein gerichteter Graph $G = (V,E)$, wobei V die Menge der Anweisungen in m und $E \subset V \times V$ die Menge der Kontrollflusskanten beschreibt. Ein Kontrollflussgraph besitzt genau einen Wurzelknoten $s_{start} \in V$, der den Eintrittspunkt in die Methode darstellt. Eine Kontrollflusskante $(s_1, s_2) \in E$ besagt, dass die Anweisung s_2 unmittelbar nach s_1 ausgeführt wird.*

Ein Aufrufgraph (engl. *call graph*) beschreibt Abhängigkeiten zwischen den Methoden eines Programms.

Definition 8 *Der Aufrufgraph eines Programms $\mathscr{P}$ ist ein gerichteter Graph $G = (V,E)$, wobei V die Menge der Methoden in $\mathscr{P}$ und $E \subset V \times V$ die Menge der Aufrufkanten beschreibt. Eine Aufrufkante $(m_1, m_2) \in E$ besagt, dass Methode m_2 von m_1 aufgerufen wird.*

Zur Veranschaulichung zeigt Abbildung 2.6 Kontrollfluss- und Aufrufgraphen für ein kurzes Programmbeispiel.

2.3.2 Datenflussanalyse

Die Datenflussanalyse ist eine Technik, um Datenflüsse innerhalb einer Methode oder eines ganzen Programms zu beschreiben und damit Aussagen über den Code bzw. das Programm treffen zu können. Die Durchführung der Analyse erfolgt mithilfe der Kontrollfluss- und Aufrufgraphen.

Es lassen sich zwei Grundtypen von Datenflussanalysen unterscheiden. *Intraprozedurale Analysen* untersuchen die Anweisungen innerhalb einer bestimmten Methode und benötigen lediglich deren Kontrollflussgraphen. *Interprozedurale Analysen* arbeiten dagegen methodenübergreifend, indem sie Informationen des Aufrufgraphen verwenden.

Der Ablauf der Datenflussanalyse lässt sich in zwei aufeinander folgende Schritte unterteilen:

1. **Faktenextraktion.** Zunächst werden für die Anweisungen bzw. Knoten eines Kontrollflussgraphen lokale Fakten extrahiert, beispielsweise die Menge der an dieser Stelle erworbenen oder freigegebenen Sperren.

2. **Faktenpropagierung.** Die gewonnenen Fakten werden entsprechend dem Kontrollflussgraphen propagiert und kombiniert. Auf diese Weise lassen sich Eigenschaften einzelner Anweisungen, Programmabschnitte (z.B. Methoden) oder des gesamten Programms feststellen. Ein Beispiel wäre die Bestimmung derjenigen Sperren, die eine Anweisung schützen.

Um Fakten zu extrahieren und zu propagieren, verwendet die Datenflussanalyse folgende Mengen:

- $GEN(s)$ enthält diejenigen Fakten, die durch Anweisung s erzeugt werden.
- $KILL(s)$ enthält diejenigen Fakten, die durch Anweisung s verworfen werden.
- $IN(s)$ ist die Eintrittsmenge einer Anweisung s. Sie enthält diejenigen Fakten, die vor s gültig sind.
- $OUT(s)$ ist die Austrittsmenge einer Anweisung s. Sie enthält diejenigen Fakten, die nach s gültig sind.

Die Faktenextraktion erfolgt also mithilfe der Mengen $GEN(s)$ und $KILL(s)$, die Faktenpropagierung mithilfe der Mengen $IN(s)$ und $OUT(s)$. Je nach Problemstellung kann die Faktenpropagierung entlang oder entgegen der Kontrollflussrichtung geschehen.

Die Eintrittsmenge $IN(s)$ einer Anweisung s wird anhand der Ausgabemengen der unmittelbar vorangehenden Anweisungen gemäß

$$IN(s) := merge\{OUT(s') : (s', s) \in V_{CFG}\}$$

berechnet, wobei V_{CFG} die Kantenmenge des zugrundeliegenden Kontrollflussgraphen sei. Abhängig von der Problemstellung entspricht *merge* meist der Vereinigung oder dem Durchschnitt von Mengen. Die Austrittsmenge $OUT(s)$ einer Anweisung ergibt sich aus der Eintrittsmenge $IN(s)$ sowie den Faktenänderungen, beschrieben durch die Mengen $GEN(s)$ und $KILL(s)$. Die Berechnung erfolgt durch eine Propagierungsfunktion ψ_s, die im Allgemeinen wie folgt definiert ist:

$$OUT(s) = \psi_s(IN(s), GEN(s), KILL(s)) := (IN(s) \setminus KILL(s)) \cup GEN(s).$$

	Anweisung s_i	$GEN(s_i)$	$KILL(s_i)$	$IN(s_i)$	$OUT(s_i)$
	`int m() {`				
s_1	`int x = 0;`	$\{\}$	$\{\}$	$\{\}$	$\{\}$
s_2	`entermonitor o1;`	$\{o1\}$	$\{\}$	$\{\}$	$\{o1\}$
s_3	`x = x + o1.x;`	$\{\}$	$\{\}$	$\{o1\}$	$\{o1\}$
s_4	`entermonitor o2;`	$\{o2\}$	$\{\}$	$\{o1\}$	$\{o1,o2\}$
s_5	`x = x + o1.x + o2.x;`	$\{\}$	$\{\}$	$\{o1,o2\}$	$\{o1,o2\}$
s_6	`exitmonitor o2;`	$\{\}$	$\{o2\}$	$\{o1,o2\}$	$\{o1\}$
s_7	`exitmonitor o1;`	$\{\}$	$\{o1\}$	$\{o1\}$	$\{\}$
s_8	`return x;`	$\{\}$	$\{\}$	$\{\}$	$\{\}$
	`}`				

Abbildung 2.7: Datenflussanalyse für den Rumpf einer Methode m

Zur Illustration zeigt Abbildung 2.7 eine Methode m, für die mithilfe einer Datenflussanalyse berechnet wird, welche Anweisungen durch welche Sperren geschützt werden. Die dargestellte Datenflussanalyse entspricht einer vereinfachten Form der Sperrmengenanalyse (engl. *lockset analysis*), die in Kapitel 4 behandelt wird. Zunächst werden die Mengen GEN und $KILL$ für jede Anweisung s_i initialisiert. Wenn eine Sperre l durch eine Anweisung s_i erworben wird, ist $GEN(s_i) := \{l\}$; wird eine Sperre l durch eine Anweisung s_i freigegeben, gilt entsprechend $KILL(s_i) := \{l\}$. Werden keine Sperren durch s_i erworben oder freigegeben, ist $GEN(s_i) = KILL(s_i) = \{\}$. Die Berechnung der Mengen IN und OUT erfolgt nach dem zuvor erläuterten Schema. $OUT(s_i)$ enthält anschließend genau diejenigen Sperren, die die Anweisung s_i schützen. So kann z.B. die Aussage getroffen werden, dass Anweisung s_5 durch die Sperren $o1$ und $o2$ geschützt ist.

2.3.3 Zeigeranalyse

Die Zeigeranalyse dient dazu, Informationen über Zeiger bzw. Referenzen, die im Programmcode verwendet werden, zu gewinnen. Insbesondere bei der Analyse objektorientierter Programme können Zeigerinformationen daher sehr wertvoll sein.

Die Zeigeranalyse berechnet für eine Variable v eine Zielmenge $Z(v)$, die aus denjenigen Variablen bzw. abstrakten Speicherstellen (engl. *allocation nodes*) besteht, auf die v zur Laufzeit verweisen kann. So lassen sich beispielsweise Aussagen darüber treffen, ob zwei verschiedene Variablen auf dieselbe Speicherstelle zeigen können.

Die Durchführung von Zeigeranalysen kann sich in Abhängigkeit von Programmgröße und -komplexität sehr aufwändig gestalten. Es gilt daher, einen Mittelweg zwischen Aufwand und Genauigkeit der Analyse zu finden. Als wichtigste Parameter sind zu nennen:

- **Kontrollfluss.** Eine Zeigeranalyse ist *flusssensitiv*, wenn sie die Ausführungsreihenfolge der Anweisungen beachtet. Andernfalls ist sie *flussinsensitiv*.
- **Aufrufkontext.** Eine Zeigeranalyse ist *kontextsensitiv*, wenn sie die Aufrufkontexte von Methodenaufrufen berücksichtigt. Unterschiedliche Aufrufe derselben Methoden führen i.A. zu unterschiedlichen Zielmengen der darin verwendeten Variablen. Das bedeutet, dass eine mehrmals aufgerufene Methode mehrmals analysiert werden muss. Im Unterschied dazu wird in *kontextinsensitiven* Zeigeranalysen nicht zwischen den Aufrufkontexten differenziert.

Listing 2.1: Beispielprogramm

```
1  class C {
2    A a1 = new A(); A a2 = new A(); A a3 = new A();
3
4    void m1(A a) {
5      A x = a;
6      x = a3;
7    }
8
9    void m2() {
10     m1(a1);  // context c1
11     m1(a2);  // context c2
12   }
13 }
```

Abhängig von der Art der Analyse ergeben sich für das in Listing 2.1 dargestellte Beispielprogramm folgende Zielmengen für Variable x in Methode $m1$:

- Fluss- und kontextinsensitiv: $Z(x) = \{a1, a2, a3\}$
- Flusssensitiv, kontextinsensitiv: $Z(x) = \{a3\}$
- Flussinsensitiv, kontextsensitiv: $Z_{c1}(x) = \{a1, a3\}$ und $Z_{c2}(x) = \{a2, a3\}$ für die Kontexte $c1$ bzw. $c2$
- Fluss- und kontextsensitiv: $Z_{c1}(x) = Z_{c2}(x) = \{a3\}$

Die Ergebnisse der Zeigeranalyse lassen sich in Form eines *Zeigergraphen* darstellen. Dessen Knoten sind Programmvariablen oder abstrakte Speicherstellen, während die Verbindungen zwischen zwei Knoten als Zeiger aufgefasst werden.

2.4 Leistungsbestimmung und -optimierung

Ein Beitrag der vorliegenden Arbeit liegt in der Optimierung der Performanz von Stromprogrammen im Produktivbetrieb. Dieser Abschnitt fasst die wesentlichen Grundlagen der Leistungsbestimmung und -optimierung aus [92] zusammen, die für die Optimierung von Stromprogrammen relevant sind.

2.4.1 Grundbegriffe

Definition 9 *Ein Tuningparameter p ist eine Programmvariable, deren Wert die Leistung, nicht aber die Semantik eines Programms beeinflusst. p besitzt eine diskrete, endliche Wertemenge V_p sowie einen Standardwert $def(p) \in V_p$.*

Tuningparameter lassen sich bezüglich ihrer Wertemenge klassifizieren:

- *Ordinale Tuningparameter* besitzen eine Wertemenge V_p, für die eine Ordnung definiert ist. Beispiele für ordinale Tuningparameter sind die Anzahl der Fäden, die für eine bestimmte Berechnung verwendet werden, oder die Anzahl von Stufen in einem Fließband. Ein möglicher Wertebereich für beide Beispiele wäre $V_p := \{1, ..., 8\}$ mit der Ordnung $(V_p, <)$.

- Die Wertemenge *nominaler Tuningparameter* besteht dagegen aus einer Aufzählung von Parameterwerten, für die keine Ordnung definiert ist. Ein Beispiel wäre ein Parameter zur Auswahl einer bestimmten Implementierungsvariante mit einem Wertebereich $V_p := \{ImplA, ImplB, ImplC\}$.

- *Boolesche Tuningparameter* besitzen die Wertemenge $V_p = \{true, false\}$. Beispielsweise kann die Entscheidung, ob zwei benachbarte Fließbandstufen fusioniert werden sollen, durch einen booleschen Tuningparameter ausgedrückt werden.

Definition 10 *Ein Programm $\mathscr{P}$ besitze die Tuningparameter $p_1, ..., p_n$. Eine Parameterkonfiguration k für $\mathscr{P}$ ist ein n-Tupel*

$$k := (v_1, ..., v_n) \in V_{p_1} \times ... \times V_{p_n}.$$

Die Standardkonfiguration k_{def} ist definiert durch

$$k_{def} := (def(p_1), ..., def(p_n)).$$

Unmittelbar aus der vorangegangenen Definition ergibt sich der Begriff des Suchraums.

Definition 11 *Ein Programm $\mathscr{P}$ besitze die Tuningparameter $p_1, ..., p_n$. Der Suchraum $S_{\mathscr{P}}$, der durch die Parameter aufgespannt wird, ist definiert durch das kartesische Produkt der Wertemengen aller Parameter, also*

$$S_{\mathscr{P}} := V_{p_1} \times ... \times V_{p_n}.$$

Der Suchraum $S_{\mathscr{P}}$ für ein Programm $\mathscr{P}$ entspricht also der Menge aller Parameterkonfigurationen für $\mathscr{P}$. Die Größe des Suchraums ist gegeben durch

$$|S_{\mathscr{P}}| = \left|V_{p_1} \times ... \times V_{p_n}\right| = \left|V_{p_1}\right| \cdot ... \cdot \left|V_{p_n}\right|.$$

Definition 12 *Der Leistungswert ist eine Größe, welche die Leistung eines Programms $\mathscr{P}$ mit Suchraum $S_{\mathscr{P}}$ bezüglich einer bestimmten Parameterkonfiguration $k \in S_{\mathscr{P}}$ beschreibt. Der Leistungswert ergibt sich aus einer Leistungsfunktion*

$$\lambda : S_{\mathscr{P}} \to \mathbb{R}.$$

Die Leistungsfunktion bildet also eine Konfiguration $k \in S_{\mathscr{P}}$ auf einen Leistungswert $\lambda(k)$ ab, der sich auf ein konkretes Leistungskriterium bezieht. Mögliche Leistungskriterien sind z.B. Ausführungszeit, Speicherverbrauch oder Energieeffizienz. Es sei angemerkt, dass die Leistungsfunktion i.d.R. unbekannt ist. Die Anwendung rein mathematischer Optimierungsverfahren ist somit nicht möglich, was den Einsatz von suchbasierten Methoden rechtfertigt.

Zur Gewinnung von Leistungswerten werden Messpunkte benötigt. Ein Messpunkt ist eine Stelle im Programm, an der gemäß dem Leistungskriterium Daten erfasst werden, die zur Leistungsbestimmung dienen.

2.4.2 Optimierungsproblem

Mit den Definitionen des vorigen Abschnittes lässt sich nun das Optimierungsproblem formulieren. Ausgehend von einem Programm $\mathscr{P}$, dessen Parameter den Suchraum $S_{\mathscr{P}}$ aufspannen, gilt es, für die Leistungsfunktion λ diejenige Konfiguration $k_{opt} \in S_{\mathscr{P}}$ zu finden, so dass die folgende Bedingung gilt:

$$\lambda(k_{opt}) \leq \lambda(k) \forall k \in S_{\mathscr{P}}.$$

Wir bezeichnen k_{opt} als *beste* Konfiguration von $\mathscr{P}$. Entsprechend heißt eine Konfiguration k_1 *besser* als k_2, wenn $\lambda(k_1) < \lambda(k_2)$.

Bei dieser Definition haben wir o.B.d.A. angenommen, dass die Programmleistung umso besser ist, je geringer der Leistungswert ist. Dies ist z.B. der Fall für die Leistungskriterien Ausführungszeit oder Speicherverbrauch, an deren Minimierung man interessiert ist. Für Leistungskriterien wie Durchsatz lässt sich das Suchproblem analog als Maximierungsproblem definieren.

Zur Lösung des Optimierungsproblems können verschiedene Verfahrenstypen herangezogen werden. Die naheliegende Methode ist die *erschöpfende Suche*. Hierbei werden die Leistungswerte für alle Parameterkonfigurationen gemessen. Auf diese Weise erhält man die tatsächliche Optimalkonfiguration. Der Nachteil dieser Methode liegt in ihrem hohen Aufwand: da alle möglichen Konfigurationen betrachtet werden, ist dieses Vorgehen nur für verhältnismäßig kleine Suchräume, d.h. Programme mit wenigen Tuningparametern und kleinen Wertemengen, sinnvoll einsetzbar.

Statt der erschöpfenden Suche verwendet man daher Verfahren, die nur Teile des Suchraums betrachten, d.h. eine Teilmenge $\hat{S}_{\mathscr{P}} \subset S_{\mathscr{P}}$. So erhält man eine *beste bekannte* Konfiguration $\hat{k}_{opt} \in \hat{S}_{\mathscr{P}}$, nimmt dafür allerdings in Kauf, dass die beste gefundene Konfiguration nicht die tatsächlich beste ist; im Allgemeinen gilt also $\lambda(\hat{k}_{opt}) \geq \lambda(k_{opt})$. Beispiele für Verfahrenstypen, die nur Teile des Suchraums betrachten, sind folgende:

- **Zufallsprinzip.** Eine bestimmte Anzahl von Parameterkonfigurationen wird zufällig ausgewählt, deren Leistungswerte bestimmt werden. Dieses Verfahren entspricht einer zufallsbasierten Reduktion des Suchraums vor der eigentlichen Suche. Für den reduzierten Suchraum $\hat{S}_{\mathscr{P}} \subset S_{\mathscr{P}}$ wird eine erschöpfende Suche durchgeführt.
- **Lokale Suche.** Es wird eine Startkonfiguration k gewählt und deren Leistungswert $\lambda(k)$ bestimmt. Durch Variation eines oder mehrerer Parameter von k wird eine Parameterkonfiguration k' in einer Umgebung von k berechnet. Ist der $\lambda(k') < \lambda(k)$, so wird k' als neue Konfiguration festgelegt und in dessen Umgebung weitergesucht. Dieser Schritt wird wiederholt, bis eine Konfiguration $\hat{k}_{opt}$ gefunden wurde, in dessen Umgebung keine bessere existiert. Das Hauptproblem dieses Verfahrens liegt darin, dass es nur lokal konvergiert und seine Güte somit stark von der gewählten Startkonfiguration abhängt. Verfahren wie der Bergsteigeralgorithmus oder der Simplex-Algorithmus nach Nelder

und Mead [73] modifizieren das Verfahren, indem sie Gewichtungsfaktoren oder Toleranzwerte verwenden, um das Lokalitätsproblem zu entschärfen. Inbesondere der Algorithmus nach Nelder und Mead erweist sich i.d.R. als robust und wird daher in der vorliegenden Arbeit verwendet.

- **Globale Suche.** Globale Suchverfahren verwenden mehrere Startkonfigurationen und führen hierfür lokale Suchen durch. Die Startkonfigurationen können zufallsbasiert oder systematisch gewählt werden. Von Wichtigkeit sind die Abbruchkriterien der lokalen Suchen, um möglichst schnell diejenigen Teile des Suchraums zu identifizieren, die das beste Potenzial zeigen. Beispiele für globale Suchverfahren sind evolutionäre und genetische Algorithmen oder die Kombination von stichprobenbasierter Suchraumexploration und lokaler Suche [61].

Die beste Parameterkonfiguration ist abhängig von der Beschaffenheit der Eingabedaten sowie der zugrundeliegenden Rechnerarchitektur. Für unterschiedliche Programmeingaben bzw. Rechner können also unterschiedliche Optimalkonfigurationen gelten. Es ist also von besonderer Wichtigkeit, ein Programm konfigurierbar zu halten.

Des Weiteren ist die beste Parameterkonfiguration im Allgemeinen nicht über die Zeit konstant. Beispielsweise könnten in Anwendungen, die einer sich ändernden Arbeitslast ausgesetzt sind, je nach Zeitpunkt unterschiedliche Konfigurationen optimal sein. Solche Szenarien werden in dieser Arbeit nicht behandelt, bieten aber Raum für zukünftige Forschungsarbeiten.

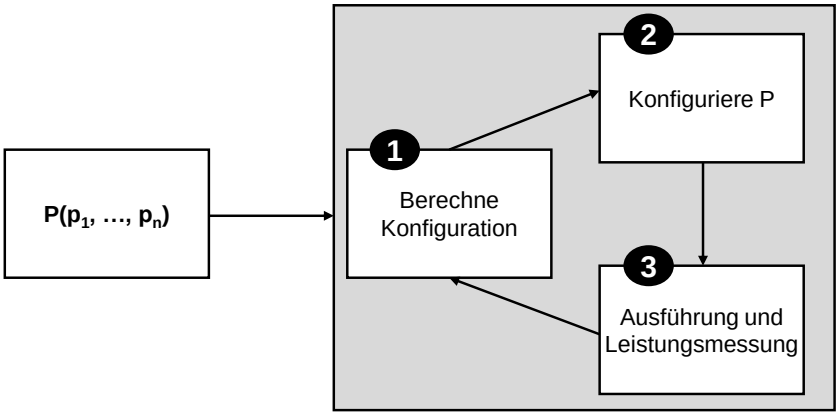


Abbildung 2.8: Aufbau eines Tuningzyklus

2.4.3 Auto-Tuning

Ein Auto-Tuner ist eine Komponente, die ein Programm $\mathscr{P}$, welches die Tuningparameter $p_1, ..., p_n$ besitzt, automatisch hinsichtlich eines Leistungskriteriums optimiert. Der Tuning-

prozess besteht aus einer Folge von Tuningzyklen. Der Aufbau eines Tuningzyklus ist in Abbildung 2.8 dargestellt. Es lassen sich folgende Phasen unterscheiden:

1. **Berechnung einer Parameterkonfiguration.** Dies lässt sich als Funktion

$$\tau : S_{\mathscr{P}} \times \mathbb{R} \to S_{\mathscr{P}}$$

   beschreiben, die für eine bereits getestete Konfiguration k und den dafür gemessenen Leistungswert $\lambda(k)$ eine neue Konfiguration $k_{neu} = \tau(k, \lambda(k))$ berechnet. Wurde bisher noch keine Konfiguration getestet, d.h. noch kein Tuningzyklus durchlaufen, entspricht k_{neu} der Startkonfiguration.

2. **Zuweisung der Parameterwerte.** Die Parameterwerte der aktuellen Konfiguration k_{neu} werden den Tuningparametern im Programm zugewiesen. Das Programm ist somit neu konfiguriert.

3. **Programmausführung und Leistungsbestimmung.** Das Programm wird als Ganzes oder teilweise mit der neuen Konfiguration k_{neu} ausgeführt und der Leistungswert $\lambda(k_{neu})$ bestimmt. Ist ein bestimmtes Abbruchkriterium erreicht, wird der Tuningprozess beendet. Andernfalls beginnt ein neuer Tuningzyklus, beginnend mit Phase 1.

Es lassen sich zwei Grundtypen des Auto-Tunings unterscheiden:

- Bei *Offline-Tuning* wird ein Programm mehrfach hintereinander ausgeführt, wobei die Leistung für eine gesamte Programmausführung bestimmt wird und die Parameter ausschließlich zwischen den Ausführungen angepasst werden. Bezogen auf den Entwicklungsprozess ist Offline-Tuning somit der Testphase zuzuordnen. Dieses Verfahren misst die Gesamtleistung des Programms und stellt keinerlei Anforderungen an die Programmstruktur. Allerdings ist für jede zu untersuchende Parameterkonfiguration eine vollständige Programmausführung erforderlich.

- *Online-Tuning* bzw. *Laufzeit-Tuning* passt die Parameterwerte während der Programmausführung, also im Produktivbetrieb an. Somit sind mehrere Programmausführungen nicht erforderlich. Voraussetzung ist allerdings eine spezielle Programmstruktur, in welcher bestimmte Programmabschnitte wiederholt durchlaufen werden. Für diese Abschnitte werden abhängig von der aktuellen Parameterkonfiguration Leistungswerte bestimmt. Auf diese Weise lassen sich neue Parameterkonfigurationen berechnen und deren Auswirkung auf die Programmleistung vergleichen.

Beide Grundtypen bieten Vor- und Nachteile. Da Offline-Tuning unabhängig von der Struktur eines Programms anwendbar ist, eignen sich auch stromorientierte Programme für diese Art von Optimierung. Die Testläufe, die im Rahmen des Offline-Tunings notwendig sind, können jedoch einen hohen Aufwand erzeugen und müssen im Allgemeinen für jeden Rechnertyp neu durchgeführt werden. Daher verfolgen wir das Ziel, Stromprogramme durch Laufzeit-Tuning zu optimieren und so ein höheres Maß an Flexibilität und Portabilität zu erreichen.

2.5 Zusammenfassung

Dieses Kapitel bildet das theoretische Fundament für die Lösungskonzepte der vorliegenden Arbeit. Zunächst wurden zentrale Begriffe und Modelle der Stromverarbeitung behandelt. Anschließend wurden Grundbegriffe und -konzepte der Parallelverarbeitung besprochen, welche sowohl für die parallele Programmierung im Allgemeinen als auch für die Stromprogrammierung im Besonderen relevant sind. Des Weiteren wurden Grundlagen der Programmanalyse, der Leistungsbestimmung und Leistungsoptimierung (Auto-Tuning) erläutert.

Die Lösungskonzepte der vorliegenden Arbeit sind Gegenstand der Kapitel 4 und 5. Zuvor werden in Kapitel 3 verwandte Arbeiten diskutiert.

Kapitel 3

Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt und von der vorliegenden Arbeit abgegrenzt. Wir diskutieren zunächst stromorientierte Programmieransätze (Abschnitt 3.1) sowie andere parallele Programmiermodelle (Abschnitt 3.2), ergänzt durch eine kurze Zusammenfassung von Ansätzen zur Identifikation von Wettlauffehlern (Abschnitt 3.3). Eine Besprechung von Arbeiten zur Optimierung paralleler Programme (Abschnitt 3.4) rundet das Kapitel ab.

3.1 Arbeiten zur stromorientierten Programmierung

Die vorliegende Arbeit entwickelt Konzepte, welche stromorientierte Programmierung in objektorientierten Sprachen ermöglichen. Gegenstand dieses Abschnitts sind daher verwandte Stromsprachen und Ansätze zur stromorientierten Programmierung.

3.1.1 Brook

Brook [19] ist eine stromorientierte Erweiterung der Sprache C. Sie ist konzipiert für die Entwicklung rechenintensiver, datenparalleler Anwendungen und ermöglicht die Ausnutzung von Grafikkarten als Coprozessor. Brook erlaubt lediglich die Definition von zustandslosen Filtern, sogenannten Kernels. Listing 3.1 zeigt ein einfaches Brook-Programm zur elementweisen Multiplikation zweier Datenströme.

Ein Datenstrom wird als typisierte Variable deklariert, für die eine Stromlänge spezifiziert werden kann. Im Beispiel in Zeile 7 werden die Ströme `x`, `y` und `result` vom Typ `float4` deklariert, die allesamt aus 100 Elementen bestehen. Nun werden aus den Arrays `X` und `Y` die Ströme `x` und `y` erzeugt (Zeilen 9 und 10). Diese Ströme werden vom Kernel `multiply` bearbeitet (Zeile 11), indem die Stromelemente paarweise multipliziert werden. Schließlich wird das Ergebnis, welches in Form des Stroms `result` vorliegt, in das entsprechende Array `Result` zurückgeschrieben (Zeile 12).

Listing 3.1: Brook-Programm

```
kernel void multiply(float4 x<>, float4 y<>, out float4 result<>) {
  result = x*y;
}

void main(void) {
  float4 X[100], Y[100], Result[100];
  float4 x<100>, y<100>, result<100>;
  ... // initialisiere X, Y
  streamRead(x, X);
  streamRead(y, Y);
  multiply(x, y, result);
  streamWrite(result, Result);
}
```

3.1.1.1 Abgrenzung

Im Unterschied zur vorliegenden Arbeit ist Brook für die stromorientierte Implementierung von Algorithmen und deren Ausführung auf Grafikkarten konzipiert. Der Fokus liegt auf der Ausnutzung von Datenparallelität; aus diesem Grund erlaubt Brook lediglich die Definition *zustandsloser* Filter. Eine weitere Restriktion von Brook besteht darin, dass innerhalb von Kernels keine beliebigen Funktionen aufgerufen bzw. Bibliotheken verwendet werden können. Zugriffe auf gemeinsame Variablen innerhalb von Kernels sind ebenfalls nicht zulässig. Optimierungstechniken werden nicht thematisiert.

Die vorliegende Arbeit ist dagegen weniger restriktiv. Stromverarbeitung ist hier im objektorientierten Kontext möglich und geschieht nicht nur auf datenparalleler, sondern auch fließband- und aufgabenparalleler Ebene. Zustandsbehaftete Filter sowie Zugriffe auf gemeinsame Variablen sind erlaubt und werden automatisiert identifiziert. Zudem besteht durch implizites Auto-Tuning die Möglichkeit, im Produktivbetrieb Optimierungen durchzuführen.

3.1.2 StreamIt

StreamIt [39, 107, 41, 106] ist eine stromorientierte Programmiersprache, die auf dem Konzept des synchronen Datenflusses (engl. *synchronous dataflow*, SDF) [58] aufbaut. Die Kommunikationsraten der Filter müssen explizit angegeben und konstant sein. Auf diese Weise können zur Übersetzungszeit statische Ablaufpläne berechnet und der Stromgraph auf eine konkrete Hardwarearchitektur abgebildet werden. Darüber hinaus werden Optimierungen wie die Fusion und Replikation von Filtern durchgeführt.

Ein Stromprogramm wird mithilfe von vier grundlegenden Sprachkonzepten beschrieben:

- *Filter* stellen das Grundkonzept von StreamIt dar. Ein Filter ist eine Programmkomponente, die einen Eingabe- und einen Ausgabekanal besitzt. Der Eingabekanal empfängt die Elemente des Eingabestroms. Der Filter transformiert jedes Element und schickt es an seinen Ausgabekanal.

- *Pipelines* (Fließbänder) bezeichnen die Verkettung von Filtern über deren Ein- und Ausgabekanäle.

- *Splitjoins* (Aufteiler-Zusammenführer) dienen dazu, einen Datenstrom aufzuteilen bzw. zusammenzuführen.

- *Feedback loops* (Rückkopplungsschleifen) ermöglichen die Definition von Zyklen im Stromgraphen.

Listing 3.2 zeigt ein StreamIt-Programm zur Berechnung des gleitenden Durchschnitts eines Datenstroms.

Listing 3.2: StreamIt-Programm

```
1  void->int filter IntSource {
2    int x;
3    init { x = 0; }
4    work push 1 {
5      push(x);
6      x++;
7    }
8  }
9
10 int->int filter Averager(int n) {
11   work pop 1 push 1 peek n {
12     int sum = 0;
13     for (int i = 0; i < n; i++)
14       sum += peek(i);
15     push(sum/n);
16     pop();
17   }
18 }
19
20 int->void filter IntPrinter {
21   work pop 1 { println(pop()); }
22 }
23
24 void->void pipeline MovingAverage {
25   add IntSource();
26   add Averager(10);
27   add IntPrinter();
28 }
```

Das Beispielprogramm besteht aus einem Fließband `MovingAverage` (Zeilen 24-28), das sich entsprechend der `add`-Anweisungen aus den Filtern `IntSource`, `Averager` und `IntPrinter` zusammensetzt. Filter, Fließbänder, Splitjoins sowie Rückkopplungsschleifen, besitzen einen Eingabe- und Ausgabetyp. Die Transformation eines Stromelements wird definiert durch den work-Block im Rumpf eines Filters (Zeilen 4, 11, 21). Die Datenraten werden mithilfe der Schlüsselwörter `pop`, `push` und `peek` festgelegt: beispielsweise gibt `pop 1 push 1 peek n` (Zeile 11) an, dass bei jedem Verarbeitungsschritt ein Element dem Eingabestrom entnommen wird, ein Element dem Ausgabestrom hinzugefügt wird und *n* Elemente des Eingabestroms gelesen werden, ohne entfernt zu werden.

3.1.2.1 Abgrenzung

StreamIt erlaubt beliebige Kombinationen und Verschachtelungen von Filtern und ermöglicht es somit, Aufgaben-, Daten- und Fließbandparallelität auf verschiedenen Abstraktionsebenen auszunutzen. Die kompakte Syntax dient daher als Inspiration für den Sprachentwurf der vorliegenden Arbeit.

Im Unterschied zu unserem Sprachentwurf handelt es sich bei StreamIt um eine reine Stromsprache ohne Unterstützung für objektorientierte Programmierung. StreamIt erlaubt zwar die Definition zustandsbehafteter Filter, aber keine Zugriffe auf gemeinsame bzw. globale Variablen. Die Eingabe- und Ausgabetypen der Filter sind primitive Typen, zusammengesetzte Typen (engl. *structs*) oder Arrays, während in der vorliegenden Arbeit beliebige Typen bzw. Objektreferenzen zulässig sind. Des Weiteren setzt StreamIt voraus, dass die Datenraten der Filter vom Programmierer spezifiziert sind, so dass der Übersetzer statische Ablaufpläne berechnen kann. Aufgrund der Ausführungsdynamik von objektorientiertem Code verwendet die vorliegende Arbeit einen dynamischen Ablaufplaner.

Während StreamIt statische Optimierungstechniken verwendet, die speziell auf die Architektur des Raw-Prozessors [104] abgestimmt sind, werden in der vorliegenden Arbeit Methoden für Laufzeit-Tuning konzipiert, die kein spezielles Architekturwissen voraussetzen und daher flexibler sind. Die Optimierungstechniken in StreamIt beziehen sich neben der Berechnung von Ablaufplänen im Wesentlichen auf die Fusion und Replikation von Filtern. Folgearbeiten befassen sich mit Übersetzeroptimierungen speziell für konfigurierbare Schaltkreise (engl. *field programmable gate arrays*, FPGAs) [48] und Multikernarchitekturen [56, 40] sowie adaptive Übersetzungstechniken für heterogene Architekturen [49]. Ein Verfahren zur Partitionierung von Stromgraphen durch maschinelles Lernen wird in [113] vorgestellt.

3.1.3 WaveScript

WaveScript [75] ist eine funktionale Sprache zur Programmierung von Stromanwendungen für verteilte Systeme. Ein WaveScript-Programm besteht aus Funktionen, die gruppiert und

bestimmten Rechnerknoten zugeordnet sind. Stromverarbeitung erfolgt nicht durch explizite Filter, sondern sogenannte Iteratoren. Iteratoren definieren die Umwandlung von Eingabe- in Ausgabeströme; syntaktisch handelt es sich dabei um schleifenartige Konstrukte, welche im Rumpf von Funktionen verwendet werden.

Listing 3.3: WaveScript-Iterator

```
1 iterate x in strm {
2   state { cnt = 0 }
3   cnt += 1;
4   emit x+cnt;
5   emit x-cnt;
6 }
```

Listing 3.3 zeigt einen Iterator, der in einem Verarbeitungsschritt ein Stromelement `x` liest und zwei Stromelemente mithilfe von `emit`-Anweisungen generiert. Ein Iterator kann einen Zustand besitzen; entsprechende Zustandsvariablen müssen durch das Schlüsselwort `state` explizit definiert werden.

Iteratoren können funktional kombiniert werden, um Datenströme zu verarbeiten. So kann die Ausgabe eines Iterators als Eingabe des nächsten fungieren. Darüber hinaus kann der Übersetzer Iteratoren replizieren und fusionieren. Ströme können mithilfe von speziellen Operatoren aufgeteilt und zusammengeführt werden. Der Übersetzer extrahiert aus den so definierten Stromgraphen Fließband-, Aufgaben- und Datenparallelität.

3.1.3.1 Abgrenzung

WaveScript ermöglicht die funktionale Programmierung von Stromanwendungen für verteilte Systeme, während die vorliegende Arbeit Ansätze zur objektorientierten Programmierung von Stromanwendungen für Multikernsysteme entwickelt. Gemeinsame Variablen sind in WaveScript nicht zulässig; Zustandsvariablen müssen explizit deklariert werden.

WaveScript ist für die Domäne der Signalverarbeitung konzipiert und umfasst speziell auf diesen Bereich zugeschnittene algebraische Optimierungen. Statische Stromgraphoptimierungen werden in Form der Replikation und Fusion von Iteratoren unterstützt und basieren auf zuvor gesammelten Laufzeitprofilen. Die vorliegende Arbeit optimiert dagegen Stromprogramme im Produktivbetrieb mithilfe von suchbasierten Methoden, um Flexibilität zu gewährleisten und nicht auf spezielle Problembereiche beschränkt zu sein.

3.1.4 StreamFlex

StreamFlex [96] ist ein Rahmenwerk für die Sprache Java zur stromorientierten Programmierung echtzeitfähiger Anwendungen. Klassen zur Implementierung von Filtern und Kanälen werden in Form einer Bibliothek bereitgestellt. Die Definition von Filtern und Stromgraphen erfolgt durch entsprechende Unterklassen der Bibliotheksklassen `Filter` bzw. `StreamFlexGraph`.

Listing 3.4: Dreistufiges Fließband in StreamFlex

```
1  class Gen extends Filter { ... }
2  class Mid extends Filter { ... }
3  class Sink extends Filter { ... }
4
5  class SimplePipeline extends StreamFlexGraph {
6    SimplePipeline(int period) {
7      Clock clk = makeClock(period);
8      Filter gen = makeFilter(Gen.class);
9      Filter mid = makeFilter(Mid.class);
10     Filter sink = makeFilter(Sink.class);
11     connect(clk, gen, "timer");
12     connect(gen, "out", mid, "in", 1);
13     connect(mid, "out", sink, "in", 1);
14     validate();
15   }
16 }
```

Listing 3.4 zeigt einen Codeausschnitt für ein lineares, dreistufiges Fließband. Die Filter sind durch die Klassen `Gen`, `Mid` und `Sink` definiert (Zeilen 1-3); das Fließband wird durch die Klasse `SimplePipeline` beschrieben (Zeilen 5-16). Die Kanäle zwischen den Filtern werden mithilfe der Methode `connect` erzeugt (Zeilen 11-13); die Methode `validate` prüft die Struktur des Stromgraphen. Ein `Clock`-Objekt (Zeile 7) dient als Taktgeber für den ersten Filter.

Die Echtzeitfähigkeit von StreamFlex-Programmen wird durch den Einsatz einer eigenen virtuellen Maschine (VM) erreicht. Stromorientierter Code wird in einem speziellen Faden ausgeführt, der nicht der herkömmlichen Speicherverwaltung und -bereinigung (engl. *garbage collection*) der Java VM unterliegt. Die Laufzeitbibliothek und VM von StreamFlex verwenden stattdessen Speicherbereinigungsmechanismen, welche zeitliche Verzögerungen vermeiden und so eine Programmausführung unter Echtzeitbedingungen ermöglichen. Um dies zu erreichen, dürfen Datenströme jedoch keine Objekte beliebiger Java-Klassen enthalten, da diese Klassen und deren Referenzen den Einsatz der herkömmlichen Speicherverwaltung erfordern könnten.

3.1.4.1 Abgrenzung

Sowohl StreamFlex als auch die vorliegende Arbeit erlauben Stromverarbeitung im objektorientierten Kontext. Im Unterschied zur vorliegenden Arbeit liegt der Fokus von StreamFlex auf echtzeitfähigen Stromanwendungen. Filter dürfen nur auf exklusive Variablen sowie eine stark eingeschränkte Menge von vordefinierten Klassen zugreifen, welche einer speziellen Speicherverwaltung unterliegen. Diese Anforderung lässt die Anwendungsdomäne von StreamFlex gegenüber der vorliegenden Arbeit erheblich schrumpfen.

Des Weiteren ist die Parallelverarbeitung bzw. Optimierung von Stromprogrammen nicht Gegenstand von StreamFlex; die Performanz der in [96] beschriebenen Beispielanwendungen wurden lediglich in sequenzieller Ausführung auf einem Prozessorkern untersucht. Hierfür wurden Beschleunigungen bis zu Faktor 4 gegenüber sequenziellen Java-Code gemessen. Ob und wie viel zusätzliche Leistung bei paralleler Ausführung bzw. auf Multikernprozessoren erreicht werden kann, wird nicht untersucht.

3.1.5 FastFlow

FastFlow [9, 8, 7] ist ein quelloffenes Rahmenwerk für die Sprache C++, welches algorithmische Skelette auf stromorientierte Strukturen abbildet. Das Rahmenwerk ist als Schichtenarchitektur konzipiert, die auf effizienten Datenstrukturen wie nichtblockierenden, sperrfreien Puffern und Synchronisationsmechanismen aufbaut. Die nächsthöhere Schicht stellt Konstrukte bereit, mit der sich allgemeine Stromgraphen erzeugen lassen.

Listing 3.5: Fließband in FastFlow

```
1 #include <ff/pipeline.hpp>
2 using namespace ff;
3
4 class Stage: public ff_node { ... }
5
6 int main() {
7   ff_pipeline pipe;
8   for(int i=0; i<n; ++i)
9     pipe.add_stage(new Stage);
10  if (pipe.run_and_wait_end()<0)
11    return -1;
12  return 0;
13 }
```

Die oberste Schicht stellt die eigentliche Programmierschnittstelle dar, welche die algorithmischen Skelette „Farm“ und „Pipeline“ zur Verfügung stellt. Hiermit können Aufgaben- und Fließbandparallelität formuliert und kombiniert werden. Filter werden in Form von virtuellen

Methoden implementiert, welche für jedes Stromelement aufgerufen werden. Zur Illustration zeigt Listing 3.5 das Codegerüst eines n-stufigen Fließbands. Die Filter sowie das Fließband werden mithilfe der vordefinierten Klassen `ff_node` bzw. `ff_pipeline` implementiert (Zeilen 4 bzw. 7-9); die Funktion zum Starten des Fließbands (Zeile 10) beinhaltet gleichzeitig eine implizite Barriere.

3.1.5.1 Abgrenzung

FastFlow teilt sich mit der vorliegenden Arbeit das Ziel, das Potenzial der Stromverarbeitung in objektorientierten Anwendungen auszunutzen. Während die vorliegende Arbeit Spracherweiterungen konzipiert, handelt es sich bei FastFlow um einen Bibliotheksansatz. Filterzustände und kritische Abschnitte müssen vom Programmierer erkannt werden. Die vorliegende Arbeit unterstützt den Programmierer dagegen durch Programmanalysen.

Mithilfe der algorithmischen Skelette der FastFlow-Bibliothek lassen sich Aufgaben- und Fließbandparallelität implementieren, jedoch nicht automatisiert optimieren. FastFlow beschränkt sich auf die Bereitstellung effizienter Kanäle. Im Unterschied zur vorliegenden Arbeit sind Stromgraphoptimierungen oder Auto-Tuning nicht Gegenstand des Ansatzes von FastFlow.

3.1.6 ACOTES

ACOTES („Advanced Compiler Technologies for Embedded Streaming“) [71] ist ein Ansatz, um sequenzielle Anwendungen automatisiert in eine stromorientierte Form zu überführen. Hierzu fügt der Programmierer in den Programmcode Pragma-Direktiven ein, welche parallelisierungsrelevante Informationen enthalten. So können beispielsweise Schleifenabschnitte, welche schleifenlokale Variablen lesen oder schreiben, entsprechend spezifiziert werden. Im Zentrum des Ansatzes steht ein Übersetzer, der ein annotiertes Programm in eine parallele, stromorientierte Form überführt und mithilfe von Modellen der Anwendungsdomäne und Zielplattform optimiert.

Zur Illustration zeigt Listing 3.6 annotierten C-Code. Die Schleife liest eine Folge von Zeichen (Zeile 4), wandelt Groß- in Kleinbuchstaben um (Zeile 6) und gibt diese aus. Diese beiden Arbeitsschritte sind per Pragma-Direktive als Tasks bzw. Filter gekennzeichnet, zwischen denen eine Eingabe-Ausgabe-Abhängigkeit bezüglich der Variable `c` existiert. Der ACOTES-Übersetzer kann mithilfe dieser Informationen ein Fließband erzeugen.

Listing 3.6: C-Code mit ACOTES-Annotationen

```
1  int main() {
2    char c;
3    #pragma acotes taskgroup
4    while (fread(&c, sizeof(c), 1, stdin)) {
5      #pragma acotes task input(c) output(c)    // filter 1
6      if ('A'<=c && c<='Z') c = c-'A'+'a';
7      #pragma acotes task input(c)              // filter 2
8      fwrite(&c, sizeof(c), 1, stdout);
9    }
10   return 0;
11 }
```

3.1.6.1 Abgrenzung

Im Unterschied zur vorliegenden Arbeit handelt es sich bei ACOTES weniger um einen Ansatz zur stromorientierten Programmierung, sondern um ein Verfahren zur automatisierten Parallelisierung mithilfe des Strommodells. Der Fokus liegt hierbei auf eingebetteten Systemen; als Zielgrößen für den Übersetzer stehen daher der Energie- oder Speicherverbrauch im Mittelpunkt. Darüber hinaus bietet ACOTES ein geringeres Abstraktionsniveau, da der Programmierer Zustände und gemeinsame Variablen explizit berücksichtigen und behandeln muss.

Der ACOTES-Übersetzer setzt zudem voraus, dass sowohl für die Anwendungsdomäne als auch die Zielplattform detaillierte Modelle vorliegen, um das generierte Stromprogramm effizient auf Hardware abbilden zu können. Insofern kann das Verfahren als modellbasiertes Offline-Tuning bezeichnet werden. Die Erstellung der Modelle ist jedoch im Allgemeinen schwierig und aufwändig. ACOTES stellt somit im Unterschied zu unserer Arbeit einen stark spezialisierten und weniger flexiblen Ansatz dar.

3.1.7 Lime

Lime („Liquid Metal“) [50, 15] ist eine Sprache zur parallelen Programmierung von Anwendungen für heterogene Architekturen, insbesondere Grafikkarten (GPUs) und rekonfigurierbare Schaltkreise (engl. *field programmable gate arrays*, FPGAs). Die Sprache kombiniert Aspekte der imperativen, objektorientierten, stromorientierten, funktionalen und hardwarenahen Programmierung.

Stromverarbeitung wird mithilfe von zwei speziellen Operatoren, `task` und `=>`, erreicht. Der `task` Operator konvertiert eine Methode in eine stromverarbeitende Komponente, die vergleichbar mit einem Filter ist. Mithilfe des `=>` Operators lassen sich solche Komponenten zu

Listing 3.7: Fließband in Lime

```
1 task Decode().zigzag
2 => task Coefficient().dcDecode
3 => task Quantization(color).deQuantize
4 => task Transforms.iDCT
5 => task Decode().center(128, block);
```

Stromgraphen verbinden. Listing 3.7 zeigt beispielhaft ein fünfstufiges Fließband zur Dekodierung eines Farbkanals.

3.1.7.1 Abgrenzung

Sowohl Lime als auch die vorliegende Arbeit verfolgen den Ansatz, Stromverarbeitung und Objektorientierung zu kombinieren. Während die vorliegende Arbeit Spracherweiterungen für bestehende objektorientierte Sprachen definiert und auch auf die Ausnutzung grobgranularer Parallelität abzielt, handelt es sich bei Lime um eine eigenständige, tlw. sehr hardwarenahe Sprache, in deren Fokus eher feingranulare Parallelität liegt. In Lime bestehen Datenströme aus Werttypen, während in der vorliegenden Arbeit auch beliebige Objekte bzw. Objektreferenzen zulässig sind. Filter, die auf gemeinsame Variablen zugreifen, sind in Lime nicht zulässig.

Ein weiterer zentraler Unterschied zwischen Lime und der vorliegenden Arbeit liegt in ihrer technischen Ausrichtung. Lime ist für die einheitliche Programmierung heterogener Architekturen, Grafikkarten und FPGAs konzipiert. Ein spezielles Laufzeitsystem sorgt dafür, die Programmteile diesen verschiedenen Ressourcen dynamisch zuzuteilen. Somit kann man von dynamischer, ressourcenspezifischer Ablaufplanung für heterogene Systeme sprechen, nicht jedoch von Auto-Tuning.

3.1.8 Weitere Arbeiten

Neben den zuvor beschriebenen Arbeiten existieren weitere Ansätze zur stromorientierten Programmierung, die im Folgenden kurz zusammengefasst werden. Für eine ausführliche Übersicht über ältere Stromsprachen wie Esterel, Signal, Lucid oder Lustre sei auf [97] verweisen.

Cg [65] ist, ähnlich wie Brook, eine stromorientierte Sprache, deren Ursprung in der Programmierung von Grafikkarten liegt. Cg bietet Sprachunterstützung für Fließband- und Datenparallelität, jedoch nicht für Aufgabenparallelität. Die Programmierung erfolgt sehr hardwarenah; es müssen bei der Implementierung eines Algorithmus die Verarbeitungsstufen von GPUs explizit berücksichtigt und angesprochen werden.

sH [69, 68] stellt einen makrobasierten Programmieransatz für C++ dar, der ebenfalls zur Grafikverarbeitung konzipiert ist. Der Fokus liegt auf Fließbandparallelität; Optimierungen werden in Form von dynamischer Stufenfusion unterstützt. Beliebige Stromgraphen lassen sich mit sH jedoch nicht implementieren. Wie in den meisten anderen Stromsprachen werden zustandsbehaftete Filter nicht unterstützt.

Der Schwerpunkt von Accelerator [103] liegt auf der Ausnutzung von GPUs durch C#-Programme. Im Fokus stehen dabei Berechnungen auf speziellen Arraytypen, für welche datenparallele Optimierungen dynamisch durchgeführt werden. Explizite Filter und deren Verknüpfung zu Stromgraphen sind nicht Bestandteil dieses Programmieransatzes.

Spidle [30] ist ein Ansatz, der speziell für den Einsatz in eingebetteten Systemen konzipiert ist. Ein Spidle-Programm besteht ausschließlich aus Filtern und Verbindern; innerhalb von Filtern kann auf externen C-Programmcode verwiesen werden.

3.1.9 Zusammenfassung

Tabelle 3.1 fasst die Merkmale der Arbeiten zur stromorientierten Programmierung zusammen:

- Die *Umsetzung* gibt an, auf welche Art stromorientierte Programmierung unterstützt wird (Sp: Sprache, SpE: Spracherweiterung, Bib: Bibliothek, API: Programmierschnittstelle, VM: Virtuelle Maschine, ÜD: Übersetzerdirektiven).
- Ein Ansatz ist *domänenspezifisch*, wenn die Übersetzungs- und Optimierungstechniken speziell auf bestimmte Algorithmentypen, Anwendungsbereiche oder Hardware (z.B. GPUs oder FPGAs) zugeschnitten ist.
- Das Merkmal *Objektorientierung* ist gegeben, wenn die Verwendung objektorientierter Konzepte bzw. Polymorphie unterstützt wird.
- Ist die Definition *zustandsbehafteter Filter* zulässig, so erfolgt deren Identifikation entweder manuell (M) durch den Programmierer oder automatisch (A).
- Sind Zugriffe auf gemeinsame Variablen durch Filter zulässig, so erfolgt die Identifikation *kritischer Abschnitte* entweder manuell (M) durch Programmierer oder automatisch (A).
- Die *Ablaufplanung* kann statisch (S) oder dynamisch (D) erfolgen.
- Das Merkmal *Laufzeit-Tuning* ist gegeben, wenn Auto-Tuning (Leistungsmessung und -optimierung) im Produktivbetrieb möglich ist.

Merkmal	Brook	StreamIt	WaveScript	StreamFlex	FastFlow	ACOTES	Lime	diese Arbeit
Umsetzung	SpE	Sp	Sp	Bib/VM	Bib	ÜD	Sp	SpE
Domänenspezifisch	X	X	X	X	-	X	X	-
Objektorientierung	-	-	-	X	X	-	X	X
Zustandsbehaftete Filter	-	A	M	M	M	M	M	A
Kritische Abschnitte	-	-	-	-	M	M	-	A
Ablaufplanung	S	S	S	S	D	S	D	D
Laufzeit-Tuning	-	-	-	-	-	-	-	X

Tabelle 3.1: Vergleich der Arbeiten zur stromorientierten Programmierung

Es kann festgehalten werden, dass die meisten Arbeiten zur stromorientierten Programmierung und deren Optimierungstechniken auf spezielle Anwendungsgebiete oder Hardware zugeschnitten sind. Viele dieser Ansätze sind restriktiv, indem sie Filterzustände oder Zugriffe auf gemeinsamen Variablen ausschließen. Laufzeit-Tuning kommt in keiner der verwandten Arbeiten zum Einsatz und kann daher als ein Alleinstellungsmerkmal der vorliegenden Arbeit gelten.

FastFlow befindet sich bezüglich Zielsetzung und Grundidee am nächsten an der vorliegenden Arbeit. Der Schwerpunkt liegt auf effizient implementierten Datenstrukturen und der Abbildung von algorithmischen Skeletten auf Stromgraphen. Im Unterschied zur vorliegenden Arbeit beinhaltet FastFlow weder übersetzergesteuerte Zustands- und Konfliktanalysen noch Konzepte zur automatisierten Performanzoptimierung.

3.2 Arbeiten zu anderen parallelen Programmiermodellen

In diesem Abschnitt werden Arbeiten zu parallelen Programmiermodellen besprochen, welche nicht in den Bereich der Stromprogrammierung fallen und sich bereits in diesem Punkt von der vorliegenden Arbeit unterscheiden. Sie sind jedoch insofern als relevant einzustufen, als sie sich mit Sprach-, Ausführungs- oder Optimierungskonzepten für spezielle Parallelitäts- bzw. Programmarten befassen.

3.2.1 Cilk

Cilk/Cilk++ [17, 87, 36] ist eine Spracherweiterung für C/C++, welche für die effiziente Ausnutzung von Aufgabenparallelität konzipiert ist. Cilk stellt mit `spawn` und `sync` zwei Schlüsselwörter zum Erzeugen und Synchronisieren nebenläufiger Aufgaben bereit.

Listing 3.8: Cilk-Programm

```
1  cilk int fib(int n) {
2    if (n > 2) {
3      return n;
4    }
5    int x, y;
6    x = spawn fib(n-1);
7    y = spawn fib(n-2);
8    sync;
9    return (x + y);
10 }
```

Listing 3.8 illustriert das Konzept anhand einer Funktion zur parallelen Berechnung der n-ten Fibonaccizahl. Durch das Schlüsselwort `cilk` wird die Funktion `fib` als parallel ausführbare Aufgabe deklariert (Zeile 1). Der Rekursionsschritt in den Zeilen 6 und 7 wird durch das Voranstellen des Schlüsselworts `spawn` vor die entsprechenden Funktionsaufrufe parallelisiert. Die `sync`-Anweisung in Zeile 8 erzeugt schließlich eine Barriere, welche auf die Fertigstellung der zuvor mittels `spawn` erzeugten nebenläufigen Aufgaben wartet.

Einen wichtigen Beitrag von Cilk stellt der Workstealing-Algorithmus zur Ausführung paralleler Aufgaben dar. Dieser Algorithmus garantiert im aufgabenparallelen Kontext eine effiziente Ablaufplanung, welche beweisbar lediglich um einen kleinen, konstanten Faktor vom Optimum abweicht [18]. Hierfür werden Aufgaben auf mehrere Ausführungsfäden verteilt, wobei ein Faden ohne Arbeit Aufgaben von anderen Fäden „stehlen“ kann. Der Cilk-Ablaufplaner erzeugt standardmäßig für jeden Rechenkern einen Ausführungsfaden. Die Prinzipien der Cilk-Funktionen und des Workstealing-Algorithmus kommen in zahlreichen anderen Arbeiten bzw. Programmiersprachen zum Einsatz. Beispielsweise existiert mit dem Fork-Join-Rahmenwerk [57] eine an Cilk angelehnte Implementierung für Java.

3.2.1.1 Abgrenzung

Im Unterschied zur vorliegenden Arbeit liegt der Sprachfokus von Cilk auf Aufgabenparallelität; für Fließband- und Datenparallelität bzw. Stromverarbeitung existiert keine explizite Sprachunterstützung. Insbesondere die Implementierung von Fließbandparallelität wäre in Cilk deutlich aufwändiger, da beispielsweise Kanäle, Aufteiler und Zusammenführer explizit programmiert werden müssten. Das Setzen von Synchronisierungspunkten erfolgt in Cilk

explizit mithilfe des Schlüsselworts `sync`, während die vorliegende Arbeit parallele Anweisungen mit impliziten Barrieren verwendet und somit durch mehr Abstraktion das Fehlerrisiko senkt.

Wie die vorliegende Arbeit lässt sich Cilk in objektorientierte Sprachen integrieren. Zugriffe auf gemeinsame Variablen müssen in Cilk vom Programmierer erkannt werden. Der Work-stealing-Algorithmus kann zwar als eine spezielle Form der dynamischen Optimierung aufgefasst werden; jedoch beschränkt sich diese Art der Optimierung auf den Bereich Ablaufplanung. Ein auf Tuningparametern basierendes Laufzeit-Tuning ist damit nicht gegeben; so ist z.B. die Spezifikation algorithmischer Alternativen und deren automatischer Auswahl durch einen Auto-Tuner in Cilk, anders als in der vorliegenden Arbeit, nicht möglich.

3.2.2 OpenMP

OpenMP („Open Multi-Processing“) [24, 25] ist eine Programmierschnittstelle, welche Übersetzerdirektiven und Bibliotheksfunktionen zur Parallelprogrammierung von Systemen mit gemeinsamen Speicher umfasst. Das ursprüngliche Ziel war es, einen Ansatz zu schaffen, um Schleifenparallelität in numerischen Algorithmen und Programmen auf einfache Art implementieren zu können. Das Grundkonzept besteht darin, Schleifeniterationen mithilfe von Pragma-Direktiven, die vom Programmierer in den Programmcode eingefügt werden, automatisiert auf verschiedene Fäden zu verteilen. Listing 3.9 zeigt eine einfache OpenMP-Schleife zur parallelen Vektoraddition.

Listing 3.9: Parallele Schleife in OpenMP

```
1 #pragma omp parallel for
2 for (i = 0; i < N; i++) {
3   c[i] = a[i] + b[i];
4 }
```

Während der Funktionsumfang von OpenMP über die Jahre gewachsen ist, liegt der Fokus nach wie vor auf Programmen mit Fork-Join-Parallelität, also Programmen mit aufeinander folgenden sequenziellen und parallelen Sektionen. In neueren Versionen wird das fadenbasierte durch ein aufgabenbasiertes Parallelisierungskonzept ersetzt, um auch Programme mit sogenannter „unstrukturierter Parallelität“ (z.B. while-Schleifen oder Rekursion) abdecken zu können. Für Variablen innerhalb paralleler Sektionen können unterschiedliche Sichtbarkeiten (z.B. `shared`, `private`) oder Reduktionsschritte (`reduction`) definiert werden; ebenso stehen zahlreiche Synchronisierungskonstrukte wie z.B. Barrieren, kritische oder atomare Abschnitte zur Verfügung.

Viele OpenMP-Übersetzer führen Optimierungen wie das Ausrollen oder Verschmelzen von Schleifen durch. Der Programmierer kann zudem aus verschiedenen Ablaufplanungs- bzw. Lastverteilungsstrategien (`static`, `dynamic`, `guided`) wählen, um die Verteilung von Arbeit auf Ausführungsfäden zu beeinflussen.

3.2.2.1 Abgrenzung

OpenMP stellt einen wichtigen und weit verbreiteten Ansatz zur Implementierung von Fork-Join-Parallelität dar. Im Mittelpunkt des Interesses stehen daten- und aufgabenparallele Berechnungen. Für Fließband- bzw. Stromverarbeitung ist OpenMP dagegen weniger geeignet.

Im Unterschied zur vorliegenden Arbeit bietet OpenMP keine höheren Sprachabstraktionen, sondern erfordert mehr „Handarbeit“ vom Programmierer. Beispielsweise müssen gemeinsame oder lokale Variablen speziell gekennzeichnet und explizite Synchronisierungskonstrukte verwendet werden; ebenso müssen Parameter wie die Fadenanzahl oder die Art der Lastverteilung manuell festgelegt werden. Die vorliegende Arbeit abstrahiert dagegen von dieser Problematik und bietet durch Laufzeit-Tuning ein flexibles, dynamisches Optimierungskonzept.

3.2.3 Parallele HPCS Programmiersprachen

Im Rahmen des HPCS („High Productivity Computing Systems“) Programms der DARPA (Defense Advanced Research Projects Agency) wurden in den letzten Jahren Prototypen neuer Programmiersprachen vorgestellt, die zur Programmierung von Anwendungen für Systeme mit gemeinsamen Adressraum konzipiert sind. Im Folgenden werden die Sprachen Chapel, Fortress und X10 kurz vorgestellt, deren Schwerpunkt auf der Programmierung von parallelen Hochleistungsrechnern liegt. Diese Sprachen unterscheiden sich in ihren Zielsetzungen, Einsatzgebieten und Sprachentwürfen von der vorliegenden Arbeit, teilen sich jedoch den Ansatz, parallele und objektorientierte Konzepte zu verbinden.

Chapel [23] ist eine eigenständige Sprache, die syntaktische Ähnlichkeiten mit High Performance Fortran (HPF) aufweist. Im Fokus der Sprache stehen Daten- und Aufgabenparallelität. So sind Konstrukte wie Schleifen und Reduktionsoperatoren implizit datenparallel. Nebenläufige Aufgaben werden mithilfe von `cobegin`-Blöcken erzeugt und gestartet. Um Zugriffe auf gemeinsame Variablen korrekt zu synchronisieren, stehen statt expliziten Sperren atomare Blöcke zur Verfügung.

Fortress [10] ist eine auf Fortran basierende objektorientierte Sprache, welche eine Reihe implizit paralleler Operatoren umfasst. Die implizite Parallelverarbeitung beschränkt sich jedoch

auf eine sehr niedrige Abstraktionsebene. Um Parallelität auf höherer Ebene, z.B. für ganze Programmteile, zu generieren, müssen explizite Fäden erzeugt und koordiniert werden. Um die Synchronisierungsproblematik zu vereinfachen, stellt Fortress spezielle atomare Ausdrücke bereit.

X10 [26] basiert auf der Sprache Java, ist jedoch keine Erweiterung, sondern eine eigenständige Programmiersprache. Ein zentraler Bestandteil von X10 ist das Konzept des logischen Orts (engl. *place*), welches dazu dient, Daten kohärent zu verwalten. Der globale Speicher ist in Orte aufgeteilt, denen bestimmte Daten und Fäden fest zugeordnet sind. Zugriffe auf ein Datum eines bestimmten Orts können nur durch Fäden desselben Orts erfolgen. Diese Eigenschaft gewährleistet das sogenannte PGAS Speichermodell (partitioned global address space bzw. partitionierter globaler Adressraum). Ein besonderer Schwerpunkt in X10 liegt auf diversen Array-Operationen und funktionalem Programmieren. Wie in Chapel erfolgt die Synchronisierung von Variablenzugriffen durch die Verwendung atomarer Blöcke.

3.2.3.1 Abgrenzung

Die vorgestellten HPCS Programmiersprachen sind für hochgradig parallele Hochleistungsrechner konzipiert; der Fokus liegt auf Daten- und Aufgabenparallelität. Laufzeit-Tuning ist bisher nicht Bestandteil der HPCS Sprachen. Die vorliegende Arbeit stellt dagegen stromorientierte Anwendungen für Multikernsysteme in den Mittelpunkt und bietet mit Laufzeit-Tuning ein flexibles und für Stromprogramme geeignetes Optimierungskonzept.

Um Zugriffe auf gemeinsame Variablen bzw. kritische Abschnitte zu schützen, stehen in allen drei HPCS Sprachen atomare Blöcke zur Verfügung. In keiner der Sprachen besteht übersetzerseitige Unterstützung für die Identifikation kritischer Abschnitte, während die vorliegende Arbeit Programmanalysen für objektorientierte Stromprogramme beinhaltet. Insgesamt erfordern Chapel, Fortress und X10 deutlich mehr explizite, feingranulare Synchronisierung, wobei Abstraktion auch nicht das primäre Ziel dieser Sprachen ist.

3.2.4 MapReduce

MapReduce [32] ist ein Programmiermodell zur Parallelverarbeitung für sehr große Datenmengen. Die Eingabe eines MapReduce-Programms ist ein Menge von Schlüssel-Wert-Paaren, für welche eine Ausgabemenge von Schlüssel-Wert-Paaren berechnet wird. Gemäß der Namensgebung lassen sich zwei Phasen bzw. Funktionen unterscheiden, die vom Programmierer zu implementieren sind:

- *Abbildungsfunktion („map“).* Die map-Funktion produziert für ein Eingabe-Paar eine Menge von vorläufigen Schlüssel-Wert-Paaren.

- *Reduktionsfunktion („reduce“).* Die Eingabe der reduce-Funktion ist ein vorläufiger Schlüssel sowie eine Menge von Werten für diesen Schlüssel. Die Funktion fügt diese Werte zusammen, d.h. reduziert sie zu einer i.d.R. kleineren Datenmenge.

Die MapReduce-Bibliothek ist dafür zuständig, die in der map-Phase erzeugten Zwischenwerte, die demselben vorläufigen Schlüssel zugeordnet sind, zu gruppieren und der reduce-Funktion zu übergeben. Zwischen der map- und reduce-Phase kann ein combine-Schritt eingefügt werden. Dieser besitzt prinzipiell dieselbe Funktionalität wie die reduce-Funktion, wird jedoch nur auf denjenigen Teil der vorläufigen Wertmenge angewandt, der einem einzelnen Rechenknoten zugeordnet ist.

Die Aufrufe sowohl der map- als auch der reduce-Funktion lassen sich parallelisieren. Der Fokus des MapReduce-Konzepts liegt somit auf der Ausnutzung von Datenparallelität. Ursprünglich für Rechnerbündel entwickelt, wurde der Ansatz auch für Multikernsysteme evaluiert und angepasst [88, 101]. Allerdings kann hierbei die strikte zeitliche Trennung der map- und reduce-Phase zu suboptimaler Ressourcennutzung führen, wenn sehr große Datenmengen zur selben Zeit verarbeitet werden. Ein Ansatz, dieses Problem für Multikernrechner mit gemeinsamem Speicher zu lösen, wird in [27] vorgestellt und diskutiert. Dieses Verfahren („Tiled-MapReduce“) partitioniert die Eingabedatenmenge in kleinere Teilmengen, welche nacheinander nach dem MapReduce-Prinzip verarbeitet werden.

3.2.4.1 Abgrenzung

Im Unterschied zum MapReduce-Ansatz ist die vorliegende Arbeit nicht nur auf Datenparallelität, sondern auch auf Fließband- und Aufgabenparallelität ausgerichtet. MapReduce geht von sehr großen Datenmengen aus, die in *unabhängige* Teilmengen zerlegt werden können, in denen eine gesonderte Berücksichtigung von Zuständen oder gemeinsamen Variablen nicht erforderlich ist.

Wie stromverarbeitende Anwendungen besitzen auch MapReduce-Programme Tuningparameter, welche die Performanz maßgeblich beeinflussen, z.B. die Anzahl der reduce-Aufgaben oder verschiedene Block- und Puffergrößen [16]. Die (Vor-) Konfiguration der Parameter gestaltet sich im MapReduce-Kontext meist als schwierig, da Größe und Beschaffenheit der zu verarbeitenden Datenmenge hierbei wesentliche, aber schwer zu charakterisierende Einflussfaktoren darstellen. Vielversprechende Optimierungstechniken für MapReduce basieren daher auf dynamischer Profilerstellung und Kostenmodellen [45, 44]. Diese Techniken können als eine sehr domänenspezifische Form von modellbasiertem Laufzeit-Tuning betrachtet werden. Die Optimierungskonzepte der vorliegenden Arbeit befassen sich zwar ebenfalls mit Laufzeit-Tuning, basieren jedoch nicht auf einem spezialisierten Kostenmodell, sondern sind für allgemeine Stromprogramme anwendbar.

3.2.5 Algorithmische Skelette für Parallelverarbeitung

Algorithmische Skelette für Parallelverarbeitung [31, 29, 20, 62] übertragen das Konzept algorithmischer Skelette [28] auf Mehrkernrechner bzw. Rechnerbündel. Skelette beschreiben grundlegende, oft verwendete Programmiermuster und trennen somit Struktur von Funktionalität. Um ein Skelett zu implementieren, verwendet man i.d.R. eine Programmierbibliothek, um die entsprechende Struktur nach einem vorgegebenen Gerüst automatisch zu erzeugen und mit selbst implementierter Funktionalität bzw. konkreten Algorithmen zu „füllen“.

Listing 3.10: Komposition algorithmischer Skelette zu einem Fließband

```
1 class Execute1 implements Execute { ... }
2 class Execute2 implements Execute { ... }
3
4 Skeleton stage1 = new Seq(new Execute1());
5 Skeleton stage2 = new Seq(new Execute2());
6 Skeleton pipe = new Pipe(stage1, stage2);
```

Listing 3.10 zeigt ein verkürztes Codebeispiel aus [20] zur Implementierung eines zweistufigen Fließbands. Die Skelettbibliothek stellt eine Schnittstelle `Execute` bereit, welche eine nebenläufige Aufgabe kapselt. Der Programmierer implementiert deren Funktionalität in Form der Klassen `Execute1` und `Execute2` (Zeilen 1 und 2). Diese Bausteine werden sequenziellen Skeletten `Seq` zugeordnet (Zeilen 4 und 5), welche schließlich zu einem Fließbandskelett `Pipe` kombiniert werden (Zeile 6).

Ein Vorteil algorithmischer Skelette liegt in der sauberen Trennung von Struktur und Funktionalität sowie der daraus resultierenden Abstraktion. Dem Programmierer stehen Implementierungsmuster mit vorgegebenen Strukturen und vorhersagbaren Verhaltensweisen zur Verfügung. Skelette sind i.d.R. kombinier- und schachtelbar, um neue, komplexere Skelette zu erzeugen. Darüber hinaus können Skelette Ausnahmen (engl. *exceptions*) im parallelen Kontext behandeln [62] sowie die Identifikation von Performanzproblemen bei ungünstig geschachtelten Skeletten unterstützen [21].

3.2.5.1 Abgrenzung

Der Schwerpunkt algorithmischer Skelette für Parallelverarbeitung liegt in der Automatisierung der Implementierung bestimmter, häufig verwendeter paralleler Codemuster wie Fließbänder, Auftraggeber-Auftragnehmer oder Datenzerlegung. Insofern kann der Ansatz zur Entlastung des Entwicklers und mittelbar zur Fehlerreduktion beitragen. Dieser Abstraktionsschritt ist vergleichbar mit dem der vorliegenden Arbeit, bei der allerdings keine Skelette, sondern Spracherweiterungen die Basis bilden.

Im Unterschied zur vorliegenden Arbeit beinhalten Skelette keine Konzepte zur unmittelbaren Unterstützung der Programmkorrektheit bzw. Performanzoptimierung. Die vorliegende Arbeit integriert dagegen Konzepte zur Erkennung von Zustandsvariablen und gemeinsamen Variablen und bietet durch Laufzeit-Tuning ein Verfahren zur automatischen Performanzoptimierung.

3.2.6 Weitere Arbeiten

Eine Übersicht und Kategorisierung paralleler objektorientierter Sprachen bezüglich der Aspekte Parallelitätserzeugung und -verwaltung findet sich in [86]. Dieser Artikel aus dem Jahr 2000 erfasst zwar keine neuen Sprachen bzw. Sprachkonzepte der letzten Jahre; dennoch werden viele der heute etablierten Sprachen darin abgedeckt.

Funktionale Programmiersprachen beschreiben ein Programm als Menge von Funktionen, die in einer bestimmten Reihenfolge kombiniert und ausgewertet werden. Das Parallelisierungspotenzial liegt darin, dass Funktionen definitionsgemäß frei von Seiteneffekten sind. Die Sprachmächtigkeit ist, ähnlich wie die von stromorientierten Sprachen, jedoch eingeschränkt. Erlang [14] ist eine funktionale, nebenläufige Sprache, die insbesondere zur Entwicklung von industriellen Echtzeitsystemen konzipiert ist. Die Sprache Scala [78, 77] kombiniert funktionale und objektorientierte Konzepte und bietet Bibliotheksfunktionen zur parallelen Programmierung.

Intel Threading Building Blocks (TBB) [90] ist eine Rahmenwerk zur parallelen Programmierung in C++. TBB bietet verschiedene parallele Konstrukte in Form von Schablonen (engl. *templates*) an, durch die von der expliziten Erstellung und Verwaltung von Ausführungsfäden abstrahiert wird. Einige Schablonen sind vergleichbar mit algorithmischen Skeletten und dienen dem Erzeugen paralleler Schleifen, Aufgabenparallelität und linearen Fließbändern. Aktuelle Versionen der TBB unterstützten mittlerweile auch nichtlineare Fließbänder. StreamCpp [35] ermöglicht stromorientierte Programmierung auf Basis von TBB und wurde dabei von der vorliegenden Arbeit beeinflusst. Darüber hinaus existiert eine Vielzahl an parallelen Datenstrukturen, atomaren Operationen und Synchronisierungskonstrukten.

Die Task Parallel Library (TPL) [60] ist ein Bibliotheksansatz für das .NET Rahmenwerk, der sehr ähnliche Ziele wie TBB verfolgt. Der Fokus von TPL liegt dabei mehr auf Aufgabenparallelität und Schleifenparallelisierung, wobei auch Konstrukte zur Implementierung von Fließbändern existieren.

Sowohl TBB als auch TPL adressieren sehr unterschiedliche Abstraktionsebenen. Die Programmierkonstrukte sind teilweise sehr feingranular, was einerseits dem Programmierer differenzierte Kontrollmöglichkeiten bietet, andererseits auch eine Fehlerquelle darstellt. Sowohl

TBB als auch TPL besitzen eigene Ablaufplaner, die auf Workstealing basieren. Diese Ablaufplaner können innerhalb des Anwendungscodes angesprochen und modifiziert werden. Weitergehende Optimierungskonzepte zur Definition, Einstellung und Anpassung von Tuningparametern werden nicht adressiert.

Die Parallel Pattern Language [66] ist ein Ansatz zum Auffinden und Beschreiben von musterbasierter Parallelität. Dieses Modell kann als Orientierungshilfe für Programmierer verstanden werden, um den Entwurfs- und Entwicklungsprozess zu unterstützen. Hierbei handelt es sich jedoch in erster Linie um ein Vorgehensmodell und weniger um eine Sprache.

Merkmal	**Cilk/Cilk++**	**OpenMP**	**HPCS Sprachen**	**MapReduce**	**Algorithmische Skelette**	**diese Arbeit**
Umsetzung	SpE	ÜD	Sp	Bib	Bib	SpE
Parallelitätsarten	A	A/D	A/D	D	A/D/F	A/D/F
Explizite Synchronisierung	X	X	X	-	-	-
Kritische Abschnitte	M	M	M	M	M	A
Ablaufplanung	D	D/S	D	D/S	D	D
Laufzeit-Tuning	-	-	-	X	-	X

Tabelle 3.2: Vergleich der Arbeiten zur parallelen Programmierung

3.2.7 Zusammenfassung

Tabelle 3.2 fasst die Merkmale der Arbeiten zu parallelen, nicht-stromorientierten Programmierung zusammen:

- Die *Umsetzung* gibt an, auf welche Art parallele Programmierung unterstützt wird (Sp: Sprache, SpE: Spracherweiterung, Bib: Bibliothek, ÜD: Übersetzerdirektiven).
- Ein Programmieransatz kann auf bestimmte *Parallelitätsarten* spezialisiert sein (A: Aufgabenparallelität, D: Datenparallelität, F: Fließbandparallelität).
- *Explizite Synchronisierung* von Aufgaben oder Aktivitäten ist dann erforderlich, wenn vom Programmierer explizite Synchronisierungskonstrukte wie z.B. Barrieren in den Programmcode eingebaut werden müssen.

- Die Identifikation *kritischer Abschnitte* erfolgt entweder manuell (M) durch den Programmierer oder automatisch (A).
- Die *Ablaufplanung* kann statisch (S) oder dynamisch (D) erfolgen.
- Das Merkmal *Laufzeit-Tuning* ist gegeben, wenn Auto-Tuning (Leistungsmessung und -optimierung) im Produktivbetrieb möglich ist.

Parallele Programmiersprachen, die nicht in den Bereich der stromorientierten Programmierung fallen, fokussieren sich meist auf Aufgaben- und Datenparallelität. Lediglich algorithmische Skelette bieten auch explizite Unterstützung zur Implementierung von Fließbandparallelität.

Keine der besprochenen Sprachen bzw. Bibliotheken bieten integrierte Programmanalysen zur automatischen Erkennung kritischer Abschnitte; die Mehrheit der Ansätze erfordert zudem die Verwendung expliziter Synchronisierungskonstrukte.

Die meisten Arbeiten bieten zudem bisher keine Möglichkeiten, Laufzeit-Tuning durchzuführen. Lediglich im Rahmen von MapReduce wurde ein dynamisches Optimierungsverfahren entwickelt, welches jedoch auf vorab gesammelten Laufzeitprofilen und speziellen Kostenmodellen basiert.

3.3 Arbeiten zur Identifikation von Wettlauffehlern

Ergänzend verweist dieser Abschnitt auf ausgewählte Arbeiten, die sich mit der statischen Identifikation von ungeschützten gemeinsamen Variablen und kritischen Abschnitten befassen. Im Wesentlichen fallen diese Arbeiten in den Bereich der Wettlauferkennung, welcher ein eigenständiges Forschungsgebiet darstellt und hier in seiner Gesamtheit nicht behandelt werden soll.

Statische Wettlauferkenner [34, 72, 67] führen Programmanalysen für gesamte Programme durch, um unsynchronisierte Zugriffe auf gemeinsame Variablen und somit mögliche Wettläuffehler zu identifizieren. Hierbei handelt es sich i.d.R. entweder um typbasierte flussinsensitive oder flusssensitive Verfahren, die auf der Berechnung von Locksets beruhen. Locksets sind Mengen, die angeben, durch welche Sperren eine bestimmte Anweisung geschützt ist.

Konfliktanalysen basierend auf Objektverwendungsgraphen (engl. *object use graph*) [112] modellieren die Laufzeitstruktur und Interaktion gemeinsamer Objekte in Abhängigkeit von verschiedenen Ausführungsfäden. Objektverwendungsgraphen enthalten unter anderem Informationen zu Zeigern und zeitlichen Beziehungen zwischen Variablenzugriffen. Diese Informationen können sowohl zur statischen Wettlauferkennung als auch zur Laufzeitprüfung mithilfe entsprechender Codeinstrumentierungen verwendet werden.

Die Identifikation atomarer Regionen [111] erweitert die Idee der klassischen Wettlauferkennung. Hierbei werden zunächst ungeschützte Zugriffe auf gemeinsame Variablen ermittelt. Darauf aufbauend werden diejenigen Anweisungen bestimmt, die zu einer atomaren Region zusammengefasst werden müssen, um Konfliktserialisierbarkeit zu erreichen. Dieses Verfahren basiert auf Konfliktgraphen, die nach dem Prinzip stark zusammenhängender Komponenten (engl. *strongly connected components*) aufbereitet werden. Atomare Regionen können durch die Verwendung von transaktionalem Speicher implementiert werden. Dessen Praktikabilität ist jedoch aufgrund des oftmals erheblichen Mehraufwands durch neue Speicheroperationen nicht unumstritten [22]. Ein Ansatz zur Verringerung dieses Aufwands wird in [6] vorgestellt und diskutiert; dieser Mechanismus basiert auf der statischen Erkennung redundanter Lese- und Schreiboperationen.

3.4 Arbeiten zur Optimierung paralleler Programme

In diesem Abschnitt werden Arbeiten diskutiert, die sich mit der Optimierung der Performanz paralleler Programme befassen. Entsprechend dem Fokus dieser Arbeit stehen keine einzelnen Optimierungsalgorithmen im Mittelpunkt, sondern vielmehr Gesamtkonzepte.

3.4.1 PetaBricks

PetaBricks [12, 11, 13] ermöglicht Entwicklern, algorithmische Alternativen in Programmen zu spezifizieren. Ein PetaBricks-Programm definiert somit nicht einen einzigen algorithmischen Pfad, sondern einen Suchraum aus verschiedenen Pfaden. Die Auswahl einer bestimmten Alternative wird durch einen Auto-Tuner getroffen und basiert auf zuvor gesammelten Laufzeitprofilen.

Ausgangspunkt ist eine eigene an C++ angelehnte Sprache, durch welche ein Programm als Verknüpfung bzw. Verschachtelung von Transformationen beschrieben wird. Eine Transformation ist eine Vorschrift, die Eingabe- in Ausgabedaten umwandelt, und kann aus mehreren algorithmischen Alternativen, sogenannten Regeln, bestehen. Listing 3.11 zeigt den Code einer Transformation, welche ein Array `A` in ein sortiertes Array `B` überführt; die Regeln dieser Transformation definieren verschiedene Sortierverfahren.

Das Optimierungsproblem von PetaBricks besteht nun darin, für jede Transformation diejenige Regel bzw. Alternative auszuwählen, die für eine gegebene Hardwarearchitektur die beste Performanz erzielt. Hierfür verwendet PetaBricks einen Auto-Tuner, der vor der Programmausführung Laufzeitprofile der verschiedenen Implementierungen bzw. Regeln erstellt. Die Auswahl einer Regel erfolgt zur Laufzeit des Programms.

Listing 3.11: Aufbau einer Transformation in PetaBricks

```
transform Sort
from A[n]
to B[n]
{
  // Regel 1 (Alternative 1)
  to(B sorted) from (A a) {
    InsertionSort(a, sorted);
  }
  // Regel 2 (Alternative 2)
  to(B sorted) from (A a) {
    QuickSort(a, sorted);
  }
  // Regel 3 (Alternative 3)
  to(B sorted) from (A a) {
    MergeSort(a, sorted);
  }
}
```

3.4.1.1 Abgrenzung

Die Sprache PetaBricks erlaubt die Definition algorithmischer Alternativen, ermöglicht im Unterschied zur vorliegenden Arbeit jedoch keine Stromverarbeitung. Der PetaBricks Auto-Tuner ist speziell für die Auswahl algorithmischer Alternativen konzipiert und basiert auf zuvor erstellten Laufzeitprofilen. Die Leistungsmessung findet somit vor der Programmausführung (offline) statt, die Auswahl einer Alternative dagegen während der Programmausführung (online). Interpretiert man die Laufzeitprofile als Modell, so kann der Tuning-Mechanismus von PetaBricks als modellbasiertes Laufzeit-Tuning bezeichnet werden. Der Vorteil ist, dass sowohl Suchraum als auch Leistungsfunktion bekannt sind und Programmkonfigurationen schnell angepasst werden können. Der entscheidende Nachteil ist, dass die algorithmischen Alternativen für eine konkrete Hardwarearchitektur vorab vermessen werden müssen.

Die vorliegende Arbeit basiert dagegen nicht auf zuvor erstellten Laufzeitprofilen, sondern bestimmt die Programmleistung ebenfalls im Produktivbetrieb; somit handelt es sich um reines Laufzeit-Tuning.

3.4.2 Active Harmony

Eine ähnliche Zielsetzung wie PetaBricks verfolgt Active Harmony [102, 109]. Dieser Ansatz geht davon aus, dass Programme Standardfunktionen aus Bibliotheken verwenden und hierfür unterschiedliche Implementierungen existieren, die zu unterschiedlichen Programmleistungen

führen. Durch eine einheitliche Schnittstelle soll von der konkreten Implementierung abstrahiert werden und die Auswahl einer geeigneten Implementierung mithilfe von Auto-Tuning erfolgen.

Im Unterschied zu PetaBricks ermöglicht Active Harmony sowohl Offline- als auch Online-Tuning. Da die Programmleistung auch zur Laufzeit bestimmt werden kann, ist im Fall von Online-Tuning keine vorherige Profilerstellung erforderlich. Als Leistungskriterium können außer der Ausführungszeit auch andere Kriterien definiert werden. Sowohl Leistungskriterium als auch Messpunkte müssen vom Programmierer im Code spezifiziert werden. Zusätzlich müssen Informationen zur Anwendung sowie ggf. Hardwareanforderungen in einer separaten Beschreibungssprache („Resource Specification Language“, RCL) angegeben werden.

Einen zentralen Platz in Active Harmony nimmt der Entwurf und die Verfeinerung des Optimierungsalgorithmus ein [100, 110, 108], der auf dem Simplex-Verfahren nach Nelder und Mead [73] aufbaut.

3.4.2.1 Abgrenzung

Die Überschneidung von Active Harmony und der vorliegenden Arbeit besteht darin, mithilfe von suchbasiertem Auto-Tuning Programme im Produktivbetrieb zu optimieren. Active Harmony fokussiert sich dabei auf Programme mit austauschbaren Bibliotheksfunktionen, während die vorliegende Arbeit von stromorientierten Programmen ausgeht.

Ein weiterer, wesentlicher Unterschied zwischen beiden Ansätzen liegt in der Form, in der der Programmierer mit Auto-Tuning konfrontiert wird. Active Harmony erfordert es, Tuningparameter, Messpunkte, Eigenschaften der Anwendung sowie Hardwareanforderungen explizit zu spezifizieren und bietet hierfür eine eigene Beschreibungssprache. Die vorliegende Arbeit abstrahiert hiervon, indem Tuningparameter, Messpunkte und Kontextinformationen über die Anwendung ohne Zutun des Programmierers vom Übersetzer extrahiert werden.

3.4.3 FIBER

FIBER („Framework of Install-time, Before Execute-time and Run-time Optimization“) [53] stellt ein für Fortran implementiertes Rahmenwerk dar, um Bibliotheksfunktionen basierend auf Leistungsmessungen iterativ zu optimieren. Der Schwerpunkt liegt auf numerischen Anwendungen für Systeme mit verteiltem Speicher. Der Ansatz unterscheidet Tuningparameter bezüglich ihres Optimierungszeitpunkts: bei der Installation des zu optimierenden Programms, vor der Ausführung und während der Ausführung.

Der Programmierer fügt hierzu Funktionsaufrufe und Annotationen in den Quelltext ein und ordnet jeden Tuningparameter einer Kategorie zu, um den Optimierungszeitpunkt festzulegen. Der Optimierer stellt für jede Kategorie, d.h. jeden Optimierungszeitpunkt, eine eigene Schicht zur Verfügung.

Die Leistungsbestimmung erfolgt über die Aufzeichnung von Laufzeitprofilen für die benötigten Bibliotheksfunktionen. Hierdurch wird ein Performanzmodell erstellt, auf dessen Grundlage der Auto-Tuner entscheidet, wie die Tuningparameter zu konfigurieren sind. Tuningparameter werden dann gemäß ihrer Kategorie zum Zeitpunkt der Installation, vor oder während der Ausführung eingestellt.

3.4.3.1 Abgrenzung

Während in der vorliegenden Arbeit objektorientierte Stromprogramme für Multikernsysteme im Mittelpunkt stehen, ist FIBER für numerische Anwendungen konzipiert, die auf parallelen Systemen mit verteiltem Speicher ausgeführt werden. Im Vergleich zur vorliegenden Arbeit entsteht dem Programmierer ein Mehraufwand für explizite, detaillierte Annotationen, welche Voraussetzung für die Optimierbarkeit eines Programms sind.

Des Weiteren erfolgt die Leistungsbestimmung für die Bibliotheksfunktionen in FIBER vor der Programmausführung; die Konfiguration der Tuningparameter basiert auf einem daraus gewonnenen Performanzmodell. Die vorliegende Arbeit führt dagegen die Leistungsbestimmung im Produktivbetrieb durch; eine (ggf. sehr aufwändige) Vermessung des Stromprogramms bzw. seiner Teile ist vorab nicht notwendig.

3.4.4 Orio

Orio [43] umfasst eine Annotationssprache für C-Programme zur Generierung funktional äquivalenter Codevarianten. Des Weiteren steht eine Komponente zur Konfiguration des transformierten Programms zur Verfügung, welches mit suchbasierten Methoden die performanteste Codevariante auswählt.

Die Annotationssprache dient zur Beschreibung der Transformationsvorschriften. Diese verweisen auf ein externes Skript, in welchem besondere Randbedingungen festgelegt werden. So spezifiziert das Skript die konkreten Tuningparameter, deren Wertebereiche sowie Eingabewerte für die zu optimierenden Funktionen. Der Orio-Präprozessor generiert aus dem annotierten Programm und dem Skript zur Tuningspezifikation reinen C-Code. Dieser fungiert als Eingabe für den Optimierer, welcher basierend auf den spezifizierten Eingabewerten Leistungsmessungen für verschiedene Parameterkonfigurationen durchführt.

Die Suchkomponente verwendet wahlweise das Verfahren der simulierten Abkühlung oder den Simplex-Algorithmus nach Nelder und Mead [73]. Die Optimierung findet iterativ für

eine Folge von Testläufen statt, jedoch nicht im Produktivbetrieb des zu optimierenden Programms. Der manuelle Vorbereitungsaufwand für Annotationen und Skripterstellung ist als hoch einzustufen. Im Fokus liegen daher eher kleine Programme und Algorithmen mit wenigen Parametern.

3.4.4.1 Abgrenzung

Ähnlich wie Active Harmony und FIBER setzt auch Orio eine detaillierte Spezifikation von Tuninginformationen durch den Programmierer voraus, während die vorliegende Arbeit den Ansatz verfolgt, tuningrelevante Informationen aus dem Quelltext automatisch zu extrahieren. Darüber hinaus bietet Orio keine explizite Unterstützung für parallele Programme.

Der Schwerpunkt von Orio liegt auf der Spezifikationssprache sowie der Transformation eines Programms in eine optimierbare Form. Die Optimierung erfolgt durch Offline-Tuning, während die vorliegende Arbeit Laufzeit-Tuning durchführt.

3.4.5 Atune

Atune [92] befasst sich mit dem Entwurf, der Implementierung und Optimierung paralleler Softwarearchitekturen. Das Gesamtkonzept unterteilt sich in drei Teilkonzepte: eine Instrumentierungssprache zur Spezifikation von Tuninginformationen, eine Sprache zur Architekturbeschreibung sowie einem suchbasierten Auto-Tuner.

Die Instrumentierungssprache Atune-IL [93] erlaubt es, mithilfe von Pragma-Direktiven Tuningparameter und deren Wertebereiche im Code festzulegen sowie Abhängigkeiten und Wirkungsbereiche der Parameter zu spezifizieren. Diese Information kann von einem Auto-Tuner verwendet werden, um den Suchraum derart zu partitionieren und reduzieren, dass „überflüssige“ Parameterkonfigurationen nicht mehr getestet werden müssen [91].

Die Architekturbeschreibungssprache baut auf der Instrumentierungssprache auf und dient dazu, die Struktur einer parallelen Anwendung zu spezifizieren. Auf diese Weise lassen sich konfigurierbare Anwendungen erstellen, die eine Optimierung ermöglichen [94]. Die Optimierungskomponente führt schließlich suchbasiertes Offline-Tuning durch. So können in Abhängigkeit von Eingabedaten und zugrundeliegender Hardware Parameterkonfigurationen ermittelt werden, die zu bestmöglicher Performanz führen.

Im Rahmen von Fallstudien wird gezeigt, dass das Gesamtkonzept Atune, unabhängig von Größe, Anwendungstyp und Zielplattform, die Optimierung von parallelen Applikationen ermöglicht.

3.4.5.1 Abgrenzung

Im Unterschied zur vorliegenden Arbeit ist Atune für Offline-Tuning konzipiert. Die Leistungsbestimmung und -optimierung erfolgt in Testläufen. Die vorliegende Arbeit erfordert dagegen keine Testläufe, sondern führt Programmoptimierungen im Produktivbetrieb durch. Parameterwerte können somit während der Programmausführung verändert werden.

Ein weiterer Unterschied zwischen den Arbeiten liegt in der Umsetzung. Atune basiert auf einer eigenen Tuning-Instrumentierungssprache sowie einer Bibliothek zur Implementierung optimierbarer paralleler Muster. Die vorliegende Arbeit basiert auf einer Spracherweiterung, die aus Sicht des Entwicklers von Auto-Tuning abstrahiert.

3.4.6 Application Heartbeats

Die Application Heartbeats Programmierschnittstelle [47] befasst sich mit der Leistungsbestimmung von Programmen im Produktivbetrieb. Die Grundidee hierbei ist es, die Leistung eines Programms nicht in Form von Zeit, sondern durch sogenannte „Herzschläge“ bzw. Pulse (Ereignisse pro Zeit) zu messen. Auf diese Weise können beispielsweise Performanzziele für Anwendungen vorgegeben werden. Die Programmoptimierung durch Anpassung von Tuningparametern liegt dagegen nicht im unmittelbaren Fokus dieser Arbeit.

Um den Puls, also die Performanz eines Programms zu messen, muss der Programmierer zunächst geeignete Stellen im Code identifizieren, an denen ein Ereignis auszulösen ist. Dies sind typischerweise Schleifen, deren Iterationen vergleichbare Arbeitslast zu bewältigen haben. Ein Ereignis wird durch einen Funktionsaufruf der Heartbeat-Programmierschnittstelle (API) ausgelöst und an zentraler Stelle registriert, d.h. entweder im Programm oder in einer externen Komponente. Soll die Performanz nicht nur gemessen werden, sondern auch beeinflussbar sein, sind zusätzlich Tuningparameter zu deklarieren. Um ein Performanzziel auszugeben, kann das Programm ebenfalls per Funktionsaufruf einen Soll-Wert des Pulses festlegen.

Die Anpassung der Parameter basierend auf deren Wertehistorie sowie dem Soll-Ist-Vergleich wird in der Literatur nicht weiter beschrieben. In den untersuchten Fallbeispielen werden hierfür meist externe Komponenten verwendet, die offenbar nicht Bestandteil des Heartbeats-Systems sind. Selbstoptimierende Anwendungen verwenden eigene, programmintern spezifizierte Optimierungsmodule.

3.4.6.1 Abgrenzung

Der Ansatz der Application Heartbeats konzentriert sich auf das Problem der Leistungsbestimmung, befasst sich jedoch nicht mit der Optimierung der Leistung. Hierfür muss der Program-

mierer den Quelltext manuell mit Funktionsaufrufen der Heartbeat-Programmierschnittstelle instrumentieren. Die vorliegende Arbeit erweitert das Konzept des Pulses, so dass auf dieser Grundlage die Leistung von Stromprogrammen bestimmt und optimiert werden kann. Die Instrumentierung des Codes erfolgt zudem automatisch.

3.4.7 Just-in-time Übersetzung

Just-in-time Übersetzung (JIT-Übersetzung) stellt ein Konzept dar, um Programmteile nicht vor, sondern während der Ausführung in Maschinencode zu überführen. Insofern kann diese Art der Übersetzung als dynamische Optimierungsform verstanden werden.

JIT-Übersetzung spielt vor allem im Zusammenhang mit virtuellen Maschinen eine Rolle, die für die Ausführung von Programmen zuständig sind. Beispielsweise werden Sprachen wie Java oder C# nicht in Maschinencode, sondern in eine plattformunabhängige Zwischensprache übersetzt, welche von einer virtuellen Maschine interpretiert bzw. ausgeführt wird. Diese Form der Ausführung bringt tlw. erhebliche Performanzeinbußen mit sich. Um dem entgegenzuwirken, kompilieren JIT-Übersetzer Programmteile direkt in performanteren Maschinencode. Da die Übersetzung zur Laufzeit ebenfalls Rechenleistung kostet, müssen JIT-Übersetzer entscheiden, ob und mit welchen Optimierungen ein Programmteil übersetzt werden soll. Viele JIT-Übersetzer wie die HotSpot-VM [55] konzentrieren sich im Wesentlichen auf häufig aufgerufene Methoden, wobei die minimale Aufrufhäufigkeit per Schwellwert definiert ist. Die Optimierungen umfassen unter anderem das Einflechten (engl. *inlining*) kurzer Methodenrümpfe und das Ausrollen von Schleifen.

3.4.7.1 Abgrenzung

JIT-Übersetzer können als eine Form der dynamischen Optimierung angesehen werden. Die Optimierung konzentriert sich auf die selektive Übersetzung von Zwischensprachen in performanteren Maschinencode. Im Unterschied zur vorliegenden Arbeit handelt es sich dabei jedoch nicht um Auto-Tuning, welches auf Laufzeitmessungen und der Modifikation von Tuningparametern basiert.

3.4.8 Zusammenfassung

Tabelle 3.3 fasst die Merkmale der Arbeiten zur Optimierung paralleler Programme zusammen:

- Ein Optimierungsansatz ist *domänenspezifisch*, wenn er auf spezielle Algorithmen, Anwendungs- oder Hardwaretypen zugeschnitten ist.

- Ein Ansatz erfordert *explizite Tuninginstruktionen*, wenn Tuninginformationen, z.B. Tuningparameter oder Hardwareanforderungen, explizit spezifiziert werden müssen.
- Sofern *Leistungsmessungen* unterstützt werden, können diese offline (Off) oder online (On) erfolgen, also vor oder während der Programmausführung.
- Die *Leistungsoptimierung* kann offline (Off) oder online (On) stattfinden.

Merkmal	**PetaBricks**	**Active Harmony**	**FIBER**	**Orio**	**Atune**	**Application Heartbeats**	**JIT-Übersetzung**	**diese Arbeit**
Domänenspezifisch	-	-	X	-	-	-	-	-
Explizite Tuninginstruktionen	X	X	X	X	X	X	-	-
Leistungsmessung	Off	Off/On	Off	Off	Off	On	-	On
Leistungsoptimierung	On	Off/On	Off/On	Off	Off	-	On	On

Tabelle 3.3: Vergleich der Arbeiten zur Optimierung paralleler Programme

Die vorgestellten Arbeiten aus dem Bereich der Leistungsoptimierung bzw. Auto-Tuning befassen sich sowohl mit algorithmenspezifischen als auch flexibleren Methoden zur Optimierung paralleler Programme, setzen aber eine Bereitstellung entsprechender Tuninginformationen durch den Programmierer voraus. Insofern kann der Anspruch der vorliegenden Arbeit, Tuninginformationen automatisiert durch den Übersetzer zu gewinnen, als ein Alleinstellungsmerkmal gelten.

PetaBricks kann als einer der wenigen Ansätze gelten, die Auto-Tuning als Bestandteil des Programmiermodells betrachten und somit ein ähnliches Ziel verfolgen wie die vorliegende Arbeit. Der Fokus von PetaBricks liegt jedoch auf der Auswahl algorithmischer Alternativen basierend auf zuvor gesammelten Laufzeitprofilen.

3.5 Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten vorgestellt und von der vorliegenden Arbeit abgegrenzt.

Zunächst wurden Arbeiten zur stromorientierten Programmierung besprochen. Stromsprachen besitzen gegenüber anderen parallelen Programmieransätzen den Vorteil, dass verschiedene

Arten von Parallelität auf kompakte Weise implementiert und kombiniert werden können, wobei von expliziter Synchronisierung abstrahiert wird. Die meisten Stromsprachen sind jedoch für Probleme der Grafik- und Signalverarbeitung oder spezielle Hardwarearchitekturen konzipiert; Optimierungen erfolgen meist statisch und basieren auf speziellem Domänenwissen, was die Flexibilität dieser Konzepte einschränkt.

Im Mittelpunkt nicht-stromorientierter paralleler Programmieransätze stehen meist Aufgaben- oder Datenparallelität, nicht jedoch Fließband- bzw. Stromverarbeitung, welche in vielen realen Anwendungen zum Einsatz kommt. Darüber hinaus erfordern viele dieser Sprachen und Bibliotheken die Verwendung expliziter Synchronisierungskonstrukte und bieten keine direkte Unterstützung für Auto-Tuning.

Schließlich wurden Arbeiten zur automatischen Performanzoptimierung vorgestellt. Im Unterschied zur vorliegenden Arbeit wird hierbei meist vorausgesetzt, dass Tuninginformationen für das zu optimierende Programm explizit spezifiziert sind. Active Harmony kann als einer der wenigen Ansätze gelten, die sowohl die Bestimmung als auch die Optimierung der Leistung einzelner Programmteile im Produktivbetrieb ermöglichen. Mit dem Konzept der Application Heartbeats existiert ein Ansatz, um die Gesamtleistung eines Programms im Produktivbetrieb zu erfassen, so dass diese Technik als Grundlage für die Leistungsmessung und -optimierung von Stromprogrammen dienen kann.

Nur wenige Arbeiten haben sich bisher damit befasst, das Potenzial der Stromprogrammierung für ein breiteres Anwendungsgebiet zugänglich zu machen. An diesem Punkt setzt die vorliegende Arbeit an, indem sie

1. Stromprogrammierung und Objektorientierung kombiniert,
2. Programmanalysen konzipiert, welche zur Programmkorrektheit beitragen, sowie
3. mit Laufzeit-Tuning ein flexibles, anwendungs- und plattformneutrales Optimierungskonzept für Stromprogramme vorstellt.

Kapitel 4

Sprachkonzepte zur objektorientierten Stromprogrammierung

Dieses Kapitel behandelt Sprachkonzepte zur objektorientierten Stromprogrammierung. Wir erläutern und diskutieren unseren Sprachentwurf und stellen Programmanalysen vor, um Filterzustände und Zugriffe auf gemeinsame Variablen zu identifizieren. Ergänzend wird aufgezeigt, wie zentrale parallele Muster mithilfe strombasierter Konzepte implementiert werden können.

Vorteile von Spracherweiterungen

Die vorliegende Arbeit verfolgt den Ansatz einer Spracherweiterung; als Alternative käme ein Bibliotheksansatz in Betracht. Im Folgenden nennen wir allgemeine Vorteile von Spracherweiterungen gegenüber Bibliotheken.

Spracherweiterungen erlauben die Definition neuer Syntax, welche für einen klar spezifizierten Zweck präzise zugeschnitten und einfach zu verwenden ist. Bibliotheken greifen dagegen auf bestehende Sprachkonstrukte zurück, deren Verwendung für den angestrebten Einsatzzweck möglicherweise weniger intuitiv ist und daher fehleranfälliger sein kann. Zudem bieten Bibliotheken nicht dasselbe Maß an Abstraktion und Kompaktheit wie Spracherweiterungen.

Ein weiterer Vorteil von Spracherweiterungen liegt darin, dass die Semantik der Sprachkonstrukte dem Übersetzer bekannt ist. Diese Informationen können bei der semantischen Analyse, der Codegenerierung sowie für Programmoptimierungen verwendet werden. Die Semantik von Bibliotheksfunktionen ist dem Übersetzer dagegen nicht bekannt.

4.1 Sprachentwurf

Ausgangspunkt der Arbeit ist das objektorientierte Programmiermodell. Hierfür spezifizieren wir Spracherweiterungen zur Implementierung von Filtern und Stromgraphen. Wir stellen die Konzepte des objektorientierten Filters, des parallelen Ausdrucks und der parallelen Anweisung vor, welche als Member, Ausdruck bzw. Anweisung in eine objektorientierte Sprache integriert werden können. Die Konzepte wurden als Erweiterung der Programmiersprache Java [42] umgesetzt, so dass wir uns im Folgenden auf diese Sprache beziehen. Grundsätzlich sind die Prinzipien hinreichend allgemeingültig, dass sie sich auf beliebige objektorientierte Sprachen übertragen lassen. Eine prototypische Sprachbeschreibung sowie die Grundzüge der Übersetzung und Ausführung konnte bereits veröffentlicht werden [81, 82].

Unser Sprachentwurf erweitert eine objektorientierte Sprache um neue Member-, Ausdrucks- und Anweisungstypen:

$$
\begin{array}{rcl}
\mathit{Member} & \rightarrow & \ldots \mid \mathit{FilterDecl} \\
\mathit{Expr} & \rightarrow & \ldots \mid \mathit{ParallelExpr} \\
\mathit{Stmt} & \rightarrow & \ldots \mid \mathit{ParallelStmt} \mid \mathit{PushStmt} \mid \mathit{AddFilterStmt}
\end{array}
$$

Der neue Membertyp *FilterDecl* dient dazu, Filter in Klassen oder Schnittstellen deklarieren zu können. Stromgraphen werden mithilfe der neuen Ausdrucks- und Anweisungstypen erzeugt. Zusätzlich führen wir neue Schlüsselwörter und Operatoren ein, deren Verwendung und Semantik im Laufe des Kapitels erläutert werden.

4.1.1 Filter

Ein Filter transformiert einen Strom von Eingabeelementen in einen Strom von Ausgabeelementen. Um Elemente empfangen und verschicken zu können, besitzt ein Filter jeweils einen typisierten Eingabe- und Ausgabekanal. Zusätzliche Verwaltungsinformationen werden zum Zwecke der Ausführung und Leistungsoptimierung gesammelt. Der Programmierer beschreibt lediglich die Verarbeitung eines Stromelements durch eine Folge von Anweisungen.

Ein Filter ähnelt syntaktisch und funktional einer Methode. Um den stromverarbeitenden Charakter explizit herauszustellen, führen wir hierfür einen eigenen Membertyp ein. Ein Filter wird in Klassen oder Schnittstellen deklariert und besteht aus einem Kopf (engl. *header*) und einem Rumpf (engl. *body*):

FilterDecl → *FilterHeader* *FilterBody*
FilterHeader → *Modifier* $Type_{in}$ `=>` $Type_{out}$
Identifier `(` *ParamList* `)` *Exceptions*
FilterBody → `;` | *MethodBody*
| `{` *StmtList* *WorkBlock* *StmtList* `}`
WorkBlock → `work` `(` *FormalDecl* `)` *Block*

4.1.1.1 Kopf

Die zulässigen Modifikatoren (engl. *modifier*) eines Filters entsprechen im Wesentlichen denen von Methoden; ausgenommen ist der Modifikator `synchronized`. Auf diese Weise lassen sich unterschiedliche Sichtbarkeiten oder auch Instanz- und Klassenfilter definieren.

Mit $Type_{in}$ und $Type_{out}$ werden die Typen der Elemente des Eingabe- bzw. Ausgabestroms festgelegt. Im Unterschied zu klassischen Stromsprachen beschränken sich diese Typen somit nicht auf primitive Datentypen, sondern erlauben auch beliebige Referenztypen. Ebenfalls zulässig ist der leere Typ (engl. *void*), wenn ein Filter keinen Eingabestrom empfängt bzw. Ausgabestrom generiert. Auf diese Weise lassen sich Quellen bzw. Senken definieren.

Die Signatur eines Filters besteht also aus Eingabe- und Ausgabetyp, dem Namen des Filters und optional einer Liste von Eingabeparametern. Zusätzlich können Ausnahmen (engl. *exceptions*) deklariert werden, die bei der Ausführung des Filters ausgelöst werden können.

4.1.1.2 Rumpf

Sofern ein Filter nicht `abstract` deklariert ist, besitzt er einen Rumpf. Der Rumpf beschreibt, wie der Filter seine Eingabe- in Ausgabeströme umwandelt. Hierbei unterscheiden wir periodische und nichtperiodische Filter.

Periodische Filter *Periodische Filter* besitzen einen Rumpf, der genau einen work-Block definiert. Der work-Block entspricht einer Schleife, welche die Bearbeitung eines jeden empfangenen Stromelements festlegt; diese Bearbeitungsvorschrift nennen wir im Folgenden *Aktion*. Empfängt ein periodischer Filter einen Eingabestrom der Länge n, führt er also im Laufe der Programmausführung n Aktionen durch. Da periodische Filter einen Eingabestrom erfordern, können sie nicht als Quelle fungieren; der Eingabetyp `void` ist also unzulässig.

Hinter dem Schlüsselwort `work` wird ein formaler Parameter deklariert, dessen Bezeichner das aktuelle Stromelement repräsentiert. Der Typ dieses Parameters muss mindestens dem Eingabetyp des Filters entsprechen. Der daran anschließende Block legt fest, welche Anweisungen für jede Aktion bzw. jedes Stromelement ausgeführt werden.

Optional können vor und nach dem work-Block beliebige Anweisungen stehen, beispielsweise um Filtereinstellungen zu initialisieren oder abschließende Berechnungen durchzuführen. Diese Anweisungen werden unabhängig von der Stromlänge nur einmalig ausgeführt.

■ SEMANTISCHE REGELN FÜR WORK-BLÖCKE:

1. Ein work-Block darf nur in Filtern verwendet werden, deren Eingabetyp nicht `void` ist.

2. Der Parameter des work-Blocks muss denselben Typ besitzen wie der Eingabetyp des Filters oder ein Obertyp sein.

Nichtperiodische Filter Im Unterschied zu periodischen Filtern besitzen *nichtperiodische Filter* keinen work-Block, der die Bearbeitung eines Eingabestroms festlegt. Nichtperiodische Filter werden entweder als Quelle oder zur Schachtelung von Parallelität verwendet. Die Schachtelung von Parallelität ist Gegenstand von Abschnitt 4.1.2.6, da hierfür zunächst parallele Ausdrücke eingeführt werden müssen (vgl. Abschnitt 4.1.2).

4.1.1.3 Erzeugen von Ausgabeströmen

Sowohl periodische als auch nichtperiodische Filter können Ausgabeströme erzeugen. Ausgabeströme entstehen, indem Objekte oder Werte in den Ausgabekanal eines Filters gelegt werden; dies erfolgt mithilfe einer push-Anweisung. Eine push-Anweisung besitzt die Form

PushStmt → `push` *Expr* ;

und kann überall im Rumpf sowohl periodischer als auch nichtperiodischer Filter verwendet werden, sofern der Ausgabetyp nicht `void` ist. Der Wert dieses Ausdrucks wird in den Ausgabekanal des Filters gelegt.

■ SEMANTISCHE REGELN FÜR PUSH-ANWEISUNGEN:

1. push-Anweisungen dürfen nur im Rumpf solcher Filter verwendet werden, deren Ausgabetyp nicht `void` ist.

2. Das push-Argument *Expr* muss denselben Typ besitzen wie der Ausgabetyp des Filters oder ein Subtyp sein. Als push-Argument darf somit kein paralleler Ausdruck verwendet werden.

Listing 4.1: Deklaration von Filtern

```
1  public class Indexer {
2    public Iterator<File> fileIterator(File rootDir) { ... }
3    public Set<String> getWordSet(File f) throws IOException { ... }
4    public void updateIndex(Set<String> s) { ... }
5
6    public void => File visitFiles(File rootDir) {
7      for (Iterator<File> i = fileIterator(rootDir); i.hasNext(); ) {
8        push i.next();
9      }
10   }
11
12   public File => Set<String> index() {
13     work (File f) {
14       try {
15         push getWordSet(f);
16       } catch (IOException e) {
17         System.err.println("Could not open file " + f);
18       }
19     }
20   }
21
22   public Set<String> => void update() {
23     work (Set<String> s) {
24       updateIndex(s);
25     }
26   }
27 }
```

4.1.1.4 Beispiel

Zur Veranschaulichung zeigt Listing 4.1 eine Klasse `Indexer` für die Indizierung von Dateien. Diese Klasse deklariert drei Filter `visitFiles`, `index`, `update` sowie drei Methoden `fileIterator`, `getWordSet`, `updateIndex`.

Der Filter `visitFiles` erzeugt einen Strom von Dateien in Form von `File` Objekten, welche in einem durch den Parameter `rootDir` übergebenen Wurzelverzeichnis enthalten sind. Da kein Eingabestrom erwartet bzw. verarbeitet wird, ist der Eingabetyp `void`, und es existiert kein work-Block im Rumpf des Filters. Der Ausgabetyp ist dagegen `File`; in jeder Schleifeniteration wird ein `File` Objekt mithilfe der push-Anweisung in den Ausgabestrom des Filters gelegt.

Der Filter `index` empfängt gemäß seines Eingabe- und Ausgabetyps einen Strom von Elementen des Typs `File` und erzeugt einen Strom von Elementen des Typs `Set<String>`. Das in einem Aktionsschritt empfangene `File` Objekt wird der lokalen Variable `f` zugewiesen, die

im Anschluss an das Schlüsselwort `work` deklariert wird. Mithilfe der Methode `getWordSet` wird für jede Datei eine Wortmenge erzeugt, welche durch die push-Anweisung in den Ausgabestrom gelegt wird.

Der Filter `update` empfängt schließlich einen Strom von Elementen des Typs `Set<String>`, generiert aber keinen Ausgabestrom. Dementsprechend ist der Eingabetyp `Set<String>`, der Ausgabetyp `void`. Durch letzteren ist festgelegt, dass innerhalb des Filters keine push-Anweisung verwendet wird. Die im work-Block aufgerufene Methode `updateIndex` fügt `s` einer globalen Indexstruktur hinzu.

4.1.1.5 Aufrufen und Beenden von Filtern

Aufrufen Filter werden nach demselben Prinzip aufgerufen wie Methoden. Der Aufruf eines Instanzfilters erfolgt über einen Objektbezeichner, der Aufruf eines Klassenfilters über den Klassenbezeichner. Wie ein Methodenaufruf wird ein Filteraufruf zur Laufzeit aufgelöst (dynamische Bindung). Die durch den Aufruf erzeugte Filterinstanz wird an einen Ablaufplaner bzw. Ausführungsdienst übergeben (vgl. Kapitel 5).

Reguläres Beenden Ein Filter f wird beendet, wenn das Ende des Stroms erreicht und die letzte Anweisung des Filterrumpfes ausgeführt wurde. Das Ende des Stroms ist erreicht, wenn alle Vorgängerfilter von f beendet sind. Sofern es sich bei f nicht um eine Senke handelt, wird sein Ausgabekanal mit einer Marke versehen, welche die nachfolgenden Filter über das Stromende informiert. Auf diese Weise werden alle Filter sukzessive beendet und das Stromprogramm terminiert.

Vorzeitiges Beenden Ein Filter kann durch eine normale return-Anweisung oder das Auslösen einer Ausnahme vorzeitig beendet werden. Die Behandlung von Ausnahmen wird in Abschnitt 4.1.1.7 separat thematisiert. Im Falle einer return-Anweisung wird der Ausgabekanal des Filters markiert, um nachfolgende Filter zu informieren und sukzessive zu beenden.

Eine return-Anweisung besitzt entweder kein Argument oder ein Argument, dessen Typ dem Ausgabetyp des Filters entspricht. Die folgenden zwei Codefragmente sind dabei äquivalent:

- `push elem; return;`
- `return elem;`

Das vorzeitige Beenden eines Filters durch eine return-Anweisung kann in solchen Fällen sinnvoll sein, wenn nur ein ganz bestimmtes Element von Interesse ist, welches eine weitere Stromverarbeitung überflüssig macht. Beispielsweise könnte ein Filter einen Strom von Näherungslösungen für ein bestimmtes Problem berechnen, während ein nachfolgender Filter die

Qualität dieser Lösungen ermittelt, um beim Erreichen einer Mindestqualität die Berechnung zu stoppen.

■ SEMANTISCHE REGELN FÜR RETURN-ANWEISUNGEN IN FILTERN:

1. Auf eine return-Anweisung dürfen keine weiteren Anweisungen folgen.
2. Falls eine return-Anweisung ein Argument besitzt, so muss das Argument denselben Typ besitzen wie der Ausgabetyp des Filters oder ein Subtyp sein.

4.1.1.6 Zustandslose und zustandsbehaftete Filter

Je nach Struktur des Rumpfes bzw. work-Blocks können zustandslose und zustandsbehaftete Filter unterschieden werden:

- Ein work-Block und damit der zugehörige Filter ist *zustandslos*, wenn zwischen den Aktionen keine lokalen Datenabhängigkeiten existieren.
- Andernfalls ist ein work-Block und damit der zugehörige Filter *zustandsbehaftet*. In diesem Fall existiert mindestens eine Variable, die außerhalb des work-Blocks deklariert ist und einen Zustand speichert. Wird innerhalb des work-Blocks erst lesend, dann schreibend auf diese Variable zugegriffen wird, so liegt zwischen zwei Aktionen eine Schreib-Lese-Abhängigkeit und somit ein Zustand vor.

Die Kenntnis der Zustandslosigkeit ist wichtig, um einen Filter korrekt zu replizieren, d.h. mehrere Instanzen dieses Filters parallel ausführen zu können. Zu diesem Zweck werden Programmanalysen durchgeführt, die in Abschnitt 4.3.2.2 erläutert werden.

4.1.1.7 Behandlung von Ausnahmen

Der Code, der von einem Filter ausgeführt wird, kann Ausnahmen (engl. *exceptions*) auslösen. Es bestehen zwei Möglichkeiten, Ausnahmen zu behandeln:

- *try-catch-Block.* Der Filterrumpf kann die auftretenden Ausnahmen durch Verwendung eines try-catch-Blocks selbst abfangen und behandeln. Auf diese Weise kann erreicht werden, dass die Ausnahme für andere Filter nicht sichtbar ist und die Stromverarbeitung fortgesetzt werden kann.
- *throws-Klausel.* Der Filter kann eine entsprechende throws-Klausel deklarieren. Wird eine Ausnahme ausgelöst, wird das Ausnahmeobjekt gemäß dem Aufrufstapel an den nächsten Filter bzw. die nächste Methode weitergeleitet. Der Filter, der die aufgetretene Ausnahme weitergeleitet hat, wird vorzeitig beendet. Zudem fügt dieser Filter eine Ausnahmemarke in seinen Eingabe- und Ausgabekanal ein, so dass Vorgänger- und

Nachfolgefilter nach dem in Abschnitt 4.1.1.5 beschriebenen Prinzip sukzessive beendet werden.

■ SEMANTISCHE REGEL FÜR DIE BEHANDLUNG VON AUSNAHMEN
Kann innerhalb des Filterrumpfes eine Ausnahme ausgelöst werden, so muss diese entweder durch einen try-catch-Block abgefangen werden oder in der throws-Klausel des Filters deklariert werden.

4.1.1.8 Überladen und Überschreiben von Filtern

Filter können überladen und überschrieben werden. Ein Filter kann einen anderen Filter überladen, indem er denselben Bezeichner, aber eine unterschiedliche Signatur besitzt. Beim Überschreiben eines Filters bleibt die Signatur dagegen unverändert; zudem gilt das Substitutionsprinzip, nach dem Objekte einer Klasse durch Objekte einer ihrer Unterklassen „ersetzbar“ sein müssen.

■ SEMANTISCHE REGELN FÜR DAS ÜBERSCHREIBEN VON FILTERN:

1. Die Signatur muss dieselbe sein.
2. Die deklarierten Ausnahmen müssen dieselben sein oder Subklassen der deklarierten Ausnahmen des überschriebenen Filters.
3. Der Eingabetyp muss derselbe sein oder ein Obertyp (Kontravarianz).
4. Der Ausgabetyp muss derselbe sein oder ein Subtyp (Kovarianz).
5. Die Sichtbarkeit darf nicht reduziert werden.

Listing 4.2: Überladen und Überschreiben von Filtern

```
1  class X { ... }
2  class Y extends X { ... }
3  class Z extends Y { ... }
4
5  class C {
6    Y => Y f(X x) { ... }
7  }
8
9  class D extends C {
10   X => Z f(X x) { ... }    // überschreibt f in C
11   Y => Y f(Y y) { ... }    // überlädt f
12 }
```

Listing 4.2 zeigt eine Klasse `C` mit einer Unterklasse `D`. Der erste Filter `f` in `D` überschreibt den von `C` geerbten Filter `f`; der zweite Filter überlädt `f`, da der Parametertyp und somit die Signatur verschieden ist.

4.1.1.9 Entwurfsdiskussion

An dieser Stelle diskutieren wir die zentralen Entwurfsentscheidungen, die wir bei der Definition eines objektorientierten Filters zugrunde gelegt haben. Der wesentliche Grund für die Konzeption eines neuen Membertyps für Filter liegt darin, eine syntaktisch saubere Trennung zwischen Filtern und Methoden zu erreichen.

Eine denkbare Alternative wäre es, zwischen Methoden und Filtern syntaktisch nicht zu unterscheiden. So könnten Eingabeströme über Methodenparameter empfangen und Ausgabeströme über return-Anweisungen der Methode erzeugt werden, wobei zur Kennzeichnung von Stromvariablen ein spezieller Modifikator verwendet werden könnte. Unsere Syntax vollzieht eine klare Trennung zwischen Aufrufparametern und Datenströmen und entspricht somit einer intuitiven Darstellung eines Filters, wie er in Kapitel 2 definiert wurde. Die Eingabe- und Ausgabetypen der Ströme sind explizit hervorgehoben und verschaffen dem Programmierer mehr Klarheit.

Das Zusammenspiel von work-Block und push-Anweisung ermöglicht es, die Verarbeitung von Stromelementen intuitiv und unabhängig von Vorgänger- und Nachfolgefiltern zu beschreiben. Damit lassen sich Filter deklarieren, die gleichviele, weniger oder mehr Stromelemente produzieren als konsumieren. Der work-Block stellt zudem eine klare Trennung der Initialisierungs-, Verarbeitungs- und Aufräumphase eines Filters her. Des Weiteren abstrahieren work-Block und push-Anweisung vom konkreten Verhalten der Eingabe- und Ausgabekanäle. Zwar erhält der Programmierer damit keine feingranularen Kontrollmöglichkeiten; andererseits würde gerade dies dem Abstraktionsanspruch der Stromprogrammierung widersprechen. Weiterhin müssten diese Einstellungen ggf. für einen bestimmten Einsatzkontext, d.h. abhängig vom Verhalten der Vorgänger- und Nachfolgefilter, vorgenommen werden. In diesem Fall wäre ein Filter nicht mehr eine unabhängige, flexibel einsetzbare Komponente.

Da zwischen Filtern und Methoden syntaktisch, also statisch, unterschieden werden kann, ist sichergestellt, dass Filter und Methoden sich nicht überschreiben können. In Klassen und Schnittstellen kann eindeutig spezifiziert werden, ob es sich bei einem Member um einen Filter oder eine Methode handelt. Bei fehlender Unterscheidung könnte durch die daraus resultierende Unschärfe zusätzliches Fehlerpotenzial entstehen.

4.1.2 Objektorientierte Stromgraphen

In Kapitel 2 wurde ein Stromgraph als gerichteter Graph $G = (V, E)$ definiert, wobei V eine Menge von Filtern und $E \subset V \times V$ eine Menge von Verbindungen (Kanälen) zwischen den Filtern ist.

Um Stromgraphen adäquat beschreiben zu können, führen wir in diesem Kapitel das Konzept paralleler Anweisungen bzw. Ausdrücke ein:

$$
\begin{array}{lcl}
\textit{ParStmt} & \rightarrow & \textit{ParExpr}\ ; \mid \textit{ParBlock} \\
\textit{ParExpr} & \rightarrow & \textit{PipeExpr} \mid \textit{TaskParExpr} \mid \textit{AlternativeExpr} \mid \textit{FilterCall} \\
\textit{ParBlock} & \rightarrow & \textit{PipeBlock} \mid \textit{TaskParBlock}
\end{array}
$$

Parallele Ausdrücke bestehen aus Filteraufrufen, die durch Operatoren verknüpft sind. Wir definieren Operatoren für Fließband-, Aufgaben- und Datenparallelität, welche die Grundbausteine azyklischer Stromgraphen darstellen, sowie einen Operator zur Formulierung von Alternativen. Durch das Konzept des Teleports schaffen wir darüber hinaus eine Möglichkeit, beliebige Verbindungen und damit allgemeine, insbesondere zyklische, Stromgraphen zu beschreiben.

4.1.2.1 Fließbandparallelität

Ein Fließband ist eine Kette von Filtern $f_1, ..., f_n$, wobei zwischen zwei miteinander verbundenen Filter eine Produzenten-Konsumenten-Beziehung vorliegt. Ein Filter f_i konsumiert Daten, die vom vorangehenden Filter f_{i-1} produziert werden, und produziert Daten, die vom nachfolgenden Filter f_{i+1} konsumiert werden. Fließbandparallelität wird durch Fließbandausdrücke erzeugt:

$$
\textit{PipeExpr} \rightarrow \textit{FilterCall}\ \texttt{=>}\ \textit{FilterCall} \mid \textit{FilterCall}\ \texttt{=>}\ \textit{PipeExpr}
$$

Betrachten wir die Filter `visitFiles`, `index` und `update` aus Listing 4.1, so lassen sich diese zu einem parallelen Ausdruck

```
idx.visitFiles(dir) => idx.index() => idx.update()
```

kombinieren, wobei `idx` ein Objekt des Typs `Indexer` sei. Der Ausdruck erzeugt ein dreistufiges Fließband, welches alle Dateien eines Verzeichnisses `dir` ermittelt, deren Wortmengen berechnet und einen globalen Index aktualisiert. Jeder Filteraufruf stellt eine potenziell nebenläufige Aktivität dar. Abbildung 4.1 skizziert den entsprechenden Stromgraphen.

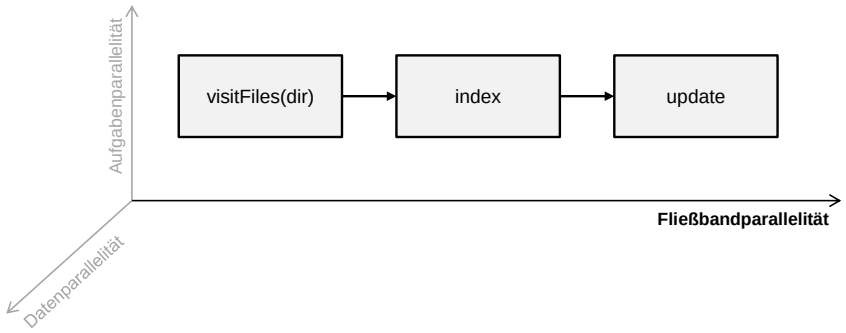


Abbildung 4.1: Fließbandparallelität

■ SEMANTISCHE REGELN FÜR FLIESSBANDANWEISUNGEN:

1. Eine Fließbandanweisung bestehe aus n Filtern $f_1, ..., f_n$, wobei f_i und f_{i+1} durch den Operator `=>` miteinander verbunden seien ($1 \leq i < n$). Dann muss der Ausgabetyp von f_i derselbe sein wie der Eingabetyp von f_{i+1} oder ein Subtyp.

2. Falls eine Fließbandanweisung nicht im Rumpf eines Filters verwendet wird, müssen der Eingabetyp des ersten Filters f_1 sowie der Ausgabetyp des letzten Filters f_n `void` sein.

3. Falls eine Fließbandanweisung im Rumpf eines Filters verwendet wird, handelt es sich um verschachtelte Parallelität und es gelten die dafür definierten semantischen Regeln (vgl. Abschnitt 4.1.2.6).

4.1.2.2 Aufgabenparallelität

Zwei Filter f_1, f_2 sind aufgabenparallel, wenn sie sich auf unterschiedlichen Pfaden des Stromgraphen befinden. Zwischen diesen Filtern besteht also keine Produzenten-Konsumenten-Abhängigkeit; sie können jedoch gemeinsame Vorgänger- oder Nachfolgerfilter besitzen. Aufgabenparallelität wird durch aufgabenparallele Ausdrücke erzeugt:

$$\begin{aligned} \textit{TaskParExpr} \quad \rightarrow \quad & \textit{FilterCall} \;\; \texttt{|||} \;\; \textit{FilterCall} \\ & \mid \; \textit{FilterCall} \;\; \texttt{|||} \;\; \textit{TaskParExpr} \end{aligned}$$

Die Filter, aus denen dieser Ausdruck besteht, weisen im Unterschied zur Fließbandparallelität keinerlei Eingabe-Ausgabe-Abhängigkeiten auf, sind also funktional unabhängig voneinander. Sollen beispielsweise mithilfe des Filters `visitFiles` aus Listing 4.1 Datenströme für zwei verschiedene Verzeichnisse `dir1` und `dir2` erzeugt werden, so kann dies durch einen aufgabenparallelen Ausdruck

```
visitFiles(dir1) ||| visitFiles(dir2)
```

umgesetzt werden. Abbildung 4.2 veranschaulicht den entsprechenden Teil des Stromgraphen.

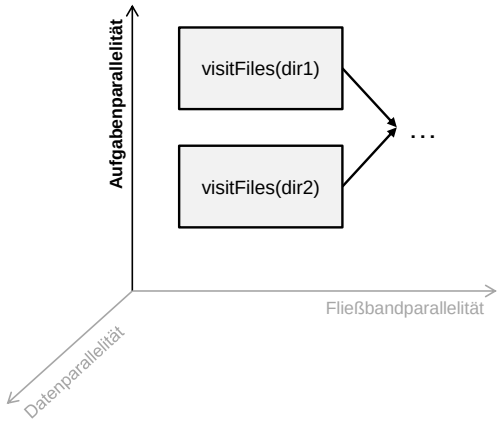


Abbildung 4.2: Aufgabenparallelität

■ SEMANTISCHE REGELN FÜR AUFGABENPARALLELE ANWEISUNGEN:

1. Eine aufgabenparallele Anweisung bestehe aus n Filtern $f_1, ..., f_n$. Dann müssen folgende Bedingungen gelten: (1) Entweder jedes f_i oder kein f_i besitzt den Eingabetyp `void`. (2) Entweder jedes f_i oder kein f_i besitzt den Ausgabetyp `void`.
2. Falls eine aufgabenparallele Anweisung nicht im Rumpf eines Filters verwendet wird, müssen sowohl Eingabe- als auch Ausgabetyp jedes Filters `void` sein.
3. Falls eine aufgabenparallele Anweisung im Rumpf eines Filters verwendet wird, handelt es sich um verschachtelte Parallelität und es gelten die dafür definierten semantischen Regeln (vgl. Abschnitt 4.1.2.6).

4.1.2.3 Definition von Alternativen

Alternativen sind eine Menge von Filtern, welche unterschiedliche Implementierungen zur Lösung desselben Problems bieten. Ein Beispiel wären Filter, welche unterschiedliche Sortieralgorithmen kapseln. Alternativen werden durch Alternativenausdrücke beschrieben:

$$\begin{aligned} \textit{AlternativeExpr} \quad \rightarrow \quad & \textit{FilterCall} \;\texttt{??}\; \textit{FilterCall} \\ & \mid \textit{FilterCall} \;\texttt{??}\; \textit{AlternativeExpr} \end{aligned}$$

Beispielsweise gibt der Ausdruck

```
mergesort() ?? quicksort() ?? bubblesort()
```

drei alternative Sortieralgorithmen bzw. -filter an, von denen einer auszuwählen ist. Der entsprechende Teil des Stromgraphen ist in Abbildung 4.3 dargestellt; die Alternativen können als Spezialform von Aufgabenparallelität begriffen werden.

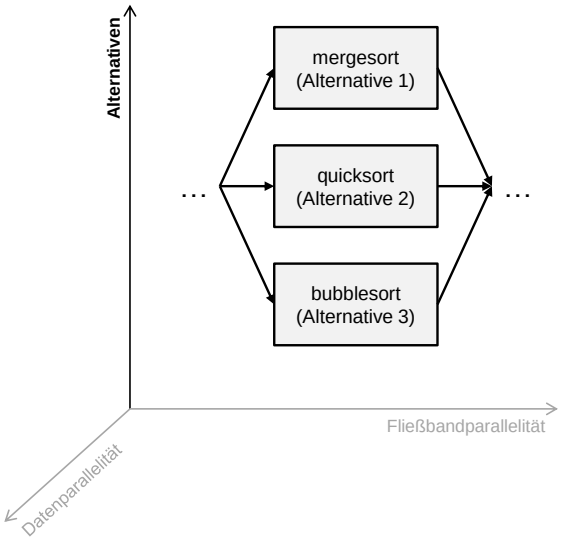


Abbildung 4.3: Alternativen

■ SEMANTISCHE REGELN FÜR ALTERNATIVENANWEISUNGEN:

1. Eine Alternativenanweisung bestehe aus n Filtern $f_1, ..., f_n$. Dann müssen folgende Bedingungen gelten: (1) Entweder jedes f_i oder kein f_i besitzt den Eingabetyp `void`. (2) Entweder jedes f_i oder kein f_i besitzt den Ausgabetyp `void`.
2. Falls eine Alternativenanweisung nicht im Rumpf eines Filters verwendet wird, müssen sowohl Eingabe- als auch Ausgabetyp jedes Filters `void` sein.
3. Falls eine Alternativenanweisung im Rumpf eines Filters verwendet wird, handelt es sich um verschachtelte Parallelität und es gelten die dafür definierten semantischen Regeln (vgl. Abschnitt 4.1.2.6).

4.1.2.4 Datenparallelität durch Filterreplikation

Analog zu Methoden wird beim Aufruf eines Filters eine Filterinstanz erzeugt. Um Datenparallelität auszunutzen, kann ein Filter repliziert werden, so dass der Datenstrom an dieser Stelle von mehreren Instanzen desselben Filters parallel bearbeitet wird. Dies setzt voraus, dass der Filter keinen inneren Zustand besitzt, durch den Datenabhängigkeiten zwischen den Aktionen entstehen.

In der objektorientierten Stromprogrammierung können Filteraufrufe mit Operatoren kombiniert werden, um die Ausnutzung von Datenparallelität zu erlauben. Die Voraussetzung hierfür, dass der Filter zustandslos ist, wird durch die Zustandsanalyse geprüft (vgl. Abschnitt 4.3.1); bei Nichterfüllung wird eine Übersetzerwarnung ausgegeben. Hintergrund dieser Entwurfsentscheidung ist die Tatsache, dass die Zustandsanalyse keine absolute Genauigkeit garantiert und daher lediglich unterstützenden Charakter besitzen, nicht aber die endgültige Entscheidung fällen soll. Zudem ist durch die Operatoren dem Programmierer die Möglichkeit gegeben, den Grad der Parallelität entweder selbst einzustellen oder dies einem Auto-Tuner zu überlassen.

Wir erweitern Filteraufrufe daher nach folgendem Schema:

$$
\begin{array}{rcl}
\textit{FilterCall} & \rightarrow & \textit{SimpleFilterCall} \mid \textit{ExtFilterCall} \\
\textit{ExtFilterCall} & \rightarrow & \textit{SimpleFilterCall}\ \texttt{:}\ \texttt{[}\ \textit{Expr}\ \texttt{]} \\
 & & \mid\ \textit{SimpleFilterCall}\ \texttt{:}\ \texttt{[+]} \\
 & & \mid\ \textit{SimpleFilterCall}\ \texttt{+}
\end{array}
$$

Ein *SimpleFilterCall* ist ein einfacher Filteraufruf, der syntaktisch identisch mit einem Methodenaufruf ist. Ein einfacher Filteraufruf resultiert in einer einzigen Filterinstanz und ist daher nicht datenparallel. Um mehrere Filterinstanzen und somit Datenparallelität zu ermöglichen, können Filteraufrufe um verschiedene Suffixe erweitert werden:

- `t():[n]`
  Es werden *n* Instanzen des Filters `t` erzeugt. Die Anzahl der Instanzen wird also vom Programmierer festgelegt und ist konstant. Beispielsweise ist der Ausdruck `t():[3]` semantisch äquivalent zu `t() ||| t() ||| t()`.
- `t():[+]`
  Die Anzahl der Filterinstanzen wird nicht vom Programmierer, sondern heuristikbasiert festgelegt und ist ebenfalls konstant.
- `t()+`
  Die Anzahl der Filterinstanzen wird zunächst per Heuristik festgelegt. Allerdings ist die Anzahl nicht zwingend konstant, sondern kann durch einen Auto-Tuner angepasst werden (vgl. Abschnitt 5.3).

Das zuvor verwendete Fließband zur Dateiindizierung könnte auf diese Weise Datenparallelität ausnutzen. Beispielsweise könnte der Filter `index` wie folgt repliziert werden:

```
idx.visitFiles(dir) => idx.index():[4] => idx.update()
idx.visitFiles(dir) => idx.index():[(n+1)*2] => idx.update()
idx.visitFiles(dir) => idx.index():[+] => idx.update()
idx.visitFiles(dir) => idx.index()+ => idx.update()
```

Abbildung 4.4 skizziert den resultierenden Stromgraphen als Variante des Stromgraphen aus Abbildung 4.1.

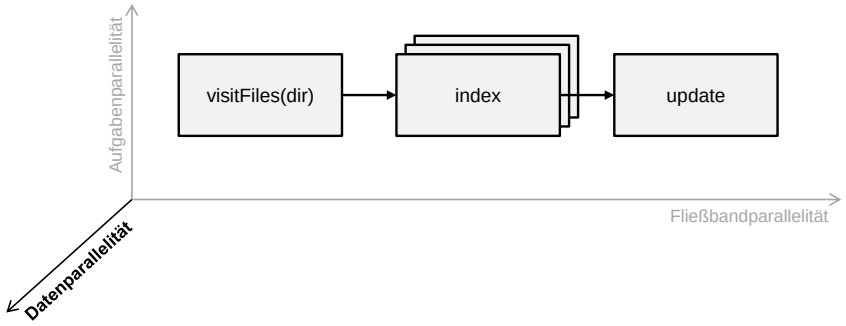


Abbildung 4.4: Datenparallelität durch Filterreplikation

■ SEMANTISCHE REGELN FÜR ERWEITERTE FILTERAUFRUFE:

1. Ein erweiterter Filteraufruf muss zustandslos sein.[1]
2. Ist die Anzahl der Filterinstanzen durch einen Ausdruck (*Expr*) festgelegt, so muss dieser Ausdruck vom Typ `int` sein.

4.1.2.5 Parallele Blöcke

Mithilfe von Fließbandausdrücken und aufgabenparallelen Ausdrücken lassen sich Fließbänder und Aufgabenparallelität statisch beschreiben. In manchen Fällen kann es jedoch sinnvoll sein, die Struktur bzw. die Anzahl der Filter dynamisch festzulegen. Dies kann mithilfe paralleler Blöcke erreicht werden. Ein paralleler Block entspricht einer parallelen Anweisung, der mithilfe von addfilter-Anweisungen Filter hinzugefügt werden. addfilter-Anweisungen besitzen die Form

AddFilterStmt → `addfilter` *FilterCall* ;

und dürfen ausschließlich in parallelen Blöcken verwendet werden. Ein paralleler Block ist entweder ein Fließbandblock *PipeBlock* oder ein aufgabenparalleler Block *TaskParBlock*; diese besitzen die Form

[1] Ist die automatische Zustandsprüfung positiv, so wird jedoch statt eines Übersetzungsfehlers eine Übersetzerwarnung ausgegeben. Hintergrund ist die Tatsache, dass auch harmlose Zustandsvariablen wie z.B. lokale Zähler existieren, welche die Replikation eines Filters nicht ausschließen. Die Zustandsanalyse kennt jedoch nicht die Variablensemantik und kann dementsprechend Falschmeldungen verursachen. Die Analyse unterstützt den Programmierer, ist aber nicht Entscheidungsträger.

$$\begin{array}{rcl} \textit{PipeBlock} & \rightarrow & \texttt{=>}\ \textit{Block} \\ \textit{TaskParBlock} & \rightarrow & \texttt{|||}\ \textit{Block} \end{array}$$

Beispielsweise erzeugt folgender Fließbandblock eine dynamische Anzahl von Fließbandstufen:

```
=> {
  for (int i = 0; i < n; i++) addfilter f1();
  if (cond) addfilter f2();
}
```

Der daraus resultierende Stromgraph ist in Abbildung 4.5 dargestellt.

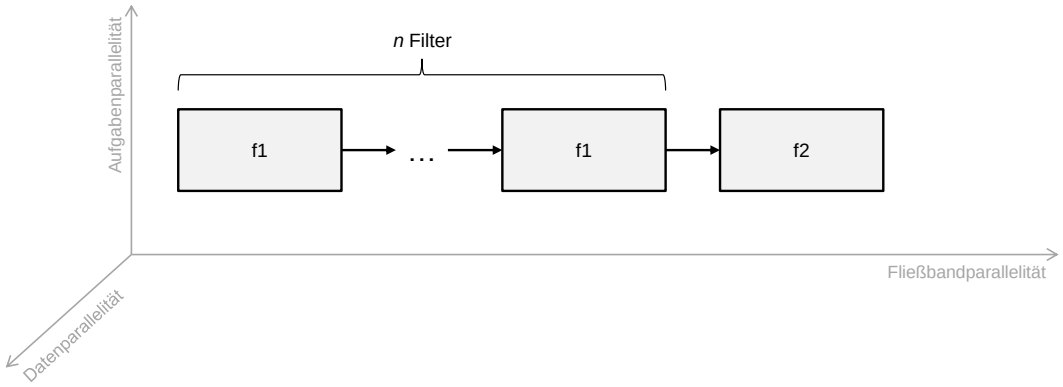


Abbildung 4.5: Dynamische Fließbandparallelität

■ SEMANTISCHE REGELN FÜR ADDFILTER-ANWEISUNGEN UND PARALLELE BLÖCKE:

1. Eine addfilter-Anweisung darf ausschließlich innerhalb eines parallelen Blocks verwendet werden.
2. In einem Fließbandblock müssen alle Filter denselben Eingabe- und denselben Ausgabetyp besitzen, wobei der Typ `void` nicht zulässig ist. Der Ausgabetyp muss gleich dem Eingabetyp sein oder ein Subtyp.
3. Ein Fließbandblock kann daher ausschließlich im Rumpf eines Filters verwendet werden. In diesem Fall handelt es sich um verschachtelte Parallelität und es gelten die dafür definierten semantischen Regeln (vgl. Abschnitt 4.1.2.6).
4. Ein aufgabenparalleler Block enthalte n Filter $f_1, ..., f_n$. Dann müssen folgende Bedingungen gelten: (1) Entweder jedes f_i oder kein f_i besitzt den Eingabetyp `void`. (2) Entweder jedes f_i oder kein f_i besitzt den Ausgabetyp `void`.

5. Falls ein aufgabenparalleler Block nicht im Rumpf eines Filters verwendet wird, müssen sowohl Eingabe- als auch Ausgabetyp jedes Filters `void` sein.
6. Falls ein aufgabenparalleler Block im Rumpf eines Filters verwendet wird, handelt es sich um verschachtelte Parallelität und es gelten die dafür definierten semantischen Regeln (vgl. Abschnitt 4.1.2.6).

Die restriktiven Typanforderungen an die in Fließbandblöcken verwendeten Filter sind notwendig, um Fließbänder auch dynamisch typsicher erzeugen zu können. Da in diesem Fall der Aufbau des Fließbands nicht statisch bekannt ist, wird so garantiert, dass die Ausgabe jedes Filters als Eingabe jedes Filters fungieren kann.

4.1.2.6 Geschachtelte Parallelität

Komplexere parallele Applikationen verwenden verschiedene Arten von Parallelität auf unterschiedlichen Abstraktionsebenen. Diese Ebenen und Schachtelungen können in der objektorientierten Stromprogrammierung realisiert werden, indem parallele Anweisungen innerhalb von Filterrümpfen verwendet werden.

So könnte beispielsweise der Filter `index` zwei verschiedene replizierbare Filter `indexA` und `indexB` enthalten:

```
public File => Set<String> index() {
  indexA()+ ||| indexB()+;
}
```

Abbildung 4.6 veranschaulicht diesen verschachtelt-parallelen Filter als Teil eines Stromgraphen. Der Eingabestrom des Elternfilters `index` wird somit an die geschachtelten Filter weitergeleitet; aus deren Ausgabeströmen wird der Ausgabestrom des Elternfilters gebildet. Da der Eingabestrom also nicht direkt von `index` verarbeitet wird, darf dieser keinen zusätzlichen work-Block enthalten. Auf diese Weise lassen sich beliebig verschachtelte parallele Strukturen definieren.

■ SEMANTISCHE REGELN FÜR GESCHACHTELTE PARALLELE ANWEISUNGEN IN EINEM ELTERNFILTER e:

1. Für Fließbandanweisungen müssen folgende Bedingungen gelten: (1) Der Eingabetyp des ersten Filters f_1 ist derselbe wie der Eingabetyp von e oder ein Obertyp. (2) Der Ausgabetyp des letzten Filters f_n ist derselbe wie der Ausgabetyp von e oder ein Subtyp.
2. Für aufgabenparallele Anweisungen, Alternativenanweisungen, Fließbandblöcke sowie aufgabenparallele Blöcke müssen folgende Bedingungen gelten: (1) Der Eingabetyp jedes Filters f_i ist derselbe wie der Eingabetyp von e oder ein Obertyp. (2) Der Ausgabetyp jedes Filters f_i ist derselbe wie der Ausgabetyp von e oder ein Subtyp.

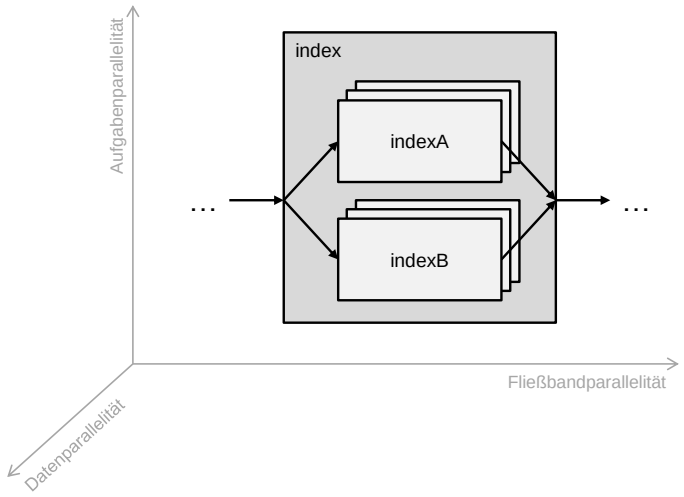


Abbildung 4.6: Verschachtelte Parallelität

3. Für Ausdrucksanweisungen, die aus einem einfachen oder erweiterten (datenparallelen) Filteraufruf f bestehen, müssen folgende Bedingungen gelten: (1) Der Eingabetyp von f ist derselbe wie der Eingabetyp von e oder ein Obertyp. (2) Der Ausgabetyp von f ist derselbe wie der Ausgabetyp von e oder ein Subtyp.

4.1.2.7 Teleports

In den vorigen Abschnitten wurden Sprachkonzepte beschrieben, welche die Implementierung von Fließband-, Aufgaben- und Datenparallelität erlauben. Damit sind die wichtigsten Parallelitätsformen, die in gängigen Applikationen auftreten, abgedeckt. Diese „Bausteine“ lassen sich zu azyklischen Stromgraphen zusammensetzen.

Für manche Anwendungen sind jedoch zyklische Strukturen bzw. Kanäle zwischen nicht explizit miteinander verbundenen Filtern von Interesse. So kann es etwa für Fließband- oder Master-Worker-Implementierungen sinnvoll sein, Teilergebnisse an eine vorige Stufe bzw. den Master-Filter zurückzuschicken, beispielsweise wenn ein bestimmtes Gütekriterium nicht erfüllt ist.

Zur Lösung dieses Problems wird das Konzept des Teleports eingeführt. Ein Teleport erlaubt die Weitergabe eines Stromelements (Objekts) zwischen Filtern, die nicht unmittelbar durch einen Kanal miteinander verbunden sind.

Ein Teleport wird als typisierte Variable deklariert, aber nicht explizit initialisiert. Die Initialisierung erfolgt implizit, indem der Teleportbezeichner einem Filteraufruf vorangestellt wird. Der Bezeichner zeigt somit auf den Eingabekanal dieses Filters. Um ein Stromelement an

diesen Eingabekanal zu schicken, wird der entsprechende Teleportbezeichner an eine push-Anweisung mithilfe des => Operators angehängt.

Wir erweitern somit die Produktionsregeln für Variablendeklarationen (*VarDecl*), Filteraufrufe (*FilterCall*) und push-Anweisungen (*PushStmt*) um entsprechende Regeln für Teleportdeklarationen (*TeleportDecl*), implizite Teleportinitialisierung (*TeleportFilterCall*) und teleportierende push-Anweisungen (*TeleportPushStmt*).

VarDecl	→	... | *TeleportDecl*
TeleportDecl	→	`teleport <` *Type* `>` *Identifier*
FilterCall	→	... | *TeleportFilterCall*
TeleportFilterCall	→	`[` *Identifier* `]` *SimpleFilterCall*
		| `[` *Identifier* `]` *ExtFilterCall*
PushStmt	→	... | *TeleportPushStmt*
TeleportPushStmt	→	`push` *Expr* `=>` *Identifier* `;`

Man beachte, dass teleportierende push-Anweisungen ein Stromelement unmittelbar in den referenzierten Eingabekanal bzw. den Eingabestrom des empfangenden Filters einfügen.

Listing 4.3: Rückkopplung mithilfe eines Teleports

```
1  class C {
2    void => X s1() { ... }
3    X => X s2() { ... }
4    X => void s3(teleport<X> t) {
5      work (X x) {
6        if (cond)
7          push x => t;
8        else { ... }
9      }
10   }
11
12   void m() {
13     teleport<X> t;
14     s1() => [t] s2() => s3(t);
15   }
16 }
```

Zur Veranschaulichung zeigt Listing 4.3 ein dreistufiges Fließband mit teleportbasierter Rückkopplung; der entsprechende Stromgraph ist in Abbildung 4.7 skizziert. Der Eingabekanal der Filterinstanz `s2` wird der Teleportvariablen `t` zugewiesen. `t` wird als Parameter an die Filterinstanz `s3` übergeben, so dass `s3` bei Eintreten einer Bedingung `cond` das aktuelle Stromelement an `s2` zurückschicken kann. Nach diesem Prinzip können nicht nur Rückkopplungen, sondern

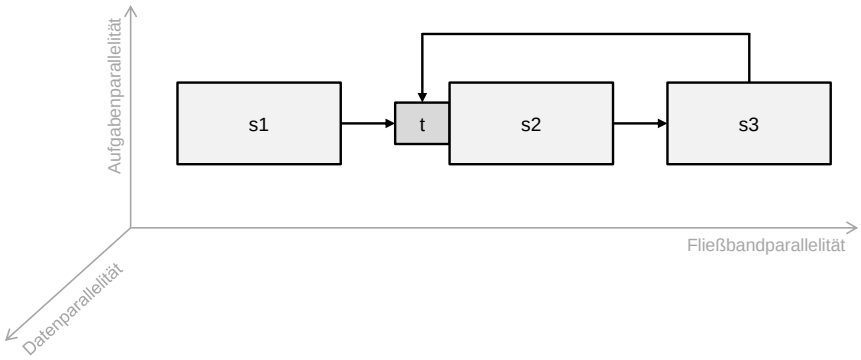


Abbildung 4.7: Teleport

beliebige Teleportverbindungen zwischen nicht benachbarten Filtern hergestellt werden. Auf diese Weise lassen sich beliebige Stromgraphen implementieren.

■ SEMANTISCHE REGELN FÜR TELEPORTS:

1. Wird der Eingabekanal eines Filters einer Teleportvariablen zugewiesen, so muss der Typ des Teleports derselbe sein wie der Eingabetyp des Filters oder ein Subtyp.

2. Eine teleportierende push-Anweisung muss im Rumpf eines Filters verwendet werden.

3. Für eine teleportierende push-Anweisung muss der Typ des push-Arguments *Expr* derselbe sein wie der Typ des empfangenden Teleports oder ein Subtyp. Der Typ des push-Arguments und der Ausgabetyp des Filters, in dessen Rumpf die teleportierende push-Anweisung verwendet wird, können somit verschieden sein.

4.1.2.8 Aufteilen und Zusammenführen von Strömen

Ist der Ausgangs- oder Eingangsgrad eines Filters größer als 1, so muss der Datenstrom an dieser Stelle aufgeteilt bzw. zusammengeführt werden. Dies ist beispielsweise in nichtlinearen Fließbändern oder Auftraggeber-Auftragnehmer-Mustern der Fall. Hierbei muss festgelegt werden, nach welchem Mechanismus der Datenstrom aufgeteilt bzw. zusammengeführt wird.

Für einen beliebigen Stromgraphen G betrachten wir also o.B.d.A. einen Teilgraphen $G' \subset G$ mit den Filtern $V' = \{a, b^{(1)}, ..., b^{(n)}, c\}$ und den Kanälen $V' = \bigcup_{i=1}^{n} \{(a, b^{(i)}), (b^{(i)}, c)\}$. Der Ausgangsgrad von a sowie der Eingangsgrad von c ist n; die Menge der Instanzen $b^{(i)}$ repräsentiert Aufgaben- oder Datenparallelität.

Wir unterscheiden folgende Strategien zum Aufteilen und Zusammenführen von Datenströmen:

- **Roundrobin:** Die Stromelemente werden von a an die Instanzen b_i reihum verteilt bzw. von den Instanzen b_i reihum an c gesendet und zusammengeführt.
- **Firstcome:** Ein Stromelement wird von derjenigen Filterinstanz b_i empfangen, die als erstes zur Bearbeitung bereit ist, bzw. c empfängt ein Stromelement derjenigen Instanz b_i, die als erstes ein solches Element bereitstellen kann. Diese Strategie kann die Wartezeiten der Filterinstanzen reduzieren und so deren Durchsatz erhöhen; die Reihenfolge der Stromelemente kann sich dadurch verändern.
- **Broadcast:** Diese Strategie kann nur für das Aufteilen von Stromelementen angewendet werden. Jedes Stromelement wird demnach an jede Instanz b_i gesendet.

Die Roundrobin-Strategie legen wir als Standardfall fest. Um eine der anderen genannten Strategien anwenden zu können, stellt unser Sprachentwurf den Firstcome-Operator `?` und den Broadcast-Operator `*` bereit. Diese Operatoren können im Kopf einer Filterdeklaration oder vor bzw. nach datenparallelen Filteraufrufen verwendet werden. Somit ergänzen wir die in den Abschnitten 4.1.1 und 4.1.2.4 definierten Produktionsregeln für Filterköpfe (*FilterHeader*) und Filteraufrufe (*FilterCall*) wie folgt:

FilterHeader	→	*Modifier* $Type_{in}$ *SplitOpt* `=>` *JoinOpt* $Type_{out}$ *Identifier* `(` *ParamList* `)` *Exceptions*
FilterCall	→	*SimpleFilterCall* | *SplitOpt ExtFilterCall JoinOpt*
SplitOpt	→	| `?` | `*`
JoinOpt	→	| `?`

■ SEMANTISCHE REGELN FÜR DAS AUFTEILEN UND ZUSAMMENFÜHREN VON STRÖMEN:

1. Für Aufteilungs- und Zusammenführungsoperatoren in Filterköpfen gilt: (1) Der Rumpf des Filters enthält geschachtelte Aufgabenparallelität, d.h. eine aufgabenparallele Anweisung oder einen aufgabenparallelen Block. (2) Ein Aufteilungsoperator ist unzulässig, wenn der Eingabetyp des Filters `void` ist. (3) Ein Zusammenführungsoperator ist unzulässig, wenn der Ausgabetyp des Filters `void` ist.
2. Für datenparallele Filteraufrufe gilt: (1) Ein Aufteilungsoperator ist unzulässig, wenn der Eingabetyp des Filters `void` ist. (2) Ein Zusammenführungsoperator ist unzulässig, wenn der Ausgabetyp des Filters `void` ist.

Die folgenden Beispiele zeigen eine zulässige Verwendung der Aufteilungs- und Zusammenführungsoperatoren. In beiden Beispielen erfolgt das Aufteilen nach der Broadcast-Strategie und das Zusammenführen nach der Firstcome-Strategie.

- Filterdeklaration mit geschachtelter Aufgabenparallelität:

```
X *=>? Y f1() { f2() ||| f3(); }
```

- Datenparalleler Filteraufruf in einem Fließband:

```
f4() =>* f5():[3] ?=> f6();
```

4.1.2.9 Entwurfsdiskussion

In diesem Abschnitt erläutern wir Entwurfsentscheidungen für die vorgestellten Sprachkonzepte zur Erzeugung objektorientierter Stromgraphen. Grundsätzlich verfolgt unser Sprachentwurf ein ähnliches Ziel wie die Sprache StreamIt; Parallelität wird durch strukturierte, hierarchische Komposition von Filtern erzeugt. Die Ziele Abstraktion und Einfachheit stehen dabei im Vordergrund.

Unser Sprachentwurf folgt der Philosophie, für die häufigsten Bausteine eines Stromgraphen einfache, intuitive Anweisungen bereitzustellen. Zentraler Bestandteil ist dabei die parallele Anweisung, welche Filter mithilfe von Operatoren zu einer bestimmten Parallelitätsform, Fließband-, Aufgaben- oder Datenparallelität zusammenfasst oder als Alternativen kennzeichnet. Der Aufbau paralleler Anweisungen ist syntaktisch ähnlich zu einfachen arithmetischen Ausdrücken; die genannten Parallelitätsformen können somit präzise und in sequenziell erscheinender Weise formuliert werden. Wir gehen davon aus, dass diese Darstellungsform dem Programmierer in hohem Maße zugänglich ist.

Stromgraphen werden aus parallelen Anweisungen zur Beschreibung von Fließbändern, Aufgaben- und Datenparallelität hierarchisch aufgebaut, was zum Verständnis und zur Nachvollziehbarkeit der Parallelität beiträgt. Vordefinierte Strategien zum Aufteilen und Zusammenführen von Strömen können mithilfe einfacher Operatoren festgelegt werden. Auf diese Weise sind die Kanäle, welche die Filter verbinden, implizit definiert. Die Alternative, Kanäle als explizite Objekte zu modellieren und Filtern explizit zuzuweisen, würde zum einen die Programmierkomplexität erhöhen, zum anderen die Nachvollziehbarkeit der Parallelität und des Datenflusses beeinträchtigen.

Als Ergänzung zu parallelen Anweisungen und impliziten Kanälen bietet unser Sprachentwurf durch das Konzept des Teleports eine Möglichkeit, beliebige Verbindungen zu definieren und damit allgemeine, insbesondere zyklische, Stromgraphen typsicher zu implementieren.

4.2 Stromorientierte Darstellung paralleler Muster

Ein wichtiges Ziel der objektorientierten Stromprogrammierung ist die Eignung zur Implementierung zentraler paralleler Muster. Ähnlich wie Entwurfsmuster vereinfachen parallele Muster die Entwicklung, das Verständnis und die Wartbarkeit paralleler Software. In diesem Abschnitt wird daher illustriert, wie die in Abschnitt 2.2.4 vorgestellten parallelen Muster mithilfe der Stromprogrammierung implementiert werden können.

Diese Muster kommen beim Entwurf und der Implementierung paralleler Anwendungen häufig zum Einsatz und decken zudem in ihrer Gesamtheit die Arten Fließband-, Aufgaben- und Datenparallelität ab.

4.2.1 Fließband

Die Implementierung des Fließbandmusters wurde bereits in Abschnitt 4.1.2.1 erläutert und wird an dieser Stelle nicht ausführlicher thematisiert. Grundsätzlich lassen sich lineare und nichtlineare Fließbänder unterscheiden. Nichtlineare Fließbänder enthalten mindestens eine daten- oder aufgabenparallele Stufe, so dass ein Aufteilen und Zusammenführen des Datenstroms erforderlich wird. Für beide Arten von Fließbändern stellt die objektorientierte Stromprogrammierung effiziente Möglichkeiten in Form von Filterreplizierung, verschachtelter Parallelität und Split-Join-Strategien bereit. Etwaige Rückkopplungen zwischen Fließbandstufen lassen sich mithilfe von Teleports einfach umsetzen.

4.2.2 Erzeuger-Verbraucher

Das Erzeuger-Verbraucher-Muster (engl. *producer-consumer*) kann als Spezialfall des Fließbandmusters angesehen werden, genauer gesagt als zweistufiges, nichtlineares Fließband. Listing 4.4 skizziert eine generische Klasse `C`, die zwei Filter deklariert, welche als Erzeuger bzw. Verbraucher fungieren. Die Methode `m` in Klasse `D` kombiniert diese Filter zu einem Erzeuger-Verbraucher-Muster. Der Modifikator `?` legt fest, dass die produzierten `Item` Objekte in beliebiger Reihenfolge konsumiert werden können, wie es für Erzeuger-Verbraucher-Muster typischerweise der Fall ist.

4.2.3 Auftraggeber-Auftragnehmer

Das Auftraggeber-Auftragnehmer-Muster (engl. *master-worker*) kommt in vielen parallelen Applikationen zur Anwendung; es besteht aus einem Auftraggeber (engl. *master*), der eine Menge von Aufgaben an Auftragnehmer bzw. Arbeiter (engl. *worker*) verteilt. In der Regel werden die Aufgaben von den Arbeitern vollständig bearbeitet; manche Problemstellungen können jedoch auch eine Rückkopplung erfordern, so dass Aufgaben nur teilweise bearbeitet und ggf. in die Warteschlange zurückgelegt werden.

In Listing 4.5 deklariert die Klasse `C` zwei Filter `master` und `worker`, wobei letzterer einen Teleport verwendet. Die Methode `m` in Klasse `D` realisiert ein Auftraggeber-Auftragnehmer-Szenario mit Aufgabenrückkopplung. Die Anzahl der Arbeiter wird hierbei vom Auto-Tuner eingestellt. Darüber hinaus wird wie im Produzenten-Konsumenten-Beispiel angenommen,

Listing 4.4: Strombasierte Implementierung des Erzeuger-Verbraucher-Musters

```
class Item { ... }

class C<E> {
  void => E produce() { ... }

  E => void consume() {
    work (E item) { ... }
  }
}

class D {
  void m() {
    C<Item> c = new C<Item>();
    c.produce():[2] =>? c.consume():[4];
  }
}
```

Listing 4.5: Strombasierte Implementierung des Auftraggeber-Auftragnehmer-Musters mit Rückkopplung

```
class Job { ... }

class C<E> {
  boolean isComplete(E e) { ... }

  void => E master() { ... }

  E => void worker(teleport<E> t) {
    work (E item) {
      ...
      if (!isComplete(item)) push item => t;
    }
  }
}

class D {
  void m() {
    C<Job> c = new C<Job>();
    teleport<Job> t;
    c.master() =>? [t] c.worker(t)+;
  }
}
```

dass die Stromelemente (in diesem Fall Aufgaben bzw. `Job` Objekte) in beliebiger Reihenfolge an die Arbeiter verteilt werden können.

4.2.4 Teile und Herrsche

Eine wichtiges paralleles Muster ist das aus der Algorithmik bekannte Teile-und-Herrsche-Prinzip (engl. *divide and conquer*). Dabei werden größere Aufgaben rekursiv in kleinere unabhängige Aufgaben zerlegt, die einfacher zu berechnen sind. Deren Teilergebnisse werden anschließend wieder schrittweise zusammengefügt. Die Teilaufgaben eignen sich aufgrund ihrer Unabhängigkeit gut zur parallelen Ausführung.

Listing 4.6: Implementierung des Teile-und-Herrsche-Musters

```
1  class MergeSort {
2    void merge(int[] a, int fromIdx, int midIdx, int toIdx) { ... }
3    void sort(int[] a, int fromIdx, int toIdx) { ... }
4
5    void seqMsort(int[] a, int fromIdx, int toIdx) {
6      if (toIdx - fromIdx > 1) {
7        int midIdx = fromIdx + (toIdx - fromIdx) / 2;
8        seqMsort(a, fromIdx, midIdx);
9        seqMsort(a, midIdx + 1, toIdx);
10       merge(a, fromIdx, midIdx, toIdx);
11     } else {
12       sort (a, fromIdx, toIdx);
13     }
14   }
15
16   void => void parMsort(int[] a, int fromIdx, int toIdx) {
17     if (toIdx - fromIdx > 1) {
18       int midIdx = fromIdx + (toIdx - fromIdx) / 2;
19       parMsort(a, fromIdx, midIdx) ||| parMsort(a, midIdx + 1, toIdx);
20       merge(a, fromIdx, midIdx, toIdx);
21     } else {
22       sort (a, fromIdx, toIdx);
23     }
24   }
25 }
```

Ein Beispiel ist der Mergesort-Algorithmus, dessen sequenzielle und parallele Variante in Listing 4.6 dargestellt ist. Das Beispiel zeigt, dass sich sequenzielle und parallele Implementierung lediglich in zwei Punkten unterscheiden: zum Sortieren wird statt einer Methode (Zeile 5) ein Filter (Zeile 16) deklariert; statt zwei Methodenaufrufen (Zeilen 8 und 9) erfolgt der Rekursionsschritt durch eine aufgabenparallele Anweisung (Zeile 19). Diese Parallelisierung basiert nicht auf Datenströmen, sondern auf reiner Aufgabenparallelität nach dem Prinzip von

Cilk [87]. Der Ausführungsmodus für die Filteraufrufe einer Rekursionsebene, d.h. das Umschalten zwischen paralleler und sequenzieller Ausführung, wird während der Ausführung festgelegt.

4.2.5 Datenzerlegung durch MapReduce

Das MapReduce-Muster fällt in die Kategorie der Datenzerlegungsmuster. MapReduce besteht aus drei Phasen: dem Aufteilen einer Eingabemenge, der Berechnung der Teilaufgaben und dem Zusammenführen der Ergebnisse. Im klassischen MapReduce-Ansatz sind diese Phasen zeitlich voneinander getrennt; Varianten wie Tiled-MapReduce lockern diese Bedingung [27]. Das für Multikernsysteme effizientere Tiled-MapReduce-Verfahren lässt sich stromorientiert als dreistufiges Fließband implementieren, welches die Eingabedatenmenge partitioniert und iterativ parallel verarbeitet.

Listing 4.7: Strombasierte Implementierung des MapReduce-Musters

```
1  class Data { ... }
2
3  class C<E,F> {
4    void => E map(E in) { ... }
5
6    E => F process() {
7      work (E item) { ... }
8    }
9
10   F => void reduce(F result) {
11     work (F item) { ... }
12   }
13 }
14
15 class D {
16   void m(Data in, Data result) {
17     C<Data, Data> c = new C<Data, Data>();
18     c.map(in) =>? c.process()+ ?=> c.reduce(result);
19   }
20 }
```

Listing 4.7 zeigt eine generische Klasse `C`, welche drei Filter deklariert. Der Filter `map` wandelt eine MapReduce-Eingabe `in` in einen Strom von Teilaufgaben des generischen Typs `E` um. Dieser Strom wird von dem replizierbaren Filter `process` verarbeitet. Der Filter `reduce` fasst den Strom aus Teilergebnissen des generischen Typs `F` zu einem Ergebnisobjekt `result` zusammen. Die Methode `m` in Klasse `D` kombiniert die Filter zu einem dreistufigen Fließband,

welches nach dem MapReduce-Prinzip arbeitet. In diesem Beispiel wird angenommen, dass bei der Bearbeitung des Aufgabenstroms keine Reihenfolge einzuhalten ist, so dass sich der Strom nach dem Firstcome-Verfahren aufteilen und zusammenführen lässt.

4.3 Identifikation von zustandsbehafteten Filtern und kritischen Abschnitten

Bei der Übertragung stromorientierter Programmierkonzepte auf objektorientierte Sprachen ergeben sich Herausforderungen bezüglich der Korrektheit eines Programms. Die meisten klassischen Stromsprachen erlauben nur zustandslose, leichtgewichtige Filter, um Parallelität aggressiv ausnutzen zu können. Dies stellt eine Einschränkung der Ausdrucksmächtigkeit einer Sprache dar, die insbesondere im objektorientierten Kontext nicht wünschenswert ist.

In einigen Anwendungen ist es erforderlich, dass Filter einen Zustand besitzen oder gemeinsame Variablen verwenden. Ein Beispiel eines zustandsbehafteten Filters ist in Listing 4.8 dargestellt; Listing 4.9 zeigt eine Klasse mit zwei Filtern, welche über Methoden auf eine gemeinsame Variable `x` zugreifen. Solche Fälle müssen bei der Ausführung berücksichtigt werden.

Listing 4.8: Zustandsbehafteter Filter

```
1 int => int foo() {
2   int state = 0;
3   work (int i) {
4     push i + state;
5     state = i;
6   }
7 }
```

- In zustandsbehafteten Filtern existieren Datenabhängigkeiten zwischen zwei oder mehreren Aktionen. Im Beispiel besitzt der Filter `foo` eine lokale Variable `state`, die in einer Aktion geschrieben und in der folgenden Aktion gelesen wird. Die Aktionen sind als voneinander abhängig, so dass diese Art von Filter nicht replizierbar ist. Im Falle der Replikation könnte die Variable `state` falsche Werte annehmen und so zu fehlerhaften Ausgaben führen.
- Filter, die ungeschützt auf gemeinsame Variablen zugreifen, können Wettlauffehler verursachen. Es muss daher sichergestellt werden, dass gleichzeitige Zugriffe auf diese Variablen ausgeschlossen sind. Dies kann vom Programmierer durch entsprechende Synchronisierung geregelt werden oder automatisiert durch den Übersetzer, z.B. durch die Generierung von Sperren, geschehen.

Listing 4.9: Ungeschützter Zugriff auf eine gemeinsame Variable durch zwei Filter

```
1  class C {
2    private int x = 0;
3    void update(int i) { x = x + i; }
4    int get() { return x; }
5
6    int => int foo() {
7      work (int i) { update(i); push i; }
8    }
9
10   int => int bar() {
11     work (int i) { push get(); }
12   }
13 }
```

Diese Arbeit begegnet dieser Problematik, indem in der Übersetzungsphase Zustands- und Konfliktanalysen eingesetzt werden. Diese werden im Folgenden erläutert.

4.3.1 Zustandsanalyse

Die Zustandsanalyse prüft, ob ein Filter zustandsbehaftet ist. Grundsätzlich können Zustände in beliebigen Variablen gespeichert werden. Gemäß der Definition in Abschnitt 4.1.1.6 beschränkt sich unsere Analyse auf solche Zustände, die in lokalen, exklusiven Variablen gespeichert sind.

Wurde ein Filter als zustandsbehaftet identifiziert, so wird eine Übersetzerwarnung ausgegeben. Hintergrund ist die Tatsache, dass die Zustandsanalyse keine absolute Genauigkeit garantiert, da auch harmlose Zustände wie beispielsweise einfache Zählvariablen gemeldet würden. Da Falschmeldungen somit nicht ausgeschlossen sind, gibt der Übersetzer entsprechende Warnungen aus, um den Programmierer zu unterstützen.

Konzeptionell ist die Zustandsanalyse verwandt mit Abhängigkeitsanalysen zur Schleifenparallelisierung. Eine Datenabhängigkeit liegt demnach dann vor, wenn zwei Anweisungen dieselbe Variable lesen oder schreiben. Diese Bedingung ist jedoch zu allgemein, um auf einen Filterzustand schließen zu können. Vielmehr müssen genau solche Fälle betrachtet werden, in denen eine Aktion auf eine Variable erst lesend und dann schreibend zugreift (Lese-Schreib-Abhängigkeit). In diesem Fall wird ein Zustand gespeichert, der in der darauf folgenden Aktion gelesen wird; die Aktionen sind somit voneinander abhängig.

Algorithmus 1 zeigt den Ablauf der Zustandsanalyse in Pseudocode. Für einen (periodischen) Filter f werden basierend auf dessen Kontrollflussgraphen alle Anweisungen des work-Blocks durchlaufen und ermittelt, welche Variablen durch eine Anweisung gelesen oder geschrieben

Algorithmus 1 : Zustandsanalyse

```
1  procedure StateAnalysis(filter f)
2      R := ∅;                                   ▷ Menge der gelesenen Variablen
3      W := ∅;                                   ▷ Menge der geschriebenen Variablen
4      foreach statement s in work block of f do
5          if s reads some filter local variable v then
6              if v ∉ W then
7                  R := R ∪ {v};
8              end
9          end
10         if s writes some filter local variable v then
11             if v ∉ R then
12                 W := W ∪ {v};
13             else
14                 report f;                     ▷ Melde f als zustandsbehaftet
15                 return;
16             end
17         end
18     end
19 end
```

werden. Gemäß der Definition von Zustandsvariablen (vgl. Abschnitt 4.1.1.6) verwenden wir hierbei eine intraprozedurale Analyse, welche nur filterlokale Variablen berücksichtigt. Die Mengen R und W enthalten diejenigen Variablen, die bisher ausschließlich gelesen bzw. geschrieben wurden. Bei einem Lesezugriff auf eine Variable v wird v also nur dann der Menge R hinzugefügt, wenn v zuvor noch nicht geschrieben wurde ($v \notin W$). Der Fall eines Schreibzugriffes wird analog behandelt; wird hierbei eine Variable v geschrieben, die bereits in R enthalten ist, wird der Filter f als zustandsbehaftet markiert, da eine Lese-Schreib-Abhängigkeit vorliegt.

4.3.2 Konfliktanalyse

Um Zugriffe auf gemeinsame Variablen zu schützen, erlaubt das Modell der objektorientierten Stromprogrammierung das Verwenden von Sperren. Aufgrund der Komplexität der parallelen Programmierung werden Sperren jedoch leicht falsch verwendet oder vergessen, was zu Wettlaufsituationen führen kann.

Ziel der Konfliktanalyse ist es, kritische Abschnitte zu identifizieren. Die Analyse untersucht Methodenpaare, welche zeitgleich aufgerufen werden könnten und auf gemeinsame Variablen ungeschützt zugreifen. Die Konfliktanalyse basiert auf der Locksetanalyse, die für jede Anweisung eines Programms die Menge der gehaltenen Sperren berechnet. Damit lassen sich mögliche Wettlauffehler bereits zur Übersetzungszeit identifizieren.

Die Erkennung von kritischen Abschnitten bzw. möglichen Wettlauffehlern stellt grundsätzlich ein äußerst komplexes Thema dar, welches die vorliegende Arbeit nicht vollständig lösen soll bzw. kann. Wir beschränken uns daher auf solche kritische Abschnitte, in denen Zugriffe auf *einzelne* gemeinsame Variablen erfolgen. Wettlaufsituationen, die in Zusammenhang mit Zugriffen auf *mehrere* Variablen entstehen können, liegen nicht im Fokus unserer Analyse – ein Beispiel sind korrelierte Variablen, welche nur als Gruppe gelesen und geschrieben werden dürfen.

Grundsätzlich handelt es sich bei unserer Analyse um einen konservativen Ansatz. Für Arrays oder solche Variablen, für im Code viele Aliase besitzen, trifft die Zeigeranalyse pessimistische Annahmen. Unter solchen Umständen können viele Falschmeldungen auftreten.

4.3.2.1 Locksetanalyse

Die Locksetanalyse berechnet, welche Programmteile durch welche Sperren geschützt werden. Dieses Verfahren besteht aus intra- und interprozeduralen Analysen, welche ursprünglich zur statischen Erkennung von Synchronisierungsfehlern entwickelt wurden [79, 80]. Die intraprozedurale Analyse untersucht einzelne Filter- oder Methodenrümpfe; die interprozedurale Analyse findet filter- und methodenübergreifend statt. Auf dieser Basis können später Aussagen darüber getroffen werden, welche Teile des Gesamtprogramms zueinander in Konflikt stehen können (vgl. Abschnitt 4.3.2.2).

Intraprozedurale Analyse

Die intraprozedurale Locksetanalyse berechnet für jede Anweisung im Rumpf einer Methode oder eines Filters, welche Sperren dort gehalten werden. Algorithmus 2 skizziert das Verfahren.

Die Prozedur *IntraLocksetAnalysis* führt die Faktenextraktion durch; die anschließende Faktenpropagierung erfolgt durch die Prozedur *DoAnalysis*. Eingabeparameter beider Prozeduren sind der Kontrollflussgraph CFG einer Methode oder eines Filters sowie eine Eintrittssperrmenge L. Letztere gibt an, welche Sperren beim Eintritt in den Kontrollflussgraphen, also beim Aufruf der Methode bzw. des Filters, gehalten werden. Die Prozedur *IntraLocksetAnalysis*

Algorithmus 2 : Intraprozedurale Locksetanalyse

```
1  procedure IntraLocksetAnalysis(control flow graph CFG, entry lockset L)
2      foreach statement s in CFG do
3          GEN(s) := ∅;
4          KILL(s) := ∅;
5          if s acquires lock l then
6              GEN(s) := {l};
7          else
8              if s releases lock l then
9                  KILL(s) := {l};
10             end
11         end
12     end
13     DoAnalysis(CFG, L);
14 end

15 procedure DoAnalysis(control flow graph CFG, entry lockset L)
16     foreach statement s in CFG do
17         PRE(s) := {predecessors of s};
18         if PRE(s) = ∅ then
19             IN(s) := L;
20         else
21             IN(s) := ⋃_{s'∈PRE(s)} OUT(s');
22         end
23         OUT(s) := (IN(s) \ KILL(s)) ∪ GEN(s);
24         Lockset(s) := OUT(s);
25     end
26 end
```

durchläuft alle Knoten des Kontrollflussgraphen der Methode m und ermittelt für jede Anweisung s die Mengen $GEN(s)$ und $KILL(s)$. Diese Mengen geben an, welche Sperren beim Ausführen einer Anweisung s gesetzt bzw. freigegeben werden.

Stehen für einen Methodenrumpf alle Mengen *GEN* und *KILL* fest, kann die Prozedur *DoAnalysis* die eigentliche Datenflussanalyse durchführen und so die Sperrmengen für jede Anweisung berechnen. Die Mengen $IN(s)$ und $OUT(s)$ enthalten diejenigen Sperren, die vor bzw. nach dem Durchlaufen einer Anweisung s gehalten werden. Für Anweisungen, die keinen Vorgänger im Kontrollflussgraphen besitzen, entspricht *IN* der Eintrittssperrmenge L. Andernfalls ergibt sich *IN* aus der Vereinigung der OUT-Mengen aller direkter Vorgängerknoten.

Interprozedurale Analyse

Die im vorigen Abschnitt beschriebene intraprozedurale Locksetanalyse untersucht den Rumpf einer Methode oder eines Filters. Um Aussagen über das gesamte Stromprogramm machen zu können, muss eine interprozedurale Analyse durchgeführt werden. Algorithmus 3 zeigt das Vorgehen in Pseudocode.

Algorithmus 3 : Interprozedurale Locksetanalyse

```
1  procedure InterLocksetAnalysis(set F of filters)
2      foreach filter f ∈ F do
3          VisitBody(CFG(f), ∅);
4      end
5  end

6  procedure VisitBody(control flow graph CFG, entry lockset L)
7      IntraLocksetAnalysis(CFG, L);          ▷ Analysiere CFG mit Eintrittssperrmenge L
8      mark (CFG, L);                         ▷ Markiere (CFG, L) als besucht
9      foreach statement s in CFG do
10         foreach method m' called by s do
11             if (CFG(m'), IN(s)) is not marked then
12                 VisitBody(CFG(m'), IN(s));
13             end
14         end
15     end
16 end
```

Grundsätzlich müssen intraprozedurale Locksetanalysen für alle Methoden durchgeführt werden, die von einem Filter aus erreichbar sind. Auf diese Weise werden alle Codestellen abgedeckt, die parallel ausgeführt werden könnten. Die Prozeduren *InterLocksetAnalysis* und *VisitBody* untersuchen dementsprechend jeden Filter und alle davon erreichbaren Methoden. Um Endlosschleifen zu vermeiden, wird jede besuchte Methode markiert.

4.3.2.2 Identifikation möglicher Konflikte

Die Konfliktanalyse ermittelt Programmstellen, die auf gemeinsame Variablen zugreifen und bei zeitgleicher Ausführung Wettlauffehler verursachen könnten. Dies ist der Fall für ungeschützte Lese-Schreib- oder Schreib-Schreib-Zugriffe. Die im vorigen Abschnitt beschriebene Locksetanalyse bildet insofern einen wichtigen Bestandteil der Konfliktanalyse, da mithilfe

der Sperrmengen ungeschützte von geschützten Zugriffen unterschieden werden können. Algorithmus 4 veranschaulicht die Konfliktanalyse in Pseudocode.

Die Prozedur *ComputeRWSets* bestimmt die unmittelbaren und mittelbaren Lese- und Schreibzugriffe durch einen Filter f. Hierfür werden die Kontrollflussgraphen von f sowie aller von f erreichbaren Methoden durchlaufen. Die Variablenzugriffe werden sukzessive den Zugriffsmengen $R(f)$ bzw. $W(f)$ hinzugefügt. Für jeden erfassten Zugriff werden folgende Informationen festgehalten: (1) die Anweisung s, (2) die dadurch gelesene bzw. geschriebene Variable v, (3) die Methode m, welche die Anweisung s enthält, und (4) die für s gesetzten Sperren $Lockset(s)$.

Anhand der Zugriffsmengen R und W kann die Prozedur $IdentifyConflicts$ mögliche Konflikte zwischen zwei Filtern erkennen. Für jedes Zugriffspaar (a_1, a_2), welches aus mindestens einem Schreibzugriff besteht, wird mithilfe einer Zeigeranalyse geprüft, ob die Zielmengen $PointsToSet(a_1.v), PointsToSet(a_2.v)$ der Variablen $a_1.v, a_2.v$ eine nichtleere Schnittmenge bilden. In diesem Fall könnten sich die Zugriffe auf dieselbe Speicherstelle beziehen. Sind zusätzlich die Sperrmengen $Lockset(a_1.s), Lockset(a_2.s)$ der zugehörigen Anweisungen $a_1.s, a_2.s$ disjunkt, so sind die Zugriffe nicht konsistent geschützt; für das Zugriffspaar (a_1, a_2) wird somit ein möglicher Konflikt gemeldet.

4.3.2.3 Zeigeranalyse und Filterkontext

Die Identifikation möglicher Konflikte hängt ab von der Qualität der Zeigeranalyse. In Kapitel 2 wurden die Freiheitsgrade der Fluss- und Kontextsensitivität erläutert, die Aufwand und Genauigkeit der Zeigeranalyse beeinflussen.

Wir verwenden die Zeigeranalyse in zwei unterschiedlichen Konfigurationen. Beide Varianten sind flussinsensitiv, unterscheiden sich aber bezüglich der Kontextsensitivität.

- Die *kontextinsensitive* Variante ignoriert den Aufrufkontext von Methoden und Filtern, unterscheidet also weder zwischen verschiedenen Methoden- noch verschiedenen Filterinstanzen.

- Die *filterkontextsensitive* Variante berücksichtigt den Aufrufkontext von Filtern sowie der unmittelbar in Filtern aufgerufenen Methoden und ignoriert den Aufrufkontext aller anderen Methoden. Ziel dieser Variante ist es, mit moderater Aufwandserhöhung eine höhere Genauigkeit zu erreichen.

Algorithmus 4 : Konfliktanalyse

```
1  procedure ConflictAnalysis(set F of filters)
2      foreach filter f ∈ F do
3          ComputeRWSets(f);                    ▷ Bestimme Lese- und Schreibzugriffe durch f
4      end
5      foreach filter pair (f1, f2) ∈ F × F do
6          IdentifyConflicts(f1, f2);           ▷ Identifiziere mögliche Konflikte
7      end
8  end

9  procedure ComputeRWSets(filter f)
10     R(f) := ∅;                               ▷ Lesezugriffe durch f
11     W(f) := ∅;                               ▷ Schreibzugriffe durch f
12     Reachable(f) := {f} ∪ {reachable methods from f};
13     foreach m ∈ Reachable(f) do
14         foreach statement s in control flow graph CFG(m) of m do
15             if s reads some variable v then
16                 R(f) := R(f) ∪ {(s, v, m, Lockset(s))};
17             end
18             if s writes some variable v then
19                 W(f) := W(f) ∪ {(s, v, m, Lockset(s))};
20             end
21         end
22     end
23 end

24 procedure IdentifyConflicts(filter f1, filter f2)
25     foreach pair of accesses
       (a1, a2) ∈ (R(f1) × W(f2)) ∪ (W(f1) × R(f2)) ∪ (W(f1) × W(f2)) do
26         if PointsToSet(a1.v) ∩ PointsToSet(a2.v) ≠ ∅ then
27             if Lockset(a1.s) ∩ Lockset(a2.s) = ∅ then
28                 report (a1, a2);             ▷ Melde möglichen Konflikt
29             end
30         end
31     end
32 end
```

4.3.2.4 Behandlung möglicher Konflikte

Die Verwendung von Sperrmechanismen kann auch in Stromprogrammen zu Verklemmungen führen. Wir behandeln daher mögliche Konflikte in Form von Übersetzerwarnungen, indem potenziell kritische Zugriffe dem Programmierer gemeldet werden. Der Programmierer behält dabei die Kontrolle über die Beurteilung und Behebung der identifizierten Probleme.

Eine sinnvolle automatische Behebung ist für den allgemeinen Fall schwierig. Optional kann unser Übersetzer, basierend auf den Ergebnissen der Konfliktanalyse, den Code jedoch auf zwei verschiedene Arten instrumentieren:

- *Einfügen von Sperren.* Methoden, die einen Konflikt verursachen können, wenn sie auf demselben Objekt aufgerufen werden, wandelt der Übersetzer in `synchronized` Methoden um. Auf diese Weise wird ein gegenseitiger Ausschluss erreicht.
- *Kennzeichnung von Transaktionen.* Alle Methoden, die einen Konflikt verursachen könnten, werden als Transaktionen gekennzeichnet. Der Übersetzer versieht diese Methoden mit einer `@atomic`-Annotation aus dem Rahmenwerk Deuce [54]. Dieses Rahmenwerk besteht aus einem Laufzeitsystem für softwareseitigen transaktionalen Speicher (engl. *software transactional memory*, STM) und erlaubt es, `@atomic` annotierte Methoden als Transaktion auszuführen. Von Nachteil ist jedoch, dass die Ausführung und Verwaltung einer solchen Transaktion je nach Komplexität der Methode und der darin referenzierten Datenstrukturen sehr teuer sein kann.

Die Listings 4.11 und 4.12 illustrieren die verschiedenen Instrumentierungsmöglichkeiten für das eingangs verwendete Beispiel, welches der Übersicht halber nochmals in Listing 4.10 dargestellt ist.

4.4 Zusammenfassung

In diesem Kapitel wurden Konzepte vorgestellt und erläutert, welche die Grundidee der Stromprogrammierung erweitern und auf allgemeine, objektorientierte Programmiersprachen übertragen.

Zunächst wurden hierfür Spracherweiterungen definiert, welche stromorientierte Konstrukte in objektorientierte Sprachen integrieren. Diese Erweiterungen wurden für die Programmiersprache Java definiert, sind aber prinzipiell auf andere Sprachen anwendbar. Die Sprachkonstrukte erlauben die Definition von beliebigen Stromgraphen im objektorientierten Kontext. Auf diese Weise lassen sich Aufgaben-, Daten- und Fließbandparallelität effizient implementieren. Dies sind zudem die Grundbausteine für wichtige parallele Entwurfsmuster. Die kompakte Syntax abstrahiert von Ausführungsfäden und feingranularen Synchronisierungsmechanismen.

Listing 4.10: Ursprünglicher Code: die Filter `foo` und `bar` greifen über die Methoden `update` bzw. `get` ungeschützt auf Variable `x` zu

```
1  class C {
2    private int x = 0;
3    void update(int i) { x = x + i; }
4    int get() { return x; }
5
6    int => int foo() {
7      work (int i) { update(i); push i; }
8    }
9
10   int => int bar() {
11     work (int i) { push get(); }
12   }
13 }
```

Listing 4.11: Konfliktbehebung durch Einfügen von Sperren: kritische Methoden werden in `synchronized` Methoden umgewandelt

```
1  class C {
2    private int x = 0;
3    synchronized void update(int i) { x = x + i; }
4    synchronized int get() { return x; }
5
6    int => int foo() { ... }
7    int => int bar() { ... }
8  }
```

Listing 4.12: Konfliktbehebung durch Kennzeichnung von Transaktionen: kritische Methoden werden mit einer `@atomic`-Annotation versehen

```
1  class C {
2    private int x = 0;
3    @atomic void update(int i) { x = x + i; }
4    @atomic int get() { return x; }
5
6    int => int foo() { ... }
7    int => int bar() { ... }
8  }
```

Der Übersetzungsvorgang wurde um Zustands- und Konfliktanalysen erweitert. Die Zustandsanalyse dient der Erkennung zustandsloser Filter, welche Datenparallelität ausnutzen zu können. Die Konfliktanalyse identifiziert kritische Abschnitte, welche zu Wettlauffehler führen könnten. Insofern tragen beide Analysen zur Korrektheit objektorientierter Stromprogramme bei.

Das folgende Kapitel befasst sich mit der Ausführung und Optimierung von Stromprogrammen, insbesondere mit Methoden des Laufzeit-Tunings.

Kapitel 5

Ausführung und Optimierung von Stromprogrammen

In Kapitel 4 wurden der Sprachentwurf zur objektorientierten Stromprogrammierung sowie Ansätze zur Zustands- und Konfliktanalyse vorgestellt. Dieses Kapitel behandelt Konzepte zur Ausführung und Optimierung von Stromprogrammen.

5.1 Ausführung objektorientier Stromprogramme

Ein wichtiges Ziel der parallelen Programmierung im Allgemeinen und der objektorientierten Stromprogrammierung im Besonderen ist es, unabhängig vom Rechnertyp möglichst gute Programmleistungen zu erzielen. Im diesem Zusammenhang ist es von besonderer Wichtigkeit, die Struktur eines parallelen Programm flexibel zu halten und somit einen hohes Maß an Anpassungsfähigkeit zu erreichen. Hierfür müssen unterschiedliche leistungsrelevante Programmparameter in Betracht gezogen werden, z.B. die Anzahl der Ausführungsfäden, die Größen von Datenpartitionen oder etwa die Anzahl der Stufen eines Fließbands.

Abbildung 5.1: Ausführung von Stromprogrammen

Beim Aufrufen von Filtern werden Filterinstanzen erzeugt und in einer Warteschlange abgelegt. Zur Ausführung der Filterinstanzen verwenden wir das Konzept eines Ausführungsdienstes (vgl. Abbildung 5.1). Ein Ausführungsdienst besteht aus mehreren Ausführungsfäden, deren Anzahl unabhängig von der Anzahl der Filter einstell- und kontrollierbar ist. Würde man für jeden Filter einen eigenen Faden verwenden, bestünde die Gefahr eines zu hohen Mehraufwands, insbesondere bei einer großen Anzahl wenig rechenintensiver Filter.

Algorithmus 5 : Arbeitsprinzip eines Ausführungsfadens für periodische Filter

```
1 procedure Execute()
2     while true do
3         take filter f with highest workload from queue;
4         execute some number of actions for f;
5         if f not finished then
6             put f back to queue;
7         end
8     end
9 end
```

Periodische Filter Das Arbeitsprinzip eines Ausführungsfadens für periodische Filter ist in Listing 5 dargestellt. Ein Ausführungsfaden entnimmt der Warteschlange die Filterinstanz mit der aktuell höchsten Arbeitslast[1]. Ziel dieser Priorisierung ist es, Flaschenhalseffekte zu vermeiden und so die Last in einem Stromgraphen auszubalancieren. Der Faden führt für diesen Filter eine bestimmte Anzahl von Aktionen aus und legt den Filter, sofern dieser nicht beendet ist, anschließend in die Warteschlange zurück.

Nichtperiodische Filter Nach der Spezifikation in Abschnitt 4.1.1.2 werden nichtperiodische Filter entweder als Quelle oder zur Schachtelung von Parallelität verwendet. Da diese Filter Stromelemente somit nicht unmittelbar verarbeiten, kann die Ausführung nicht nach demselben Prinzip erfolgen wie für periodische Filter. Nichtperiodische Filter werden daher in eigenen Fäden ausgeführt.

5.2 Optimierbare Stromprogramme

Tuningparameter manuell zu identifizieren, im Code zu spezifizieren und zu konfigurieren ist eine zeitaufwändige Aufgabe. Methoden des Auto-Tunings helfen dabei, den Optimierungsprozess zu automatisieren. Die Grundlagen des Auto-Tunings wurden in Kapitel 2 gelegt.

Auf dem Gebiet der Stromprogrammierung stellt die Idee, das Problem der Leistungsoptimierung mit Methoden des Auto-Tunings zu lösen, einen neuartigen Ansatz dar. Großes Potenzial liegt dabei in der Ausnutzung von implizit gegebenen Strukturinformationen über ein konkretes Stromprogramm. Dies eröffnet die Möglichkeit, nicht nur die Leistungsoptimierung selbst, sondern bereits die tuningspezifische Vorverarbeitung eines Programms zu automatisieren. Die Automatisierung betrifft die folgenden vier Schritte:

[1] Die Arbeitslast ist Teil der Kontextinformationen eines Filters, welche in Abschnitt 5.2.2.2 besprochen werden. Die Last wird approximiert durch die Anzahl der Stromelemente im Eingabekanal des Filters

1. Identifikation von Tuningparametern durch den Übersetzer sowie Instrumentierung des generierten Codes mit entsprechenden Tuninginformationen (Abschnitt 5.2.2.1)

2. Erfassung von Kontextwissen bzw. der Semantik von Tuningparametern (Abschnitt 5.2.2.2)

3. Vorhersage von Parameterwerten anhand von Heuristiken unter Ausnutzung von Kontextwissen (Abschnitt 5.2.3)

4. Durchführung von Leistungsmessungen und Anpassung der Parameterwerte zur Laufzeit (Abschnitt 5.3)

Die Schritte 1 bis 3 fallen in den Bereich der Vorverarbeitung eines zu optimierenden Programms und sind in bisherigen Auto-Tuning-Ansätzen manuell durchzuführen. Die eigentliche (suchbasierte) Programmoptimierung findet in Schritt 4 statt.

Im folgenden Abschnitt werden zunächst wichtige Typen von Tuningparametern und deren mögliche Auswirkungen auf die Performanz eines Programms beschrieben. Darauf aufbauend wird in Abschnitt 5.2.2 dargestellt, wie diese Parameter, ihre Wertebereiche und Kontextinformationen aus einem Stromprogramm extrahiert und nutzbar gemacht werden können. Die Abschnitte 5.2.3 und 5.3 befassen sich mit der Konfiguration dieser Parameter mithilfe von Heuristiken bzw. suchbasiertem Laufzeit-Tuning.

5.2.1 Leistungsrelevante Parameter in Stromprogrammen

Dieser Abschnitt katalogisiert Typen von Tuningparametern, die die Leistung von parallelen Programmen im Allgemeinen und Stromprogrammen im Besonderen maßgeblich beeinflussen.

5.2.1.1 Fadenanzahl (*TC*)

Die Anzahl der Fäden (engl. *thread count*, *TC*), die ein Programm oder Teile davon ausführen, hat starke Auswirkungen auf die Performanz. Wird dieser Wert zu gering gewählt, entsteht möglicherweise zu wenig Parallelität, um die verfügbaren Rechenkerne sinnvoll auszulasten. Auf der anderen Seite führen zu viele Fäden und der damit verbundene Mehraufwand bezüglich Koordination und Speicherverbrauch zu Verlangsamungen.

5.2.1.2 Ausführmodus (*EM*)

Der Ausführmodus (engl. *execution mode*, *EM*) legt fest, ob ein bestimmter Programmteil parallel oder sequenziell ausgeführt werden soll. Die Entscheidung ist beispielsweise von besonderer Bedeutung für Programme, die verschachtelte parallele Berechnungen durchführen. Parallelität auf zu niedriger Abstraktionsebene kann die Performanz negativ beeinflussen, wenn die Koordinations- und Synchronisationskosten die zu erwartende Beschleunigung dominieren. Ab einer bestimmten Verschachtelungstiefe ist es also sinnvoller, den parallelen Code sequenziell auszuführen.

5.2.1.3 Replikationsfaktor (*RF*)

Der Replikationsfaktor (engl. *replication factor*, *RF*) gibt an, wie viele Instanzen einer Programmkomponente erzeugt werden. Werden diese Instanzen parallel ausgeführt, können Aufgaben i.d.R. schneller berechnet werden. Besondere Bedeutung besitzt dieser Parameter für Auftraggeber-Auftragnehmer- und Erzeuger-Verbraucher-Muster sowie Fließbandstrukturen. Im Kontext der Stromverarbeitung ist der Replikationsfaktor vor allem für einzelne Filter relevant, um den Durchsatz des Stromprogramms zu erhöhen. Hierfür muss ein Filter zustandslos sein, d.h. die einzelnen Aktionen müssen voneinander unabhängig sein (vgl. Abschnitt 4.3).

Bei der Wahl des Replikationsfaktors gelten ähnliche Überlegungen wie beim Parameter „Fadenanzahl“: ein zu niedriger Wert beschränkt den Parallelitätsgrad, während ein zu hoher Wert den Mehraufwand zur Verwaltung der Replikate steigert und so der möglichen Beschleunigung entgegenwirkt.

5.2.1.4 Stufenfusion (*SF*)

Stufenfusion (engl. *stage fusion*, *SF*) ist die funktionale Komposition von Fließbandstufen. Bei der Ausführung von Fließbändern muss entschieden werden, welche Stufen auf welche Fäden abgebildet werden. Im Standardfall wird jede Stufe von einem Faden ausgeführt. Im Falle von n Stufen würden somit n Fäden und $n-1$ Puffer zwischen den Stufen benötigt. Je größer n, desto größer ist der Mehraufwand für die Pufferung.

Durch Stufenfusion kann dieser Mehraufwand reduziert werden, indem die Anzahl tatsächlicher Stufen und somit die Anzahl der benötigten Fäden und Puffer verringert wird. Werden allerdings zu viele Stufen zusammengefasst, schrumpft die maximal mögliche Beschleunigung. Im Extremfall, d.h. bei der Verschmelzung aller Fließbandstufen zu einer einzigen (nicht replizierbaren) Stufe, wird das Fließband von einem einzigen Faden und somit sequenziell ausgeführt.

5.2.1.5 Alternative (*AL*)

Der Parameter „Alternative“ (engl. *alternative*, *AL*) bezeichnet eine Auswahl aus einer Menge funktional äquivalenter Implementierungen einer Berechnung. Zur Lösung eines Teilproblems existieren unter Umständen unterschiedliche Implementierungen, die zwar dieselbe Funktionalität, aber unterschiedliche Laufzeiteigenschaften (z.B. Ausführungsdauer oder Speicherverbrauch) besitzen. Beispielsweise können für das Sortieren eines Arrays verschiedene Verfahren implementiert sein.

Aufgrund der Vielfalt paralleler Rechnerarchitekturen und der Beschaffenheit möglicher Eingabedaten kann die Frage nach der besten Implementierung i.d.R. nicht allgemeingültig beantwortet werden. Aus diesem Grund kann es sinnvoll sein, mehrere Alternativen im Programm zu definieren und einen Auto-Tuner die Auswahl treffen zu lassen.

5.2.1.6 Aktionszahl (*AC*)

Der Parameter „Aktionszahl“ (engl. *action count*, *AC*) ist von Bedeutung bei der Ablaufplanung von periodischen Aufgaben. Periodische Aufgaben, z.B. periodische Filter, besitzen eine Schleife, die eine Folge von gleichartigen Berechnungen (Aktionen) durchführt. Beispielsweise handelt es sich bei Fließbandstufen oder Arbeitern einer Auftraggeber-Auftragnehmer-Implementierung um periodische Aufgaben; die Verarbeitung eines Arbeitspakets entspricht einer Aktion.

Periodische Aufgaben werden oftmals durch Ausführungsdienste ausgeführt, wobei die Anzahl der Ausführungsfäden typischerweise kleiner ist als die Anzahl der Aufgaben. Eine Aufgabe wird dabei i.d.R. nicht zusammenhängend, sondern stückweise ausgeführt[2]. Der Parameter „Aktionszahl“ legt fest, wie viele Aktionen einer Aufgabe am Stück auszuführen sind. Ein sehr geringer Wert führt zu einem erhöhten Mehraufwand durch häufiges Umschalten zwischen den Aufgaben; ein zu hoher Wert kann zu ungleichmäßig ausgelasteten Fließbandstufen und überfüllten Puffern führen.

5.2.2 Erfassung von Tuningparametern und Kontextinformationen

Um ein Programm optimierbar im Sinne des Auto-Tunings zu machen, müssen zunächst leistungsrelevante Parameter im Code definiert werden. Dies ist bisher Aufgabe des Programmierers. Unser Ziel ist es, bereits diesen Schritt zu automatisieren. Hierfür wird ein Konzept vorgestellt, welches Stromprogramme in der Übersetzungsphase mit Tuningparametern und Kontextinformationen instrumentiert [83]. Dabei wird ausgenutzt, dass die stromorientierten

[2] Ablaufplaner von Betriebssystemen arbeiten auf ähnliche Weise, indem sie Fäden oder Prozessen abwechselnd Zeitscheiben zuweisen, innerhalb derer ein Rechenkern benutzt werden darf.

Algorithmus 6 : Erfassung von Tuningparametern

```
1  procedure RetrieveTuningParameters(program p)
2      AddTuningParameter(p, TC, tc);                          ▷ Fadenanzahl
3      foreach parallel statement s in p do
4          AddTuningParameter(s, EM, em_s);                    ▷ Ausführmodus
5          foreach filter instance fi in s do
6              AddTuningParameter(fi, AC, ac_fi);              ▷ Aktionszahl
7              if fi is replicable ∧ fi is stateless then
8                  AddTuningParameter(fi, RF, rf_fi);          ▷ Replikationsfaktor
9              end
10             if fi has predecessor ∧ fi has no teleport then
11                 AddTuningParameter(fi, SF, sf_fi);          ▷ Stufenfusion
12             end
13         end
14     end
15     foreach alternative statement s in p do
16         AddTuningParameter(s, AL, al_s);                    ▷ Alternative
17     end
18 end
```

Sprachkonstrukte es dem Übersetzer erlauben, die parallele Struktur einer Anwendung zu erfassen und so zu entscheiden, welche Tuningparameter an welchen Stellen sinnvoll sind und in welchem Kontext sie stehen.

5.2.2.1 Erfassung von Tuningparametern

Voraussetzung für die automatische Erfassung von Tuningparametern in der Übersetzungsphase ist ein Katalog von Tuningparametern, wie er in vorigem Abschnitt beschrieben wurde. Damit lässt sich formal beschreiben, welche Tuningparametertypen in welchen Szenarien wirksam sind und auf welche konkreten Programmteile sie sich auswirken.

Algorithmus 6 skizziert ein Instrumentierungsverfahren für die zuvor beschriebenen Tuningparametertypen. Die Funktion $AddTuningParameter(e, typ, param)$ ordnet dabei einer Programmeinheit e einen Tuningparameter $param$ vom Typ typ zu. Eine Programmeinheit ist ein Teil des Syntaxbaums eines Programms, also eine Filterinstanz, eine Filterdeklaration, eine parallele Anweisung oder das gesamte Programm. Die Wertebereiche der Tuningparameter werden unmittelbar vor der Ausführung festgelegt (vgl. Abschnitt 5.2.3).

Insgesamt werden sechs Typen von Tuningparametern berücksichtigt, die sich auf unterschiedliche Programmeinheiten auswirken:

- Dem gesamten Programm wird ein Tuningparameter des Typs „Fadenanzahl“ (TC) zugeordnet. Dieser Parameter bezieht sich auf die Anzahl der Fäden, aus denen der Ausführungsdienst besteht, der für die Ausführung der Filter des Stromprogramms zuständig ist.
- Jeder parallelen Anweisung wird ein Parameter *EM* zugewiesen, der spezifiziert, ob die Anweisung parallel oder sequenziell auszuführen ist.
- Jeder Filterinstanz wird ein Parameter *AC* zugewiesen, der die Anzahl zusammenhängend durchzuführender Aktionen angibt. Dieser Parametertyp wirkt sich nur auf periodische Filter aus; ob eine Filterinstanz tatsächlich periodisch ist, lässt sich jedoch erst während der Ausführung feststellen. Ist dies nicht der Fall, ist der Parameter *AC* bedeutungslos und wird deaktiviert.
- Ist eine Filterinstanz als replizierbar gekennzeichnet, so erhält sie einen Parameter *RF* für den Replikationsfaktor. Voraussetzung der Replizierbarkeit ist Zustandslosigkeit, die mit statischen Analysen geprüft wird (vgl. Abschnit 4.3.2.2).
- Ist eine Filterinstanz teleportfrei[3] und Bestandteil eines Fließbands, erhält sie einen Tuningparameter *SF*. Dieser Parameter gibt an, ob der Filter mit dem Vorgängerfilter verschmolzen werden soll.
- Einer Alternativenanweisung, bestehend aus Filterinstanzen $f_{alt1}, ..., f_{altN}$, wird ein Tuningparameter vom Typ *AL* zugeordnet. Der Wert dieses Parameters bezeichnet die Auswahl einer Alternative.

5.2.2.2 Erfassung von Kontextinformationen

Im vorigen Abschnitt wurde erläutert, wie anhand der stromorientierten Syntax automatisiert Tuningparameter identifiziert und Codestellen zugeordnet werden können. In der Arbeit von Schaefer [91] wurde die Idee vorgestellt, Tuningparameter mit Kontextinformationen zu verknüpfen, die es dem Auto-Tuner erlauben, schneller „gute“ Parameterwerte zu finden. Kontextinformationen sind z.B. der Verwendungszwecks eines Parameters oder Abhängigkeiten von anderen Parametern.

Wir übertragen diese Idee auf das Strommodell, so dass Kontextinformationen automatisch aus der stromorientierten Syntax gewonnen werden können. In diesem Zusammenhang beschränken sich die Kontextinformationen nicht auf Tuningparameter, sondern werden auch für Filterinstanzen erfasst, um Wissen über die parallele Struktur des Stromprogramms zu gewinnen. Diese Informationen werden schließlich dazu verwendet, Parameterwerte möglichst

[3] Eine Filterinstanz, mit deren Eingabekanal ein Teleport assoziiert ist, kann nicht mit dem Vorgängerfilter verschmolzen werden, da sonst der Teleport elimiert würde.

gut vorherzusagen. Die so voreingestellten Parameter können einerseits als gute Startkonfiguration für einen Auto-Tuner dienen, andererseits den Suchraum deutlich reduzieren.

Kontextinformationen für einen Tuningparameter tp sind:

- (K1-tp) Parametertyp gemäß der Kategorisierung in Abschnitt 5.2.1. Die Kenntnis des Typs ermöglicht es, Wertebereiche und Standardwerte für tp sinnvoll festzulegen.

- (K2-tp) Gültigkeitsbereich bzw. die Programmeinheit, deren Ausführungsverhalten durch den Parameter tp beeinflusst wird. Mit diesen Informationen kann der Suchraum partitioniert werden.

- (K3-tp) Wertehistorie und aktueller Wert von tp sowie Kennzeichnung „guter“ Parameterwerte. Optimierungen können so zielgerichteter durchgeführt werden, da ggf. Tendenzen erkennbar sind, in welche „Richtung“ sich die Modifikation des Parameterwerts lohnen könnte.

Kontextinformationen für Filterinstanzen fi sind:

- (K1-fi) Replizierbarkeit: kann fi repliziert werden?

- (K2-fi) Periodizität: handelt es sich bei fi um einen periodischen oder nichtperiodischen Filter? Anders ausgedrückt: besitzt die zugehörige Filterdeklaration einen work-Block?

- (K3-fi) Filterabhängigkeiten: zu welchen anderen Filterinstanzen bestehen direkte Eingabe- bzw. Ausgabeabhängigkeiten, d.h. welche Filter fungieren als Produzenten bzw. Konsumenten für fi?

- (K4-fi) Elternfilter: wird fi innerhalb eines Filters aufgerufen?

- (K5-fi) Verschachtelungstiefe: auf welcher Parallelitätsebene befindet sich fi?

- (K6-fi) Aktuelle Arbeitslast: wie viele Stromelemente befinden sich im Eingabepuffer von fi?

Man beachte, dass Kontextinformationen zwar beim Übersetzen in den Code eingebaut werden, tlw. aber erst zur Laufzeit aufbereitet bzw. abgefragt werden können. Dies betrifft vor allem Informationen über Filterinstanzen, da diese dynamisch erzeugt werden. Beispielsweise wird die Verschachtelungstiefe (K5-fi) einer Filterinstanz rekursiv, d.h. anhand der Verschachtelungstiefe des Elternfilters, zur Laufzeit bestimmt.

Parametertyp	Bezug	Wertebereich	Heuristik
TC	Programm p	$\{1,...,2n\}$	n
EM	ParStmt s	$\{par, seq\}$	$seq \Leftrightarrow$ $availableExecutors = \emptyset$
RF	Filterinstanz fi	$\{1,...,2n\}$	n
SF	Filterinstanz fi	$\{true, false\}$	$true \Leftrightarrow$ $fi, pre(fi)$ replizierbar
AL	AlternativeStmt s	$\{a_1,...,a_m\}$	a_1
AC	Filterinstanz fi	$\mathbb{N}$	100

Tabelle 5.1: Heuristiken zur Parametervorhersage (n = Anzahl der Rechenkerne)

5.2.3 Vorkonfiguration von Tuningparametern

Ein übersetztes Stromprogramm weist gemäß Abschnitt 5.2.2 Tuningparameter auf, die zur Laufzeit eingestellt werden müssen. In diesem Abschnitt schlagen wir Heuristiken vor, mit denen Tuningparameter unter Berücksichtigung von Kontextwissen vorkonfiguriert werden können [83].

Die Evaluation in Kapitel 7 zeigt, dass diese Heuristiken in vielen Fällen bereits zu guten Programmleistungen führen. Da sich die Vorkonfiguration oftmals in einem vielversprechenden Bereich des Suchraums befindet, ergeben sich zudem Vorteile für Laufzeit-Tuning.

Tabelle 5.1 fasst unsere Heuristiken für die jeweiligen Tuningparametertypen zusammen. Im Folgenden wird die Wahl der Heuristiken und Wertebereiche motiviert.

5.2.3.1 Vorkonfiguration der Fadenanzahl (TC)

Besitzt ein Rechner n Kerne, so ist die intuitive Wahl für die Anzahl der Fäden n. Dennoch kann es sinnvoll sein, mehr Fäden als Kerne zu beschäftigen, um Wartezeiten einzelner Fäden auszugleichen und dadurch die Kerne besser auszulasten. Daher verwenden wir die Heuristik $TC := n$ und den Wertebereich $dom(TC) := \{1,...,2n\}$.

5.2.3.2 Vorkonfiguration des Ausführungsmodus (EM)

Die Entscheidung, ob eine parallele Anweisung parallel (*par*) oder sequenziell (*seq*) ausgeführt werden soll, wird abhängig von der Verfügbarkeit der Ausführungsfäden getroffen. Filter werden somit genau dann sequenziell ausgeführt, wenn aktuell keine Ausführungsfäden zur Verfügung stehen. Dabei kann nicht nur von paralleler auf sequenzielle Ausführung „umgeschaltet“ werden; auch die Umkehrung ist möglich. Startet z.B. ein sequenziell ausgeführter Filter weitere Filter, so können diese wieder parallel ausgeführt werden, wenn in

der Zwischenzeit Ausführungsfäden wieder verfügbar sind. Der Wertebereich für den Parametertyp EM ist also $dom(EM) := \{par, seq\}$. Wenn alle Ausführungsfäden beschäftigt sind, wird $EM := seq$ gewählt, andernfalls ist $EM := par$. Eine Anpassung durch Laufzeit-Tuning basierend auf Leistungsmessungen erfolgt daher nicht.

5.2.3.3 Vorkonfiguration des Replikationsfaktors (RF)

Der Replikationsfaktor für eine Filterinstanz lässt sich intuitiv anhand der Anzahl der Rechenkerne festlegen, um im Idealfall eine Auslastung aller Kerne zu erreichen. Hierbei gelten ähnliche Überlegungen wie für die Wahl der Fadenanzahl. Als Heuristik für diesen Parametertyp wird demnach $RF := n$ gewählt. Der Wertebereich wird mit $dom(RF) := \{1, ..., 2n\}$ festgelegt.

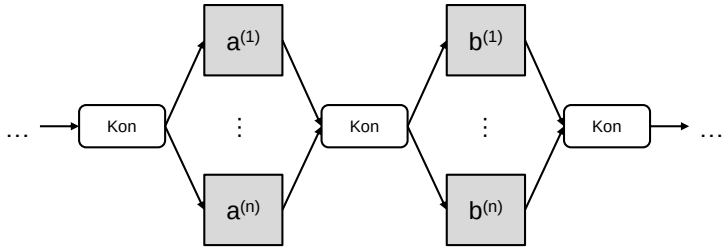
5.2.3.4 Vorkonfiguration der Stufenfusion (SF)

In einem Fließband kann jede Stufe grundsätzlich von einem eigenen Faden ausgeführt werden. Der Tuningparameter „Stufenfusion“ (SF) einer Filterinstanz fi, die Teil eines Fließbands ist, gibt an, ob fi mit der Vorgängerstufe $pre(fi)$ verschmolzen werden soll. Der Parameter besitzt also den Wertebereich $dom(SF) = \{true, false\}$. Insbesondere bei aufeinander folgenden replizierbaren Stufen entsteht Mehraufwand für das Aufteilen und Zusammenführen von Stromelementen, was zu Flaschenhalseffekten führen kann. Daher wird die Heuristik angewendet, direkt aufeinander folgende replizierbare Filterinstanzen zu verschmelzen. Somit ist $SF(fi) = true$ wenn sowohl fi als auch $pre(fi)$ replizierbar sind, andernfalls wird $SF(fi) = false$ gewählt.

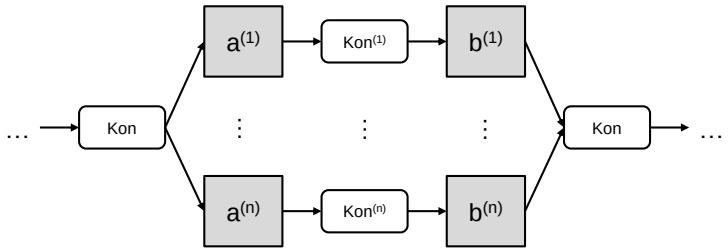
Um allein den Mehraufwand für das Aufteilen und Zusammenführen von Strömen zu reduzieren, wäre es natürlich auch denkbar, keine Fusion replizierbarer Filterinstanzen durchzuführen, sondern für jedes Paar von Filterinstanzen einen eigenen Kanal bzw. Konnektor zu verwenden. Unsere Heuristik basiert jedoch auf der Annahme, dass neben dem Aufwand für das Aufteilen und Zusammenführen von Strömen auch die Anzahl der Konnektoren sowie die Anzahl der resultierenden Filterinstanzen Kosten verursachen, die es zu minimieren gilt. Abbildung 5.2 illustriert diese Überlegung für zwei replizierbare Filter a und b.

Da in der objektorientierten Stromprogrammierung polymorphe Filteraufrufe möglich sind, erfolgt die Stufenfusion dynamisch. Für jeden Filteraufruf werden wie gewöhnlich Filterinstanzen erzeugt. Um hierfür eine Stufenfusion durchzuführen, werden die Filterinstanzen unmittelbar vor ihrer Ausführung zu einer einzigen Filterinstanz verknüpft, welche schließlich dem Ablaufplaner zur Ausführung übergeben wird.

Ohne Fusion:
$2n$ Filterinstanzen, 3 Konnektoren, 2 Splits, 2 Joins



Ohne Fusion, aber mit exklusiven Konnektoren:
$2n$ Filterinstanzen, $n+2$ Konnektoren, 1 Split, 1 Join



Mit Fusion:
n Filterinstanzen, 2 Konnektoren, 1 Split, 1 Join

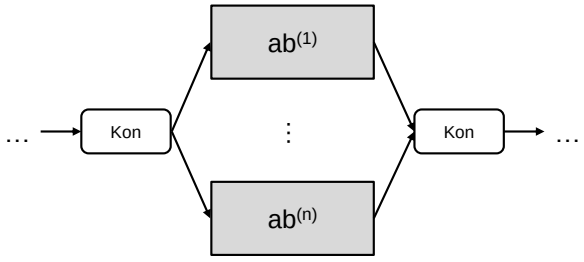


Abbildung 5.2: Fusion benachbarter replizierbarer Stufen

5.2.3.5 Vorkonfiguration von Alternativen (*AL*)

Eine Alternativenmenge besteht aus „alternativen“ Filterinstanzen $a_1, ..., a_m$, welche dieselbe Funktionalität auf unterschiedliche Weise implementieren. Aus dieser Menge ist diejenige Alternative auszuwählen, welche die beste Performanz liefert. Es ist also $dom(AL) := \{a_1, ..., a_m\}$, wobei a_i bedeutet, dass Implementierung a_i ausgewählt ist und alle anderen Filterinstanzen $a_1, ..., a_{i-1}, a_{i+1}, ..., a_m$ deaktiviert sind. Die Auswahlentscheidung ist i.d.R. abhängig von der zugrunde liegenden Rechnerarchitektur. Somit wird zunächst $AL := a_1$ festgelegt, wobei diese Auswahl durch Laufzeit-Tuning angepasst werden kann (vgl. Abschnitt 5.3).

5.2.3.6 Vorkonfiguration der Aktionszahl (*AC*)

Die optimale Aktionszahl lässt sich ähnlich wie die Auswahl einer Alternative schwer vorhersagen. Als Standardwert wird $AC := 100$ festgelegt; der Wertebereich ist $dom(AC) := \mathbb{N}$.

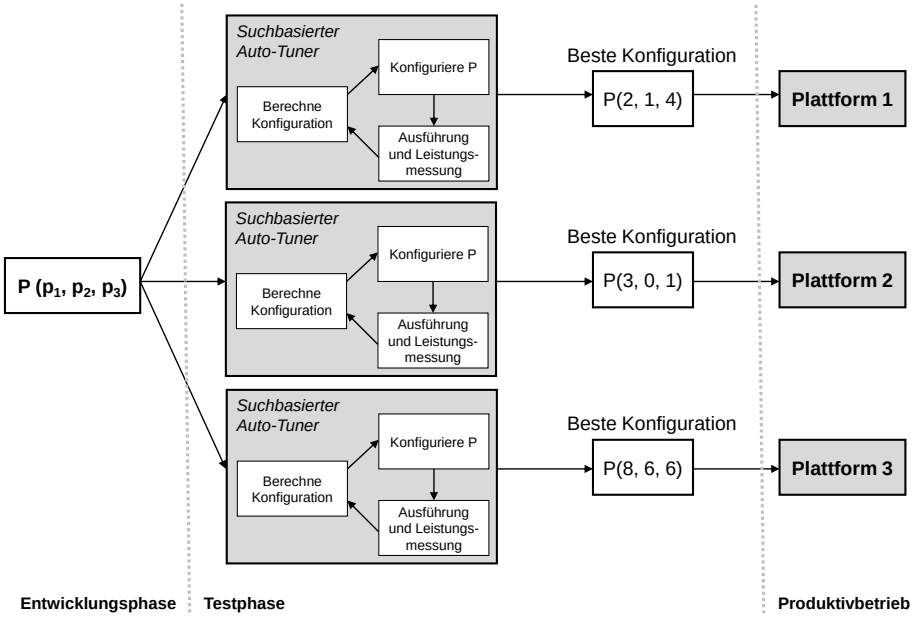
5.3 Laufzeit-Tuning für Stromprogramme

In diesem Abschnitt stellen wir unsere Konzepte zum laufzeitbasierten Auto-Tuning (Laufzeit-Tuning) von Stromprogrammen vor [105]. Die Konzepte erlauben es, die voreingestellten Parameterkonfigurationen während der Programmausführung zu verbessern. Hierbei wird die Eigenschaft ausgenutzt, dass Stromprogramme schleifenartige Strukturen aufweisen und somit eine Form von Periodizität bzw. Wiederholung gegeben ist. Wiederholbarkeit ist eine Voraussetzung dafür, Messabschnitte definieren zu können und Rückkopplungen zwischen gemessenen Leistungswerten und der Anpassung von Parameterwerten zu ermöglichen.

Laufzeit-Tuning bringt zwei Herausforderungen mit sich: da der Optimierungsprozess zur Laufzeit stattfindet, darf der dadurch entstehende Mehraufwand nicht die Performanz des gesamten Anwendungsprogramms nicht negativ beeinflussen. Des Weiteren sollte der verwendete Optimierungsalgorithmus möglichst schnell eine gute Parameterkonfiguration finden. Hierfür liefern die im vorigen Abschnitt erläuterten Heuristiken oftmals bereits gute Startwerte.

In Kapitel 2 wurden mit Offline- und Online-Tuning zwei grundlegende Möglichkeiten vorgestellt, um Programme vor bzw. während der Ausführung zu optimieren, sowie die Vor- und Nachteile beider Ansätze erläutert. Abbildung 5.3 stellt diese Verfahren beispielhaft gegenüber. Die beste Parameterkonfiguration für ein Programm ist von Rechner zu Rechner unterschiedlich und muss i.d.R. separat bestimmt werden. Gegenüber statischen Programmoptimierungen, die für Stromprogramme meist zum Einsatz kommen, sowie suchbasiertem Offline-Tuning verspricht Laufzeit-Tuning einige Vorteile:

Offline-Tuning für ein Programm mit den Tuningparametern p_1, p_2, p_3:



Laufzeit-Tuning (Online-Tuning) für ein Programm mit den Tuningparametern p_1, p_2, p_3:

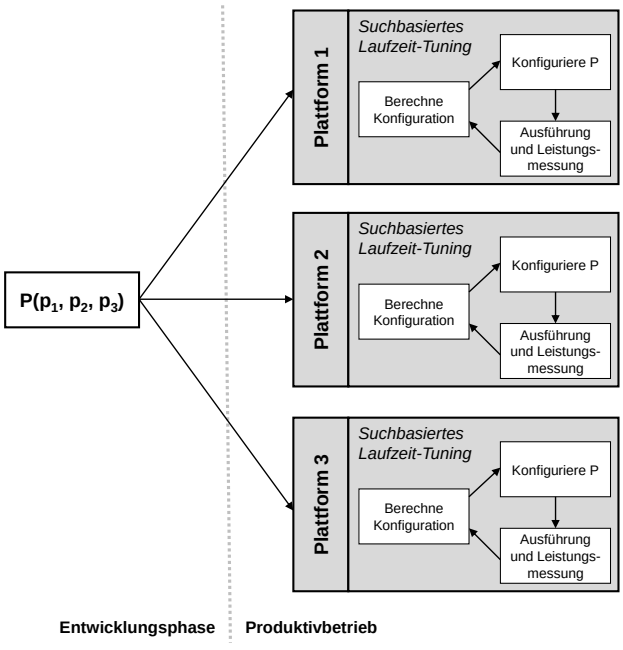


Abbildung 5.3: Gegenüberstellung von Offline- und Laufzeit-Tuning

- Da die Optimierung zur Laufzeit, also im Produktivbetrieb der Anwendung stattfindet, sind im Unterschied zu Offline-Tuning keine plattformspezifischen Testläufe notwendig.
- Die Optimierung ist unabhängig von der Beschaffenheit der Rechnerarchitektur und der Eingabedaten. Auf die Ausführungsdynamik eines Programms kann reagiert werden, da Parameterwerte verändert werden können.

Die Integration von Laufzeit-Tuning in das stromorientierte Programmiermodell lässt sich in folgende Teilprobleme zerlegen:

1. Schaffung einer Möglichkeit, um Tuningparameter und deren Wertebereiche automatisch dem Auto-Tuner zu übergeben (Abschnitt 5.3.1)
2. Entwicklung eines Verfahrens zur Leistungsbestimmung von Stromprogrammen mithilfe automatisch definierter Messabschnitte (Abschnitt 5.3.2)
3. Konzeption eines Mechanismus, der es erlaubt, anhand von gemessenen Leistungswerten neue Parameterkonfigurationen zu bestimmen (Abschnitt 5.3.3) und diese während der Programmausführung zu übernehmen (Abschnitt 5.3.4)

Die Lösungen für die genannten Teilprobleme ergeben in der Synthese einen erweiterten Tuningzyklus, der die Durchführung von Laufzeit-Tuning für Stromprogramme ermöglicht. Der Tuningzyklus ist Gegenstand von Abschnitt 5.3.6.

Zur Umsetzung unseres Optimierungsverfahrens verwenden wir den Auto-Tuner von Karcher et al. [52]. Dieser ist als Kernmodul für das Betriebssystem Linux implementiert und stellt im Wesentlichen drei tuningspezifische Systemaufrufe bereit:

- $addParameter(v)$ markiert eine Variable v als Tuningparameter und erlaubt somit dem Auto-Tuner, dessen Wert innerhalb des definierten Wertebereichs anzupassen.
- $startMeasurement()$ legt den Beginn eines Messabschnitts fest und startet die Leistungsmessung.
- $stopMeasurement()$ legt das Ende eines Messabschnitts fest, d.h. stoppt die Messung und berechnet eine neue Parameterkonfiguration.

5.3.1 Übergabe von Tuningparametern

In Abschnitt 5.2.1 wurden die wichtigsten Tuningparameter sowie deren automatische Identifikation durch den Übersetzer erläutert. In diesem Abschnitt beschreiben wir die Übergabe der Tuningparameter an den Auto-Tuner.

Nach Definition 9 in Kapitel 2 handelt es sich bei einem Tuningparameter um eine Programmvariable mit einem diskreten, endlichen Werteberich. Ein Wertebereich lässt sich daher auf

eine Teilmenge der natürlichen Zahlen abbilden. Somit können alle Tuningparameter einheitlich beschrieben und dem Auto-Tuner übergeben werden. Wir definieren hierfür mithilfe eine Klasse `TunableInteger`; deren Aufbau ist in Listing 5.1 skizziert.

Listing 5.1: Aufbau der Klasse `TunableInteger`

```
1 public class TunableInteger {
2   public TunableInteger(int init, int min, int max, int step) { ... }
3   public int getValue() { ... }
4 }
```

Der Konstruktor erzeugt eine neue Instanz eines Tuningparameters und kennzeichnet diesen für den Auto-Tuner mithilfe eines entsprechenden *addParameter*-Systemaufrufs. Die Aufrufparameter legen die Eigenschaften des Tuningparameters fest:

- *init* bezeichnet den Standard- bzw. Startwert des Tuningparameters.
- *min* und *max* definieren die Unter- und Obergrenze des Wertebereichs.
- *step* definiert die Schrittweite und somit die „Dichte“ des Wertebereichs.

Der Konstruktoraufruf `new TunableInteger(4, 1, 11, 3)` würde also einen Tuningparameter mit dem Standardwert 4 und dem Wertebereich $\{1,4,7,10\}$ erzeugen.

Nach diesem Schema lassen sich auch die in Tabelle 5.1 aufgeführten Tuningparameter erfassen und dem Auto-Tuner übergeben, beispielsweise

- Ausführmodus (*EM*): `new TunableInteger(h, 1, 2, 1)`, wobei $par := 1, seq := 2$ und h nach der Heuristik in Tabelle 5.1 berechnet wird
- Replikationsfaktor (*RF*): `new TunableInteger(n, 1, 2*n, 1)`, wobei n die Anzahl der Rechenkerne ist
- Alternative (*AL*): `new TunableInteger(1, 1, m, 1)`, wobei m die Anzahl der Alternativen ist

5.3.2 Leistungsbestimmung

Um Laufzeit-Tuning zu ermöglichen, muss ein Weg gefunden werden, um die Gesamtleistung eines Stromprogramms während der Ausführung zu bestimmen. Im Einzelnen erstrecken sich die Herausforderungen auf

- die Wahl des Leistungskriteriums zur Bewertung der Gesamtleistung des Programms,
- die Platzierung der Messpunkte sowie
- die Wahl der Messdauer.

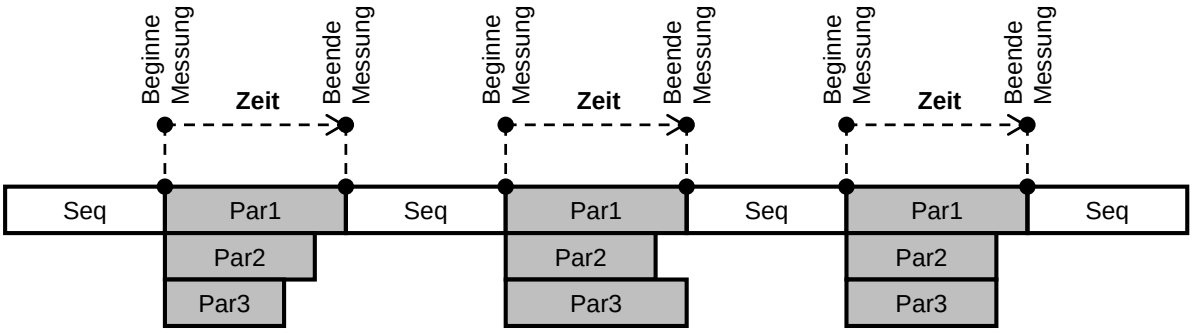


Abbildung 5.4: Messabschnitte in einem Programm mit Fork-Join-Parallelität

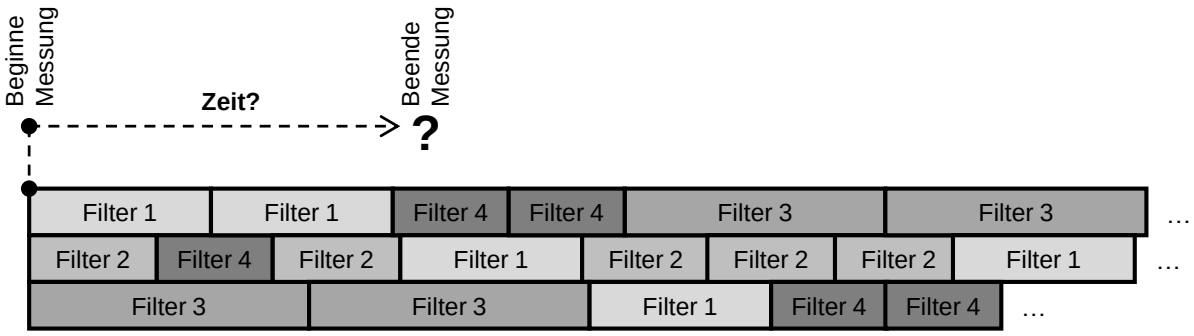


Abbildung 5.5: Problematik von Messabschnitten in Stromprogrammen

Die Problematik der Leistungsbestimmung für ein stromorientiertes Programm lässt sich im Vergleich zu einem Programm mit Fork-Join-Parallelität motivieren. Abbildung 5.4 zeigt exemplarisch den Programmablauf eines dreifädigen Programms mit Fork-Join-Parallelität. Solche Programme unterteilen sich in sequenzielle und parallele Abschnitte, die zeitlich nacheinander ausgeführt werden. Nimmt man an, dass die parallelen Abschnitte in eine Schleife eingebettet sind, so lassen sich diese mit Messabschnitten assoziieren. Stromprogramme

zeichnen sich hingegen dadurch aus, dass die Filter zeitlich überlappend ausgeführt werden. Ein beispielhafter Programmablauf ist in Abbildung 5.5 dargestellt. Somit ist es nicht möglich, einen globalen Messabschnitt zu definieren, der regelmäßig durchlaufen wird.

5.3.2.1 Leistungskriterium

Eine naheliegende Lösung wäre es, den work-Block eines Filters mit einem Messabschnitt zu assoziieren und die Zeit zu messen, die für die Durchführung einer Aktion benötigt wird. Auf diese Weise würden sich also mehrere lokale Messabschnitte ergeben, die sich überlagern. Die einzelnen Messungen sind jedoch kein aussagekräftiger Indikator bezüglich der Gesamtleistung des Programms. Tuningparameter beziehen sich meist nicht auf einzelne Filterinstanzen, sondern auf (veränderliche) Strukturen aus mehreren Filtern. Daher lässt sich ein solcher Parameter nicht einer bestimmten Filterinstanz zuordnen. Der für eine lokale Filteriteration bestimmte Leistungswert spiegelt somit nicht die Leistung des Gesamtprogramms wider. Ein Beispiel ist die Replikation eines Filters f, durch die n Instanzen $fi_1, ..., fi_n$ entstehen. Die Zeit für eine Aktion durch eine Filterinstanz fi_i ist unabhängig vom Wert n. Vielmehr sollte bestimmt werden, wie viele Stromelemente pro Zeit verarbeitet werden können.

Diese Überlegung führt schließlich dazu, als Leistungskriterium den Durchsatz des Gesamtprogramms zu betrachten. Die Messung des Durchsatzes kann basierend auf dem Konzept des Herzschlags (engl. *application hearbeat*) erfolgen [47]. Messpunkte generieren Ereignisse, die an zentraler Stelle gezählt werden. Dies führt uns zur Definition des Pulses.

Definition 13 *Der* Puls *eines Programms wird erzeugt durch seine Messpunkte, welche bei Erreichen ein Ereignis auslösen. Der Puls entspricht der Ereignisfrequenz (ausgelöste Ereignisse pro Zeiteinheit).*

Der Puls lässt sich als Maß für den Durchsatz eines Programms verwenden. Werden die Messpunkte eines Programms regelmäßig erreicht, so ist die Gesamtleistung des Programms umso besser, je höher sein Puls ist.

Mithilfe des Pulses ist es möglich, die individuellen Leistungswerte verschiedener Filterinstanzen zu einem einzigen Leistungswert zusammenzufassen. Somit entsteht ein globaler Messabschnitt, dem sämtliche Tuningparameter zugeordnet werden können. Dadurch lässt sich die für Laufzeit-Tuning essentielle Rückkopplung zwischen Parameterkonfiguration und Programmleistung herstellen.

5.3.2.2 Messpunkte

Im vorigen Abschnitt wurde die pulsbasierte Leistungsmessung für Stromprogramme motiviert. Nun muss entschieden werden, an welchen Stellen im Programm ein Messpunkt gesetzt und somit ein Ereignis ausgelöst werden soll.

Damit der Puls den Durchsatz eines Programms möglichst gut approximiert, werden nur solche Filter mit Messpunkten ausgestattet, die als Senke fungieren, also keinen Ausgabestrom produzieren. Stark vereinfacht lässt sich die Messschleife zur Leistungsbestimmung also wie folgt auffassen:

```
foreach elem in stream
  process(elem);
  generateEvent();
end
```

Somit wird für jedes Stromelement, welches den Stromgraphen vollständig durchlaufen hat, ein Ereignis erzeugt. Bei der gewählten Platzierung der Messpunkte gehen wir davon aus, dass der von Senken empfangene Datenstrom hinreichend lang ist und Senken regelmäßig erreicht werden. Die Grenzen unseres Messverfahrens zeigen sich beispielsweise für Stromgraphen mit ungünstig platzierten Teleports, so dass der Datenstrom zunächst sehr viele Zyklen durchläuft, bevor eine Senke erreicht wird.

Die durch Messpunkte hervorgerufenen Ereignisse werden an zentraler Stelle gezählt. Der Mehraufwand für das Zählen der Ereignisse ist dabei vernachlässigbar. Zum einen handelt es sich bei dem Auslösen eines Ereignisses um eine nicht-blockierende Operation, welche somit die Ausführung eines Filters nicht verzögert. Zum anderen ist die Anzahl der Messpunkte bzw. der dadurch ausgelösten Ereignisse lediglich abhängig von der Anzahl der Senken, nicht aber von der Größe des gesamten Stromgraphen. Da Senken gewöhnlich nur einen kleinen Teil des Stromgraphen darstellen, kann davon ausgegangen werden, dass auch für Programme, die aus sehr vielen Filtern bestehen, das zentrale Zählen der Ereignisse nicht zum Engpass wird.

5.3.2.3 Messdauer

Programme mit explizit definierten Messabschnitten besitzen die Form

```
while(cond) {
  startMeasurement();
  doSomething(p1, p2);
  stopMeasurement();
}
```

sodass die Messdauer durch die Zeit bestimmt ist, die für die Ausführung der Anweisungen zwischen *startMeasurement*() und *stopMeasurement*() benötigt werden. Im Beispiel handelt es sich also um die Laufzeit eines Aufrufs der Methode `doSomething`, deren Leistung von zwei Tuningparametern `p1` und `p2` beeinflusst wird.

Stromorientierte Programme besitzen dagegen nur implizit definierte Messabschnitte, aus denen die Messdauer nicht direkt hervorgeht. Es existieren zwar filterlokale Messpunkte, die Ereignisse zur Bestimmung der Gesamtleistung des Programms generieren; es wäre jedoch nicht sinnvoll, beim Erreichen eines jeden Messpunkts die Zeitmessung zu stoppen.

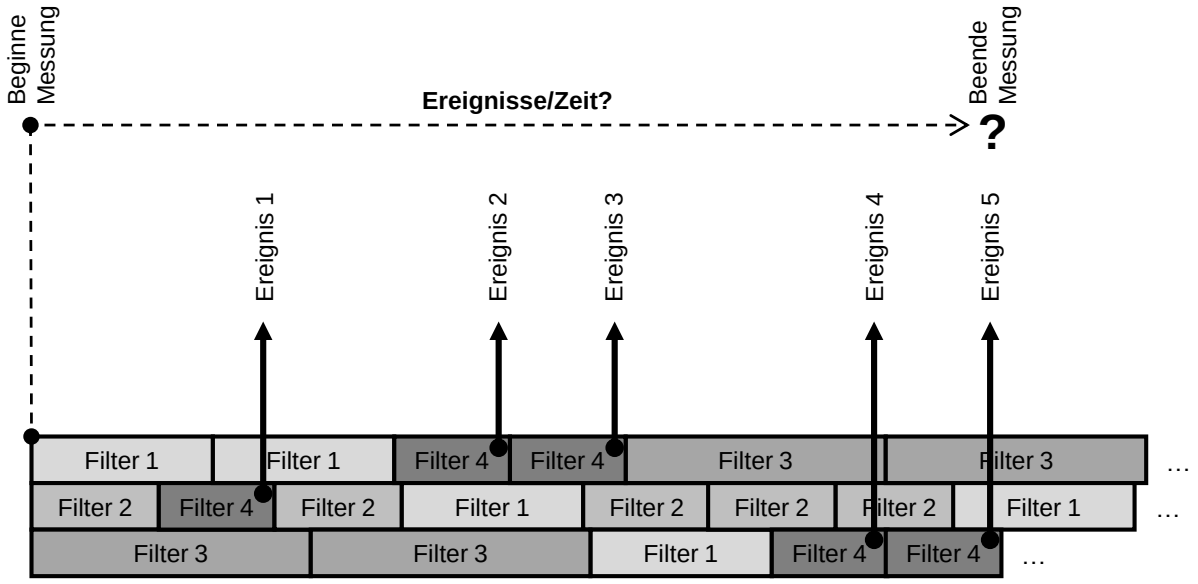


Abbildung 5.6: Wahl der Messdauer in Stromprogrammen

Zur Illustration greift Abbildung 5.6 den beispielhaften Programmablauf eines Stromprogramms aus Abbildung 5.5 auf. Die durchgezogenen Pfeile illustrieren die ausgelösten Ereignisse, wobei wir annehmen, dass Filter 4 als Senke fungiert. Aus der Grafik wird ersichtlich, dass der Abschluss einer Messung nach jedem Ereignis im Allgemeinen zu schwankenden Messwerten führen würde. Sinnvoller ist es, nur nach jedem *n*-ten Ereignis eine Messung zu stoppen. *n* nennen wir im Folgenden Ereigniszahl.

Es stellt sich die Frage nach einer „guten“ Ereigniszahl. Eine zu niedrige Zahl führt zu starken Messschwankungen, eine zu hohe Zahl zu gleichmäßigen, aber langen Messdauern, die letztendlich den gesamten Optimierungsprozess verlangsamen würden. Statt die Ereigniszahl festzulegen, könnte auch eine feste Messdauer eingestellt und die Ereignisse in diesem Zeitraum gezählt werden. Der Nachteil dieser Variante ist, dass eine feste Messdauer unabhängig von den Programmeigenschaften nicht sinnvoll gewählt werden kann. Für Programme, die aufgrund ihrer Komplexität zu selten Messereignisse auslösen, könnten Messungen ohne

Ereignisse entstehen, was fälschlicherweise als Leistungsverschlechterung bewertet werden würde.

Dem gegenüber steht die Möglichkeit, die Anzahl der Messereignisse, die den Abschluss einer Messung definieren, konstant zu halten. Eine prinzipielle Eignung dieses Ansatzes für unterschiedlich rechenintensive Stromprogramme wäre zwar gegeben; allerdings bestünde hier die Gefahr, dass die Messdauern in Programmen mit weniger aufwändigen Filtern zu kurz werden. Dies kann zu unverhältnismäßig hohem Aufwand für Auto-Tuning führen.

Zur Lösung der Messdauerproblematik wird daher ein Mittelweg zwischen diesen beiden Ansätzen gewählt. Die Ereigniszahl n wird so eingestellt, dass sie dem Parallelitätsgrad der zugrundeliegenden Rechnerarchitektur bzw. der Anzahl der verfügbaren Rechenkerne entspricht. Eine Messung wird abgeschlossen, wenn mindestens n Ereignisse gezählt wurden und eine Mindestmessdauer t_{min} erreicht ist. So wird sichergestellt, dass einerseits die Messdauer nicht zu gering ist, andererseits genügend Ereignisse vorliegen, damit die Ereignisfrequenz und somit die Leistungsbewertung für den Auto-Tuner aussagekräftig ist.

5.3.2.4 Bestimmung des Leistungswerts

Das Starten und Beenden einer Messung erfolgt durch die Systemaufrufe $startMeasurement()$ und $stopMeasurement()$, die vom verwendeten Auto-Tuner [52] bereitgestellt werden. Grundsätzlich wird die erste Messung dann gestartet, wenn das erste Stromelement den Stromgraphen passiert. In Abschnitt 5.3.2.3 wurde festgelegt, dass eine Messung beendet wird, wenn eine bestimmte Anzahl von Ereignissen erreicht ist und eine bestimmte Zeit seit Start der Messung vergangen ist.

Das Ergebnis einer Messung M ist ein Leistungswert λ, der die Leistung eines Programms gemäß des gewählten Leistungskriteriums erfasst. Der unmittelbare Messwert, der aus einer Messung M bzw. aus den Aufrufen von $startMeasurement()$ und $stopMeasurement()$ resultiert, ist ein Zeitwert t. Der für den Auto-Tuner relevante Leistungswert λ_M, der Puls, berechnet sich gemäß Definition 13 durch

$$\lambda_M = \frac{n}{t}$$

wobei n die Anzahl der Ereignisse ist, die im Zeitraum t der Messung M gezählt wurden.

5.3.3 Optimierungsschritt

Im Optimierungsschritt berechnet das Optimierungsverfahren eine neue Parameterkonfiguration. Nach Abschnitt 2.4.3 lässt sich der Optimierungsschritt als Funktion

$$\tau : S_{\mathscr{P}} \times \mathbb{R} \to S_{\mathscr{P}}$$

beschreiben, wobei $S_{\mathscr{P}}$ den Suchraum des Programms $\mathscr{P}$ darstellt. Die Berechnung der neuen Konfiguration $k_{neu} = \tau(k, \lambda(k))$ hängt also von der vorherigen Konfiguration k und dem dafür gemessenen Leistungswert $\lambda(k)$ ab.

Das Optimierungsverfahren führt eine Folge von Optimierungsschritten durch, um das Optimierungsproblem zu lösen. Das Optimierungsproblem lässt sich wie folgt vereinfacht formulieren:

- Die Tuningparameter $p_i \in \mathbb{N}, i \in \{1,...,n\}$ eines Programms spannen den Suchraum $S_{\mathscr{P}} \subset \mathbb{N}^n$ auf.
- Die Leistungsfunktion $\lambda : S_{\mathscr{P}} \to \mathbb{R}$ ordnet einer Parameterkonfiguration $k \in S_{\mathscr{P}}$ einen Leistungswert $\lambda(k) \in \mathbb{R}$ zu.
- Das Suchproblem besteht darin, eine Konfiguration $k^* \in S_{\mathscr{P}}$ zu finden, welche die Leistungsfunktion f maximiert[4]. Im Idealfall handelt es sich dabei um ein globales Optimum, welches die tatsächlich bestmögliche Konfiguration liefert.

Zur Lösung des Optimierungsproblems verwenden wir das Optimierungsverfahren nach Nelder und Mead. Dieses wird im folgenden Abschnitt skizziert.

5.3.3.1 Optimierungsverfahren nach Nelder und Mead

Das Optimierungsverfahren nach *Nelder und Mead* [73] ermittelt zunächst den Leistungswert der Startkonfiguration sowie sukzessive die Leistungswerte der im Suchraum „benachbarten“ Konfigurationen. Aus dieser Menge wird die beste und schlechteste Konfiguration bestimmt. Das Verfahren bricht ab, wenn die beste Konfiguration eine bestimmte Güte aufweist. Ansonsten wird die schlechteste Konfiguration durch eine neue ersetzt und das Verfahren beginnt von vorn. Die neue Konfiguration ergibt sich, indem die schlechteste Konfiguration am Mittelpunkt der konvexen Hülle der restlichen Konfigurationen (Simplex) reflektiert wird, d.h. es wird in derjenigen Richtung weitergesucht, in der die größte Leistungssteigerung erwartet wird.

Das Verfahren erweist sich als relativ robust, läuft allerdings Gefahr, gegen lokale Extrema zu konvergieren. Wir versuchen daher, mithilfe der Heuristiken aus Abschnitt 5.2.3 eine sinnvolle Startkonfiguration zu erreichen, welche sich in einem vielversprechenden Teil des Suchraums befindet. Wie Kapitel 7 zeigt, liegen die so bestimmten Startkonfigurationen der untersuchten Anwendungen in vielen Fällen bereits in der Nähe des empirisch ermittelten globalen Optimums. Wir werten dies als Indiz, dass die Vorkonfiguration die Gefahr „schlechter“ lokaler Optima zwar nicht ausschließen, aber dennoch reduzieren kann.

[4] An dieser Stelle nehmen wir an, dass sich die Programmleistung proportional zum Leistungswert verhält. Im Fall der pulsbasierten Optimierung ist dies gegeben, da die Leistung umso besser ist, je höher der gemessene Puls ist. Würde man die Laufzeit als Leistungskriterium wählen, so bestünde das Optimierungsproblem in der Minimierung der Leistungsfunktion.

5.3.4 Aktualisierung der Parameterwerte

Wurde im Optimierungsschritt eine neue Parameterkonfiguration berechnet, so müssen die neuen Werte den Parametern zugewiesen werden. Für Laufzeit-Tuning ergibt sich hierbei das Problem der verzögerten Parameterwirksamkeit. Da die Ausführung des Stromprogramms zwischen zwei Messungen nicht pausiert, könnten geänderte Parameterwerte sich auf Filter beziehen, die sich noch in der Ausführung befinden. Tuningparameter wie z.B. der Replikationsfaktor, die sich nicht auf einzelne Filter, sondern auf Filterkombinationen beziehen, können nicht wirksam werden, wenn diese Filter noch aktiv sind.

Um diesen Sachverhalt berücksichtigen zu können, werden alle aktiven Filter, die von geänderten Parameterwerten betroffen sind, markiert. Eine Markierung besagt, dass neue Parameterwerte vorliegen. Ein markierter Filter führt seine aktuelle Aktion zu Ende und wird in die Filter-Warteschlange zurückgelegt. Nun werden die neuen Parameterwerte übernommen und die Markierung entfernt. Wenn kein Filter mehr markiert ist, ist die neue Parameterkonfiguration für alle Filter wirksam.

5.3.5 Erholungsphasen

Unmittelbar nachdem die neu berechnete Parameterkonfiguration wirksam wird, befinden sich noch Stromelemente in den Puffern zwischen den Filtern. Diese stammen aus der alten Messphase und lösen Ereignisse aus, die nicht in den gemessenen Puls der neuen Messphase einfließen dürfen, da der Puls sonst verfälscht würde.

Es muss also sichergestellt werden, dass zwei aufeinanderfolgende Messungen sauber voneinander getrennt werden. Dies ist Voraussetzung dafür, dass Messungen vergleichbar und somit für Optimierungsalgorithmen verwertbar sind. Um zwei Messungen voneinander zu trennen, findet eine Erholungsphase zwischen der Aktualisierung der Parameterwerte und dem Starten einer neuen Messung statt.

Die Erholungsphase stellt sicher, dass die neue Messung erst dann gestartet wird, wenn der Stromgraph von solchen Stromelementen, welche einen Messwert verfälschen können, bereinigt ist. Anhand der Kontextinformationen über die Arbeitslast eines Filters lässt sich die Summe s aller Stromelemente berechnen, die noch zwischen Filtern gepuffert sind. Jedes Stromelement, das den Stromgraphen vollständig durchquert hat, löst ein Ereignis aus, so dass nach s ausgelösten Messereignissen der Stromgraph als bereinigt betrachtet wird[5] und die zweite Phase abgeschlossen ist. Nun kann die neue Messung beginnen.

[5] In einigen Stromgraphen ist es nicht ausgeschlossen, dass manche Elemente andere „überholen“. Dieser Fall tritt jedoch nur selten ein und „verfälscht“ die nachfolgende Messung daher nur in geringem Ausmaß.

Eine Erholungsphase bzw. Stromgraphbereinigung dauert umso länger, je mehr Stromelemente zu diesem Zeitpunkt in den Kanälen gepuffert sind. Unserem Ansatz liegt jedoch die grundsätzliche Annahme der Stromverarbeitung zugrunde, dass Datenströme hinreichend lang sind. Insofern gehen wir davon aus, dass die Anzahl der gepufferten Elemente in einem zu bereinigenden Stromgraphen einen hinreichend kleinen Anteil des Gesamtstroms darstellt. Die Dauer der Erholungsphase ist unter dieser Annahme also nicht absolut, sondern im Verhältnis zur gesamten Stromlänge bzw. Ausführungsdauer des Stromprogramms zu betrachten.

In jedem Fall sei jedoch angemerkt, dass weder die Parameteraktualisierung noch die Erholungsphase die Ausführung des Stromprogramms verzögern. Bei diesen „Wartephasen“ handelt es sich lediglich um Zeiten, die aus Sicht des Auto-Tuners zwischen Beenden einer Messung und Starten einer neuen vergehen. Die Wartephasen verlangsamen also nicht die Ausführung des Programms, sondern lediglich die Tuninggeschwindigkeit. Im Allgemeinen ist die Qualität der Optimierung dabei höher, da die Messungen aufgrund der sauberen Trennung besser vergleichbar sind. Ohne diese Trennung wäre die Tuninggeschwindigkeit möglicherweise etwas höher, allerdings wäre die Optimierungsqualität aufgrund von Messverfälschungen und Fehlinterpretationen durch den Auto-Tuner beeinträchtigt.

5.3.6 Tuningzyklus

In den vorigen Abschnitten wurden Konzepte zur Leistungsbestimmung und -optimierung vorgestellt und auf Stromprogramme abgestimmt. Der Gesamtablauf des Laufzeit-Tunings ergibt sich aus der Synthese dieser Konzepte. Abbildung 5.7 zeigt den daraus resultierenden erweiterten Tuningzyklus zur Optimierung von Stromprogrammen im Produktivbetrieb.

Der Ablauf des Laufzeit-Tunings ergibt sich unmittelbar aus dem Tuningzyklus. Der Tuningzyklus wird solange durchlaufen, bis der Optimierungsalgorithmus keine wesentlichen Verbesserungen mehr erreichen kann.

Ausgehend von einem optimierbaren, gemäß Abschnitt 5.2 vorkonfigurierten Stromprogramm $\mathscr{P}$, lässt sich der Ablauf des Laufzeit-Tunings für $\mathscr{P}$ wie folgt zusammenfassen:

1. **Messung:** Zur Leistungsbestimmung wird der Puls für einen bestimmten Zeitraum gemessen (Abschnitt 5.3.2).

2. **Optimierungsschritt:** Basierend auf dem gemessenen Leistungswert $\lambda(k)$ für die Konfiguration k berechnet das Optimierungsverfahren eine neue Parameterkonfiguration k_{neu} (Abschnitt 5.3.3).

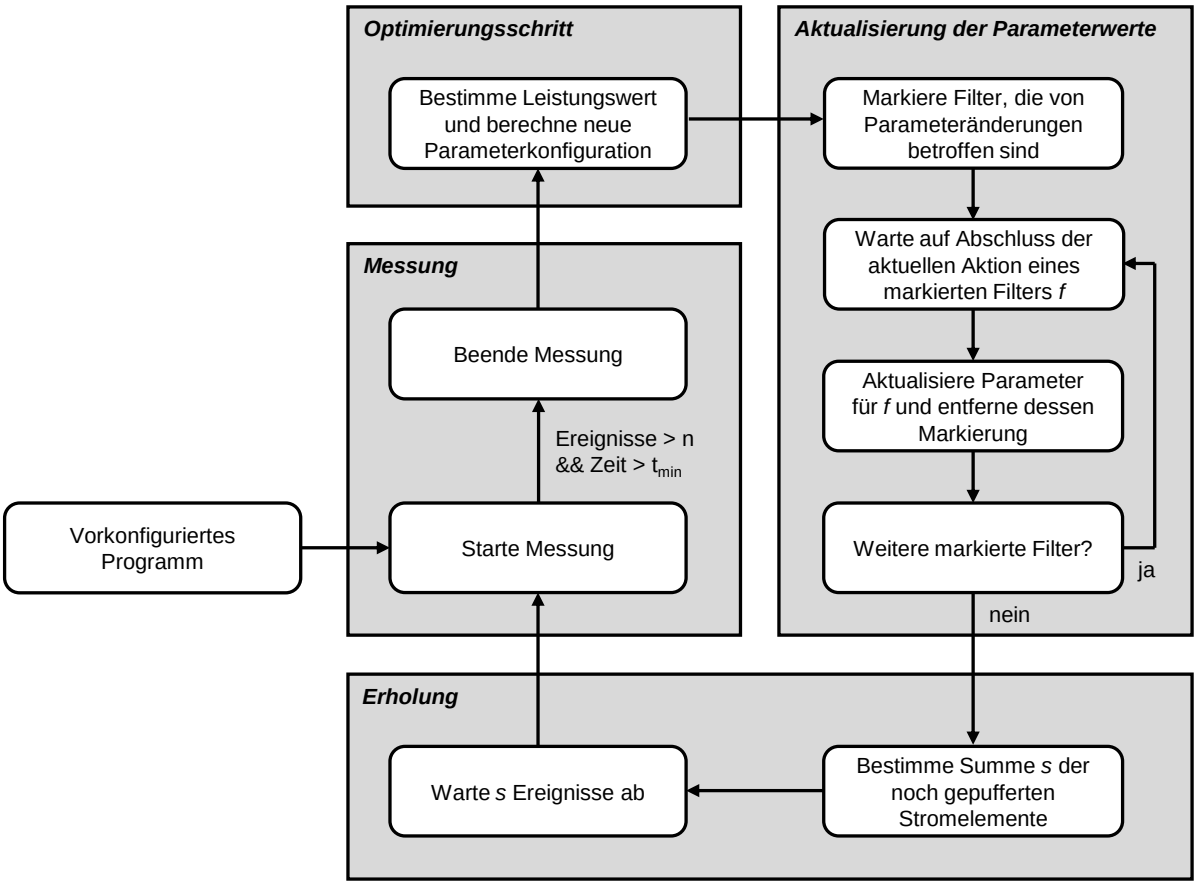


Abbildung 5.7: Tuningzyklus für Stromprogramme

3. **Aktualisierung der Parameterwerte:** Die Filter, die von der neuen Konfiguration betroffen sind, aktualisieren ihre Parameterwerte (Abschnitt 5.3.4).

4. **Erholung:** Der Stromgraph wird bereinigt. Stromelemente aus der abgeschlossenen Messung müssen den Stromgraph passieren, bevor eine neue Messung gestartet wird (Abschnitt 5.3.5).

5.4 Zusammenfassung

In diesem Kapitel wurden Konzepte zur Ausführung und Optimierung von Stromprogrammen vorgestellt und diskutiert. Im Mittelpunkt stand dabei die Überführung von Stromprogrammen in eine optimierbare Form sowie deren Optimierung mithilfe von Laufzeit-Tuning.

Hierfür wird ein Stromprogramm automatisch mit Tuningparametern, Kontextinformationen und Messpunkten instrumentiert. Zudem werden Tuningparameter nach ihrem Zweck klassifiziert. Die Kombination aus klassifizierten Tuningparametern und Kontextinformationen erlaubt es nun, Tuningparametern sinnvolle Standardwerte und Wertebereiche zuzuordnen; das Programm ist somit vorkonfiguriert.

Mithilfe der Messpunkte ist es möglich, in Form eines Pulses die Gesamtleistung von Stromprogrammen zu bestimmen. Der Puls dient einem Laufzeit-Tuner zur Optimierung der Parameter eines laufenden Stromprogramms. Somit wird es möglich, Stromprogramme im Produktivbetrieb, d.h. ohne vorherige Testläufe, zu optimieren.

Kapitel 6

Implementierung

Dieses Kapitel skizziert die technische Umsetzung der in den Kapiteln 4 und 5 entwickelten Konzepte. Unser Prototyp *XJava* ist eine Erweiterung der Sprache Java. Hierfür wurden der Präprozessor `xjavac` sowie das Laufzeitsystem `xjavart` entworfen und implementiert.

Abbildung 6.1 gibt einen Überblick über das Gesamtsystem XJava. Des Weiteren veranschaulicht die Abbildung die Schritte und Zwischenergebnisse bei der Überführung eines objektorientierten Stromprogramms in optimierbaren, ausführbaren Code. Im Folgenden beschreiben wir Aufbau und Implementierung von `xjavac` und `xjavart`.

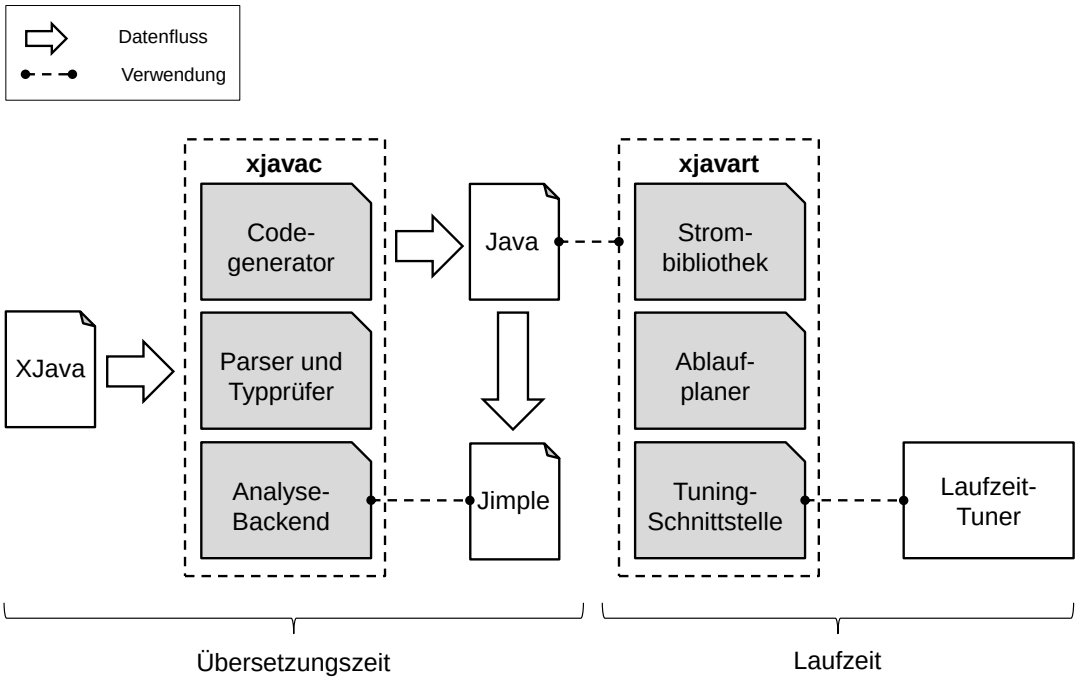


Abbildung 6.1: Implementierung von XJava

6.1 Implementierung von `xjavac`

Der Präprozessor `xjavac` erfüllt die Funktion eines erweiterten Java-Übersetzers. `xjavac` führt für ein objektorientiertes Stromprogramm, also XJava-Quelltext, eine syntaktische und semantische Analyse durch, erzeugt einen abstrakten Syntaxbaum und generiert für jeden Syntaxbaumknoten Java-Code. Bei der Implementierung von `xjavac` handelt es sich um eine Erweiterung des Übersetzerrahmenwerks Polyglot [76, 4].

6.1.1 Polyglot

Polyglot ist ein in Java geschriebenes Frontend für Java. Die Sprachdefinition basiert auf dem Lexer JFlex [3] sowie einer modifizierten Variante des LALR-Parsergenerators CUP [2]. Polyglot unterstützt die Sprachdefinition von Java 1.4, welche beispielsweise keine generischen Typen beinhaltet. Die Implementierung von `xjavac` verwendet eine Erweiterung von Polyglot 1.3.5, welche auch Java 5 unterstützt.

Die zentralen Komponenten der Polyglot-Architektur lassen sich folgenden Paketen zuordnen:

- `main`: Starten des Übersetzers und Auswerten von Kommandozeilenoptionen
- `frontend`: Frontend zur Steuerung der Arbeitsschritte des Übersetzers
- `parse`: Parser
- `ast`: Aufbau von Syntaxbäumen
- `visit`: Besucherklassen, welche die Knoten von abstrakten Syntaxbäumen durchlaufen
- `types`: Typsystem
- `ext`: Spracherweiterungen (hier existieren wiederum entsprechende Unterpakete `parse`, `ast`, `visit`, `types`)

Entsprechend dieser Architektur wurde Polyglot um die in Kapitel 4 dargestellte Sprachdefinition zur objektorientierten Stromprogrammierung erweitert.

Sprachkonstrukt	AST-Klasse (XJava)	AST-Oberklasse (Java)
Filterdeklaration	FilterDecl_c	JL5MethodDecl_c
Work-Block	Work_c	While_c
(datenparalleler) Filteraufruf	FilterOrMethodCall_c	JL5MethodCall_c
Fließbandausdruck	PipelineExpr_c	Expr_c
Fließbandblock	PipelineBlock_c	Block_c
Aufgabenparalleler Ausdruck	ConcurrentExpr_c	Expr_c
Aufgabenparalleler Block	ConcurrentBlock_c	Block_c
Alternativenausdruck	AlternativeExpr_c	Expr_c
Alternativenblock	AlternativeBlock_c	Block_c
Push-Anweisung	Push_c	Stmt_c
Splitjoin-Operator	SplitJoinMode	–

Tabelle 6.1: Abbildung der XJava-Sprachkonstrukte auf AST-Klassen

6.1.2 AST-Klassen

Das Paket `polyglot.ext.xjava.ast` enthält AST-Klassen, die zum Aufbau von XJava-Syntaxbäumen benötigt werden. Die Klassenbezeichner sind im Wesentlichen dieselben wie die in Kapitel 4 verwendeten Namen der Nichtterminale.

Tabelle 6.1 zeigt die wichtigsten Sprachkonstrukte unserer Spracherweiterung, deren entsprechende AST-Klassen sowie deren Oberklassen aus der Polyglot-Rahmenarchitektur. Diese AST-Klassen überschreiben Methoden zur semantischen Analyse eines Syntaxbaumknotens oder Teilbaums sowie zur Codegenerierung. Darüber hinaus definieren sie neue Methoden, mit denen Struktur- und Kontextinformationen für die Syntaxbaumknoten angelegt und abgefragt werden können.

6.1.3 Codegenerierung

Die Codegenerierung für die Syntaxbaumknoten ist in den jeweiligen AST-Klassen definiert. Die AST-Klassen implementieren bzw. überschreiben hierfür Methoden `translate()` und `prettyPrint()`.

Filterdeklarationen werden zu Klassen, Filteraufrufe zu Instanzen dieser Klassen übersetzt. Im generierten Code ist eine Strombibliothek (vgl. Abschnitt 6.2.1) aus dem Laufzeitsystem `xjavart` eingebunden. Diese Strombibliothek umfasst Klassen und Datenstrukturen zur Konstruktion und Verwaltung von Stromgraphen. Filterklassen enthalten neben dem Anwendungscode somit weitere Methoden, mit denen die identifizierten Tuningparameter verändert sowie jene Struktur- und Kontextinformationen abgefragt werden können, welche zuvor durch

die semantische Analyse gewonnen wurden. Auf diese Weise werden die Informationen während der Ausführung nutzbar gemacht und können vom Laufzeitsystem `xjavart` bzw. einem Laufzeit-Tuner verwendet werden.

Listing 6.1: Auftraggeber-Auftragnehmer-Muster in XJava

```
1  class Item { ... }
2
3  class C {
4    void => Item m() {  /* push elements */ }
5
6    Item => void w() {
7      work (Item i) { /* do something */  }
8    }
9  }
10
11 public class D {
12   public static void main(String[] args) {
13     C c = new C();
14     c.m() =>? c.w()+;
15   }
16 }
```

Zur Veranschaulichung zeigt Listing 6.1 das XJava-Codegerüst eines Auftraggeber-Auftragnehmer-Musters; der daraus generierte Java-Code ist in Listing 6.2 vereinfacht dargestellt. Sämtliche Bezeichner mit dem Präfix `_xjava_` sind generierte Variablen, die nicht Teil des ursprünglichen Codes sind; bei den eingebundenen Klassen aus dem Paket `xjava.runtime` handelt es sich um Funktionalität der Strombibliothek. Die im Beispiel verwendeten XJava-Sprachkonstrukte werden wie folgt übersetzt:

- Die Filterdeklarationen werden in entsprechende Umwicklerklassen überführt (Zeilen 5 und 13). Diese Klassen sind Unterklassen von `NonPeriodicFilter` bzw. `PeriodicFilter` aus der Strombibliothek und implementieren Methoden, die den Anwendungscode der Filter enthalten (Zeilen 6 und 14-16). Zusätzlich erben die Umwicklerklassen Methoden zur Verwaltung von Tuningparametern und Kontextinformationen.

- Instanzen dieser Filter werden durch das Aufrufen der Methoden `_xjava_m_NEW` bzw. `_xjava_w_NEW` erzeugt (Zeile 31 bzw. 39).

- Die Implementierung des Auftraggeber-Auftragnehmer-Musters ist im XJava-Code mithilfe einer Fließbandanweisung umgesetzt; diese wird auf ein `Pipeline` Objekt abgebildet (Zeile 28). Diesem `Pipeline` Objekt werden die Filterinstanzen für `m` und `w` übergeben (Zeilen 36 und 43).

Listing 6.2: Generierter Code für das Auftraggeber-Auftragnehmer-Muster

```
1  class Item { ... }
2
3  class C {
4    /* [XJAVA] GENERATED CODE FOR void => Item m() { ... } */
5    class _xjava_m_WrapperClass extends xjava.runtime.NonPeriodicFilter {
6      public void run() { ... }
7      ...
8    }
9    xjava.runtime.AbstractFilter _xjava_m_NEW() { ... }
10   /* [XJAVA] END OF GENERATED CODE FOR void => Item m() { ... } */
11
12   /* [XJAVA] GENERATED CODE FOR Item => void w() { ... } */
13   class _xjava_w_WrapperClass extends xjava.runtime.PeriodicFilter {
14     public void beforeWork() { ... }
15     public int work(int actionCount) { ... }
16     public void afterWork() { ... }
17     ...
18   }
19   xjava.runtime.AbstractFilter _xjava_w_NEW() { ... }
20   /* [XJAVA] END OF GENERATED CODE FOR Item => void w() { ... } */
21 }
22
23 public class D {
24   public static void main(String[] args) {
25     C c = new C();
26
27     /* [XJAVA] GENERATED CODE FOR c.m() => c.w()+ */
28     xjava.runtime.Pipeline _xjava_pipe = new xjava.runtime.Pipeline();
29
30       /* [XJAVA] GENERATED CODE FOR c.m() */
31       xjava.runtime.AbstractFilter _xjava_m = c._xjava_m_NEW()
32           .setSplitMode(xjava.runtime.SplitJoinMode.FIRSTCOME);
33
34       /* [XJAVA] END OF GENERATED CODE FOR c.m() */
35
36     _xjava_pipe.add(_xjava_m);
37
38       /* [XJAVA] GENERATED CODE FOR c.w()+ */
39       xjava.runtime.AbstractFilter _xjava_w = c._xjava_w_NEW()
40           .dynamicallyReplicable(true);
41       /* [XJAVA] END OF GENERATED CODE FOR c.w()+ */
42
43     _xjava_pipe.add(_xjava_w);
44     _xjava_pipe.run();
45     /* [XJAVA] END OF GENERATED CODE FOR c.m() =>? c.w()+ */
46   }
47 }
```

- Die Aufteilungsstrategie zwischen Auftraggeber und Auftragnehmern ist durch den Firstcome-Operator `?` festgelegt; im generierten Java-Code wird hierfür die Methode `setSplitMode` auf dem Auftraggeberobjekt aufgerufen (Zeile 32).
- Die Methode `setDynamicallyReplicable` kennzeichnet die Filterinstanz `_xjava_w` als replizierbar (Zeile 40) und fügt dieser damit einen Tuningparameter für den Replikationsfaktor *RF* hinzu.

6.1.4 Analyse-Backend

`xjavac` enthält neben den bereits genannten Paketen ein weiteres Paket `polyglot.ext.xjava.analysis`, welches Implementierungen der Zustands- und Konfliktanalyse aus Kapitel 4 enthält. Beide Implementierungen basieren auf der Rahmenarchitektur Soot [5].

6.1.4.1 Soot

Soot ist eine Rahmenarchitektur zur Codeanalyse und -transformation für Java-Programme. Die Rahmenarchitektur beinhaltet verschiedene Analysen, z.B. Zeigeranalysen [63], und bietet darüber hinaus die Möglichkeit, Analysen und Transformationen zu erweitern oder neu zu entwickeln.

Der Aufbau von Soot orientiert sich an einem sogenannten Phasenmodell. Eine Phase kapselt eine bestimmte Analyse und kann Abhängigkeiten zu anderen Phasen bzw. Analysen definieren. So ist beispielsweise die Konstruktion eines Aufrufgraphen eine Phase, welche Vorbedingung für alle Phasen ist, die interprozedurale Analysen durchführen. Für unsere Arbeit sind die Phasen der Aufrufgraphkonstruktion sowie der Zeigeranalyse von besonderer Bedeutung.

6.1.4.2 Integration von Soot in `xjavac`

Ein in Java überführtes XJava-Programm wird mithilfe von Soot in die Zwischenform Jimple überführt. Bei Jimple handelt es sich um eine typisierte anweisungsbasierte Darstellung, auf deren Grundlage Datenflussanalysen durchgeführt werden können. Dabei werden die Elemente eines Java-Programms auf Soot-interne Datenstrukturen bzw. Klassen abgebildet. Tabelle 6.2 zeigt diejenigen Datenstrukturen, die bei der Analyse eines objektorientierten Stromprogramms von besonderer Bedeutung sind. Um Filter bzw. Filterklassen gesondert berücksichtigen zu können, wurde hierfür eine separate Klasse `FilterClass` implementiert.

Analysen selbst werden in Klassen gekapselt, welche von der abstrakten Soot-Klasse `soot.SceneTransformer` erben und deren Methode `internalTransform` implementieren.

Klasse	Bedeutung
SootClass	Klasse
FilterClass	Filterdeklaration (XJava) bzw. Filterklasse (Java)
SootMethod	Methode
SootField	Attribut
Body	Methodenrumpf
UnitGraph	intraprozeduraler Kontrollflussgraph einer Methode
Stmt	Anweisung
Value	abstrakter Wert (z.B. lokale Variable, Konstante oder Ausdruck)

Tabelle 6.2: Datenstrukturen in Soot

6.1.4.3 Zustandsanalyse

Die Klasse StateAnalysis kapselt die Zustandsanalyse. Hierfür werden zunächst alle Filter eines Programms, also alle FilterClass Objekte, ermittelt. Die Filter werden von einer Methode isStateless(FilterClass), welche das in Kapitel 4 beschriebene Verfahren implementiert, auf Zustände überprüft.

6.1.4.4 Konfliktanalyse

Gemäß Kapitel 4 wird vor der Konfliktanalyse eine Locksetanalyse durchgeführt, um die Sperrmengen jeder Anweisung zu bestimmen. Die Konfliktanalyse kann somit korrekt synchronisierte Zugriffe auf gemeinsame Variablen ignorieren, so dass hierfür keine (falschen) Warnungen ausgegeben werden.

Die Locksetanalyse ist in der Klasse LocksetAnalysis implementiert. Diese besucht mithilfe des Aufrufgraphen alle Methoden des Programms und ermittelt für jede Anweisung, d.h. jedes Stmt Objekt, die Menge der gesetzten Sperren. Diese Informationen stehen der anschließenden Konfliktanalyse zur Verfügung.

Die Konfliktanalyse findet sich in der Klasse ConflictAnalysis. Die Methode getFieldAccesses(FilterClass) ermittelt zunächst für jeden Filter die Menge aller Lese- und Schreibzugriffe auf SootField Objekte. Anschließend prüft die Methode checkFilterConflicts(), welche Zugriffspaare zu Wettlauffehlern führen könnten.

6.2 Implementierung von xjavart

Das Laufzeitsystem xjavart besteht aus einer Strombibliothek, einem Ablaufplaner und einer Schnittstelle für Laufzeit-Tuning. Diese Komponenten sind für die Ausführung und Op-

timierung eines Stromprogramms zuständig. Deren Verwendung erfolgt automatisch, d.h. der Anwendungsprogrammierer wird mit diesem Laufzeitsystem nicht konfrontiert.

6.2.1 Strombibliothek

Die Strombibliothek (`xjava.runtime`) umfasst Hilfsklassen und Datenstrukturen zur Erzeugung und Verwaltung objektorientierter Stromgraphen. Der Übersetzer `xjavac` bindet diese Klassen in den generierten Code ein.

6.2.1.1 Filter und Stromgraphkomponenten

Die Strombibliothek enthält unter anderem Klassen zur Verwaltung von Filtern und Stromgraphkomponenten; Abbildung 6.2 zeigt die wichtigsten Klassen in Form eines Klassendiagramms. Die Schnittstelle `IFilter` repräsentiert einen Filter oder eine Stromgraphkomponente, die aus mehreren Filter zusammengesetzt ist.

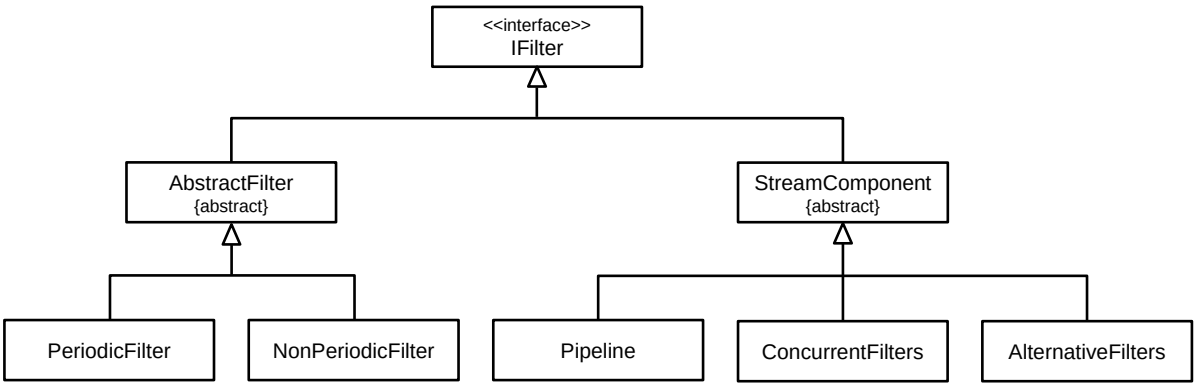


Abbildung 6.2: Klassen zur Verwaltung von Filtern und Stromgraphkomponenten

Die abstrakte Klasse `AbstractFilter` stellt eine Basisimplementierung der Schnittstelle `IFilter` dar. `AbstractFilter` definiert Methoden zur Abfrage und Modifikation von Eigenschaften eines allgemeinen Filters. Diese Eigenschaften umfassen bereits einen Großteil der relevanten Verwaltungs- und Kontextinformationen, die für die Ablaufplanung und Optimierung von Bedeutung sind. Beispiele sind die Parallelitätsebene, auf der sich der Filter befindet, die Replizierbarkeit oder die aktuelle Arbeitslast. `AbstractFilter` besitzt zwei Unterklassen `PeriodicFilter` und `NonPeriodicFilter`, welche die Funktionalität für periodische bzw. nichtperiodische Filter verfeinern. Diese beiden Klassen dienen als direkte Oberklassen der von `xjavac` generierten Filterklassen (vgl. Listing 6.2).

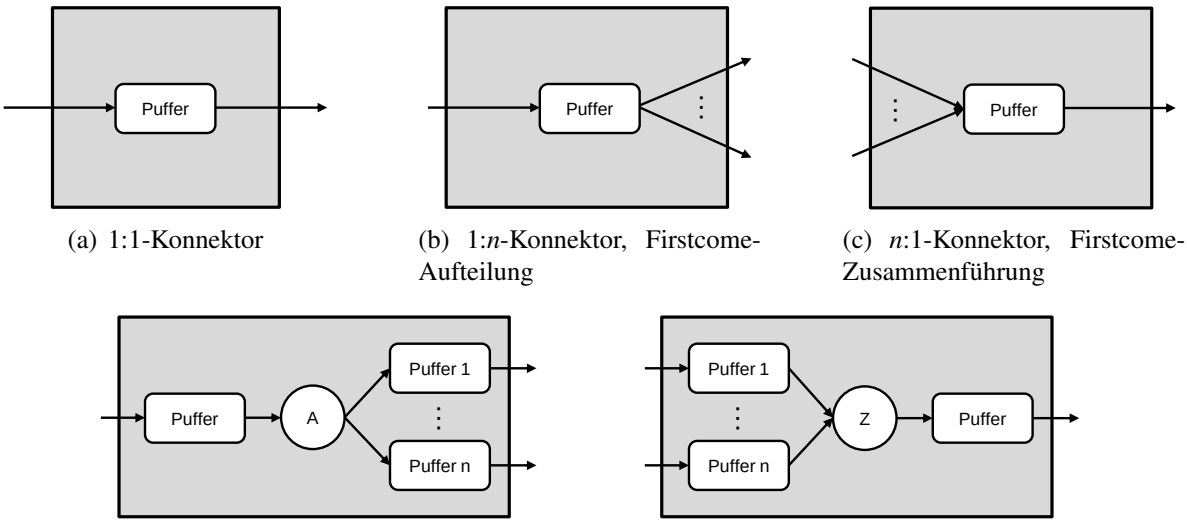
`StreamComponent` ist eine abstrakte Klasse, welche eine allgemeine Stromgraphkomponente beschreibt, die sich aus mehreren Filtern zusammensetzt. Die Unterklassen `Pipeline`,

`ConcurrentFilters` und `AlternativeFilters` kapseln konkrete Stromgraphkomponenten für Fließband- und Aufgaben- bzw. Datenparallelität sowie zur Zusammenfassung alternativer Filter.

6.2.1.2 Konnektoren

Um Filter zu Stromgraphen zu kombinieren, müssen Datenstrukturen bereitstehen, die den Austausch von Stromelementen ermöglichen. Hierfür steht die Klasse `Connector` bereit, die im Kern aus einem oder mehreren fadensicheren Puffern besteht. `Connector` Instanzen werden von `xjavac` erzeugt, um Filter nach dem Produzenten-Konsumenten-Prinzip zu verbinden.

Im einfachsten Fall verbindet ein Konnektor zwei Filterinstanzen und entspricht somit einem 1:1-Kanal. Ein Konnektor kann auch als Aufteiler (1:*n*-Kanal) oder Zusammenführer (*n*:1-Kanal) fungieren. Die Klasse `Connector` bietet hierfür entsprechende Methoden, um die Anzahl der Eingangs- und Ausgangsströme sowie die Aufteilungs- und Zusammenführungsstrategie festzulegen. Der vom Übersetzer `xjavac` generierte Code beinhaltet entsprechende Anweisungen, um Konnektoren zu erzeugen, zu konfigurieren und Filterinstanzen zuzuordnen.



(a) 1:1-Konnektor

(b) 1:*n*-Konnektor, Firstcome-Aufteilung

(c) *n*:1-Konnektor, Firstcome-Zusammenführung

(d) 1:*n*-Konnektor, Roundrobin- oder Broadcast-Aufteilung (A = Aufteilungskomponente)

(e) *n*:1-Konnektor, Roundrobin-Zusammenführung (Z = Zusammenführungskomponente)

Abbildung 6.3: Aufbau der Konnektoren

Abbildung 6.3 skizziert die Implementierung der Konnektoren. 1:1-Konnektoren sowie First-come-Aufteiler und -Zusammenführer (vgl. Abbildungen 6.3(a), 6.3(b) und 6.3(c)) enthalten einen FIFO-Puffer zur Übergabe von Stromelementen. Roundrobin- und Broadcast-Aufteiler (vgl. Abbildung 6.3(d)) bestehen aus einen FIFO-Puffer, der seinen Inhalt auf n FIFO-Puffer entsprechend der gewählten Strategie verteilt und somit aus einem Strom n Ströme erzeugt. Analog dazu besitzen Roundrobin-Zusammenführer (vgl. Abbildung 6.3(e)) n FIFO-Puffer, welche n Ströme empfangen, sowie einen weiteren FIFO-Puffer, der die zusammengeführten Stromelemente enthält.

6.2.2 Ablaufplaner

Der Übersetzer `xjavac` bildet Filterdeklarationen auf Klassen und Filteraufrufe auf Instanzen dieser Klassen ab. Die Filterinstanzen werden einem Ablaufplaner übergeben, der Bestandteil des Laufzeitsystems `xjavart` ist.

Der Ablaufplaner befindet sich im Paket `xjava.runtime.scheduling`; die Klasse `Scheduler` definiert dessen Grundfunktionalität. Der Ablaufplaner besitzt eine Warteschlange, in welcher Filterinstanzen abgelegt werden, die zur Ausführung bereit sind. Die Ausführungsfäden, implementiert durch die Klasse `Executor`, holen sich nach und nach eine Filterinstanz, führen eine bestimmte Anzahl von Aktionen aus, und legen die Filterinstanz zurück in die Warteschlange.

6.2.3 Schnittstelle zum Laufzeit-Tuner

Der Laufzeit-Tuner [52] stellt Systemaufrufe bereit, welche die Markierung eines Tuningparameters sowie das Starten und Beenden von Messungen erlauben. Zur Anbindung des Laufzeit-Tuners an das XJava Laufzeitsystem müssen diese Systemaufrufe zunächst innerhalb der Sprache Java nutzbar gemacht werden. Dies wird mithilfe des Java Native Interface (JNI) [64] erreicht. JNI stellt eine Programmierschnittstelle dar, welche das Aufrufen von C/C++-Methoden in Java erlaubt. Die Klassen und Datenstrukturen, die Laufzeit-Tuning für XJava-Programme ermöglichen, sind dem Paket `xjava.runtime.tuning.online` zugeordnet.

6.2.3.1 Tuningparameter

Zur Kapselung eines Tuningparameters steht die Klasse `TunableInteger` zur Verfügung. Da gemäß der Definition aus Kapitel 2 die Wertemenge eines Tuningparameters diskret ist, lassen sich sämtliche Parameterarten mithilfe einer Klasse `TunableInteger` beschreiben, deren

Aufbau in Kapitel 5 skizziert wurde. Über den Konstruktor der Klasse `TunableInteger` werden der Standardwert und Wertebereich eines Parameters festgelegt. Dieser Tuningparameter wird mittels JNI durch einen entsprechenden Systemaufruf dem Laufzeit-Tuner übergeben. Der Laufzeit-Tuner kann den Wert dieses Parameters verändern; der aktuelle Wert lässt sich durch die Methode `getValue()` in der Klasse `TunableInteger` abfragen.

6.2.3.2 Starten und Beenden von Messungen

In der Klasse `AutoTuner` stehen native Methoden `startMeasure()` und `stopMeasure()` bereit, mit denen Messungen gestartet und beendet werden können. Diese lösen den entsprechenden Systemaufruf des Laufzeit-Tuners aus. Es ist zu beachten, dass der Laufzeit-Tuner unmittelbar nach dem Beenden und Auswerten einer Messung eine neue Parameterkonfiguration berechnet. Die veränderten Parameterwerte liegen somit in entsprechenden C-Datenstrukturen vor; die entsprechenden `TunableInteger` Objekte enthalten Kopien dieser Werte.

6.2.3.3 Tuningzyklus

Der in Kapitel 5 konzipierte Tuningzyklus für Laufzeit-Tuning ist in der Klasse `TunerThread` implementiert, welche einen separaten Ausführungsfaden darstellt. Der Hauptgrund liegt darin, dass der verwendete Auto-Tuner Messabschnitte einem bestimmten Faden zuordnet. Somit muss eine Messung von demselben Faden beendet werden, der sie gestartet hat. Darüber hinaus würde die dezentrale Verwaltung des Tuningzyklus einen Mehraufwand für Synchronisation verursachen, der sich bei zentraler Verwaltung stets desselben Fadens vermeiden lässt.

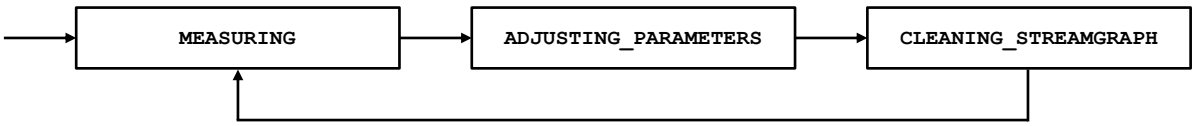


Abbildung 6.4: Zustände des Tuningfadens

Die Zustände und Zustandsübergänge des Tuningfadens sind in Abbildung 6.4 dargestellt. Der Anfangszustand `MEASURING` tritt ein, sobald die erste Messung gestartet wird. Dem Tuningzyklus entsprechend befindet sich der Tuningfaden fortan in einem der folgenden Zustände:

- `MEASURING`: Das laufende Programm wird vermessen.
- `ADJUSTING_PARAMETERS`: Der Optimierungsalgorithmus des Auto-Tuners berechnet eine neue Parameterkonfiguration. Die neuen Parameterwerte werden aktualisiert.
- `CLEANING_STEAMGRAPH`: Der Stromgraph wird bereinigt, um die abgeschlossene und die nächste Messung sauber voneinander zu trennen (Erholungsphase).

6.3 Zusammenfassung

Die in den Kapiteln 4 und 5 entwickelten Lösungskonzepte wurden in Form von XJava, einer strombasierten Erweiterung der Sprache Java, umgesetzt. Hierfür wurden der Übersetzer `xjavac` sowie das Laufzeitsystem `xjavart` prototypisch implementiert. Dieses Kapitel beschrieb die wichtigsten Komponenten von `xjavac` und `xjavart`.

`xjavac` generiert für ein XJava-Programm Java-Code und führt außer syntaktischen und semantischen Analysen auch Zustands- und Konfliktanalysen durch. Der generierte Code wird durch `xjavart` ausgeführt. `xjavart` besteht aus einer Strombibliothek, einem Ablaufplaner mit speziellen Ausführungsfäden sowie einer Komponente zur Anbindung eines Laufzeit-Tuners.

Im folgenden Kapitel diskutieren wir die Ergebnisse, die anhand von Fallstudien und unter Verwendung der beschriebenen Implementierung gewonnen wurden.

Kapitel 7

Evaluation

Dieses Kapitel evaluiert und diskutiert die Lösungskonzepte der vorliegenden Arbeit anhand unterschiedlicher Anwendungen. Es wird gezeigt, dass die Ziele und Thesen der Arbeit erfüllt werden.

7.1 Untersuchte Anwendungen

Zunächst beschreiben wir die Anwendungen und Programme, die für die Evaluation als Fallstudien dienen. Hierbei wurde sowohl auf bestehenden Code zurückgegriffen als auch neue Programme geschrieben. Es wurde darauf Wert gelegt, Vertreter aus möglichst verschiedenen Anwendungsbereichen zu wählen. Die Anwendungen unterscheiden sich zudem bezüglich Codeumfang, Struktur und Granularität der Parallelität. Alle Anwendungen liegen in sequenziellen und fadenbasierten Versionen vor; zusätzlich haben wir stromorientierte Versionen implementiert [105].

7.1.1 Desktopsuche (DS)

DS ist eine Anwendung zum Indizieren und Auffinden von Textdateien. Der Indexgenerator erzeugt hierfür eine Menge von Schlüssel-Wert-Paaren. Ein Paar besteht aus einem Suchbegriff und einer Liste derjenigen Textdateien, die den Suchbegriff enthalten.

Die Anwendung existiert sowohl in sequenzieller als auch in fadenbasierter, optimierter Form [70]. Zur Evaluation wurden diese Implementierungen um eine stromorientierte Version ergänzt. Abbildung 7.1 zeigt den Aufbau der stromorientierten Implementierung.

Das Stromprogramm besteht aus einem dreistufigen Fließband. Die erste Stufe liest ein Verzeichnis und generiert daraus einen Strom von Dateien. Die zweite Stufe berechnet für jede Datei eine Wortmenge, d.h. sie transformiert den Strom von Dateien in einen Strom von

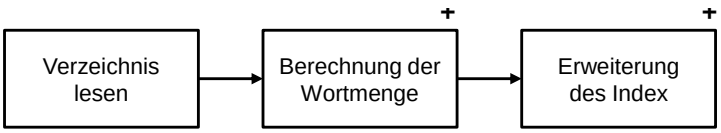


Abbildung 7.1: Aufbau der Desktopsuche

Wortmengen. Die dritte Stufe aktualisiert sukzessive den Index entsprechend der empfangenen Wortmengen. Die zweite und dritte Stufe sind zustandslos und somit replizierbar. Die Aufteilung und Zusammenführung der Ströme erfolgt nach dem Firstcome-Prinzip, da die Reihenfolge, in der die Dateien indiziert werden, keine Rolle spielt.

Diese Fließbandstruktur ist vergleichbar mit dem Tiled-MapReduce Muster, einer für Multikernsysteme optimierten Variante von MapReduce (vgl. Kapitel 3, Abschnitt 3.2.4). Statt der gesamten Datenmenge werden hierbei Partitionen, in diesem Fall Dateien, iterativ parallel verarbeitet. Die stromorientierte Verarbeitung unterliegt insofern demselben Prinzip, da nicht der gesamte Strom auf einmal, sondern nach und nach nur „Teilströme“ (d.h. eine bestimmte Anzahl von Stromelementen) parallel bearbeitet werden.

7.1.2 Electric

Electric [1] ist eine quelloffene Anwendung für den Entwurf und die Analyse von VLSI-Schaltkreisen. Mit etwa 400.000 Codezeilen handelt es sich hierbei um die größte und komplexeste Anwendung, die in der vorliegenden Arbeit untersucht wird.

Die Funktionalität von Electric ist äußerst umfangreich und in Teilen bereits parallelisiert und optimiert. Eine zentrale Funktion stellt die Planung der Bausteinplatzierung (engl. *placement*) dar. Deren Ziel ist es, für eine abstrakte Schaltungslogik ein möglichst platzsparendes und somit kostengünstiges Chiplayout zu berechnen. Ein bereits bestehendes kräftebasiertes (engl. *force directed*) Platzierungsverfahren, welches bereits parallel arbeitet, wurde in stromorientierter Form reimplementiert.

Das kräftebasierte Platzierungsverfahren modelliert das Minimierungsproblem mithilfe von Kräften, die zwischen den Bausteinen „wirken“. Je größer die Entfernung zwischen zwei Bausteinen ist, desto stärker wirkt die daraus resultierende Kraft. Durch Änderung der Position der Bausteine auf der Chipfläche neutralisieren sich die Kräfte, wobei Randbedingungen wie Überlappungsfreiheit zu berücksichtigen sind. Somit besitzen die Bausteine das Bestreben, eine kompakte Anordnung anzunehmen und so die benötigte Chipfläche zu minimieren.

Der Aufbau der stromorientierten Version ist in Abbildung 7.2 dargestellt. Die Implementierung beruht auf einem Fließband mit Rückkopplung und drei replizierbaren Stufen. Ein Fließbanddurchlauf entspricht einer Iteration des kräftebasierten Verfahrens. Die erste Stufe erzeugt für die initiale Schaltungslogik einen Strom von Platzierungsaufgaben. Die nächste

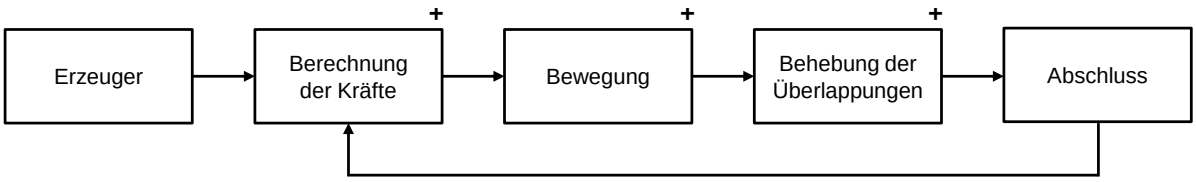


Abbildung 7.2: Aufbau von Electric

Stufe berechnet die Kräfte für die Platzierungen, so dass die dritte Stufe die Bausteine gemäß diesen Kräften bewegen kann. Mögliche Überlappungen sind hierbei nicht ausgeschlossen und werden in der vierten Fließbandstufe behoben. Die fünfte und letzte Stufe bewertet die Güte der neuen Platzierungen und schickt diese ggf. per Teleport an die zweite Stufe zurück, so dass die nächste Verarbeitungsiteration erfolgen kann. Das Abbruchkriterium der letzten Stufe ist erfüllt, wenn die Platzierung eine definierte Mindestgüte erfüllt oder eine bestimmte Zeitschranke überschritten ist. Die mittleren Stufen sind replizierbar, da zwischen den Platzierungsaufgaben keine Abhängigkeiten vorliegen. Die letzte Stufe ist nicht replizierbar, da sie die Ergebnisse der vorangehenden Stufe zusammenführt.

7.1.3 Java Grande (JG)

Die Java Grande Benchmark Suite ist eine Programmsammlung, um die Leistung von Java-Ausführungsumgebungen bestimmen und vergleichen zu können. Die Programme führen numerische Berechnungen mit unterschiedlichem Verarbeitungs- und Speicheraufwand durch. Acht Programme liegen in sowohl sequenziellen als auch mehrfädigen Implementierungen vor. Die Programme der mehrfädigen Benchmark Suite basieren auf Daten- und Aufgabenparallelität [95].

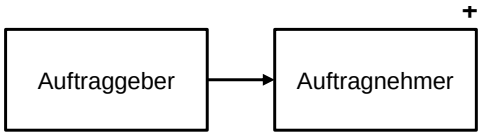


Abbildung 7.3: Aufbau der Java Grande Series Benchmark

Für die Evaluation dieser Arbeit wurden stromorientierte Varianten der parallelen Benchmark-Programme erstellt, um Codestruktur und Performanz vergleichen zu können. Die Parallelität der stromorientierten Programme basieren auf dem Auftraggeber-Auftragnehmer-Muster, welches in Abbildung 7.3 dargestellt ist.

7.1.4 JPEG-Transkodierung (Jpeg)

Jpeg ist ein Programm zur Transkodierung von Bildern im jpeg-Format. Das Programm führt eine Neukomprimierung eines Bildes durch, so dass das Bild anschließend in einer anderen Qualitätsstufe vorliegt. Die Implementierung basiert auf einem Programm aus dem StreamIt-Benchmark; es handelt sich also um ein „klassisches“ Stromprogramm mit feingranularer, geschachtelter Parallelität und Datenströmen primitiven Typs.

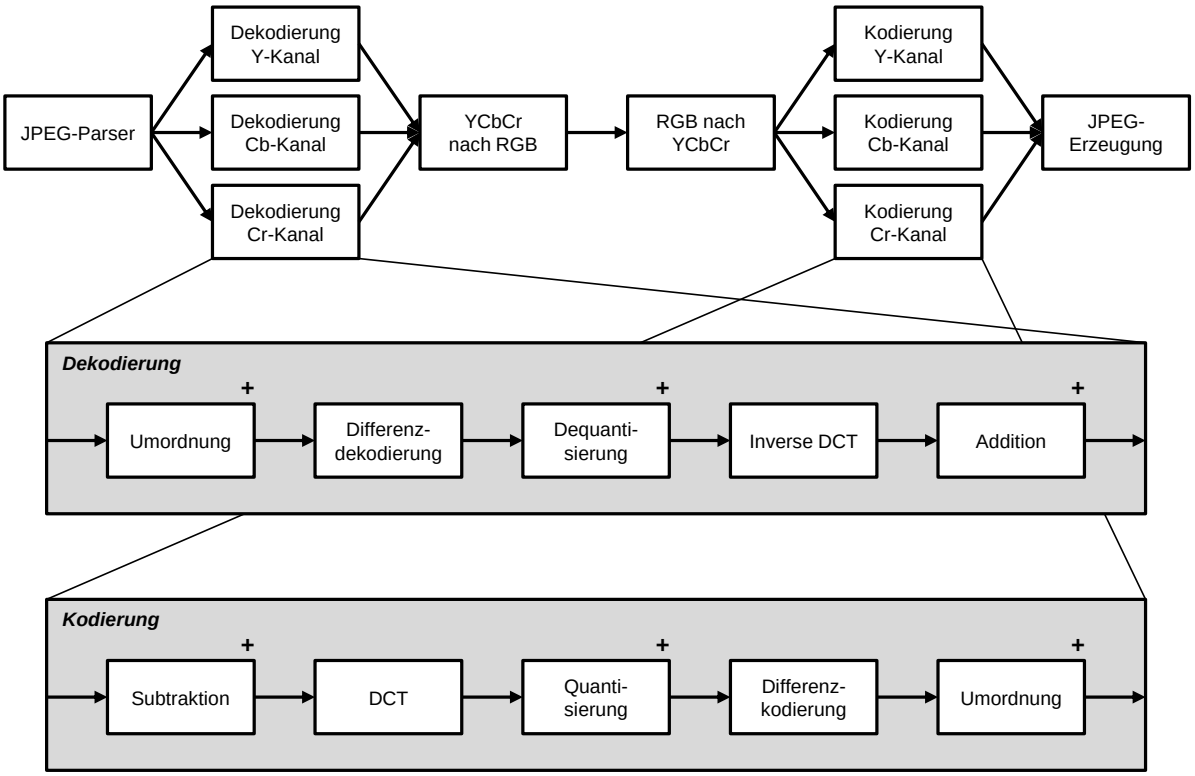


Abbildung 7.4: Aufbau der JPEG-Transkodierung

Der Aufbau der stromorientierten Version ist in Abbildung 7.4 dargestellt. Das Grundmuster ist ein sechsstufiges, nichtlineares Fließband. Die ersten drei Stufen sind für die Zerteilung, Kanaldekodierung und Farbkonversion von YCbCr- in RGB-Form zuständig. Die letzten drei Stufen kehren diese Schritte wieder um, indem sie entsprechend neuer Qualitätsparameter die RGB-Darstellung zurück in YCbCr-Form konvertieren, die Farbkanäle kodieren und zu einem jpeg-Bild anderer Qualitätsstufe zusammenführen. Die Dekodierung und Kodierung (zweite und fünfte Stufe) erfolgen farbkomponentenweise und bestehen wiederum aus Fließbändern.

In der ursprünglichen StreamIt-Version besteht der Datenstrom aus einzelnen Bytes, die als Ganzzahl dargestellt sind. Da die Dekodierung bzw. Kodierung auf Blöcken der Größe 8×8

Bytes basiert, müssen einige Filter 64 aufeinander folgende Stromelemente puffern. Damit sind diese Filter zustandsbehaftet und können nicht repliziert werden.

In der hier untersuchten Implementierung werden daher keine einzelnen Bytes, sondern Blöcke der Größe 8×8 Bytes als Stromelemente verwendet. Diese Blöcke können unabhängig voneinander transformiert werden, so dass für einige Filter Zustandslosigkeit und somit Replizierbarkeit erreicht wird. Lediglich die Filter für die Differenzkodierung bzw. -dekodierung bleiben zustandsbehaftet und daher nicht replizierbar, da an diesen Stellen Abhängigkeiten zwischen den Byte-Blöcken bestehen.

In der sequenziellen Implementierung ist die Funktionalität der Filter durch entsprechende Methoden beschrieben. Die sequenzielle Transkodierung wird erreicht durch eine Schleife, die über sämtliche Blöcke eines Bilds iteriert und dabei die genannten Methoden aufruft.

Die fadenbasierte Implementierung entspricht einer Parallelisierung der sequenziellen Version. Hierbei wird das zu verarbeitende Bild in Bereiche aufteilt, die von mehreren Arbeiterfäden transkodiert werden. Diese naheliegende Implementierung unterscheidet sich also deutlich von der strombasierten Version, in der die Parallelität wesentlich feingranularer strukturiert ist.

7.1.5 Mergesort (Msort)

Msort sortiert ein Array aus etwa 33 Millionen zufällig erzeugten Ganzzahlwerten mithilfe des Mergesort-Algorithmus. Dieses Sortierverfahren ist ein klassischer Vertreter des Teile-und-Herrsche-Prinzips (engl. *divide and conquer*). Das zu sortierende Array wird rekursiv in Teile zerlegt und sortiert. Die sortierten Teilarrays werden anschließend sukzessive zusammengeführt.

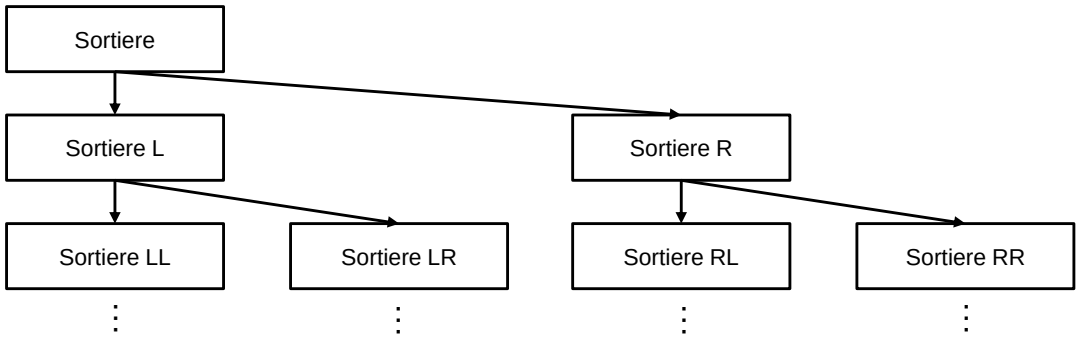


Abbildung 7.5: Aufbau von Mergesort

Die stromorientierte Version ist implementiert durch einen Filter, der sich selbst rekursiv aufruft (vgl. Abbildung 7.5). Insofern handelt es sich hierbei also um keine echte Stromverarbeitung. Potenziell vergrößert sich der Parallelitätsgrad in jedem Rekursionsschritt um den

Faktor 2, so dass ein möglichst guter Kompromiss zwischen Parallelitätsgrad und dessen Verwaltungsaufwand gefunden werden muss.

7.1.6 Videoverarbeitung (Vscale und Vzoom)

Bei der ursprünglichen Version der Videoverarbeitung handelt es sich um eine C#-Implementierung, die in Java-Quelltext umgewandelt und erweitert wurde [33]. Wir betrachten die Komponenten Vscale und Vzoom; deren Aufbau in den Abbildungen 7.6 bzw. 7.7 illustriert ist. In beiden Fällen handelt es sich um Fließbänder, die ein Video einlesen, einen Strom aus Einzelbildern erzeugen, diesen verarbeiten und anschließend in einer neuen Videodatei abspeichern.

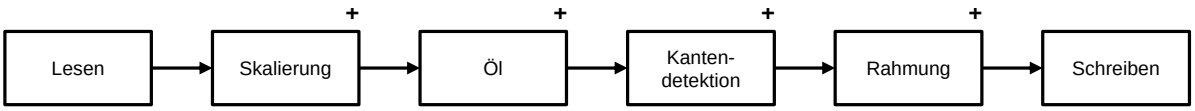


Abbildung 7.6: Aufbau der Videoverarbeitung Vscale

Vscale dient dazu, die Auflösung eines Videos zu erhöhen. Hierfür werden die Bilder skaliert, deren Auflösung durch die Stufen Ölgemälde und Kantendetektion erhöht und schließlich neu gerahmt. Diese Schritte sind als aufeinander folgende replizierbare Fließbandstufen angeordnet.

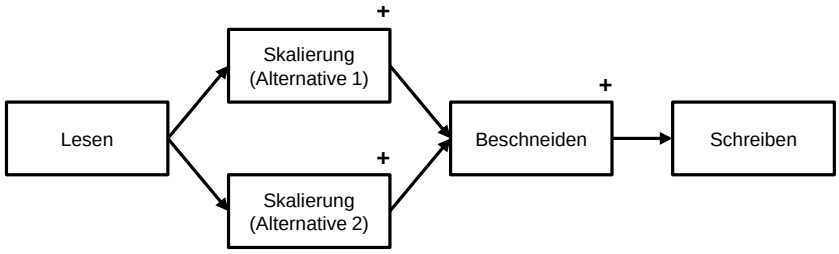


Abbildung 7.7: Aufbau der Videoverarbeitung Vzoom

Vzoom erlaubt das „Hineinzoomen“ in ein Video. Ähnlich wie in Vscale werden die Einzelbilder zunächst skaliert. Das Format der entstehenden Bilder ist nun größer und wird durch das anschließende Beschneiden auf das Ursprungsformat verkleinert, so dass sich ein Zoom-Effekt ergibt. Eine Besonderheit von Vzoom ist die Existenz alternativer Implementierungen der Skalierungsfunktion. Ein interessanter Tuningparameter dieser Anwendung ist demnach die Auswahl einer Implementierung. Da die Bilder eines Videos voneinander unabhängig sind, sind die Stufen zum Skalieren und Beschneiden replizierbar.

7.2 Ergebnisse

Nachdem die untersuchten Anwendungen im vorigen Abschnitt vorgestellt wurden, widmet sich dieser Abschnitt den Ergebnissen der Fallstudien. Entsprechend den Zielen und Thesen dieser Arbeit untersuchen wir zunächst die Implementierungen bezüglich ihrer Codestruktur (Abschnitt 7.2.1), gefolgt von der Auswertung der Zustands- und Konfliktanalyse (Abschnitt 7.2.2) sowie den Ergebnissen der Performanzoptimierung (Abschnitt 7.2.3).

7.2.1 Implementierungsaufwand und Fehleranfälligkeit

Dieser Abschnitt soll Anhaltspunkte liefern, inwieweit sich stromorientierte Konzepte zur Implementierung allgemeiner Anwendungen eignen. Für die in Abschnitt 7.1 beschriebenen Anwendungen vergleichen wir jeweils drei verschiedene Implementierungen:

- *seq*: sequenzielle Implementierung
- *par*: parallele Implementierung basierend auf Fäden und/oder Konstrukten der Bibliothek `java.util.concurrent`
- *stb*: strombasierte Implementierung

7.2.1.1 Evaluationsmetriken

Zunächst legen wir die Metriken fest, mit deren Hilfe der Code bewertet und verglichen wird. Um Aussagen über die Codestruktur und somit die Präzision der Sprachkonzepte treffen zu können, ermitteln wir für die sequenziellen (*seq*), fadenbasierten (*par*) und strombasierten (*stb*) Implementierungen nachfolgende Kenngrößen, welche wir als Indikatoren für Codeumfang bzw. Implementierungsaufwand (*PLOC*) und Fehleranfälligkeit (*SHV*, *CRS*, *SYN*) betrachten:

- *PLOC*: Anzahl der Codezeilen (engl. *lines of code*) der Programmkomponenten, die für die Parallelisierung relevant sind
- *SHV*: Anzahl der gemeinsamen Variablen (engl. *shared variables*)
- *CRS*: Anzahl der kritischen Abschnitte (engl. *critical sections*), innerhalb derer auf gemeinsame Variablen zugegriffen wird
- *SYN*: Anzahl der verwendeten Synchronisierungskonstrukte (Sperren, `wait()`, `join()`, `notify()`/`notifyAll()` sowie Datenstrukturen aus `java.util.concurrent`)

Tabelle 7.1 zeigt die Codemetriken, die sich aus den sequenziellen, fadenbasierten und stromorientierten Implementierungen der untersuchten Anwendungen ergeben. Die folgenden Abschnitte gehen näher auf die einzelnen Anwendungen ein.

		DS	Electric	JG	Jpeg	Msort	Vscale	Vzoom
PLOC	*seq*	991	677	3613	1119	72	396	396
	par	1330	719	4510	1168	118	491	491
	stb	937	675	4426	992	72	458	460
	stb ↔ *seq*	-54	-2	813	-127	0	62	64
		-5%	0%	+23%	-11%	0%	+16%	+16%
	stb ↔ *par*	-393	-44	-84	-176	-46	-33	-31
		-30%	-6%	-2%	-15%	-39%	-7%	-6%
SHV	*par*	3	2	1	1	0	0	0
	stb	2	2	1	0	0	0	0
	stb ↔ *par*	-1	0	0	-1	0	0	0
		-33%	0%	0%	-100%	–	–	–
CRS	*par*	3	4	1	2	0	0	0
	stb	2	3	1	0	0	0	0
	stb ↔ *par*	-1	-1	0	-2	0	0	0
		-33%	-25%	0%	-100%	–	–	–
SYN	*par*	23	10	11	4	1	2	2
	stb	3	5	1	0	0	0	0
	stb ↔ *par*	-20	-5	-10	-4	-1	-2	-2
		-87%	-50%	-91%	-100%	-100%	-100%	-100%

Tabelle 7.1: Vergleich der sequenziellen (*seq*), fadenbasierten (*par*) und strombasierten (*stb*) Implementierungen der untersuchten Anwendungen

7.2.1.2 DS

Der Codeumfang der strombasierten Version reduziert sich gegenüber der fadenbasierten Implementierung besonders stark (rund 30%). Selbst im Vergleich zur sequenziellen Implementierung ist der strombasierte Code kürzer. Dies ist dadurch zu erklären, dass die Varianten *seq* und *par* tuningfähig implementiert wurden und somit zusätzliche Codevarianten und Parameter existieren [70]. Die Tuningfähigkeit der Stromvariante ist dagegen durch die automatische Extraktion von Tuningparametern implizit gegeben.

Des Weiteren wird die Anzahl der gemeinsamen Variablen und der damit verbundenen kritischen Abschnitte von 3 auf 2 reduziert. Zudem werden 87% weniger Synchronisierungskonstrukte benötigt, so dass sich das damit verbundene Fehlerpotenzial deutlich verringert.

7.2.1.3 Electric

Die Fließbandstruktur von Electric, bestehend aus replizierbaren Stufen und einer Rückkopplungsschleife, lässt sich mithilfe stromorientierter Konzepte wesentlich einfacher umsetzen als

unter Verwendung von Fäden. In der fadenbasierten Version verwendet Electric bereits interne Klassen und Datenstrukturen zum Erzeugen und Verwalten von Ausführungsfäden. Ohne diese Hilfsklassen müsste wesentlich mehr Parallelität „von Hand“ programmiert werden, der fadenbasierte Code wäre somit noch umfangreicher.

Der Codeumfang der strombasierten Implementierung verringert sich im Vergleich zur fadenbasierten Version um rund 6%. In beiden Varianten existieren zwei gemeinsame Variablen. In der fadenbasierten Implementierung ergeben sich daraus vier kritische Abschnitte, in der stromorientierten Version sind es nur noch drei. Die Anzahl der benötigten Synchronisierungsprimitiven wird um 50% verringert.

Der Umfang des sequenziellen und strombasierten Codes ist interessanterweise in etwa gleich. Die Filter der strombasierten Version sind hier sehr ähnlich aufgebaut wie ihre korrespondierenden Methoden aus der sequenziellen Implementierung.

7.2.1.4 JG

Die parallele Java Grande Benchmark Suite besteht aus acht Programmen, deren Parallelität meist durch einen einfachen Fork-Join-Abschnitt erzeugt wird. In der stromorientierten Version wird hierfür entweder reine Aufgabenparallelität oder das Auftraggeber-Auftragnehmer-Muster verwendet. Die Codeeinsparung ist nicht ganz so deutlich wie bei den vorigen Anwendungen.

Innerhalb der gesamten Benchmark Suite existiert sowohl für die ursprüngliche fadenbasierte als auch die stromorientierte Version jeweils ein kritischer Abschnitt. In der stromorientierten Implementierung werden außer einer Sperre für den kritischen Abschnitt keine weiteren Synchronisierungskonstrukte benötigt, was gegenüber der fadenbasierten Implementierung einer Reduktion um 91% entspricht.

7.2.1.5 Jpeg

Der Codeumfang der stromorientierten Version ist rund 11% geringer als in der sequenziellen Version. Die Implementierung der Filter und ihrer entsprechenden sequenziellen Methoden besitzt in etwa denselben Umfang; die Codereduktion resultiert im Wesentlichen aus der Beschreibung der Verarbeitungsstruktur. Insbesondere die für das Verfahren notwendige Aufteilung und Zusammenführung von Bilddaten erfolgt in der stromorientierten Implementierung auf sehr einfache Weise, nämlich durch die Verwendung entsprechender Split/Join-Operatoren. Im sequenziellen Fall müssen Zwischenergebnisse explizit behandelt und ggf. in temporären Arrays abgelegt werden, die in der stromorientierten Version nicht notwendig sind.

Bei der fadenbasierten Version wurde darauf Wert gelegt, eine möglichst naheliegende Implementierung zu verwenden. Der fadenbasierte Code orientiert sich daher nicht an der exakten, aufwändig zu implementierenden Fließbandstruktur der stromorientierten Version, sondern stellt eine Parallelisierung der sequenziellen Version dar. Hierbei wird das Eingabebild in Gebiete zerlegt, die nebenläufig transkodiert und anschließend wieder zusammengeführt werden. Der resultierende Code ist umfangreicher als im sequenziellen und strombasierten Fall. Bei exakter Reimplementierung der stromorientierten Fließbandstruktur wäre der Codeumfang noch größer.

Der fadenbasierte Code enthält eine gemeinsame Variable, zwei kritische Abschnitte sowie vier Synchronisierungsprimitiven, welche allesamt in der stromorientierten Implementierung eingespart werden.

7.2.1.6 Msort

Msort sortiert ein Array nach dem Mergesort-Verfahren; Stromverarbeitung im eigentlichen Sinne findet hierbei nicht statt. Die Implementierung der stromorientierten Version ist nahezu identisch zur sequenziellen Implementierung. Lediglich die Methode, die den Teile-und-Herrsche-Schritt kapselt, wird in der strombasierten Version durch einen (nicht-stromverarbeitenden) Filter mit derselben Funktionalität ersetzt; dementsprechend besteht der Rekursionsschritt hier nicht aus zwei Methodenaufrufen, sondern zwei aufgabenparallelen Filteraufrufen.

Der Code der fadenbasierten Version ist dagegen umfangreicher, da eine weitere Klasse zur Kapselung eines nebenläufigen Sortierschritts benötigt wird. Darüber hinaus ist ein Umschaltkriterium im Code spezifiziert, welches den Wechsel vom parallelen zum sequenziellen Modus bestimmt. Das Synchronisieren der Fäden vor Zusammenführung der Teilergebnisse erfolgt hier explizit, in der stromorientierten Version implizit.

7.2.1.7 Vscale und Vzoom

Sowohl für Vscale als auch Vzoom ist der stromorientierte Code kürzer als der fadenbasierte Code und umfangreicher als die sequenzielle Version. In den fadenbasierten Implementierungen wurden die Fließbänder durch die Zusammenfassung von replizierbaren Stufen bereits maximal gekürzt; der zur stromorientierten Version äquivalente fadenbasierte Code wäre somit noch deutlich länger.

Die in den fadenbasierten Implementierungen notwendigen Synchronisierungsprimitiven zum Koordinieren und Beenden des Fließbands werden in den jeweiligen stromorientierten Varianten eingespart; hier erfolgt die Synchronisierung implizit. Sowohl die fadenbasierte als auch die stromorientierte Implementierung beinhalten keine gemeinsamen Variablen und demnach auch keine zu schützenden kritischen Abschnitte.

7.2.1.8 Fazit

Es lässt sich festhalten, dass stromorientierte Programmierkonzepte sich nicht nur für spezielle Programmtypen, sondern für Anwendungen aus verschiedenen Bereichen und unterschiedlicher Größe eignen. Die relevanten parallelen Muster lassen sich präzise implementieren, so dass im Vergleich zur Verwendung von Fäden kürzerer Code entsteht, der von einem Großteil der Parallelisierung abstrahiert und somit auch das Fehlerrisiko reduziert. Im Einzelnen lassen sich folgende Aussagen treffen:

- Der Codeumfang der fadenbasierten Implementierungen ist erwartungsgemäß höher als in der sequenziellen Version.
- Im Vergleich zum sequenziellen Code ist der strombasierte Code nur geringfügig länger (JG, Vscale, Vzoom), etwa gleich lang (Electric, Msort) oder sogar kürzer (DS, Jpeg).
- Im Vergleich zum fadenbasierten Code ist der strombasierte Code nicht nur kürzer, sondern besitzt durchschnittlich auch 29% weniger gemeinsame Variablen (*SHV*), 40% weniger kritische Abschnitte (*CRS*) sowie rund 83% weniger Synchronisierungsprimitiven (*SYN*).

7.2.2 Zustandsbehaftete Filter und kritische Abschnitte

Dieser Abschnitt befasst sich mit den Ergebnissen der Zustands- und Konfliktanalyse. Mit Ausnahme von Msort besitzen alle Anwendungen sowohl zustandslose als auch zustandsbehaftete Filter. Wie Abschnitt 7.2.1 zu entnehmen ist, existieren in drei Anwendungen (DS, Electric, JG) gemeinsame Variablen bzw. kritische Abschnitte.

7.2.2.1 Evaluationsmetriken

Die Qualität der Zustands- und Konfliktanalyse bewerten wir anhand folgender Metriken:

- tp: Anzahl korrekter Fundstücke (true positives)
- fp: Anzahl falscher Fundstücke (false positives)
- fn: Anzahl fehlender Fundstücke (false negatives)

Anhand dieser Werte lassen sich Aussagen über Präzision und Ausbeute der Analysen treffen:

- Die Präzision (engl. *precision*) $P = \frac{tp}{tp+fp}$ gibt an, wie viele Fundstücke relevant sind.
- Die Ausbeute (engl. *recall*) $R = \frac{tp}{tp+fn}$ gibt an, wie viele relevante Stücke gefunden wurden.

Um die Wirksamkeit der Konfliktanalyse zu untersuchen, wurden zuvor alle Sperren aus dem Code entfernt. Gegenstand der Untersuchung ist also bewusst fehlerhafter Code, welcher bekannte, ungeschützte kritische Abschnitte enthält. Die Konfliktanalyse wurde in zwei verschiedenen Modi durchgeführt. Der filterkontextinsensitive Modus (fci) unterscheidet nicht zwischen unterschiedlichen Instanzen bzw. Kontexten desselben Filters. Im filterkontextsensitiven Modus (fcs) wird dieser Aspekt dagegen berücksichtigt.

Die Tabellen 7.2 und 7.3 zeigen die Ergebnisse der Zustands- bzw. Konfliktanalyse sowie deren Präzision und Ausbeute. Die einzelnen Anwendungen werden im Folgenden näher betrachtet.

	DS	**Electric**	**JG**	**Jpeg**	**Msort**	**Vscale**	**Vzoom**	**Gesamt**
Filter	4	5	17	24	1	9	10	70
mit Zustand	1	2	9	16	1	2	3	34
tp	1	1	9	16	1	2	3	33
fp	1	0	0	7	0	0	0	8
ohne Zustand	3	3	8	8	0	7	7	36
tp	2	3	8	1	0	7	7	28
fp	0	1	0	0	0	0	0	1
P	75%	80%	100%	71%	100%	100%	100%	87%

Tabelle 7.2: Ergebnisse der Zustandsanalyse

	DS	**Electric**	**JG**	**Jpeg**	**Msort**	**Vscale**	**Vzoom**	**Gesamt**
CRS	2	3	1	0	0	0	0	6
tp	2	3	1	0	0	0	0	6
fp-fci	8	1	44	1	2	0	0	56
fp-fcs	3	1	0	1	0	0	0	5
fn	0	0	0	0	0	0	0	0
P-fci	20%	75%	2%	0%	0%	–	–	10%
P-fcs	40%	75%	100%	0%	–	–	–	55%
R	100%	100%	100%	–	–	–	–	100%

Tabelle 7.3: Ergebnisse der Konfliktanalyse

7.2.2.2 DS

Die Zustandsanalyse klassifiziert drei der vier Filter korrekt. Ein Filter besitzt einen „harmlosen“ Zustand in Form einer Zählvariable, die die vom Filter gelesene Datenmenge erfasst. Da die Variablensemantik außerhalb der Reichweite von Programmanalysen ist, wird dieser Filter als zustandsbehaftet markiert.

Die Konfliktanalyse erkennt alle kritischen Abschnitte und erreicht somit eine Ausbeute von 100%. Im filterkontextinsensitiven Modus (fci) ergeben sich acht Falschmeldungen; im genaueren filterkontextsensitiven Modus (fcs) nur noch drei, was zu einer Präzision von 20% bzw. 40% führt.

7.2.2.3 Electric

Für die Anwendung Electric klassifiziert die Zustandsanalyse vier der fünf Filter korrekt, was einer Präzision von 80% entspricht. Ein zustandsbehafteter Filter wird als zustandslos eingestuft. Hierbei handelt es sich um den letzten Filter des Fließbands, der die Teilergebnisse seiner Vorgängerstufe zusammenführt, um diese ggf. per Rückkopplung an eine vordere Stufe zurückzugeben. Der letzte Filter besitzt einen „versteckten“ Zustand, der in Datenstrukturen außerhalb des Filters gespeichert wird und zur Zusammenführung der Teilergebnisse dient. Eine für die Analyse erkennbare Zustandsvariable innerhalb des Filters existiert daher nicht.

Die Ausbeute der Konfliktanalyse liegt bei 100%; alle kritischen Abschnitte werden erkannt. Sowohl im filterkontextinsensitiven als auch -kontextsensitiven Modus tritt eine Falschmeldung auf. Diese bezieht sich auf einen Variablenzugriff durch jenen zustandsbehafteten Filter, der zuvor fälschlicherweise als zustandslos klassifiziert wurde. Somit nimmt die Konfliktanalyse an, dass mehrere Instanzen dieses Filters zeitgleich existieren könnten; in diesem Fall wäre ein Zugriffskonflikt in der Tat möglich. Insgesamt ergibt sich eine Präzision von 75%.

7.2.2.4 JG

Die Programme der Java Grande Benchmark bestehen aus insgesamt neun zustandsbehafteten und acht zustandslosen Filtern, die allesamt korrekt klassifiziert werden. Die Präzision der Zustandsanalyse liegt somit bei 100%.

Es existiert ein kritischer Abschnitt; dieser wird von der Konfliktanalyse erkannt. Im filterkontextinsensitiven Modus entstehen 44 Falschmeldungen, die im filterkontextsensitiven Modus allesamt eliminiert werden. In einigen Programmen der Benchmark Suite greifen replizierbare Filter auf Methoden und Variablen zu, die zwar in derselben Klasse deklariert sind, aber stets unterschiedlichen Objekte angehören. Die Präzisionen von 2% und 100% verdeutlichen in diesem Beispiel den Nutzen der Kontextsensitivität.

7.2.2.5 Jpeg

Die Zustandsanalyse klassifiziert 17 der 24 Filter korrekt, was einer Präzision von 71% entspricht. Bei den 17 Filtern handelt es sich um 16 zustandsbehaftete und einen zustandslosen Filter. Die restlichen sieben Filter sind zustandslos, besitzen jedoch Zählvariablen und somit

„harmlose“ Zustände. Es besteht also eine ähnliche Problematik wie bei der Anwendung DS; Filter, deren Replikation zulässig wäre, werden als zustandsbehaftet klassifiziert.

Darüber hinaus wird ein Zugriffskonflikt identifiziert, der sich als Falschmeldung herausstellt.

7.2.2.6 Msort

Msort besteht aus lediglich einem zustandsbehafteten Filter, der auch als solcher erkannt wird. Die filterkontextinsensitive Konfliktanalyse produziert zwei Falschmeldungen für das Sortieren und Zusammenführen von Teilarrays. Hierbei wird nicht berücksichtigt, dass die entsprechenden Filter- bzw. Methodenaufrufe auf unterschiedlichen Teilarrays erfolgen. Die filterkontextsensitive Analyse erkennt dies jedoch und liefert korrekte Ergebnisse.

7.2.2.7 Vscale und Vzoom

Vscale und Vzoom bestehen aus zwei bzw. drei zustandsbehafteten und jeweils sieben zustandslosen Filtern, die allesamt korrekt klassifiziert werden. Die Präzision liegt somit bei 100%. Es existieren keine gemeinsamen Variablen bzw. kritischen Abschnitte; Falschmeldungen werden durch die Konfliktanalyse nicht hervorgerufen.

7.2.2.8 Fazit

Die Zustandsanalyse kann den Programmierer dabei unterstützen, zustandsbehaftete Filter zu erkennen. Für die untersuchten Anwendungen erreicht sie eine Präzision von 87%. Die meisten Falschmeldungen (8 von 9) beziehen sich auf zustandslose Filter, die als zustandsbehaftet klassifiziert werden. In diesen Fällen besaßen die Filter einen „harmlosen“ Zustand wie z.B. eine lokale Zählvariable. Da die Variablensemantik von Programmanalysen nicht ohne Weiteres erfasst werden kann, ist eine Unterscheidung zwischen echten und harmlosen Zuständen schwierig.

Für die Parallelisierung problematischer sind zustandsbehaftete Filter, da diese im Falle einer Replizierung zu einer Veränderung der Programmsemantik und somit falschen Ergebnissen führen können. Hier konnte die Zustandsanalyse 33 von 34 zustandsbehafteten Filtern korrekt identifizieren.

Drei der sieben strombasierten Implementierungen besitzen gemeinsame Variablen und kritische Abschnitte. Die Konfliktanalyse konnte alle kritischen Abschnitte erkennen. Ohne Berücksichtigung des Filterkontexts entstehen 56, andernfalls nur 5 Falschmeldungen, was zu einer Präzision von 10% bzw. 55% führt. Dies zeigt, dass Konfliktanalysen einen möglichen Weg darstellen, möglichen Wettlauffehlern bereits zur Übersetzungszeit vorzubeugen.

7.2.3 Performanz

Dieser Abschnitt untersucht die Wirksamkeit der Konzepte zur Performanzoptimierung. Es werden Laufzeitmessungen auf drei verschiedenen Rechnerarchitekturen durchgeführt:

- **Q6600 (Vierkern).** Intel Core 2 Quad Q6600 Prozessor (4 Kerne, 2,4 GHz Taktfrequenz), 4 GB Hauptspeicher
- **E5320 (Doppelvierkern).** Zwei Intel Xeon E5320 Prozessoren (2×4 Kerne, 1,86 GHz Taktfrequenz), 8 GB Hauptspeicher
- **T2 (Niagara).** Sun UltraSPARC T2 Prozessor (8 Kerne mit je 8 Hardwarefäden, 1,6 GHz Taktfrequenz), 16 GB Hauptspeicher

7.2.3.1 Evaluationsmetriken

Die folgenden Ausführungszeiten werden gemessen und verglichen:

- *seq*: Die Ausführungszeit ohne Verwendung von Parallelität
- *par*: Die Ausführungszeit der parallelen, von Hand optimierten Java-Implementierung
- *stb-pre*: Die Ausführungszeit der strombasierten Implementierung, die mithilfe der Tuningheuristiken konfiguriert ist (vgl. Abschnitt 5.2.3)
- *stb-lt*: Die Ausführungszeit der strombasierten Implementierung bei Durchführung von Laufzeit-Tuning (vgl. Abschnitt 5.3)
- *stb-best*: Die Ausführungszeit der strombasierten Implementierung bei Verwendung der besten bekannten Parameterkonfiguration, welche durch Offline-Tuning empirisch ermittelt wurde

Anhand dieser Ausführungszeiten berechnen wir folgende Metriken, um die Qualität der Stromverarbeitung und der Optimierung zu beurteilen:

- Die *Beschleunigung* (engl. *speedup*)

  $$S(I) = \frac{t_{seq}}{t_I}$$

  gibt an, um welchen Faktor sich die Ausführungszeit t_I einer parallelen Implementierung I gegenüber der sequenziellen Laufzeit t_{seq} verbessert. Die Beschleunigung ist ein Maß für den Leistungsgewinn durch Parallelisierung.

- Die *Leistungsqualität* (engl. *performance quality*)

$$PQ(I) = \frac{S(I)}{S(stb\text{-}best)}$$

beschreibt das Verhältnis zwischen der Beschleunigung $S(I)$ einer Implementierung I und der besten Beschleunigung $S(stb\text{-}best)$. Die Leistungsqualität gibt somit die prozentuale Ausschöpfung derjenigen Leistung an, die für ein Stromprogramm bei bester bekannter Parameterkonfiguration erreichbar wäre.

- Der *Laufzeit-Tuning-Gewinn* (engl. *online tuning performance gain*)

$$OTPG = \frac{S(stb\text{-}lt) - S(stb\text{-}pre)}{S(stb\text{-}pre)} = \frac{PQ(stb\text{-}lt) - PQ(stb\text{-}pre)}{PQ(stb\text{-}pre)}$$

ist ein Maß für den Mehrwert von Laufzeit-Tuning. Der Mehrwert ist der Leistungsgewinn, der für ein vorkonfiguriertes Programm (*stb-pre*) allein durch Laufzeit-Tuning erreicht wird.

Parametertyp	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom
TC	1	1	1	1	1	1	1
EM	1	1	1	9	1	1	1
SF	2	3	1	25	0	5	1
RF	2	3	1	18	0	4	3
AL	0	0	0	0	0	0	1
AC	2	4	1	13	0	5	4
Gesamt	8	12	5	66	2	16	11

Tabelle 7.4: Identifizierte Tuningparameter (*TC* = Fadenanzahl, *EM* = Ausführmodus, *SF* = Stufenfusion, *RF* = Replikationsfaktor, *AL* = Alternative, *AC* = Aktionszahl)

Tabelle 7.4 gibt einen Überblick über die Tuningparameter, die in den Stromprogrammen automatisch identifiziert wurden. Der Suchraum ist umso größer, je mehr Parameter eine Anwendung besitzt. Abgesehen von Msort verarbeiten alle Anwendungen Datenströme und eignen sich somit für Laufzeit-Tuning. Da Msort lediglich aus einem nicht stromverarbeitenden Filter mit Rekursion besteht, ist hier keine nutzbare Periodizität gegeben und somit der Einsatz eines Laufzeit-Tuners effektlos. Da für Msort somit kein Unterschied zwischen den Varianten *stb-pre* und *stb-lt* besteht, gilt für die Beschleunigungs- und Leistungsqualitätswerte $stb\text{-}lt = stb\text{-}pre$.

	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom
Q6600	100	63	1052	112	10	988	280
E5320	100	290	1525	180	12	1573	514
T2	121	6646	6721	131	57	17688	11532

Tabelle 7.5: Sequenzielle Ausführungszeiten (Sekunden) der untersuchten Anwendungen

Die sequenziellen Ausführungszeiten der Programme sind Tabelle 7.5 zu entnehmen. Die Abbildungen 7.8 und 7.9 zeigen die Beschleunigungen S und Leistungsqualitäten PQ der untersuchten Anwendungen; eine Übersicht der Laufzeit-Tuning-Gewinne $OTPG$ findet sich in Abbildung 7.10.

7.2.3.2 DS

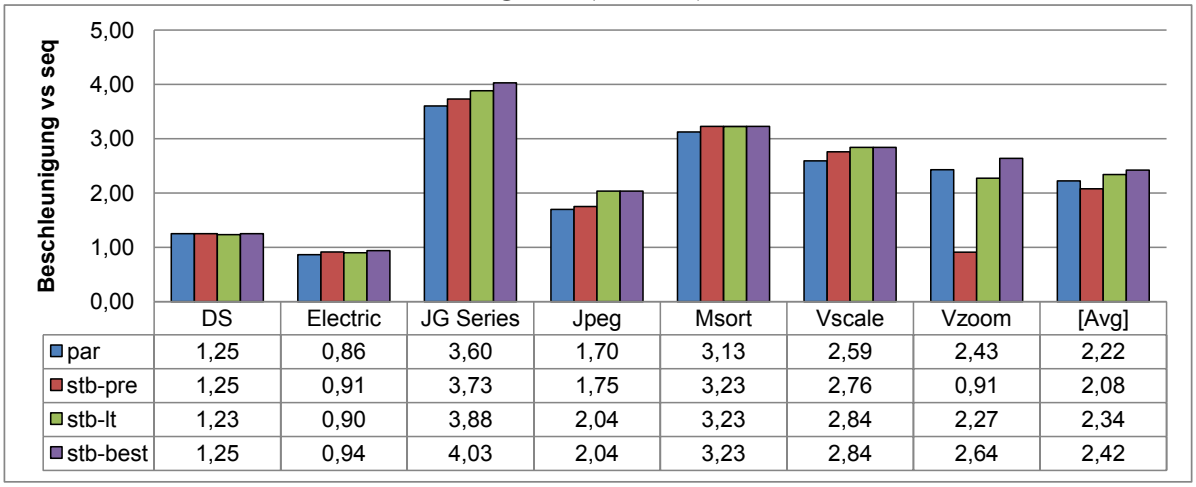
Bei DS handelt es sich um eine I/O-lastige Anwendung, so dass die bestmögliche Beschleunigung im Vergleich zur sequenziellen Laufzeit auf allen Rechnern nur leicht über 1 liegt. Auf dem Vierkernrechner und der Niagara wird durch die Vorkonfiguration der Parameter (*stb-pre*) bereits die beste Performanz erreicht; für den Doppelvierkernrechner liegt die Laufzeit knapp unter der sequenziellen Laufzeit. Demzufolge besteht bei Laufzeit-Tuning (*stb-lt*) wenig Raum für Verbesserungen gegenüber der Vorkonfiguration (*stb-pre*), andererseits sind auch keine nennenswerten Verschlechterungen zu beobachten. Die fadenbasierte, von Hand optimierte Implementierung (*par*) erreicht dieselbe Leistung wie ein Stromprogramm in bester Konfiguration (*stb-best*).

7.2.3.3 Electric

Electric besteht aus einem fünfstufigen Fließband mit Rückkopplung. Das Leistungspotenzial des Platzierungsverfahrens ist durch die Synchronisierung beim Zusammenführen von Teillösungen sowie die Rückkopplung im Fließband begrenzt. Auf dem Vierkernrecher entspricht die beste Konfiguration der sequenziellen Ausführung. Die Leistungsqualität der sequenziellen Implementierung liegt hier leicht über der Leistungsqualität der besten strombasierten Implementierung. Demgegenüber ergeben sich bei Anwendung der besten Parameterkonfiguration für den Doppelvierkernrechner und die Niagara gute Beschleunigungswerte von etwa $2,5$ bzw. 9; die vorkonfigurierte Version erreicht Beschleunigungen von etwa $1,75$ bzw. $5,8$. Die Leistung der optimierten fadenbasierten Implementierung (*par*) ist in etwa vergleichbar mit *stb-pre* und *stb-lt*, liegt jedoch auf allen Rechnern unter der Leistung der strombasierten Implementierung in bester Konfiguration (*stb-best*).

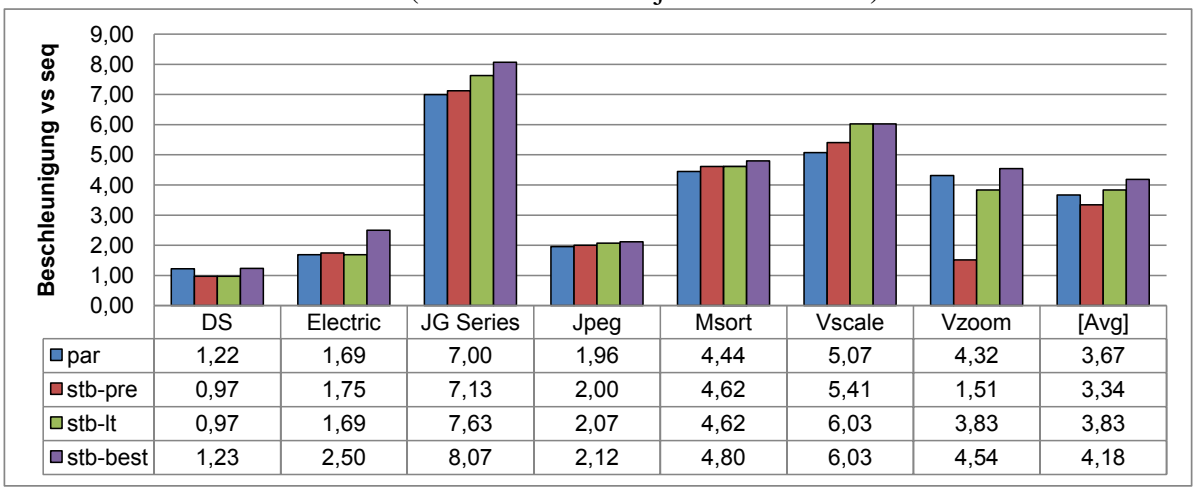
Mit Ausnahme des Vierkernrechners zeigt sich hier Potenzial für Verbesserungen durch Laufzeit-Tuning. Dieses wird jedoch nicht ausgenutzt, da sich das vom Auto-Tuner verwendete

Q6600 (4 Kerne)



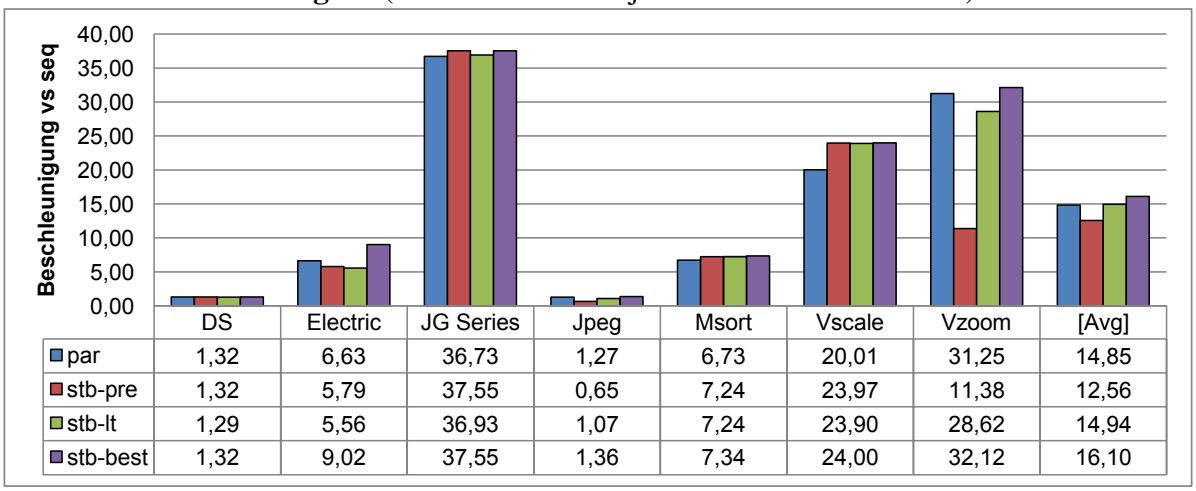
	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
par	1,25	0,86	3,60	1,70	3,13	2,59	2,43	2,22
stb-pre	1,25	0,91	3,73	1,75	3,23	2,76	0,91	2,08
stb-lt	1,23	0,90	3,88	2,04	3,23	2,84	2,27	2,34
stb-best	1,25	0,94	4,03	2,04	3,23	2,84	2,64	2,42

E5320 (2 Prozessoren mit jeweils 4 Kernen)



	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
par	1,22	1,69	7,00	1,96	4,44	5,07	4,32	3,67
stb-pre	0,97	1,75	7,13	2,00	4,62	5,41	1,51	3,34
stb-lt	0,97	1,69	7,63	2,07	4,62	6,03	3,83	3,83
stb-best	1,23	2,50	8,07	2,12	4,80	6,03	4,54	4,18

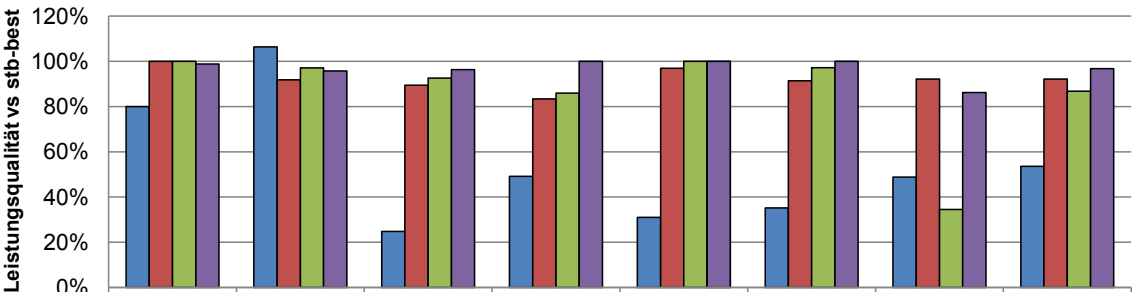
T2 Niagara (8 Prozessoren mit jeweils 8 Hardware-Fäden)



	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
par	1,32	6,63	36,73	1,27	6,73	20,01	31,25	14,85
stb-pre	1,32	5,79	37,55	0,65	7,24	23,97	11,38	12,56
stb-lt	1,29	5,56	36,93	1,07	7,24	23,90	28,62	14,94
stb-best	1,32	9,02	37,55	1,36	7,34	24,00	32,12	16,10

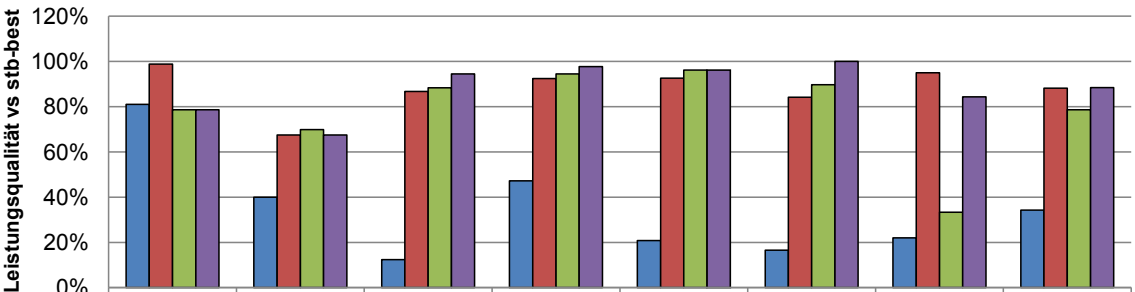
Abbildung 7.8: Beschleunigungen *S* gegenüber sequenzieller Ausführung

Q6600 (4 Kerne)



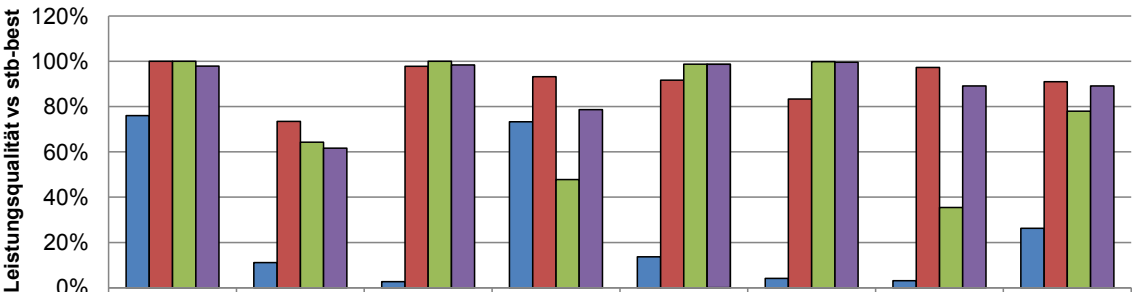
	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
seq	80%	106%	25%	49%	31%	35%	49%	54%
par	100%	92%	89%	83%	97%	91%	92%	92%
stb-pre	100%	97%	93%	86%	100%	97%	35%	87%
stb-lt	99%	96%	96%	100%	100%	100%	86%	97%

E5320 (2 Prozessoren mit jeweils 4 Kernen)



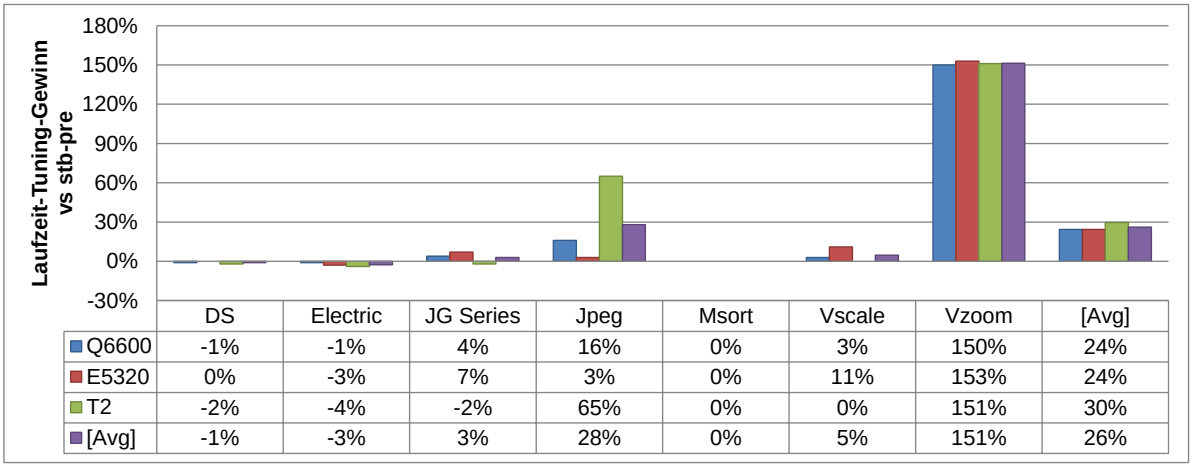
	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
seq	81%	40%	12%	47%	21%	17%	22%	34%
par	99%	67%	87%	92%	93%	84%	95%	88%
stb-pre	79%	70%	88%	94%	96%	90%	33%	79%
stb-lt	79%	67%	95%	98%	96%	100%	84%	88%

T2 Niagara (8 Prozessoren mit jeweils 8 Hardware-Fäden)



	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
seq	76%	11%	3%	73%	14%	4%	3%	26%
par	100%	73%	98%	93%	92%	83%	97%	91%
stb-pre	100%	64%	100%	48%	99%	100%	35%	78%
stb-lt	98%	62%	98%	79%	99%	100%	89%	89%

Abbildung 7.9: Leistungsqualitäten PQ bezüglich bester Beschleunigung. Für $PQ = 100\%$ ist die Leistungsqualität optimal



	DS	Electric	JG Series	Jpeg	Msort	Vscale	Vzoom	[Avg]
Q6600	-1%	-1%	4%	16%	0%	3%	150%	24%
E5320	0%	-3%	7%	3%	0%	11%	153%	24%
T2	-2%	-4%	-2%	65%	0%	0%	151%	30%
[Avg]	-1%	-3%	3%	28%	0%	5%	151%	26%

Abbildung 7.10: Laufzeit-Tuning-Gewinn *OTPG*

Optimierungsverfahren nach Nelder und Mead [73] in lokalen Extrema verfängt. Dies erklärt auch die geringfügigen Leistungseinbußen von 1% bis 4% gegenüber der vorkonfigurierten Version. An dieser Stelle sollte die Verwendung eines Verfahrens, welches eine bessere Suchraumexploration durchführt, zu besseren Ergebnissen führen.

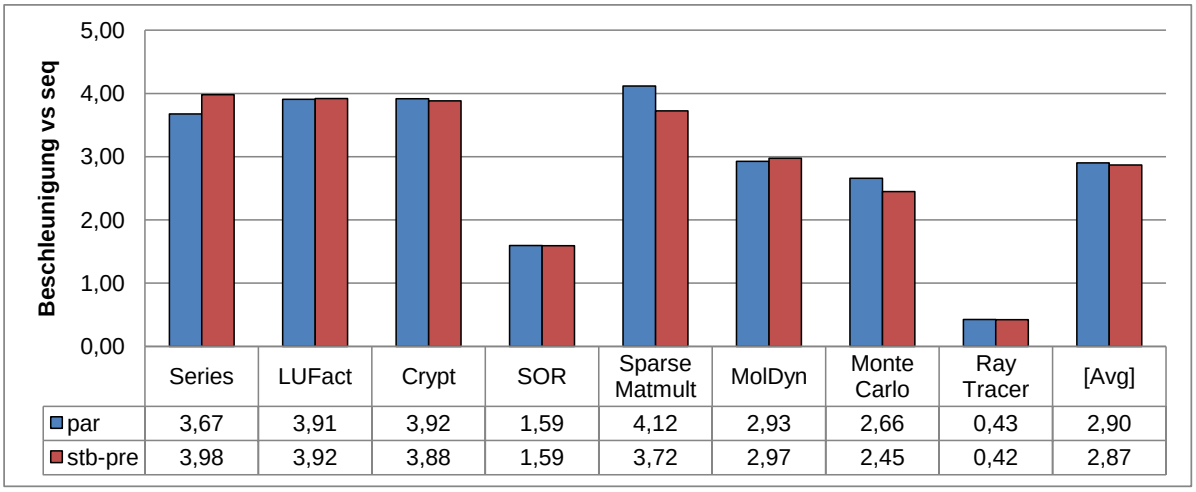
7.2.3.4 JG

Die parallele Java Grande Benchmark Suite weist mit Series lediglich ein Programm auf, welches sich für Laufzeit-Tuning eignet. Für die anderen Programme wurden zwar Tuningparameter identifiziert; allerdings liegt hier keine für Laufzeit-Tuning nutzbare Periodizität vor.

Series berechnet mithilfe eines Auftraggeber-Auftragnehmer-Musters eine Folge von Fourierkoeffizienten, die als Stromelemente fungieren. Für die Evaluation des Laufzeit-Tunings wurde die Eingabegröße von Series erhöht, um die Gesamtlaufzeit des Programms auf eine für den Auto-Tuner hinreichende Länge zu vergrößern. Die restlichen Programme weisen keine für Laufzeit-Tuning geeigneten Tuningparameter auf, so dass die Verwendung des Auto-Tuners effektlos wäre.

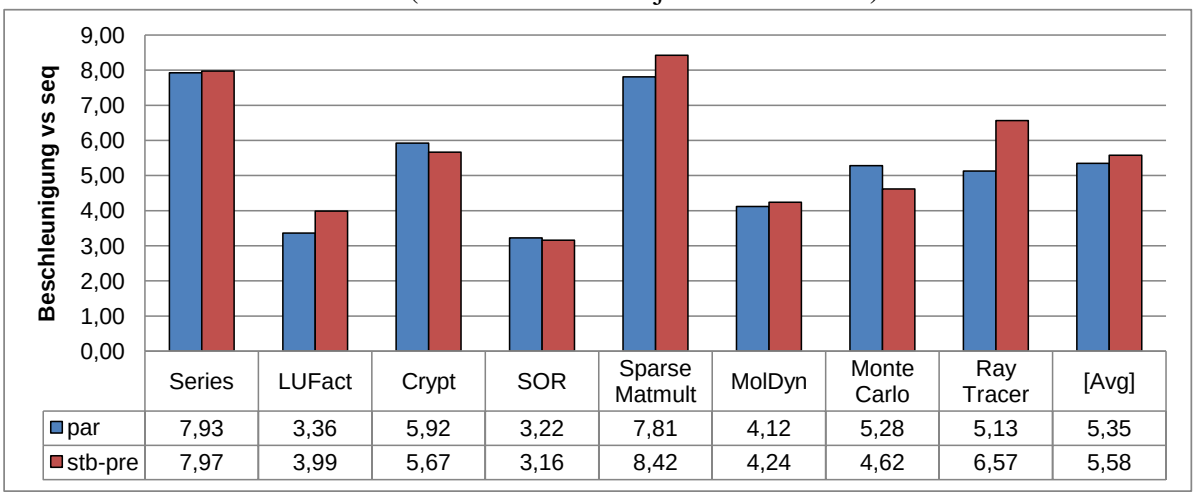
Series (mit Laufzeit-Tuning). Auf der Niagara entspricht die Beschleunigung der vorkonfigurierten Version *stb-pre* der Beschleunigung bei Anwendung der besten bekannten Parameterkonfiguration. Auf dem Vierkern- und Doppelvierkernrechner liegt die Beschleunigung leicht darunter und eröffnet Möglichkeiten für Laufzeit-Tuning. Der Laufzeit-Tuning-Gewinn *OTPG* liegt bei 4% auf dem Vierkern- bzw. 7% auf dem Doppelvierkernrechner, was dem oberen Bereich des durch *stb-best* begrenzten Tuning-Potenzials entspricht.

Q6600 (4 Kerne)



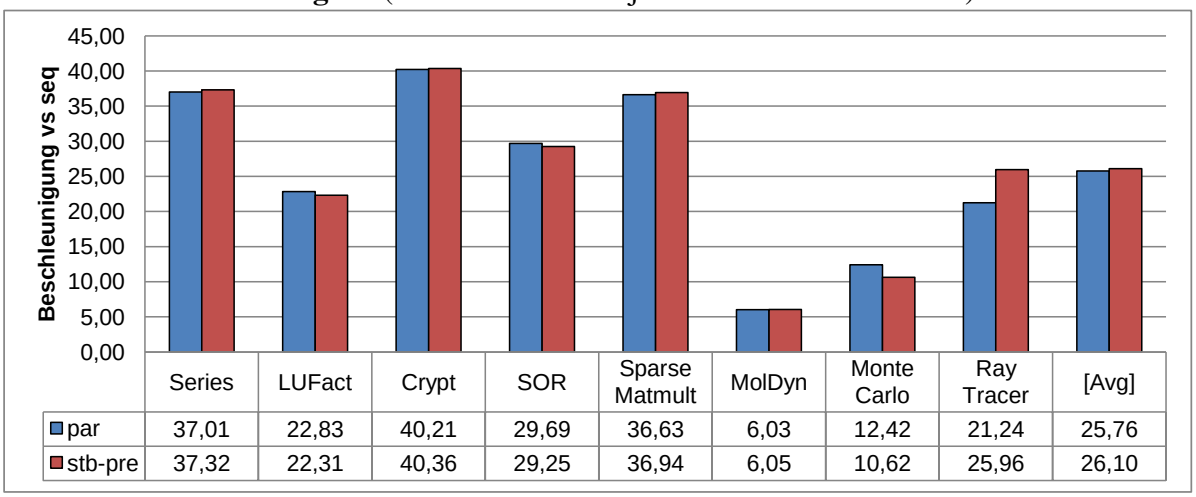
	Series	LUFact	Crypt	SOR	Sparse Matmult	MolDyn	Monte Carlo	Ray Tracer	[Avg]
par	3,67	3,91	3,92	1,59	4,12	2,93	2,66	0,43	2,90
stb-pre	3,98	3,92	3,88	1,59	3,72	2,97	2,45	0,42	2,87

E5320 (2 Prozessoren mit jeweils 4 Kernen)



	Series	LUFact	Crypt	SOR	Sparse Matmult	MolDyn	Monte Carlo	Ray Tracer	[Avg]
par	7,93	3,36	5,92	3,22	7,81	4,12	5,28	5,13	5,35
stb-pre	7,97	3,99	5,67	3,16	8,42	4,24	4,62	6,57	5,58

T2 Niagara (8 Prozessoren mit jeweils 8 Hardware-Fäden)



	Series	LUFact	Crypt	SOR	Sparse Matmult	MolDyn	Monte Carlo	Ray Tracer	[Avg]
par	37,01	22,83	40,21	29,69	36,63	6,03	12,42	21,24	25,76
stb-pre	37,32	22,31	40,36	29,25	36,94	6,05	10,62	25,96	26,10

Abbildung 7.11: Beschleunigungen der Programme der Java Grande Benchmark Suite gegenüber den sequenziellen Implementierungen

JG (ohne Laufzeit-Tuning). Die restlichen Programme der Java Grande Benchmark Suite eignen sich zwar nicht für Laufzeit-Tuning; die Leistung der vorkonfigurierten Varianten *stb-pre* liegt, wie in Abbildung 7.11 zusammengefasst, dennoch auf demselben Niveau wie die Leistung der fadenbasierten Originalprogramme (*par*). Das Programm Series ist hier der Vollständigkeit halber nochmals aufgeführt, diesmal jedoch mit der ursprünglichen Eingabegröße, welche zu geringfügig anderen Beschleunigungen führt. Es kann festgehalten werden, dass die strombasierten Implementierungen gegenüber den Originalprogrammen keinen Mehraufwand aufweisen. Auf dem Doppelvierkernrechner und der Niagara schneiden die strombasierten Programme im Schnitt sogar leicht besser ab als die Originale.

7.2.3.5 Jpeg

Jpeg wurde aus dem StreamIt-Benchmark übernommen; auf dem Vierkern- und Doppelvierkernrechner zeigt sich eine beste Beschleunigung von etwas über 2, auf der Niagara liegt diese lediglich bei etwa $1,4$. In der vorkonfigurierten Version ergeben sich Beschleunigungen, die auf dem Vierkern- und Doppelvierkernrechner leicht unter der jeweils besten Beschleunigung liegen; auf der Niagara ist diese Version sogar langsamer als bei sequenzieller Ausführung. Durch den „Abstand“ zur bestmöglichen Version ergibt sich hier Potenzial für Laufzeit-Tuning. Der Laufzeit-Tuning-Gewinn liegt bei durchschnittlich 28%. Auf dem Vierkernrechner wird durch Laufzeit-Tuning die beste Konfiguration gefunden. Die Leistungen der fadenbasierten Variante (*par*) sind auf dem Vierkern- und Doppelvierkernrechner vergleichbar mit der strombasierten, vorkonfigurierten Variante *stb-pre*, auf der Niagara schneidet *par* leicht schlechter ab als die beste strombasierte Variante *stb-best*.

7.2.3.6 Msort

Msort enthält Tuningparameter für die Fadenanzahl TC und den Ausführungsmodus EM. Der Einsatz von Laufzeit-Tuning ist hierbei wirkungslos, da das Programm keine nutzbare Periodizität aufweist. Dennoch lässt sich die Beschleunigung der vorkonfigurierten Version mit der Beschleunigung bei bester Konfiguration vergleichen. Auf allen drei Rechnern liegen diese Werte sehr nah beieinander; die Leistungsqualität PQ liegt zwischen 96% und 100%. Dieses Beispiel zeigt, dass Filter rekursive Aufgabenparallelität effizient ausnutzen können, selbst wenn kein Datenstrom verarbeitet wird. Auf allen Rechnern liegt die Leistung der fadenbasierten Variante *par* leicht niedriger als die Leistungen der stromorientierten Varianten *stb-pre* und *stb-best*.

7.2.3.7 Vscale und Vzoom

Vscale und Vzoom weisen gute Skalierungseigenschaften auf. Die durch Vorkonfiguration erreichten Beschleunigungen liegen für Vscale nur leicht unter der besten Beschleunigung. Durch Laufzeit-Tuning sind auf der Niagara lediglich vernachlässigbare Verbesserungen möglich. Etwas größer ist das Tuningpotenzial auf dem Doppelvierkernrechner; insgesamt liegt die Leistungsqualität bei Laufzeit-Tuning auf allen Rechnern bei 100%. Der Laufzeit-Tuning-Gewinn $OTPG$ beträgt auf dem Doppelvierkernrechner etwa 11%.

Aus Tuningsicht interessanter ist das Programm Vzoom. Dessen Fließband besitzt unter anderem eine Stufe, welche alternative Filter zum Skalieren eines Videobildes definiert. Da die beste Alternative statisch nicht zu bestimmen ist, werden in der Version *stb-pre* die alternativen Filter wiederholt reihum ausgewählt. Hierbei ergeben sich Beschleunigungen von etwa $0,9$ auf dem Vierkern- und $1,5$ auf dem Doppelvierkernrechner sowie $11,4$ auf der Niagara. Die Leistungsqualität beträgt auf allen Rechnern 33% bis 35%, also etwa nur ein Drittel der bestmöglichen Beschleunigung. Der Einsatz von Laufzeit-Tuning erweist sich hier am wertvollsten: der Laufzeit-Tuning-Gewinn liegt bei etwa 150% gegenüber der ausschließlichen Anwendung der Vorkonfiguration. Die durch Laufzeit-Tuning erreichten Beschleunigungen steigert sich somit auf etwa $2,3$ für den Vierkern- und $3,8$ für den Doppelvierkernrechner sowie auf $28,6$ für die Niagara. Gemessen an der bestmöglichen Parameterkonfiguration ergeben sich bei Laufzeit-Tuning Leistungsqualitätswerte zwischen 84% und 89%. Dieses Beispiel zeigt den besonderen Nutzen von Laufzeit-Tuning im Zusammenhang mit der Auswahl von Alternativen.

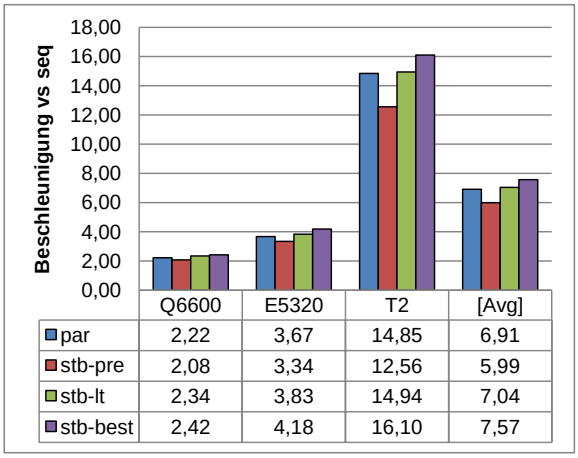
Für Vscale zeigt die strombasierte, vorkonfigurierte Variante *stb-pre* eine etwas bessere Leistung als die fadenbasierte, von Hand optimierte Variante *par*. Die Leistung von Vzoom ist in der Variante *par* dagegen deutlich besser als *stb-pre* und leicht besser als *stb-lt*. Dies ist dadurch zu erklären, dass in der Variante *par* bereits vorab hinsichtlich der schnellsten Alternative optimiert wurde. In der strombasierten Implementierung erfolgt die Auswahl der Alternative dagegen durch den Auto-Tuner, wobei die Alternativen erst im Produktivbetrieb vermessen werden.

7.2.3.8 Fazit

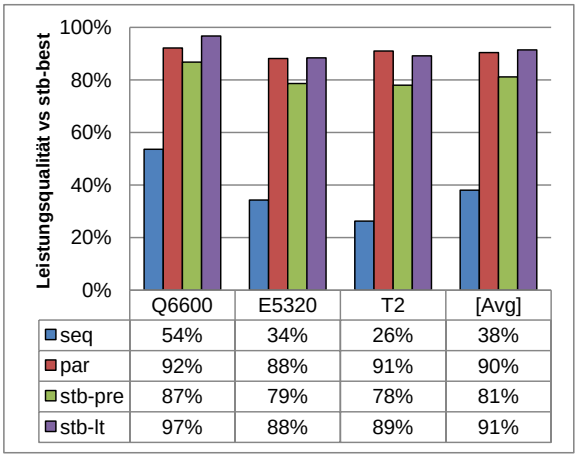
Es wurden verschiedene Applikationen aus unterschiedlichen Anwendungsgebieten, unterschiedlicher Größe und Parallelitätsstruktur betrachtet. Der Übersetzer ist in der Lage, für alle diese Applikation wichtige Tuningparameter und -informationen ohne das Einwirken des Programmierers zu extrahieren. Die so ermöglichte Vorkonfiguration der Tuningparameter führt in vielen Fällen bereits zu guten Beschleunigungen.

Die Laufzeit-Tuning-Gewinne $OTPG$ (vgl. Abbildung 7.10) fallen für die Anwendungen Vzoom (durchschnittlich 151%) und Jpeg (durchschnittlich 28%) am höchsten aus. Für Se-

ries und Vscale zeigen sich Verbesserungen um bis zu 11%, wobei die Vorkonfigurationen bereits zu sehr guten Programmleistungen führen, so dass der Spielraum für Laufzeit-Tuning begrenzt ist. Die Anwendungen DS und Electric erfahren keine Leistungssteigerungen durch Laufzeit-Tuning, andererseits auch keine nennenswerten Verschlechterungen. Im Durchschnitt über alle Anwendungen zeigen sich auf allen Rechnern Verbesserungen von 24% bis 30%. Der Laufzeit-Tuning-Gewinn steigt dabei leicht mit dem Parallelitätsgrad des Rechners.



	Q6600	E5320	T2	[Avg]
par	2,22	3,67	14,85	6,91
stb-pre	2,08	3,34	12,56	5,99
stb-lt	2,34	3,83	14,94	7,04
stb-best	2,42	4,18	16,10	7,57



	Q6600	E5320	T2	[Avg]
seq	54%	34%	26%	38%
par	92%	88%	91%	90%
stb-pre	87%	79%	78%	81%
stb-lt	97%	88%	89%	91%

Abbildung 7.12: Durchschnittliche Beschleunigungen *S* und Leistungsqualitäten *PQ*

Abbildung 7.12 fasst die durchschnittlich erreichten Beschleunigungen *S* und Leistungsqualitäten *PQ* für die jeweiligen Rechner zusammen und stellt die Ergebnisse bei Vorkonfiguration (*stb-pre*), Laufzeit-Tuning (*stb-lt*) und beste Konfiguration (*stb-best*) in Bezug. Im Durchschnitt erreicht die Vorkonfiguration 81%, zusätzliches Laufzeit-Tuning 91% der besten bekannten Laufzeit.

Daraus lässt sich ableiten, dass die auf Kontextwissen basierte Vorkonfiguration der Tuningparameter ein geeignetes Mittel sein kann, um ohne Tuning-Läufe bereits gute Leistungen zu erzielen. Anschließendes Laufzeit-Tuning kann von der Vorkonfiguration profitieren, da hierdurch oftmals schon ein vielversprechender Teil des Suchraums lokalisiert ist.

Vergleich mit Java

Die Performanzmessungen zeigen, dass die strombasierten Programme die Leistungen der fadenbasierten, von Hand optimierten Implementierungen erreichen, teilweise sogar leicht übersteigen. Anhand Abbildung 7.12 lässt sich feststellen, dass die fadenbasierten Programme (*par*) bezüglich der durchschnittlichen Performanz

- etwas besser abschneiden als vorkonfigurierte Stromprogramme (*stb-pre*),
- in etwa gleich abschneiden wie Stromprogramme bei Verwendung von Laufzeit-Tuning (*stb-lt*) und
- etwas schlechter abschneiden als Stromprogramme in der besten bekannten Konfiguration (*stb-best*).

Da die Optimierung eines Stromprogramms automatisch und zur Laufzeit erfolgt, entsteht diesbezüglich für den Entwickler kein Mehraufwand. Um dieselbe Leistung mit fadenbasierter Programmierung und manueller Optimierung zu erreichen, müsste der Entwickler die leistungsrelevanten „Stellschrauben“ selbst identifizieren und – möglicherweise für jede Rechnerarchitektur – im Rahmen von zahlreichen Testläufen konfigurieren, was mit einem enormen Aufwand verbunden wäre. Dieser Vergleich unterstreicht also den Nutzen und die Wirksamkeit von Laufzeit-Tuning.

7.3 Erfüllung der Thesen

In diesem Abschnitt setzen wir die Ergebnisse dieses Kapitels mit den Thesen aus Kapitel 1 in Bezug.

- **Zu These 1.** Es wurde gezeigt, dass stromorientierte Programmierung nicht nur für spezielle Probleme der Grafik- und Signalverarbeitung, sondern auch zur Entwicklung allgemeiner paralleler Anwendungen für Multikernrechner geeignet ist. Die Erweiterung objektorientierter Sprachen um strombasierte Konzepte ermöglicht es, diverse Arten von parallelen Mustern kompakt zu implementieren. Gegenüber der Verwendung expliziter Ausführungsfäden ist der stromorientierte Code kürzer und bietet weniger Fehlerpotenzial.
- **Zu These 2.** Viele Anwendungen enthalten zustandsbehaftete Filter, in einigen Anwendungen greifen Filter auf gemeinsame Variablen zu. Zustände müssen bei der Ausführung berücksichtigt und Zugriffe auf gemeinsame Variablen geschützt werden. Die Zustands- und Konfliktanalyse können zur Korrektheit eines objektorientierten Stromprogramms beitragen. Für die untersuchten Anwendungen wurden Filterzustände bis auf wenige Ausnahmen korrekt identifiziert und alle kritischen Variablenzugriffe bei nur wenigen Falschmeldungen erkannt.
- **Zu These 3.** Die übersetzerbasierte Identifikation von Tuningparametern hat es ermöglicht, ohne Einwirken des Programmierers optimierbaren Code zu erzeugen, der neben Tuningparametern Kontext- und Tuninginformationen enthält. Die heuristikbasierte Vorkonfiguration der Parameter führte in vielen Fällen bereits zu guten Ausführungszeiten.

- **Zu These 4.** Es wurde gezeigt, dass der Einsatz von Laufzeit-Tuning die Performanz von Stromprogrammen im Produktivbetrieb weiter steigern kann. Die in der Arbeit entwickelten Konzepte zur Leistungsbestimmung und -optimierung haben sich als geeignet erwiesen, um Auto-Tuning als integralen Bestandteil des stromorientierten Programmiermodells zu behandeln.

7.4 Zusammenfassung

In diesem Kapitel wurden die Lösungskonzepte der vorliegenden Arbeit anhand von Fallstudien evaluiert. Untersucht wurden der Implementierungsaufwand objektorientierter Stromprogramme, die Wirksamkeit der Zustands- und Konfliktanalyse, die Überführung von Stromprogrammen in optimierbare Form sowie deren Leistungsbestimmung und -optimierung.

Die Ergebnisse belegen die in Kapitel 1 aufgestellten Thesen. Die gewonnenen Erkenntnisse unterstreichen den Nutzen stromorientierter Programmierkonzepte im Kontext allgemeiner Anwendungen für Multikernsysteme.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde das Gesamtkonzept der objektorientierten Stromprogrammierung vorgestellt. Die in diesem Zusammenhang erarbeiteten Konzepte liefern einen Beitrag auf dem Gebiet der Programmiermodelle für Multikernsysteme. In diesem Kapitel fassen wir die Konzepte und Ergebnisse zusammen und geben einen Ausblick auf Forschungsfragen, die sich an diese Arbeit anschließen.

8.1 Zusammenfassung

In einem ersten Schritt wurde ein Paradigma definiert und in Form einer Spracherweiterung umgesetzt, welches die Verwendung von Datenströmen, Filtern und Stromgraphen in objektorientierten Sprachen erlaubt. Mit diesen Konstrukten zur stromorientierten Programmierung ist es möglich, diverse Klassen paralleler Muster kompakt zu implementieren und dabei von Ausführungsfäden und expliziter Synchronisierung zu abstrahieren.

Um die Nutzbarkeit dieser Sprachkonzepte auch im objektorientierten Kontext zu gewährleisten, wurden Analysen entwickelt, welche zustandsbehaftete Filter und kritische Abschnitte beim Übersetzen identifizieren können. Diese Informationen sind für die korrekte parallele Ausführung eines Programms von besonderer Wichtigkeit.

Es wurde eine Methodik entwickelt, die wichtige leistungsrelevante Stellschrauben, sogenannte Tuningparameter, in objektorientierten Stromprogrammen automatisch identifiziert. Hierbei nutzt der Übersetzer das in der stromorientierten Darstellung inhärente Wissen über die parallele Struktur eines Programms. Ein Stromprogramm kann somit in eine geeignete optimierbare Form überführt und mithilfe von Tuningheuristiken vorkonfiguriert werden.

Schließlich wurde ein Konzept vorgestellt, welches die Optimierung von Stromprogrammen im Produktivbetrieb erlaubt. Hierfür wurde ein Mechanismus zur Leistungsbestimmung entwickelt, der von einem Auto-Tuner zur Anpassung von Tuningparametern verwendet werden kann.

Die Ergebnisse aus den Fallstudien haben die in dieser Arbeit aufgestellten Thesen belegt. Es wurden Applikation und Programme unterschiedlicher Größe und aus verschiedenen Anwendungsdomänen stromorientiert implementiert. Im Vergleich zur nicht stromorientierten Implementierung wurde der Code kürzer und wies weniger gemeinsame Variablen und kritische Abschnitte auf, woraus sich ein geringeres Fehlerrisiko ableiten lässt. Die Programmanalysen konnten zur Korrektheit der Programme beitragen, indem Filterzustände bis auf wenige Ausnahmen erkannt sowie alle kritischen Abschnitte bei geringer Falsch-Positiv-Rate lokalisiert werden konnten. Die Methodik zur automatischen Identifikation und heuristikbasierten Vorkonfiguration von Tuningparametern erzielte ebenfalls vielversprechende Resultate. Es konnten Laufzeiten gemessen werden, die im Schnitt 81% der Leistung bei bester Parameterkonfiguration erreichten. Durch den Einsatz von Laufzeit-Tuning wurde dieser Wert weiter auf durchschnittlich 91% verbessert, wobei ein Mehraufwand für Laufzeit-Tuning nicht festgestellt werden konnte. Diese Erkenntnisse unterstreichen die Praktikabilität des sprachintegrierten Laufzeit-Tunings, das zudem keinerlei Interaktion des Entwicklers erfordert.

8.2 Ausblick

Die Stromverarbeitung besitzt großes Potential für die Entwicklung und automatische Optimierung paralleler Anwendungen. Die Erkenntnisse der vorliegenden Arbeit führen zu weiteren Forschungsfragen, die im Folgenden skizziert werden.

Kontrollierte Experimente könnten Aufschluss über die Produktivität von Programmierern bei Verwendung von Stromsprachen im Vergleich zu anderen Programmierparadigmen bzw. -sprachen geben. Mithilfe empirischer Methoden ließe sich etwa der zeitliche Aufwand für Implementierung, Fehlerfindung und Performanzoptimierung oder die Codequalität bestimmen und vergleichen.

Ein zentrales Forschungsgebiet stellen die Programmanalysen dar, die entweder wie in der vorliegenden Arbeit übersetzerintegriert oder als eigenständiges Werkzeug zur Anwendung kommen. Insbesondere im Kontext der parallelen Programmierung zeigt sich deren Nutzen, um Fehlerquellen möglichst früh zu identifizieren sowie manuell oder automatisch zu beheben. Hier gilt es, ein Mittelmaß zwischen Aufwand und Genauigkeit der Analysen zu finden. Im Zusammenhang mit Spracherweiterungen ergeben sich Möglichkeiten, die Genauigkeit durch Ausnutzen von Übersetzerwissen zu erhöhen.

Die Integration von Methoden des Auto-Tunings in Übersetzer bzw. Laufzeitsysteme birgt großes Forschungspotenzial. Die übersetzergestützte Identifikation und Vorkonfiguration von Tuningparametern hat sich in dieser Arbeit als vielversprechend erwiesen. Daher liegt es nahe, zu untersuchen, inwieweit sich die vorgestellten Mechanismen auf andere Programmiersprachen bzw. Sprachkonzepte übertragen bzw. verallgemeinern lassen. Hierbei könnten statt der

Laufzeit auch andere Leistungskriterien wie der Energieverbrauch Gegenstand der Optimierungsfunktion sein.

In diesem Zusammenhang sollte die Kombination von modellbasiertem und suchbasiertem Auto-Tuning untersucht werden. Insbesondere beim Laufzeit-Tuning ist das schnelle Finden einer guten Parameterkonfiguration von großer Wichtigkeit. In dieser Arbeit wurde durch das Konzept der kontextbasierten Vorkonfiguration von Tuningparametern ein Schritt in diese Richtung unternommen. Geeignete Performanzmodelle für einzelne Tuningparameter könnten die Qualität und Geschwindigkeit der Optimierung erhöhen. Mit entsprechenden Modellen ließe sich der Suchraum genauer eingrenzen oder gute Parameterkonfigurationen vorhersagen.

Großes Potenzial liegt im anwendungsübergreifenden Laufzeit-Tuning. Insbesondere Multikernsysteme sind oft mit Szenarien konfrontiert, in denen mehrere Anwendungen gleichzeitig ausgeführt werden, die um gemeinsame Ressourcen konkurrieren. Wird die Optimierung einer Anwendung als isoliertes Problem begriffen, leidet möglicherweise die Gesamtleistung aller Anwendungen. Die kooperative Optimierung der Anwendungen kann diesem Problem entgegenwirken und sollte daher erforscht werden.

Literaturverzeichnis

[1] Electric VLSI design system. http://staticfreesoft.com/. Last retrieved March 2012.

[2] Java CUP website. http://www2.cs.tum.edu/projects/cup/. Last retrieved April 2012.

[3] JFlex website. http://www.jflex.de/. Last retrieved April 2012.

[4] Polyglot website. http://www.cs.cornell.edu/projects/polyglot/. Last retrieved April 2012.

[5] Soot website. http://www.sable.mcgill.ca/soot/. Last retrieved April 2012.

[6] Yehuda Afek, Guy Korland, and Arie Zilberstein. Lowering STM overhead with static analysis. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 31–45, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with FastFlow. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 170–181. Springer Berlin / Heidelberg, 2011.

[8] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*. Wiley, 2011.

[9] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, September 2009.

[10] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification 1.0, April 2008.

[11] Jason Ansel and Cy Chan. PetaBricks. *XRDS*, 17(1):32–37, September 2010.

[12] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[13] Jason Ansel, Yee Lok Won, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *The International Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.

[14] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, http://citeseer.ist.psu.edu/393979.html, 1996.

[15] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[16] Shivnath Babu. Towards automatic optimization of MapReduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[18] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[20] Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53, Washington, DC, USA, 2008. IEEE Computer Society.

[21] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer Berlin/Heidelberg, 2007.

[22] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, September 2008.

[23] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[24] R. Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[25] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[26] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[27] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.

[28] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[29] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[30] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: A DSL approach to specifying streaming applications. In Frank Pfenning and Yannis Smaragdakis, editors, *GPCE*, volume 2830 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2003.

[31] Marco Danelutto and P. Teti. Lithium: A structured parallel programming environment in Java. In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science*, volume 2330 of *Lecture Notes in Computer Science*, pages 844–853. Springer, 2002.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[33] Matthias Dempe. Automatisierte Optimierung von Aufgaben- und Fließbandparallelität. Master's thesis, Karlsruher Institut für Technologie (KIT), 2010.

[34] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[35] Johannes Franz. StreamCpp: Stromprogrammierung für C++ auf Mehrkern-Architekturen. Master's thesis, Karlsruher Institut für Technologie, October 2010.

[36] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994.

[38] U. Gleim and T. Schüle. *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C#.* dpunkt, 2011.

[39] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, October 2002.

[40] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, 2010.

[41] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[42] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[43] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[44] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[45] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[46] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

[47] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceeding of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[48] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: efficient realization of streaming applications on FPGAs. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 41–50, New York, NY, USA, 2008. ACM.

[49] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, Washington, DC, USA, 2009. IEEE Computer Society.

[50] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. Liquid Metal: Object-oriented programming across the hardware/software boundary. In *ECOOP*, pages 76–103, 2008.

[51] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland Publishing Company, Amsterdam, Aug 1974.

[52] Thomas Karcher and Victor Pankratius. Run-time automatic performance tuning for multicore applications. In *Proceedings of the 17th International Euro-Par Conference on Parallel Processing*. Euro-Par 2011, September 2011.

[53] Takahiro Katagiri, Kenji Kise, Hiroaki Honda, and Toshitsugu Yuba. Fiber: A generalized framework for auto-tuning software. In *ISHPC*, pages 146–159, 2003.

[54] Guy Korland, Pascal Felber, and Nir Shavit. Deuce: Noninvasive concurrency with a Java STM. In *MultiProg '10: Programmability Issues for Multi-Core Computers*, page 10 pages, 2010.

[55] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot(TM) client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[56] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, New York, NY, USA, 2008. ACM.

[57] Doug Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[58] E. Lee and D. Messerschmitt. Pipeline interleaved programmable DSPs: Synchronous data flow programming. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 35(9):1334 – 1345, September 1987.

[59] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, September 1987.

[60] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 227–242, New York, NY, USA, 2009. ACM.

[61] Andreas Leppert. Ein heuristischer Optimierungsalgorithmus für Laufzeit-Tuning. Master's thesis, Karlsruher Institut für Technologie (KIT), 2011.

[62] Mario Leyton and Jose M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 289–296, 2010.

[63] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[64] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[65] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22:896–907, July 2003.

[66] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for parallel programming*. Addison-Wesley Boston, 2005.

[67] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–358, New York, NY, USA, 2006. ACM.

[68] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 787–795, New York, NY, USA, 2004. ACM.

[69] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[70] David J. Meder and Walter F. Tichy. Parallelizing an index generator for desktop search. Technical Report 2010-9, IPD, University of Karlsruhe, Germany, May 2010.

[71] Harm Munk, Eduard Ayguadé, Cédric Bastoul, Paul Carpenter, Zbigniew Chamski, Albert Cohen, Marco Cornero, Philippe Dumont, Marc Duranton, Mohammed Fellahi, Roger Ferrer, Razya Ladelsky, Menno Lindwer, Xavier Martorell, Cupertino Miranda, Dorit Nuzman, Andrea Ornstein, Antoniu Pop, Sebastian Pop, Louis-Noël Pouchet, Alex Ramírez, David Ródenas, Erven Rohou, Ira Rosen, Uzi Shvadron, Konrad Trifunović, and Ayal Zaks. ACOTES project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, 38, April 2010.

[72] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[73] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[74] J.R. Newport. An introduction to occam and the development of parallel software. *Software Engineering Journal*, 1(4):165 –169, July 1986.

[75] Ryan Newton, Lewis Girod, Michael Craig, Sam Madden, and Greg Morrisett. WaveScript: A case-study in applying a distributed stream-processing language. Technical Report MIT-CSAIL-TR-2008-005, MIT, Jan 2008.

[76] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.

[77] Martin Odersky. *Scala by Example*. École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, October 2009. (Draft).

[78] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[79] Frank Otto. Analyse von Java-Programmen auf Synchronisierungsfehler. Diplomarbeit, Universität Karlsruhe (TH), 2007.

[80] Frank Otto and Thomas Moschny. Finding synchronization defects in Java programs: extended static analyses and code patterns. In *IWMSE '08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 41–46, New York, NY, USA, 2008. ACM.

[81] Frank Otto, Victor Pankratius, and Walter F. Tichy. High-level multicore programming with XJava. In *31st International Conference on Software Engineering (ICSE), New Ideas and Emerging Results*, May 2009.

[82] Frank Otto, Victor Pankratius, and Walter F. Tichy. XJava: Exploiting parallelism with object-oriented stream programming. In H.-X. Lin H. Sips, D. Epema, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 875–886. Springer, 2009.

[83] Frank Otto, Christoph A. Schaefer, Matthias Dempe, and Walter F. Tichy. A language-based tuning mechanism for task and pipeline parallelism. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, pages 328–340, September 2010.

[84] Victor Pankratius, Ali Jannesari, and Walter F. Tichy. Parallelizing BZip2. A case study in multicore software engineering. Technical Report, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, Germany, April 2008.

[85] Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 53–60, New York, NY, USA, 2008. ACM.

[86] Michael Philippsen. A survey on concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, 2000.

[87] K. Randall. *Cilk: Efficient Multithreaded Computing*. Dep. EECS, MIT, 1998.

[88] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. *hpca*, 0:13–24, 2007.

[89] Thomas Rauber and Gudula Rünger. Parallel programming models. In *Parallel Programming*, pages 93–149. Springer Berlin Heidelberg, 2010.

[90] James Reinders. Intel threading building blocks tutorial revision 1.6, Apr 2007.

[91] Christoph A. Schaefer. Reducing search space of auto-tuners using parallel patterns. In *IWMSE '09: Proceedings of the 2nd International Workshop on Multicore Software Engineering*, May 2009.

[92] Christoph A. Schaefer. *Automatisierte Performanzoptimierung Paralleler Architekturen*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2010.

[93] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In Springer Berlin / Heidelberg, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume LNCS, pages 9–20, Aug 2009.

[94] Christoph A. Schaefer, Victor Prankratius, and Walter F. Tichy. Engineering parallel applications with tunable architectures. In *International Conference on Software Engineering (ICSE)*, May 2010.

[95] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2001. ACM Press.

[96] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: high-throughput stream programming in Java. *SIGPLAN Not.*, 42(10):211–228, 2007.

[97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[98] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):292–210, March 2005.

[99] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.

[100] Vahid Tabatabaee, Ananta Tiwari, and Jeffrey K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[101] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second International Workshop on MapReduce and its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[102] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[103] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 325–335, New York, NY, USA, 2006. ACM.

[104] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[105] Jonas Thedering. Online-Auto-Tuning für stromorientierte Programmiersprachen. Master's thesis, Karlsruher Institut für Technologie (KIT), 2011.

[106] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[107] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2002.

[108] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[109] Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan, and Jacqueline Chame. Auto-tuning full applications: A case study. *Int. J. High Perform. Comput. Appl.*, 25(3):286–294, August 2011.

[110] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Computing*, 35(8–9):475–492, 2009.

[111] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Automatic atomic region identification in shared memory SPMD programs. In *Proceedings of the ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '10, pages 652–670, New York, NY, USA, 2010. ACM.

[112] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. *SIGPLAN Not.*, 38(5):115–128, 2003.

[113] Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multicores: a machine learning based approach. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 307–318, New York, NY, USA, 2010. ACM.