Nils Potthoff

Dissertation

Globale Abhängigkeitsanalyse und Visualisierung zur Parallelisierung sequentieller Programmsysteme

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Fakultät für Elektrotechnik,
Informationstechnik
und Medientechnik

BERGISCHE
UNIVERSITÄT
WUPPERTAL

Referent:	Prof. Dr.-Ing. Dietmar Tutsch
Korreferenten:	Prof. Dr.-Ing. Dieter Brückmann
	PD Dr.-Ing. habil. Carsten Gremzow
Tag der mündlichen Prüfung:	2016-05-06

Die Dissertation kann wie folgt zitiert werden:

urn:nbn:de:hbz:468-20160621-102956-7
[http://nbn-resolving.de/urn/resolver.pl?urn=urn%3Anbn%3Ade%3Ahbz%3A468-20160621-102956-7]

Kurzfassung

Die aktuelle Entwicklung von Prozessoren hat zu einer weiten Verbreitung von Multi- und Many-Core-Prozessoren geführt. Demgegenüber steht nach wie vor das etablierte sequentielle Programmierparadigma. Eine effiziente Entwicklung mehrerer parallel arbeitender Instanzen erfordert in der Regel die explizite Programmierung von Kommunikations- und Synchronisierungsaufgaben. Durch das zusätzlich benötigte Wissen über die Zielarchitektur und parallele Algorithmen erreicht Softwareentwicklung eine höhere Komplexität. Dies geht einher mit einem erhöhten Fehlerrisiko. Zudem werden Reimplementierungen etablierter Programme notwendig. Um den Entwicklungsaufwand gering zu halten und zugleich von Mehrkernarchitekturen profitieren zu können, verfolgt diese Forschungsarbeit einen Ansatz zur rechnergestützten Parallelisierung sequentieller Programme mittels eines Zerlegungsmodells. Fragmentierungsansätze wurden herangezogen und in Hinblick auf das Ziel adaptiert, um deren Eignung für die Parallelisierung und mögliche Strategien zu prüfen.

Das für weitere Schritte gewählte Modell führt zu Fragmenten, bei denen alle Kontroll- und Datenflüsse bekannt sein müssen. Konkrete Datenabhängigkeiten sind für die Ausführungsdauer einzelner Fragmente zudem statisch. Dadurch wird eine Umordnung in der Ausführung möglich. In der Arbeit werden Parallelisierungsstrategien auf Basis ermittelter Kontroll- und Datenflussabhängigkeiten zwischen gebildeten Fragmenten eines Programmsystems präsentiert. Um Entwicklern einen Eindruck über existierende Abhängigkeiten vermitteln zu können, ist Visualisierung ein geeignetes Mittel. Diese wird für reale Anwendungen schnell komplex. Die besonderen Eigenschaften des zugrundeliegenden Modells erlauben einen Vergleich mit digitalen bzw. kombinatorischen Schaltungen. Es wird der daraus abgeleitete Visualisierungsansatz präsentiert. Darüber hinaus ergibt sich zudem eine mögliche Transformation in Schaltungen auf Systemebene, welche diskutiert wird.

Für eine Umordnung der Ausführung zur Nutzung der Parallelität wurde ein Ausführungsmodell entwickelt. Dieses basiert auf einer Ablaufplanung, welche durch Kapselung der Fragmente und Abflachung der Aufrufhierarchie möglich wird. Das Verhalten dieses Modells wird hinsichtlich parallelisierbarer Strukturen diskutiert. Konkrete Ausprägungen der theoretisch entwickelten Strategien werden anhand von Beispielen belegt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis verwendeter Abkürzungen und Symbole

ϕ	ϕ-Funktion in der statischen Einzelzuweisungsform
Ψ	Position eines Function-Slices im Function-Slice-Graph
a	Zuweisungsoperation (assignment)
e	Kante (edge)
G	Graph
j	Sprungbefehl (jump)
l	Blatt (leaf)
p	Weg (path)
T	Baum (tree)
v	Knoten (vertex)
ALU	Arithmetic Logic Unit
B	Block
BB	Basisblock
CDFG	Kontroll- und Datenflussgraph, engl. control and data flow graph
CF	Kontrollfluss (control flow)
CFG	Kontrollflussgraph, engl. control flow graph
CISC	Complex Instruction Set Computer

CPU	Central Processing Unit
d	Variable wird definiert (Wertzuweisung) (Variable)
DF	Datenfluss (data flow)
DFG	Datenflussgraph, engl. data flow graph
DSWP	Decoupled Software Pipelining
F	Funktion
FPGA	Field Programmable Gate Array
FS	Function Slice
FSC	Function-Slice-Cluster
HDL	Hardwarebeschreibungssprache (Hardware Description Language)
I	Instruktion (instruction)
ILP	Befehlsebenenparallelität (Instruction-level parallelism)
LLVM	LLVM Compiler-Infrastruktur
MMU	Memory Management Unit
MPI	Message Passing Interface
NLP	Nested Loop Program
r	Variable wird referenziert (Lesezugriff)
RISC	Reduced Instruction Set Computer
RMS	engl. root mean square, quadratisches Mittel
S	Anweisung (statement)
SB	Sub-Block
SDG	System-Dependence-Graph
SIMD	Single Instruction-Stream, Multiple Data-Stream
SSA	Static Single Assignment
TLP	Threadebenen-Parallelität (Thread Level Parallelism)
U	Programmeinheit, Befehlsfolge
u	Variable wird undefiniert/zerstört

u	undefiniert (Variable)
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Kapitel 1

Einleitung

„Ich denke, dass es weltweit einen Markt für vielleicht fünf Computer gibt“

Thomas Watson, Chairman von IBM 1943

1938 arbeitet Konrad Zuse an seiner programmierbaren mechanischen Rechenmaschine, später bekannt als Z1. Die darauf folgende mit Relais aufgebaute Z3 war dann der erste voll funktionsfähige Computer (1941). Seit dem hat die Entwicklung der Computertechnik eine enorme Entwicklung vollzogen. Inzwischen sind Computer in quasi allen Bereichen des alltäglichen Lebens vertreten und nicht mehr wegzudenken. Von 30 Tonnen Gewicht des ENIAC (1946) sind Computer inzwischen auf Chip-Größe geschrumpft. Sie sind in großer Zahl getarnt als eingebettete Systeme erst auf den zweiten Blick als solche zu erkenne. Gleichzeitig ist die Komplexität der Hardware und damit die erzielte Rechenleistung gestiegen. Dennoch basieren auch heute die meisten Computer auf der Von-Neumann-Architektur. Diese bestehen aus den Komponenten Rechenwerk (Arithemtic Logic Unit), Steuerwerk (Control Unit), Speicherwerk (Memory) und Ein-/Ausgabewerk (I/O). Wobei der Speicher gemeinsam für Befehle und Daten verwendet wird. Die Abarbeitung der Programme erfolgt streng sequentiell, was einen deterministischen Programmablauf zur Folge hat.

Parallel zur Hardwareentwicklung schreitet auch die Softwareentwicklung voran. Wie in Abbildung 1.1 beispielhaft dargestellt, ist mit der gestiegenen Leistungsfähigkeit ist die Komplexität der Programme gestiegen. Die wachsende Bedeutung dieser Technologie treibt die Entwicklung voran. Betätigungsfelder der Forschung decken ein riesiges Feld ab. Gerade der Einzug der Multicore-Prozessoren wirft neue Probleme und Fragestellungen auf, die es zu beantworten gilt. Ein Teil ist die Anpassung der Entwicklungsprozesse, ohne die Programmierer zu überfordern oder die Qualitätsstandards durch neue Fehlerquellen senken zu müssen. Diese Arbeit versucht daher über ein Modell zur Betrachtung von Software einen unterstützenden Beitrag zu leisten.

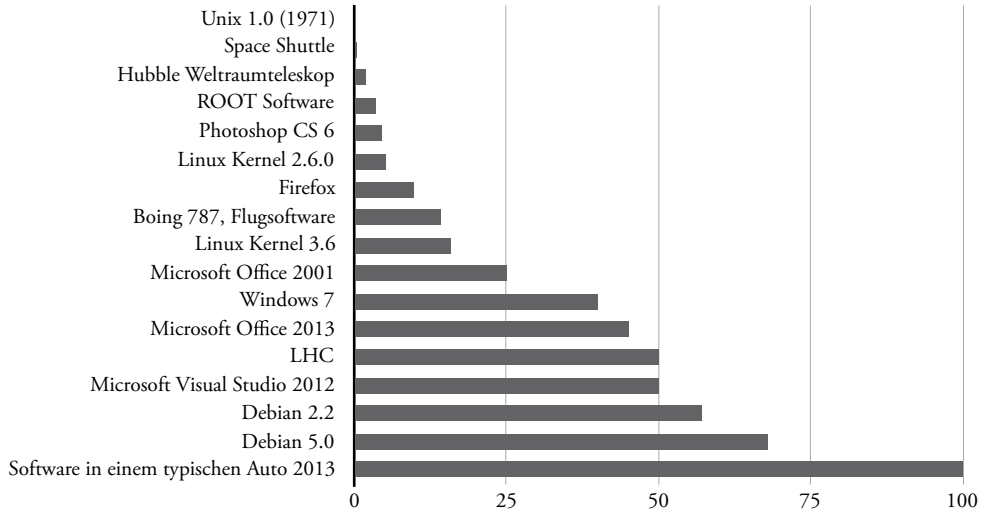


Abbildung 1.1: Anzahl der Codezeilen einiger Codebasen in Millionen (Datenquelle: [KIB15])

1.1 Motivation

Inzwischen sind Mehrkern- oder Multicore-Prozessoren weitverbreitet und ihr Anteil in computerbasierten Systemen nimmt weiterhin zu. Auch die Zahl der Kerne wird in den nächsten Jahren noch steigen. In der Entwicklung befinden sich Prozessoren mit einer Anzahl von Prozessorkernen in der Größenordnung von 100, sogenannte Manycore-Prozessoren. Zukünftige Architekturen könnten zudem vermehrt heterogene Strukturen aufweisen. Die Rechenkerne der Prozessoren wären dann nicht vom gleichen Typ, statt dessen könnten für besondere, wiederkehrende Aufgaben Spezialkomponenten integriert werden.

Durch diese Entwicklungen können höhere Rechenleistungen bei gleichzeitiger Reduzierung des Energieverbrauchs erreicht werden. Um von der Entwicklung der Hardware profitieren zu können, muss allerdings auch die Softwareentwicklung angepasst werden. Die wichtigsten bzw. am weitesten verbreiteten Programmiersprachen bringen nur eine unzulängliche Unterstützung für parallele oder nebenläufige Sequenzen. Dies wird auch durch die Tatsache unterstützt, dass der Entwicklungsprozess sequentieller Software gut bekannt ist und sich bewährt hat.

Um innerhalb einer Software gleichzeitig ausführbare Prozesse zu entwickeln, und die Ausführung effizient zu gestalten, bedarf es des Wissens über die bei der Ausführung zugrunde liegende Architektur sowie Kenntnissen über parallele Algorithmen. In den meisten Fällen ist es notwendig, die Synchronisierung und Koordination der einzelnen Programmteile manuell in der Softwareentwicklung zu codieren. Die Programmierung erreicht dadurch eine höhere Komplexität. Dies geht mit dem erhöhten Risiko von Fehlern einher.

Wird ein Programm in mehrere nebenläufige Teile zergliedert, so müssen diese Teile bei der Ausführung initialisiert und verwaltet werden. Die einzelnen Programmteile müssen in der Regel Daten austauschen, wodurch ein zusätzlicher Kommunikationsaufwand entsteht. Der zusätzliche Aufwand darf den Performancegewinn durch die Nutzung von mehreren Prozessoren oder Prozessorkernen nicht zunichte machen. Eine zu feingranulare Zerlegung kann in der Gesamtheit zu Performanceverlusten führen.

Für die Softwareentwicklung wird es also wichtig sein, Kenntnisse über interne Zusammenhänge zu erlangen. Dies betrifft besonders den Kontroll- und Datenfluss. Es reicht jedoch nicht aus, Techniken zu verwenden die im Compiler-Bau bekannt sind, um diese Informationen zu gewinnen. Besonders Schleifengrenzen oder Zugriffe über Pointer stellen Probleme dar, wenn es um die statische, nur auf dem Quelltext aufbauende, Analysen geht. Sollen alle möglichen Konstrukte, die sequentieller Quelltext aufweisen kann, abgedeckt werden können, ist eine Analyse zur Laufzeit unabdingbar. Das Programm muss also ausgeführt werden, der Ablauf und die Datenzugriffe müssen dabei beobachtet werden.

Auf dem Weg zu parallelen, nebenläufigen Programmen, ist es notwendig mögliche Parallelität zu finden bzw. zu erkennen. Ein Hilfsmittel für Softwareentwickler kann dabei die Visualisierung der Abhängigkeiten sein. Softwarevisualisierung ist ein breites Feld mit einer Vielzahl von Anwendungen. Abhängig von den Anforderungen an die Darstellung reichen die Möglichkeiten von „schön“ formatiertem Code bis hin zu der statischen Darstellung der internen Abhängigkeiten oder der dynamischen Visualisierung der Ausführungsreihenfolge.

Für die Aufteilung in mehrere Programmteile, z. B. zur parallelen Ausführung, ist es notwendig, das ursprüngliche Programm so zu zerlegen, dass die Funktionalität unverändert erhalten bleibt. Interne Zusammenhänge bzw. Abhängigkeiten zwischen einzelnen Bereichen müssen berücksichtigt werden, zuvor also bekannt sein. Erst dann kann eine Transformation für ein Zielsystem erfolgen. Nach Möglichkeit sollte der Ansatz nicht nur eine spezielle Zielarchitektur abdecken, sondern so allgemein konstruiert sein, dass möglichst wenige bis keine Einschränkungen existieren.

1.2 Stand der Technik

Bei der Realisierung von verteilten und nebenläufigigen Programmen ist eine Parallelisierung der Ausführung notwendig. Der erste Schritt besteht darin die durchzuführenden Berechnungen zu zerlegen. Um einzelne Prozessoren gleichmäßig zu nutzen, erfolgt eine Lastverteilung (Scheduling) durch Zuweisung der Teile auf Threads. Die Abbildung von Threads auf physikalische Prozessoren wird Mapping genannt.

Die Parallelisierung kann manuell, halbautomatisch oder automatisch erfolgen. Letzteres stellt das Ziel der in dieser Arbeit präsentierten Methoden dar. Für die manuelle Umsetzung stehen verschiedene Programmierschnittstellen oder Bibliotheken zur Verfügung welche sequentielle Sprachen ergänzen um Parallelität zu spezifizieren. Darüber hinaus existieren spezielle Programmiersprachen wie z. B. StreamIt [TKA02] oder Cilk [BJK+96]. Diese Ansätze bieten jedoch wenig Unterstützung was die korrekte und effek-

tive Parallelisierung angeht. Häufig lassen sich viele Programme auf Grund ihrer Struktur oder der verwendeten Datenstrukturen nicht direkt transformieren. Der Programmierer kann mittels der POSIX-Thread-Bibliothek eine nebenläufige Bearbeitung spezifizieren.

Einen alternativen Ansatz bietet das Message-Passing-Programmiermodell, welches Parallelrechner mit verteiltem Speicher abstrahiert. Jeder Prozess bzw. jede Programminstanz kann auf lokale Daten zugreifen. Der Informationsaustausch untereinander erfolgt über das explizite Versenden von Nachrichten. Als Standard hat sich MPI (Message Passing Interface) etabliert. Es handelt sich dabei um standardisierte Programmschnittstellen zur Realisierung von Kommunikationsoperationen. Die konkrete Realisierung ist Gegenstand der verwendeten Bibliothek und unterliegt nicht dem Standard.

Dem expliziten Austausch von Informationen in Form von Nachrichten gegenüber steht die Verwendung von gemeinsamen Variablen (*shared variables*). Hierzu steht allen Ausführungseinheiten gemeinsamer Speicher zur Verfügung. Jede Einheit hat somit Zugriff auf Variablen des globalen Adressraums. Dies bedingt jedoch Zugriffe zu koordinieren. Ein verbreiteter Standard im Bereich des wissenschaftlichen Rechnens für die Parallelisierung auf der Ebene von Schleifen ist OpenMP. UPC (Unified Parallel C) sei „als Beispiel einer Programmiersprache, die einen verteilten gemeinsamen Speicher auf Sprachebene unterstützt“[RR12] genannt.

Im Umfeld der parallelen Datenverarbeitung haben sich Manycore-GPUs (Graphics Processing Units) hervorgetan. Sie bieten eine kostengünstige und weit verbreitete Möglichkeit spezielle Aufgaben effizient zu bearbeiten. Das spezielle Design ermöglicht einen hohen Durchsatz von Fließkomma-Operationen, und somit einen Einsatz für allgemeine numerische Anwendungen. Die Programmierung erfolgt über Programmierumgebungen wie CUDA oder dem offenen Standard OpenCL (Open Computing Language). Es erfolgt eine Unterteilung eines Programms in einen CPU- und einen GPU-Teil. Ersterer übernimmt alle I/O-Operationen, letzterer ist für die auf der GPU-Architektur durchzuführenden Berechnungen zuständig. [RR12]

Eine Kombination von automatischen und manuellen Techniken liegt dem SUIF-Projekt zugrunde [WFW+94],[HAM+05]. Par4all strebt eine Quelltext-zu-Quelltext-Transformation für heterogene Ziele an. Dabei wird jedoch auf den Umgang mit Pointer-Arithmetik sowie Sprüngen aus Schleifenkörpern und Rekursion verzichtet. In [Ami12], [ACE+12] werden halbautomatische Ansätze beschrieben. Ein weiteres Projekt der Manipulation auf Quelltext-Ebene ist CETUS. Dies ist eine modifizierbare Compiler-Infrastruktur für das Forschungsumfeld. Schnittstellen können für Transformationen genutzt werden. Laufzeitanalysen ermöglichen Statistiken über das Verhalten zu ermitteln. Bezüglich interner Datenabhängigkeiten wird hier der umgekehrte Ansatz verfolgt; es werden unterstützende Tests auf Datenunabhängigkeit durchgeführt. [DBM+09]

In [VRD10] wird ein Compiler-Framework für die semiautomatische Parallelisierung irregulärer C-Programme präsentiert. Dem Programmierer wird hier eine Hilfestellung gegeben, um äußere Schleifen in eine Pipeline-Struktur zu überführen. Das verwendete Programmiermodell basiert auf Annotationen. Das System basiert darauf, dass der Entwickler zusätzliche Semantik an den Übersetzungsprozess weiterreicht, als auch, dass der Compiler derartige zusätzliche Semantik vorschlägt. In [RVD10] wird eine Methode beschrieben bei der der Anwender aus einer Menge möglicher Code-Transformationen

auswählen kann. Die Vorschläge basieren auf Laufzeitprofilen. Unter Berücksichtigung der besonderen Eigenschaften von Streaming-Anwendungen wie Audio-, Video- oder digitale Signalverarbeitung kann Pipeline-Parallelität bereits durch minimalistische Annotationen erreicht werden [TCA07],[GTA06].

Automatische Parallelsierungskonzepte unterscheiden sich zunächst in der Gewinnung der zugrundegelegten Daten (siehe Kapitel 3.10 bzw. 3.11). Die dort diskutierten Eigenschaften beschränken unter Umständen den zulässigen Sprachumfang der sequentiellen Beschreibung. [PJP+15] bietet eine Abriss exisiterender Tools und deren Abhängigkeitsanalyse- bzw. Schleifentransformationsmechanismen.

Ein in der Literatur häufig zugrundegelegter Ansatz kombiniert die Analyse von Laufzeitverhalten mit maschinenbasiertem Lernen. Die dynamische Analyse ermöglicht es, Einschränkungen der statischen Analyse zu umgehen. Anstelle von Heuristiken wie in [RS89] versucht [TWF+09],[TF10] das maschinenbasierte Lernen Vorhersagen für das Mapping zu treffen. Mögliche Parallelität wird auf Basis der Laufzeitprofile detektiert. Für jeden Schleifenkandidaten wird über die Mechanismen des maschinellen Lernens entschieden, ob und wie ein paralleles Mapping ausgeführt wird. Dabei wird auf die Parallelisierung von Schleifen und die Übersetzung in OpenMP abgezielt. [WTF+14]

Das Helix-Projekt [CJH+12c] wählt für eine automatische Parallelisierung einen einfachen Ansatz zur Schleifentransformation. Einzelne Iterationen werden auf unterschiedliche Ausführungseinheiten verteilt. Die daraus resultierende notwendige Kommunikation zur Synchronisierung und zum Datenaustausch darf von ihrem Aufwand den Gewinn nicht zunichtemachen. Daher wird die zu transformierende Schleife per Heuristik gewählt. Eine Transformation auf mehreren ineinander geschachtelten Ebenen ist dabei nicht vorgesehen. Mögliche Algorithmen zur Transformation finden sich in [CJH+12b]. [CBK+14] beschäftigt sich darauf aufbauend mit der weiteren Reduzierung der Latenzzeiten beim Informationsaustausch. [CJH+12a]

[LPC12] beschreibt eine Möglichkeit sequentielle imperative Programme in parallele Datenfluss-Programme zu transformierten. Dadurch lassen sich Task-, Pipeline- und Datenparallelität ausnutzen. Dieser Ansatz beschränkt sich, im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz, jedoch auf skalare Datenabhängigkeiten; per Zeiger referenzierte Daten sowie zusammengesetzte Datentypen (z. B. Arrays) werden nicht unterstützt. Die Granularität der parallel auszuführenden Einheiten lässt sich in diesem Zusammenhang vergröbern, über maschinelle Mechanismen wird ein optionales Verschmelzen propagiert.

Viele Programme aus dem wissenschaftlichen Umfeld basieren auf Algorithmen, welche sich als sogenanntes Nested Loop Program (NLP) realisieren lassen. Diese Programmklasse basiert auf geschachtelten Schleifen und dem Verzicht auf Unterprogrammaufrufe. Die geringere Komplexität gegenüber dem vollen Sprachumfang sorgt für eine statische Berechenbarkeit der Kontroll- und Datenflüsse. Drei Algorithmen zur Parallelisierung von NLPs werden in [DV95] verglichen. Im Besonderen findet diese Programmklasse Verwendung in der Parallelisierung mittels des Polytop-Modells. Dieses ist eine Untergruppe der Polyedermodelle. Das grundelegende Polytopmodell wurde in [Len93],[Fea96] zur automatischen Parallelisierung herangezogen. Die Idee ist es, eine Transformation der Schleifenindexvariablen vorzunehmen (vgl. Abb. 1.2). Dadurch

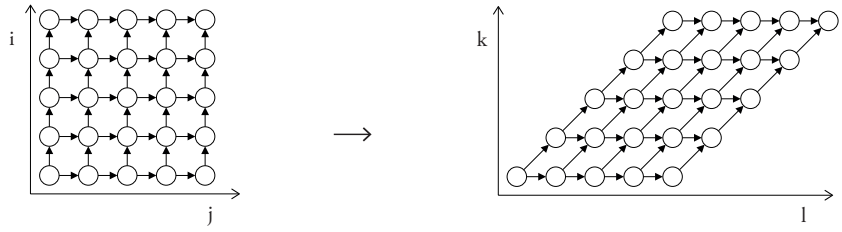


Abbildung 1.2: Transformation eines Polytops mit dem durch die Indexvariablen i und j definierten Indexraum in ein verschobenes Polytop mit den neuen Indexvariablen k und l. Pfeile repräsentieren die Abhängigkeiten, diese bleiben bei der Transformation erhalten.

kann das Programm unter Erhalt der Datenabhängigkeiten neu geordnet, jedoch möglichst viele Iterationen parallel ausgeführt werden. Die Voraussetzung für diesen Spezialfall der Schleifenparallelisierung ist, dass das Quellprogramm nur aus for-Schleifen und Zuweisungen aufgebaut ist. Als mögliche Datentypen sind Arrays und Skalare erlaubt. Schleifengrenzen und Array-Indizes dürfen nur in einem linearen Zusammenhang der Schleifenindizes stehen. Eine Ausdehnung auf while-Schleifen, bei denen die Anzahl der Schritte erst zur Laufzeit feststeht, wurde in [Gri96],[Col95] vorgenommen. Der Indexraum lässt sich dann nicht mehr als Polytop darstellen, jedoch als Polyeder. Der dynamische Indexraum ist uneben. Ein konservativer Ansatz zum Umgang mit den resultierenden Problemen z. B. der globalen Termination ist Kern von [Gri96]. Dem gegenüber verfolgt [Col97] einen spekulativen Ansatz. Kontrollabhängigkeiten der äußeren while-Schleife werden ignoriert. Somit kann ein Rollback von Schleifenschritten notwendig werden. [Gri04] betrachtet zusätzlich den Aspekt der richtigen Granularität für die Parallelisierung.

Iteration-Space-Slicing stellt eine Erweiterung des in Kapitel 5.2.6 beschriebenen Programm-Slicing dar. Dessen Anwendung für die Parallelisierung von NLPs wird in [BBC+11] behandelt.

In [SHZ+12] wird *Sambamba* als Infrastuktur vorgestellt. Diese ermöglicht es, Optimierungen nicht statisch während des Übersetzungsprozesses, sondern während der Ausführung des fertigen Programms adaptiv durchzuführen. Die Ausführung eines Programms in einer Laufzeitumgebung lässt spekulative Transformationen zu. Somit wird die Ausführung angepasst und auf Basis verfügbarer Ressourcen mit erneuten Iterationen beschleunigt ausgeführt.

Ein Verfahren zur Auslagerung von Methoden auf Hilfsprozessoren wird unter dem Namen *Program-Demultiplex* in [BS06] präsentiert. Die Ausführung von Methoden erfolgt dabei nicht in der programmierten Reihenfolge, sondern kann nebenläufig erfolgen, sobald die benötigten Eingangsdaten vorhanden sind. Dies ist sowohl bezüglich der Ausführung als auch bezüglich der Daten spekulativ.

Der in dieser Arbeit vorgestellte Ansatz strebt an, ein hohes Maß an Flexibilität zu ermöglichen. Der Einsatz der LLVM-Compiler-Infrastruktur ermöglicht eine Vielzahl

von Sprachen mit vollem Sprachumfang über entsprechende Frontends in ein universelle Zwischenräpresentation zu übersetzen. Das Modell, welches an diese Stelle ansetzt, wiederum ist offen für die Anwendung verschiedener Ansätze, so wäre die Anwendung des Polytopenmodells denkbar. Alle möglichen Arten der Parallelität, wie beschrieben, können zum Einsatz kommen. Auch der Schritt hin zu ausführbaren Einheiten ist bezüglich der ausführenden Architektur nicht eingeschränkt.

Neben einer parallelen Ausführung mittels mehrerer Standardprozessoren, lässt sich auch eine Realisierung in Hardware z. B. mittels FPGAs anstreben. Die Beschleunigung von Datenverarbeitung per FPGA hat zu Entwicklungen im Bereich der Transformation von C-Programmen in VHDL oder Verilog geführt. Sogenannte C2RTL-Tools wie [McC04], [eXCite], [GDG+03] können kleinere Programme konvertieren. Die meisten Ansätze beschränken sich jedoch auf konkrete Anwendungen. [LLH+13] präsentiert einen Ansatz für gemischt-ganzzahlige lineare Programmierung (Integer linear Programming, ILP) bei Streaminganwendungen. Die Anwendung von Pipelineverarbeitung für verschachtelte Schleifen in ILPs wird auch in [CHM+11] behandelt. Die Synthese aus generellen Verhaltensbeschreibungen wird in [Gre14] behandelt.

1.3 Aufbau der Arbeit

Das grundlegende Konzept zur Parallelisierung besteht darin, eine existierende sequentielle Programmbeschreibung in eine standardisierte Form zu überführen, in Komponenten zu zerlegen, und diese gegebenenfalls in veränderter Reihenfolge, möglicherweise auch nebenläufig auszuführen. Dabei darf sich das Verhalten des Gesamtsystems nicht verändern. Diese Arbeit folgt im wesentlichen in ihrem Aufbau den hier kurz beschriebenen Schritten. Jeder Teilschritt ist durch verwandte Methoden oder Konzepte beeinflusst. Daher werden zusätzlich zu dem Stand der Technik im vorigen Abschnitt entsprechende Grundlagen referenziert.

Eine Parallelisierung kann nur durch Transformationen eines Programmsystems erfolgen. Diese können jedoch an mehreren Stellen während des Übersetzungsprozesses, von der durch den Programmierer formulierten Beschreibung bis hin zu der für die ausführende Hardware verständliche Maschinensprache, ansetzen. Daher werden in Kapitel 2 die Grundlagen von Programmbeschreibungen bzw. Quelltext zusammengefasst und der Ablauf der Übersetzung hin zu ausführbaren Programmen, dem Kompilieren, beschrieben.

Eine weitere Basis für die Konzeptionierung des Zerlegens in Komponenten stellen die Beziehungen zwischen eben diesen Teilen dar. Diese programminternen Abhängkeiten werden daher in Kapitel 3 behandelt.

Wie zuvor erwähnt, ist die Visualisierung ein sinnvolles Hilfsmittel bei der manuellen Analyse von Programmen. In Kapitel 4 findet sich daher ein Überblick über etablierte Verfahren in Abgrenzung zu einer im weiteren Verlauf der Arbeit entwickelten Darstellungsform.

Auf den grundlegenden Kenntnissen aufbauend wird das Thema der Unterteilung behandelt. Kapitel 5 enthält dabei den Stand der Technik. In Ergänzug dazu wird in

Kapitel 6 die Auswahl eines geeigneten Modells diskutiert. Dabei wird auf relevante Eigenschaften einer geeigneten Zerlegung, welche das Ziel der Parallelisierung im Fokus hat, eingegangen.

Das für die nächsten Schritte unabdingbare Wissen über programminterne Zusammenhänge erfordert die Extraktion dieser Information aus dem gegebenen Programm. Kapitel 3.10 und 3.11 beschreiben, wie die dafür benötigten Informationen aus realer Software ermittelt werden können, und welche technischen Ansätze dafür existieren.

Aus den vorgegangenen Schritten ergeben sich komplexe Strukturen, welche auf mögliche Parallelität hin zu untersuchen sind. Zur Entwicklung einer automatisierten Verarbeitung muss zunächst eine manuelle Exploration typischer Strukturen erfolgen. Hierfür ist eine angepasste Visualisierung hilfreich. Neben bestehenden Verfahren der Softwarevisualisierung, ermöglichen die besonderen Eigenschaften der Zerlegung einen neuen Ansatz für die Darstellung. Diese Darstellung von Kontroll- und Datenflussabhängigkeiten, wird in Kapitel 7 behandelt. Hieraus ergibt sich zudem ein Ansatz die speziellen Modelleigenschaften für die Programmausführung nutzbar zu machen.

Anhand dieser Darstellung und des in Kapitel 5 eingeführten Modells können sequentielle Programme auf mögliche Parallelität hin untersucht werden. Dies wird in Kapitel 8 erörtert.

Ziel sollte es sein, die Ausführung des Programms anzupassen. Hierzu ist es erforderlich die zuvor gewonnenen Kenntnisse umsetzen zu können. Über die reine Zerlegung hinaus werden weitere Anpassungen notwendig. Diese und Probleme die sich durch neue Zielarchitekturen ergeben, und wie diese behandelt werden können, ist ein weiterer Teil des Kapitels 8. Darüber hinaus werden Konzepte für die Ausführungssteuerung entwickelt.

Abschließend findet in Kapitel 9 eine Schlussbetrachtung der gewonnenen Erkenntnisse und Ergebnisse statt.

Kapitel 2

Compiler

In diesem Kapitel werden die Grundlagen zur Erstellung von Computerprogrammen betrachtet. Der Fokus liegt dabei aus den Vorgängen bei der Übersetzung des Quelltextes in ein ausführbares Programm, dem Kompilieren (Abschnitt 2.3). Neben Eigenschaften von Programmiersprachen (Abschnitt 2.2) werden auch Zielarchitekturen beleuchtet (Abschnitt 2.4). Die einzelnen Phasen bieten jeweils unterschiedliche Optionen um Veränderungen, z. B. Optimierungen, einzubrigen.

2.1 Einführung

In der Regel werden heutzutage Computerprogramme in einer höheren Programmiersprache abgefasst. Jede Programmiersprache dient dem Formulieren von Algorithmen und Datenstrukturen. Das so beschriebene Programm kann durch einen Computer ausgeführt werden. Im Vergleich zur Maschinensprache erlauben Hochsprachen einen höheren Grad der Abstraktion und somit eine kompaktere Formulierung komplexer Strukturen. Zudem ist die Ausdrucksweise für Menschen leichter verständlich. Die Softwareentwicklung ist somit einfacher, schneller und weniger fehleranfällig verglichen mit der direkten Formulierung von Maschinenbefehlen. Liegen Programme nur in Maschinensprache vor, ist ein Nachvollziehen sehr erschwert.

Man unterscheidet mehrere Sprachgenerationen. Sprachen der ersten Generation sind Maschinensprachen. Assemblersprachen, welche leichter zu merkende Namen für Maschinenbefehle aufweisen und den Befehlssatz um Makros erweitert, gehören zur zweiten Generation. In die Kategorie der dritten Generation fallen Sprachen wie Fortran, Cobol, Lisp, C, C++ und Java. Mit der vierten Generation werden Sprachen für besondere Anwendungen, z. B. SQL, bezeichnet. Den Begriff der 5. Generation findet man im Zusammenhang mit Logik- und Bedingungsorientierten Sprachen wie Prolog.

Im weiteren Verlauf dieser Arbeit wird die in den 1970er Jahren entwickelte Sprache C im Vordergrund stehen. Sie ist eine weit verbreitete Sprache, was auch durch die Vielzahl an Programmierschnittstellen für Anwendungsprogramme gefestigt wird. Es ist daher sinnvoll, sie als eine von vielen möglichen Sprachen als Grundlage für die folgenden Arbeiten zu verwenden. Häufig dient C als Sprache für die System- und An-

wendungsentwicklung, und auch für eingebettete Systeme. C gehört in die Klasse der imperativen (auch prozeduralen) Programmiersprachen. Imperativ bedeutet, dass angegeben wird, welche Anweisungen in welcher Reihenfolge ausgeführt werden sollen. Dem gegenüber stehen deklarative Sprachen, welche beschreiben, *was* berechnet werden soll, jedoch nicht *wie*. Im Rahmen der Systemprogrammierung liegt der Vorteil in der Verwendung hardwarenaher Konstrukte und direkter Speicherzugriffe. C ist nicht typsicher, d. h. unterschiedliche Datentypen können ohne explizite Konvertierung einander zugewiesen werden.

Um ein in einer höheren Programmiersprache formuliertes Programm auf einem Computer ausführen zu können, muss das Quellprogramm jedoch zunächst in ein semantisch äquivalentes Zielprogramm überführt werden. Dies ist die Aufgabe des Compilers. Dieser Prozess gliedert sich in zwei wesentliche Punkte. Zunächst wird das Quellprogramm analysiert und in eine Zwischenrepräsentation (Abschnitt 2.3.2) überführt. Anschließend kann die Synthese das Zielprogramm generieren. Die Analyse geschieht im Frontend des Compilers (Abschnitt 2.3.1), die Synthese im Backend (Abschnitt 2.3.3). Das Frontend beinhaltet alle Schritte, welche sich auf eine spezifischen Quellsprache beziehen. Das Backend hingegen ist von der Quellsprache unabhängig, die zugehörigen Phasen beziehen sich auf die Zielsprache. Wobei sich Front- und Backend, wie im Folgenden gezeigt, in weitere Schritte untergliedern. Häufig werden während des Übersetzungsprozesses (zur Übersetzungszeit, Kompilierzeit, (*compile time*)) maschinenunabhängige Optimierungen durchgeführt. Die Eigenschaften der Zielarchitektur können in eine zweite Optimierungsphase (maschinenabhängig) einfließen.

2.2 Quellcode

Der Quellcode (auch Quelltext oder Programmcode (*source code*)) ist eine „in einer [höheren] Programmiersprache geschriebene Abfolge von Programmanweisungen, die vom Menschen gelesen, aber erst nach einer elektronischen Übersetzung vom Computer verarbeitet werden können“. [Dud13]

Der Quellcode besteht aus Anweisungen die die Funktionalität beschreiben und Kommentaren, welche keine Auswirkung auf das ausführbare Programm haben. Anweisungen werden in der Regel durch den Computer ausgeführt, die Ausnahme stellen Definitionen dar, die Informationen über Datenobjekte für den Übersetzungsprozess festlegen. Anweisungen (*statement*) können einfach oder zusammengesetzt sein. Beispiele für zusammengesetzte Anweisungen sind Blöcke, Schleifen oder Verzweigungen. Veränderliche Werte werden in Variablen gespeichert. Jede Variable wird mit einem Namen bezeichnet, über diesen Namen wird ein zugeordneter Speicherbereich angesprochen. Der dort gespeicherte Werte kann somit im Programm referenziert und durch Wertzuweisung verändert werden. In einer Umgebung (*environment*) wird ein Name auf einen Speicherort abgebildet. Der Zustand der Variablen (*state*) ist die Abbildung des Speicherortes auf den gespeicherten Wert.

Zeiger sind Variablen, die jedoch keinen Wert sondern die Speicheradresse einer Variablen beinhalten. Ein Zeiger enthält somit eine Referenz auf eine andere Variable oder

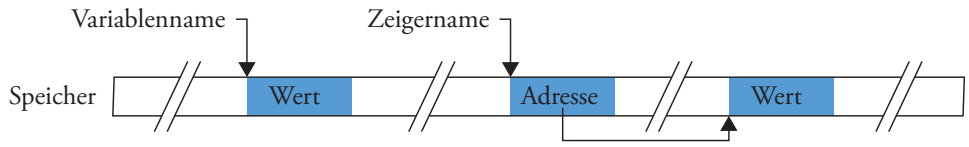


Abbildung 2.1: Variablen und Zeiger referenzieren Speicherbereiche

ein anderes Objekt im Speicher. Bei C und C++ kann ein Zeiger auch auf Unterprogramme (s. u.) verweisen (Funktionspointer). Während der Programmausführung kann ein Zeiger sein Ziel ändern, somit auf mehrere Speicherbereiche zeigen, jedoch zu jedem Zeitpunkt nur auf ein Ziel.

Ein Variablenname wird durch die Deklaration eingeführt. Hierbei wird neben dem Bezeichner auch der Datentyp und die Dimension (Felder (*array*)) festgelegt. Wird einer zuvor deklarierten Variable ein neuer Wert zugewiesen, spricht man von *Definition*. Der lesende Zugriff wird als Referenzierung bezeichnet.

Da ein Bezeichner möglicherweise mehrfach im Programm verwendet werden kann, ist der Gültigkeitsbereich (*scope*) einer Variablen von Bedeutung. Programmiersprachen können entweder dynamische oder statische Gültigkeitsbereiche verwenden. Die meisten Sprachen verwenden statische Gültigkeitsbereiche, d. h. die Gültigkeit ist im Quelltext ersichtlich. Wird beispielsweise der Name `x` als Bezeichner für eine globale Variable sowie innerhalb der Funktion `f` für eine lokale Variable verwendet, so verweist dann `x` innerhalb von `f` auf die lokale, außerhalb jedoch auf die globale Variable. Bei statischen Gültigkeitsbereichen bestimmt der Ort der Deklaration im Quelltext die Gültigkeit. In einigen Programmiersprachen (z. B. C++) ermöglichen Schlüsselwörter (`public`, `private`, `protected`) eine explizite Kontrolle. Die Grundlage für die Gültigkeitsbereiche bilden Blöcke. Wobei Blöcke eine Gruppe von Deklarationen und Anweisungen beschreiben. In C werden Blöcke durch { } begrenzt. Blöcke können verschachtelt sein. Eine innerhalb eines Blocks deklarierte Variable ist überall innerhalb dieses Blockes gültig, außer in einem anderen Block erfolgt eine erneute Deklaration. In diesem Fall verhält es sich wie im obigen Beispiel.

Ebenso wie Variablen erhalten Konstanten und Prozeduren bzw. Funktionen Namen. Prozeduren oder Funktionen sind Unterprogrammaufrufe, welche der Strukturierung des Quelltextes bzw. zur Vermeidung von Codewiederholungen dienen. Sie fassen eine Folge von Anweisungen zu einer Einheit zusammen. Nach Ausführung eines Unterprogramms kehrt der Ausführungsfaden an die Stelle des Aufrufs zurück.

Unterprogramme bzw. Subroutinen können mit Parametern definiert und aufgerufen werden. Dabei werden zwei Arten der Parameterübergabe unterschieden. Zum einen können Werte an das Unterprogramm übergeben werden (*call by value*), diese stehen dann innerhalb der Subroutine zur Verfügung. Zum anderen können Referenzen auf Werte, also die Speicheradressen, an denen die Werte gespeichert wurden, übergeben werden (*call by reference*). In dem Unterprogramm kann somit auf die Werte zugegriffen werden, und Änderungen an den Werten ändern die entsprechende Speicherstelle und sind somit auch in übergeordneten Programmteilen verfügbar. Die Übergabe einer Refe-

renz führt dazu, dass mehrere Namen auf denselben Speicherbereich verweisen können. Diese Variablen werden als Aliase bezeichnet. Da ein Zeiger eine Variable ist, welche eine Zieladresse speichert, kann dieses Ziel jederzeit durch das Programm verändert, und eine neue Zieladresse an gleicher Stelle, gespeichert werden.

Eine weitere Klassifizierung bezieht sich auf Rückgabewerte. Funktionen ermitteln einen Wert, welcher als direkte Folge des Aufrufs zur Verfügung steht. Prozeduren hingegen haben keinen Rückgabewert. In einigen Programmiersprachen wird das Schlüsselwort „void“ anstelle eines Datentyps verwendet, um anzuzeigen, dass ein Unterprogramm keine Daten zurückgibt.

2.3 Übersetzungsprozess

Der Übersetzungsprozess, also die Abbildung des Quelltextes auf Maschinenbefehle einer konkreten Hardware, gliedert sich in zwei Phasen. Zunächst wird der Quellcode analysiert und in eine Zwischenrepräsentation übersetzt (Abschnitt 2.3.1). Diese stellt eine Programmdarstellung auf einer geringeren Abstraktionsebene als das ursprüngliche Programm dar und dient als Ausgangsbasis für die zweite Phase (Abschnitt 2.3.2). In diesem zweiten Schritt erfolgt dann die Transformation hin zu einem in Hardware lauffähigen Programm (Abschnitt 2.3.3). Während des Prozesses werden neben den reinen Transformationen auch Optimierungen durchgeführt.

2.3.1 Frontend

Das Frontend analysiert den Quellcode, untersucht ihn dabei auf Fehler und liefert eine strukturierte Darstellung als Übergabe an das Backend. Dieser Vorgang unterglierdert sich in die folgenden Abschnitte:

2.3.1.1 Lexikalische Analyse

Die lexikalische Analyse wird im Compiler durch den sogenannten *Scanner* realisiert. In dieser Phase werden die Zeichen des Quelltextes eingelesen und zu zusammengehörigen Sequenzen gegliedert, den Lexemen. Für jedes Lexem wird ein Symbol bzw. Token ausgegeben. Token stehen für Kommentare, Sonderzeichen oder Bezeichner von Standardobjekten.

Die Anweisung (*statement*) S

```
y = 42 + steigung * x
```

wird in die Lexeme y, $=$, 42, $+$, $steigung$, $*$, x gegliedert. Ein Token kann entweder einen Operator, oder ein Verweis auf die Symboltabelle enthalten. In der Symboltabelle werden alle vergebenen Namen abgelegt. In diesem Beispiel werden Verweise auf eine Symboltabelle für y, $steigung$ und x ausgegeben. Zusätzlich werden drei Token für die Zuweisung, die arithmetischen Operatoren sowie die Konstante `42` an die syntaktische Analyse übergeben.

Abbildung 2.2: Symboltabelle (a) und Syntaxbaum (b)

Die Symboltabelle enthält für jeden im Programm deklarierten Bezeichner einen Eintrag. Zugeordnet werden Attribute wie der benötigte Speicherbedarf, der Typ, der Gültigkeitsbereich oder bei Unterprogrammbezeichnern die Anzahl und der Typ der Argumente.

2.3.1.2 Syntaktische Analyse

Die Syntax einer Programmiersprache beschreibt, wie bei jeder Sprache, ein Regelwerk (Grammatik), welches die Verknüpfung der Elemente (Wörter) zu Gruppen (Sätze) definiert. Die Syntaxanalyse wird auch *parsing* genannt. Der Parser erstellt, auf Basis der Grammatik, einen Syntaxbaum, eine baumartige Struktur, welche die Tokens der lexikalischen Analyse enthält. Jeder innere Knoten stellt eine Operation dar, seine Kindknoten die Operanden. Der Syntaxbaum und die Symboltabelle für die Anweisung aus Abschnitt 2.3.1.1 ist in Abbildung 2.2 dargestellt.

2.3.1.3 Semantische Analyse

> Der semantische Analysator verwendet den Syntaxbaum und die Informationen in der Symboltabelle, um das Quellprogramm auf semantische Konsistenz mit der Sprachdefinition zu überprüfen. Außerdem sammelt er Typinformationen und speichert sie entweder im Syntaxbaum oder in der Symboltabelle, damit sie beim Erstellen des Zwischencodes verwendet werden können. [ULS+08, Seite 9]

Eigenschaften, welche nicht durch die syntaktische Analyse geprüft werden können sind beispielsweise die nicht zulässige doppelte Deklaration von Bezeichnern. Auf Basis des Syntaxbaumes können auch Typüberprüfungen vorgenommen werden. Jeder Operator muss passende Operanden haben, evtl. kann der Compiler automatische Konvertierungen vornehmen. Treten im Programm syntaktische oder semantische Fehler auf, führt dies zu einem Abbruch mit einer entsprechenden Fehlermeldung.

2.3.1.4 Maschinenunabhängige Optimierung

Alle maschinenunabhängigen Optimierungen, welche sich ausschließlich auf die Quellsprache beziehen, werden dem Frontend zugeordnet. Das Ziel dieser Phase ist es, den

Zwischencode so zu verbessern, dass das Backend besseren Zielcode generieren kann. Das Optimierungsziel kann dabei entweder die schnellere Ausführung, kürzere ausführbare Dateien oder der geringere Ressourcenverbrauch (z. B. Speicher) bei der Ausführung sein.

In diesem Schritt wird ermittelt, welche Dinge bereits zur Übersetzungszeit berechnet werden können, z. B. Konvertierungen von Konstanten. Je nachdem, wie das Programm ausgeführt wird, muss die maschinenunabhängige Optimierung statisch, also zur Übersetzungszeit, oder dynamisch also zur Laufzeit durchgeführt werden.

2.3.2 Zwischencode

Der Syntaxbaum stellt das Programm in einer Form dar, welche von abstrakten Maschinen verarbeitet werden kann. Eine alternative Darstellung ist die Form des Zwischencodes. Er bildet den Syntaxbaum auf eine Folge von Instruktionen I_j (auch als Befehle bezeichnet (*instruction*)) ab, ist also eine linearisierte Form des Zwischencodes. Er lässt sich einfach auf Basis der vorherigen Schritte generieren. Die Art der Darstellung ist häufig maschinennah, die Übersetzung in Maschinenbefehle ist daher einfach. Komplexe Ausdrücke müssen aufgelöst werden. Die einfache Struktur eignet sich gut für die Anwendung von Optimierungen.

Eine einheitliche Definition des Zwischencodes ermöglicht die Verwendung diverser Front- und Backends. Folglich kann dann eine Compiler-Infrastruktur verschiedene Sprachen verarbeiten und ausführbare Programme für diverse Zielmaschinen erstellen.

2.3.2.1 Drei-Adress-Code

Eine verbreitete Form des Zwischencodes ist der Drei-Adress-Code. Es handelt sich dabei um eine Folge einfacher Befehle und ähnelt Assembler. Die verwendeten Befehle verwenden, wie der Name nahelegt, bis zu drei Operanden. Es lassen sich drei Befehlsklassen unterscheiden.

Zuweisung Eine Zuweisung (*assignment*) a hat die Form

```
x := y op z.
```

Die Operanden `x, y, z` können im Quelltext verwendeten Namen entsprechen oder Konstanten bzw. vom Compiler generierte Werte sein. Da auf der rechten Seite der Zuweisung höchstens ein Operator stehen kann, muss der Compiler temporäre Namen anlegen und komplexere Befehle aufteilen. Jeder Operand kann sich wie ein Register verhalten. Der Operator `op` kann ein binärer, arithmetischer oder logischer Operator sein.

Neben der oben genannten Form der Befehle können auch andere Fälle auftreten; für logische Negierungen oder Schiebe-Operationen (*shift*)

```
x := op y
```

oder für Kopien

```
x := y.
```

Indizierte Zuweisungen verwenden Operanden der Form `x[i]`. Der referenzierte Wert liegt hierbei `i` Speichereinheiten hinter `x`.

Die letzte Gruppe betrifft Adress- und Zeigerzuweisungen.

```
x := &y
```

weist `x` die Adresse von `y` zu,

```
x := *y
```

weist `x` den Wert zu, auf welchen der Zeiger `y` zeigt. Die Speicherstelle welche `y` referenziert enthält hierbei die Adresse an welcher der Wert, der an der Speicherstelle von `x` gespeichert wird, gespeichert ist.

```
*x := y
```

Die Speicherstelle auf welche der Zeiger `x` zeigt wird mit dem Wert der Variablen `y` beschrieben.

Kontrollflusssteuerung Die Kontrollflusssteuerung erfolgt über Sprungbefehle j. Hierbei werden unbedingte und bedingte Sprünge unterschieden. Der unbedingte Sprung hat die Form

```
goto L
```

Wobei `L` die Sprungmarke (*label*) darstellt, diese wird als nächstes ausgeführt.

Bedingte Sprünge können den Sprung zu einer Sprungmarke von einer Bedingung abhängig machen, die entweder *wahr*

```
if x goto L
```

oder *falsch*

```
ifFalse x goto L
```

sein muss. Trifft die Bedingung nicht zu, so wird das Programm bei dem darauffolgenden Befehl weiter ausgeführt.

Ein weiterer bedingter Sprung wie

```
if x rel_op y goto L
```

verwendet einen relationalen Operator `rel_op` ($<$, $=$, $>=$, …) welcher sich auf `x` und `y` bezieht. Stehen `x` und `y` in der angegebenen Beziehung, wird der Sprung ausgeführt.

Hierarchische Strukturierung Hierarchie wird, wie in Kapitel 2.2 beschrieben, durch Unterprogramme realisiert. Der Unterprogrammaufruf *call* wird durch eine Folge von Drei-Adress-Befehlen realisiert:

```
param x1
param x2
...
param xn
call p, n     // oder  y = call p, n
```

`param` implementiert die Übergabeparameter, `call` definiert den eigentlichen Aufruf mit der Anzahl der vorher definierten und zu übergebenen Parameter. Die Zuweisung des Rückgabewerts in Zusammenhang mit dem Aufruf ist optional.

In der Subroutine wird `return y` verwendet um ein Rückkehren zu der Stelle des Aufrufs, inkl. Rückgabewert `y` zu realisieren.

2.3.2.2 Static Single Assignment

Eine weit verbreitete Form des Zwischencodes ist die statische Einzelzuweisungsform (*Static Single Assignment, SSA*). Sie ermöglicht eine einfache Datenflussanalyse und darauf basierend vereinfachte Codeoptimierungstechniken. Wird jede Variable nur an einer Stelle im Programm definiert, ist der Zwischencode in SSA-Form. Der Kontrollfluss kann diese Stelle beliebig oft durchlaufen. Die Referenzierung einer Variablen kann beliebig oft auftreten. Durch diese Eigenschaft lässt sich jeder lesende Zugriff eindeutig einer Definition zuordnen. Werden Variablen definiert, jedoch nicht referenziert, wenn ein geschriebener Wert also nicht verwendet wird, dann kann der zugehörige Code entfernt werden (*dead code elimination*).

Wird in der Drei-Adress-Code-Darstellung eine Variable mehrfach als Ziel von Zuweisungsoperationen verwendet, ist ein Konvertierung notwendig um eine Darstellung in der SSA-Form zu erhalten. Hier können die Ziele der einzelnen Zuweisungsoperatoren durch Verwendung von Indizes unterschieden werden.

Ein weiterer Aspekt betrifft eine Kontrollflussverzweigung. Angenommen im Quellcode wurde eine Verzweigung der Form

```
if (bedingung == true) { x = 42; } else { x = 73; }
```

verwendet, so wird x auf zwei unterschiedlichen Kontrollflusspfaden, abhängig von `bedingung`, definiert.

```
if (bedingung == true) {
    x1 = 42;
} else {
    x2 = 73;
}
```

Bekommen im Zwischencode Ziele unterschiedliche Namen, so wird im weiteren Programmverlauf die Frage nach der korrekt definierten Variable auftreten. Die statische Einzelzuweisungsform verwendet hier die sogenannte ϕ-Funktion; ein für diese Art der Darstellung charakteristisches Konstrukt. Die Notation `x3 = ϕ(x1,x2)` gibt an, dass `x3` der Wert desjenigen Arguments zugewiesen wird, dessen Kontrollflusspfad eingeschlagen wurde. Die ϕ-Funktion liest nur einen Parameter, den der zuletzt gewählten Kontrollflusskante, daher kann der Wert nicht undefiniert sein. Die Anzahl der Parameter richtet sich nach der Anzahl der möglichen Kontrollflusspfade. [CFR+91]

2.3.3 Backend

Das Backend hat die Aufgabe, die Befehle des Zwischencodes in Maschinenbefehle zu überführen, Register zuzuweisen und die Befehlsreihenfolge festzulegen.

2.3.3.1 Maschinenabhängige Optimierung

Die Maschinenabhängige Optimierung umfasst unter anderem die Registervergabe (siehe Abschnitt 2.3.3.2). Die Ausführungsreihenfolge kann verändert werden um weniger Register für Zwischenergebnisse verwenden zu müssen, und um die Befehlsausführung des Prozessors auszunutzen (siehe Abschnitt 2.4.2.1). In Spezialfällen können allgemeine Befehle durch effizientere Maschinenbefehle ersetzt werden.

2.3.3.2 Codegenerierung

Die Codegenerierung führt eine Abbildung der Zwischensprache auf die Zielsprache durch, also eine Folge von Maschinenbefehlen. Die zentrale Aufgabe besteht in der Wahl von Speicherplätzen oder Registern für Variablen. Diese Speicher- und Adresszuordnung wird durch Maschineneigenschaften beeinflusst, denn die Wort- und Adresslänge unterscheidet sich bei verschiedenen Prozessoren. Jedem Namen im Zwischencode wird ein Speicher zugewiesen. Hierbei können für einige Namen Register reserviert werden. Register ermöglichen kürzere Zugriffszeiten als die Verwendung von Speicher.

2.4 Ziel-Architekturen

Die Ausführung von Programmen kann auf verschiedenartiger Hardware erfolgen. Die Compiler müssen sich dabei an die Entwicklung der Hardwarearchitektur anpassen. Durch die Entwicklung neuer Übersetzungsalgorithmen kann das Potential der jeweiligen Prozessoren am besten ausgenutzt werden. Neben den Vielzweck-Prozessoren können auch Spezialprozessoren für besondere Anwendungen zum Einsatz kommen. Ein Prozessor oder CPU (*Central Processing Unit*) besteht aus dem Steuerwerk, dem Rechenwerk, Registern, einem Befehlszähler und einem Speichermanager (*MMU (Memory Management Unit)*). Die Abarbeitung einzelner Instruktionen erfolgt in mehreren Schritten. [Tan06]

1. Der nächste Befehl wird aus dem Speicher geladen und in das Befehlsregister gespeichert
2. Der Befehlszähler (*program counter*) wird inkrementiert, so dass er auf den nächsten Befehl zeigt
3. Der Typ des aktuellen Befehls wird bestimmt
4. Falls der Befehl ein Speicherwort benötigt, wird dessen Position bestimmt
5. Falls notwendig, wird das Speicherwort geladen und in ein Register abgelegt
6. Der Befehl wird durch das Rechenwerk (*ALU (Arithmetic Logic Unit)*) ausgeführt

Ende der 1970er wurde versucht, die Rechenleistung mittels komplexer Befehle zu steigern. Der Befehlssatz der Prozessoren sollte die Fähigkeiten der Maschinen an die Möglichkeiten der Hochsprachen anpassen. An der Berkeley-Universität wurde 1980 angefangen einfachere Prozessoren zu entwickeln. In diesem Zusammenhang wurde der Begriff RISC (Reduced Instruction Set Computer) geprägt. Durch diese Entwicklung wurde es möglich, wesentlich mehr Befehle pro gleicher Zeiteinheit zu starten. Auch wenn die eigentliche Bearbeitung eines Befehls nicht weniger Zeit in Anspruch nimmt, so kann die Rechenleistung dennoch, im Vergleich zu CISC (Complex Instruction Set Computer), durch diesen Ansatz gesteigert werden. Befehle müssen nicht interpretiert werden, daher ist die Ausführung von mehreren Befehlen auf RISC-Prozessoren für eine Aufgabe schneller als die Verarbeitung eines einzelnen äquivalenten Befehls auf einem CISC-Prozessor. Die Ausführung des CISC-Befehlssatzes erfordert die teilweise Aufspaltung von Befehlen in Bestandteile welche als Mikrobefehle ausgeführt werden. Typische CISC-Prozessoren sind beispielsweise die INTEL-Pentium, Vertreter der RISC-Philosophie sind die Power-Architektur, SPARC, MIPS, Alpha oder ARM-Prozessoren.

Ein alternativer Ansatz der Datenverarbeitung wird durch digitale Schaltungen möglich. Hierbei kommen z. B. FPGAs (Field Programmable Gate Array) zum Einsatz. Ein FPGA ist ein digitaler integrierter Schaltkreis, der vom Anwender auf Gatter-Ebene programmiert werden kann. Es besteht aus vielen logischen Zellen wie etwa Gattern, Flip-Flops und Multiplexern. Jede diskrete digitale Logik lässt sich auf diese Technologie abbilden. Der Entwurf geschieht in der Regel durch eine Hardwarebeschreibungssprache, etwa VHDL (Very High Speed Integrated Circuit Hardware Description Language). Die resultierende Konfiguration wird dann in einem RAM des FPGAs gespeichert. Aktuelle FPGAs ermöglichen zudem eine dynamische Rekonfiguration bei der Teile der konfigurierten Logikzellen während des Betriebs geändert werden.

Eine weitere Möglichkeit mehr Rechenleistung zu erzielen, ist die Steigerung der Taktfrequenz. Diese Maßnahme hat jedoch Grenzen. Das größte Problem stellt dabei die Wärmeentwicklung und der Wärmetransport dar. Daher wird ein alternativer Ansatz verfolgt, Parallelität. Zum einen kann Parallelität innerhalb des Prozessors genutzt werden um mehr Befehle pro Zeiteinheit verarbeiten zu lassen. Zum Anderen können mehrere Prozessoren an derselben Aufgabe arbeiten. Je nach Anzahl der Prozessorkerne werden die Prozessoren als Multi- oder Manycore-Prozessoren bezeichnet. Für einige exemplarische Prozessoren ist die Entwicklung der Anzahl der Kerne in Abbildung 2.3 aufgezeigt. In der Halbleiterindustie wird versucht das Moorsche Gesetz zu erfüllen. Es besagt, dass sich die Anzahl der Transistoren pro Flächenenheit regelmäßig (alle 12 bis 24 Monate je nach Quelle) verdoppelt. Für komplexe Aufgaben werden jedoch auch verteilte Computersysteme (Cluster, Grid) bestehend aus vielen einzelnen Computern eingesetzt (Abbildung 2.4).

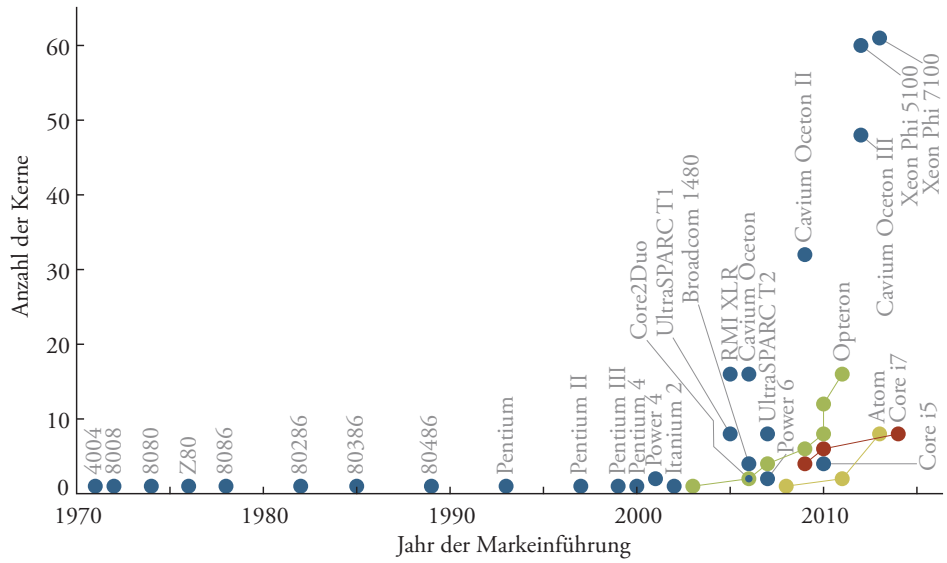


Abbildung 2.3: Entwicklung der Anzahl der Prozessorkerne pro Prozessor über die Zeit (Auswahl)

2.4.1 Speicherbereiche

Der Speicher welcher für die Programmausführung benötigt wird, lässt sich in vier Bereiche unterteilen. Zunächst werden

1. der Programmcode und
2. feste und globale Datenfelder

im Speicher abgelegt. Darüber hinaus unterscheidet man zwischen dem

3. Stack (Stapelspeicher) und dem
4. Heap (Haldenspeicher, dynamischer Speicher).

Auf Hardwareebene ist Speicher hierarchisch organisiert. Je größer die Entfernung vom Prozessor, desto höher das Speichervolumen, die Zugriffszeit und desto kleiner der Preis pro Bit. Die kleinste adressierbare Informationseinheit wird als Wort bezeichnet. Von kleinem schnellen Speicher hin zu großen langsameren Speichern ergibt sich die Reihenfolge: Register, L1-, L2 und L3-Cache, Hauptspeicher (RAM), Massenspeicher (z. B. Festplatte). Bei Mehrkernmaschinen unterscheidet man shared memory (gemeinsamer Speicher) und distributed memory (verteilter Speicher). Im ersten Fall können Prozesse über den Speicher Informationen austauschen, im zweiten muss dieser Austausch über Nachrichten (engl. message passing) erfolgen.

2.4.1.1 Stack

Der Stack arbeitet nach dem LIFO-Prinzip (Last In – First Out). Objekte werden in umgekehrter Reihenfolge vom Stack genommen wie sie dort abgelegt wurden. Verwendung findet dieser Speicher für lokale Variablen. Mit Betreten eines Blockes werden die lokalen Variablen auf den Stack gelegt. Am Ende des Blockes werden die Objekte zerstört, der

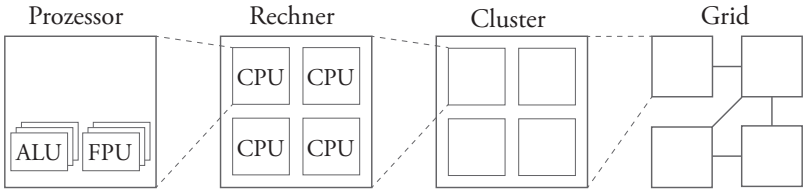


Abbildung 2.4: Ebenen der parallelen Datenverarbeitung

entsprechende Speicher somit wieder freigegeben. Eine weitere Aufgabe ist die Speicherung von Rücksprungadressen. Trifft die Ausführung auf einen Unterprogrammaufruf, wird die aktuelle Position im Programm auf dem Stack gespeichert. Wenn alle lokalen Variablen des Unterprogramms freigegeben wurden, liegt die Rücksprungadresse zuoberst. Die Programmausführung springt dann an diese Adresse. Die Größe des Stacks ist im Vergleich zum Heap stark begrenzt; häufig 1 MB. Wenn eine Programmausführung aus mehreren nebenläufigen Teilen(Threads) besteht, hat jeder einen eigenen Stack.

2.4.1.2 Heap

Der Heap dient als dynamischer Speicher. Speicherbereiche werden zur Laufzeit reserviert und wieder freigegeben. Im Gegensatz zum Stack sind die Objekte nicht geordnet. Im Programm kann jederzeit Speicher auf dem Heap reserviert werden. Dieser Speicher bleibt dann auch über Blockgrenzen hinweg erhalten. Spätestens bei Programmende muss der reservierte Speicher wieder freigegeben werden.

Da die Verwaltung weniger automatisch erfolgt, können Fehler zu Speicherlecks führen. Beispielsweise können Referenzen überschrieben werden, exisitiert dann kein Verweis auf einen Speicherbereich mehr, kann dieser auch nicht mehr freigegeben werden. Auf dem Stack werden hauptsächlich Werte (Integer, Character, …) gespeichert. Auf dem Heap werden primär Verweistypen gespeichert (z. B. Arrays).

2.4.2 Parallelität

Parallele Datenverarbeitung lässt sich in 4 Ebenen gliedern. Nach der Granularität von fein- bis grobgranular sind dies die Instruktionsebene, Threadebene, Prozessebene und die Programmebene. Letztere umfasst das nebenläufige Ausführen von mehreren in sich abgeschlossenen, voneinander unabhängigen Programmen. Diese Art der Parallelität wird nicht näher betrachtet.

2.4.2.1 Parallelität auf Befehlsebene

(*Instruction-level parallelism (ILP)*) – Entsprechend der Befehlsausführungsschritte ist es möglich Pipelining (Fließbandverarbeitung) zur Leistungssteigerung zu verwenden. Jedem Schritt kann eine Stufe (*Stage*) zugeordnet werden, und durch die jeweilige Hardware-Komponente abgewickelt werden (siehe Abbildung 2.5).

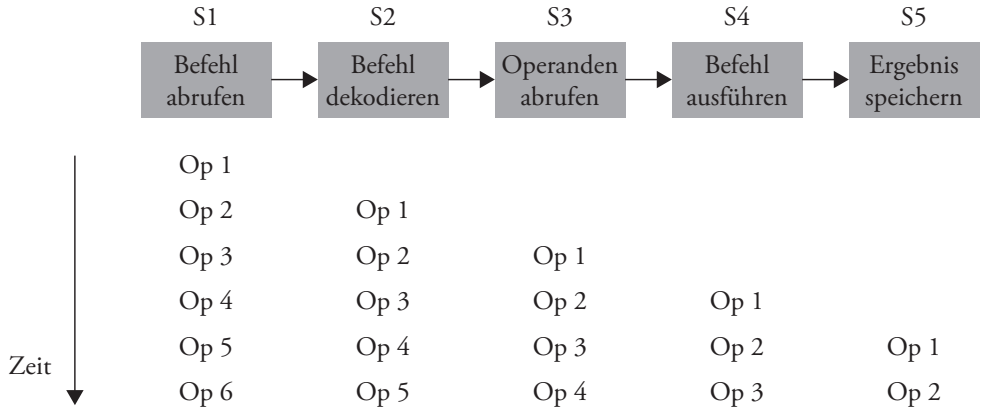


Abbildung 2.5: Eine fünfstufige Pipeline mit dem Status (Operationen Op 1–Op 6) jeder Stufe (S1–S5) über die Zeit

2.4.2.2 Parallelität auf Prozessorebene

Werden gleichzeitig gleichartige Operationen auf viele verschiedene Daten angewandt, können Feldrechner (*array processor*) eingesetzt werden. Sie bestehen aus vielen identischen Prozessoren, welche eine einheitliche Befehlsfolge abarbeiten.

Nach der Flynnschen Klassifikation (Abbildung 2.6) folgen derartige Computer dem SIMD (Single Instruction-Stream, Multiple Data-Stream)-Prinzip. Ein Feldrechner weist jedoch nur eine einzige Steuereinheit auf, so dass die Prozessoren nicht unabhängig voneinander sind.

Mehrere vollständige Prozessoren sind in Mehrprozessorsystemen (*Multiprocessor*) zu finden (MIMD-Prinzip; Multiple Instruction-Stream, Multiple Data-Stream). Je nach System gibt es entweder nur einen gemeinsamen Speicher für die gemeinsame Nutzung aller Prozessoren, oder zusätzlich je Prozessor lokalen Speicher. Im Gegensatz zu diesen eng gekoppelten Prozessoren (Kommunikation über gemeinsamen Speicher) gibt es auch lose gekoppelte Multicomputersysteme bestehend aus vielen separaten, über eine Kommunikationsstruktur verbundenen Computern (Nachrichtenaustausch, (*message passing*)), ohne gemeinsamen Speicher.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Abbildung 2.6: Flynnsche Klassifikation

Bei der Nutzung von Parallelität in Multicoresystemen gibt es zwei konkurrierende Ansätze, Threads und Prozesse. Ein Prozess besteht aus mehreren Threads. Ein Thread definiert einen Ausführungspfad, hat jedoch keine ihm exklusiv zugeordneten Ressourcen (Adressraum, Prozessoren oder weitere Betriebsmittel wie z. B. Netzwerkverbindungen). Prozessen hingegen sind eindeutig Ressourcen zugeordnet. Ein Thread gilt generell als

leichtgewichtiger als ein Prozess. Häufig wird der Begriff „Task“ verwendet, dieser Begriff ist jedoch nicht eindeutig definiert und wird sowohl als Synonym für Prozesse als auch für Threads eingesetzt.

2.5 LLVM

Die im weiteren Verlauf beschrieben Arbeiten basieren auf der LLVM Compiler-Infrastruktur, diese wird im Folgenden kurz als LLVM bezeichnet.

Ursprünglich wurde sie unter dem Namen *Low Level Virtual Machine* entwickelt. Der Name beruht auf dem Aufbau, bestehend aus einem virtuellen Befehlssatz und einer virtuellen Maschine, welche den Bitcode (Zwischencode) ausführen kann. Dem System liegt eine modulare Architektur zugrunde. Alle Phasen des Kompilationsprozesses können zur Optimierung verwendet werden. Der sprachunabhängige Instruktionssatz der Zwischensprache (LLVM Assembly Language) und das verwendete Typsystem ermöglicht den alternativen Einsatz verschiedener Frontends. So können Programme in z. B. C, C++, Java, Fortran, Haskell, Python, uvm. übersetzt werden. Ebenso werden durch Backends viele Prozessorarchitekturen, wie x86, AMD74, PowerPC, ARM, SPARC, Alpha, PIC16 MIPS, uvm. unterstützt. Die Zwischensprache setzt die in Kapitel 2.3.2.2 beschrieben statische Einzelzuweisungsform um. Da LLVM als Framework die Möglichkeit bietet beliebige Frontends und Backends zu kombinieren, bietet die Zwischensprache eine universelle Schnittstelle um Operationen auf dieser Basis durchführen zu können. Diese Darstellungsebene bietet zudem die Möglichkeit, das Programm in einer virtuellen Umgebung auszuführen.

Der Übersetzungsprozess kann entweder als statische Ahead-of-time- oder als Just-in-time-Compilierung durchgeführt werden. Im ersten Fall wird das Programm in nativer Maschinensprache vor der Laufzeit, im zweiten Fall während der Laufzeit erzeugt.

Die Zwischensprache verwendet Instruktionen, welche in [Pro15] klassifiziert werden. Terminator Instructions bezeichnen alle Instruktionen, mit denen ein Basisblock enden kann. Dies sind `ret` (return, Rücksprung zur aufrufenden Instanz, übergeordneten Funktion), `br` (branch, Sprung zu einem anderen Basisblock), `switch` (Sprung zu einem von mehreren Zielen), `indirectbr`, `invoke`, `resume`. Darüber hinaus sind binäre Operationen (add, sub, mul, etc.), bitweise binäre Operationen, Vektoroperationen (um Operationen auf Vektoren effizient auf unterschiedlichen Zielarchitekturen durchführen zu können), Speicherzugriffs- und Addressressierungsoperationen und Konvertierungsoperationen (cast) definiert. Unter „others“ werden `icmp` (Vergleich von Integerwerten, Integervektoren oder Zeigern, Rückgabe ist ein boolscher Wert), `phi` (Abbildung der ϕ-Knoten des SSA-Graphen) und `call` (Unterprogrammaufruf) zusammengefasst.

Kapitel 3

Programminterne Abhängigkeiten

Die Optimierungen, welche der Compiler durchführen kann, betreffen in der Regel lokale Änderungen, und können begrenzte Bereiche umfassen. Sollen Optimierungen in einem größeren Kontext untersucht werden, so sind Wirkzusammenhänge des Programms von elementarer Bedeutung. Die dabei resultierenden Zusammenhänge entsprechen in ihrer Struktur derer von gerichteten Graphen. Diese mathematische Abstraktion wird von der Graphentheorie abgedeckt. Eine Übersicht über die in dieser Arbeit verwendeten Eigenschaften und Begriffe findet sich zu Beginn dieses Kapitels. In 3.2 und 3.3 werden Kontroll- und Datenfluss eingeführt. Darüber hinaus werden weitere Modelle betrachtet. Diese ermöglichen sowohl Kontroll- als auch Datenabhängigkeiten in einem Modell abzubilden. Techniken um die zuvor beschriebenen Information aus Programmen zu extrahieren, werden ab 3.10 betrachtet. Hierbei können diese Informationen unterschiedlich große Programmeinheiten (Folgen von Befehlen) U betreffen. So können Zusammenhänge auf Funktionen (für jede Funktion ein Zähler / Timer beim Betreten), Knoten in Aufrufgraphen (Zähler oder Timer für jede Aufrufstelle), Basis-Blöcke (für jeden Basis-Block ein Zähler / Timer beim Betreten), Knoten in Kontrollflussgraphen (Zähler oder Timer für jeden Pfad zu einem Basis-Block), Anweisung oder Instruktionen bezogen werden.

3.1 Grundlagen Graphentheorie

Ein ungerichteter Graph G (vgl. Abbildung 3.1a)) besteht aus Knoten (*vertices*) v_i und Kanten (*edges*) e_j ($G = (V, E)$ mit $e \in E$ und $v \in V$). Jede Kante wird als Verbindung zwei Knoten zugeordnet ($e = \{v_n, v_m\}$), und kann in beide Richtungen durchlaufen werden. In gerichteten Graphen (vgl. Abbildung 3.1b)) ist dies nicht möglich. Jede gerichtete Kante (auch als Bogen bezeichnet) hat ein eindeutiges Paar Knoten, wobei es einen Anfangsknoten und einen Endknoten gibt, jedem Bogen wird somit ein geordnetes Paar Knoten aus V zugeordnet ($e = (v_n, v_m)$ mit v_n als Anfangs-

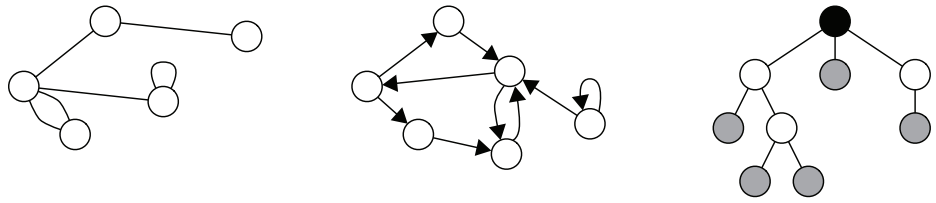


Abbildung 3.1: Beispiele für einfache Graphen: a) ungerichtet, b) gerichtet, c) Baum (Blätter grau; Wurzel schwarz)

knoten und v_m als Endknoten). In der grafischen Repräsentation wird die Richtung (Orientierung) der Kante durch einen Pfeil dargestellt. Der Grad deg v eines Knoten $v \in V$ gibt die Anzahl der von dem Knoten v ausgehenden Kanten an. Eine Kantenfolge $v_1, e_1, v_2, e_2, \ldots v_{k-1}, e_{k-1}, v_k$ ist eine Folge von Knoten und Kanten, wobei der Endknoten einer Kante immer zugleich den Startknoten der nächsten Kante darstellt (außer beim Endknoten der Folge). Die Länge der Kantenfolge ist die Anzahl der Kanten der Folge. Ein Weg p ist eine Kantenfolge innerhalb eines Graphen, wenn jeder Knoten des Graphen $V(G)$ höchstens einmal in der Folge auftritt. Ist der Start- und Endknoten eines Weges identisch ($v_1 = v_k$), so erhält man einen Kreis. Ein Baum (*tree*) T ist ein zusammenhängender, kreisfreier Graph (vgl. Abbildung 3.1c)) . Die Knoten mit Grad 1 heißen Blätter (*leaf*) l, die übrigen Knoten heißen innere Knoten. Für jeden Knoten v kann seine Tiefe als die Länge des Weges von der Wurzel bis zu v angegeben werden. Die Höhe eines Baumes entspricht der maximalen Tiefe, die Breite der maximalen Anzahl Knoten mit identischer Tiefe.

Zwei Knoten oder zwei Kanten heißen adjazent bzw. benachbart, wenn sie über eine Kante oder einen Knoten miteinander verbunden sind. Eine Adjazenzmatrix beinhaltet alle Knoten und markiert für jeden Knoten in der zugehörigen Zeile welche anderen Knoten erreichbar sind. Eine andere Repräsentation eines Graphen ist über die Inzidenzmatrix möglich. Sie bildet die Beziehung zwischen Knoten und Kanten ab. Eine gerichtete Kante ist positiv inzident zu dem Startknoten. In der Inzidenzmatrix bildet jede Zeile einen Knoten, jede Spalte eine Kante ab. Der Startknoten wird mit dem Wert -1, der Endknoten mit 1 eingetragen.

Ein ungerichteter Graph ist zusammenhängend, wenn für je zwei beliebige Knoten ein Weg mit diesen Knoten als Start- und Endknoten existiert. Ein gerichteter Graph ist stark zusammenhängend, wenn es von jedem Knoten einen Weg zu allen anderen Knoten gibt. Eine Komponente K ist ein zusammenhängender Teil eines Graphen. Es muss daher gelten, dass es für je zwei beliebige Knoten $k, l \in K$ einen Weg von k nach l gibt der nur über Knoten von K führt. In gerichteten Graphen ist eine starke Zusammenhangskomponente (*strongly connected component (SCC)*) definiert als ein maximaler stark zusammenhängender Teilgraph des Graphen (vgl. Abbildung 3.2). [Tit03]

Abbildung 3.2: Beispiele für Graphen: a) nicht zusammenhängender Graph, b) starke Zusammenhangskomponente (grau)

3.2 Kontrollfluss

Der Kontrollfluss CF eines Programms bezeichnet die zeitliche Abfolge der einzelnen Instruktionen eines Programms. Es ist möglich den Kontrollfluss auf verschiedenen Abstraktionsebenen zu betrachten. Häufig wird jede Instruktion im gegebenen Programm als ein Knoten dargestellt. Abfolgen entsprechen in ihrer Struktur gerichteten Graphen. In Kontrollflussgraphen (*control flow graph (CFG)*) werden Programmkomponenten als Knoten dargestellt. Gerichtete Kanten stellen den Ablauf bzw. Übergang zwischen den Komponenten dar. Eine Kante von Knoten U_1 zu U_2 existiert, wenn U_2 in einer Ausführungssequenz direkt auf U_1 folgt ($U_1 \rightarrow_f U_2$). U_1 ist der Vorgänger von U_2, bzw. U_2 der Nachfolger von U_1. Der Kontrollflussgraph erfasst alle möglichen Abfolgen innerhalb des Programms. Er ist definiert als ($G_{CF} = (V, E)$ mit E als Menge aller Kontrollflusskanten und V als Menge aller Programmeinheiten). Wenn ein Knoten mehrere Nachfolger besitzt, handelt es sich um eine Verzweigung. Der Kontrollfluss verläuft in Abhängigkeit vom Programmzustand alternativ auf einem der ausgehenden Pfade. Der aktive Pfad hängt von einer boolsche Bedingung (Prädikat) ab.

Programmiersprachenkonstrukte wie while-, do-while- oder for-Schleifen führen zu Schleifen im Kontrollfluss. Eine Knoten-Menge L bildet eine Schleife im Kontrollflussgraph, wenn zwei Bedingungen erfüllt sind. Erstens: es gibt nur einen Vorgänger-Knoten außerhalb von L, dieser Knoten ist der Schleifeneingang ENTRY. Zweitens: Von jedem Knoten gibt es einen Weg zu ENTRY der nur Knoten der Menge L enthält.

3.2.1 Kontrollabhängigkeit

Eine Kontrollabhängigkeit liegt dann vor, wenn es eine Bedingung $cond$ gibt, von der abhängt, ob ein anderer Programmteil U ausgeführt wird, oder nicht. Wesentlich ist es den Ort im Programm zu identifizieren, der über die Ausführung einer anderen Stelle entscheidet. In der Literatur werden verscheidene Arten der Kontrollabhängigkeit behandelt. Zum einen unterscheiden sie sich in den Kriterien, welche für die Definition zugrunde gelegt werden. Zum anderen setzen einige Definitionen bestimmte Eigenschaften, etwa des Kontrollflussgraphen, voraus.

3.2.1.1 Dominanz

Die am weitesten verbreitete Betrachtung von Kontrollabhängigkeiten basiert auf dem Begriff der Dominanz. Ein Knoten des Kontrollflussgraphen stellt den Startkonten bzw. die Wurzel dar, wenn sein Anfang die erste Instruktion des Programms ist. Ein Knoten U_1 dominiert einen Knoten U_2, wenn U_1 ein Teil jeden Pfades von der Wurzel zu U_2 ist. U_1 ist der Dominator von U_2 ($U_1 \operatorname{dom} U_2$). Jeder Knoten wird von dem Wurzelknoten dominiert, hat also mindestens einen Dominator. Zudem dominiert sich auch jeder Knoten selbst. Wenn $U_1 \operatorname{dom} U_2$ und $U_2 \operatorname{dom} U_3$, dann gilt auch $U_1 \operatorname{dom} U_3$. Eine strikte Dominanz liegt vor, wenn $U_1 \neq U_2$. Der letzte Dominator, welcher auf allen Pfaden von der Wurzel zu U_n liegt wird als direkter Dominator bezeichnet ($\operatorname{idom}(U_n)$). Für jeden Knoten existiert immer ein eindeutiger direkter Dominator. Eine Dominanz ist stark, wenn alle Pfade einer Länge $\geq k$, mit k konstant, von U_1 auch U_2 erreichen. (vgl. [CFR+91])

Dominanzbaum Bildet man einen Graphen mit allen Knoten des Kontrollflussgraphen und Kanten von dem direkten Dominator eines Knoten U_i zu eben diesem Knoten ($\operatorname{idom}(U_i) \rightarrow U_i$), so ergibt sich ein Baum, der Dominanzbaum. Die Wurzel des Dominanzbaumes entspricht der Wurzel des Kontrollflussgraphen. Der Pfad von der Wurzel zu einem Knoten U_j enthält alle Dominatoren von U_j. Ebenso beinhaltet der Teilbaum, welcher unterhalb von U_j liegt alle von U_j dominierten Knoten. Abbildung 3.3 zeigt einen Kontrollflussgraphen und den daraus abgeleiteten Dominanzbaum.

Dominanzgrenze Die Dominanzgrenze (*dominance frontier*) für einen Knoten U_n wird mit $DF(U_n)$ bezeichnet. Sie umfasst alle Knoten U_m die nicht mehr von U_n dominiert werden, von denen U_n aber mindestens einen Vorgänger dominiert. Die Dominanzgrenze enthält Knoten mit mehr als einem Vorgänger, also Knoten an denen der Kontrollfluss von verschiedenen Quellen zusammenläuft.

Postdominanz Die Postdominanz liefert die Information, welche Knoten auf jeden Fall noch durchlaufen werden müssen, bevor das Programm terminieren kann. Für die Definition der Postdominanz wird vorausgesetzt, dass es im Kontrollflussgraphen einen Endknoten, also genau einen Knoten ohne Kontrollflussnachfolger gibt. Ein Knoten U_y postdominiert den Knoten U_x, wenn alle Pfade zum Endknoten über den Knoten U_y führen. Für die Postdominanz kann äquivalent zu dem Dominanzbaum ein Postdominanzbaum gebildet werden, wenn im Kontrollflussgraphen die Kantenrichtungen umgedreht werden.

3.2.1.2 Klassische Kontrollabhängigkeit

Diese Definition der Kontrollabhängigkeit ist weit verbreitet. Eine Definition der Kontrollabhängigkeit über Postdominanz, wie in diesem Fall, setzt einen einzigen Endknoten des Kontrollflussgraphen voraus. Ein Knoten U_n ist kontrollabhängig von einem Knoten U_m ($U_m \overset{cd}{\rightarrow} U_n$), wenn es im Kontrollflussgraphen einen Pfad von U_m zu U_n gibt, so

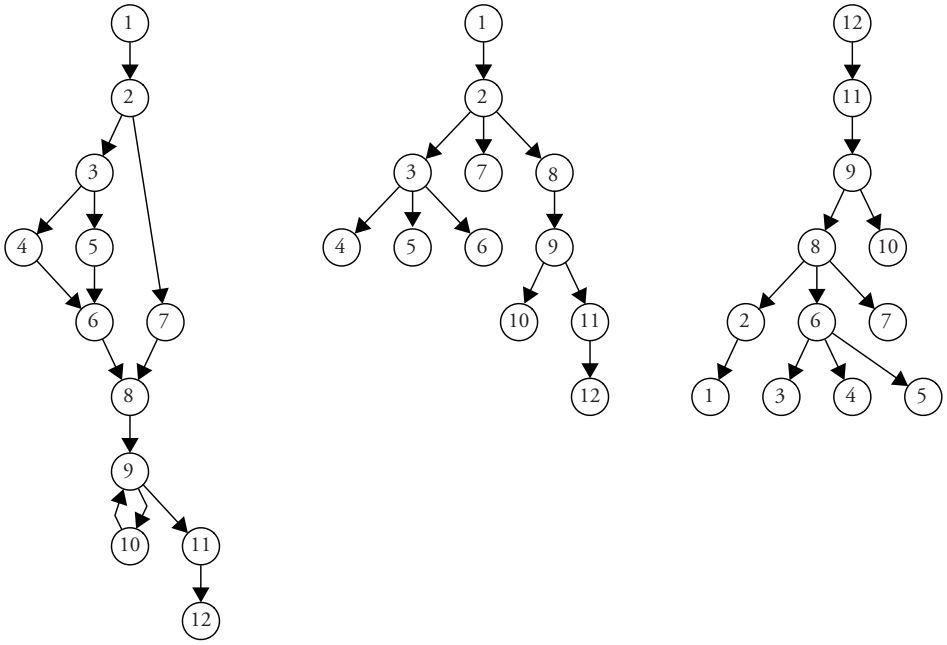


Abbildung 3.3: Kontrollflussgraph (a), Dominanzbaum (b) und Postdominanzbaum (c) eines hypothetischen Programms

dass jeder Knoten auf dem Pfad von U_n postdominiert wird und U_m nicht von U_n strikt postdominiert wird.

Es muss also einen Pfad von U_m zum Endknoten geben, der nicht über U_n führt. Zudem muss es einen Pfad von U_m zu U_n geben, auf dem alle Knoten außer U_n und U_m von U_n postdominiert werden. Nach der Definition ist es möglich, dass der Pfad von U_m zu U_n eine Schleife enthält. Auch ist es möglich, dass ein Knoten von sich selbst kontrollabhängig ist. Wenn $U_1 \overset{cd}{\rightarrow} U_2$ und $U_2 \overset{cd}{\rightarrow} U_3$, dann gilt laut dieser Definition nicht $U_1 \overset{cd}{\rightarrow} U_3$, da U_3 U_1 nicht postdominiert. Dennoch existiert eine indirekte Beziehung, denn U_1 spielt eine Rolle bei der Frage ob, U_3 ausgeführt wird oder nicht.

Bei einer fußgesteuerten Schleife liegt die Entscheidung, ob zurück zum Anfang der Schleife (Knoten a) gesprungen wird oder nicht, am Ende (Knoten e). Da der auf e folgende Knoten f (nicht a) der Postdominator von e ist, wird nach der oben genannten Definition ($e \overset{cd}{\rightarrow} f$) nicht gelten, obwohl e darüber entscheidet, ob f ausgeführt wird oder nicht. Eine Schleife kann theoretisch eine Endlosschleife sein. Andere Definitionen, wie sie im Folgenden behandelt werden, versuchen dies abzudecken.

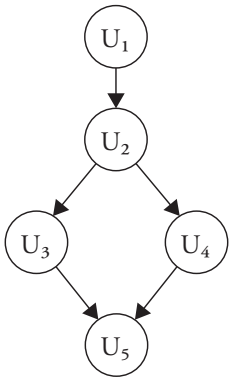


Abbildung 3.4: Beispiel-Kontrollflussgraph mit einer Verzweigung

3.2.1.3 Starke Kontrollabhängigkeit

Für zwei Knoten U_m und U_n eines Kontrollflussgraphen gilt, dass U_n von U_m stark kontrollabhängig ist ($U_m \overset{scd}{\rightarrow} U_n$), falls es einen Pfad von U_m nach U_n gibt, und dieser den direkten Postdominator von U_m nicht enthält.

Bei Programmverzweigungen entspricht die Menge der kontrollabhängigen Knoten dem Verzweigungs-Körper. Bei Schleifen ist es ebenfalls der Schleifenkörper und auch der Schleifenkopf. Alle Knoten die auf dem direkten Pfad zwischen dem Kontrollknoten und seinem direkten Postdominator liegen, sind von diesem Kontrollknoten kontrollabhängig.

Abbildung 3.4 zeigt einen Kontrollflussgraphen mit einer Verzweigung. Der direkte Postdominator von U_2 ist U_5, damit sind U_3 und U_4 stark kontrollabhängig von U_2. [PC90]

3.2.1.4 Schwache Kontrollabhängigkeit

Die schwache Kontrollabhängigkeit ist eine Verallgemeinerung der starken Kontrollabhängigkeit. Eine Knoten U_n ist von einem anderen Knoten U_m schwach kontrollabhängig ($U_m \overset{wcd}{\rightarrow} U_n$), wenn:

- U_m zwei Nachfolger U'_m und U''_m auf alternativen Pfaden hat
- U_n innerhalb einer beschränkten Anzahl von Anweisungen ausgeführt wird wenn der Pfad über U'_m gewählt wurde
- U_n gar nicht oder erst sehr viel später ausgeführt wird falls der Pfad über U''_m betreten wurde.

Anders formuliert ist U_n schwach kontrollabhängig von U_m, wenn U_n einen Nachfolger von U_m stark postdominiert, aber einen anderen Nachfolger von U_m nicht. [RAB+05; PC90]

U_n ist dann schwach kontrollabhängig von U_m, wenn es einen Pfad gibt, auf dem U_m der erste Knoten, und U_n der letzte Knoten ist, und alle anderen Knoten jeweils von ihrem Nachfolger direkt schwach kontrollabhängig sind. [PC90]

3.2.1.5 Nichtterminierende sensitive Kontrollabhängigkeit

(*non-termination sensitive control dependency*) – Die vorherigen Definitionen der Kontrollabhängigkeit setzen einen einzigen Endknoten des Kontrollflussgraphen voraus. In Programmsystemen können jedoch mehrere Endknoten oder auch keine Endknoten (Endlosschleife) vorhanden sein. Daher beziehen sich die beiden in [RAB+05] eingeführten Definitionen auf Pfade bzw. Pfadsegmente, besonders auf Pfadsegmente welche auf beiden Seiten durch einen Knoten U_m begrenzt werden. Hierzu wird ein Pfad als maximal definiert, wenn er unendlich ist, oder in einem Endknoten endet.

Eine Kontrollabhängigkeit $U_m \overset{ntscd}{\rightarrow} U_n$ liegt vor, wenn U_m mindestens zwei Nachfolger U_u und U_v hat, und zwei Bedingungen gelten. Erstens, dass der Knoten U_n in allen maximalen Pfaden die von U_u ausgehen liegt, und der Knoten U_n auf diesem Pfad immer vor dem Knoten U_m liegt. Zweitens, dass es einen maximalen Pfad von U_v ausgehend gibt, auf dem der Knoten U_n nicht liegt.

3.2.1.6 Nichtterminierende insensitive Kontrollabhängigkeit

(*non-termination insensitively control dependent*) – Die vorherige Definition betrachtet alle Pfade, welche von einer Verzweigung ausgehen. Bei der folgenden Definition einer Kontrollabhängigkeit wird dies auf endliche Pfade, die einen End-Bereich erreichen, beschränkt. Da es nicht einen End-Knoten gibt, wird eine Kontrollflusssenke eingeführt. Diese Senke ist eine Menge von Knoten für die gilt, dass jeder Knoten dieser Menge von jedem anderen Knoten der Menge erreicht werden kann und keine Kontrollflusskante aus dieser Menge herausführt. Jeder Knoten der Senke hat also ausschließlich Nachfolgeknoten die ebenfalls Teil der Senke sind. Ein Menge Senkenpfade besteht dann aus allen Pfaden, die von einem Knoten U_u zu einem Knoten führen, welcher Teil einer Senke ist. Wie zuvor wird eine Kontrollabhängigkeit ($U_m \overset{nticd}{\rightarrow} U_n$) für den Fall definiert, dass U_m zwei Nachfolger U_u und U_v besitzt. Die Abhängigkeit existiert, wenn alle Sinkpfade von U_u aus U_n beinhalten und es von U_v aus einen Sinkpfad gibt, der U_n nicht beinhaltet. [RAB+05]

3.2.2 Reduzierbarkeit

Ein wichtiges Konstrukt in Programmen sind Schleifen. In Kontrollflussgraphen stellen sie sich als Kreis dar. Im Folgenden wird die Unterscheidung zwischen Vorwärtskanten und Rückwärtskanten benötigt. Eine Rückwärtskante führt den Kontrollfluss auf einen Knoten zurück, welcher bereits durchlaufen wurde. Eine Kante von $U_x \rightarrow U_y$ ist rückwärts gerichtet, wenn $U_y \, dom \, U_x$. Schleifen besitzen in der Regel einen Schleifenkopf, der den Eintrittspunkt in die Schleife darstellt. Der Kopf dominiert alle Knoten der Schleife (Rumpf). Die Rückwärtskante führt zum Kopf zurück. Schleifen können ineinander verschachtelt sein. Diejenige Schleife, welche keine weiteren Schleifen enthält, wird als innere Schleife, diejenige welche nicht im Rumpf einer anderen liegt, als äußere Schleife bezeichnet.

Die meisten realen Programme führen zu reduzierbaren Flussgraphen. Ein Flussgraph heißt reduzierbar, wenn die Kanten in zwei disjunkte Mengen, Vorwärtskanten und Rückwärtskanten, aufgeteilt werden können. Die Vorwärtskanten bilden einen azyklischen Graphen, in dem jeder Knoten vom Startknoten aus erreichbar ist. ([ULS+08])

3.2.3 Definitions-Reihenfolge-Abhängigkeit

Eine Definitions-Reihenfolge-Abhängigkeit (*def order dependency*) ($U_1 \rightarrow_{do} U_2$) gibt an, dass zwei Instruktionen oder Programmeinheiten in ihrer Ausführungsreihenfolge nicht geändert werden dürfen. Eine geänderte Ausführungsreihenfolge hätte eine geänderte Programmlogik zur Folge.

Beispiel: `if P then x = 0; if Q then x = 1;`

Definitions-Reihenfolge-Abhängigkeiten zwischen zwei Knoten v1 und v2 existieren, wenn vier Bedingungen erfüllt sind.

1. Beide Knoten definieren die selbe Variable
2. Beide Knoten befinden sich im selben Zweig einer bedingten Verzweigung
3. Es existiert ein dritter Knoten, der sowohl von dem ersten als auch von dem zweiten Knoten datenabhängig ist
4. v1 steht im abstrakten Syntaxbaum links von v2 [HPR89]

3.3 Datenfluss

Der Datenfluss DF innerhalb eines Programms lässt sich auf verschiedene Weise darstellen. Eine gebräuchliche Form der Darstellung sind Datenflussgraphen ($G_{DF} = (V, E)$ mit E als Menge aller Datenflusskanten und V als Menge aller Programmeinheiten). Es handelt sich also um gerichtete Graphen mit Instruktionen oder Programmteilen als Knoten. Die gerichteten Kanten stellen den Datenfluss von Produzent zu Konsument dar.

Optimierungen auf Instruktionsebene führen teils zu einer Neuordnung der einzelnen Instruktionen. Die Ausführungsreihenfolge ist jedoch nicht immer änderbar ohne die Semantik des Programms zu verändern.

In dem Code-Beispiel in Listing 3.1 wird offensichtlich der Wert, welcher in der ersten Zeile (U_1) definiert wurde, in der zweiten Zeile (U_2) referenziert. U_2 hängt also von U_1 ab ($U_1\ \delta\ U_2$). Folglich muss U_1 vor U_2 ausgeführt werden.

[KKP+81] definiert die im Folgenden aufgeführten Abhängigkeiten. Eine Instruktion ist schleifenabhängig (δ^L) von dem Schleifenkopf, wenn sie zum Schleifenkörper

```
1 x := ...
2 ... := x ...
```

Listing 3.1: Zwei beispielhafte Instruktionen. Das Ergebnis der ersten Zeile wird in Zeile zwei referenziert.

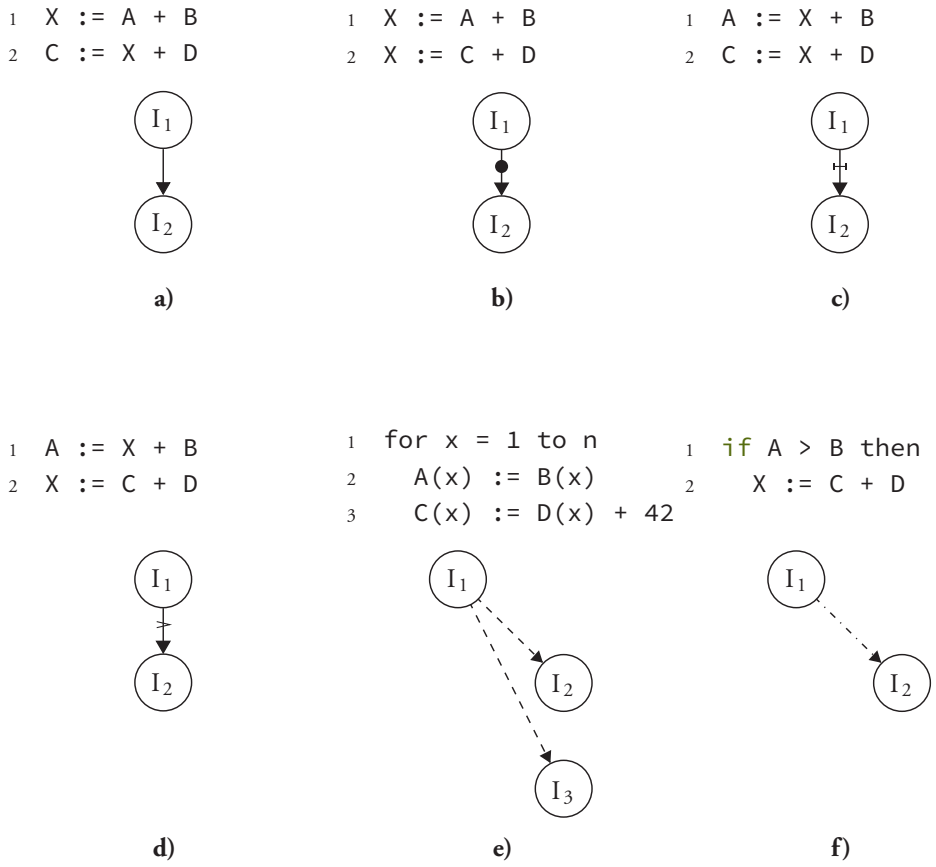


Abbildung 3.5: Abhängigkeiten zwischen Instruktionen: a) Echte Datenabhängigkeit, b) Ausgabeabhängigkeit, c) Eingabeabhängigkeit, d) Antiabhängigkeit, e) Schleifenabhängigkeit, f) Kontrollabhängigkeit

gehört. Der Datenfluss ist schleifengetragen (*loop carried*) $(v1 \rightarrow_l c(L)v2)$ wenn es einen Datenfluss von Knoten v1 nach Knoten v2 gibt, und es einen Ausführungspfad gibt, der eine rückgerichtete Kange (*Backedge*) zum Prädikat L der Schleife gibt, und sowohl v1 als auch v2 innerhalb der Schleife liegen.

Für zwei beliebige Instruktionen U_1 und U_2, gibt es eine Abhängigkeit, wenn U_2 nach U_1 ausgeführt wird und falls:

- U_2 das Ergebnis von U_1 verwendet. (Echte Abhängigkeit $U_1\ \delta\ U_2$)
- U_1 eine Variable referenziert, die von U_2 anschließend definiert wird. (Anti- oder umgekehrte Abhängigkeit $U_1\ \delta^{-1}\ U_2$)
- U_2 die gleiche Variable definiert wie U_1. (Ausgabe-Abhängigkeit $U_1\ \delta^O\ U_2$)
- U_1 und U_2 die gleiche Variable referenzieren. (Eingabe-Abhängigkeit $U_1\ \delta^I\ U_2$)

Exemplarische Instruktionsfolgen und eine graphische Repräsentation der sich dabei ergebenen Abhängigkeiten sind in Abbildung 3.5 dargestellt.

Wird ein Programm ausgeführt, so sind alle Daten zu Beginn undefiniert. Am En-

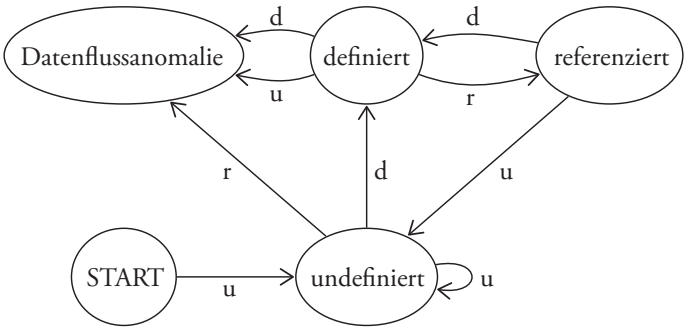


Abbildung 3.6: Zustände von Datenobjekt und Übergänge durch Zugriffe

de einer vollständigen Ausführung sollten keine Objekte mehr im Speicher verbleiben. Andernfalls bleibt dieser Speicher belegt, kann vom Betriebssystem nicht wieder freigegeben werden. Derartige Speicherlecks (*memory leak*) sind zu vermeiden. Am Ende der Ausführung werden alle Objekte wieder zerstört, somit sind alle Daten wie zu Beginn undefiniert. Mit Programmstart werden benötigte Datenobjekte deklariert und im Speicher angelegt, während der Abarbeitung kommen weitere, lokale Variablen, hinzu.

Im Rahmen eines Programms werden Sequenzen von Zugriffen auf Datenobjekte ausgeführt. Hierbei kann ein Objekt durch eine Wertzuweisung definiert werden (d), gelesen bzw. referenziert (r) oder undefiniert (u) (zerstört) werden. Mögliche Zustände und die durch Verwendung im Programm auftretenden Zustandsübergänge sind in Abbildung 3.6 gezeigt. Es können Datenanomalien auftreten, wenn gewisse Sequenzen auftreten. Eine normale Verwendung wäre beispielsweise die Sequenz udr. Wohingegen bei dd die erste Definition überschrieben wird und somit keine Auswirkung auf das Programm hat. du ohne einen referenzierenden Zugriff hat ebenso keine Auswirkung. ur führt zum Lesen nicht definierter, und somit unvorhersagbarer Werte. [Lig09]

Bei lesenden Zugriffen kann zwischen einem Zugriff im Rahmen einer Berechnung (*computation*) r_c oder im Prädikat einer Entscheidung (*predicate*) r_p unterschieden werden.

Abbildung 3.6 zeigt einen Zustandsautomaten mit den möglichen Zuständen von Variablen und die Übergänge durch Verwendung im Programm. Mit dem Programmstart werden Variablen zuerst in den Zustand undefiniert übergehen. Vor ihrer Verwendung, müssen sie zunächst definiert werden. Die korrekte Verwendung ist gegeben, wenn auf eine Definition eine oder mehrere Referenzierungen folgen. Diese Sequenz kann sich wiederholen, am Ende der Ausführung steht dann wieder der undefinierte Zustand. Wird eine undefinierte Variable referenziert, führt dies zu der ur-Anomalie (unvorhersagbare Werte werden gelesen). Die dd-Anomalie beschreibt zwei aufeinander folgende Definitionen ohne zwischenzeitliche Referenzierung. Ein Wert, welcher durch die erste Definition gespeichert wurde, wird nicht mehr gelesen, geht somit verloren. Ähnlich verhält sich das System bei der Folge du.

3.4 Kontroll- und Datenflussgraphen

Ein reiner Datenflussgraph (*data flow graph (DFG)*) bildet ausschließlich die Datenabhängigkeiten zwischen Programmkomponenten ab. Somit werden Schleifen und damit verbundene Iterationen oder Verzweigungen im Kontrollfluss nicht abgebildet. Um diese, im Kontrollflussgraphen abgebildeten Informationen, zu berücksichtigen muss ein heterogenes Modell gewählt werden, welches sowohl Kontroll- als auch Datenabhängigkeiten umfasst. Es ergibt sich ein kombinierter Kontroll- und Datenflussgraph (*control and data flow graph (CDFG)*)

3.5 Aufrufgraphen

Der Aufrufgraph (*call graph*) beschreibt welche Unterprogramme welche anderen aufrufen können. Dabei stellt jedes Unterprogramm einen Knoten in dem Graph dar. Jeder Knoten enthält eine indizierte Menge aller Stellen, an denen ein Unterprogramm aufgerufen wird (Aufrufstelle (*call site, caller*)). Wenn eine Aufrufstelle ein Unterprogramm aufruft, so gibt es in dem Aufrufgraph eine gerichtete Kante von der Aufrufstelle zu dem aufgerufenen Unterprogramm (*callee*). Häufig ist anhand des Quelltextes klar, welches Unterprogramm an welcher Stelle aufgerufen wird. Kommen jedoch z. B. Funktionszeiger zum Einsatz, kann die Kante erst zur Laufzeit ermittelt werden. Von einer Aufrufstelle können viele Kanten zu mehreren Unterprogrammen auftreten. In objektorientierten Sprachen sind derartige indirekte Aufrufe die Regel. Durch Konzepte wie Vererbung und z. B. das Überschreiben von Funktionen ist die statische Bestimmung der tatsächlich aufgerufenen Methode nicht möglich. (vgl. [ULS+08])

3.6 Synchrone Datenflussgraphen

Unter synchronen Datenflussgraphen versteht man Datenflussgraphen wie sie zuvor eingeführt wurden, ergänzt um Kantengewichte. Am Anfang einer Kante wird der Wert $prod$ am Ende der Wert $cons$ notiert. Dabei gibt $prod$ die Anzahl der erzeugten Daten, $cons$ die Anzahl der konsumierten Daten an. Die Synchronizität bezieht sich auf die Tatsache, dass diese Werte konstant sind. [TH07, Seite 56 - 60]

3.7 Dynamische Datenflussmodelle

Um auch nicht konstante Datenmengen abbilden zu können, muss das vorherige Modell (Abschnitt 3.6) ergänzt werden. In [BL93] werden daher Knoten mit bekannter und konstanter Menge konsumierter Daten als reguläre Aktoren bezeichnet. In dem Fall, dass die Anzahl nicht konstant ist, gelten die Knoten als dynamische Aktoren. Ein Graph mit dynamischen Aktoren ist somit ein dynamischer Datenflussgraph. Buck erweitert in [BL93] die synchronen Datenflussgraphen um zwei dynamische Aktoren „SWITCH“

und „SELECT“. Diese Knoten sind in ihrem Verhalten abhängig von einem Steuereingang. Somit lassen sich Schleifen und Verzweigungen des Kontrollflusses in diesem Datenfluss-basierten Modell abbilden.

Das beschriebene Modell kann als ein Spezialfall der Kahn-Process-Networks (KPN, siehe [Kah74]) angesehen werden, daher ist es deterministisch. KPNs sind ein Rechenmodell bestehend aus einer Gruppe deterministischer Prozesse. Diese beinhalten sequentielle Abläufe. Untereinander sind sie über Kommunikationskanäle verbunden. Diese beinhalten einen FIFO-Speicher (First In – First Out). Lesezugriffe sind blockierend, der Ablauf stagniert bis die Leseoperation erfolgreich durchgeführt wurde. Schreibzugriffe hingegen sind nicht-blockierend. Ein Prozess kann somit zwei Zustände aufweisen. Entweder er ist aktiv, also prozessiert Daten oder schreibt Ergebnisse auf den Ausgabekanal, oder wartend, er ist dann blockiert bis die Eingangsdaten vorliegen.

Ein weiteres Modell wird in [BB01] vorgestellt. Diesen beinhaltet ebenfalls die Möglichkeit, datenabhängige Aktivierungen von Datenflüssen abzubilden. Hierzu werden hierarchische und parametrisierte Subgraphen verwendet.

Diese Modelle ermöglichen flussbasierte Systeme zu modellieren. Der reguläre Ansatz zielt darauf ab, ausführbare Programme aus dem Modell zu erstellen.

Flussbasiertes Programmieren (*flow-based programming (FBP)*) ist ein Programmierparadigma, welches Anwendungen als Netzwerk mit einzelnen über Nachrichtenaustausch kommunizierenden Prozessen definiert. Die Prozesse werden dabei unabhängig von dem Netzwerk definiert, so können sie in anderen Kontexten wiederverwendet werden. [Mor13]

3.8 Sequenzgraph

Ein Sequenzgraph besteht aus gerichteten Graphen auf mindestens zwei Hierarchieebenen. Die oberste Ebene entspricht einem Kontrollflussgraphen. Jedem Knoten auf dieser Ebene ist ein azyklischer und polarer Datenflussgraph zugeordnet. Es gibt für jeden Datenflussgraphen auf dieser Hierarchieebenen einen Start- und einen End-Knoten. Diese beiden Knoten sind Hierarchieknoten und dienen ausschließlich der Verbindung zwischen den Ebenen. Als weitere Hierarchieknoten werden in Sequenzgraphen Aufrufe (CALL), Verzweigungen (BRANCH) und Iterationen (LOOP) definiert. Die Blätter in der Hierarchie besitzen außer dem Start- und Endkonten keinen dieser Hierarchieknoten. [TH07]

3.9 Abhängigkeitsgraph

Ein Programmabhängigkeitsgraph bildet die Abhängigkeiten zwischen Anweisungen, repräsentiert durch Knoten, innerhalb einer Prozedur ab. Um interprozedurale Abhängigkeiten abbilden zu können, werden mehrere PDGs zu einem System-Dependence-Graph (SDG) kombiniert.

3.9.1 Programmabhängigkeitsgraph

Der Programmabhängigkeitsgraph (*Program dependence graph (PDG)*) ist ein gerichteter Multigraph, um die Kontroll- und Datenabhängigkeiten innerhalb einer Funktion abzubilden. Es gibt verschiedene Varianten der Definition, je nach Anwendungsgebiet, mit der Gemeinsamkeit die Kontroll- und Datenabhängigkeiten abzubilden. Das Programm P (ohne Strukturierung mittels Unterprogrammen) wird auf den Programmabhängigkeitsgraphen $G_P = (V, E)$ abgebildet ($P \rightarrow G_P$). Anweisungen bilden die Menge der Knoten V des Graphen. Dies können Zuweisungen oder Prädikate sein, zusätzlich existiert für jede Variable ein Knoten mit der initialen Definition. Abhängigkeiten bilden die Menge der Kanten E. Die Menge aller Kanten teilt sich in zwei Untermengen auf. Ein Teilgraph repräsentiert die Kontrollabhängigkeiten (Kantenmenge E_C) ein zweiter die Datenabhängigkeiten (Kantenmenge E_D). Die Erzeugung eines PDG wird in [HR92] für eine beschränkte Sprache, welche skalare Variablen, Zuweisungen, if-then-else-Anweisungen, Ausgaben und while-Schleifen enthalten kann, beschrieben.

Knoten lassen sich in solche untergliedern, die Zuweisungen repräsentieren und solche, die Prädikate abbilden. Kanten die von Prädikatsknoten ausgehen, werden mit dem zugehörigen Entscheidungswert ($true$ oder $false$) beschriftet. Neben den Knoten für Anweisungen existiert ein zusätzlicher Startknoten (*entry node*) mit Kanten zu Top-Level-Anweisungen. Kontrollabhängigkeitskanten (*control edge*) führen von dem Prädikat zu allen davon abhängigen Knoten. Eine solche Kante geht entweder von dem Startknoten oder einem Prädikatsknoten aus. Ein ϕ-Knoten der SSA-Form wird als Zuweisungsknoten abgebildet.

Region-Nodes können genutzt werden, um Kontrollabhängigkeiten von einem Prädikat zu gruppieren. Wenn die Bedingung erfüllt ist, können alle Kindknoten (datenabhängige, nicht kontrollabhängige) ausgeführt werden. Die Ausführung dieser Kindknoten kann parallel geschehen.

Datenabhängigkeitskanten lassen sich wiederum in zwei Gruppen unterteilen. Zum einen bildet der Datenfluss (echte Datenabhängigkeit) Definitions-Reihenfolge-Abhängigkeiten ab, also den Fluss einer Information von deren Produzent zu deren nächsten Konsument im Programmablauf. Dieser Datenfluss lässt sich als schleifengetragen oder Schleifen-unabhängig bezeichnen. Wenn es einen Datenfluss von x nach y gibt, und sowohl x als auch y innerhalb eines Schleifenkörpers mit dem Prädikat p liegen, dann ist dieser Datenfluss schleifengetragen falls es einen direkten Kontrollflusspfad von x nach y gibt, welcher eine rückgerichtete Kante zu p enthält. Zum anderen enthält der PDG Kanten für Def-Order-Abhängigkeiten. [HR92], [HRB88]

3.9.2 Systemabhängigkeitsgraph

Im Gegensatz zu dem Programmabhängigkeitsgraph, stellt der Systemabhängigkeitsgraph (*System dependence graph (SDG)*) globale Abhängigkeiten für interprozedurale Programme dar. Da PDGs nur für Programme mit einer Prozedur, also ohne Unterprogrammaufrufe existieren, werden im SDG mehrere PDGs für verschiedene Unterprogramme über interprozedurale Kontroll- und Datenflusskanten vernetzt. Mittels des SDG können

Fragmente (z. B. Program-Slices, siehe Kapitel 5.2.6) auch für interprozedurale Programme gebildet werden. Jedes Programm besteht dazu aus eine PDG für die Main-Prozedur und PDGs für alle weiteren aufgerufenen Prozeduren. Inkonsistenzen können entstehen, wenn Schreiboperationen Daten im Cache ablegen. Bevor Leseoperationen auf die Speicherstelle ausgeführt werden können muss der Wert zurückkopiert werden. Unterprogrammaufrufe könnten im Analyseverfahren umgangen werden, indem der Inhalt der Unterprogramme per Inlining in die Hauptprozedur übernommen wird (beim Inlining wird der Unterprogramm-Aufruf durch den Rumpf des Unterprogramms ersetzt). Dies ist jedoch nur bei nicht rekursiven Aufrufen und in bestimmten Kontexten möglich. Weitere Probleme ergeben sich durch den Einsatz von komplexen Datenstrukturen wie Arrays und Zeigern.

Der Systemabhängigkeitsgraph ist zunächst ein PDG für die main-Funktion. An jeder Prozeduraufruf-Stelle existiert zusätzlich eine Kontrollabhängigkeitskante zum Eintrittspunkt der Prozedur; der Entry-Knoten der Prozedur ist vom Aufruf kontrollabhängig. Hinzu kommen Kanten für die transitiven Datenabhängigkeiten. Hierzu werden Knoten, welche den Datenfluss in und aus dem Unterprogramm abbilden, definiert. Actual-In- und Actual-Out-Knoten bilden den Fluss der ursprünglichen Parameter (Werte im aufrufenden Programmteil) hin zu den vorübergehenden Datenobjekten im Unterprogramm ab. Im Unterprogramm existieren Formal-In- und Formal-Out-Knoten, welche den Datenfluss vom und zu dem Aufrufenden abbilden. Die Werte aus dem Actual-In-Knoten werden mit dem Unterprogrammaufruf in den Formal-In-Knoten kopiert. Somit existiert hier auch eine Kontrollabhängigkeit vom aufrufenden Knoten.

Zusätzliche Summary-Edges repräsentieren die transitive Datenabhängigkeit über einen bestimmten Call-Ort. Sie verbinden Actual-In-Knoten mit Actual-Out-Knoten.

In Abbildung 3.7 ist der Quelltext eines Beispiel-Programms, bestehend aus vier Funktionen `main`, `a`, `add` und `increment`, mit dem resultierenden Systemabhängigkeitsgraph dargestellt. Jede Instruktion (bzw. Zeile des Quelltextes) bildet einen Knoten des Graphen. Verbunden werden die Knoten durch Kanten für den Kontrollfluss, für schleifenunabhängigen Datenfluss, für schleifengetragenen Datenfluss und für Unterprogrammaufrufe und den zugehörigen Parametern.

3.10 Statische Analyse

Die statische Analyse zeichnet sich dadurch aus, dass das Programm nicht ausgeführt wird, sondern eine reine Quellcode-Untersuchung vorgenommen wird. Alle Analysen könnten auch durch den Entwickler von Hand durchgeführt werden. Bei der Analyse gilt es, einen Satz einfacher Regeln stur auf den Code anzuwenden. Durch die Komplexität von Programmbeschreibungen ist es schwierig, eine gute Übersicht zu behalten. Daher wird manuelle Durchführung leicht zu fehlerhaften oder unvollständigen Ergebnissen führen. Durch die klare Struktur in der Analyse lässt sie sich jedoch gut automatisieren. Entsprechende Analysewerkzeuge ähneln in ihrer Struktur Compilern, z. B. verwenden sie lexikalische und syntaktische Analysen (siehe Kapitel 2.3). Daher ist die statische Analyse auch während der Übersetzung möglich. Das Einsatzgebiet der stati-

```c
int main() {
  int sum = 0;
  int i = 1;
  while (i < 11)
    a(&sum, &i);
  printf("%d", sum);
  return i;
}
```

```c
void a (int *x, *y) {
  add(x, y);
  increment(y);
  return;
}
```

```c
void add (int *a, *b) {
  a += b;
  return;
}
```

```c
void increment (int *z) {
  add(z, 1);
```

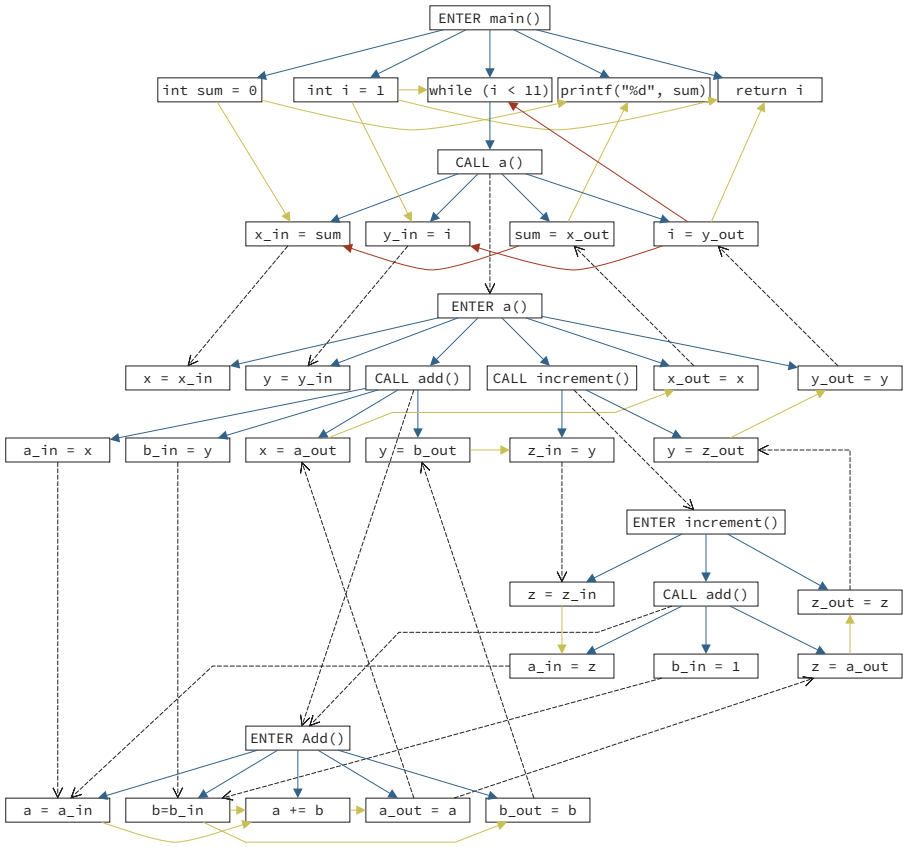


Abbildung 3.7: Beispiel-Programm mit Unterprogrammen als Quelltext und resultierender System-Dependence-Graph (vgl. [HRB88])
→Kontrollfluss
→Schleifenunabhängiger Datenfluss
→Schleifengetragener Datenfluss
- →Unterprogrammaufruf, Parameter in/out

schen Analyse reicht von der Überprüfung von Programmierkonventionen (Stilanalyse) bis hin zum Auffinden von Datenanomalien. [Lig09]

Bei der Analyse von Software, welche Unterprogramme im Quelltext aufweist, wird davon ausgegangen, dass nur ein Unterprogramm gleichzeitig ausgeführt wird. Es wird von intraprozeduraler Analyse gesprochen. Jedes Unterprogramm kann alle sichtbaren Variablen ändern und somit können Nebenwirkungen auftreten, die es zu erfassen gilt. Wird die Analyse für ein komplettes Programmsystem durchgeführt, handelt es sich um interprozedurale Analyse. In diesem Fall muss der Datenfluss von der aufrufenden Stelle (caller) zum aufgerufenen Unterprogramm (callee) ermittelt werden. Eine einfache Möglichkeit die Analyse in diesem Fall zu vereinfachen ist das Funktions-Inlining. Hierbei wird der Unterprogramm-Aufruf durch den Rumpf des Unterprogramms ersetzt. Dabei müssen geeignete Anpassungen für Parameter und Rückgabewerte durchgeführt werden. Dieser Schritt ist allerdings nicht immer möglich. So muss das Ziel des Aufrufs bekannt sein und darf nicht erst zur Laufzeit ermittelt werden können, wie es durch die Verwendung von Funktionszeigern der Fall ist. Auch wenn das Aufrufziel bekannt ist, kann es in einigen Fällen, z. B. bei rekursiven Funktionsaufrufen, für den Compiler nicht möglich sein ein Inlining durchzuführen [ULS+08].

Dadurch, dass das Programm nicht ausgeführt wird, werden keine Testdaten oder Testfälle benötigt, die Analyse ist daher allgemein gültig. Auf der anderen Seite können dadurch nicht alle Konstrukte vollständig untersucht werden. Die statische Analyse kann somit keinen Korrektheitsnachweis erbringen. [Lig09]

Die Möglichkeiten der statischen Analyse haben ihre Grenzen. Bei Programmen, welche Zeiger verwenden, ist die Bestimmung des Datenflusses nicht statisch möglich (vgl. [Hin01]). Ein wichtiger Aspekt der Zeigeranalyse ist die Frage, ob zwei Zeiger auf dieselbe Speicherstelle verweisen (Alias-Analyse) bzw. ob zwei Zeiger als Alias füreinander verwendet werden können. Neben der Zuordnung von Speicherstellen kann es aufschlussreich sein, die Zeiger in Beziehung zu mögliche Zielobjekten zu setzen. Die statische Analyse kann dabei nur eine konservative Abschätzung bieten, aber keine exakten Ergebnisse liefern. Ein empirischer Vergleich der Effektivität von Analysealgorithmen findet sich in [HP00]. Die Wahl hängt von dem einzugehenden Kompromiss zwischen Geschwindigkeit und Genauigkeit ab.

Die Bestimmung der Adresse, auf welche ein Zeiger verweist, wird als Point-To-Analysis bezeichnet. In C-Programmen ist die Zeigeranalyse komplex, da beliebige Berechnungen mit Zeigern durchgeführt werden können. (vgl. [ULS+08])

Manche Profilinformationen (z. B. Cachefehler) können nur sehr aufwändig oder gar nicht statisch ermittelt werden. Die meisten Programme weisen ein nicht-deterministisches Verhalten auf, da die Ausführung von externen Faktoren wie Eingangsdaten oder der Zeit abhängig ist. [Neu10]

3.11 Dynamische Analyse

Dynamische Programmanalysen werden verwendet um Informationen zu extrahieren die statisch nicht ermittelbar sind. Das Laufzeitverhalten macht Aussagen über die Verwen-

dung vorhandener Ressourcen (CPU-Zeit, Speicherplatzbedarf) möglich. Dabei ist das Verhalten abhängig von Aufrufparametern und somit von jeder Art Eingangsdaten. Wie gut die extrahierten Informationen sind, hängt unter anderem davon ab, wie gut sie Software durch die durchgeführten Tests abgedeckt wird (Coverage). Bei der Ausführung muss der Status protokoliert werden. Die Zustände müssen für die Weiterverarbeitung zugreifbar gemacht werden. Relevante Informationen, die es zu ermitteln gilt, können Variablenwerte oder auch das Erreichen bestimmter Stellen im Programm sein. Allgemein lässt sich der Kontroll- und Datenfluss innerhalb eines Programms zur Laufzeit bestimmen. Wenn nicht nur die Anzahl der Ausführungen an bestimmten Stellen im Programm, sondern auch die Ausführungsdauer ermittelt werden soll, dann muss jedoch berücksichtigt werden, dass durch das Ermitteln eines dynamischen Profils die Ausführung beeinflusst, verlangsamt wird.

3.11.1 Debugging

Debugging stellt eine Art der Laufzeituntersuchung dar. Das Programm wird beim Debugging Schritt für Schritt (step-by-step) ausgeführt. Ein Schritt kann einzelne Anweisungen umfassen, alternativ werden häufig Haltepunkte (*breakpoints*) an relevanten Stellen gesetzt. Erreicht die Ausführung eine so markierte Stelle wird die Ausführung angehalten. Die aktuellen Variablenwerte lassen sich auf diese Weise observieren.

3.11.2 Profiler

Der Zugriff auf die internen Programmzustände kann beispielsweise durch eine Instrumentierung erfolgen. Hierbei können zusätzliche Anweisungen in das Programm eingebaut werden. Diese beeinflussen dabei jedoch die Ausführung bzw. das Laufzeitverhalten. Es gibt mehrere Möglichkeiten der Instrumentierung. Zum einen können Profilierungsanweisungen vor dem Kompiliervorgang in den Quelltext eingebaut werden. Dies kann manuell oder compilergestützt geschehen. Dabei müssen die einzufügenden Anweisungen in der gleichen Programmiersprache formuliert sein wie das zu untersuchende Programm. Alternative Ansätze verwenden eine Instrumentierung zur Laufzeit (z. B. das Binärcode-Instrumentierungstool PIN [LCM+05]). Des weiteren ist es möglich das ausführbare Programm oder den Zwischencode, welcher während des Kompilierungsprozesses erzeugt wird, zu instrumentieren. Die Instrumentierung des Quelltextes und zur Kompilierzeit ist maschinenunabhängig, wohingegen die Modifikation des ausführbaren Programms an eine Hardware gebunden ist. Allerdings kann dann auf Hardware-Zähler zurückgegriffen werden.

Ein Profiler ist ein Programmierwerkzeug, welches die Instrumentierung und Auswertung übernimmt. Die Auswertung der Daten kann entweder parallel bereits zur Laufzeit erfolgen oder aber nach der Terminierung des zu untersuchenden Programms. Ein Aufrufgraph-Profiler bestimmt, wie oft und mit welcher Frequenz bestimmte Aufrufe getätigt werden. Häufig wird Profiling eingesetzt, um die Geschwindigkeit einzelner Funktionen zu messen bzw. die Anzahl der Aufrufe oder Durchläufe von Programmteilen zu ermitteln. Daraus lässt sich ableiten, an welchen Stellen Optimierungen eine signifikan-

te Auswirkung auf das Gesamtsystem haben. Ein weiteres Anwendungsfeld betrifft die Speichernutzung von Programmen. Die Analyse kann helfen, die Nutzung des Arbeitsspeichers zu optimieren bzw. Fehler zu finden. So können ungenutzte Speicherbereiche möglicherweise nicht wieder korrekt freigegeben werden (Speicherleck).

Neben der Instrumentierung kann eine statistische Auswertung Informationen über das Laufzeitverhalten ermitteln (z. B. oprofile). Hierbei wird in bestimmten Abständen der jeweils aktuelle Programmzustand ermittelt (Sampling). Die Untersuchung erfolgt also stichprobenartig. Aus den Zuständen kann ein statistischer Mittelwert errechnet werden. Dieses Verfahren ist allerdings nicht exakt. Das Sampling kann auch in spezieller Hardware erfolgen, dort kann z. B. der Zustand des Programmcounters verfolgt werden.

Ein bekanntes Problem bei der Verwendung von Profilern trifft aber auch genauso auf die statische Code-Analyse zu. Werden im zu untersuchenden Programm-Bibliotheken eingebunden, so steht meist deren Quelltexte nicht zur Verfügung. Dies gilt im Besonderen für Entwicklungen Dritter. Die Effekte, welche ein Bibliotheksaufruf verursacht, kann von der Analyse nicht erfasst werden. Die entsprechenden Datenflussinformationen fehlen in der Analyse. Eine mögliche Lösung besteht darin, einen Wrapper zu schreiben. Diese Wrapper-Funktion kann der Analyse gegenüber die summierten Datenflüsse simulieren und ruft selbst die Original-Funktion für die Realisierung der Funktionalität auf. Eine Instrumentierung des ausführbaren Programms umgeht diese Problematik, dafür ist es hier nicht mehr einfach möglich Rückschlüsse auf die ursprünglich im Quelltext verwendeten Datenobjekte oder Anweisungen zu ziehen. Eine grundsätzliche Schwierigkeit liegt darin, geeignete Stellen für das Einfügen der Profiling-Anweisungen zu bestimmen, insbesondere gilt dies für die Instrumentierung des Quelltextes und der Binärdatei.

3.11.3 Statistische Analyse

Bei der Untersuchung von Programmsystemen zur Laufzeit hängt die Ausführung von den Eingangsdaten ab. Je nach Wahl kann sich ein sehr unterschiedliches Verhalten der Programme ergeben. Um das Verhalten eines Programms auf Grund von Laufzeituntersuchungen umfassend bewerten zu können, ist es daher notwendig, das Verhalten in mehreren Konstellationen zu untersuchen. Es gilt daher die Eingangsdaten so zu variieren, dass alle möglichen Fälle abgedeckt werden. In wie weit dies erreicht wird, kann die Coverage-Analyse (Abschnitt 3.11.3.1) ermitteln. In Abschnitt 3.11.3.2 wird auf die statistische Untersuchung des Kontrollflusses eingegangen. Die Betrachtung des Datenflusses ist Inhalt des Kapitels 3.11.3.3.

3.11.3.1 Coverage-Analyse

Beim Testen von Programmen können einzelne Einheiten ausgeführt und dabei auf korrekte Funktionalität überprüft werden. Je kleiner die zu testenden Einheiten, desto überschaubarer ist die Menge benötigter Tests um das ganze Spektrum möglicher Programmzustände abzudecken. Bei komplexen Einheiten oder Programmsystemen ist eine Aussage darüber notwendig, welche Teile bereits durchlaufen wurden. Code Coverage stellt eine hierfür passende Messtechnik dar, und liefert ein Maß des vom Test ausgeführten

Codes. Da für diese Art der Tests der Quellcode und Kenntnisse über die innere Funktionsweise benötigt werden, handelt es sich um ein sogenanntes Whitebox-Testverfahren. Demgegenüber stehen Blackbox-Testverfahren, welche funktionsorientiert arbeiten. Bei dieser Klasse von Testverfahren wird überprüft, ob die Ausführung den Anforderungen entspricht, die innere Funktion wird nicht herangezogen. [Win03]

Man unterscheidet verschiedene Coverage-Metriken die die Ergebnisse präsentieren.

Anweisungsüberdeckung Line Coverage, Statement Coverage oder Anweisungsüberdeckung ermittelt, wieviele Zeilen des Quelltextes durchlaufen wurden. Anweisungsfolgen ohne Verzweigung können unter Umständen als Block betrachtet werden (auch Block Coverage). Ziel dieser Tests ist es, jede Anweisung mindestens einmal auszuführen, also eine Abdeckung von 100% zu erreichen.

Kantenüberdeckung Branch Coverage oder Kantenüberdeckung basiert auf den in Kapitel 3.2 eingeführten Kontrollflussgraphen. Selbst wenn das Ziel der Line Coverage erfüllt wurde, so müssen noch nicht alle Kanten betreten worden sein. Ein Programmbereich kann z. B. über mehrere unterschiedliche Kanten erreicht werden, wurde eine Kante durchlaufen, so wurden die entsprechenden Anweisungen ausgeführt, die anderen Kanten jedoch nicht. Branch Coverage versucht jede Kante einmal zu durchlaufen.

Bedingungsüberdeckung Decision Coverage oder Bedingungsüberdeckung stellt eine Erweiterung zu der Branch Coverage dar. Eine Verzweigung der Form

```
if (a <= 4711 && b) { /* dann ... /* } else { /* sonst ... */ }
```

kann zwei Pfade durchlaufen, Branch Coverage würde beide Fälle abdecken. Unter Decision Coverage müssten beide Bedingungen („`a <= 4711`“ und „`b`“) für die Verzweigung separat betrachtet werden und alle Konstellationen auftreten um eine vollständige Abdeckung zu erreichen.

Pfadüberdeckung Path Coverage oder Pfadüberdeckung betrachtet einzelne Kanten des Kontrollflussgraphen als Bestandteil von Pfaden. Ein Pfad ist ein möglicher Weg durch eine Untersuchungseinheit (z. B. Methode), also von deren Betreten bis zum Verlassen. Für eine vollständige Abdeckung müssen alle möglichen Kombinationen, alle theoretisch möglichen Wege durchlaufen werden. Dies ist nicht immer möglich, beispielsweise wenn zwei Verzweigungen von der gleichen Bedingung abhängen, sind die Ausführungen der kontrollabhängigen Einheiten korreliert.

3.11.3.2 Statistische Kontrollfluss-Analyse

Die Häufigkeit des Folgens einer Kontrollflusskante kann von den Programmeingangsdaten abhängig sein. Dies ist beispielsweise der Fall, wenn eine Schleife über Teile einer zu verarbeitenden Datei iteriert. Kanten, die besonders häufig durchlaufen werden, deuten auf Programmteile hin, die auch einen großen Anteil an der Ausführungsdauer haben.

Somit besteht besonders an diesen Stellen Optimierungspotential. Wenn trotz der vielfachen Ausführung Kanten nicht betreten wurden, so ist eine vollständige Abdeckung aller Konstellationen nicht gegeben. Wird einer Kante mit einer von den Eingangsdaten unabhängigen und von eins verschiedenen Anzahl gefolgt, so liegt eine fixe Anzahl der Iterationen einer Schleife vor. Wie in Kapitel 8.2.1 beschrieben existieren dort mehr Freiheiten für die Optimierung.

3.11.3.3 Statistische Datenfluss-Analyse

Datenkanten, welche durch Variablen mit fester Größe gebildet werden, sind für die statistische Analyse irrelevant. Bei Datenflusskanten, welche Arrays variabler Größe oder Zeiger auf Speicherbereiche repräsentieren, ist das transportierte Datenvolumen nicht statisch. In diesen Fällen kann das Volumen in Abhängigkeit der Eingangsdaten stehen. Innerhalb von Programmen, die primär Eingangsdaten verarbeiten, stellt das Datenvolumen eine Möglichkeit dar, den Fluss dieser Daten durch das gesamte Programm zu verfolgen. Somit kann die Hauptdatenverarbeitung von anderen nebengeordneten Prozessen unterschieden werden.

Kapitel 4

Softwarevisualisierung

Softwarevisualisierung verfolgt verschiedene Zielsetzungen. So kann sie Aufgaben während der Phasen Design, Implementierung, Test, Debugging, Analyse und Wartung erleichtern. Im Gegensatz zur visuellen bzw. grafischen Programmierung dient die Visualisierung nicht der Erzeugung von Programmlogik, sondern der Analyse von existierenden Programmen. Im weiteren Sinn umfasst der Begriff der Softwarevisualisierung Programme und deren Entwicklungsprozess. Dabei werden beispielsweise Anforderungen oder Änderungen während der Entwicklung dargestellt.

Werden die in Kapitel 3 dargestellten Abhängigkeiten betrachtet, ergeben sich je nach Granularität sehr komplexe Strukturen. Eine Analyse auf Instruktionsebene führt schon bei trivialen Programmen zu komplexen Kontroll- und Datenflussgraphen. Der Grad der Abstraktion spielt dabei eine wesentliche Rolle um den größtmöglichen Nutzen aus der Darstellung zu ziehen. Visualisierung hat das Ziel, die Menge der Informationen derart zu reduzieren und aufzubereiten, dass möglichst nur die essentiellen Elemente hervortreten. Es handelt sich um eine, bezüglich der Zielsetzung effiziente, Abbildung der darzustellenden Informationen auf grafische Merkmale wie Position, Größe, Form oder Farbe. Im wesentlichen finden grafische und tabellarische Darstellungen ihre Anwendung.

Visualisierung hat immer das Ziel, dem Betrachter komplexe Zusammenhänge näher zu bringen und zum besseren Verständnis beizutragen. Hierzu wird die im Kontext relevante Information in eine visuelle Form überführt. Im engeren Sinn geht es um die Darstellung von Algorithmen und Programmen. Hierbei stehen die Aspekte der Struktur und des Verhaltens im Vordergrund.

Bei der Anforderungsanalyse bzw. für den Systementwurf während des Entwicklungsprozesses können Visualisierungen eingesetzt werden. Ein Ansatz ist die strukturierte Analyse nach Tom DeMarco (siehe [DeM79]). Über einen Top-Down-Ansatz werden komplexe Systeme in einfachere Funktionen aufgeteilt und dabei der Datenfluss modelliert. Es werden Entscheidungstabellen und Zustandsübergangsdiagramme verwendet. Für die Darstellung objektorientierter Programme haben Analyse- und Entwurfsmethoden wie UML eine große Verbreitung. Viele Darstellungen lassen sich automatisiert aus dem Quelltext erzeugen.

Im Fokus dieses Kapitels liegt die Visualisierung von Programmen. Hierbei sollen die

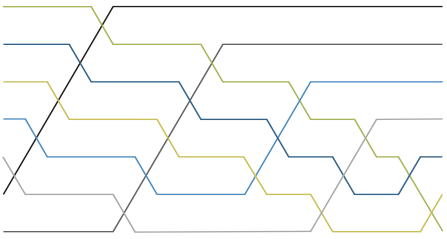
zuvor beschriebenen Abhängigkeiten zwischen Programmfragmenten dargestellt werden. Auf diese Weise werden Strukturen ersichtlich, die Optimierungspotential aufweisen. Ziel ist es, durch die Betrachtung existierender Programme und deren internen Aufbau Konzepte für eine neu organisierte Ausführung mit möglicherweise nebenläufigen Ausführungssträngen ausfindig zu machen. Die Darstellung von Algorithmen beschränkt sich darauf, deren Funktionsweise verständlich zu erklären. Dies könnte für den Bubble-Sort-Algorithmus wie in Abbildung 4.1 aussehen. Die Struktur beschreibt statische Teile eines Programms und die Beziehungen untereinander. Teile können sowohl Anweisungen als auch Datenstrukturen sein. Im Gegensatz dazu werden dynamische Eigenschaften durch die Verhaltensvisualisierung dargestellt. Es geht um eine Verdeutlichung der Änderung des Programmzustandes während der Ausführung.

```
void bubblesort (int *array, int length) {
  for (int i = length; i > 1; i--) {
    for (int j = 0; j < i-1; j++) {
      if (array[j] > array[j+1]) {
        int tmp = array[j];
        array[j] = array[j+1];
        array[j+1] = tmp;
      }
    }
  }
}
```

a)

8	9	6	7	3	4	0
8	9	6	7	3	4	0
8	6	9	7	3	4	0
8	6	7	9	3	4	0
8	6	7	3	9	4	0
8	6	7	3	4	9	0
8	6	7	3	4	0	9
6	8	7	3	4	0	9
6	7	8	3	4	0	9
6	7	3	8	4	0	9
6	7	3	4	8	0	9
6	7	3	4	0	8	9
6	7	3	4	0	8	9
6	3	7	4	0	8	9
6	3	4	7	0	8	9
6	3	4	0	7	8	9
3	6	4	0	7	8	9
3	4	6	0	7	8	9
3	4	0	6	7	8	9
3	4	0	6	7	8	9
3	0	4	6	7	8	9
0	3	4	6	7	8	9

b)



c)

Abbildung 4.1: Algorithmus-Visualisierung am Beispiel von Bubble-Sort: a) Algorithmus mit Syntaxhervorhebung, b) Änderungen an dem zu sortierenden Array. Jede Zeile entspricht einem Schritt. Blau hinterlegte Felder werden verglichen, und bei Bedarf getauscht, c) Grafische Darstellung der Änderungen

Die im vorherigen Kapitel entwickelten Modelle haben die Zielsetzung mögliche Parallelität in sequentiellen Programmen ausnutzen zu können. Dadurch ergeben sich Abhängigkeiten zwischen Fragmenten. In Hinblick auf mögliche Programmtransformationen wird es wichtig Programmstrukturen zu analysieren. Bei geänderten Programmausführungen müssen programminterne Abhängigkeiten erhalten bleiben. Visualisierungen sind ein Hilfsmittel um Strukturen greifbar und nachvollziehbar zu machen; auf deren

Basis können Nebenläufigkeiten entwickelt werden. Die Visualisierung der Programmstruktur hat Kontroll- und Datenflussabhängigkeiten in gleichem Maße zu berücksichtigen. Parallelität kann bei geeigneten Verfahren bereits teilweise ersichtlich werden.

Die einfachste Art um die Funktionsweise bzw. Struktur eines Programms für den Entwickler besser erkennbar zu machen ist, den Quelltext zu formatieren. Dies reicht von einheitlicher Einrückung bis zum Syntax-Highlighting, der Darstellung bestimmter Zeichenketten in angepassten Farben und/oder Schriften in Abhängigkeit von deren Bedeutung.

Eine darüber hinausgehende visuelle Darstellung muss sich grafischer Elemente bedienen. Die Art der Darstellung ist dabei dem Zweck angepasst. Mögliche Formen werden bei Betrachtung existierender Darstellungsarten deutlich. Im Folgenden werden verbreitete Darstellungen gezeigt. (vgl. [Die07])

Klassische statische Visualisierungen beruhen auf Programmteilen und deren Abhängigkeiten untereinander, wie sie in Kapitel 3 vorgestellt wurden. Mathematisch handelt es sich dabei um gerichtete Graphen. Deren visuelle Abbildung wird in einem eigenen Themengebiet der Informatik (vgl. z. B.[HMM00; KW01]) des Graphzeichnens (*graph drawing*) behandelt. Mit der Zielsetzung, Programmoptimierungen auf Basis von Visualisierungen zu realisieren, beschäftigen sich Forscher weltweit. Restrukturierungsmaßnahmen können durch die Visualisierung von Abhängigkeitsgraphen, wie in [KB96] beschrieben, unterstützt werden. [PGV08] zeigt eine Variante um Entwicklern mögliche Parallelität in sequentiellen Programmen zu verdeutlichen. Dabei kommt eine interaktive Visualisierung auf Basis von Traces zum Einsatz. In etablierten Design-, Entwicklungs- und Wartungsprozessen haben sich die im Folgenden gezeigten Darstellungen etabliert.

4.1 Struktogramm

Für die Darstellung von Programmabläufen können Struktogramme eingesetzt werden. Vorgestellt wurden sie in [NS73] und sind daher auch als Nassi-Shneiderman-Diagramm bekannt. Sie entsprechen der Norm [DIN 66261]. Diese Darstellung basiert auf ineinander verschachtelten Rechtecken, bzw. einem äußeren Rechteck, welches durch gerade Linien unterteilt wird. Einzelne abgetrennte Bereiche können sich somit nicht überlappen. Eindeutige Strukturen sind für Sequenzen, Verzweigungen und Schleifen definiert. Die Struktur der Darstellung ist somit eine Abbildung des Kontrollflusses. Zugriffe auf Daten und Ausgaben sind enthalten, allerdings nur in textueller Form. Eine Voraussetzung für diese Darstellungsform ist die Aufteilung des Programms in fest vorgegebene Strukturblöcke wie Verzweigungen und Schleifen. Dies lässt sich aus dem Quelltext ableiten. Abhängigkeitsgraphen beinhalten diese Informationen indirekt. Abbildung 4.2 zeigt das Struktogramm für den Bubble-Sort-Algorithmus.

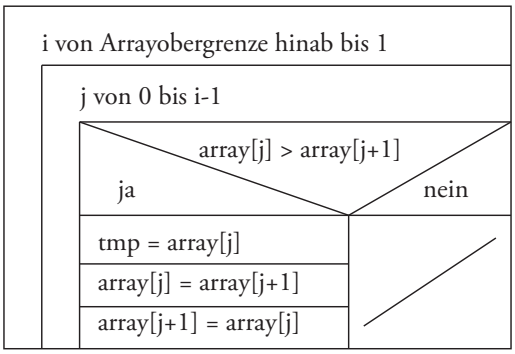


Abbildung 4.2: Bubble-Sort dargestellt als Struktogramm

4.2 Programmablaufplan

Goldstine und von Neumann haben 1947 den Kontrollflussgraphen eingeführt [GN47]. Daraus hervorgegangen ist die Norm [DIN 66001], welche Sinnbilder und ihre Anwendung in der Informationsverarbeitung definiert. Das Ziel ist die Darstellung von „Daten und ihrer Verarbeitung“. Hervorzuheben ist hierbei der Programmablaufplan, welcher auch als Flussdiagramm (*flowchart*) bezeichnet wird. Die Reihenfolge der Verarbeitungsschritte wird als Graph dargestellt. Es erfolgt eine Aufteilung in Blöcke und Verbindungen, in Form von gerichteten Kanten, zwischen diesen. Anweisungen werden in Rechtecken dargestellt, es können auch Sequenzen (z. B. Basisblöcke) in einem Knoten zusammengefasst werden. Verzweigungen werden durch Rauten gesondert dargestellt. Programmablaufpläne sind durch ihre Blockstruktur flexibler als beispielsweise Struktogramme. Abbildung 4.3 zeigt einen beispielhaften Programmablaufplan.

Neben dem Programmablaufplan werden in [DIN 66001] weiterhin Datenflussplan, Programmhierarchie, Programmnetz, Datennetz, Datenhierarchie und Konfigurationsplan als Darstellungsformen definiert. Wie in [Hai59; Sco58] gezeigt, sind Programmablaufpläne automatisch generierbar.

4.3 UML

Die Unified Modeling Language (kurz UML) spezifiziert eine grafische Notation um Softwaresysteme zu modellieren, dokumentieren, visualisieren und zu analysieren. 2015 wurde die aktuelle Version 2.5 veröffentlicht. Dabei werden die 14 in Tabelle 4.1 aufgeführten Diagrammtypen definiert.

Das primäre Ziel der UML ist die Betrachtung objektorientierter Programme. Viele Diagrammtypen finden ihre Anwendung bei der Modellierung. Die Struktur objektorientierter Programme wird durch Klassendiagramme, Paketdiagramme, Objektdiagramme, Kompositionsstrukturdiagramme und Komponentendiagramme erfasst. Das Verteilungsdiagramm beschreibt wie Komponenten zur Laufzeit auf verschiedene Hardware verteilt werden. Das Verhalten der Komponenten selbst ist nicht Gegenstand der Dar-

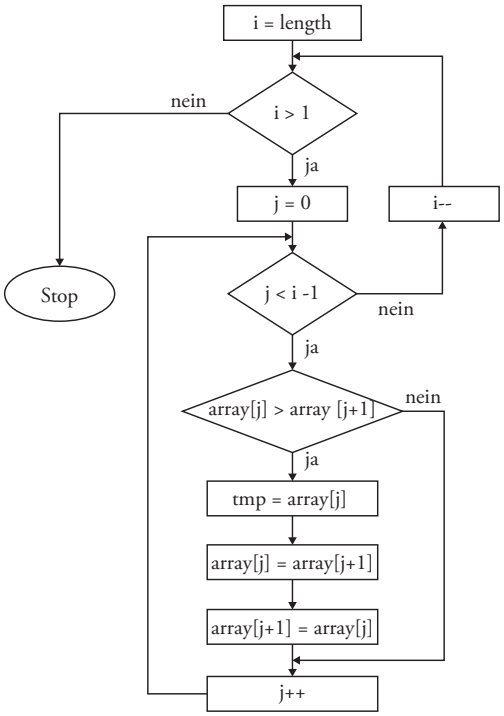


Abbildung 4.3: Bubble-Sort dargestellt als Programmablaufplan

stellung. Das Use-Case-Diagramm bietet eine Darstellung des Verhaltens aus Sicht des Benutzers, also der Schnittstelle zur Umgebung. Dieser Diagrammtyp stellt die Funktionalität des Programms dar und Aktionen, die ein Benutzer (Mensch oder anderes Programm) ausführen muss um Ausgaben zu erzeugen.

Um Abläufe zu modellieren, kann das *Aktivitätsdiagramm* dienen. Operationen werden durch Kontroll- und Objektflüsse verkettet. Es besteht die Möglichkeit der Hierarchiebildung um die Übersichtlichkeit zu erhöhen. In der Regel werden diese Diagramme von Hand im Entwurfsprozess und nicht automatisiert erzeugt. Aktivitäten sind keines-

Tabelle 4.1: Diagramme der UML 2 [RQS12]

Strukturdiagramme	Verhaltensdiagramme	
		Interaktionsdiagramme
Klassendiagramm	Use-Case-Diagramm	Sequenzdiagramm
Paketdiagramm	Aktivitätsdiagramm	Kommunikationsdiagramm
Objektdiagramm	Zustandsautomat	Timingdiagramm
Kompositionsstrukturdiagramm		Interaktionsübersichtsdiagramm
Komponentendiagramm		
Verteilungsdiagramm		
Profildiagramm		

wegs trivial, sondern neigen zu hoher Komplexität [RQS12]. Die Komponenten des Diagrammtyps und funktionale Eigenschaften wie das beinhaltete Token-Konzept machen Aktivitätsdiagramme zu einem mächtigen Werkzeug.

Interaktionen zwischen Systemen oder innerhalb eines Systems lassen sich mit Sequenzdiagrammen, Timingdiagrammen oder Kommunikationsdiagrammen visualisieren. Der Informationsaustausch wird sowohl im Sequenzdiagramm als auch im Timingdiagramm über einer Zeitachse aufgetragen. Jedes Objekt weist eine Lebenslinie auf, welche dessen Existenz über die Zeit repräsentiert. Nachrichten mit aktivierenden Operationen verbinden diese.

Kommunikationsdiagramme stellen eine Alternative zu Sequenzdiagrammen dar. Es steht jedoch nicht der zeitliche Ablauf im Fokus der Darstellung, sondern die Kommunikationsstruktur. Für die zeitliche Abfolge können Nachrichten nummeriert werden. Es werden zwischen Objekten übertragene Nachrichten und die Assoziation zwischen Klassen abgebildet, um darzustellen, welche Teile wie zusammen arbeiten um eine Funktion zu erfüllen.

Das Timingdiagramm zeigt die Änderung der Zustände der Kommunikationspartner unter Angabe der Zeit. Die Darstellungsform stammt aus der Elektrotechnik und wird dort verwendet, um den zeitlichen Verlauf von Signalen bzw. in größerer Analogie zu Software das zeitliche Verhalten einer digitalen Schaltung darzustellen.

Das Interaktionsübersichtsdiagramm ist eine Variante des Aktivitätsdiagramms zur Darstellung in welcher Reihenfolge und unter welchen Bedingungen Interaktionen stattfinden.

4.4 Tabellarische Darstellung

Eine Möglichkeit die Datenabhängigkeiten innerhalb eines Programms kompakt darzustellen ist die Verwendung einer Variablen-Cross-Reference-Tabelle. Jede Spalte entspricht einer Variablen, jede Zeile einer Funktion oder eines anderen Programmteils. Ein Tabelleneintrag gibt an, ob die jeweilige Stelle im Programm die per Spalte zugeordnete Variable liest und/oder schreibt oder nicht zugreift (r, w, rw, -).

Auf Basis dieser Darstellung wurde eine kombinierte Daten- und Kontrollfluss-Darstellung wie in Abbildung 4.4 abgebildet kreiert. Die Quellen wurden als Zeilen, die Senken als Spalten umgesetzt. Die Zusammenhänge wurden nicht textuell, sondern farblich kodiert. Bei einer Sortierung der Einheiten entsprechend ihrer Tiefe im Kontrollflussgraph, stellt die obere Dreiecksmatrix vorwärts gerichtete Kontrollflüsse, die untere Dreiecksmatrix rückwärts gerichtete Kontrollflüsse, welche durch Schleifen entstehen, dar.

4.5 Kontroll- und Datenflussgraph

Die in Kapitel 3 betrachteten Abhängigkeiten stellen für Programmanalyse und Optimierung relevante Informationen dar. Somit ist deren Visualisierung zur Analyse möglicher

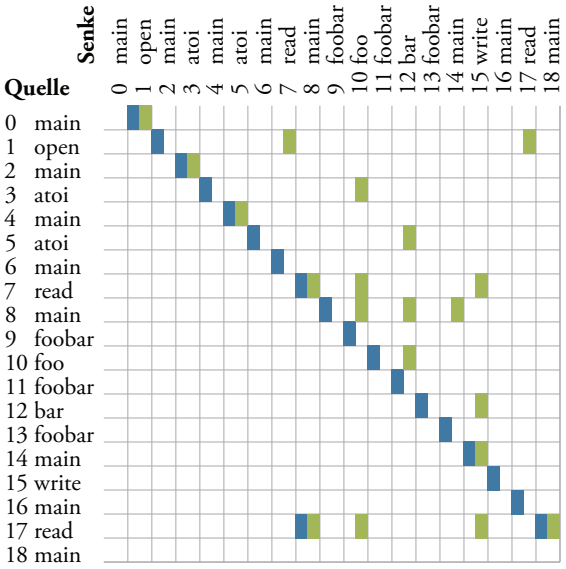


Abbildung 4.4: Tabellarische Darstellung der Kontroll- und Datenflüsse innerhalb eines Beispiel-Programms (siehe Listing 6.1). Blau: Kontrollfluss; Grün: Datenfluss

Konstellationen hilfreich. Die Abhängigkeiten entsprechen in ihrer Struktur Graphen. Die Graphvisualisierung lässt sich für diese Aufgabe verwenden. Besonders wertvolle Informationen enthält ein Kontrollflussgraph (CFG), kombinierter Kontoll- und Datenflussgraph (CDFG) oder ein Datenabhängigkeits- und Kontrollabhängigkeitsgraph (Graphenrepräsentation der in Kapitel 3 eingeführten Daten- und Kontrollabhängigkeiten).

Beim Graphzeichnen ist das Ziel eine ästhetische Umsetzung. Es gibt verschiedene Ansätze die Positionierung der Knoten und Kantenführung auf einer zweidimensionalen Fläche zu optimieren. So können die Kantenkreuzungen, die Anzahl der Knicke, die Fläche des Graphen oder die Länge der Kanten minimiert werden (vgl. [BRS+07; WPC+02; PCJ96]). Zur besseren Unterscheidung der Kanten gilt es, die Winkel zwischen Kanten an einem Knoten zu maximieren. Dementsprechend finden sich diverse Zeichen-Algorithmen. Typische Vertreter führen zu orthogonalen, kräftebasierten oder hierarchischen Layouts. Ein bedeutendes Programmpaket zur Graph-Darstellung ist Graphviz [Gra]. Für Programmgraphen geringer Komplexität können geeignete Graphen produziert werden. Für komplexere Programme ergeben sich viele Kanten, besonders bei kombinierten Kontroll- und Datenflussgraphen. Diese sind nicht mehr handhabbar.

Kapitel 5

Fragmentierungs- und Restrukturierungsmethoden

Um den Programmablauf nachvollziehen zu können, ist es notwendig, die Betrachtungsebene und deren Komponenten zu definieren. Nach der Ausführung einer Einheit wird die Aktivität an eine nachfolgende Einheit weitergereicht. Neben dem Programmablauf wird es relevant sein, Wissen über die Verwendung von Daten zu erlangen. Dabei auftretende Folgen von lesenden und schreibenden Zugriffen führen zu einem Datenfluss (vgl. Kapitel 3.3).

Wird eine Programmbeschreibung durch den Compiler für einen einzelnen Prozessor übersetzt, so entsteht eine Anweisungsfolge, welche in dieser Art ausgeführt werden kann. Alle Schritte von der Programmbeschreibung, bis hin zur ausführbaren Datei basieren auf dem gleichen Paradigma. In modernen Architekturen ist jedoch ein anderer Abstraktionslevel vonnöten. Einzelne Prozessorkerne, besonders aber lose gekoppelte Prozessoren wie in Rechnerclustern, benötigen mehr Zeit für den Datenaustausch (vgl. z. B. [CGP07; AI88]). Daher muss ein zusammenhängendes Programm in eine Menge von einzelnen Aufgaben zergliedert werden. Jede Aufgabe, welche einzelnen Prozessoren zugewiesen wird, enthält dann wiederum eine Folge von Instruktionen. Eine solche beliebige Folge kann dann ausgeführt werden, wenn bekannt ist, welche und wie viele Daten sie benötigt und wenn diese Daten tatsächlich vorliegen. Vorgänger im ursprünglich sequentiellen Ablauf müssen die entsprechenden Daten möglicherweise zuvor produzieren. Sind die Abhängigkeiten nicht erfüllt, kann die korrekte Ausführung mit der Ausgabe korrekter Ergebnisse nicht erfolgen. Grundlegende Anforderungen an die Zerlegung bezüglich des Kontrollflusses wurden in [EG95] betrachtet. Dort werden Methoden zur Bestimmung von parallelen Einheiten und Synchronisierungspunkten präsentiert. Als Basis dient dabei der Quelltext, jedoch lässt sich dies nicht, wie in dieser Arbeit angestrebt, auf C-Programme mit der Pointerproblematik übertragen, grundsätzlich wurde der Datenfluss in [EG95] außenvorgelassen.

Zur Steigerung der Parallelität auf der Instruktionsebene (*instruction level parallelism (ILP)*) werden unter anderem Bereichsbildungstechniken wie Superblocks oder Hyperblocks eingesetzt. Prozessoren können gleichzeitig mehrere voneinander unabhängige

Anweisungen ausführen. Beim Instruction-Pipelining überlappt die Ausführung teilweise (siehe Kapitel 2.4.2.1). Superskalare oder VLIW-Prozessoren (Very Long Instruction Word) verfügen über mehrere Ausführungseinheiten und können mehrere Anweisungen parallel ausführen. Daraus wurde ein Programmierparadigma (Explicitly Parallel Instruction Computing (EPIC)) für eben diese Architekturen abgeleitet. [Wal91] greift Studien auf, dass die Parallelität innerhalb von Basisblöcken im Mittel kaum den Faktor 3 bis 4 übersteigt. Eine Ausnahme stellen besondere stark parallele nummerische Programme auf unendlichen Hardwareressourcen dar. Die Studien von Wall ([Wal91]) belegen, dass eine Instruktionsebenenparallelität um 5 parallele Einheiten möglich ist, jedoch selten mehr.

Wenn die interne Struktur eines Programms geändert wird, ohne das Verhalten zu verändern, spricht man in der Softwareentwicklung vom Restructuring oder Refactoring. Werden Funktionen beispielsweise zu groß oder beinhalten zu viele Aufgaben, kann der Quellcode restrukturiert werden. Hierfür kann Program Slicing (siehe Kapitel 5.2.6) herangezogen werden. [LD98] beschreibt beispielsweise einen Weg, große Funktionen in kleinere zu zerlegen.

Ein oft eingesetztes Kriterium ist die Kohäsion (*cohesion*) und die Kopplung (*coupling*) (vgl. [LZ04]). Kohäsion beschreibt die Zusammengehörigkeit innerhalb einer Programmeinheit. Wenn jede Einheit (Funktion oder Klasse) genau eine definierte Aufgabe erfüllt, liegt eine starke Kohäsion vor. In objektorientierten Sprachen wird dadurch die Zusammengehörigkeit von Attributen und Methoden einer Klasse erhöht. Schwache Kohäsion zeichnet sich unter anderem durch Code-Wiederholungen aus. Die Kopplung ist ein Maß für existierende Verknüpfungen zwischen Programmeinheiten.

Es werden mehrere Arten der Kopplung unterschieden. Inhaltskopplung sollte vermieden werden. Hierbei bezieht sich ein Modul auf eine konkrete Implementierung eines anderen Moduls. Dabei werden interne Datenobjekte des anderen Moduls manipuliert. Wenn die Kommunikation zweier Module über globale Variablen erfolgt, handelt es sich um Bereichskopplung. Bei gegenseitigem Aufruf einzelner Module wird dies als Schnittstellenkopplung klassifiziert. Wird der Kontrollfluss durch ein anderes Modul beeinflusst, handelt es sich um Kontrollkopplung. Diese Art der Kopplung kann durch Funktionsparameter oder über Rückgabewerte realisiert werden. Liegt eine Datenübergabe per Parameter vor, handelt es sich um Datenkopplung. Wenn die dabei verwendeten Datenobjekte komplexe Strukturen sind, die Kommunikation jedoch nur über einen Teil der Datenelemente erfolgt, wird dies als Strukturkopplung bezeichnet. Das Ziel in der Softwareentwicklung ist es, eine lose Kopplung und starke Kohäsion zu erreichen.

Dieses Kapitel beschreibt relevante Aspekte für die Zerlegung eines sequentiellen Programms. Zunächst ist die Frage nach der geeigneten Größe der zu bildenden Fragmente zu klären. Darauf folgt eine Betrachtung bekannter Modelle. Etablierte Restrukturierungsmaßnahmen betreffen primär durch Schleifen definierte Bereiche. Daher werden entsprechende Strategien am Ende dieses Kapitels beleuchtet.

5.1 Ideale Quantität der Teilaufgaben

Um die Ausführung von Programmen restrukturieren zu können, ist es notwendig das ursprüngliche Programm zu zerlegen. Im Folgenden werden die Anforderungen für diese Zerlegung aufgezeigt. Daraus leitet sich im Anschluss ein Konzept für die Zerlegung zur parallelisierten Ausführung von Programmen ab. Dieses kann, wie später gezeigt, auch zur Visualisierung herangezogen werden.

Wenn ein Programm aufgeteilt und in anderer Reihenfolge bzw. parallel ausgeführt werden soll, darf sich das Verhalten nicht ändern. Instruktionen müssen im gleichen Kontext stehen und mit den korrekten Daten ausgeführt werden. Die relevanten Kontroll- und Datenflussabhängigkeiten müssen also erhalten bleiben. Um über die Instruktionsebenenparallelität hinausgehende Programmeigenschaften für die schnellere Ausführung ausnutzen zu können, sind Restrukturierungen auf einer höheren Ebene notwendig. Daher gilt es, Programme in gröbere Elemente zu zergliedern. Die Voraussetzung ist jedoch, Programmteile vorliegen zu haben, von denen die Abhängigkeiten vollständig bekannt sind. Zudem dürfen sich die Abhängigkeiten nicht ändern, weder zur Laufzeit noch in unterschiedlichen Programmaufrufen. Um einen Performancegewinn erzielen zu können, wird es vonnöten sein die Granularität anpassen zu können. Die Granularität eines parallelen Programms ist die durchschnittliche Größe der sequentiellen Programmteile, also der Anweisungsfolgen ohne Synchronisation oder Interprozessorkommunikation. Für die Performance einer verteilten Programmausführung ist das Verhältnis zwischen der Anzahl und der durchschnittlichen Größe der sequentiell auszuführenden Einheiten relevant. Die Skalierbarkeit ist die Befähigung eines Mehrprozessorsystems die Ausführungsgeschwindigkeit linear mit der Anzahl der Prozessoren zu erhöhen. Eng gekoppelte Systeme haben eine kleinere Granularität und Skalierbarkeit als lose gekoppelte Systeme. Die kleinere Granularität der Hardware ermöglicht eine Ausführung einer größeren Bandbreite paralleler Programme. Für Software ist eine große Granularität erstrebenswert, da somit eine effiziente Ausführung auf einer größeren Anzahl verschiedener Hardware möglich ist.

Das Problem der Programmzerlegung untergliedert sich in zwei Bereiche. Zum einen die Partitionierung, zum anderen das Scheduling. Die Partitionierung muss sicherstellen, dass die Granularität grob genug für die Hardwarearchitektur ist, ohne zu viel Parallelität zu verschenken. Das Scheduling, also die Zuordnung der Programmteile auf Prozessoren, sollte versuchen die Interprozessorkommunikation zu optimieren. Es gilt die Granularität gegenüber dem Overhead (Mehraufwand) durch die Synchronisation abzuwägen. Eine feingranulare Zerlegung ermöglicht einen hohen Grad der Parallelität, erfordert jedoch auch einen höheren Verwaltungs- und Kommunikationsaufwand. Wenn der Kommunikationsaufwand zwischen verteilten Einheiten minimiert werden soll, ist eine gröbere Zerlegung notwendig.

5.1.1 Overhead

Die benötigte Zeit für die Programmausführung ergibt sich aus der benötigten Zeit für die Anweisungen des Programms sowie der Zeit für Scheduling/Koordination sowie

Kommunikation (kombiniert in einem Overheadfaktor).

Abbildung 5.1 zeigt für fiktive Programme, welche auf 10 Prozessoren ausgeführt werden, die Abhängigkeit der Ausführungsdauer von der Partitionierung (durchschnittliche Größe der sequentiellen Programmeinheiten). Die Granularität variiert von 10 (eine Instruktion pro Task) bis zu 100000 (vollständiges Programm in einem Task). Die normalisierte parallelisierte Ausführungszeit, welche dargestellt wird, entspricht dem Verhältnis $\frac{n}{s}$ mit n als der Anzahl der Prozessoren und s als Beschleunigung im Vergleich zu einem Einzelkernprozessor. Durch den Verlust an Parallelität steigt die ideale normalisierte parallele Ausführungszeit des parallelisierten Programms ohne die Berücksichtigung von Overhead von 1 auf 10 an. Der Overhead-Faktor wird mit $\frac{(g+o)}{g}$ ermittelt. g ist dabei die Task-Größe und o der Overhead. Die sich ergebene reale Ausführungsdauer des parallelisierten Programms ergibt sich aus der Multiplikation der idealen Ausführungsdauer mit dem Overhead-Faktor. In Abbildung 5.1 wurde die sequentielle Ausführungszeit mit 10^5 Takten angenommen, und der Overhead als 10^3 Takte. Die Grafik zeigt, dass es aufgrund des Overheads unmöglich ist die ideale Ausführungszeit zu erreichen, und dass die tatsächliche Ausführungszeit ein Minimum bei einer mittleren Granularität aufweist. Das Problem der Partitionierung besteht darin, die Größe der Programmeinheiten an das Optimum anzunähern. Eine Unterteilung in gleich große Einheiten wird für reale Probleme nicht möglich sein. Ebenso ist eine Bestimmung des Overheads problematisch, da dieser ebenfalls von der Partitionierung abhängt. [Sar89]

5.1.2 Amdahlsches Gesetz

Das Amdahlsche Gesetz (vgl. [Amd67]) versucht die maximale Beschleunigung durch Parallelisierung vorherzusagen. Dabei wird ein Programm in parallelisierbare und nicht parallelisierbare Anteile zerlegt. Der maximal erreichbare Geschwindigkeitsgewinn (*speedup*) kann durch dieses Gesetz theoretisch vorhergesagt werden. Die Beschleunigung hängt von der Anzahl der Prozessoren p, und dem sequentiellen Anteil f ab. Der parallelisierbare Anteil ist dann $1 - f$.

Die Laufzeit eines parallelen Programms $T_p(\nu)$ ist die Zeit vom Programmstart bis alle parallelen Teile beendet wurden. Sie hängt von der Anzahl der Prozessoren und der Problemgröße ν ab. Dabei haben folgende Aspekte einen Anteil an der Laufzeit. Die Rechenzeit T_{CPU}, welche für die Ausführung der Anweisungen benötigt wird, die Kommunikationszeit T_{COM} für den Austausch von Daten zwischen Prozessoren, bei ungleicher Auslastung der Prozessoren für das Warten auf Datenabhängigkeiten die Wartezeit T_{WAIT}, die Synchronisationszeit T_{SYN} für die Synchronisation zwischen Prozessen, die Platzierungszeit T_{Place} für die Zuordnung der Aufgaben auf Prozessoren und die Lastverteilung und die Startzeit T_{Start} für die benötigte Zeit zum Starten der Tasks auf den jeweiligen Prozessoren. Der Overhead ist die Summe aus Kommunikations-, Warte-, und Synchronisationszeit. Die Summe der Platzierungs- und Startzeit wird als Rüstzeit bezeichnet. Die Zeit für die Ausführung des parallelen Programms auf einem Prozessor ist $T_1(\nu)$, welche nicht mit der Zeit der schnellsten bekannten Ausführung eines sequentiellen Algorithmus $T'(\nu)$ übereinstimmen muss. Der Geschwindigkeitsgewinn wird als

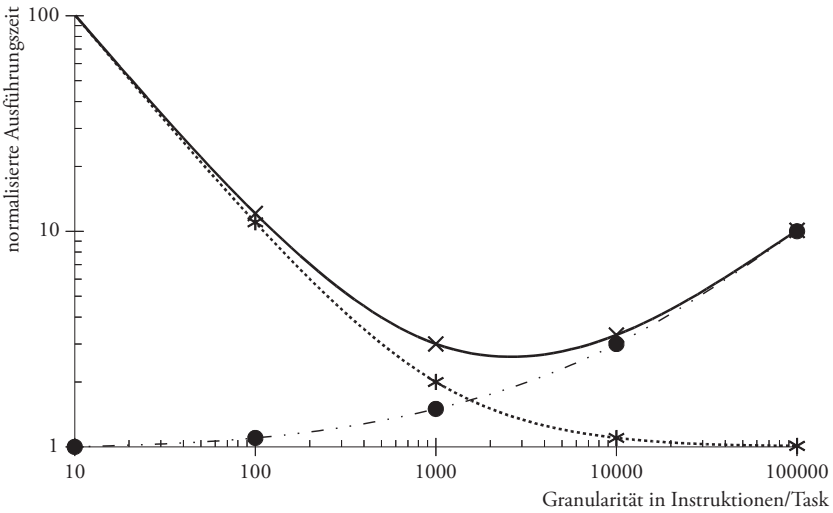


Abbildung 5.1: Normalisierte Ausführungszeit eines parallelisierten Programms über die Granularität; für ein sequentielles Programm mit 100 000 Instruktionen, einem Overhead von 1 000 Takten und einer parallelisierten Ausführung auf 10 Prozessoren. Die Kurve mit •-Markierungen zeigt die ideale normalisierte parallele Ausführungszeit des parallelisierten Programms ohne die Berücksichtigung von Overhead. Die Kurve mit $*$-Markierungen entspricht dem Overhead-Faktor. Die reale Ausführungsdauer des parallelisierten Programms ist durch die Kurve mit $\times$-Markierung dargestellt. (vgl. [Sar89])

Speedup bezeichnet $S_p = \frac{T'}{T_p}$. Wird der Speedup auf die Anzahl der Prozessoren bezogen, so ergibt sich die Effizienz $E_p(\nu) = \frac{S_p(\nu)}{p}$. S_p beschreibt, wie effizient die Prozessoren durch das parallele Programm ausgenutzt werden. Die Gesamtlaufzeit eines Programms ist größer oder gleich der Summe des sequentiellen und parallenen Anteils durch die Anzahl der Prozessoren.

$$T_p(\nu) \geq f \cdot T'(\nu) + \frac{(1-f) \cdot T'(\nu)}{p}$$

Das Amdahlsche Gesetz beschreibt den maximal möglichen Speedup:

$$S_p(\nu) = \frac{T'(\nu)}{T_p(\nu)} = \frac{1}{f + \frac{(1-f)}{p}}$$

Daraus ergibt sich, dass der sequentielle Anteil den maximalen Speedup begrenzt, so ist bei 95% parallelisierbarem Anteil ein maximaler Speedup von 20, bei 90% nur der Faktor 10 möglich. Eine Parallelisierung ist nur für eine kleine Anzahl Prozessoren oder ein kleines f sinnvoll, bzw. für Probleme die sich gut parallelisieren lassen. [BBK+08]

5.1.3 Gustafson-Gesetz

Das Amdahlsche Gesetz geht davon aus, dass die Problemgröße konstant ist. Das Gustafsonsche Gesetz (vgl. [Gus88]) nimmt hingegen eine konstante Laufzeit an. Diese wird durch Erhöhung der Problemgröße und eine variable Anzahl der Prozessoren kompensiert. Es wird also eine Zeit festgelegt, in dem die Problemgröße mit der Anzahl der Prozessoren wächst. Bei doppelter Zahl der Prozessoren können dann idealerweise doppelt so viele Aufgaben in gleicher Zeit bearbeitet werden. Im Gegensatz zu dem Gesetz von Amdahl beschränkt der sequentielle Anteil die Parallelisierung nicht, denn mit steigender Parallelität wird der sequentielle Anteil unbedeutender. Für p gegen Unendlich wächst der Speedup linear.

$$S_p(n) = p * (1 - f) + f$$

Eine Anwendung ist skalierbar, wenn die Aufgabe an die Leistungsfähigkeit des Systems (durch mehr Prozessoren) angepasst werden kann. Dies kann beispielsweise durch feinere Datenmodelle geschehen. [BBK+08]

5.2 Bekannte Programmfragmente

Die Ebene, auf der eine Zerlegung eines Programmsystems möglich ist, hängt unter anderem von dem Schritt des Entwicklungs- bzw, Übersetzungsprozesses, an dem die Zerlegung ansetzt, ab. Daher werden im Folgenden zunächst die Programmbestandteile des Quelltextes, anschließend die des Zwischencodes und der Maschinenebene 5.2.3 betrachtet. Hierbei gilt, dass eine Funktion aus einer Folge von Anweisungen aufgebaut ist, welche durch den Compiler wiederum auf eine Folge von Instruktionen abgebildet wird. Auf die durch den Übersetzungsprozess anfallenden Strukturen folgt eine Betrachtung von in der Literatur verwendeten Strukturen, die aus den zuvor genannten Elementen zusammengesetzt werden.

5.2.1 Funktion

Wie in Kapitel 2.2 erläutert, sind Funktionen eine übliche Art der Strukturierung von Quellcode. Ein Programm wird somit in logische zusammengehörende Teile aufgeteilt. Eine Funktion F kann an mehreren Stellen im Programm, in unterschiedlichen Kontexten, aufgerufen werden. Somit hat eine Funktion keine festen Kontrollflussabhängigkeiten. Ebenso kann der Datenfluss in der Quantität als auch örtlich variieren. Besonders bei der Verwendung von Call-by-Reference sind die Datenquellen nicht statisch.

5.2.2 Anweisung

Eine Anweisung (*statement*) S ist ein Konstrukt höherer Programmiersprachen. Anweisungen können dabei ineinander verschachtelt werden. Durch den Compiler werden diese wie in Kapitel 2 beschrieben mit Maschinenbefehlen umgesetzt.

5.2.3 Instruktion

Instruktionen oder Befehle I, wie sie im Zwischen- bzw. Maschinencode auftreten, sind meist in einen sequentiellen Ablauf eingebettet. Außer bei Sprungzielen, deren Ursprung nicht unbedingt unveränderlich sein muss, ist somit leicht ermittelbar in welchem zeitlichen Ablauf diese auftreten. Die Stellen im Programm können durch Sprungbefehle mehrfach durchlaufen werden. Die Vorgänger und Nachfolger auf Kontrollflussebene bleiben dabei unverändert.

Für jede Instruktion I existieren Datenquellen und Datensenken, in der Regel entsprechen sie der rechten bzw. linken Seite einer Zuweisungsoperation. Bei der Verwendung von Pointern kann der tatsächliche Ort dabei jedoch variieren. Eine Instruktion lässt sich als Tupel $(ergebnis, operation, operanden)$ darstellen. Wobei das $ergebnis$ ein Register, eine Speicherstelle oder ein Bezeichner ist. Die $operanden$ wiederum sind ein Tupel mit variabler Anzahl an Elementen, die von der $operation$ abhängt.

Auf Instruktionsebene ist die Ausführungsreihenfolge primär durch die Reihenfolge der Anweisungen im Programm gegeben (sequentielle Abarbeitung). Durch Kontrollfluss-Anweisungen (z. B. Sprung-Anweisungen) kann von dieser Reihenfolge abgewichen werden.

5.2.4 Basisblock

Ein Basis- oder Grundblock (*basic block*) BB ist eine Folge von Instruktionen des Zwischencodes $(I_1, I_2, \ldots, I_{n-1}, I_n)$. Die Sequenz ist dabei möglichst lang, und weist zwei wesentliche Eigenschaften auf. Erstens kann ein Basisblock nur durch die erste Instruktion betreten werden. Im Programmsystem gibt es also keine Sprünge, die als Ziel die Mitte eines Basisblocks haben. Zweitens wird ein Basisblock nur mit dem letzten Befehl des Blocks verlassen. I_{exit} ist die Menge aller möglicher Sprunginstruktionen, es gilt somit $I_n \in I_{exit}$. Innerhalb eines Basisblocks gibt es keine Verzweigungen oder Unterbrechungen. Von dem Eintrittspunkt eines Basisblocks gehören alle folgenden Instruktionen zu eben diesem bis einschließlich dem nächsten Sprung oder bedingten Sprung.[Tan06, Kap. 8.4]

$$BB = \{I_x^{BB} | 1 \leq x \leq n \wedge x = n \rightarrow I_x \in I_{exit} \wedge x \neq n \rightarrow I_x \notin I_{exit}\}$$

Sprunganweisungen enthalten das Sprungziel meist in der Form einer Sprungmarke (*label*), dementsprechend ist der Anfang eines Basisblocks durch eben eine solche Sprungmarke gekennzeichnet. Die Sprünge selber können bedingt oder unbedingt sein. Ein unbedingter Sprung wird immer ausgeführt, ein bedingter nur, wenn eine im Sprungbefehl angegebene Bedingung zutrifft. Auf Maschinenebene unterscheidet man absolute Sprünge mit festen Sprungzielen (Adresse), relative Sprünge mit einem Ziel relativ zur aktuellen Position, Register-Sprünge, bei denen das Ziel berechnet wird, und Speicher-Sprünge, bei denen das Ziel des Sprungs an einer Position im Speicher hinterlegt ist.

Die Zwischensprache des LLVM verwendet `branch`-Instruktionen für einen unbedingten Sprung sowie als bedingten Sprung mit zwei alternativen Sprungzielen. Ein weiterer Sprungbefehl ist die `switch`-Instruktionen. Sie wählt eines von mehreren Sprung-

zielen aus und verzweigt dorthin. Neben den Sprunginstruktionen endet ein Basisblock auch durch einen Rücksprung (*return*). Das Hauptprogramm (*main*) stellt den Startpunkt der Programmausführung dar. An entsprechenden Stellen der Programmausführung kann eine Funktion aufgerufen werden (*call*). Bevor der Sprung (Sprung mit Rückkehrabsicht) ausgeführt wird, wird die aktuelle Programmposition, bzw. die Adresse des folgenden Befehls, gespeichert um an diese Stelle zurückkehren zu können, wenn die Funktion ausgeführt wurde. Das Beenden der Funktion und das Zurückkehren an die Stelle des Aufrufs wird durch den `return`-Befehl initiiert. `call` und `return` stellen somit prinzipiell eine besondere Art der Sprunginstruktion dar. Eine `call`-Instruktion führt bei LLVM jedoch nicht zum Ende eines Basisblocks. Ein LLVM-IR-Basisblock kann somit mehrere Unterprogrammaufrufe enthalten, endet jedoch immer mit einer `branch`, `switch` oder `return`-Instruktion, somit gilt für die LLVM-IR

$$I_{exit} = \{return, branch, switch, \dots\}.$$

Viele Compiler zerlegen den Zwischencode in derartige Basisblöcke. Die Untergliederung ermöglicht es dem Compiler, den Ablauf eines Programms besser zu ermitteln. Diese Information kann in die Codeerzeugung einfließen. Daraus gewonnene Kenntnisse über die Definition und Referenzierung von Variablen kann die Registervergabe optimieren.

Für zwei Basisblöcke B und C ergibt sich ein Kontrollfluss $B \rightarrow_f C$ für den Fall, dass B mit einem Sprung endet und der Anfang von C das Sprungziel ist.[1] C ist somit der Nachfolger von B bzw. B der Vorgänger von C.

Ein Basisblock kann mehrere Vorgänger und Nachfolger haben. Die Abhängigkeiten lassen sich durch einen gerichteten Graphen darstellen (vgl. Kapitel 3.1). In einem solchen Kontrollflussgraphen werden Basisblöcke als Knoten, die Abhängigkeiten durch Kanten dargestellt. Für Optimierungen bilden Compiler häufig einen Datenflussgraph pro Basisblock. Da ein Basisblock durch einen Eintritts- und einen Austrittspunkt ohne zwischenzeitliche Verzweigungen begrenzt wird, können innerhalb des Basisblocks keine Zyklen auftreten.

5.2.5 Bereichsbildungstechniken

Auf der Grundlage von Basisblöcken und dem Kontrollfluss zwischen ihnen können größere Strukturen, bzw. Einheiten U gebildet werden. Basisblöcke werden zu größeren Einheiten zusammengefügt wie sie in Abbildung 5.2 dargestellt sind.

5.2.5.1 Trace

Eine schleifenfreie Folge von Basisblöcken (Weg im Kontrollflussgraph) wird als Trace bezeichnet. Ein Trace kann sowohl Knoten enthalten, an denen sich der Kontrollfluss

[1] Kann in einer Zwischensprache ein Basisblock ohne eine Kontrollfluss-Instruktion enden, so ergibt sich auch ein Kontrollfluss von B nach C, für den Fall, dass C auf B folgt und B nicht mit einem Sprung endet.

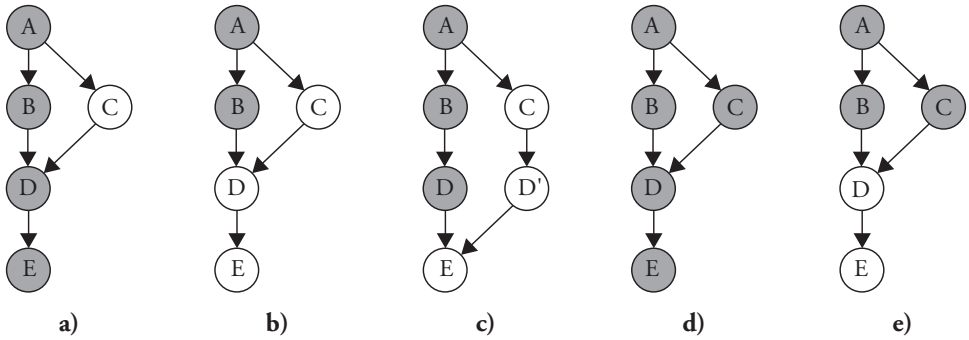


Abbildung 5.2: Beispielhafte Darstellung bekannter Programmfragmente (Trace (a), Superblock (b), Superblock mit Tail-Duplizierung (c), Hyperblock (d), Treegion (e)), welche als Kombination mehrerer Basisblöcke gebildet werden; graue Knoten sind Teil der jeweiligen Einheit.

verzweigt (Seitenausgang), als auch solche, die den Kontrollfluss aus mehreren Quellen zusammenführen (Seiteneingang). Bei Verzweigungen darf allerdings nur ein Nachfolger Teil des Traces sein (Abbildung 5.2a)). Als Eingangsblock wird ein Basisblock gewählt, der nicht Teil eines anderen gebildeten Bereichs ist, und am häufigsten ausgeführt wird. Bei einer Verzweigung wird ein Pfad gewählt. [Fis81]

5.2.5.2 Superblock

Ein Superblock nach [HMC+93] ist ein Trace ohne Seiteneingänge. Punkte an denen der Kontrollfluss von mehreren anderen Orten kommen kann (Merge Point) können zu Problemen führen, da dies meist mit spekulativen Ausführungen einhergeht. Der Kontrollfluss kann den Superblock somit ausschließlich am Anfang betreten. Jedoch ist es möglich den Superblock an mehreren Stellen zu verlassen. Superblöcke werden auf der Basis von Profiling-Informationen gebildet. Wie in Abbildung 5.2c) ersichtlich ist, kann der Bereich eines Superblocks durch Tail-Duplizierung vergrößert werden. Hierbei werden Seiteneingänge eliminiert, indem der folgende Code dupliziert wird.

Für das Instruction Scheduling mittels Superblocks ist es wichtig einen möglichst häufig ausgeführten Trace zu ermitteln. Dabei kann ein Superblock keine parallelen Kontrollflüsse enthalten.

5.2.5.3 Hyperblock

Ein Hyperblock (Abbildung 5.2d)) ist ebenfalls ein Satz bedingter Basisblöcke. Dabei kann dieser Bereich nur am Anfang betreten, jedoch an einer oder mehreren Stellen verlassen werden. Um eine Einheit für Compiler-Optimierungen wie Instruktions-Scheduling oder Register-Zuweisung zu bilden, ist das Ziel, viele Basisblöcke aus mehreren Kontrollflusspfaden zu gruppieren. Ein häufig auftretendes Problem bei der Anwendung von Optimierungen besteht in bedingten Verzweigungen. Diese können mittels If-Conversion (Beispiel siehe Listing 5.1) durch bedingte Anweisungen ersetzt werden.

```
if ( Bedingung )
   a = x;                    ⟶        a = cond ? x : y;
else
   a = y;
```

Listing 5.1: If-Conversion

Aus einem Block kontrollabhängiger Anweisungen werden somit eine Menge datenabhängiger Anweisungen.

Weitere Strategien bei der Bildung von Hyperblocks sind Loop Peeling (die erste/-letzte Iteration wird aus dem Schleifenkörper vor/nach die Schleife verschoben), Tail Duplication und Node Splitting. Sprunginstruktionen werden bei Hyperblocks innerhalb des Blocks in bedingte Vergleiche transformiert. [MLC+92]

5.2.5.4 Treegion

Ein Programm kann in eine Menge von Entscheidungsbäumen oder Treegions zerlegt werden. Jede Treegion ist dabei ein Teilgraph mit baumartiger Struktur eines Kontrollflussgraphen. Die Anzahl und Größe der Treegions hängt nicht von Profil-Informationen ab, sondern nur von der Struktur des Kontrollflussgraphen. Bei den oben aufgeführten Einheiten (Kapitel 5.2.5.1 bis 5.2.5.3) werden die Profil-Informationen genutzt, um häufig betretene Pfade bei der Bildung zu berücksichtigen und entlang dieser Pfade zu optimieren. Heuristiken können genutzt werden um auch Treegions zu vergrößern indem z. B. Tail-Duplizierung angewendet wird. Gebildet werden Treegions indem ausgehend von dem Programmeintrittsknoten dem Kontrollfluss gefolgt wird und alle Basisblöcke in die Treegion aufgenommen werden, bis ein Merge-Point auftritt. Alle Knoten (Basisblöcke) welche Teil der Treegion sind, haben somit genau eine eingehende Kontrollflusskante, mit Ausnahme des Wurzelknotens. Merge-Point-Knoten sind Ausgangspunkte bzw. die Wurzel einer neuen Treegion. Dieses Verfahren wird durchgeführt, bis alle Basisblöcke des Kontrollflussgraphen Teil einer Treegion sind. [BHC97]

5.2.6 Program Slice

Slicing stellt eine Methode dar, Programme automatisch in Fragmente aufzuspalten um den Daten- und Kontrollfluss zu analysieren. Die Analyse für umfangreiche Programmsysteme kann sehr komplex sein. Ermöglicht wird die Analyse der überschaubaren Komponenten durch die Unterteilung des gesamten Problems in eine Reihe kleinerer Teilprobleme. Die Teilergebnisse lassen sich anschließend zu einem Gesamtergebnis zusammenfügen. Ermittelte Informationen werden beispielsweise für das Debugging, Testen, Warten oder zur Umstrukturierung von Programmen verwendet. Erstmals wurde Slicing durch Mark Weiser in [Wei79] beschrieben. Slicing beschreibt das Verfahren zur Berechnung von Program Slices. Eine Übersicht über Konzepte und Methoden zur Bestimmung von Slices findet sich in [Tip94] bzw. [XQZ+05], wie eine Sammlung wichtiger Arbeiten zu dem Thema Program Slicing. Seit dem wurden viele darauf aufbauende Arbeiten

durchgeführt. In [OO84] wurde erstmals eine Methode zur Bestimmung von Program Slices mittels Programmabhängigkeitsgraphen (vgl. Kapitel 3.9.1) vorgestellt. [RRB13] gibt eine Übersicht wie aus PDG mittels Breitensuche (*breadth first search*) oder Tiefensuche (*depth first search*) Slices gebildet werden können.

Ein Program Slice beinhaltet Informationen über die Wirkzusammenhänge innerhalb eines Programms. Konkret wird ein Program Slice als die Menge aller Anweisungen definiert, die eine Variable v an einem bestimmten Ort s im Programm beeinflusst, oder alternativ Orte die von der Variablen beeinflusst werden. Ein Program Slice bezieht sich immer auf ein Slice-Kriterium welches ein Tupel $\langle v, s \rangle$ ist. Wird untersucht welche Anweisungen eine betrachtete Variable beeinflussen, handelt es sich um rückwärts gerichtetes Slicing (*backward slicing*). Dem gegenüber steht vorwärts gerichtetes Slicing (*forward slicing*). Ein solcher Program Slice umfasst alle Anweisungen die durch den betrachteten Wert bzw. das Slice-Kriterium beeinflusst werden.

In [Wei79] wird iterative Datenflussanalyse als Methode zur Bestimmung von Program Slices beschrieben. Ein weiterer, oft verwendeter Ansatz verwendet eine Erreichbarkeitsanalyse in Programmabhängigkeitsgraphen (siehe Kapitel 3.9.1). In diesem Graphen entsprechen Knoten einzelnen Anweisungen des Programms. Relevante Kanten sind zum einen die Datenabhängigkeitskanten, zum anderen Kontrollabhängigkeitskanten. Wenn die Ausführung einer Anweisung von einer anderen kontrolliert wird (z. B. durch Schleifen oder if-Abfragen) treten letztere auf. Für die Berechnung eines Program Slices zu einem gegebenen Slice-Kriterium werden alle rückwärts erreichbaren Knoten dem Program Slice hinzugefügt.

Das Slicing variiert in seiner Komplexität, abhängig von der Struktur und im Programm verwendeter Konstrukte. Der einfachste Fall sind nicht verzweigende Programme, die Ausführung findet rein sequenziell statt. Treten Verzweigungen oder Schleifen auf, handelt es sich um ein strukturiertes Programm. Sowohl für strukturierte als auch unstrukturierte Programme, solche mit `GOTO`- oder `BREAK`-Anweisungen, ist die Berechnung der Program Slices trivial.

Bei der Verwendung von Arrays ist jedoch nicht immer ersichtlich, welche Elemente definiert oder referenziert werden, somit müssen alle Anweisungen, welche potentiellen Einfluss haben, in das Program Slice aufgenommen werden. Treten darüber hinaus Unterprogramme auf, müssen zusätzlich Aufrufabhängigkeiten berücksichtigt werden. Hierbei werden kontext-sensitives und kontext-insensitives Slicing unterschieden. [Kri04] führt hierfür ein Beispiel an. Bei kontext-insensitivem Slicing wird nicht berücksichtigt, in welchem Zusammenhang Unterprogramme aufgerufen werden, und welche Auswirkungen dies auf Abhängigkeiten haben kann.

Für die manuelle Untersuchung von Programmen ergibt sich bei Slicing jedoch das Problem, dass unter Umständen umfangreiche und unübersichtliche Program Slices entstehen. Zudem wird die Verwendbarkeit dadurch beschränkt, wie Slicingverfahren bei umfangreichen Programmsystemen skalieren.

Ein weiteres Klassifizierungsmerkmal der Slicing-Methoden betrifft die mögliche Beachtung einer Startkonfiguration. Diese beschreibt die Menge aller einem Programm übergebenen Parameter sowie aller eingelesener Werte. Erfolgt das Slicing unabhängig von diesen Werten, handelt es sich um statisches Slicing. Werden die Wirkzusammen-

hänge für eine bestimmte Eingangskonstellation ermittelt, handelt es sich um dynamisches Slicing.

Beim statischen rückwärts gerichteten Slicing soll das Program Slice alle Anweisungen umfassen, welche unabhängig von einer Startkonfiguration Einfluss auf das Slice-Kriterium nehmen können. Dies sind im Besonderen:

- bei Programmen ohne Verzweigung alle Anweisungen, die einen direkten oder indirekten Einfluss haben
- bei Programmen mit Verzweigungen oder Schleifen zusätzlich die Kontrollfluss steuernden Anweisungen in deren Rumpf sich Slice Anweisungen befinden
- bei Arrays (ähnliches gilt für Pointer) zusätzlich Anweisungen, die auf das Array zugreifen. Wird nur ein konstanter Index verwendet, entspricht dies einer regulären Variablen. Greift das Slice Kriterium über einen konstanten Index auf das Array zu müssen auch Anweisungen mit variablem Index im Slice berücksichtigt werden. Tritt im Slice-Kriterium ein variabler Index auf, müssen alle das Array beeinflussende Anweisungen berücksichtigt werden.

Beim dynamischen Slicing werden die Wirkzusammenhänge abhängig von einer bestimmten Startkonfiguration zur Laufzeit ermittelt. Zur Berechnung kann das Programm um zusätzliche Instruktionen ergänzt werden. Diese Instruktionen ermöglichen die Ausführung zur Laufzeit zu verfolgen. Alternativ wird der Quelltext interpretiert. Eine Mischform von statischem und dynamischen Slicing stellt das Approximate Dynamic Slicing dar. Da die Berechnung der dynamischen Program Slices sehr aufwändig sein kann und statische Program Slices zu umfangreich werden können, wird versucht, einen geeigneten Mittelweg zu eröffnen. Durch dynamisches Slicing werden Informationen ermittelt, ob bestimmte Abhängigkeiten aufgetreten sind. Hängt der Programmablauf jedoch von Daten ab (Programmverzweigung), so ist nicht ersichtlich warum. Hierzu kann eine Kombination mit dynamischen Pfadbedingungen dienlich sein (vgl. [HGK06]).

[Wei79] verwendet eine iterative Datenflussanalyse zur Bestimmung der Slices. Andere Algorithmen basieren auf Program-Dependence-Graphen. Deren Knoten repräsentieren Anweisungen, Kanten existieren für Kontroll- und Datenabhängigkeiten. Das Slice-Kriterium lässt sich auf einen Knoten abbilden. Dem zugehörigen Program Slice werden alle Knoten hinzugefügt, welche rückwärts erreichbar sind. Für Programme ohne Unterprogrammaufrufe ist diese Vorgehensweise trivial. Wenn im Quelltext jedoch Unterprogramme zur Strukturierung eingesetzt werden, muss auch der Aufrufkontext berücksichtigt werden. Eine Anweisung, auf welche das Slice-Kriterium zeigt, referenziert eine Variable x. Wenn x in einer Funktion F definiert wird, so müssen alle möglichen in Q referenzierten Datenquellen in das Slice aufgenommen werden. (vgl. [Kri03]) Unter Einbeziehung dieser Eigenschaft muss das intraprozedurale Slicing kontext-sensitiv erweitert werden.

Wenn das Slicing kontext-sensitiv ist, dann können nur im Ablauf tatsächlich auftretende Quellen in das Program Slice aufgenommen werden. Die Berechnung lässt sich effizient über Summary-Kanten realisieren (vgl. [Kri03]). In [WWB+08] werden spekulative Ansätze in das Slicing eingebracht um Parallelverarbeitung ausnutzen zu können. Es werden Hotspots des Programms identifiziert und rückwärtsgerichtete Slices bezüglich ihrer Kommunikation zu Threads kombiniert.

5.3 Schleifenrestrukturierung

Schleifenrestrukturierungen dienen in der Regel dazu, ein höheres Optimierungspotential auf Instruktionsebene zu erreichen. Compiler können somit eine effizientere Ausnutzung der Hardwareressourcen erzielen. Häufig werden Berechnungen auf Datenstrukturen iterativ ausgeführt. In der Programmierung wird dies durch Schleifen ausgedrückt. Eine Schleife besteht aus dem Kopf bzw. Fuß, welche die Iteration eines Index über dessen Indexraum übernimmt, und die Abbruchbedingung enthält, sowie dem Schleifenkörper, welcher die mehrfach zu durchlaufenden Anweisungen oder Einheiten enthält. Die klassische Abarbeitung erfolgt sequentiell, dass heisst die i-te Iteration wird nach Beendigung der (i-1)-ten Iteration begonnen. Für eine parallele Bearbeitung wird häufig zwischen forall, dopar und doall unterschieden. Die Bezeichnungen stammen von FORTRAN ab. Eine forall-Schleife verwendet in jeder Iteration ausschließlich die Werte, welche vor Ausführung der Schleife aktuell waren. Sie besteht aus ein oder mehreren Zuweisungen an Feldelemente. Eine dopar-Schleife kann darüber hinaus weitere Anweisungen oder Schleifen beinhalten. Die Iterationen einer dopar-Schleife werden parallel zueinander ausgeführt und verwenden, wie auch die forall-Schleife, nur die vor der Ausführung aktuellen Variablenwerte. Veränderungen während einer Schleifeniteration bleiben vor den anderen Iterationen verborgen. Falls mehrere Iterationen die selbe Variable manipulieren, ergibt sich ein nichtdeterministisches Verhalten. Falls eine dopar-Schleife nur Elemente verwendet, die in der gleichen Iteration manipuliert wurden, wird sie auch als doall-Schleife bezeichnet. Alle Iterationen dieses Schleifentyps sind somit unabhängig voneinander und können parallel ausgeführt werden, ohne dass sich das Verhalten ändert. [RR12]

Schleifen in mehrere unabhängig ausführbare Teile zu zerlegen kann zwei Ansätzen folgen. Die erste Möglichkeit der Unterteilung ist die Loop Distribution. Das Ergebnis für die Schleife aus Listing 5.2 ist in Listing 5.3 abgebildet. Hierbei wird der Schleifenkörper unterteilt, der Indexraum bleibt unverändert. Eine ideale Verteilung ist möglich, wenn keine Datenabhängigkeiten zwischen den Teilen auftreten *und* die Teile den gleichen Arbeitsaufwand erzeugen. Falls Datenabhängigkeiten zwischen Blöcken existieren, ist eine zusätzliche Synchronisierung notwendig, damit der korrekte Fluss vom Produzent zum Konsument realisiert werden kann. Loop Splitting stellt den zweiten Ansatz zur Aufteilung dar. Der Schleifenkörper bleibt unverändert, der Indexraum wird stattdessen unterteilt (Listing 5.4). Eine Form des Loop Splitting ist das Loop Unrolling.

In der Literatur finden sich neben den bereits erwähnten weitere Operationen zum Transformieren von Schleifen. Exemplarisch seien hier genannt:

Statement Reordering sorgt dafür, dass die Reihenfolge der Instruktionen im Schleifenkörper verändert wird. Dies ist nur möglich wenn schleifenunabhängige (*loop independent*) Datenabhängigkeiten vorhanden sind.

Stripmining nutzt die Eigenschaften von Vektor- oder SIMD-Architekturen aus. Eine Schleife wird in zwei geschachtelte Schleifen transformiert, wobei die neue, innere Schleife konstante Schleifengrenzen besitzt. Die äußere Schleife durchläuft den Indexraum in

```
for (i=0; i<N; i++) {
  x[i]=a[i]*a;
  y[i]=x[i]+b;
}
```

Listing 5.2: Einfache Schleife

```
for (i=0; i<N; i++) {
  x[i]=a[i]*a;
}
for (i=0; i<N; i++) {
  y[i]=x[i]+b;
}
```

Listing 5.3: Loop Distribution

```
for (i=0; i<N/2; i++) {
  x[i]=a[i]*a;
  y[i]=x[i]+b;
}
for (i=N/2; i<  N; i++) {
  x[i]=a[i]*a;
  y[i]=x[i]+b;
}
```

Listing 5.4: Loop Splitting

Blöcken. Jeder Block wird durch die innere Schleife durchlaufen. Bei Vektorrechnern oder SIMD Prozessoren entspricht der Umfang der inneren Schleife der SIMD-Weite bzw. Vektorlänge der Zielarchitektur.

```
for (i = 0; i < N; i++)
  a[i]=f[i];
```

⟶

```
for (j = 0; j < N; j+=32)
  for (i = j; i < min(j+32,N); i++)
    a[i]=f[i];
```

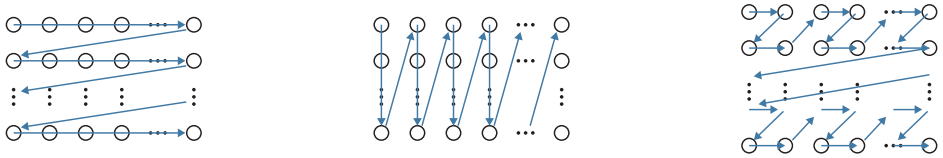
Interchange oder Schleifenvertauschung tauscht die Reihenfolge geschachtelter Schleifen, die innere wird somit zur äußeren und umgekehrt. Für mehr als zwei Schleifen spricht man von Schleifenpermutation (siehe Abbildung 5.4).

Tiling ist eine Kombination aus Stripmining und Interchange in mehrfach verschachtelten Schleifen. n Schleifen werden in $2n$ Schleifen umgewandelt indem Kacheln im ursprünglichen Indexraum gebildet werden (siehe Abbildung 5.4). Die Blockgröße im folgenden Beispiel liegt bei 16.

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
      x[i] = a[i][j] * b[j];
```

⟶

```
for (ii = 0; i < N; i += 16)
  for (jj = 0; jj < M; jj += 16)
      for (i = ii; i < min(ii+16,N); i++)
        for (j = jj; j < min(jj+16,M); j++)
            x[i] = a[i][j] * b[j];
```



a) b) c)

Abbildung 5.4: Ausführungsreihenfolge, Iteration über zweidimensonales Array: a) Ursprüngliche Reihenfolge, b) Reihenfolge nach Interchange, c) Reihenfolge nach Tiling

Fusion verschmilzt mehrere Schleifen mit identischen Schleifengrenzen zu einer einzigen Schleife.

```
for (i = 0; i < N; i++) {
  x[i] = a[i] + b[i];
}
for (j = 0; j < N; j++) {
  y[j] = c[j] * 2;
}
```

⟶

```
for (i = 0; i < N; i++) {
  x[i] = a[i] + b[i];
  y[i] = c[i] * 2;
}
```

Fission teilt eine Schleife auf. Durch diese Transformation ist es möglich, existierende Abhängigkeiten zwischen Iterationen zu eliminieren.

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    x[i][j] = a[i][j] + b[i][j];
    o[i][j] = x[i][j] * 2;
  }
}
```

⟶

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    x[i][j] = a[i][j] + b[i][j];
  }
  for (j = 0; j < N; j++) {
    o[i][j] = x[i][j] * 2;
  }
}
```

Unrolling als eine Form der Indexraum-Transformation kombiniert mehrere Iterationen und reduziert somit die Anzahl der Schleifendurchläufe.

```
for (i = 0; i < N; i++)
  sum += A[i];
```

⟶

```
for (i = 0; i < N; i+=4) {
  sum += A[i];
  sum += A[i+1];
  sum += A[i+2];
  sum += A[i+3];
}
```

5.4 Pipelining

Um mehrere Verarbeitungseinheiten gleichzeitig zu beschäftigen ist es möglich, eine funktionale Zerlegung des Gesamtsystems zu vollziehen. Dabei erfolgt eine Unterteilung in einzelne Arbeitsschritte bzw. Aufgaben. Jede dieser Aufgaben bearbeitet einen unabhängigen Teilbereich des ursprünglichen Gesamtsystems. Da die Teilbereiche voneinander unabhängig bearbeitet werden können, sind sie parallel ausführbar. Wenn Datenabhängigkeiten zwischen Bearbeitungsschritten existieren und es einen Datenstrom gibt, welcher alle Arbeitsschritte durchläuft, dann handelt es sich um eine Pipeline oder Fließbandverarbeitung. In [Kog81] wurde Pipelining in Computerarchitekturen erstmalig behandelt.

Die Idee des Softwarepipelining ist es, den Schleifenkörper so umzuorganisieren, dass die nächste Schleifeniteration bereits gestartet werden kann, bevor die vorherige abgeschlossen wurde. [LA92] Jedes Teilproblem verarbeitet die Ergebnisse des vorhergehenden, mit Ausnahme des ersten und letzten. Am Anfang werden die Daten wie bei einer einheitlichen Verarbeitung angenommen, und am Ende werden die äquivalenten

Ergebnisse geliefert. Die Teilprobleme werden auch als Stufen, welche nacheinander ausgeführt werden, bezeichnet.

Eine Realisierung von Pipelining ist das in [RVV+04] vorgestellte Decoupled Software Pipelining (DSWP). Im Gegensatz zu früheren Ansätzen wird hierbei eine Aufteilung in mehrere nicht spekulative Threads vorgenommen. Schleifen werden bei DSWP in umfangreiche Threads, in Anlehnung an Kahn Process Networks (KPN, vgl. [Kah74]), segmentiert. Die benötigten Kommunikations- und Synchronisierungsmethoden führen jedoch zu einem Overhead (Mehraufwand), welcher den Gewinn reduziert oder wieder zunichte macht. Daher wurde ein Synchronisierungsarray vorgeschlagen.

[ORS+05] beschreibt den Algorithmus zur Bildung von DSWPs bestehend aus 4 Schritten.

1. Zunächst wird der Program-Dependence-Graph (PDG) gebildet. Die Instruktionen des Programms bilden die Knoten des gerichteten Graphen. Kanten repräsentieren die Daten- und Kontrollabhängigkeiten. Ausgabe- und Antiabhängigkeiten können ignoriert werden, da eine Verteilung auf mehrere Threads auch die Verwendung unterschiedlicher Register mit sich bringt.
2. Auf dieser Basis wird der PDG partitioniert indem starke Zusammenhangskomponenten (SCC) gesucht werden. Das Ergebnis ist ein Graph mit Knoten, die eine Folge von Instruktionen umfassen. Dieser Graph ist gerichtet und azyklisch.
3. Der Code muss daraufhin aufgeteilt und einzelnen Threads zugeordnet werden. Hierbei wird ein heuristischer Lastenausgleich angesetzt.
4. Zuletzt ist es erforderlich, Threads zu synchronisieren und Daten untereinander auszutauschen. Dementsprechend werden Produzenten- und Konsumenten-Anweisungen hinzugefügt. Datenabhängigkeiten werden somit zu Datenflüssen, und Kontrollabhängigkeiten zu Flags, welche den weiteren Verlauf kennzeichnen. Entsprechend ihrer Position in der Schleife lässt sich zwischen Durchfluss (Loop Flow), Anfang (Initial Flow) und Ende (Final Flow) unterscheiden. Anfang und Ende sind jeweils nur in eine Richtung an die Pipeline gebunden.

Eine Erweiterung dieser Technik wurde in [ROR+08] präsentiert. Zusätzlich zu der reinen Ausnutzung von Pipelinestrukturen wird in Parallel Stage Decoupled Software Pipelining (PS-DSWP) Datenparallelität in den einzelnen Pipelinestufen ausgenutzt. Es handelt sich auch hierbei um eine nicht spekulative Maßnahme. Falls keine schleifengetragenen Abhängigkeiten in den starken Zusammenhangskomponenten existieren, die einzelnen Iterationen also unabhängig voneinander sind, ist es möglich, diese zu replizieren. Es können dann mehrere Threads für eine Pipelinestufe gleichzeitig ausgeführt werden, diese führen dann parallel mehrere Iterationen aus. In [LPC11] wird eine Implementierung, mit den Ziel OpenMP-Anwendungen zu erzeugen, vorgestellt. Dabei kommen zwei Zwischenrepräsentationen in SSA-Form zum Einsatz, zum einen ein Baum welcher Kontrollabhängigkeiten (bedingt durch Verzweigungen und Schleifen) enthält, zum anderen eine Ergänzung um Datenflussinformationen zwischen Bereichen mit einem einzigen Eintrittspunkt und einem einzigen Austrittspunkt. Das Konzept basiert dabei auf Treegion, da davon ausgegangen wird, dass der Kontrollabhängigkeitsgraph einen Baum bildet.

Kapitel 6

Fragmentierungsmodell für die Identifikation potentieller Parallelität

Das Ziel dieses Kapitels ist es, die Grundlage für eine Untersuchung auf Parallelität innerhalb sequentieller Programme zu schaffen. In Kapitel 8 werden dafür geeignete Methoden auf Basis des hier beschriebenen Modells behandelt.

Der erste Schritt hin zu einer vorschriftbasierten Zerlegung, bzw. einer automatisierten Parallelisierung, besteht darin, das ursprüngliche Programm in Einheiten mit statischen Kontroll- und Datenabhängigkeiten zu zerlegen. Die Unveränderlichkeit der Abhängigkeiten bezieht sich dabei auf die Laufzeit der Einheit. Eine Funktion kann beispielsweise durch Unterprogrammaufrufe unterbrochen werden, und in dieser Zeit können Referenzen extern verändert werden. Während eine Einheit aktiv ist, wird die Konstanz der Kontroll- und Datenflusskanten gefordert. Somit wird es möglich, die beschriebenen Voraussetzungen zu erfüllen. Aus den Schritten ergibt sich, dass eine Zerlegung einhergehen kann mit einer anschliessenden Gruppierung kleiner zu gröberen Einheiten. Zunächst wird in diesem Kapitel das Konzept möglicher Zerlegungen erarbeitet. Darauf folgen potentielle Realisierungen sowie eine Diskusion ihrer Eignung in Hinblick auf eine Parallelisierung. Mögliche Realisierungen ergeben sich aus den teilweise zuvor im Stand der Technik beschriebenen Programmfragmenten bzw. bekannten Modellen. Das Novum dieses Kapitels bildet die Betrachtung und Untersuchung dieser Modelle in dem speziellen Kontext dieser Arbeit. Darüber hinaus werden Modell-Erweiterungen diskutiert.

6.1 Konzept

Die zuvor erläuterten Programmfragmente bieten eine Zerlegung eines Programmsystems auf unterschiedlichen Ebenen und somit in unterschiedlicher Granularität an. Darauf basierende Ansätze zur Optimierung (z. B. um die Befehlsebenenparallelität zu er-

höhen wurden bereits genannt). Zusammen mit den in Kapitel 3 beschriebenen Abhängigkeiten und Techniken der Analyse soll das Potential von Parallelität auf höherer Ebene untersucht werden. Als Grundlage sollen logische Einheiten dienen, die konstante und bekannte Abhängigkeiten untereinander aufweisen. Alle zwischen diesen Blöcken existierenden relevanten Abhängigkeiten müssen zu ermitteln sein. Bei Realisierung parallelisierter Programme spielt die Granularität der Einheiten eine Rolle.

Zunächst sind für das Ziel relevante Abhängigkeiten festzulegen. Daraufhin können bekannte Programmfragmente auf ihre Eignung hin untersucht werden. Während des Übersetzungsprozesses vom Quellcode hin zum ausführbaren Programm existieren mehrere mögliche Ansatzpunkte um ein Programm zu fragmentieren (Quellcode-, Zwischencode- oder Maschinencode-Ebene).

Ein Programm dient der Datenverarbeitung. Damit die Funktionalität eines Programms unverändert bleibt, müssen die auf Daten angewandte Operationen nach der Restrukturierung in gleicher Reihenfolge auf die identischen Daten angewandt werden. Der Datenfluss ist also ein zentraler Aspekt, den es zu erhalten gilt. Da einige Operationen nur unter bestimmten Bedingungen ausgeführt werden, ist in diesen Fällen der Kontrollfluss bzw. die Kontrollabhängigkeit zu beachten.

Wie bereits aufgeführt, gibt es verschiedene Zeitpunkte während des Übersetzungsprozesses um die Fragmentierung und Restrukturierung anzusetzen. Quellcode als Ausgangspunkt hat den Nachteil, dass Funktionen und komplexe Instruktionen auftreten können. Innerhalb einer Funktion können viele Kontrollabhängigkeiten existieren und Quellen oder Ziele manipuliert werden, so dass die erwünschte Bedingung der festen Abhängigkeiten nicht gegeben ist. [RVD07] beschreibt einen Ansatz welcher auf Funktionsebene aus Laufzeitprofilen Datenflussgraphen und Informationen über geteilte Daten erzeugt; daraus können spekulativ Datenströme abgeleitet werden. Maschinencode hat den Nachteil, dass das Programm bereits an eine Zielarchitektur gebunden ist. Für die Ausnutzung der zu ermittelnden Parallelität wird das Programm jedoch auf mehrere Prozessoren verteilt werden. Selbst wenn eine einheitliche Zielarchitektur zum Einsatz kommt, so wird dennoch die vom Compiler durchgeführte Registervergabe nicht passend sein. Daraus ergibt sich, dass der Zwischencode eine geeignete Basis bietet. Die einfache Struktur der Instruktionen ermöglicht die Analyse des Kontroll- und Datenflusses zwischen kleinen Programmeinheiten auf der einen Seite. Auf der anderen Seite sind die Instruktionen für eine Manipulation abstrakt genug.

Wie in Kapitel 2.3.2 besteht der Zwischencode aus Instruktionen, die der Wertzuweisung dienen, aus solchen, die den Kontrollfluss steuern und Instruktionen um Unterprogramme aufzurufen. Eine Zuweisungsinstruktion liest und schreibt Daten. In der Ausführung sind im Augenblick der Ausführung dieser Operation Daten-Quellen und -Ziele bekannt und unveränderlich. Ebenso gilt dies für Kontrollfluss steuernde Anweisungen. Nicht statische Orte, welche über Zeiger referenziert werden, müssen zuvor aufgelöst worden sein. Die Problematik der Analyse in diesen Fällen wurde in Kapitel 3.10 und 3.11 erörtert. An dieser Stelle ist davon auszugehen, dass die Kontroll- und Datenflussanalyse für jeden Programmpunkt bzw. Kontext Datenquellen und -senken sowie Ziele des Kontrollflusses ermittelt. Dann lässt sich auf Instruktionsebene ein gerichteter Graph mit Daten- und Kontrollflusskanten bilden.

Für die Restrukturierung und Umsetzung von Parallelität auf Thread-Ebene sind Instruktionen jedoch zu feingranular. Die nächstgrößere Einheit ist der Basisblock. Nach der Definition hat ein Basisblock je genau einen Eintritts- und Austrittspunkt. Wird der Basisblock betreten, werden somit alle Instruktionen innerhalb ausgeführt. Für die Betrachtung des Kontrollflusses kann der Basisblock als eine feste Einheit angesehen werden, es wird die Anforderung nach festen Kontrollflussabhängigkeiten somit erfüllt. Da jede Instruktion innerhalb des Basisblocks genau einmal ausgeführt wird, sobald der Basisblock betreten wurde, ist ebenso eindeutig welche Daten referenziert und welche durch den Basisblock definiert werden. Auch für den Datenfluss lässt er sich also als eine Einheit betrachten.

Die Realisierung von Basisblöcken im LLVM-Compiler behandelt Unterprogrammaufrufe nicht als potentielle letzte Instruktionen. Eine `call`-Instruktion wird als reguläre Instruktion innerhalb eines Basisblocks behandelt. Wie in 2.2 erläutert, kann jedes Unterprogramm potentiell die Ziele von Zeigern ändern, und somit auch die Datenquelle oder -senke des aufrufenden Basisblocks. In dieser Realisierung wird die Bedingung konstanter Datenabhängigkeiten durch einen Basisblock nicht erfüllt.

Ein Basisblock welcher als

$$BB = \{I_x^{BB} | 1 \leq x \leq n \wedge x = n \rightarrow I_x \in I_{exit} \wedge x \neq n \rightarrow I_x \notin I_{exit}\}$$

mit

$$I_{exit} = \{return, branch, switch, call, \dots\}$$

wie in Kapitel 5.2.4 definiert wäre, hätte genau einen Eintritts- und Austrittspunkt. Zudem könnten innerhalb der Anweisungssequenz keine Seiteneffekte auftreten. Zu dem Zeitpunkt, an dem ein solcher Basisblock betreten wird, stehen die Ausgaben fest, sie sind nur von den Eingangsdaten abhängig. Diese Programmeinheit erfüllt somit die geforderte Bedingung zur Restrukturierung.

Eine Einheit, wie sie gefordert ist, lässt sich als vektorwertige Funktion darstellen, welche i Eingangsdaten auf j Ausgangsdaten abbildet. Zwei solcher Funktionen ρ_1 und ρ_2 lassen sich zu einer komplexeren Funktion $\sigma = \rho_1 \circ \rho_2$ kombinieren, wenn gilt:

$$\begin{aligned}
&\rho_1 : \mathbf{x_1} \rightarrow \mathbf{y_1} \quad \text{mit } \mathbf{x_1} = \{x_{11}, \dots x_{1i}\}, \mathbf{y_1} = \{y_{11}, \dots y_{1j}\} \\
&\rho_2 : \mathbf{x_2} \rightarrow \mathbf{y_2} \quad \text{mit } \mathbf{x_2} = \{y_{21}, \dots y_{2k}\}, \mathbf{y_2} = \{y_{21}, \dots y_{2l}\} \\
\Rightarrow\ &\sigma : \mathbf{x}_\sigma \rightarrow \mathbf{y}_\sigma \\
\text{mit}\quad &\mathbf{x}_\sigma = \mathbf{x_1} \cup (\mathbf{x_2} \setminus (\mathbf{x_2} \cap \mathbf{y_1})) \\
&\mathbf{y}_\sigma = \mathbf{y_1} \cup \mathbf{y_2}
\end{aligned}$$

Die neue Einheit erhält als Eingangsdaten alle Werte welche die erste Funktion benötigt, und die der zweiten Funktion welche nicht durch die erste bestimmt werden. Alle ermittelten Werte beider Funktionen können außerhalb erneut ihre Verwendung finden und sind somit vollständig als Ausgabevektor zu betrachten.

6.2 Mögliche Realisierungen

Das zuvor entwickelte Konzept beschreibt eine Gruppierung einzelner Instruktionen zu Sequenzen. Wie zuvor dargelegt, erfüllen LLVM-Basisblöcke die geforderten Kriterien nicht. Jedoch können die durch das Konzept beschriebenen Eigenschaften von verschiedenen Fragmentierungsregeln erfüllt werden. Es ergeben sich Strukturen mit unterschiedlicher Granularität. Hinsichtlich des Nutzens für die Parallelisierung ist dieser Aspekt zu berücksichtigen. Mittels eines Beispielprogramms (Listing 6.1) werden die Konzepte exemplarisch verdeutlicht. Dieses Beispiel weist eine geringe Komplexität auf, und ermöglicht dennoch wesentliche Aspekte des Fragmentierungsansatzes deutlich zu machen. Daher wird diese Anwendung im weiteren Verlauf als exemplarische Ausgangsbasis herangezogen. Die Übersetzung mittels des LLVM-Compilers in der Version 7.0.0 per `llvm-gcc -S -emit-llvm quelltext.c` liefert den Zwischencode. Das Resultat für die transformierte `main`-Funktion ist in Listing 6.2 aufgeführt. Der erste Basisblock enthält die Variablendeklarationen und -definitionen. In den Zeile 127, 132 und 138 werden die Funktionen `open` und `atoi` aufgerufen. Die `while`-Schleife wird durch die zwei folgenden Basisblöcke realisiert. Der erste enthält die Instruktionen zur Schleifensteuerung (`read`-Funktionsaufruf, Prüfen der Abbruchbedingung). Der Basisblock ab Zeile 153 umfasst den Schleifenkörper (Aufruf der Funktionen `foobar` und `write`). Der letzte Basisblock enthält den Aufruf der `close`-Funktion. Anhand dieses Ausschnitts lassen sich die folgenden Schritte der Fragmentierung nachvollziehen.

6.2.1 Atomare Blöcke

Wenn ein Basisblock `call`-Instruktionen enthält, und diese nicht die letzte Instruktion des Basisblocks ist, dann ist eine Transformation notwendig. Ausgehend von der Definition der LLVM-Basisblöcke werden Basisblöcke an den Stellen, an denen `call`-Instruktionen existieren, unterteilt. Jeder Basisblock mit N Unterprogrammaufrufen zerfällt dann in $N + 1$ Subblöcke. Abbildung 6.1 zeigt den Zerfall eines Basisblocks in Subblöcke für eine hypothetische Funktion. Daraus ergibt sich ein Subblock als größtmögliche Instruktionssequenz mit genau einem Eintrittspunkt und einer in der Menge und Reihenfolge definierten Instruktionssequenz.

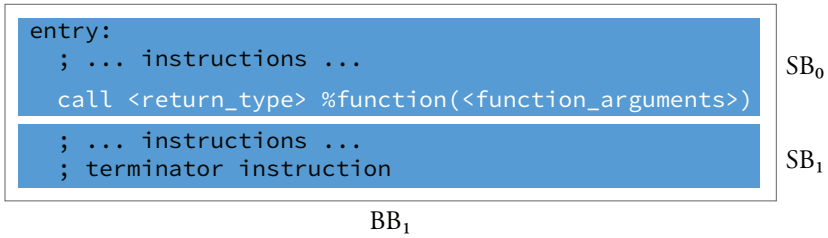


Abbildung 6.1: Zerfall eines LLVM-Basisblocks BB_1 in zwei Subblöcke SB_0 und SB_1 durch Unterteilung an der call-Instruktion

```
struct stuff {
  int cnt;
  char buf[1024];
};

volatile void foo (char *buf_foo, int len_foo, char fac_foo) {
  int i;
  for (i = 0; i < len_foo; i++)
    buf_foo[i] *= fac_foo;
}

volatile void bar (char *buf_bar, int len_bar, char off_bar) {
  int i;
  for (i = 0; i < len_bar; i++)
    buf_bar[i] += off_bar;
}

volatile void foobar (char *buf_foobar, int len_foobar, char fac_foobar,
                      char off_foobar) {
  foo (buf_foobar, len_foobar, fac_foobar);
  bar (buf_foobar, len_foobar, off_foobar);
}

int main (int ac, char **av) {
  int fd;
  struct stuff st;
  char fac, off;
  fd = open (av[1], O_RDONLY);
  fac = (char)atoi(av[2]);
  off = (char)atoi(av[3]);
  while ((st.cnt = read (fd, st.buf, 1024)) > 0) {
    foobar (st.buf, st.cnt - 24, fac, off);
    write(2, st.buf, st.cnt);
  }
  close(fd);
}
```

Listing 6.1: Beispielanwendung zur linearen Datentransformation – Quellcode

In der realen Ausführung können durch Compiler-Optimierungen Anweisungen restrukturiert werden, das Verhalten des Gesammtsystems ändert sich dadurch jedoch nicht, so dass dieser Schritt hier keine Bedeutung hat. Eine Fragmentierung in Subblöcke kann als Abbildung des Programms auf atomare Einheiten angesehen werden. Dabei bleiben Funktionsgrenzen erhalten. Atomarität bezüglich der Abgeschlossenheit in der Ausführung kann jedoch auch über Funktionsgrenzen oder Gruppen von Subblöcken hinweg auftreten. Daraus ergeben sich molekulare Fragmente wie sie im Abschnitt 6.2.2 betrachtet werden. Die aus der Beispielanwendung entstehenden Basisblöcke und Subblöcke sowie alle möglichen Kontrollflusskanten sind in Abbildung 6.2 aufgezeigt. Hierbei ist die Hierarchie durch die Zerlegung von Funktionen in Basisblöcke (BB), und falls vorhanden in Subblöcke (SB) erkennbar. Funktionen in weißen abgerundeten Rechtecken sind externe Bibliotheksaufrufe, deren interner Aufbau nicht zugänglich ist. Externe Bibliothekten liegen in der Regel in kompilierter Form und nicht als Quelltext vor, nicht für alle Bibliotheken ist der Quelltext frei verfügbar.

```
113 define i32 @main(i32 %ac, i8** %av) #0 {
114   %1 = alloca i32, align 4
115   %2 = alloca i32, align 4
116   %3 = alloca i8**, align 8
117   %fd = alloca i32, align 4
118   %st = alloca %struct.stuff, align 4
119   %fac = alloca i8, align 1
120   %off = alloca i8, align 1
121   store i32 0, i32* %1
122   store i32 %ac, i32* %2, align 4
123   store i8** %av, i8*** %3, align 8
124   %4 = load i8*** %3, align 8
125   %5 = getelementptr inbounds i8** %4, i64 1
126   %6 = load i8** %5, align 8
127   %7 = call i32 (i8*, i32, ...)* @"\01_open"(i8* %6, i32 0)
128   store i32 %7, i32* %fd, align 4
129   %8 = load i8*** %3, align 8
130   %9 = getelementptr inbounds i8** %8, i64 2
131   %10 = load i8** %9, align 8
132   %11 = call i32 @atoi(i8* %10)
133   %12 = trunc i32 %11 to i8
134   store i8 %12, i8* %fac, align 1
135   %13 = load i8*** %3, align 8
136   %14 = getelementptr inbounds i8** %13, i64 3
137   %15 = load i8** %14, align 8
138   %16 = call i32 @atoi(i8* %15)
139   %17 = trunc i32 %16 to i8
140   store i8 %17, i8* %off, align 1
141   br label %18

143 ; <label>:18                                     ; preds = %25, %0
144   %19 = load i32* %fd, align 4
145   %20 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 1
146   %21 = getelementptr inbounds [1024 x i8]* %20, i32 0, i32 0
147   %22 = call i32 (i32, i8*, i32, ...)* bitcast (i32 (...)* @read to i32 (i32, i8*,   ↩
        i32, ...)*)(i32 %19, i8* %21, i32 1024)
148   %23 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 0
149   store i32 %22, i32* %23, align 4
150   %24 = icmp sgt i32 %22, 0
151   br i1 %24, label %25, label %38

153 ; <label>:25                                     ; preds = %18
154   %26 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 1
155   %27 = getelementptr inbounds [1024 x i8]* %26, i32 0, i32 0
156   %28 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 0
157   %29 = load i32* %28, align 4
158   %30 = sub nsw i32 %29, 24
159   %31 = load i8* %fac, align 1
160   %32 = load i8* %off, align 1
161   call void @foobar(i8* %27, i32 %30, i8 signext %31, i8 signext %32)
162   %33 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 1
163   %34 = getelementptr inbounds [1024 x i8]* %33, i32 0, i32 0
164   %35 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 0
165   %36 = load i32* %35, align 4
166   %37 = call i32 (i32, i8*, i32, ...)* bitcast (i32 (...)* @write to i32 (i32, i8*,  ↩
        i32, ...)*)(i32 2, i8* %34, i32 %36)
167   br label %18

169 ; <label>:38                                     ; preds = %18
170   %39 = load i32* %fd, align 4
171   %40 = call i32 (i32, ...)* bitcast (i32 (...)* @close to i32 (i32, ...)*)(i32   ↩
        %39)
172   %41 = load i32* %1
173   ret i32 %41
174 }
```

Listing 6.2: Beispielanwendung zur linearen Datentransformation – Zwischencode der main-Funktion

Definition Block

Im Folgenden wird ein Block B als Bezeichnung für die größtmögliche sequentielle Folge von Instruktionen verwendet, bei der ausschließlich die letzte Instruktion eine den Kontrollfluss steuernde Anweisungen (Sprung, Funktionsaufruf, …) sein darf. Somit ergibt sich eine Instruktionssequenz zwischen dem Basisblock-Label bzw. einer `call`-Instruktion am Anfang einerseits und andererseits einer `call`-, `branch`-, `switch`- oder `return`-Instruktion am Ende. Im Falle des LLVM-Zwischencodes ist ein Block also ein Subblock oder bei Abwesenheit von `call`-Instruktionen ein vollständiger Basisbock.

$$
\begin{aligned}
& B = (Label, I_1, I_2, \ldots I_n) \\
\text{mit } & I_n \in I_{exit} \\
& (I_1, \ldots I_{n-1}) \notin I_{exit} \\
& I_{exit} = \{return, branch, switch, call\}
\end{aligned}
$$

Der Basisblock ab Label `; <label>:18` (Zeilen 143–151) enthält einen Funktionsaufruf in Zeile 147, welcher den Kontrollflussfaden an `read` übergibt. Die Aufspaltung dieses Basisblocks führt zu zwei Subblöcken wie in Listing 6.3 dargestellt.

```
; <label>:18                                      ; preds = %25, %0
  %19 = load i32* %fd, align 4
  %20 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 1
  %21 = getelementptr inbounds [1024 x i8]* %20, i32 0, i32 0
  %22 = call i32 (i32, i8*, i32, ...)* bitcast (i32 (...)* @read to   ↩
      i32 (i32, i8*, i32, ...)*)(i32 %19, i8* %21, i32 1024)
  br label %18_BB1SB1

; <label>:18_BB1SB1                                  ; preds = %25,   ↩
    %0
  %23 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 0
  store i32 %22, i32* %23, align 4
  %24 = icmp sgt i32 %22, 0
  br i1 %24, label %25, label %38
```

Listing 6.3: Beispiel für zwei durch Aufspaltung eines Blocks entstandene Subblöcke

6.2.2 Function Slice

Bereits bei trivialen Programmen zerfällt eine Anwendung bzw. Funktion in eine Vielzahl von Subblöcken oder Basisblöcken. Eine im Rahmen dieser Arbeit durchgeführten Untersuchung des Quellcodes gängiger Anwendungen zeigt, dass ein Block nur sehr selten mehr als 50 Instruktionen umfasst.

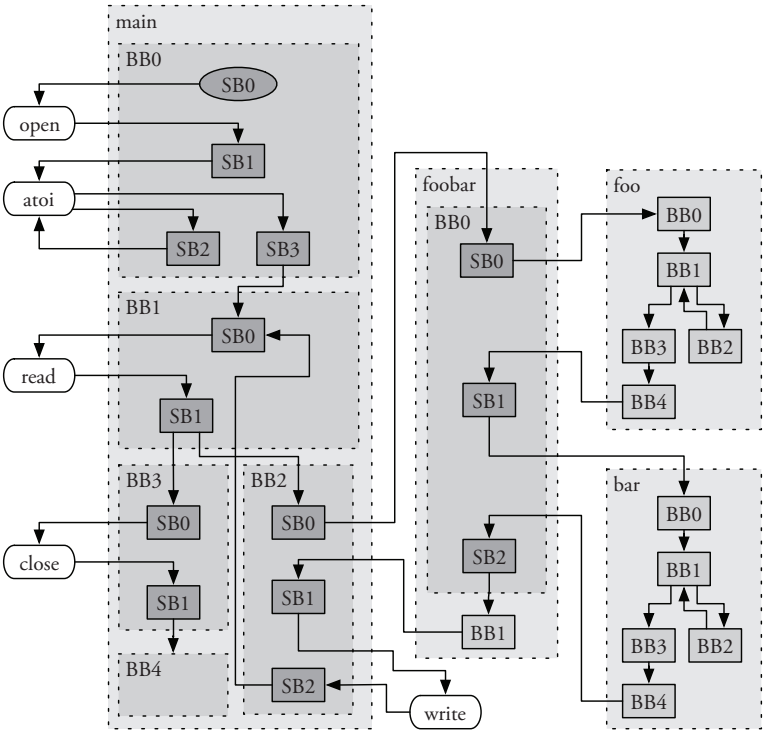


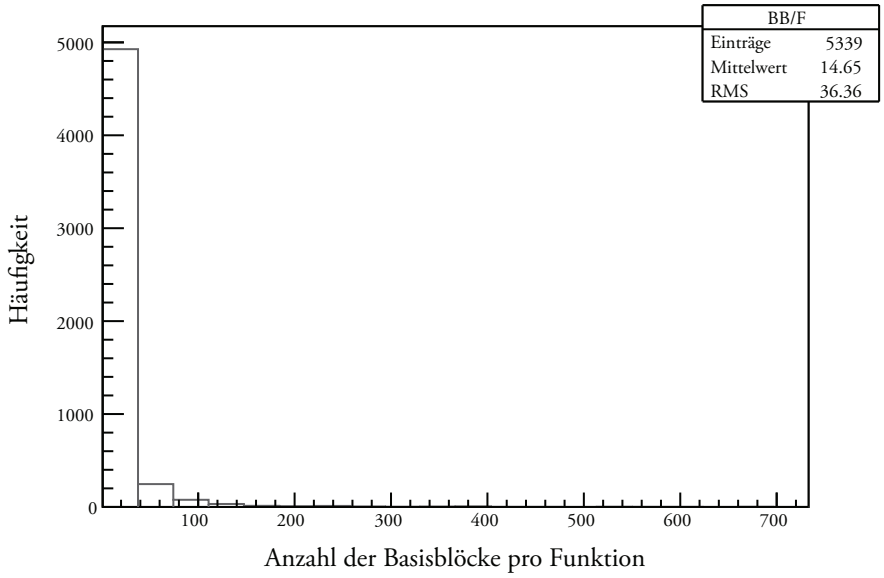
Abbildung 6.2: Beispielanwendung zur linearen Datentransformation – Kontrollfluss auf Blockebene

Untersucht wurde der Code von 14 Open-Source-Projekten[1] mit insgesamt 5339 Funktionen. Die Abbildungen 6.3a) bzw. b) zeigen, in wie viele Basisblöcke bzw. Blöcke Funktionen zerfallen. Wie in Abbildung 6.4a) zu sehen ist, umfasst ein Block im Mittel 1,2 Instruktionen und ist somit im Mittel halb so groß wie ein Basisblock. Häufig folgen viele Unterprogrammaufrufe aufeinander, teils getrennt durch Ladeoperationen für Parameterwerte. In diesen Fällen entstehen Blöcke mit 1 oder 2 Instruktionen pro Block.

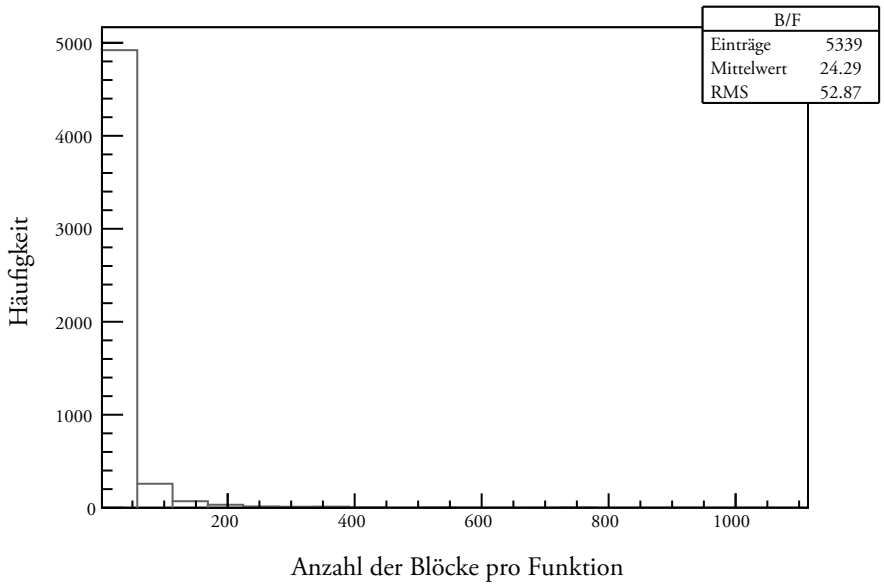
Die ideale Größe im Beispiel aus Abbildung 5.1 ist mit ungefähr 1000 um eine bis drei Größenordnungen größer als die tatsächliche Größe der Blöcke. Auch wenn die optimale Granularität von den bereits genannten Faktoren abhängt, so ist sie in der Regel dennoch größer als die Blockgröße. Daher wird im Folgenden die Möglichkeit betrachtet, Blöcke zu Block-Sequenzen zu kombinieren, ohne die Randbedingung der festen Abhängigkeiten zu verletzen. Für darauf basierende Operationen ergibt sich zudem eine Komplexitätsreduktion.

Wie oben erläutert, stellen Funktionsgrenzen eine Hürde bei der Bildung von geeigneten Programmeinheiten dar. Eine Kombination von Blöcken führt somit zu einer Sequenz innerhalb einer Funktion. Da es sich dabei um einen Teil einer Funktion handelt, wird diese Einheit Function Slice FS genannt. Der Programmablauf wird somit nicht durch andere Funktionen unterbrochen und Abhängigkeiten können folglich während

[1] bc, bison, diffutils, flex, iftop irssi, mtr, multitail, mutt, pv, rsync, screen, siege und wget

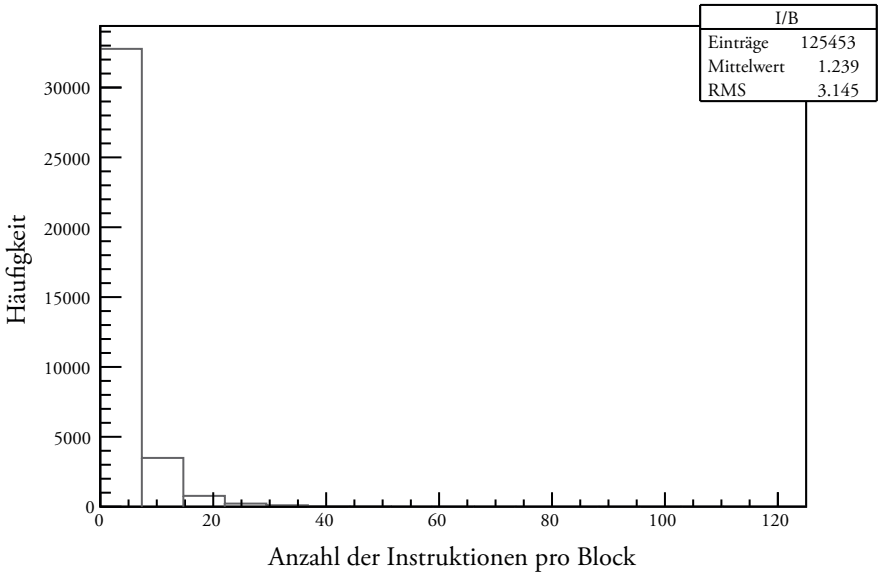


a)

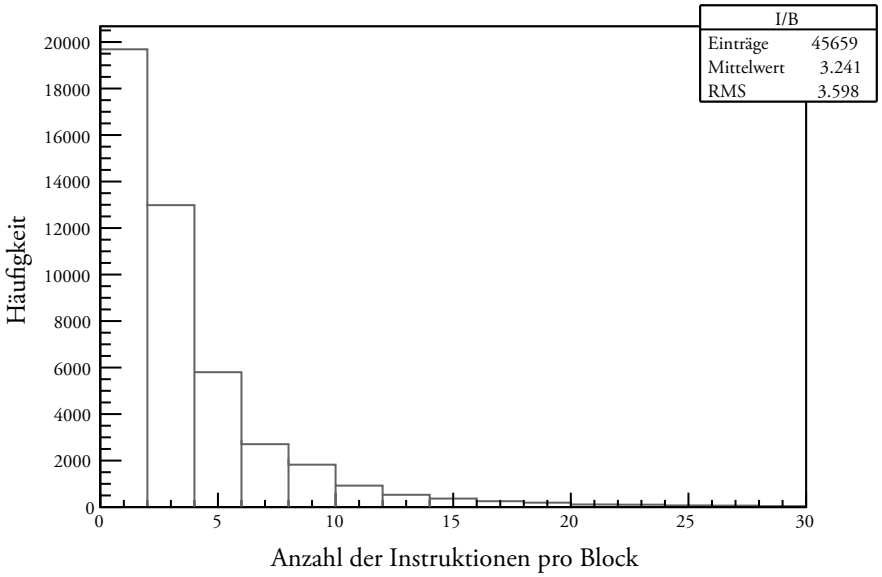


b)

Abbildung 6.3: Histogramme über den Umfang von Programmeinheiten aus der Untersuchung von 14 Open-Source-Projekten



a)



b)

Abbildung 6.4: Histogramme über den Umfang von Programmeinheiten aus der Untersuchung von 14 Open-Source-Projekten (Fortsetzung). In 6.4a) ist die Anzahl der Instruktionen I pro Block aller untersuchten Blöcke dargestellt, in b) nur die Blöcke mit $1 \leq \text{anz}(I) \leq 30$

der Ausführung nicht extern verändert werden. Es können also keine unvorhersagbaren Änderungen von z. B. Pointern auftreten.

Ein Function Slice wird definiert als eine maximierte Menge zusammengehöriger Blöcke, ein Block-Cluster, mit der Eigenschaft für seine Ausführungsdauer statische Abhängigkeiten zu anderen Function Slices aufzuweisen. Die Existenz genau eines Eintrittspunktes und genau eines Austrittspunktes, wie auch bei Basisblöcken, soll erhalten bleiben. Ein Function Slice kann als makroskopische Operation angesehen werden. Ein Function Slice wird durch genau eine Eintrittspunkt-Austrittspunkt-Kombination identifiziert. Wird ein Function Slice während der Programmausführung mehrfach durchlaufen, so können sich mehrere mögliche Datenquellen ergeben. Jede dieser Datenquellen führt zu einer eingehenden Datenflusskante. Analog gilt dies für ausgehende Datenflusskanten, ein- und ausgehende Kontrollflusskanten. Für Unterprogrammaufrufe ist der Kontext der `call`-Anweisung relevant. Ein Aufruf an einer anderen Stelle des Programms bildet ein neues Function Slice.

$$FS = \{B_x | 1 \leq x \leq n \wedge x = n \rightarrow B_x \in B_{exit}^{FS} \wedge x \neq n \rightarrow B_x \notin B_{exit}^{FS}\}$$
$$B_{exit}^{FS} = \{B | \exists I \in B : (I = \text{call}) \vee (I = \text{return})\}$$

Für jede Sequenz innerhalb einer Funktion wird ein Function Slice gebildet. Diese kann

1. von dem Funktionseintritt entweder
   a) bis zum ersten Unterprogrammaufruf, oder
   b) bis zum Funktionsende (return)
2. oder von einem Unterprogrammaufruf entweder
   a) bis zum folgenden Unterprogrammaufruf, oder
   b) bis zum Funktionsende

reichen. Ein Function Slice weist weiterhin ein Label am Anfang, gefolgt von Instruktionen und einem `branch, switch, return` oder `call` am Ende auf.

Abbildung 6.5 zeigt für eine hypothetische Funktion die möglichen Function Slices. Hier werden die folgenden Sequenzen als Function Slices

$$FS_1 = \{BB_0, BB_1 : SB_0\}$$
$$FS_2 = \{BB_0, BB_2, BB_3, BB_4\}$$
$$FS_3 = \{BB_1 : SB_1, BB_4\}$$

kombiniert.

Für die Beispielanwendung aus Listing 6.1 ergibt sich z.B. die Blockfolge

$$BB_0 : SB_3 \rightarrow BB_1 : SB_0$$

als ein Function Slice. Der resultierende, gekapselte Code ist in Lisiting 6.4 aufgeführt. Der Subblock 0 des Basisblocks 1 ist auch in der Folge

$$BB_2 : SB_2 \rightarrow BB_1 : SB_0$$

enthalten. Für eine Kapselung der Function Slices werden entsprechende Instruktionen dupliziert. Die Label innerhalb eines Function Slices können, falls sie innerhalb des Function Slices nicht benötigt werden entfernt werden, ebenso unbedingte Sprünge, welche

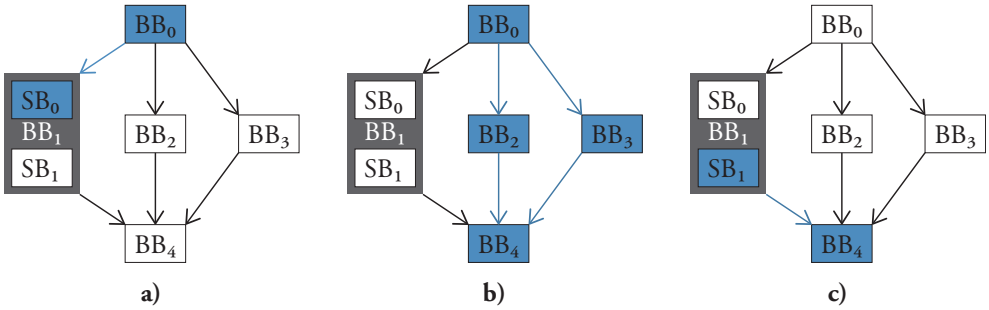


Abbildung 6.5: Erzeugung von Function Slices (FS_1 (a), FS_2 (b), FS_3 (c)) einer hypothetischen Funktion. In dem jeweiligen Function Slice enthaltene Blöcke sind blau markiert.

durch die Instruktionsfolge inklusiv exisitieren. Die Funktionen `foo` und `bar` bestehen aus mehreren Basisblöcken, bilden jedoch entsprechend der Function-Slice-Definition jeweils nur ein Function Slice. In diesen Fällen, können Label und Sprünge nicht eliminiert werden, da diese Ziele für Sprunginstruktionen innerhalb des Function Slices sind.

```
; <label>:main_BB0SB3
  %17 = trunc i32 %16 to i8
  store i8 %17, i8* %off, align 1
  %19 = load i32* %fd, align 4
  %20 = getelementptr inbounds %struct.stuff* %st, i32 0, i32 1
  %21 = getelementptr inbounds [1024 x i8]* %20, i32 0, i32 0
  %22 = call i32 (i32, i8*, i32, ...)* bitcast (i32 (...)* @read to  ↩
      i32 (i32, i8*, i32, ...)*)(i32 %19, i8* %21, i32 1024)
```

Listing 6.4: Zwischencode eines Function Slices hervorgegangen aus $BB_0 : SB_3$ und $BB_1 : SB_0$

Die Zusammenhänge zwischen Function Slices entsprechen denen gerichteter Graphen. Daher wird ein vollständiger Function-Slice-Graph als gerichter Graph mit der Knotenmenge aller existierender Function Slices V_{FS}, und zwei disjunkten Kantenmengen für Kontrollfluss E_{CF} und Datenfluss E_{DF} definiert.

$$FSG = (V_{FS}, E_{CF}, E_{DF})$$

Zur Abbildung eines Programms auf einen Function-Slice-Graph ist eine Ermittlung des Kontroll- und Datenflusses notwendig. Hierzu wird das LLILA-Framework (Low Level Intermediate Language Analyzer, vgl. [Gre07]) verwendet. Basierend auf LLVM wird eine Laufzeitanalyse durchgeführt. Eine virtuelle Maschine[2] bildet ein Gesamtsystem (Hardware und Software) in Software ab. Dadurch wird eine einfacher Zugriff auf die Laufzeitumgebung ermöglicht. Eine Kombination mit der geringeren Komplexität

[2]Der Name LLVM entstand als Abkürzung für „Low Level Virtual Machine“, hat jedoch nur wenig mit traditionellen virtuellen Maschinen zu tun, daher wird LLVM als eigenständiger Name verwendet.

(im Vergleich zum Ausgangsquelltext) der Zwischenrepräsentation bietet eine gute Basis für Analysen. Aus der Zwischensprache wird augmentierter C-Code erzeugt. Dieser liefert bei der Ausführung Profilinformationen. Aus dem Laufzeitprofil können statisch nicht ermittelbare Kontroll- und Datenflussabhängigkeiten bestimmt werden. Um den Datenfluss, insbesondere mittels per Pointer referenzierter Datenobjekte, zu bestimmen, wird eine rückwärts gerichtete Datenflussanalyse verwendet. Durch Beobachtung und die Korrelation von Schreib- und Leseoperationen ergeben sich Paare, die eine Datenquelle-Datensenke-Kombination bilden. Über die Laufzeit werden für alle Variablen die Datenflüsse kumuliert und ergeben den Datenflussgraphen. Somit lässt sich der Datenfluss in Qualität und Quantität bestimmen.

Für ein Programmsystem ergeben sich eine Menge Function Slices und deren Abhängigkeiten untereinander. Die Abhängigkeiten entsprechen dem Datenfluss und Kontrollfluss. Wie zuvor erörtert lässt sich auf dieser Basis ein kombinierter Kontroll- und Datenflussgraph erzeugen. Für die Beispielanwendung (Listing 6.1) ist der resultierende Function-Slice-Graph in Abbildung 6.6 dargestellt.

Jedes Function Slice konsumiert N Argumente und produziert M Ausgaben:

$$FS : \{i_0 \times i_1 \times \cdots \times i_{N-1}\} \rightarrow \{o_0 \times o_1 \times \cdots \times o_{M-1}\}.$$

Datenabhängigkeiten zwischen Function Slices können durch verschiedene Konstellationen des Quellcodes bzw. Zwischencodes entstehen. Es treten funktionslokale oder basisblocklokale Datenobjekte auf, auf welche in mehreren Slices zugegriffen werden kann. Ein Function Slice hängt von funktionslokalen Daten ab, wenn mindestens ein Block des Function Slices von diesen Daten abhängt. Basisblocklokale Daten können zu einem Datenfluss zwischen zwei separaten Function Slices führen, wenn der Basisblock in Subblöcke zerfällt, und beide Function Slices einen dieser Subblöcke enthalten (siehe Kapitel 8.3.4). Schreib-Lese-Kombinationen über Function-Slice-Grenzen hinweg stellen sich als Datenfluss dar. Eine weitere Möglichkeit, aus der eine Datenabhängigkeit resultieren kann, ist die Verwendung von Pointern die auf Objekte des Stacks oder Heaps zeigen. Tritt ein Schreibe- *und* Lesezugriff auf solche Speicherbereiche innerhalb eines Function Slices auf, so ergibt sich eine Datenabhängigkeit des Function Slices von einer vorherigen Ausführung des selben Function Slices.

Bei Betrachtung von Function-Slice-Graphen zeigen sich Sequenzen aufeinanderfolgender Function Slices, ohne dass innerhalb dieser Sequenz das Ziel einer rückgerichteten Kante liegt. Im Beispiel-Graph aus Abbildung 6.6 existiert die Startsequenz (main) $\rightarrow$ (open) $\rightarrow$ (main) $\rightarrow$ (atoi) $\rightarrow$ (main) $\rightarrow$ (atoi) $\rightarrow$ (main). Diese Sequenz kann ausschließlich in dieser gegebenen Reihenfolge, vollständig, und ohne Unterbrechung ausgeführt werden. Diese Sequenz umfasst alle Modifikationen von Datenobjekten, so dass keine externen Veränderungen Ziele von Pointern ändern könnten. Somit ist es möglich, auch eine solche Sequenz als eine Einheit zu betrachten. Sie wird im Folgenden als Function-Slice-Cluster (FSC) bezeichnet. Jede Sequenz, die durch rückgerichtete Kontrollflusskanten begrenzt wird, bildet ein Cluster. Auf diese Weise kann die Komplexität des Kontroll- und Datenflussgraphen weiter reduziert werden.

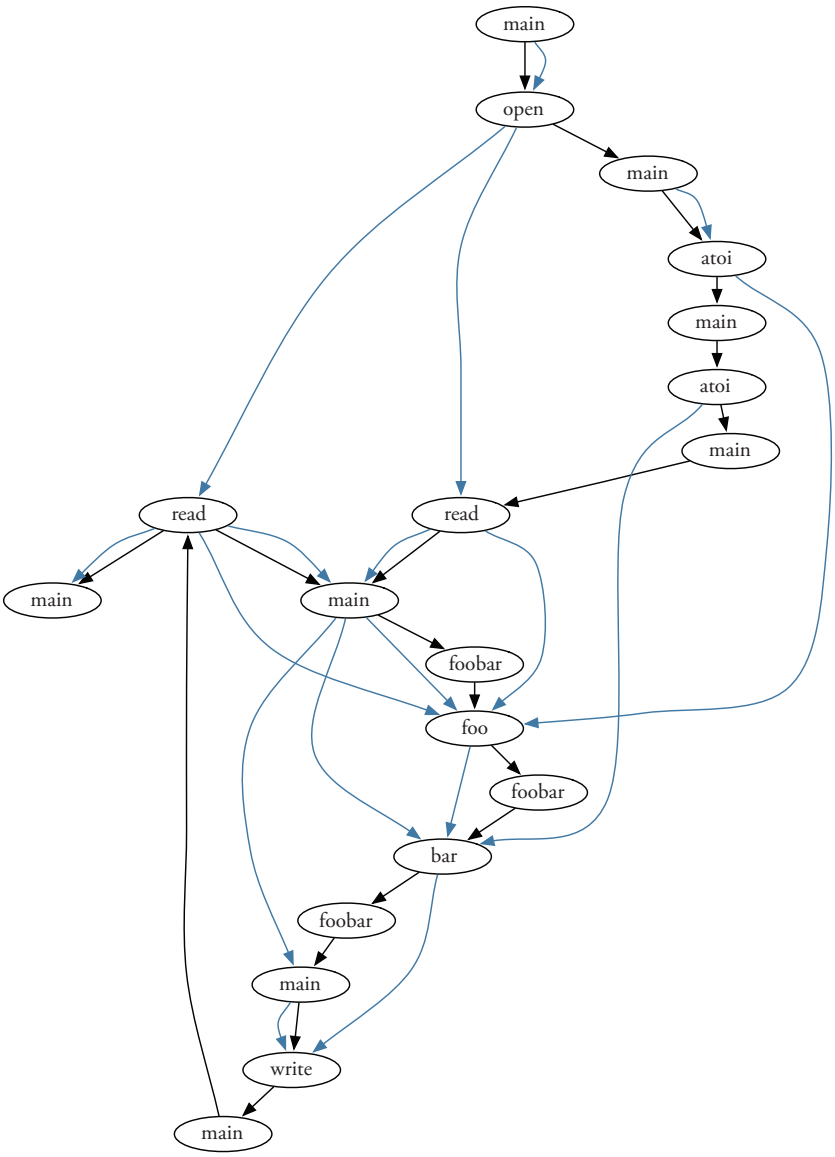


Abbildung 6.6: Beispielanwendung zur linearen Datentransformation – Function-Slice-Graph (Knoten: Function Slices, schwarze Pfeile: Kontrollfluss, blaue Pfeile: Datenfluss)

6.2.3 Selbstähnliche Programmfragmente

Sequentielle Programme zeichnen sich dadurch aus, dass sie genau einen Eintrittspunkt (`main`-Funktion) besitzen, den der Kontrollfluss beim Programmstart durchläuft. Weiterhin können sie mehrere Austrittspunkte, an denen das Programm terminiert, aufweisen. Der Kontrollfluss kehrt von all diesen Punkten zu der das Programm startenden Instanz, an genau einen Punkt, zurück. Übertragen auf die interne Struktur von Programmen ergeben sich vergleichbare Bereiche. Von einem Knoten U_1 im Kontrollflussgraph (z. B. auf Block-Ebene) kann der Kontrollfluss verzweigen, führt final durch seinen Postdominator-Knoten $U_{ipostdom(U_1)}$. Dem Verhalten sequentieller Programme folgend ergibt sich somit ein Subprogramm mit dem Eingang v_1 welches alle Knoten v_i enthält, so dass für Wege p gilt:

$$\text{ipostdom}\,(v_1) \notin \{p \mid p = (v_1, e_1, v_2, \ldots, e_{k-1}, v_k) \wedge (v_k, e_j, \text{ipostdom}\,(v_1)) \in CFG\}$$

6.3 Eignung der Fragmentierungsmodelle für die Parallelisierung

Zuvor wurden verschiedene mögliche Realisierungen für die Zerlegung eines Programms dargelegt. In Hinblick auf die Eignung für weitere Schritte hin zu einem parallelisierten Programm bestehen Unterschiede. Daher ist es für diese Arbeit elementar, Fragmentierungsmodelle auf ihre Eignung hin zu untersuchen. Insbesondere das Function-Slice-Modell, welches mit einer anderen Zielsetzung entwickelt wurde, wirkt durch seine charakteristischen Eigenschaften als aussichtsreicher Kandidat.

6.3.1 Subblock

Wie bereits erörtert, entspricht die durchschnittliche Größe von Blöcken, also der kleinsten verfügbaren Einheit (entweder Basis- oder Subblöcke), nicht der idealen Granularität. Für eine Parallelität auf Thread-Ebene wird der Overhead der Kommunikation, Synchronisation und Verwaltung den Vorteil durch die parallele Verarbeitung aufheben.

Bezüglich des Kontrollflusses bildet ein Subblock eine Einheit, die, wenn sie ausgeführt wird, immer komplett mit genau der gleichen Instruktionsfolge zur Ausführung kommt. Der Kontrollfluss kann einen Subblock während dessen nicht verlassen. Somit können keine externen Programmteile Einfluss auf Teile des Subblocks nehmen. Gleiches gilt jedoch nicht für den Datenfluss. Gefordert ist eine Programmeinheit, deren Datenquellen für die Ausführungsinstanz bekannt und determiniert sind. Dadurch soll es möglich sein, alle für die Ausführung benötigten Daten vor Betreten vorliegen zu haben. Dies kann jedoch nicht für alle Subblöcke garantiert werden. Der Pseudocode in Listing 6.5 zeigt einen Fall, bei dem das Ziel des Zeigers `a` von zuvor gelesenen Daten abhängt. Der Wert von `c` kann nicht ohne ein Teil der Instruktionen innerhalb des Subblocks bestimmt werden.

```
int *a = &x

b = *a ...
a = &a + *a
c = *a ...
```

Listing 6.5: Quelltextausschnitt, Zeiger a verweist innerhalb eines Blocks auf zwei unterschiedliche Speicherstellen, die zweite Zuweisung (zu c) ist abhängig von dem Wert, welchen der Zeiger zuvor referenziert hat

6.3.2 Function-Slice und Function-Slice-Cluster

Function Slices und auch die funktionsgrenzenübergreifenden FS-Cluster bieten durch ihre Eigenschaften eine gute Basis zur Neuanordnung. Die per Definition festgelegte Begrenzung auf genau einen Eintritts- und Austrittspunkt bezüglich des Kontrollflusses sowie die für die Ausführungsdauer konstanten Datenabhängigkeiten zwischen den Einheiten bieten eine statische Basis für die weitere Analyse. Wenn die Ausführungsreihenfolge geändert wird, kann somit sichergestellt werden, dass die Abhängigkeiten weiterhin erfüllt sind. In der Realisierung ergeben sich jedoch einige Unzulänglichkeiten dieses Modells, die einer weiteren Verwendung für die Restrukturierung mit möglicherweise nebenläufiger Ausführung widersprechen.

6.3.2.1 Verzweigung

In Function-Slice-Graphen auftretende Verzweigungen können durch Sprunganweisungen am Ende des letzten Blocks des Slices entstehen. Eine andere Ursache ist die Existenz alternativer Sequenzen innerhalb einer Funktion mit gemeinsamen Blöcken. Die Beispiel-Anwendung in Abschnitt 6.2.2 weist zwei aus der `main`-Funktion hervorgehende Function-Slices auf, welche beide auf den `read`-Funktionsaufruf innerhalb der Schleife folgen. Abbildung 6.7a) zeigt den relevanten Ausschnitt des Kontrollflussgraphen. Analog zu Abbildung 6.5 sind die resultierenden Function Slices in Abbildung 6.7b) markiert. Im Quellcode existiert eine verzweigende Sprunganweisung nur am Ende von Block $BB_1 : SB_1$. Die Darstellung als Function-Slice-Graph in Abbildung 6.7c) suggeriert eine Verzweigung am Ende des `read`-Funktionsaufrufs. Die den weiteren Verlauf bestimmende Instruktion folgt jedoch später. Auch kann aus der Darstellung nicht hervorgehen, welche Daten für die Sprungentscheidung herangezogen werden, bzw. woher diese Information tatsächlich stammt. So kann sie von dem vorherigen Function Slice vollkommen unabhängig sein. Der Inhalt bzw. die Zusammensetzung der auf eine Verzweigung folgenden Function Slices kann deutlich komplexere Formen annehmen. Hier müsste die Definition von Function Slices derart angepasst werden, dass Verzweigungen zu einer Aufspaltung der Slices führen. Dadurch wird der Granularitätsgewinn gegenüber der Subblock-Ebene jedoch stark reduziert.

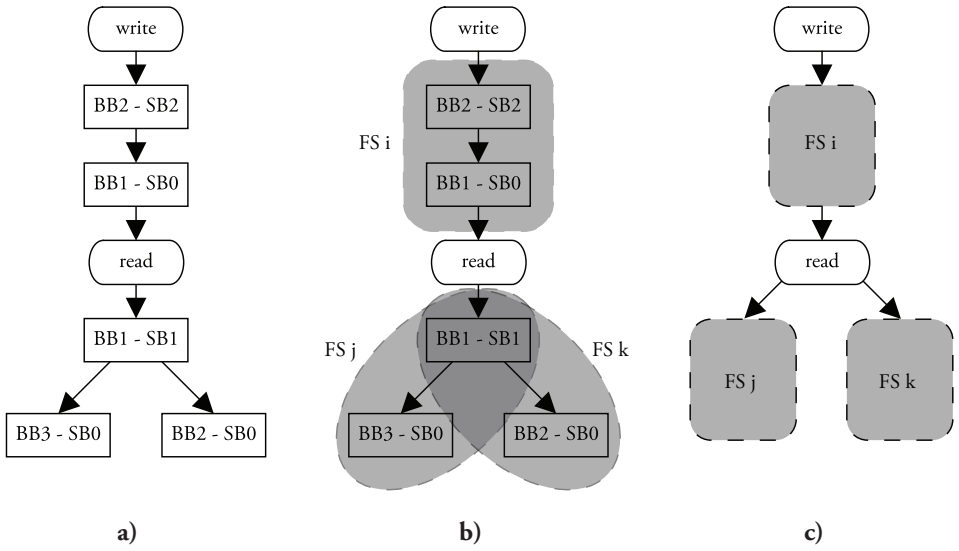


Abbildung 6.7: Ausschnitt aus Kontrollflussgraph der Beispielanwendung aus Abbildung 6.2 und resultierende Function Slices: a) Kontrollfluss auf Block-Ebene, b) Gruppierung von Blöcken zu Function Slices, c) Resultierender Function-Slice-Graph (nur Kontrollfluss)

Eine weitere Unzulänglichkeit des Modells wird bei Betrachtung der Ausführung deutlich. Der Ausführungsfaden befindet sich zunächst in Block $BB_1 : SB_1$ und somit sowohl in FS j als auch in FS k. Dieser Katzenzustand[3] bleibt solange bestehen bis die Sprunganweisung erreicht wurde. Erst dann kann feststehen, welcher Function Slice aktuell bearbeitet wird. Da die Anweisungen bis zu diesem Punkt in beiden Function Slices identisch ist, hat dies auf die Funktionalität der Anwendung keine weitere Auswirkung.

Eine Darstellung in der Form, wie sie Abbildung 6.8 zeigt, ermöglicht die Verzweigungsentscheidung unabhängig von den zuvor ausgeführten Einheiten darzustellen. Eine zusätzliche Abhängigkeitskante macht deutlich, woher die für die Verzweigung herangezogene Information stammt.

6.3.2.2 Wechselnde Quellen

Die Identifikation eines Function Slices erfolgt über die Eintritts-Austrittspunkt-Kombination und den Kontext, in dem der Slice betreten wird. Daher können über die Programmlaufzeit mehrere Datenquellen in Erscheinung treten. Der Function-Slice-Graph umfasst alle Kontroll- und Datenflusskanten die auftreten können, bzw. die im Rahmen der Laufzeitanalyse aufgetreten sind. Aus dem Graphen ist nicht abzuleiten, welche Datenflusskanten gleichzeitig existieren können, oder ob und welcher Zusammenhang zwischen einer Kontrollflusskante und einer oder mehreren Datenflusskanten besteht. Dies ist nur aus dem zeitlich aufgelösten Profil ablesbar. Für die Ausführung einer Programm-

[3] in Anlehnung an Schrödingers Katze

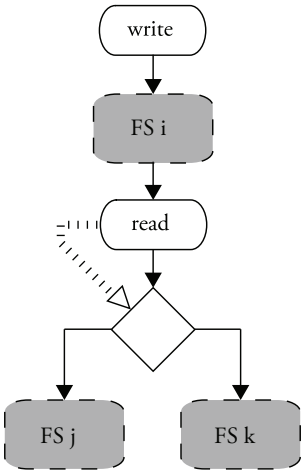


Abbildung 6.8: Verzweigung im Function-Slice-Graph mit zusätzlichem Verzweigungsknoten und dessen Datenabhängigkeit

einheit reicht es nicht aus, die Datenquellen zu kennen, sondern es muss im Rahmen des Scheduling, ausgehend von einer veränderten Ausführungsreihenfolge, auch sichergestellt werden können, dass zu verarbeitende Eingangsdaten vorhanden sind bzw. die aktuellen Informationen enthalten. Unter der Voraussetzung, dass eine Analyse ein vollständiges Profil erzeugt hat, kann aus dem aktuell ausgeführten Kontrollflussverlauf auf die Datenquellkonstellation geschlossen werden. Auf Basis des kombinierten Kontrollfluss- und Datenfluss-Graphen ist dies jedoch nicht möglich.

6.3.2.3 Function Slice Generierung durch LLILA

Die Analyse eines sequentiellen Programmsystems in seiner Zwischencodedarstellung mittels des LLILA-Frameworks führt zu Function Slices und liefert die verbindenden Abhängigkeitskanten. Die Laufzeitanalyse basiert dabei auf einem Eingangsdatensatz. Dieser bestimmt den Programmablauf und durch die ausgeführte Blockfolge auch den Inhalt der Function Slices. Innerhalb von Schleifen werden Programm-Orte mehrfach betreten und können somit in allen Konstellationen untersucht werden. Beim ersten Betreten tritt jedoch genau eine Konstellation auf. Somit ergibt sich ein Function Slice und dessen Nachfolger. Bei anderen Eingangsdaten kann die Programmausführung jedoch anderen Pfaden folgen, somit kann sich der Inhalt des Function Slices unterscheiden, und vor allem sich ein anderer Nachfolger ergeben. Die Kontrollfluss-Abhängigkeiten sind daher nicht allgemeingültig, sondern nur für einen oder im besten Fall für die meisten Fälle gültig. Abhilfe schafft hier eine Coverage-Analyse wie in Kapitel 3.11.3. Der Analyse kann dann das Profil mit Angaben zur Statistik zugrunde gelegt werden.

Das Beispiel in Abbildung 6.9 zeigt den Function-Slice-Graphen sowie für die Funktion `bubble` den Kontrollflussgraphen auf Block-Ebene. Diese Funktion weist zwei mögliche Eintrittspunkte und zwei mögliche Austrittspunkte der möglichen Function Slices auf. Jede Ein-, Ausgangskombination kann auftreten, für jeden Eingang existieren somit

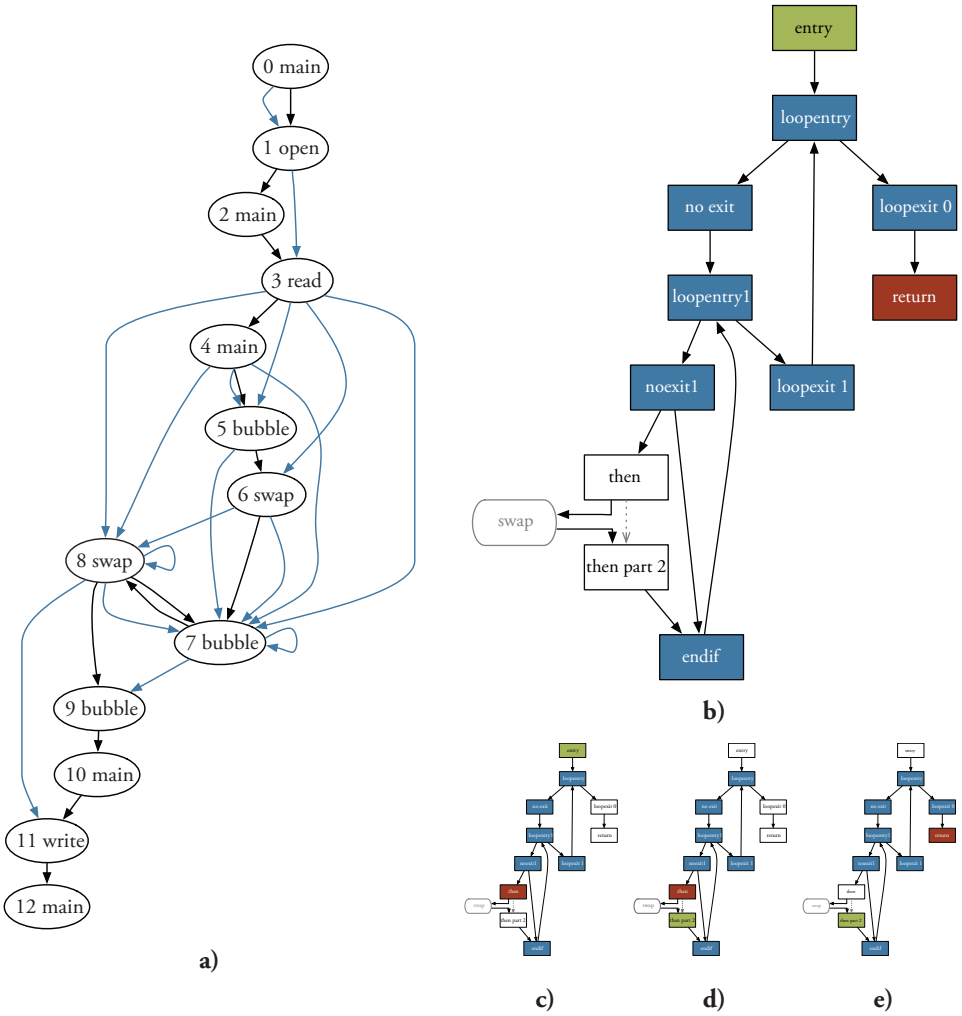


Abbildung 6.9: Beispiel Bubble-Sort: a) Function-Slice-Graph, b-e) Vier Konstellationen des Block-Graphen der bubble-Funktion; Vier mögliche Function Slices (grün: Eintrittspunkt, rot: Austrittspunkt, blau: Teil des Function Slices)

vorübergehend zwei aktive Function Slices. Im exemplarischen FS-Graphen treten drei aus dieser Funktion hervorgehende Slices auf, es können somit nicht alle Kombinationen abgedeckt worden sein.

Die Ermittlung des Datenflusses erfolgt mittels konsumentengetriebener Analyse. Für jeden Datenkonsumenten wird ermittelt, an welchem Ort im Programm die korrespondierende Speicherstelle zuletzt beschrieben wurde. Anschließend kann eine Datenflusskante vom Produzenten zum Konsumenten dem Function-Slice-Graphen hinzugefügt werden. Wenn zwischenzeitlich der Ort eines Pointers manipuliert wird, kann LLILA nur den im Analysefall auftretenden Datenfluss ermitteln, nicht jedoch eine Bedingte Datenflusskante erzeugen. Abbildung 6.10 zeigt einen entsprechenden hypothe-

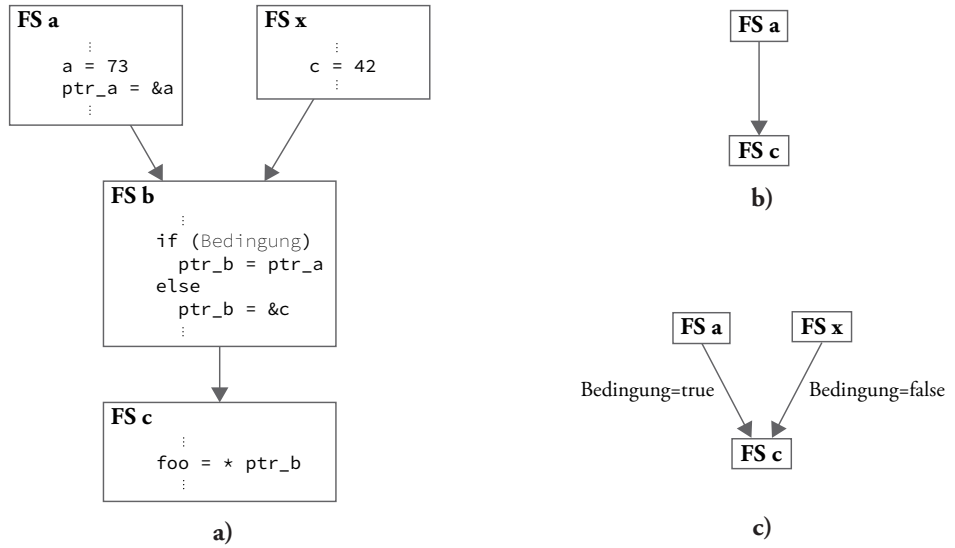


Abbildung 6.10: Unvollständige Datenflussinformationsgenerierung durch LLILA: a) Function Slices mit relevanten Instruktionen, b) Ausschnitt des erzeugten Function-Slice-Graphen, c) Erwartete Datenflusskanten

tischen Fall, den resultierenden Datenflussgraph, und eine der Realität entsprechende Darstellung.

6.3.2.4 Schleifengetragene Abhängigkeiten

Datenflusskanten innerhalb eines Schleifenkörpers können vorwärts gerichtet oder aber rückwärts gerichtet sein. Rückwärts gerichtete Kanten stellen eine Abhängigkeit zwischen einzelnen Iterationen der Schleifen dar. Bei vorwärtsgerichteten Kanten ist dies anhand des Function-Slice-Graphen nicht erkennbar. Grundsätzlich ist es auf dieser Basis nicht möglich die Abhängigkeiten zwischen Iterationen abzuleiten. Abbildung 6.11a) zeigt einen Teil eines Function-Slice-Graphen, welcher eine Schleife abbildet. Hier existiert eine Abhängigkeit von einer Iteration zur nächsten, der Slice der Funktion `foo` konsumiert Daten einer vorherigen Ausführung und produziert für eine seiner nächsten Ausführungen Daten. In Abbildung 6.11b) ist eine Abhängigkeit zwischen Schleifendurchläufen nicht erkennbar.

6.3.2.5 Abhängigkeiten ohne Kontroll- oder Datenflusskanten

Ein Datenfluss liegt vor, wenn ein Schreib-Lese-Instruktionspaar über Function-Slice-Grenzen hinweg existiert. Falls die Variablendeklaration und die zugehörige Speicherreservierung Teil eines Function Slices sind, die erste Schreiboperation jedoch in einem anderen Function Slice auftritt, so existiert zwischen diesen Beiden kein Datenfluss, und dennoch eine Abhängigkeit, die es zu berücksichtigen gilt, oder aber durch Transformieren eliminiert werden kann.

6.3.3 Superblock

Potentielle Parallelität innerhalb sequentieller Programme mittels der Umorganisation von Superblocks wurde in [YC12] beschrieben. Aussagen über Parallelität basieren in dieser Arbeit auf einer Laufzeitanalyse, aus welcher ein dynamischer Datenflussgraph abgeleitet wird. Dessen Knoten stellen Superblocks dar, Kanten repräsentieren Datenflüsse (write→read). Da typische Datenflussgraphen auf dieser Abstraktionsebene sehr umfangreich werden (vgl. [Net04]), werden Aussagen über mögliche Parallelität nur aus der Breite bzw. Höhe des Graphen abgeleitet. Das abgeleitete Scheduling bringt Superblocks zur Ausführung, wenn alle eingehenden Abhängigkeiten erfüllt sind. Die Verwendung eines reinen Datenflussgraphen ignoriert jedoch Kontrollabhängigkeiten und somit im Programm vorhandene Kontrollstrukturen. Diese sind für die Majorität sequentieller Programme notwendig, um das korrekte Verhalten sicherzustellen.

6.3.4 Selbstähnliche Programmfragmente

In Block-Graphen, welche aus Programmbeschreibungen hervorgehen, treten diese Einheiten auf. Jedoch sind sie ineinander verschachtelt und/oder überlappend. Somit besteht zwar die Möglichkeit hierarchische System zu realisieren, jedoch ergibt sich eine zusätzliche Komplexität durch die Auswahl geeigneter Kombinationen.

Die Menge der innerhalb der Kombination konsumierten Daten hängt unter Umständen stark von dem intern verfolgten Kontrollfluss ab. Es kann somit nur eine Obergrenze über alle potentiell konsumierten Daten angegeben werden.

Solange eine Kombination von Blöcken oder eine Bündelung von Instruktionen intern keine alternativen Pfade aufweist, ist die Menge der ausgeführten Instruktionen ein-

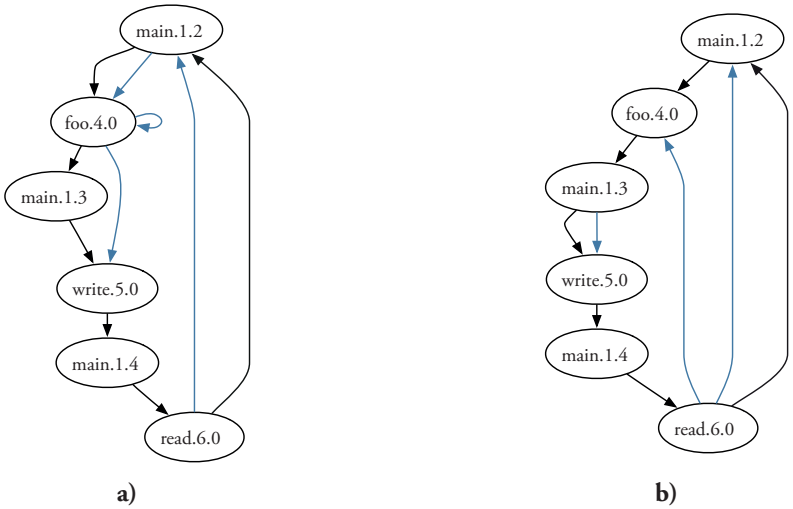


Abbildung 6.11: Ausschnitte aus Function-Slice-Graphen: a) Datenabhängigkeit zwischen Schleifendurchläufen, b) Keine erkennbare Datenabhängigkeit zwischen Schleifendurchläufen

deutig und unveränderlich. Somit ist auch die Menge der konsumierten und produzierten Daten fest.

Sobald ein Programmfragment eine Kombination alternativer Pfade enthält, wie es bei Hyperblöcken oder Treegions der Fall ist, kann die Menge der konsumierten und produzierten Daten, ebenso wie deren Quellen und Senken variieren. Dies gilt somit auch für die selbstähnlichen Programmfragmente. In diesem Fall führt der Ausführungsverlauf im Anschluss jedoch definiert zu einem Punkt. Der weitere Verlauf muss also mit eben genau den tatsächlich zuvor produzierten Daten möglich sein. Für Datenflusskanten kann auch hier nur eine Obergrenze angegeben werden, der weitere Verlauf wird jedoch maximal durch die in jedem Fall produzierten Daten beeinflusst, es können für folgende Operationen keine Informationen fehlen.

6.4 Ergebnis der Fragmentierung

In diesem Kapitel wurden die Vorraussetzungen zur Fragmentierung eines ursprünglich sequentiellen Programms definiert. Darauf aufbauend wurden mögliche Realisierungen diskutiert und Probleme sowie mögliche Problembehandlungsmethoden vorgestellt. Nachdem eine, zuvor nur als eine Einheit betrachtbare, Anwendung zerlegt wurde, soll deren Ausführung schlussendlich in einer restrukturierten Form, mit Nebenläufigkeiten, erfolgen. Dazu gilt es jedoch, zunächst Zusammenhangskonstellationen und wiederkehrende Strukturen zu erfassen. Diese können der Auffindung von Parallelität und einer automatisierten oder rechnergestützten Transformation zugrunde gelegt werden. Hinsichtlich einer Analyse ist die bereits erwähnte Visualisierung ein geeignetes Hilfsmittel. Unter den Umständen des an dieser Stelle definierten Modells wird im nächsten Kapitel ein neues Darstellungsprinzip vorgestellt. Die in diesem Kapitel aufgeführten Transformationen reichen noch nicht aus um eine tatsächliche Restrukturierung bzw. Kapselung der Modelleinheiten durchzuführen. Weitergehende Transformationen werden in Kapitel 8 diskutiert.

Die Frage nach dem Modell mit der besten Eignung für weitere Analyse- und Transformationsschritte sowie einer abschließenden Ausführung lässt sich nicht eindeutig klären. Zentrales Element bilden die Eigenschaften Einheiten zu bilden, für die alle Kontroll- und Datenflussabhängigkeiten stets bekannt sind, sowie die Unveränderlichkeit dieser Abhängigkeiten während der Ausführung einer Einheit. Durch diesen zentralen Aspekt ergibt sich das Potential für die im Folgenden behandelten Transformationen und Restrukturierungen. Erfüllt werden diese Randbedingungen durch Subblöcke oder durch geeignete Kombinationen eben dieser Subblöcke. Eine konkrete Ausprägung einer geeigneten Kombination stellt das Function-Slice-Modell dar. Dieses weist jedoch die beschriebenen Mängel auf, Subblöcke hingegen entsprechen, wie gezeigt wurde, nicht der, für eine Parallelisierung, geeigneten Granularität. Die in den folgenden Kapiteln beschriebenen Konzepte und Methoden gelten für jede Fragmentierung mit den hier geforderten charakteristischen Eigenschaften. Deren Beschreibung basiert im Wesentlichen auf dem Function-Slice-Modell. Eine Rückbesinnung auf Blöcke oder alternative rechnergestützte Gruppierungsstrategien kann sinnvoll sein.

Das in 6.3.1 beschriebene Problem, dass nicht alle Quellen vor Betreten eines Subblocks bekannt sein könnten, gilt ebenso für alle Kombinationen von Blöcken. Weiterhin gilt, dass alle innerhalb eines Blockes zu konsumierende Daten mit Betreten vorhanden sind, bzw. sich nicht mehr ändern, außer durch den Block selbst, bzw. dass alle Quellen und Senken mit Betreten bestimmt werden können. Für den hier beschriebenen Fall ist es somit notwendig, alle Zeiger manipulierenden Instruktionen innerhalb des Blockes in einen Pre-Block zu verschieben. Nach dessen Ausführung sind alle Referenzen bekannt, und die Ausführung des Haupt-Blocks kann unverändert, mit den Ergebnissen der Pre-Block-Instruktionen, durchgeführt werden.

Kapitel 7

Besonderheiten eines Modells mit den Eigenschaften kombinatorischer Logik

Im Kontext rechnergestützter Programmtransformationen wären teils Eingriffe durch den Entwickler notwendig. Dafür ist jedoch zuvor ein Programmverständnis wichtig. Dieses kann Softwareentwicklern helfen potentielle Parallelität zu finden, oder aber innerhalb halbautomatischer Verfahren präsentierte Restrukturierungen zu verifizieren oder falsifizieren. Im Rahmen dieser Arbeit ist eine Visualisierung auf Basis des zuvor dargelegten Fragmentierungsmodells von Bedeutung. Dabei soll neben der reinen Darstellung der Kontrollfluss- und Datenflussabhängigkeiten nach Möglichkeit ein geeignetes Verfahren gefunden werden, welches insbesondere ein Auffinden von Parallelität unterstützt. Die in Kapitel 4 beschriebenen Verfahren enthalten teils mehr Information als es für den hier beschrieben Anwendungszweck notwendig ist (Struktugramm, Programmablaufplan). Diese zusätzliche Komplexität ist einer zweckgebundenen Darstellung nicht zuträglich. UML wurde zur Darstellung objektorientierter Programme konzipiert, und ist somit ebenfalls nicht an das gestellte Problem angepasst. Wie bereits zuvor dargelegt ergibt sich für die Kontroll- und Datenflussabhängigkeiten ein gerichter Graph. Generelle Graphzeichenalgorithmen sind jedoch nicht für die Komplexität der darzustellenden Graphen ausgelegt bzw. nicht hinreichend nützlich um Quell- zu Ziel-Flüsse und Parallelität aufzuzeigen. Daher wird in diesem Kapitel ein neuer Ansatz präsentiert.

Das zuvor etablierte Fragmentierungsmodell weist Eigenschaften auf, welche einen Vergleich mit logischen Schaltungen nahelegt. Das neue, aus den besonderen Eigenschaften abgeleitete Darstellungskonzept basiert auf den zuvor diskutierten Programmmodellen und unterscheidet sich dadurch von etablierten Visualisierungen. Die durch den Darstellungsansatz gewonnenen Erkenntnisse können Ihre Verwendung auch in einen Transformationsansatz finden. Diese Transformationsmöglichkeiten und Parallelisierungsansätze sollen am Ende dieses Kapitels diskutiert werden.

7.1 Vergleich von Software- und Schaltungsstrukturen

Schaltsysteme unterteilen sich in Schaltnetze und Schaltwerke. Ein Schaltnetz wird auch als kombinatorische Logik bezeichnet. Es ist eine zyklusfreie, digitale Schaltung mit Ein- und Ausgängen ohne Variablenspeicher. Der Ausgang der Schaltung hängt nur vom Zustand am Eingang ab. Die Werte der Ausgänge lassen sich durch eine boolsche Funktion in Abhängigkeit der Eingänge bestimmen. Dem gegenüber steht das Schaltwerk bzw. die sequentielle Logik. Auch ein Schaltwerk beinhaltet eine logische Verknüpfung der Ausgänge mit den Eingängen, jedoch mit der zusätzlichen Fähigkeit Variablenzustände speichern zu können. Der Ausgang hängt dann nicht mehr ausschließlich vom aktuellen Eingang, sondern zusätzlich auch von der Vergangenheit ab. Der Informationsfluss ist durch die Trennung von Ein- und Ausgängen immer gerichtet.

Die Einheiten, in welche Programmsysteme dem Modell aus Kapitel 5 folgen, weisen einen festen Umfang der Informationsverarbeitung auf. Dies sind, wie im Folgenden verwendet, z. B. Function-Slices. Die zu verarbeitenden Daten sind während der Ausführung konstant. Sobald Daten an den Eingängen anliegen, können die Daten der Datenflussausgänge bestimmt werden. Durch die Analyse umfasst das Kontroll- und Datenflussmodell Daten-Eingangskanten für alle benötigten Daten. Das Verhalten eines Knotens entspricht somit dem kombinatorischer Schaltungen. Die bereits eingeführten Graphen weisen durch die gerichteten Kanten ebenso einen gerichtete Informationsfluss auf. Dementsprechend kann auch der Kontrollfluss als Leitung angesehen werden. Diese Kontrollfluss-Leitungen verbinden die den Knoten des Graphen entsprechenden Bauteile.

Es folgt, dass sich die Struktur und das Verhalten von Schaltsystemen und Programmsystemen, wenn diese auf das Modell abgebildet wurden, ähneln. Daher liegt eine vergleichbare Darstellung nahe. Im Folgenden wird ein derartiges System zur Visualisierung von Programmsystemen mit Schaltungsvisualisierungstechniken eingeführt.

7.2 Darstellung von Programmabhängigkeiten als kombinatorische Schaltung

Struktogramme und Programmablaufpläne kommen als Visualisierung im gewünschten Rahmen nicht infrage, da diese primär der Darstellung des Kontrollflusses dienen. Im Folgenden sollte jedoch der Datenfluss im Vordergrund stehen, da dieser im Weiteren unverändert sein muss, um die Funktionalität des Programms zu gewährleisten. Der Kontrollfluss wird benötigt, um Programmstrukturen aufzuzeigen. Bei Transformationen müssen jedoch Kontrollabhängigkeiten erhalten bleiben, der effektive Kontrollfluss wird mit Ausnutzung von Parallelität geändert.

Die Konzeption des UML Aktivitätsdiagramms passt zu den Anforderungen an eine Visualisierung. Jedoch geht die Mächtigkeit der Modellierung über das benötigte Maß hinaus. Von den funktionalen Eigenschaften kann kein sinnvoller Gebrauch gemacht

werden. Die grafische Notation bringt eine höhere Komplexität der Darstellung gegenüber einer Graphdarstellung mit sich. Sequenzdiagramme und Timingdiagramme enthalten eine zeitliche Komponente, sind also nicht für die statische, sondern für die dynamische Visualisierung geeignet.

Tabellarische Darstellungen ermöglichen einen hohen Informationsgehalt pro benötigter Darstellungsfläche, sind jedoch zur Erkennung von Abläufen und Zusammenhängen nicht intuitiv nutzbar. Den größten Nutzen haben Kontrollflussgraphen, Datenflussgraphen oder kombinierte Kontroll- und Datenflussgraphen. Automatisierte Graphlayouts mit grundlegenden Algorithmen sind, falls innerhalb sinnvoller Zeit generierbar, aufgrund der Komplexität (hohe Knoten- und Kanten-Zahl) für reale, nicht-triviale Programmsysteme nicht aussagekräftig.

In der Elektrotechnik und im Bereich der digitalen Logik werden Schaltpläne verwendet, um das System-Design darzustellen. Dies ermöglicht einen leicht verständlichen Zugang zu deren Aufbau. Die Generierung von Schaltsystem-Layouts setzt sich aus der Bauteil-Platzierung und Verdrahtung zusammen. Ein Bauelement kann dabei beliebig viele Ein- und Ausgänge zur Kopplung mit weiteren Komponenten aufweisen. Eine gebräuchliche Methode zur Steigerung der Übersichtlichkeit ist es, Teilnetze einzuführen und somit auf Hierarchieebenen zurückzugreifen.

Eine Adaption der Schaltsystemdarstellung auf Softwaresysteme erfordert eine Trennung von Kontroll- und Datenfluss. Bei der Darstellung von kombinierten Kontroll- und Datenflussgraphen werden zwei disjunkte Kantenmengen verwendet; Schaltungen werden ebenso unterscheidbare Verbindungen bieten.

7.2.1 Layoutsynthese

Das Problem der Layoutsynthese entspricht einer Überführung des Schaltungsentwurfs in eine geometrische Darstellung. Die Erzeugung eines Schaltungslayouts basiert auf einer Netzliste, einer textuellen Beschreibung der Verbindungen zwischen Bauelementen. Informationen über die Bauelemente sind nur in Form der verwendeten Ein- und Ausgänge ableitbar. Die Kontroll- und Datenflussgraphen enthalten diese Information und lassen sich in einer zu Netzlisten äquivalenten Form exportieren. Die Spezifikation der Logik ist für die Realisierung des Layouts nicht von Belang. Der Entwurf digitaler Schaltkreise gliedert sich in die Schritte:

1. Systemspezifikation
2. Architekturentwurf
3. Verhaltensentwurf/Logischer Entwurf
4. Schaltungsentwurf
5. Layoutsynthese
6. Layoutverifikation
7. Herstellung
8. Test

Der Schritt der Layoutsynthese, welcher Gegenstand dieses Abschnitts ist, gliedert sich wiederum in Partitionierung, Floorplanning, Platzierung, Verdrahtung und Kompaktierung.

Die Partitionierung sorgt für eine Aufteilung in Teilschaltungen, so dass die Verbindungen zwischen diesen Teilschaltungen minimiert werden. Dieser Schritt wird für Softwarelayouts nicht weiter verfolgt.

Der Begriff des Floorplannings beschreibt den Vorgang Blöcke anzuordnen. Dabei werden die Abmessung der Blöcke und auch die Verdrahtung berücksichtigt. Abmessungen sind in Hardwaresystemen durch deren Inhalt gegeben. Für Software-Fragmente und die reine Software-Visualisierung gibt es keine sinnvollen Vorgaben für die geometrische Form oder Größe. Diese wird für die gegebene Anzahl der Ein- und Ausgänge sinnvoll gewählt. Das Ziel des Floorplannings ist es, die gesamt benötigte Fläche zu minimieren.

Platzierung entspricht einer Anordnung der Elemente mit bekannten Abmessungen auf einem Gitter. Da Verbindungen einem gerichteten Graphen entsprechen und Schaltpläne einen gerichteten Informationsfluss von links nach rechts anstreben, sollten alle Elemente ohne Eingang in der ersten Spalte (links) positioniert werden. Mittels Graphtraversierung wird jedes weitere Element einer Spalte zugeordnet, die der Tiefe im Graph entspricht. Der folgende Optimierungsschritt strebt an, jedem Element eine größere Spaltennummer zuzuweisen als seinen Vorgängern (Quellen der Fluss-Abhängigkeiten). Der zweite Schritt sorgt für eine Anordnung in Zeilen, so dass die Anzahl der Überschneidungen minimiert wird [Rhi01]. Ein einfacher, schneller Algorithmus ist das Bubbling. Elemente der ersten Spalte bekommen Werte 100, 200, 300, … zugewiesen. Für jeden weiteren Knoten wird die Summe der Werte von Knoten, mit denen eine Verbindung über eine Eingangskante existiert, gebildet. Diese Summe dividiert durch die Anzahl der Kanten ergibt den zuzuweisenden Wert. Spaltenweise wird nach diesen Werten sortiert.

Die Verdrahtung sorgt für das Routing, also geeignete Verbindungswege zu ermitteln. Ein gebräuchliches Verfahren ist der Lee-Algorithmus. Hierzu wird die Fläche in Zellen unterteilt. Schrittweise wird diese Fläche vom Startpunkt aus bis zum Ziel durchlaufen [Rhi01]. Im ersten Schritt werden alle Zellen welche horizontal oder vertikal an den Startpunkt angrenzen mit dem Wert 1 befüllt. Daraufhin werden alle an die zuletzt befüllten Zellen angrenzenden Zellen mit einem um 1 erhöhten Wert befüllt; dies wird fortgesetzt, bis das Ziel erreicht wurde oder keine Zelle mehr frei ist. Im letzten Fall konnte kein Weg bestimmt werden. Mittels Backtracking wird der Weg vom Ziel immer zu der Zelle mit dem nächst kleineren Wert verfolgt. Bei mehreren angrenzenden Zellen mit gleichem Wert wird zur Vermeidung von Knicken die zuletzt eingeschlagene Richtung beibehalten. Das Backtracking ermöglicht zunächst keine Kreuzungen, da bereits mit Leitungen belegte Zellen nicht mehr mit Werten befüllt werden. Daher wird eine zweite Ebene eingeführt, markiert mit gleichen Werten. Das Backtracking kann die Ebene wechseln. Somit werden Überschneidungen von genau zwei Verbindungen möglich. Um den Platzbedarf der Gesamtschaltung bzw. der insgesamt für die Visualisierung benötigten Fläche zu minimieren, kann der optionale Schritt der Kompaktierung durchgeführt werden. (vgl. [Lie06])

Diese Schritte wurden durch eine Bibliothek zur Generierung von Schaltungslayouts aus Netzlisten abgedeckt. Die Concept Engineering GmbH bietet mit Nlview eine in vielen Bereichen der Electronic Design Automation eingesetzte Software Technologie.

Für die Darstellung müssen Symbole und deren Beschriftung festgelegt werden. Für Function-Slices bieten die Analysetools eine ID sowie den Funktionsnamen der Funk-

tion aus denen das Function-Slice hervorgegangen ist. Jedes Symbol benötigt darüber hinaus Anschlusspins mit lokal – innerhalb des Symbols – eindeutigen Namen. Zur Unterscheidung von Kontroll- und Datenfluss werden im Folgenden *C* bzw. *D* als Präfix des Pinnamens verwendet. Zur besseren Unterscheidung werden diese zusätzlich farblich gekennzeichnet: Datenfluss grün / Kontrollfluss blau. Die zweite Stelle der Bezeichnung gibt die Richtung (*I* - Eingang; *O* - Ausgang) an. Darauf folgt eine eindeutige Nummer. Entsprechend der üblichen Konvention finden sich Eingänge an der linken, Ausgänge an der rechten Kante. Die Anzahl der Ausgänge hängt nicht von der Anzahl der Datenobjekte oder dem Datenvolumen ab. Für jeden Konsumenten von Daten, die im betrachteten Function-Slice produziert wurden, wird ein Ausgang erzeugt. Die Anzahl der Ausgänge für ein Datenquellen-Function-Slice richtet sich somit nach der Anzahl der Orte im Programm, welche die Daten referenzieren.

Eine Darstellung mit ausschließlich Kontrollflusskanten wie in Abbildung 7.1a) bietet keinen Anhaltspunkt für Parallelisierung. Sie kann nur die Struktur des Programms verdeutlichen. Im Vergleich dazu zeigt Abbildung 7.1b) für das gleiche Programm eine reine Datenflussdarstellung. Es werden zu Beginn drei unabhängige Sequenzen, welche potentiell parallel ausgeführt werden können, ersichtlich. Jede Sequenz verarbeitet in diesem Fall einen anderen Teil der Aufrufparameter des Programms. Knoten ohne Datenkanten (Knoten, die den Kontrollfluss z. B. an Unterprogramme weiterreichen) werden in dieser Darstellung eliminiert.

Eine Kombinationsdarstellung wie in Abbildung 7.1c) führt zu einer stark wachsenden Komplexität. Die Anzahl der Pins und der Verbindungen nimmt zu. Daraus erhöht sich der Aufwand der Layouterzeugung und verringert sich die Klarheit der Darstellung. Daher werden im nächsten Abschnitt Methoden zur Komplexitätsreduktion betrachtet.

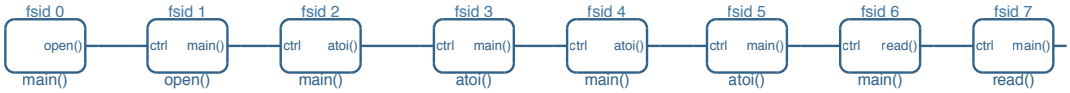
Neben dem beschriebenen Spaltenlayout kann ein Matrixlayout verwendet werden. Hierdurch ergibt sich ein kompakteres Bild. Jedoch erhöht sich der Verdrahtungsaufwand und liefert bezüglich der Lesbarkeit schlechtere Ergebnisse.

7.2.2 Komplexitätsreduktion

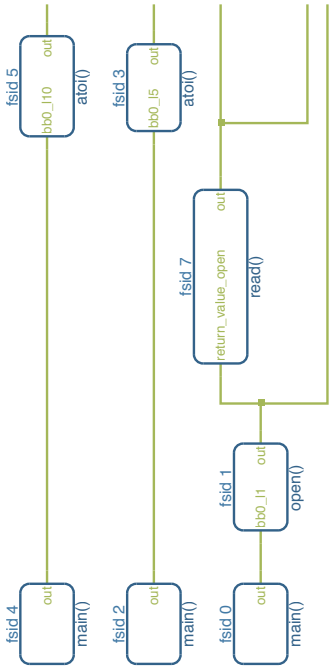
Die Anwendung dieses Visualisierungsansatzes bietet eine performante Layouterzeugung. Jedoch ergibt sich für nicht-triviale Anwendungen eine Vielzahl von Knoten und Kanten, welche keine übersichtliche und durch den Menschen gut zu erfassende Darstellung ermöglichen. Daher gilt es im Folgenden, Strategien zur Effizienzsteigerung dieses Visualisierungsansatzes zu beleuchten.

7.2.2.1 Hierarchie

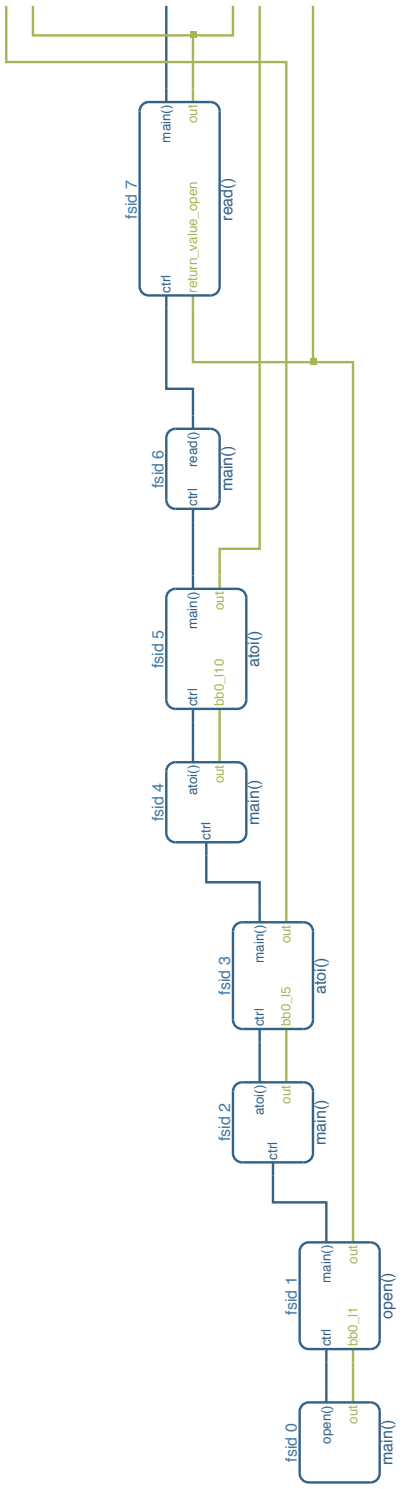
Wie auch bei anderen Darstellungsmethoden kann ein möglicher Schritt zur Komplexitätsreduktion die Einführung von Hierarchieebenen sein. Wenn Teilnetze verwendet werden, so kann deren Inhalt ausgeblendet werden. In den Datenstrukturen der Analyse findet sich ein geeignetes Konstrukt um Teilnetze zu realisieren. In Kapitel 6.2.2 wurden bereits Function-Slice-Cluster definiert. Im Folgenden werden die Bezeichnungen FS für Function-Slices und FSC für Function-Slice-Cluster verwendet. Deren Betrachtung er-



a) Kontrollfluss



b) Datenfluss



c) Kombinierter Kontroll- und Datenfluss

Abbildung 7.1: Ausschnitt der Darstellung der Beispielanwendung aus Listing 6.1

möglicht Kontroll- und Datenflüsse in und aus FSCs zu betrachten, ohne die tatsächlich beinhalteten FS zu berücksichtigen. In Schaltsystemdarstellungen werden Hierarchien gemeinhin unterstützt. Üblich sind Darstellungen von z. B. Komponenten, logischen Blöcken, Gattern etc. Die Voraussetzungen für hierarchische Darstellungen sind somit erfüllt.

Die Analyse liefert neben den Flüssen zwischen Function-Slices auch kumulierte Informationen auf Function-Slice-Cluster-Ebene. Auf diese wird bei der Darstellung jedoch nicht explizit zurück gegriffen. Mit der Einführung der FSC werden Verbindungen über Clustergrenzen hinweg in drei Abschnitte unterteilt. Der erste Abschnitt führt vom Ausgang des Quell-Function-Slices zu einem Pin am inneren Rand des FSCs. Der zweite Abschnitt führt von einem korrespondierenden Pin am äußeren Rand dieses FSCs zum äußeren Rand des FSC, welches die Senke beinhaltet. Der zugehörende innere Pin wird mit dem dritten Segment an die Senke gekoppelt. Mit Ausblenden des Inhalts eines FSC bleibt somit die Verbindung auf FSC-Ebene bestehen. Abbildung 7.2 zeigt einen entsprechenden Ausschnitt. In dieser Abbildung ist zu sehen, wie das linke Function-Slice-Cluster 0 expandiert wurde, so dass der Inhalt sichtbar wird. Hingegen ist der Inhalt des rechten FSC 1 verborgen.Um diesen unterschiedlichen Zustand optisch deutlich zu machen, weist diese Darstellung keine abgerundeten Ecken auf, im Gegensatz zu FSC 0.

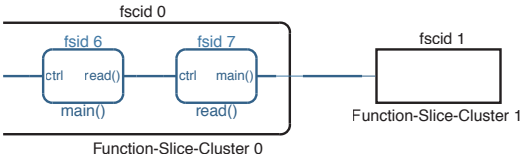


Abbildung 7.2: Hierarchie mittels Function-Slice-Cluster

7.2.2.2 Zwischenleitungsführung

Ein wesentlicher Anteil an informationstragenden Grafikobjekten wird von den Verbindungen getragen. Dementsprechend benötigen Darstellungen von nicht-trivialen Programmen viel Fläche um die Abhängigkeiten darzustellen. Jede ein- und ausgehende Kontrollflusskante belegt jeweils einen Pin, gleiches gilt für Datenflusskanten. Function-Slices, welche im Programmablauf stark eingebunden sind, und somit viele Kanten zu anderen Function-Slices aufweisen, werden dementsprechend komplex. Daher versucht der in diesem Abschnitt beschriebene Ansatz an diesem Punkt anzusetzen.

Die für den Betrachter relevanten Informationen lassen sich konkreter fassen. Steht ein FS im Fokus, so ist für dessen Ausführung wichtig, dass alle Daten zur Verarbeitung zur Verfügung stehen, und möglicherweise wo diese produziert wurden. Steht nur die Ausführung eines Function-Slices im Mittelpunkt, so ist die weitere Verwendung der produzierten Daten irrelevant, nicht jedoch ob oder welche Daten für die weitere Verwendung produziert wurden. Ähnliche Überlegungen können für den Kontrollfluss gemacht werden. Relevant ist eine Aktivierung des FS, also dass es mit gegebenen Daten ausgeführt wird, nicht jedoch welches andere Slice der direkte Vorgänger ist. Wohin-

gegen ausgangsseitig eine Kontrollflussverzweigung in Abhängigkeit der aktuellen Datenverarbeitung auftreten kann, welche Teil des betrachteten Prozesses sein kann. Der Kontrollfluss wird also vorwärts, der Datenfluss rückwärts gelesen.

Dieser Sachverhalt lässt sich in der grafischen Umsetzung berücksichtigen. Für jedes produzierte Datum wird unabhängig von den Konsumenten ein Ausgangspin erzeugt. Die vorher verwendeten Leitungen werden gebündelt. Der Datenfluss verzweigt sich dann an geeigneten Stellen. Dadurch wird nicht nur die Komplexität der Bauteile verringert, auch der benötigte Platz für die Leitungsführung wird verringert. Analog werden eingehende Kontrollflüsse gebündelt. Um dennoch die Verläufe verfolgen zu können, lassen sie sich in der Anwendung auswählen und werden daraufhin markiert. Zusätzlich werden Pins eindeutig beschriftet. Kontrollflussausgänge erhalten das Ziel der Kante. Datenflusskanten werden mit der Information, welche Datenobjekte transportiert werden, beschriftet. In Abbildung 7.3 sind die Ergebnisse dieser Maßnahme zu sehen.

In Softwaresystemen treten häufig Datenflüsse zwischen zwei Function-Slices auf, welche mehrere verschiedene Datenobjekte umfassen. Diese führen in der trivialen Darstellung zu je einer Verbindung pro Objekt. Die Bündelung zu Bussen ermöglicht auch hier eine Reduzierung des Zeichenaufwandes. Informationen über zugrundeliegende Datenobjekte oder über das transportierte Datenvolumen lassen sich dann bei Bedarf sichtbar machen.

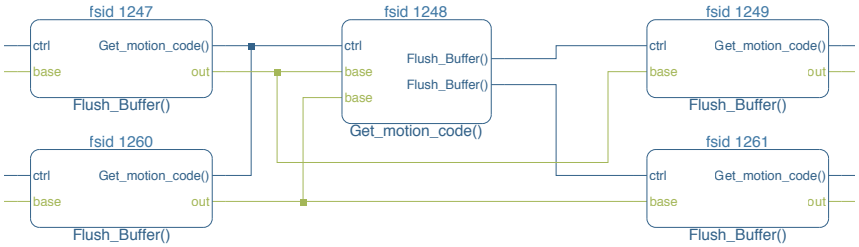


Abbildung 7.3: Bündelung von Verbindungen zwischen Function-Slices mit angepasster Pin-Beschriftung

7.2.2.3 Externe Verbindungen

Wird eine Betrachtung der Programmstruktur auf FSC-Ebene bevorzugt, so ist ein weiterer Optimierungsschritt möglich. Innerhalb eines FSC werden nur die Kanten, deren Quelle und Ziel im selben FSC liegen, gezeichnet. Für Verbindungen zu Zielen in anderen FSCs werden nur die mittleren Segmente dauerhaft gezeichnet. Die Verbindung vom Cluster-Rand zu dem Ziel-Function-Slice bzw. vom Quell-Function-Slice zum Cluster-Rand werden ausgeblendet bzw. als versteckte Verbindung realisiert. Mit Selektion eines Pins wird die Verbindung als Luftlinie angezeigt.

Dadurch wird der Inhalt der Function-Slice-Cluster überschaubarer, trotzdem ist eine gute Lesbarkeit gewahrt. Die Pins der FSC für Verbindungen untereinander wurden an den oberen bzw. unteren Rand gelegt. An diesen Seiten steht durch die horizontale

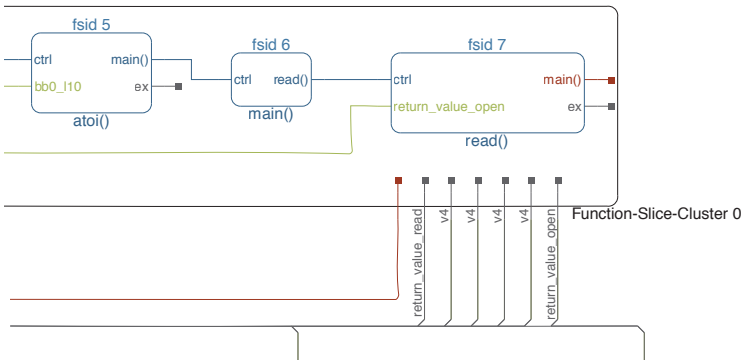


Abbildung 7.4: Externe Flüsse per Bus, ohne dauerhaft dargestellte Verbindung zu Function-Slices innerhalb der FSC

Ausdehnung des Inhalts der meiste Platz zur Verfügung. Zur Reduzierung des Inter-Cluster-Verbindungsaufwandes lässt sich auch eine Bündelung der Datenflüsse mittels Bussystem realisieren (siehe Abbildung 7.4).

7.2.3 Ergebnisse für das Beispiel MPEG-2-Dekoder

Für eine Untersuchung der Darstellung komplexer Programmsysteme wurde die Referenzimplementierung des MPEG-2-Videokodierungsverfahrens der MPEG Software Simulation Group (MSSG) herangezogen. Wie in Tabelle 7.1 zu sehen ist, führte eine steigende Komplexität nicht in gleichem Maße zu einem Anstieg der Ladezeit. Auch die absoluten Zeiten für eine Layoutdarstellung zeigen eine hohe Performance für umfangreiche Programme.

Abbildung 7.5 zeigt einen Ausschnitt der Kontrollflussdarstellung, bei der nur die externen Verbindungen zwischen den Function-Slice-Clustern sichtbar sind. Im gesamten Layout treten maximal 13 eingehende oder ausgehende Kontrollflusskanten an FSC-Grenzen auf. Trotz des höheren Verdrahtungsaufwandes bleibt der erzeugte Schaltplan

Tabelle 7.1: Vergleich der Strukturen des Programms aus Listing 6.1 und des MPEG-2-Dekoders

	Listing 6.1	MPEG-2	Anstiegsfaktor
Slice-Cluster	2	38	19
Function Slices	19	1471	≈77
Kontrolflusskanten	19	3752	≈197
Datenflusskanten	20	15809	≈790
Elemente insgesamt	60	21070	≈351
Ladezeit	≈0,1 s	≈0,5 s	5
Optimierung	≈0,005 s	≈0,025 s	5

auch bei dem komplexen Beispiel lesbar. Eine Bündelung der Kontrollflüsse als Bus ist möglich, es zeigt sich jedoch, dass der Gewinn dadurch marginal ist.

Bei einem Verzicht auf die Gruppierung von Function-Slices zu Function-Slice-Clustern orientiert sich das erzeugte Layout am Programmablauf. Es bilden sich lange Sequenzen, welche zu einer großen horizontalen Ausdehnung der Darstellung führen.

Mit Aktivierung der Function-Slice-Cluster wird eine hohe Anzahl der Kanten über FSC-Grenzen hinweg notwendig. Dies führt zu einer unübersichtlichen Darstellung wie in Abbildung 7.6 deutlich wird. Die vielfache und verteilte Referenzierung von Daten führt, wie in dem Beispiel, zu einer Vielzahl Verbindungen. Hier ist der Ausgang von `main()` verbunden mit vielen Pins am FSC-Rand, um Verbindungen zu anderen FSCs darzustellen. Die Einführung des Bussystems reduziert zwar die Fläche außerhalb der FSCs, dafür wird das Auffinden von Verbindungen und ihre Verfolgung erschwert.

Ein weiterer Effekt durch die Hierarchie und die Aufteilung der Quellen und Senken auf unterschiedliche Function-Slice-Cluster ist die veränderte Struktur im Layout innerhalb der Function-Slice-Cluster. Hier werden nur noch lokale Datenflüsse berücksichtigt, die relevante Menge der externen hingegen bleiben beim Layout unberücksichtigt. Eine potentielle Parallelität wird somit nicht mehr auf die gleiche Art und Weise offensichtlich.

Durch einen Verzicht auf die Hierarchiebildung entfällt dieses Problem. Man erhält in der Darstellung eine Netzstruktur mit vertikaler Ausdehnung (Abbildung 7.7). Bei diesem Ausschnitt der Datenflussdarstellung besteht offenkundig kaum die Möglichkeit manuell Strukturen zu erkennen. Das Ziel der Layout-Algorithmen ist eine platzsparende Darstellung. Wie durch die rot markierte Verbindung deutlich wird, kann eine Quelle in der Mitte und Senken sowohl rechts als auch links davon platziert werden. Ein Informationsfluss ist nicht immer in nur eine Richtung orientiert. Für elektrische Schaltungen ist dies sinnvoll, hier jedoch eher negativ zu bewerten. Eine mögliche Abhilfe ist es Spalten für die Anordnung vorzugeben. Diese wird anhand der Tiefe der Function-Slices im Datenflussgraphen bestimmt.

7.2.4 Anwendung für die dynamische Visualisierung

Statische Visualisierungen können nur die Programmstruktur abbilden. Um das Verhalten eines Programms darzustellen, sind Informationen über das Laufzeitverhalten notwendig. Hierdurch können z.B. zeitaufwändige Bereiche der Software besser identifiziert werden. Es gibt mehrere Ansätze die Änderungen über die Zeit darzustellen. Entweder wird die Struktur wie zuvor betrachtet um Laufzeitinformationen angereichert, oder aber es werden Verhaltensdiagramme aus den Verhaltensdaten generiert. Die Daten können akkumuliert, über eine zusätzliche Dimension in der Darstellung (räumlich) aufgetragen oder als Animation dargestellt werden. Akkumulation führt zu einer Reduzierung der Information auf einen aussagekräftigen Wert. So lässt sich die Information, wie häufig ein Programmteil ausgeführt wurde, auf einen Mittelwert reduzieren. Eine räumliche Erweiterung der Darstellung entspricht einem Plot über eine Zeitachse. Der zeitliche Verlauf kann ebenfalls als Sequenz von Einzelbildern Schritt für Schritt animiert werden. In der

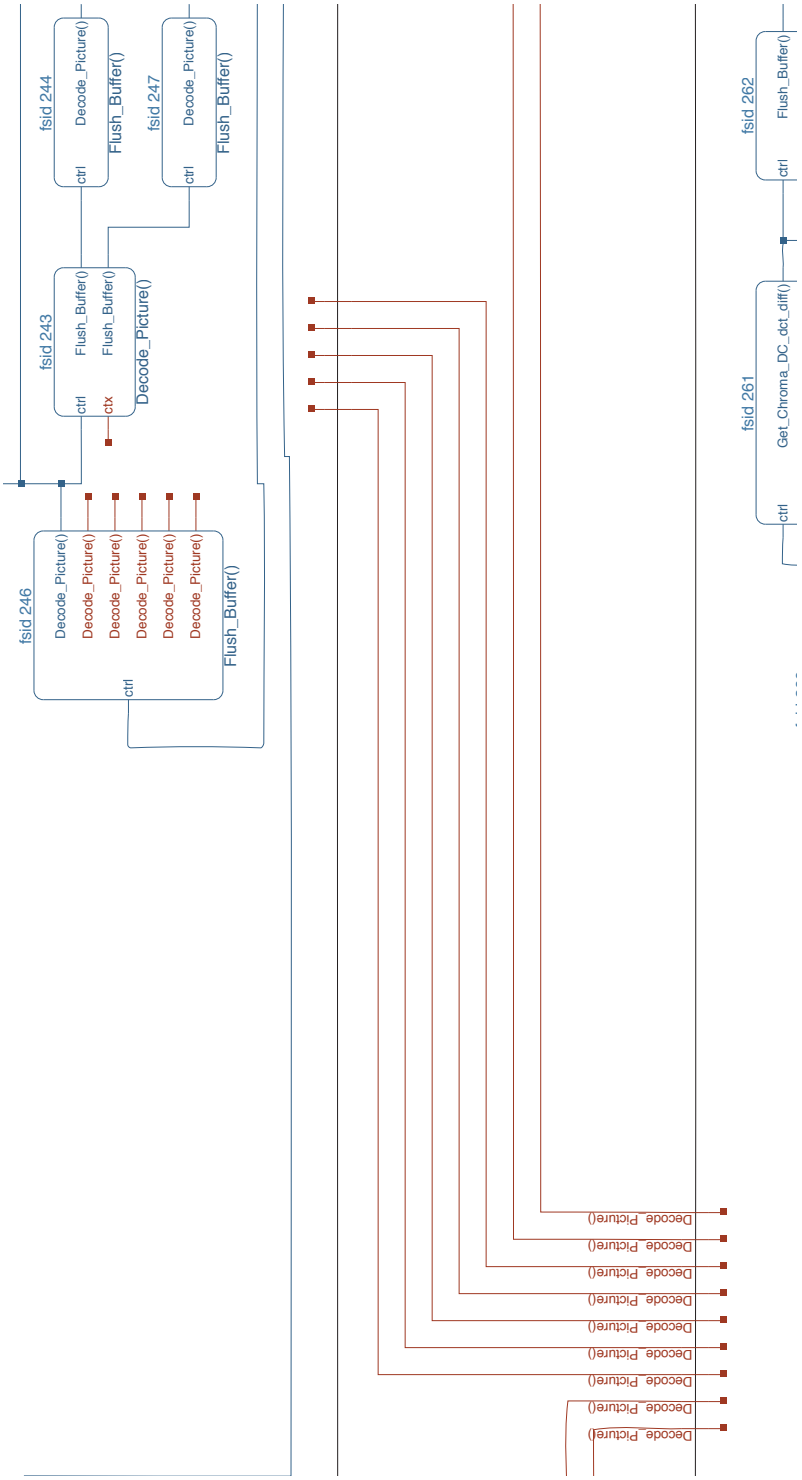


Abbildung 7.5: Kontrollfluss zwischen Function-Slice-Clustern innerhalb des MPEG-2-Dekoders (Ausschnitt)

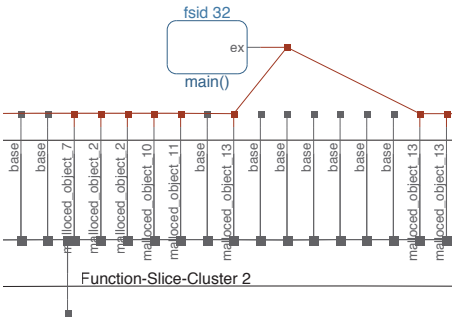


Abbildung 7.6: Datenfluss mit Ziel in anderen FSC

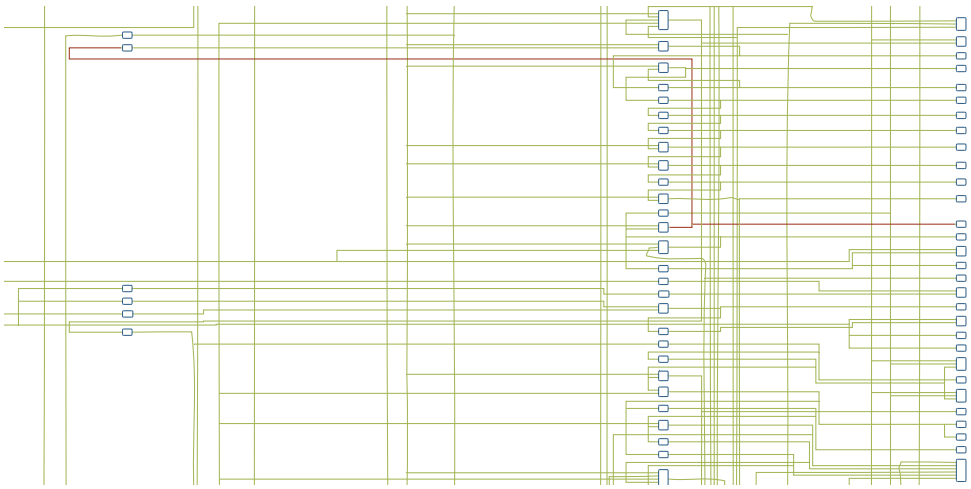


Abbildung 7.7: Ausschnitt der Datenflussdarstellung ohne Hierarchie

Regel basiert die Darstellung auf einer gegebenen Menge Eingangsdaten, deckt somit nur einen Testfall ab. (vgl. [Die07])

Um Optimierungsbedarf in Programmausführungen zu erkennen, ist die Abfolge und die Aufenthaltsdauer in Programmbereichen für Entwickler von Bedeutung. Daher wird die dynamische Visualisierung im Folgenden nur auf den Kontrollfluss bezogen. Als Basis wird die Schaltungsstruktur aus dem vorherigen Kapitel verwendet und um das Verhalten angereichert. Hierzu werden exportierte Vektorgrafiken analysiert und zusammen mit dem Laufzeitprofil über eine Anwendung in der Darstellung für jeden Teilschritt adaptiert. Für jeden darzustellenden Zeitpunkt sind die aktiven Teile des Programms hervorzuheben. Durch das Anzeigen der Einzelbilder entsteht für den Betrachter ein bewegtes Bild. Je nach Animationsgeschwindigkeit wird ein Verfolgen für das menschliche Auge schwierig, besonders bei feinen Strukturen entsteht nur ein minimaler optischer Reiz. Daher wurde neben der Hervorhebung des aktuelle aktiven Bereichs ein langsamer Übergang zu der Darstellung als nicht-aktives Element eingeführt. Dies erscheint dem Betrachter als Nachleuchteffekt.

Für große Graphen ist eine Darstellung des vollständigen Programms nicht möglich. In diesem Fall ist ein Zoom auf aktive Areale und Verschieben des gezeigten Ausschnitts zum Verfolgen des Kontrollflusses notwendig. Abbildung 7.8 zeigt einige Einzelbilder der Darstellung eines Beispielprogramms.

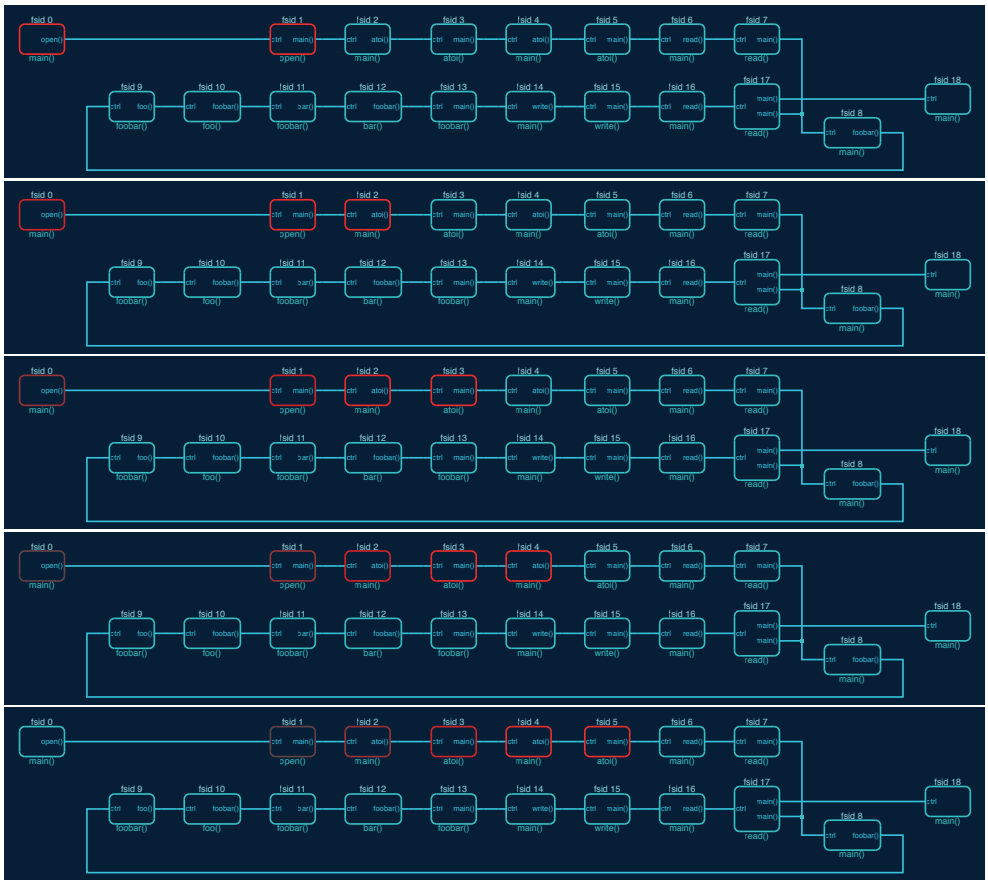


Abbildung 7.8: Teil einer Animation eines Beispielprogramms, wobei hier nur jedes 24-ste Einzelbild gezeigt wird. Dargestellt wurde die Anwendung aus Listing 6.1 mit einer Animationsgeschwindigkeit von 1 sec pro Teilschritt

Alternativ können auch Function-Slice-Graphen z. B. als 2D-Plot für die Darstellung ihres zeitlichen Programmverhaltens herangezogen werden. In Abbildung 7.9 ist die Aktivität des MPEG-2-Dekoders über die Zeit dargestellt. Für Function Slices gibt es keine natürliche Ordnung. Daher werden sie entsprechend ihrer Position Ψ im Function-Slice-Graph nummeriert. Die Ordinate stellt somit die Position des zu einem gegebenen Zeitpunkt aktiven Function-Slices im Function-Slice-Graph dar. Die Abszisse stellt Zeitschritte im Programmablauf dar. Für jedes in Ausführung befindliche Function-Slice wird ein Punkt in diesem Koordinatensystem dargestellt. Auftretende Kontroll- und Datenflüsse verbinden die beteiligten Punkte.

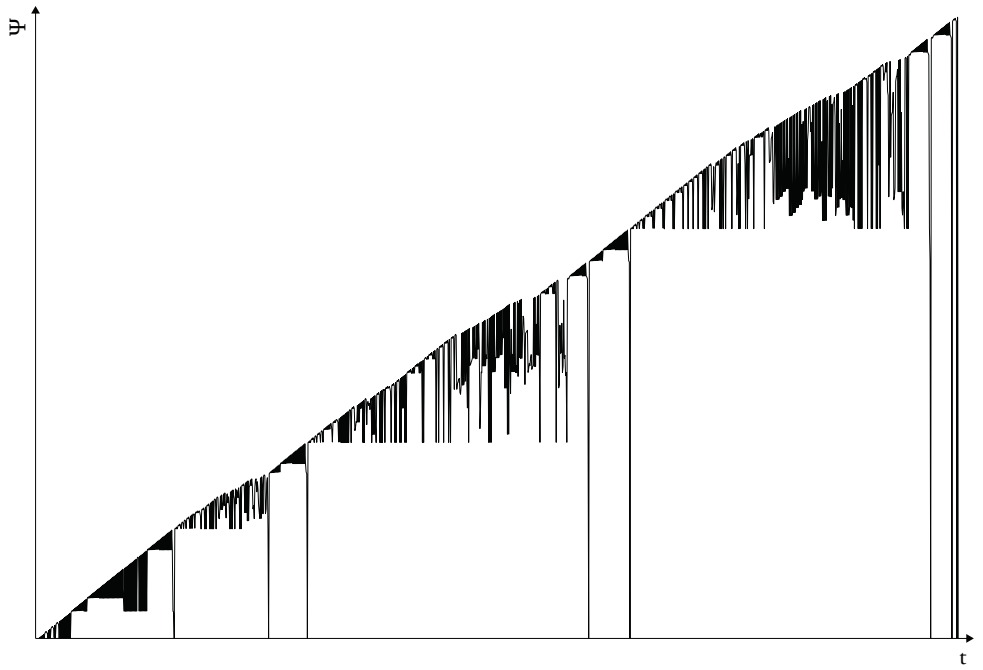


Abbildung 7.9: Kontrollflussaktivität in Form der aktiven Sliceposition im Function-Slice-Graphen des MPEG-2-Dekoders über die Zeit

7.3 Transformation in eine Schaltungsstruktur

Der Systementwurf erfolgt häufig mittels textueller Beschreibungen. In der Softwareentwicklung sind es Programmiersprachen, in der Hardwareentwicklung Hardwarebeschreibungssprachen (*Hardware Description Language (HDL)*) wie VHDL oder Verilog. Eine Beschreibung mittels einer zweckgebundenen Sprache ermöglicht komplexe Sachverhalte kompakt und nachvollziehbar darzustellen. Es gibt viele Ansätze die beiden Welten der Soft- und Hardwareentwicklung zu vereinen. So können Programmiersprachen aus der Softwareentwicklung auch zur Erzeugung von Hardware verwendet werden. Software besteht aus einer geordneten Folge von Anweisungen welche sequentiell interpretiert bzw. abgearbeitet werden. Eine Schaltung ist jedoch aus einer Menge kleinerer Module aufgebaut. Diese weisen keine Abfolge auf, sondern sind nebeneinander aktiv und verarbeiten Informationen somit parallel. HDLs beschreiben daher eine statische Struktur, Programmiersprachen hingegen einen Prozess.

Um die unterschiedlichen Anforderungen an das Ziel einer Übersetzung abzudecken, werden Programmiersprachen entweder modifiziert oder erweitert. System-C ist beispielsweise eine Klassenbibliothek, welche Funktionen und Makros beinhaltet. Dadurch können typische Eigenheiten von Hardware bei der Modellierung berücksichtigt werden. Dies betrifft Synchronisation, Parallelität und Interprozesskommunikation. Handel-C ist ein Vertreter der Spracherweiterung. Diese basiert auf C, ist jedoch eine eigene Sprache mit Konstrukten um Hardwarekomponenten zu instantiieren und Par-

allelität zu formulieren. [DM01] beschreibt einen weiteren Ansatz eine abstrakte Hochsprachenbeschreibung in rekonfigurierbare Logik zu übersetzen. Entwicklern wird dabei eine Möglichkeit geboten, Module, welche Verhalten beschreiben, und deren Interaktion zu definieren. Hierbei wird auf eine eigene, abstrakte Sprache und dazugehörige Kompilertechnik aufgebaut.

Die Hardwarebeschreibung unterscheidet verschiedene Abstraktionsebenen (siehe Abbildung 7.10). Eine makroskopische Sicht ist die Gatterebene, welche der Verhaltensbeschreibung gegenübersteht. Die Gatterebene besteht aus kombinatorischen und sequentiellen Schaltungsblöcken. Um Software auf Schaltungen abzubilden, müssen Mechanismen zur Ablaufsteuerung eingesetzt werden. Diese ermöglichen eine Umsetzung von Programmkontrollstrukturen (Verzweigung, Schleifen). Operationen wie Addition oder Multiplikation kann regelbasiert in äquivalente Schaltungsstrukturen übersetzt werden. Es ergeben sich kombinatorische Schaltungen mit Enable-Signalen, welche signalisieren, wann die Zuweisung stattfinden soll. Bedingungen können durch Steuerung des Enable-Signals umgesetzt werden. Ebenso können Schleifen oder sequentielle Abfolgen realisiert werden, indem diese Enable-Signale gesteuert werden. Variablen bzw. deren Deklaration erfordert es, getaktete Register zum Speichern der Werte anzulegen. Die auf diese Weise resultierenden Schaltungen sprengen jedoch für nicht triviale Probleme den technisch sinnvollen Rahmen. Es können daher Unterschaltungen oder Zustandsautomaten eingesetzt werden, welche an mehreren Stellen im Programm zum Einsatz kommen. Auf diese Weise ist es möglich Hardwareressourcen einzusparen. Ein Unterprogrammaufruf entspricht dann einer Unterbrechung in der Ausführung eines Zustandsautomaten und der gleichzeitigen Aktivierung der aufgerufenen Struktur. Mit dem Zurückkehren wird die Bearbeitung dann fortgesetzt. [Wir98]

Wie zuvor gezeigt, lässt sich eine Programmstruktur unter bestimmten Umständen als Schaltung darstellen. Die Voraussetzung hierfür ist, dass alle Kontroll- und Datenflusskanten bekannt und im Kontext konstant sind. Wenn die Darstellung der Programmstruktur das Verhalten abbildet, und unter der Voraussetzung, dass der Inhalt der Knoten äquivalent ist, dann gilt auch umgekehrt, dass eine identische Schaltungsstruktur das gleiche Verhalten wie die Software aufweist. Nach dem Äquivalenztheorem von [HPR88] weisen zwei Programme identisches Verhalten auf, falls deren Programmabhängigkeitsgraphen isomorph sind. Jeder Knoten produziert an den Ausgängen Daten, die von den Eingängen abhängen. Wenn ein Datum eines vorherigen Durchgangs erneut referenziert wird, existiert eine entsprechende Datenflusskante. Da nicht auszuschließen ist, dass Kontrollstrukturen oder Schleifen innerhalb der Knoten existieren, lässt sich deren Inhalt nur bedingt auf eine kombinatorische Schaltung abbilden. Eine Betrachtung von außen zeigt jedoch ein kombinatorisches Verhalten. Die Umsetzung von Zwischencode innerhalb von Basisblöcken bzw. der Knoten der Schaltungsdarstellung soll nicht Gegenstand der weiteren Betrachtung sein. Vielmehr geht es um die Realisierung der makroskopischen Struktur.

Auf dieser Basis wurde eine Methode zur Realisierung der zuvor beschriebenen Modelle in einer Schaltungsstruktur entwickelt. Dabei wird sowohl die Realisierung des Kontroll- als auch des Datenflusses behandelt. In den gezeigten Beispielen werden Blöcke und nicht wie zuvor Function-Slices als Fragmentierungsmodell verwendet. Die ge-

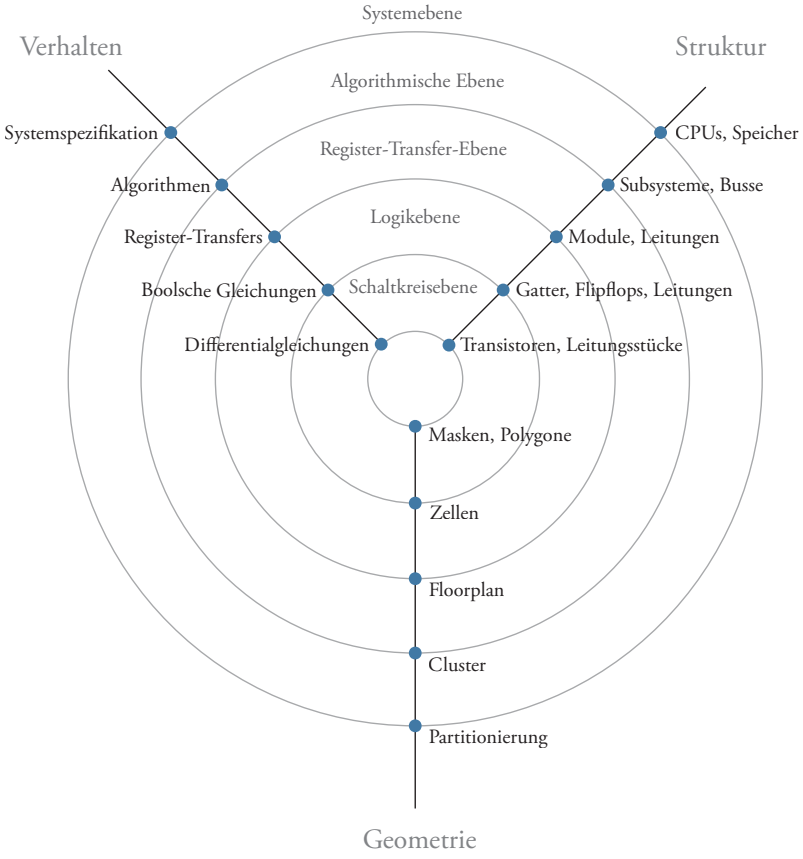


Abbildung 7.10: Y-Diagramm oder Gajski-Diagramm zeigt die Sichtweisen im Hardwareentwurf [GK83]

zeigten Maßnahmen sind jedoch für diese Modelle kompatibel. Die in diesem Kapitel gezeigten Schaltplanausschnitte entstammen der Register-Transfer-Ebenen-Darstellung der ISE Design Suite der Firma Xilinx Inc.

Zusätzlich zu der dargestellten Struktur erfordert eine Realisierung als getaktete Schaltung einen Takteingang für jedes synchron arbeitende Element. Darüber hinaus sind optional Kontrollfluss- und Datenfluss-Ein- und Ausgänge vorhanden. Intern wird ein Zustandsautomat (*Finite State Maschine (FSM)*) mit den Zuständen

$$\{\texttt{idle}, \texttt{work}, \texttt{finalize}\}$$

realisiert. Eine steigende Flanke am Kontrollflusseingang führt zu einer Übernahme der anliegenden Eingangsdaten und den Übergang von `idle` nach `work`. In diesem Zustand werden die Daten prozessiert. Darauf folgt ein Übergang in den `finalize`-Zustand, in dem sichergestellt ist, dass die Ausgangsdaten bereit stehen und der Kontrollflussausgang wird auf einen High-Pegel gesetzt, um das nächst folgende Element zu aktivieren. Mit dem nächsten Takt kann zurück in den `idle`-Zustand gewechselt, und dabei der Kontrollflussausgang zurückgesetzt werden.

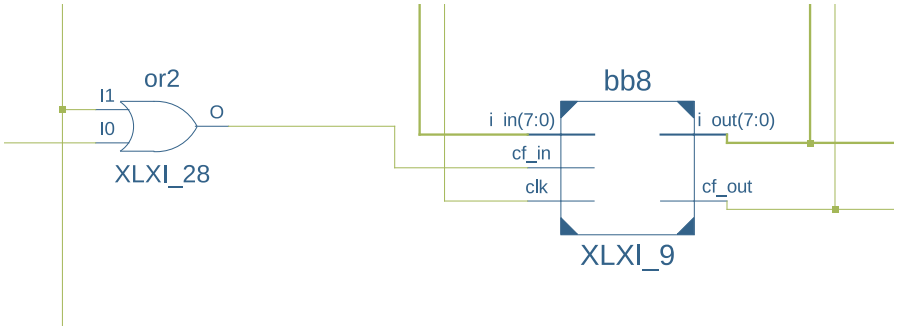


Abbildung 7.11: Aktivierung einer Programmeinheit von zwei alternativen Kontrollflussquellen

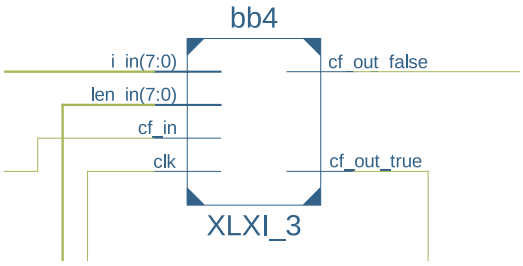
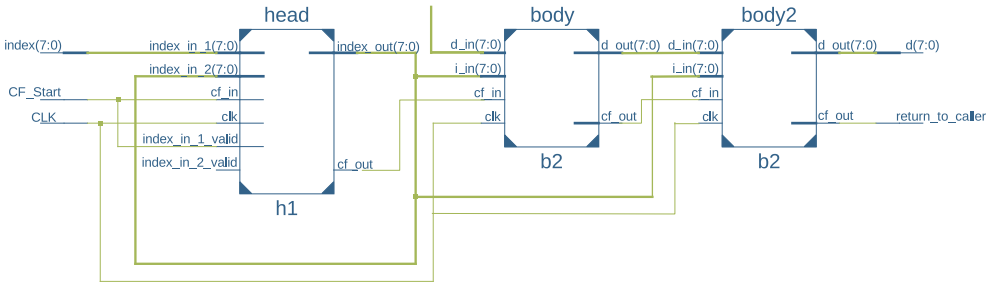


Abbildung 7.12: Exklusive Kontrollflussverzweigung in einer Schaltung

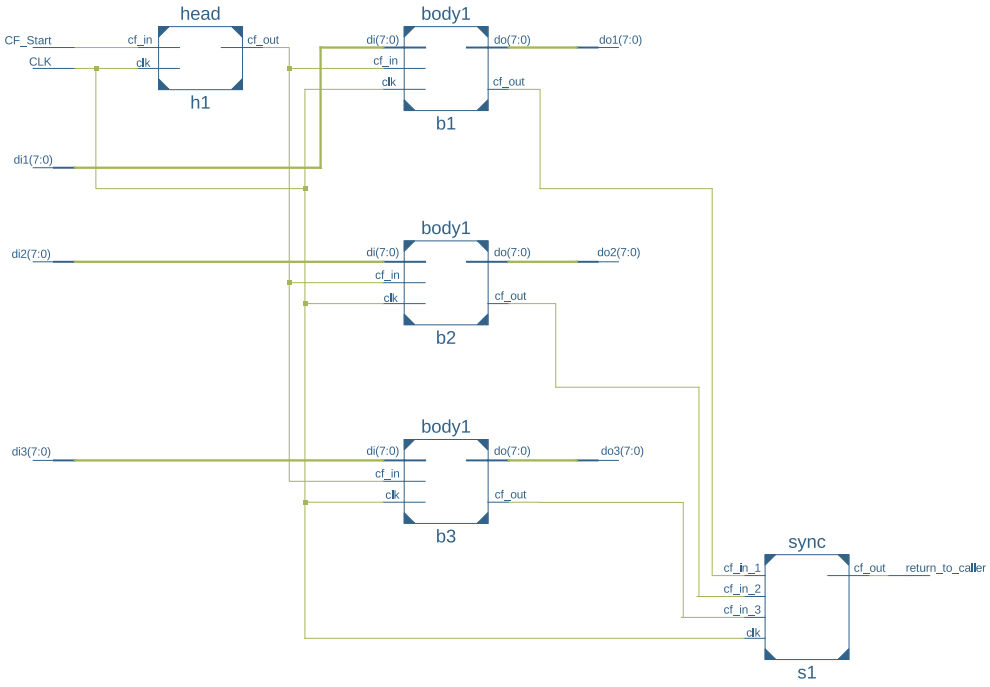
7.3.1 Kontrollfluss

Die hier als Beispiel zugrundeliegende Realisierung basiert auf einer Darstellung der Bubble-Funktion auf Block-Ebene (siehe Abbildung 6.9). Für den Kontrollfluss sind Blöcke mit eindeutigem Sprungziel an deren Ende, und solchen mit mehreren alternativen Zielen zu unterscheiden. Dementsprechend ist in Abbildung 7.11 ein Basisblock mit einem exklusiven Kontrollflussausgang gezeigt. Dieser Ausgang signalisiert die erfolgreiche Ausführung der beinhalteten Logik und die Validität der Datenausgänge. Der Block in Abbildung 7.12 verzweigt den Kontrollfluss (`branch`), daher sind zwei exklusiv aktivierbare Kontrollflussausgänge vorhanden. Falls ein Knoten von mehreren Vorgängern aktiviert werden kann ergibt sich eine Struktur wie in Abbildung 7.11.

Die Ausnutzung von Parallelität wird möglich, in dem auf der einen Seite Kontrollflussverzweigungen, die Verbindung eines Ausgangs mit mehreren Eingängen, eingefügt werden. Auf der anderen Seite werden dann unter Umständen Synchronisierungspunkte erforderlich. Beispiele sind in Abbildung 7.13 dargestellt. Vergleichbar mit der Parallelität, welche in Abschnitt 8.4.3 beschrieben wurde, wird die Schleifen in Abbildung 7.13a) durch den Schleifenkopf (Indextransformation) getriggert.



a)



b)

Abbildung 7.13: Parallität per Kontrollflussverzweigung in Schaltungsstrukturen: a) Indexiteration führt zu mehreren aktiven Knoten (Pipelining), b) Parallel Aktivierung mehrerer voneinander unabhängiger Knoten und anschließende Synchronisierung

7.3.2 Datenfluss

In Programmen wird die Parameterübergabe als Call-by-Value und Call-by-Reference unterschieden. Durch die Transformation in eine Schaltungsstruktur, wie zuvor beschrieben, ergibt sich jedoch ein anderes Systemverhalten, welches die Informationsübergabe nach diesen Prinzipien nicht direkt ermöglicht.

In der erzeugten Struktur finden sich Datenverbindungen resultierend aus dem Datenfluss der Programmanalyse. Deren Verhalten ähnelt dem des Call-by-Value, da hier die zu verarbeitenden Daten übermittelt werden können. Eine Übermittlung von Adressen (Zeigern) erfordert neben den funktionalen Bausteinen gemeinsame Speicherelemente zu realisieren. Jeder Block, welcher Zugriff auf die gemeinsam verwendeten Daten benötigt, ist dann an diesen Speicher über Adress- und Datenleitungen anzubinden. In Hinblick auf die Ausnutzung von Parallelität können mehrere Blöcke in der Schaltung gleichzeitig aktiviert werden. Somit ergibt sich, wie bereits im vorherigen Abschnitt erläutert, zusätzlicher Synchronisierungsbedarf. Weiterhin liefert die Analyse Informationen darüber, welcher Datenfluss (Verbindung zwischen Produzent und Konsument) existiert, nicht jedoch welche Programmteile für die Adressverwaltung- und übermittlung verantwortlich sind. Daher liegt es nahe, Call-by-Reference in Call-by-Value bzw. einfache Datenflüsse zu überführen.

Hierzu ist es jedoch notwendig Datenübertragungsmechanismen einzuführen. Müssen größere Datenmengen übertragen werden, kann dies aus Platzgründen nicht parallel erfolgen. Insbesondere in den Fällen, in denen die Menge der zu verarbeitenden Daten variabel ist, muss eine sequentielle Verbindung realsisiert werden. Dies führt für den Zustandsautomaten zu weiteren Zuständen (z. B. `data-receive`, `data-send`).

Im trivialen Fall erfolgt ein Datenfluss zwischen zwei in der Kontrollflussfolge direkten Nachbarn. In diesem Fall können die an den Eingängen anliegenden Daten mit Aktivierung übernommen werden (siehe Abbildung 7.14). Dieses Beispiel kombiniert zwei mögliche Kontrollflussvorgänger und die Auswahl der jeweils gültigen Daten. Häufig sind die Datenproduzenten nicht zugleich direkte Vorgänger im Kontrollflussablauf. Daher sind deren Ergebnisse zwischenzuspeichern. In Abbildung 7.15 wurden für die beiden eingehenden Datenobjekte `i` und `array` jeweils ein expliziter Puffer vorgeschaltet. Für jeden möglichen Produzenten existiert ein Eingangspaar für die Daten und ein Data-valid-Signal. Wenn der Produzent den Kontrollfluss weiterreicht, wird dieses Signal zugleich für die Speicherung der validen Daten in diesem Puffer verwendet. Dieser vorgeschaltete Block lässt sich ebenso in den `idle`-Zustand, der eigentlichen Blockrepräsentation, integrieren; ist hier jedoch zur Veranschaulichung separat gehalten.

Die Hierarchie durch Unterprogrammaufrufe geht bei der Transformation in eine Schaltungsstruktur verloren. Funktionen, welche als Call-by-Value implementiert wurden, haben Eingänge, die den Parametern entsprechen; der Ausgang entspricht dem Rückgabewert. Bei der zuvor beschriebenen Umsetzung von Call-by-Reference in eine Datenflussstruktur ist zu unterscheiden, ob vollständige Datenstrukturen der aufrufenden Einheit übergeben werden oder nur Teile. Für jedes per Call-by-Reference übergebene Datenobjekt entsteht je ein Ein- und Ausgang. Werden an die `swap`-Funktion im Bubble-Sort-Beispiel nur einzelne Elemente einer größeren Struktur übergeben, so müs-

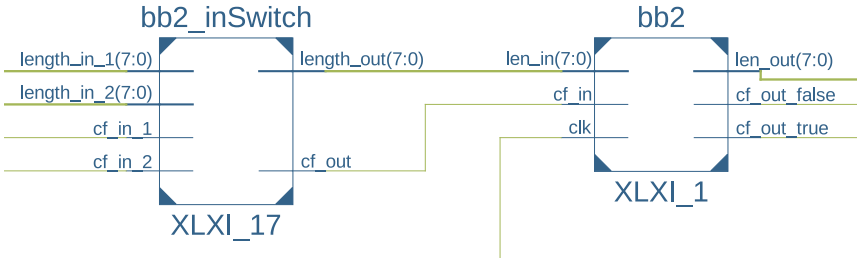


Abbildung 7.14: Ein Block kann aus zwei alternativen Kontrollflussquellen aktiviert werden, je nach Kontrollflussquelle wird die Datenquelle ausgewählt

sen die am Ausgang anliegenden Ergebniswerte im Anschluss wieder in die ursprüngliche Struktur zurückgeführt werden. Der in Abbildung 7.15 dargestellte Basisblock 7 (`bb7`) ist aus einem Subblock der Compilerzwischenrepräsentation entstanden und bestand dort ursprünglich ausschließlich aus der letzten Anweisung (Sprung) des Basisblocks. In der Schaltungsstruktur wurde zusätzlich die Zusammenführung des ursprünglichen Datenobjekts (Array) mit den Ausgabewerten der `swap`-Funktion eingefügt.

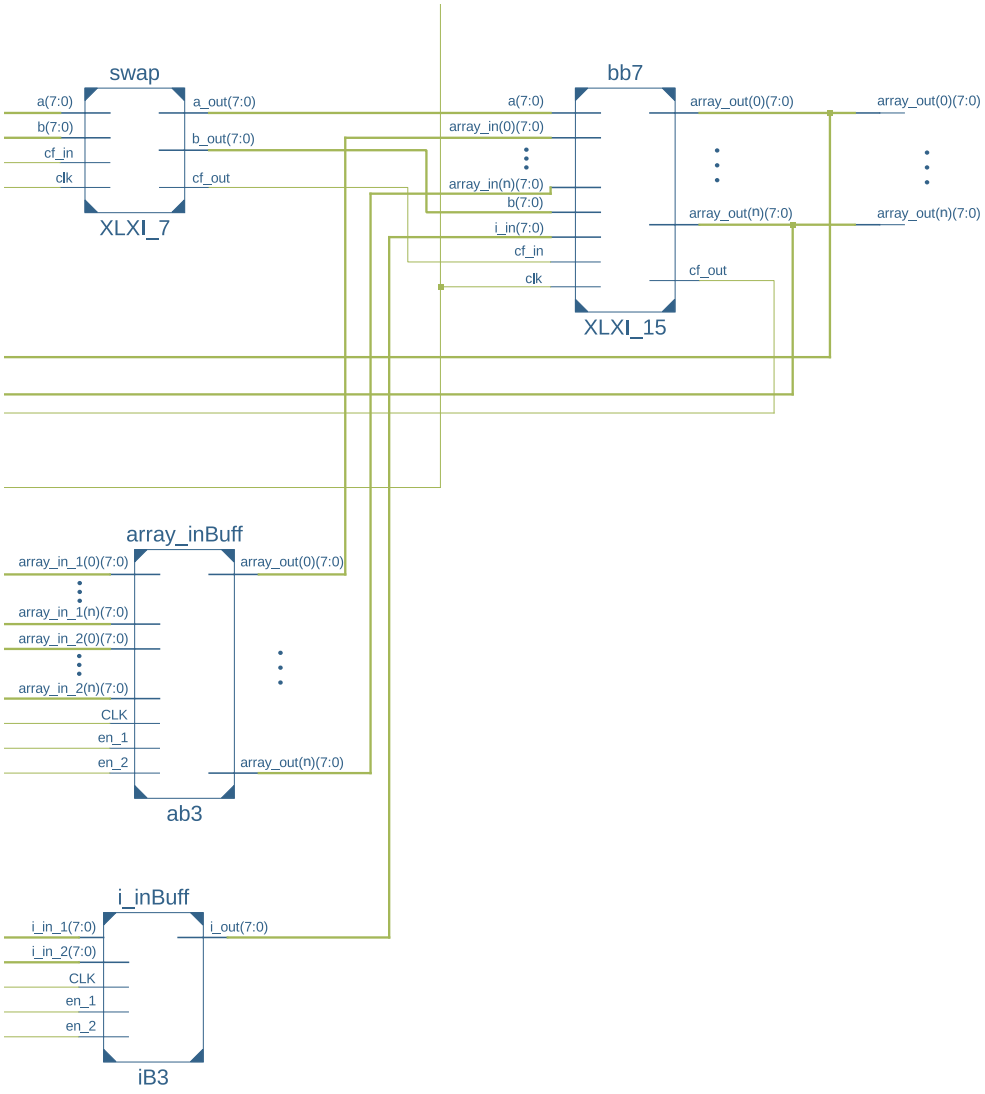


Abbildung 7.15: Ausschnitt der Bubblesort-Implementierung. Die Swap-Funktion wurde im Ausgangsquelltext per Call-by-Reference aufgerufen. Nach deren Ende kehrte der Kontrollfluss zu Basisblock 7 zurück. Diesem ist hier ein zusätzlicher Datenpuffer zur Zwischenspeicherung von Eingangsdaten (Array) aus mehreren potentiellen Quellen vorgeschaltet.

Kapitel 8

Modellbasierte Analyse und Synthese

Im vorherigen Kapitel wurde mit der Visualisierung ein Werkzeug präsentiert, welches es ermöglicht, in dem zugrundeliegenden Modell auftretende Strukturen zu erkennen und zu untersuchen. Mit diesen Erkenntnissen lassen sich Strategien zur Programmtransformation entwickeln.

Einkernprozessoren folgen im einfachsten Fall, welcher für einfache sequentielle Programme anzunehmen ist, bei der Abarbeitung der Instruktionen einem Kontrollfluss. Alle Daten werden in einem zentralen Speicher abgelegt und verwaltet. Verschiedene Prozessoren unterscheiden sich durch die Register, den Befehlssatz und Funktionseinheiten. Multithreadingtechnologien erlauben die gleichzeitige Existenz mehrerer Kontrollflussfäden. Der Compiler passt die ausführbare Programmdatei an den Prozessor entsprechend an, indem Register allokiert, geeignete Instruktionen ausgewählt und eine Ablaufplanung durchgeführt wird. Mit dem Übergang zu Mehrkernprozessoren können mehrere Kontrollflussfäden nebenläufig ausgeführt werden. Jeder Kern hat neben dem gemeinsamen Speicher (*shared memory*) auch eigenen Speicher. Verschiedene Prozessoren unterscheiden sich zusätzlich durch die Anzahl und Leistungsfähigkeit der Kerne, sowie das Kommunikations- und Synchronisierungsmodell.

Die Ausnutzung der möglichen nebenläufigen Anweisungsausführung auf getrennten Kernen mit jeweils eigenem Speicher erfordert eine Zerlegung des gesamten Programms in Threads (*Thread Level Parallelism (TLP)*). Anschließend folgt die Ausführungsplanung (Scheduling) und die Zuordnung der Threads zu den Kernen (Mapping). Neben der Möglichkeit der Taskparallelität, bei der mehrere voneinander unabhängige Anweisungsfolgen abgearbeitet werden, gibt es auch die Datenparallelität. Hierbei wird eine Operation aus mehreren unabhängigen Elementen einer Datenstruktur angewandt. Diese Form der Parallelität wird in Vektorrechnern ausgenutzt.

Die Zerlegung wurde in Kapitel 6 behandelt. Für die Ausführungsplanung müssen die Möglichkeiten einer nebenläufigen Ausführung untersucht werden. Dieses Kapitel behandelt im ersten Teil die Extraktion potentieller Parallelität. Dabei geht es um die Ausnutzung der speziellen Eigenschaften der zugrundeliegenden Zerlegung, sowie der

Adaption etablierter Transformationen auf diese Modellebene. Die größte Bedeutung fällt dabei dem Datenfluss zu. Dessen Erhalt ist für die korrekte Verarbeitung elementar. Der Kontrollfluss ermöglicht es, Schleifen und Verzweigungen zu identifizieren. Wenn keine Datenabhängigkeiten zwischen zwei Threads bestehen, lässt sich deren Ausführung voneinander unabhängig planen. Nach den Ergebnissen, die Sarkar in [Sar89] präsentiert, müssen alle Abhängigkeiten eines Program-Dependence-Graph erhalten bleiben. Eine Aufteilung in mehrere Threads erfordert somit zusätzlich Synchronisation und Kommunikation.

Die größte Arbeitslast eines Programms bündelt sich im Bereich von Schleifen innerhalb des Quellcodes. Schleifen-Strukturen bieten daher eine geeignete Grundlage für die Parallelisierung [Kuc77]. Die Schleifenstruktur ermöglicht zudem eine Gliederung. Die Möglichkeiten und die Ableitung von Parallelität aus den bisher behandelten Eigenschaften von Programmsystemen wird in Abschnitt 8.2 erörtert. Mögliche Parallelität außerhalb von Schleifen ist Teil des folgenden Abschnittes. Die verwendeten kombinierten Kontroll- und Datenflussgraphen entsprechen Function-Slice-Graphen (FSGs). Die Analysen gelten jedoch für alle Programmeinheiten mit den Eigenschaften bezüglich des Kontrollflusses, eine ununterbrochene Einheit darzustellen und im Augenblick der Ausführung vollständig bekannte und unveränderliche Daten-Quellen und -Ziele aufzuweisen.

Der Ausgangspunkt für den letzten Schritt hin zu einer parallelen Ausführung bildet eine Zerlegung in diese Einheiten. Auf Basis der Analyseergebnisse lässt sich ein geeignetes Scheduling mit Berücksichtigung der Parallelität durchführen. Die zu realisierende technische Umsetzung erfordert jedoch zusätzliche Anpassungen, welche im zweiten Teil des Kapitels diskutiert werden. So können z. B. gleichzeitige Zugriffe auf Datenobjekte auftreten. Es werden daher notwendige Transformationen in den Speicherstrukturen aufgezeigt und mögliche Ausführungs-Verwaltungs-Strukturen erarbeitet.

8.1 Eliminierung irrelevanter Kontrollflusskanten

Die Betrachtung von FSGs zeigt, dass eine signifikante Anzahl der Knoten keine Datenabhängigkeiten aufweist. Diese Knoten dienen ausschließlich dazu, den Kontrollfluss weiterzureichen. Dies erklärt auch die große Zahl der Blöcke mit sehr wenigen Instruktionen (vgl. Abbildung 6.3). Im Beispiel-Programm (siehe Listing 6.1) enthält `foobar` keine daten-manipulierenden Instruktionen sondern ruft nur weitere Unterprogramme auf und reicht Referenzen auf die zu verarbeitenden Daten weiter. Im FSG können diese Knoten eliminiert werden. Der Kontrollfluss kann direkt vom Vorgänger auf den Nachfolger des jeweiligen Knoten zeigen. Durch diese Maßnahme wird die Funktionalität nicht verändert.

Eine weitere Eigenschaft für die Optimierung ist durch die begrenzte Zahl der Datenflusskanten bedingt. In streng vorwärtsgerichteten Kontrollflussgraphen (rein sequentielle Programmabschnitte) können die Kontrollflusskanten eliminiert werden. Es verbleibt ein reiner Datenflussgraph in dem für diesen Abschnitt alle relevanten Abhängigkeiten vorhanden sind. Das Prinzip dieses Schrittes ist in Abbildung 8.1 dargestellt. Statt der

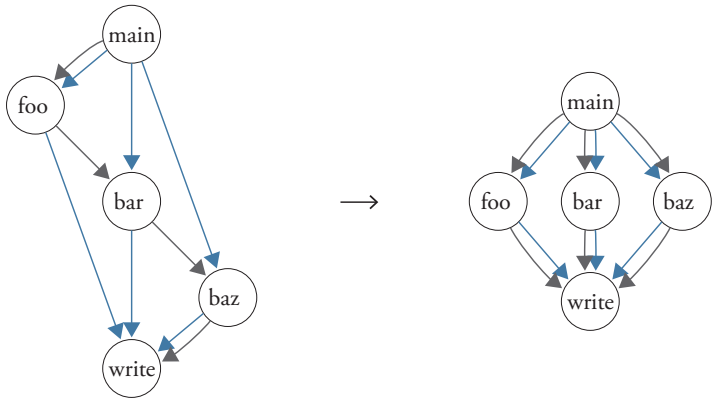


Abbildung 8.1: Parallelisierung sequentieller Abfolgen (blau: Datenfluss, grau: Kontrollfluss)

sequentiellen Folge von Unterprogrammausführungen werden `foo`, `bar` und `baz` in der transformierten Variante parallel zueinander ausgeführt, der Kontrollfluss teilt sich in diesem Fall auf und wird durch `write` wieder zusammengeführt. Abbildung 8.2a) zeigt den ursprünglichen kombinierten Kontroll- und Datenflussgraphen. Der resultierende Datenflussgraph ist in Abbildung 8.2b) dargestellt. Die maximal mögliche Parallelität wird hier direkt ersichtlich. Knoten ohne informationsverarbeitenden Inhalt stehen ohne Abhängigkeitskanten auf oberster Ebene (grüne Knoten). Sie müssen nicht ausgeführt werden. Die Tiefe t des ursprünglichen Graphen ist deutlich größer als die des resultierenden Datenflussgraphen, dafür weist dieser eine größere Breite auf. Der Grad der maximal möglichen Parallelität $S_{par} = \frac{t_{sequentiell}}{t_{parallel}}$ wird durch die reduzierte Tiefe deutlich. Diese Transformation macht deutlich, dass es mehrere Start, bzw. Eintrittspunkte geben kann. Die Ausführung muss nicht mit einem Thread starten, sondern kann von Anfang an alle Threads mit der Tiefe 0 nebenläufig starten. Dieser Sachverhalt wird in Kapitel 8.4 erneut aufgegriffen.

Im Falle nicht rein sequentieller Programmabläufe ist es nicht ausreichend, nur den Datenfluss (echte Datenabhängigkeit) zu berücksichtigen. Kapitel 8.3 wird die Behandlung von Ausgabe- und Eingabe- und Antiabhängigkeiten betrachtet. Somit ergibt sich die offensichtliche Parallelität in sequentiellen Programmen durch die Betrachtung von Datenfluss, Kontrollfluss- und Schleifenabhängigkeiten.

8.2 Schleifen

Wie in Kapitel 5.3 betrachtet kann die Ausführung von Schleifen umorganisiert werden. Techniken, die dort beschrieben wurden, kommen in der Regel zum Einsatz, um Optimierungen auf Instruktionsebene anwenden zu können. Sie lassen sich jedoch auch auf einen größeren Kontext bzw. auf das im vorigen Kapitel eingeführte Modell übertragen.

In Abbildung 8.3a) ist der exemplarische Kontrollflussgraph auf Function-Slice-Ebene mit einer Schleifenstruktur gezeigt. Das erste Function-Slice welches aus der `main`-

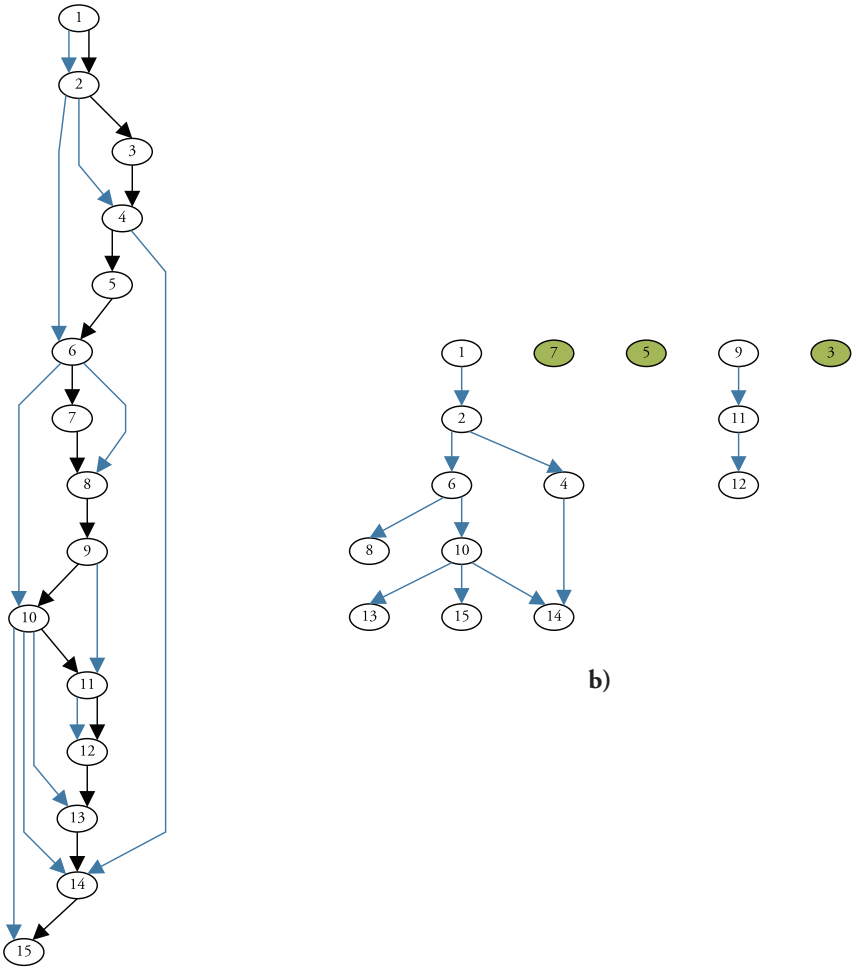


Abbildung 8.2: Parallelität durch Eliminierung des ursprünglichen Kontrollflusses: (grün: Knoten ohne Datenflusskanten, schwarz: Kontrollfluss, blau: Datenfluss) a) sequentiell, b) parallelisiert

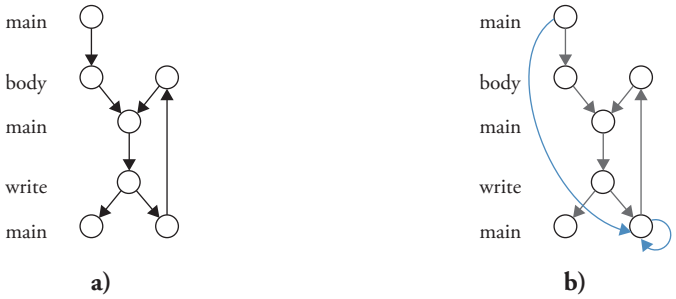


Abbildung 8.3: Schleife auf Function-Slice-Ebene: a) Kontrollfluss, b) Datenfluss der Index-Variablen (blau)

Funktion hervorgegangen ist, ruft eine Unterfunktion mit dem Schleifenkörper (body) auf. Anschließend kehrt die Prozessausführung zur `main`-Funktion zurück um eine Ausgabe (write) zu erzeugen. Der Verlauf kann dann entweder zu einem erneuten Aufruf des Schleifenkörpers oder zur Beendigung des Programms führen. Datenflusskanten, welche durch die Iteration über den Indexraum entstehen, sind in Abbildung 8.3b) aufgeführt.

Das Aufteilen einer Schleife in mehrere voneinander unabhängig ausführbare Teile setzt den Erhalt der schleifengetragenen Datenabhängigkeiten (*loop carried dependency*) voraus. Eine echte Datenabhängigkeit erfordert Kommunikation zwischen den Teilen und somit auch eine Synchronisation. Vorhandene Anti-Abhängigkeiten stellen sicher, dass Objekte zunächst gelesen und erst danach wieder beschrieben werden. Die Berücksichtigung von Ausgabeabhängigkeiten kann eine korrekte Reihenfolge definieren.

Der Schleifenindex dient der Steuerung des Ablaufs. Innerhalb des Schleifenkörpers wird der Schleifenindex zudem häufig verwendet, um auf einzelne Arrayelemente zuzugreifen. Die Datenflussanalyse zur Abbildung der Abhängigkeiten zwischen den Knoten des FSG muss sicherstellen, dass sie nicht das Array als Ganzes betrachtet, sondern nur die tatsächlich referenzierten Speicherstellen erfasst. Ein Array als eine atomare Einheit zu betrachten wäre zu unpräzise und würde zu weitreichenden Datenabhängigkeiten zwischen Schleifeniterationen führen, die eine Parallelisierung unmöglich machen. Schleifengetragene Arrayindizes hängen von dem Indexvektor, welcher alle Schleifenindexvariablen aller geschachtelten Schleifen beinhaltet, ab.

Ausgehend von der Abbildung existierender Schleifenstrukturen auf das Function-Slice-Modell, ist das Ziel mögliche Parallelität herauszubilden. Die mögliche Parallelität ist wesentlich durch die Schleifensteuerung sowie durch die Abhängigkeiten zwischen Function-Slices innerhalb des Schleifenkörpers beeinflusst. Der im Folgenden dargelegte Ansatz betrachtet mögliche Konstellationen. Daraus leiten sich Strategien zur Transformation ab.

8.2.1 Abbruchbedingungen und die Auswirkung auf die Parallelisierbarkeit

Die Abbruchbedingung einer Schleife, also die Schleifengrenzen können zu unterschiedlichen Zeitpunkten ermittelt werden. Sie ist entweder bereits zur Kompilierzeit oder mit Eintritt in die Schleife bekannt oder sie wird während der Ausführung durch den Schleifenkörper beeinflusst.

Im Function-Slice-Graph lässt sich die Unterscheidung dieser Fälle dadurch treffen, dass

- konstante Grenzen keine eingehende Datenkante
- berechnete Grenzen eine eingehende Datenkante von außerhalb der Schleife
- variable Grenzen eine Datenkante aus dem Schleifenkörper

zu dem die Schleife steuernden Knoten aufweisen. Schleifen bei denen der Index durch den Schleifenkopf verändert wird (z. B. `for`-Schleife), weisen eine Datenflusskante mit Quelle und Ziel am Schleifenkopf auf (vgl. Abbildung 8.4b)). Jede Iteration greift erneut auf die Indexvariable zu. Falls die Indexvariable innerhalb des Schleifenkörpers gelesen wird, z. B. als Index eines Arrays, resultieren Datenflusskanten wie in Abbildung 8.4a).

8.2.2 Schleifengetragene Abhängigkeiten

Für die Parallelisierung von Schleifen ohne Abhängigkeiten zwischen den Durchläufen gibt es, wie gezeigt, bekannte Ansätze. Weniger Entwürfe existieren für verkettete oder verschachtelte Schleifen mit schleifengetragenen Abhängigkeiten.

Schleifen sind in Kontrollflussgraphen durch rückgerichtete Kanten identifizierbar. Wie bereits in 6.3.2.4 erläutert ist aus dem Function-Slice-Graph nicht sicher abzuleiten, ob es eine Abhängigkeit zwischen Schleifendurchläufen gibt. Für einen trivialen Fall, bei dem die Abhängigkeit ersichtlich wird, da ein Function-Slice von seiner vorher ausgeführten Instanz abhängig ist, ergeben sich Datenflusskanten wie in Abbildung 8.4c). Hierzu ist ein Betrachtung des Verhaltens über die Zeit notwendig. Abbildung 8.5a) zeigt den Kontrollfluss für die Beispiel-Anwendung aus Kapitel 6.2.2 über die Zeit. In Abbildung 8.5b) ist ein Datenfluss über die Zeit ohne Abhängigkeiten zwischen Schleifendurchläufen abgebildet, in Abbildung 8.5c) mit einer derartigen Abhängigkeit.

Schleifen und deren Eigenschaften lassen sich in dieser Darstellung durch eine Autokorellation dieser Graphen bestimmen (vgl. Abbildung 8.6). Der Umfang des Schleifenkörpers drückt sich durch die Periode der ursprünglichen Funktion aus. Da die Autokorrelationsfunktion eines periodischen Signals ein Signal mit derselben Periode ergibt, spiegelt sich die Größe des Schleifenkörpers (Anzahl der Function-Slices) in dem Spektrum wieder.

Die Ähnlichkeit der Function-Slice-Folge mit der um τ verschobenen Folge definiert sich über die Anzahl der übereinstimmenden Function-Slices:

$$\rho(\tau) = \sum_t \begin{cases} 1 & , \text{ falls } FS_t \equiv FS_{t-\tau} \\ 0 & , \text{ sonst} \end{cases}$$

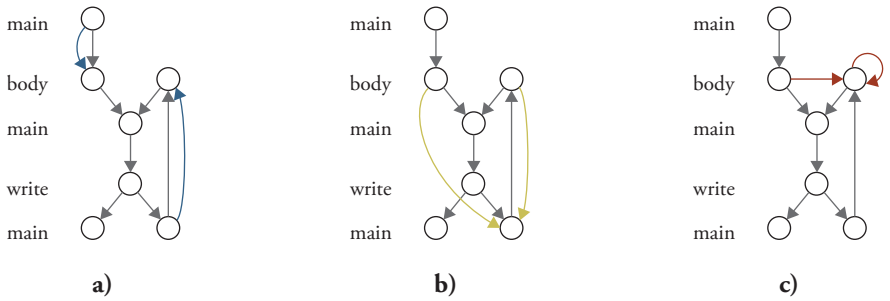


Abbildung 8.4: Datenflusskanten für Abhängigkeitskonstellationen
a) Datenflusskanten (blau) falls der Schleifenkörper abhängig von der Indexvariablen ist
b) Datenflusskanten (gelb) falls die Abbruchbedingung der Schleife vom Schleifenkörper beeinflusst wird
c) Datenflusskanten (rot) falls eine schleifengetragene Datenabhängigkeit innerhalb eines Function-Slices, welcher Teil des Schleifenkörpers ist, existiert

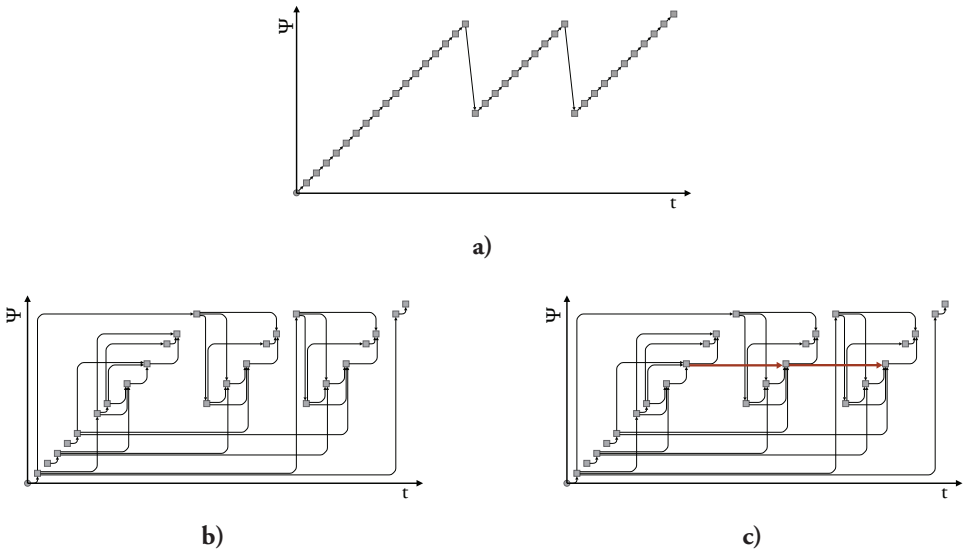
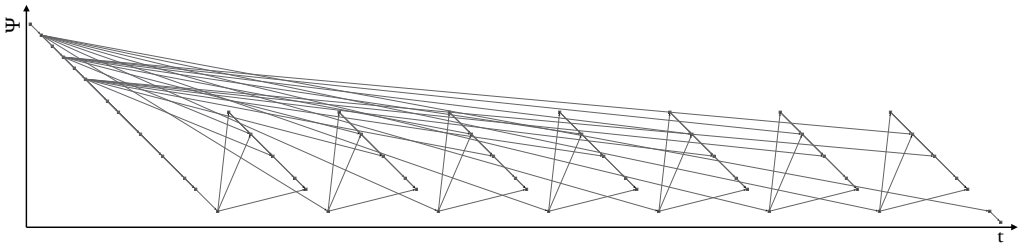
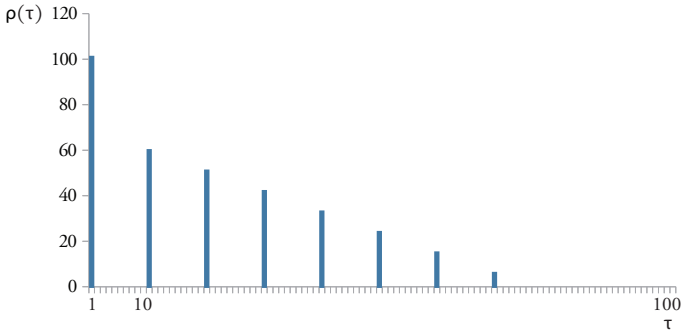


Abbildung 8.5: Programmaktivität über die Zeit
a) Kontrollfluss
b) Datenfluss ohne Abhängigkeiten zwischen Schleifeniterationen
c) Datenfluss mit einer Abhängigkeit zwischen Schleifeniterationen



a)



b)

Abbildung 8.6: Identifikation von Schleifen in Zeit-Aktivitäts-Diagrammen: a) Datenfluss über die Zeit, b) Autokorrelation des Datenflusses über die Zeit

8.2.3 Mögliche Schleifentransformationen in Function-Slice-Graphen

Welche Möglichkeiten zur Parallelisierung bestehen, hängt von den zuvor betrachteten Eigenschaften einer Schleife ab. Es bestehen die Unterschiede in den Punkten:

1. Iterationen können
   a) datenabhängig voneinander sein
   b) keine Datenabhängigkeit aufweisen
2. Die Anzahl der Iterationen kann
   a) für die gesamte Laufzeit konstant sein
   b) vor dem ersten Betreten der Schleifen berechnet werden
   c) durch Anweisungen innerhalb des Schleifenkörpers beeinflusst werden

In den Fällen 2.a) und 2.b) ist unter der Voraussetzung dass keine Datenabhängigkeit zwischen den Iterationen vorliegt ein Loop Splitting möglich. Für die Sequenz innerhalb der Schleife, welche rein sequentiell ist, kann das Verfahren aus Abschnitt 8.1 verwendet werden. Ergeben sich parallele Datenflussgraphen, kann (zusätzlich) ein Loop Distribution Ansatz für die Parallelisierung verwendet werden.

Für den Fall 2.c) kann unabhängig von der Datenabhängigkeiten zwischen Iterationen nur ein Pipelining-Ansatz gewählt werden. Pipelining bietet eine sehr universelle Möglichkeit Schleifen zu parallelisieren. Jedoch erfordert Pipelining zugleich auch einen größeren Kommunikationsaufwand.

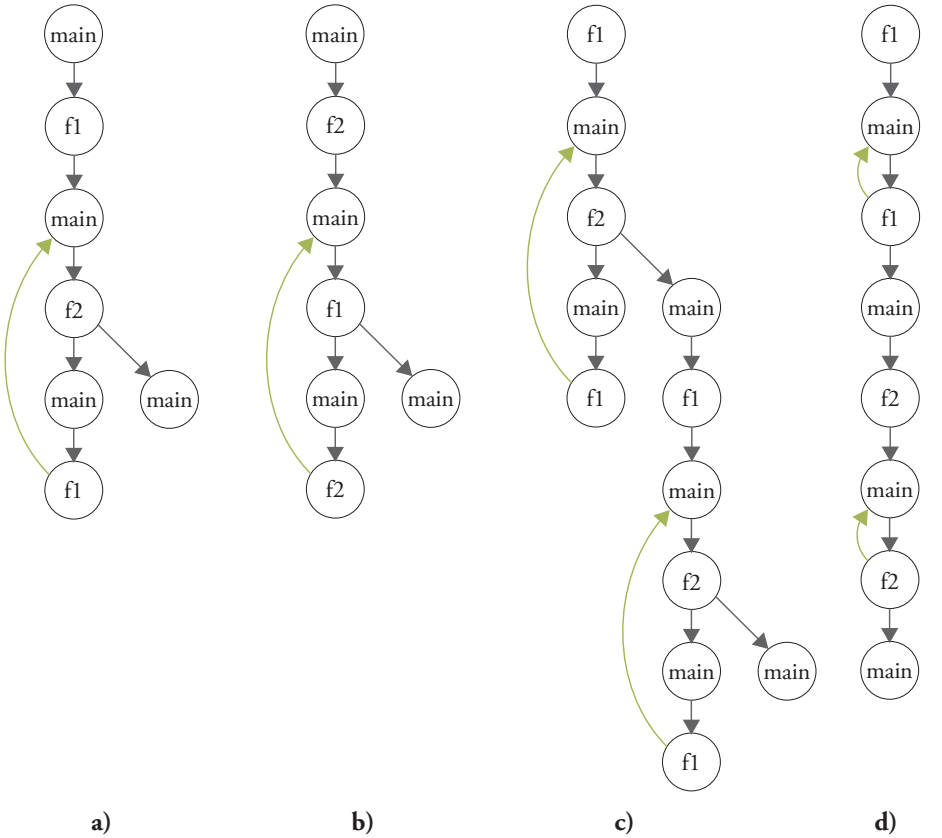


Abbildung 8.7: Exemplarische Transformationen einer Schleife (a) mit zwei Function-Slices im Schleifenkörper mittels Loop Reordering (b), Loop Splitting (c) und Loop Distribution (d) – (Pfeile stellen den Kontrollfluss dar, grün kennzeichnet die rückgerichtete Kante)

8.2.3.1 Reordering

Reordering von Function-Slices innerhalb einer Schleife, ist möglich, wenn dadurch keine Datenabhängigkeiten verletzt werden. Das erste Betreten des Schleifenkopfes erfolgt von einer anderen Stelle bzw. aus einem anderen Kontext heraus, als die nachfolgenden. Nach der Function-Slice-Definition entstehen daher zwei unabhängige Function-Slices. Dies hat zur Folge, dass es auch eine Unterscheidung des ersten Function-Slices des Schleifenkörpers beim ersten Durchlauf gegenüber seinen darauf folgenden Ausführungen gibt. Durch das Reordering wie in Abbildung 8.7b) kann dies nach der Transformation ein anderes Function-Slice betreffen. Parallelisierungsmaßnahmen welche Schleifenbereiche untersuchen, lassen dieses Function-Slice außen vor. Es kann in der Ausführung daher einen Unterschied machen, welcher Function-Slice dies ist, falls deren Workload nicht gleich verteilt ist.

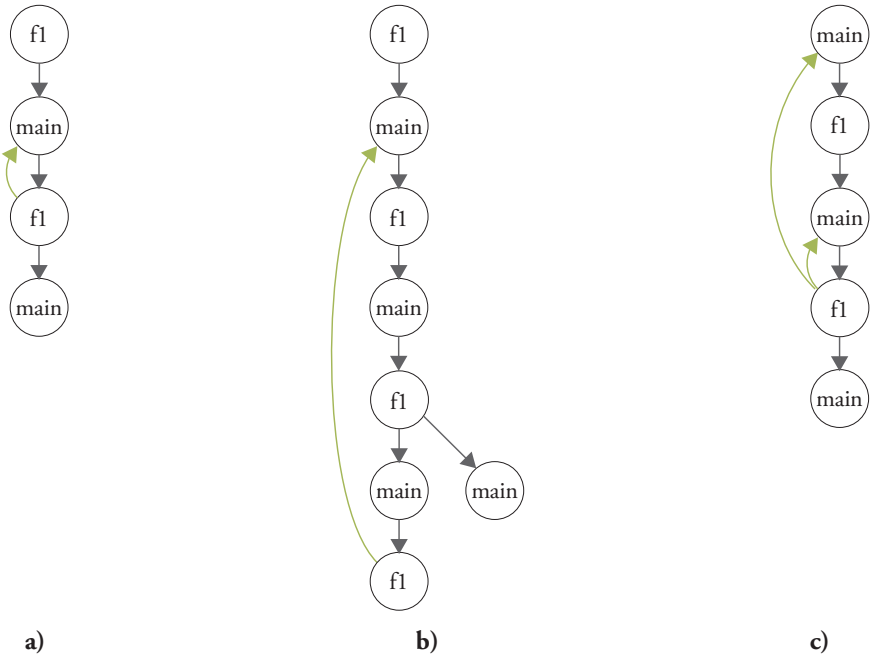


Abbildung 8.8: Exemplarische Transformationen einer Schleife (a) mit einem Function-Slice im Schleifenkörper per Loop Unrolling (b) und Stripmining (c)

8.2.3.2 Splitting

Durch das Aufteilen einer in mehrere Schleifen, lassen sich diese parallel ausführen, falls dadurch keine Datenflussabhängigkeiten verletzt werden. Schleifengetragene Abhängigkeiten verhindern ein paralleles Ausführen der einzelnen Schleifen. Anhand des Function-Slice-Graphen aus 8.7c) wird das Prinzip der Parallelisierung in Abbildung 8.9a) verdeutlicht. In diesem Fall wurde der Indexraum halbiert.

8.2.3.3 Distribution

Parallelisierung einer Function-Slice-Struktur ähnelt der des Splittings. Welche der Maßnahmen anwendbar ist, hängt von den vorhandenen Datenflussabhängigkeiten ab. Distribution ist sinnvoll, falls zwischen den einzelnen Function-Slices des Schleifenkörpers keine Datenflussabhängigkeit, aber möglicherweise eine schleifengetragene Abhängigkeit existiert. Ein exemplarischer Kontrollflussgraph auf Function-Slice-Ebene ist in Abbildung 8.9b) gezeigt.

8.2.3.4 Unrolling

Unrolling reduziert die Anzahl der Schleifendurchläufe. Dadurch ergeben sich längere Sequenzen im Schleifenkörper. Auf diese kann das Verfahren aus Kapitel 8.1 angewandt werden. Somit lässt sich der Schleifenkörper parallelisieren.

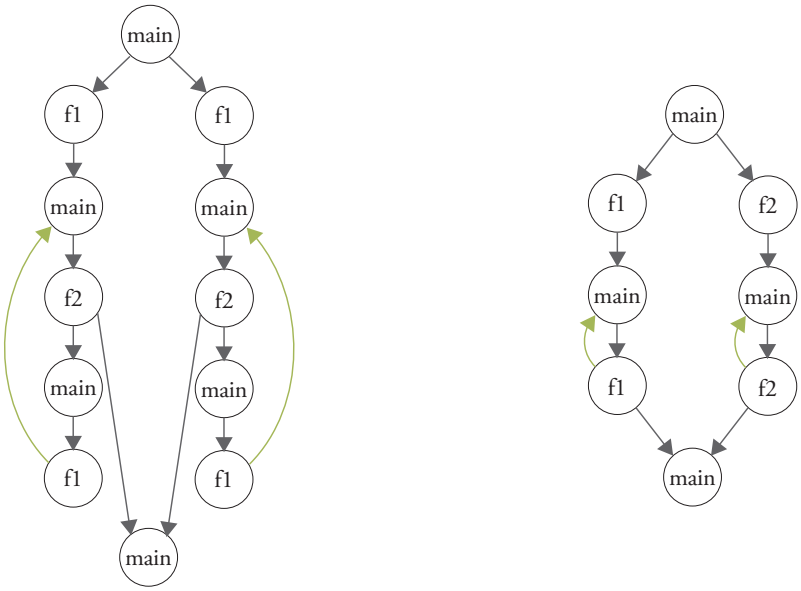


Abbildung 8.9: Parallelisierung der Schleife aus 8.7a) durch Loop Splitting (a) und Loop Distribution (b)

Ein vollständiges Unrolling (wie in Abbildung 8.10) eliminiert die vollständige Schleifenstruktur und ermöglicht es, alle Schleifendurchläufe parallel aufzurufen. Ist die Anzahl der Durchläufe einer Schleife konstant, kann diese Optimierung zur Übersetzungszeit durchgeführt werden. Wurde diese Anzahl vor dem erstmaligen Betreten ermittelt, kann die Unterteilung der Schleife in mehrere parallel ablaufende Teilschleifen erst zur Laufzeit stattfinden. Sind die Schleifengrenzen mit Betreten der Schleife unbekannt, ist ein sinnvolles Unrolling nur bedingt möglich.

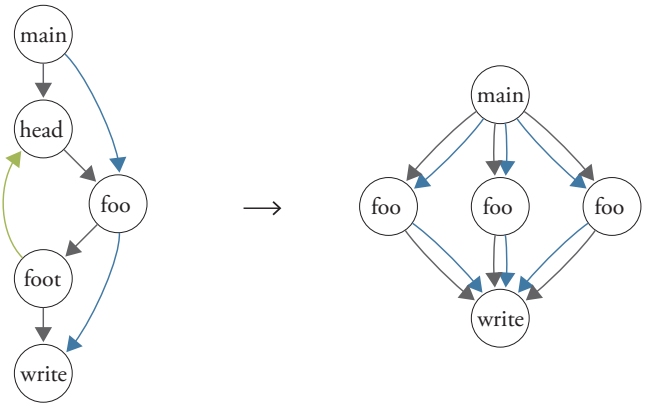


Abbildung 8.10: Parallelisierung durch vollständiges Loop-Unrolling

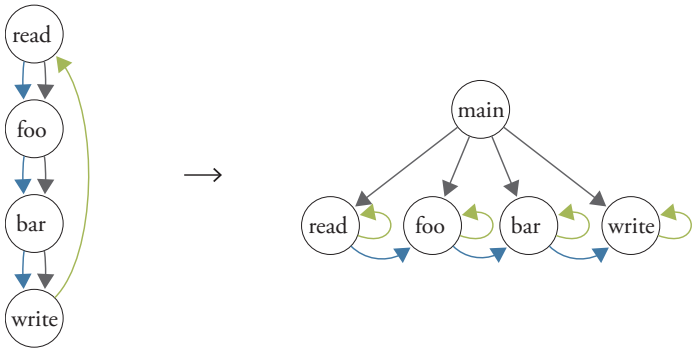


Abbildung 8.11: Parallelisierung per Pipelining (blau: Datenfluss, grau: Kontrollfluss, grün: rückgerichtete Kante (Schleife))

8.2.3.5 Stripmining

Stripmining auf Quelltextebene wie in Kapitel 5.3 beschrieben, hat keine Auswirkung auf den Function-Slice-Graph da die Konstruktion der Function-Slices die Schleifenköpfe zu einem Slice zusammenfasst. Ein Stripmining auf Function-Slice-Ebene ist in Abbildung 8.8c) gezeigt. Zur Parallelisierung können die zuvor beschriebenen Verfahren auf die jeweiligen Schleifen angewendet werden.

8.2.3.6 Pipelining

Das Prinzip des Pipelining auf Function-Slice-Ebene ist in Abbildung 8.11 dargestellt. Innerhalb des im Rahmen dieser Arbeit zugrunde gelegten Modells ist Pipelining als eine anpassbare Parallelisierungsmöglichkeit zu betrachten. Pipelining ist möglich bzw. sinnvoll, wenn die Anzahl der Durchläufe nicht konstant ist, oder wenn es Datenabhängigkeiten zwischen Schleifendurchläufen gibt. In diesen Fällen sind Optimierungen wie Unrolling nicht möglich. Um Pipelining auf Function-Slice-Ebene zu realisieren, muss der Schleifenkörper in Pipelinestufen unterteilt werden. Zur Festlegung der Stufengrenzen wird der Function-Slice-Graph zugrundegelegt, wobei der Algorithmus auf den Datenflusskanten innerhalb des Schleifenkörpers basiert. Knoten ohne Datenabhängigkeiten werden auch hier zunächst eliminiert. Datenstrukturen, die durch die Schleife verarbeitet werden, sollen in jeder Stufe prozessiert werden um das ideale Verhältnis zwischen Kommunikation und Verarbeitung zu erzielen. Die Unterscheidung von zu verarbeitenden Datenstrukturen und Parametern für die Berechnung kann auf den bei der Analyse übermittelten Datenvolumina basieren. Der verwendete Algorithmus ist in Lisiting 8.1 aufgeführt. Dem Algorithmus wird der aus der Laufzeitanalyse stammende Function-Slice-Graph übergeben. Dieser besteht aus einer Liste `FS` mit allen Function-Slices (Knoten des Graphen) sowie den Kontroll- und Datenflusskanten welche als Listen (`DF`, `CF`) mit 2-Tupeln vorliegen. Ein 2-Tupel der Form (Quelle, Senke) bildet dabei einen Kontroll- oder Datenfluss von Quelle nach Senke ab, wobei Quelle und Senke jeweils Knoten bzw. Function-Slices aus `FS` sind. Darüber hinaus existiert eine weitere Liste `r` mit Kontrollflusskanten, welche im Graphen als rückgerichtet markiert wurden.

Die Menge der Kanten in `r` ist eine Teilmenge von `CF`. Das Ergebniss ist eine Menge aller Pipelines. Diese werden wiederum durch ein 2-Tupel repräsentiert. Das erste Element verweist auf eine Schleife innerhalb des Function-Slice-Graphen, referenziert über dasjenige Function-Slice, von welchem die rückgerichtete Kante der Schleife ausgeht. Das zweite Element des 2-Tupels ist eine Liste mit allen Kontrollflusskanten, welche einer Grenze von einer Pipelinestufe zur nächsten entsprechen.

Das Ergebnis der Analyse für die Beispielanwendung aus Kapitel 6 ist in Abbildung 8.12 gezeigt. Das Ergebnis für eine Untersuchung der MPEG-2-Referenzimplementierung ist in Abbildung 8.13 dargestellt. Hierbei wurden 72 Schleifen untersucht und im Mittel 5,2 Pipelinestufen ermittelt. Jede Schleife, welche als Pipeline ausgeführt werden kann, wurde in dieser Darstellung farblich hinterlegt und die Ergebnisse der Stufenanalyse eingezeichnet. Durch die Transparenz der farblichen Kennzeichnung wird die verschachtelte Struktur innerhalb des Programmsystems deutlich.

8.3 Auswirkung der Sichtbarkeits- und Gültigkeitsbereiche von Variablen

Die Fragmentierung hat zu Folge, dass die Sichtbarkeits- bzw. Gültigkeitsbereichen von Variablen angepasst werden müssen. Für vormals blocklokale Variablen kann durch die Fragmentierung ein Übergang zu Objekten mit weitreichenderer Wirksamkeit notwendig sein.

8.3.1 Variablen-Alias

Im einfachsten Fall repräsentiert je ein Variablenname genau einen Speicherort. Jede Schreiboperation auf diese Variable führt dazu, dass der Wert an eben dieser Speicherstelle geändert wird. Im Quellcode kann es jedoch vorkommen, dass ein Variablenname unterschiedliche Speicherstellen referenziert, je nachdem an welchen Stellen im Programm der Name verwendet wird. In C wird der Gültigkeitsbereich durch Blockgrenzen begrenzt (vgl. Kapitel 2.2), in objektorientierten Sprachen bekommt jedes Objekt eine Instanz des gleichen Namens. Umgekehrt kann eine Speicherstelle jedoch auch durch mehrere Symbole referenziert werden (Alias). Da C einen statischen Gültigkeitsbereich verwendet, ist erst zur Kompilierzeit zu ermitteln, welches Objekt wo verwendet wird.

Eine Transformation hin zu nebenläufigen Threads kann Variablen daher nicht nur anhand ihres Namens betrachteten, sondern muss den Kontext berücksichtigen.

8.3.2 Globale Variablen

Seien vier Programmeinheiten U_1 bis U_4 gegeben, so dass sowohl U_1 als auch U_3 eine Speicherstelle x beschreiben, und U_2 ebenso wie U_4 den Wert von x lesen. Durch die Ausführungsreihenfolge ergäben sich dann die Datenflusskanten $U_1 \rightarrow_{DF} U_2$ und $U_3 \rightarrow_{DF} U_4$. Falls zusätzlich kein Datenfluss $\{U_1, U_3\} \rightarrow \{U_2, U_4\}$ existiert, können

```
Eingaben:
  // Liste mit allen Function-Slices (Knoten des FSG)
  FS = {FS_1, FS_2, ... }

  // Liste mit allen Datenflusskanten
  DF = {(Quelle1, Senke1),(Quelle2,Senke2),...}

  // Liste mit allen Kontrollflusskanten
  CF = {(Quelle1, Senke1),(Quelle2,Senke2),...}

  //Liste mit allen rückgerichteten Kontrollflusskanten
  r  = {(Quelle1, Senke1),(Quelle2,Senke2),...}

Algorithmus:
  k = alle FS_i welche Quelle einer Kontrollflusskante aus r sind
  pipelines = {}

  Für alle Knoten s aus k
  |   pipelinestufen = {}
  |   besuchte_DF = {}
  |   w = ( bestimme längsten Kontrollflusspfad mit s als Quelle der
  |         ersten Kontrollflusskante UND s als Ziel der letzten
  |         Kontrollflusskante )
  |
  |   Für alle Kontrollflusskanten c in w
  |   |   df_quelle = Quelle von c
  |   |   df_senke  = ( Menge aller Knoten aus w, die Teil des Weges von der
  |   |                 Senke aus c zu s sind, so dass dieser Weg eine Teil
  |   |                 von w ist )
  |   |   df_tmp    = ( Menge aller Datenflusskanten mit Quelle aus df_quelle
  |   |                 und Senke aus df_senke )
  |   |
  |   |   Wenn (df_tmp nicht leer) UND (df_tmp keine Datenflusskante aus
  |   |   |                             besuchte_DF enthält),
  |   |   |   füge besuchte_DF die Elemente aus df_tmp hinzu
  |   |   |_  füge pipelinestufen c hinzu
  |   |_
  |_  füge pipelines das Tupel (s,pipelinestufen) hinzu


Ausgabe:
  pipelines
```

Listing 8.1: Algorithmus zum Bestimmen von Pipeline-Stufen

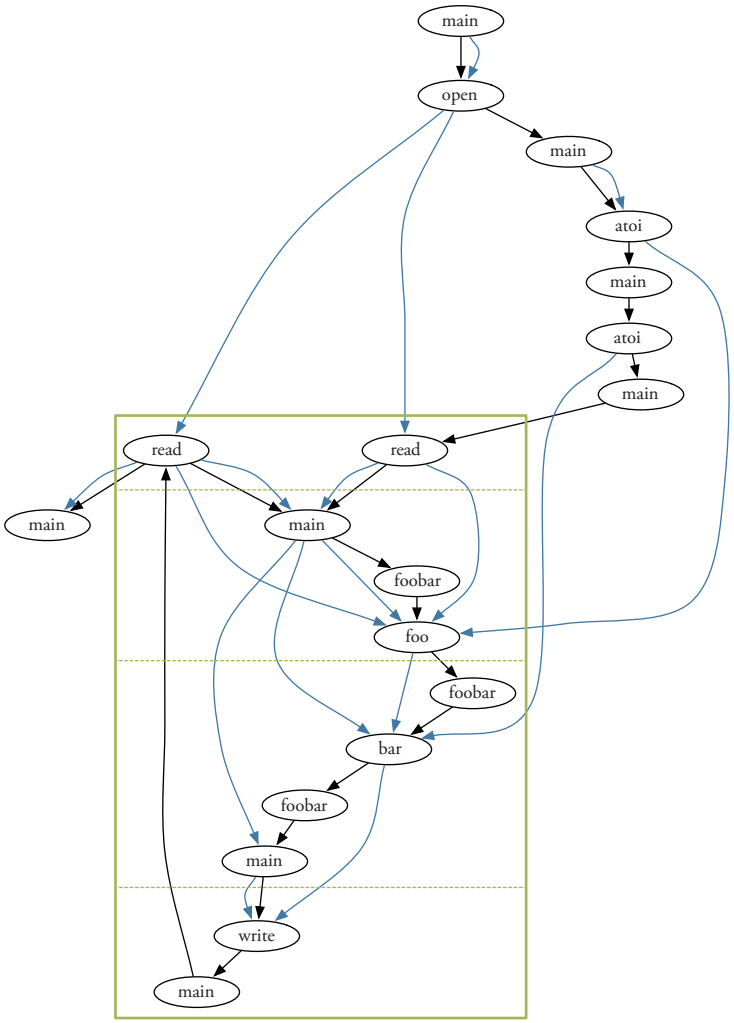


Abbildung 8.12: Function-Slice-Graph mit gekennzeichnetem Pipeline-Bereich (Rechteck) und der Unterteilung in Pipelinestufen (gestrichelte horizontale Linien)

Abbildung 8.13: Pipelinebereiche innerhalb der MPEG-2-Referenzimplementierung als grün gefärbte Rechtecke

die Paare $\{U_1, U_2\}$ und $\{U_3, U_4\}$ nebenläufig ausgeführt werden. Erfolgt der zuvor beschriebene Datenfluss über eine gemeinsame globale Variable, und somit nur über eine Speicherstelle, so muss an dieser Stelle eine Variablenduplizierung vorgenommen werden, da ansonsten mittels `load`- und `store`-Instruktionen von mehreren Stellen auf einen Speicherbereich zugegriffen wird. Das Gleiche gilt für jede Art der Variablen im Programm. Für jeden Datenfluss muss sichergestellt sein, dass ein exklusiver Speicherbereich zur Verfügung steht, welcher nicht von anderen Threads manipuliert werden kann. Alternativ kann sichergestellt werden, dass jeder vergangene Wert vorgehalten wird, bis er definitiv nicht mehr benötigt wird. Dann ist es allerdings notwendig, zusätzlich die korrekte Zuordnung von Produzent zu Konsument sicherzustellen. Diese Zuordnung wird auch notwendig, wenn durch eine Schleife U_1 das Datenobjekt n-fach schreibt (bei jedem Schleifendurchlauf), und U_2 n-mal liest. Abbildung 8.14 skizziert mögliche Konstellationen zweier Programmeinheiten über dem zeitlichen Verlauf.

8.3.3 Funktionslokale Variablen

Funktionslokale Variablen realisieren möglicherweise einen Datenfluss bzw. Informationsaustausch zwischen Blöcken. Eine Verwendung finden sie jedoch z. B. auch als Schleifenindex mehrerer aufeinenander folgender Schleifen oder als temporäre Variable. In jeder Schleife wird die Variable zunächst, beim ersten Betreten erneut initialisiert. Zwischen Blöcken, welche aus voneinander unabhängigen Schleifen hervorgegangen sind, treten somit read-write- oder write-write-Folgen über Blockgrenzen hinweg auf, je nachdem ob der Vorgänger zuletzt eine Schreib- oder Leseoperation durchgeführt hat.

Das Verhalten unterscheidet sich bezüglich der Auswirkung für die Parallelisierung nicht von dem der globalen Variablen.

8.3.4 Blocklokale Variablen

Blocklokale Variablen spielen für die Datenflussanalyse und somit für darauf basierende Transformationen keine Rolle, solange Basisblöcke nicht in Subblöcke zerlegt werden müssen. Durch eine Zerlegung ist die Variable nicht mehr als lokales Datenobjekt zu betrachten, sondern als eines welches zu Datenflüssen führen kann. Auch hier ergeben sich die für globale Variablen beschriebenen Probleme. Zusätzlich muss dafür Sorge getragen werden, dass die Variablen nicht mehr blocklokal sind. Dazu muss die Deklaration auf die nächst höhere Ebene verschoben werden.

LLVM realisiert blocklokale Variablen im Zwischencode als funktionslokale Variablen. Die letzt genannte Transformation muss in diesem Fall nicht vorgenommen werden. Um den Gültigkeitsbereich entsprechend der Definition im Quelltext zu gewährleisten, werden, falls notwendig, Duplikate erzeugt. Angenommen innerhalb einer Funktion `f()` werden in mehreren Blöcken blocklokale Variablen mit dem selben Namen `x` verwendet. Dann erzeugt der Compiler im Zwischencode funktionslokale Variablen `x1, x2,` usw.

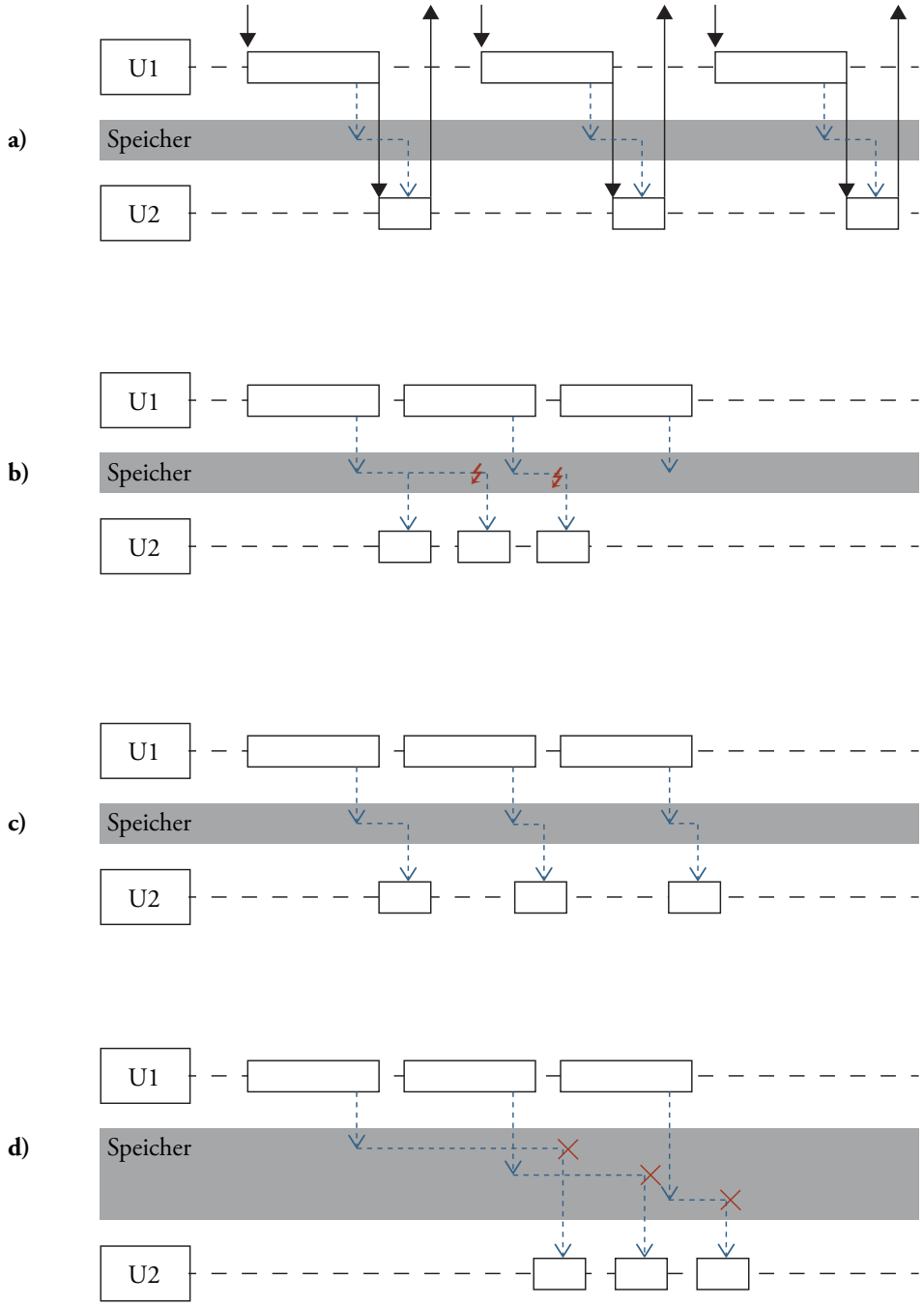


Abbildung 8.14: Mögliche zeitliche Verläufe zweier Programmeinheiten U_1 und U_2 mit einer Datenabhängigkeit $U_1 \rightarrow U_2$.
a) Sequentielle Verarbeitung (schwarz: Kontrollfluss, blau gestrichelt: Datenfluss)
b) Datenkollision durch unterschiedliche Laufzeiten der beiden Programmeinheiten (⚡)
c) Vermeiden der Kollision durch Synchronisation
d) Kollisionsvermeidung durch zusätzliche Speicherstellen, Speicher kann nach dem letzten Lesevorgang freigegeben werden (×)

8.4 Abflachen der Aufrufhierarchie

Aus den zuvor beschriebenen Schritten ergibt sich ein Modell angereichert um Informationen bezüglich möglicher Parallelität. Um das aus der Analyse hervorgehende Potential in der Programmausführung ausnutzen zu können, bedarf es eines weiteren Transformationsschrittes. Der Programmablauf soll auf den Fragmenten, welche das Modell definiert, basieren. Wenn diese Einheiten unabhängig voneinander zur Ausführung gebracht werden sollen, dann müssen die Instruktionen, welche Bestandteil einer konzeptionellen Einheit sind, auch in der Umsetzung eine abgeschlossene Einheit bilden. Ziel ist es daher, die Funktionalität jeder Einheit in eine gekapselte Funktion zu transformieren. Die so entstandenen Einheiten müssen in geeigneter Reihenfolge zur Ausführung gebracht werden. Hierzu dient das Prinzip einer flachen Aufrufhierarchie welches in der Ausprägung des Dispatchers beschrieben wird.

8.4.1 Code-Kapselung

Jeder Codeblock, welcher einer zuvor definierten Einheit mit konstanten Abhängigkeiten entspricht, muss als eigenständige Einheit ausgeführt werden können. Der hier beschriebene Ansatz sieht eine Transformation in ein objektorientiertes Modell vor. Die Funktionalität einer Einheit (z. B. eines Function-Slices) wird als Methode einer Klasse gekapselt.

Die zu verarbeitenden Daten können durch den Dispatcher als Parameter des Konstruktors übergeben werden, falls sie bereits vorliegen. Für alle weiteren eingehenden Daten existieren Setter-Funktionen. Für jedes Attribut, welches eingehende Daten speichert, ist daher zusätzlich ein „valid“-Flag zu setzen.

Eine Methode kapselt den ursprünglichen Quellcode, welcher zur Ausführung gebracht werden kann. Darüber hinaus können Methoden zur Abfrage des Objektzustandes und der Menge der validen und benötigten Daten vorhanden sein. Zustände eines Objekts werden aus der Menge

$$\{undefined, idle, running, calling, done\}$$

gesetzt. Der Vektor der Rückgabewerte enthält neben den Daten auch Informationen über das Ziel dieser Daten.

Das auf diese Art generierte Objekt für ein Function-Slice mit dem Parameter y und einem Rückgabewert ist in Listing 8.2 aufgeführt. Der Parameter wird in diesem Fall mit Erzeugen der Objektinstanz dem Konstruktor übergeben. Alle produzierten Daten werden einem Vector hinzugefügt, dieser wird an die aufrufende Instanz (siehe nächster Abschnitt) zurückgegeben.

8.4.2 Dispatcher-Prinzip

Die Programmstruktur sequentieller Programme zeichnet sich durch eine Verschachtelung von Programmstrukturen und Unterprogrammaufrufen aus. Somit ergibt sich eine Aufrufhierarchie. Um die Ausführung steuern zu können (Scheduling), ist es notwendig,

```
FS1::FS1 (int y)
{
  this->y = y;
  this->state = ReturnValueState::StateType::StateIdle;
}

std::vector<ReturnValue> FS1::run (void)
{
        printf("FS1 run\n");
  this->state = ReturnValueState::StateType::StateRunning;

  bb0 : {
    /* st  */ v0 = y;
    /* ld  */ bb0_l0 = v0;
    /* mul */ bb0_l1 = bb0_l0 * (int)0x2;
    /* st  */ v2 = bb0_l1;
    /* ld  */ bb0_l2 = v2;
    /* mul */ bb0_l3 = bb0_l2 * (int)0x2;
    /* st  */ v1 = bb0_l3;
    /* gto */ goto bb2;
  }

  bb1 : {
    /* gto */ goto bb2;
  }

  bb2 : {
    /* ld  */ bb2_l0 = v1;
    /* ret */ //this->return_value = bb2_l0;
    /*     */ //this->state = StateDone;
    /*     */ //this->current_bid = 2;
    /*     */ //return this;
                std::vector<ReturnValue> ret;
    ReturnValue val( 1, NULL, ReturnValueState::StateType::StateDone );
    ret.push_back(val);
    return ret;
  }
}
```

Listing 8.2: Beispiel-Quellcode für C++-Objekt zur Kapselung der Funktionalität eines Function-Slices

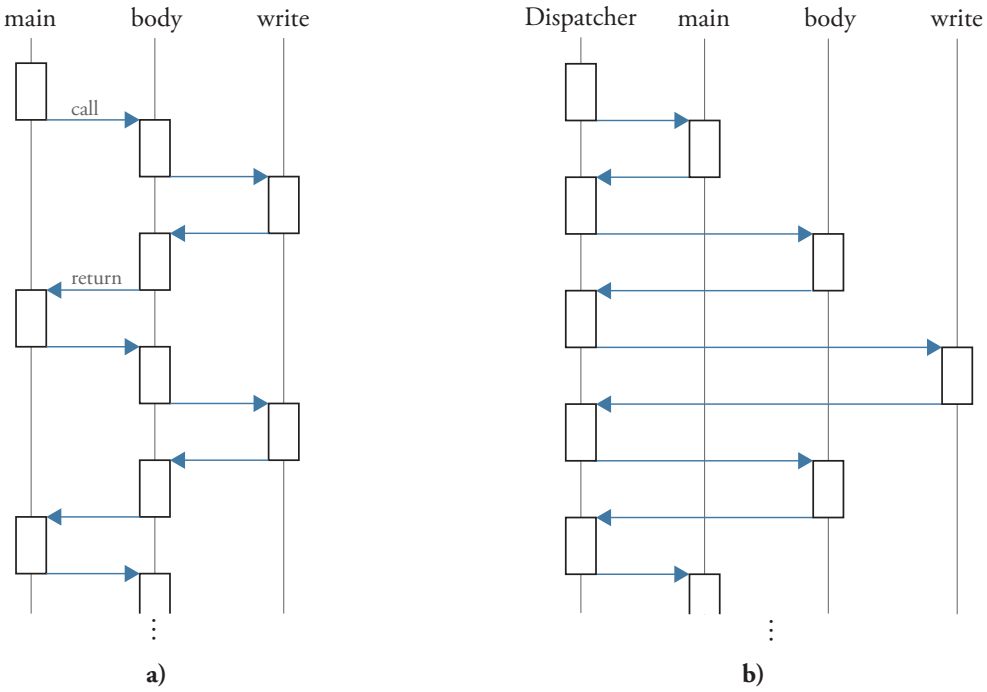


Abbildung 8.15: Transformation eines Programms von der ursprünglichen Aufruffolge (a) in das Dispatchermodell mit der veränderten Aufruffolge (b)

an einer zentralen Stelle Informationen über den Zustand des Programmsystems zu sammeln und anhand dieser Daten die weitere Ausführung zu planen. Eine Transformation des ursprünglichen Programms in Knoten mit konstanten und bekannten Abhängigkeiten untereinander eröffnet weitere Möglichkeiten zur Ausführungsplanung. Jeder Knoten kann als gekapselte Einheit betrachtet werden und somit auch als separate Einheit ausgeführt werden.

Ein Dispatcher verwaltet die zur Verfügung stehenden Mittel und sorgt für den benötigten Informationsfluss. Dieses Prinzip lässt sich auf die Ausführung transformierter Programmsysteme anwenden. Jeder Knoten kann ausgeführt werden, wenn alle benötigten Eingangsdaten bereitstehen. Um dem Dispatcher die vollständige Kontrolle über die Ausführung zu übertragen darf der Kontrollfluss mit Fertigstellung aller Berechnungen im Knotenkörper nicht an den Nachfolgeknoten übergehen, sondern muss zu dem Dispatcher zurückführen (siehe Abbildung 8.15). Jedes Knotenende, vormals $return, call, branch, switch$ wird somit zu einem $return_{to\ dispatcher}$. Das Ziel der ursprünglichen Anweisung zur Steuerung des Kontrollflusses muss zusätzlich an den Dispatcher übermittelt werden.

Der Kontrollfluss wird somit immer zwischen einem aktiven Knoten und dem Dispatcher wechseln. Für den Fall, dass mehrere Prozessoren für die Datenverarbeitung zur Verfügung stehen, kann der Dispatcher mehrere Kontrollflussfäden verwalten und somit Parallelität realisieren. Hierfür werden zu Laufzeitbeginn Threads erzeugt, welche Auf-

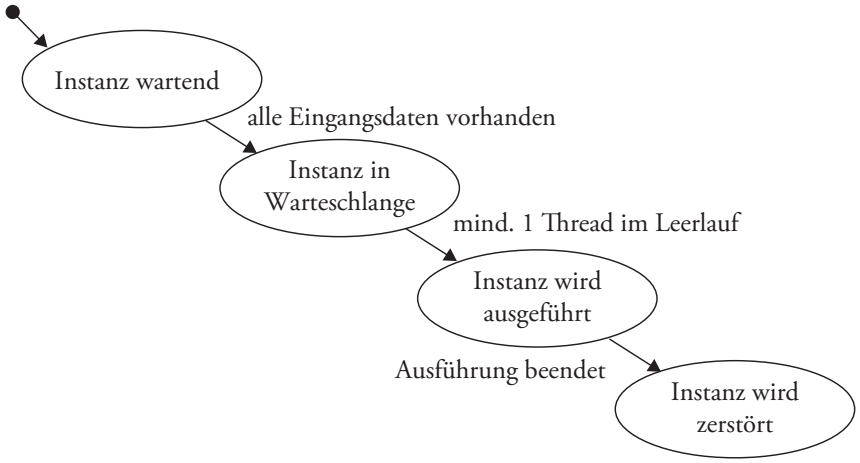


Abbildung 8.16: Zustände und deren Übergänge einzelner Knoteninstanzen im Dispatcher

gaben aus einem Aufgabenpool bearbeiten. Es werden immer mindestens zwei Threads ausgeführt. Ein Thread enthält den Dispatcher, die ausführbaren Programmteile werden den anderen Threads zugeordnet.

8.4.3 Scheduling

Der Dispatcher verwaltet eine Warteschlange (*Queue*) mit allen Objekten, die ausgeführt werden können. Zu Beginn können alle Objekte ohne Eingangsdatenabhängigkeiten in die Warteschlange abgelegt werden. Wenn ein gestartetes Objekt seine Ausführung beendet, meldet es die Zustandsänderung und produzierte Daten an den Dispatcher. Dieser ordnet die empfangenen Daten den jeweiligen Zielen zu. Dabei kann er entweder eine neue Instanz der Datensenken erzeugen, oder falls bereits ein passendes Objekt exisitiert diesem die neuen Daten hinzufügen. Im Anschluss wird für alle in der Schwebe befindlichen Objekte, solche die sich noch nicht in der Warteschlange befinden, geprüft, ob alle benötigten Daten vorhanden sind. Ist die Bedingung erfüllt, wird das Objekt in die Warteschlange gelegt (vgl. Abbildung 8.16).

Im Dispatcher wird neben der Ausführungs-Warteschlange eine Liste mit allen noch nicht ausführbaren Objekten vorgehalten. Wenn ein Objekt seine Ausführung beendet hat, bekommt der Dispatcher die berechneten Daten jeweils um das Ziel des Datenflusses angereichert. Der Dispatcher aktualisiert die Daten in den Zielobjekten und prüft jeweils, ob alle benötigten Daten zugewiesen wurden. Ist dies der Fall, wird das Objekt in die Warteschlange verschoben.

Die Warteschlange kann im einfachsten Fall als FIFO implementiert werden. Eine optimierte Reihenfolge kann sich ergeben, wenn zunächst alle Aufgaben bearbeitet werden, von denen besonders viele weitere abhängig sind. Somit kann sichergestellt werden, dass eine möglichst gute Auslastung der Hardware erreicht wird.

Ein Scheduling als reine Datenflussanwendung, bei der Einheiten nur Aufgrund der konsumierten Daten zur Ausführung gebracht werden, ist aufgrund von Kontrollstrukturen im sequentiellen Programm bzw. daraus abgeleiteter Kontrollabhängigkeiten nicht möglich. Ausgehend von einer datenbasierten Ausführung ergibt sich teilweise direkte Parallelität. Dieses Verhalten, und wo zusätzliche Bedingungen notwendig sind, wird im folgenden Abschnitt diskutiert.

8.4.4 Effekte bei ausgewählten Kontrollflussstrukturen

Im wesentlichen ist das Betrachten von Verzweigungen und Schleifen und deren naturgemäßes Verhalten im Dispatcher relevant. Dadurch wird offensichtlich, inwieweit Parallelität unmittelbar durch Transformation für die Ausführung per Dispatcher ausgenutzt wird.

Eine Verzweigung im Kontrollfluss entspricht alternativen Pfaden durch das Programm. Wenn der Kontrollfluss nicht weiter betrachtet wird, sondern die Ausführung von Programmfragmenten durch Vorhandensein der Eingangsdaten getriggert wird, dann führt dies zum Start aller alternativer Pfade. Auf diese Weise könnte, selbst wenn das die Verzweigung beeinflussende Datum noch nicht ermittelt wurde, eine spekulative Ausführung der Alternativen erfolgen. Die korrekten Ergebnis sind dann im Anschluss über einen neuen Knoten aus den spekulativ produzierten Daten auszuwählen. Alternativ ist die Logik der Verzweigung zu erhalten, indem Kontrollabhängigkeitskanten berücksichtigt werden. Diese sind im Dispatchermodell in Datenkanten entsprechend boolscher Variablen zu realisieren.

Im Kontrollfluss der sequentiellen Ausgangsbeschreibung auftretende Schleifenstrukturen, wie sie in Kapitel 8.2 beschrieben sind, bzw. mögliche Fälle auftretender Kombinationen sind in Tabelle 8.1 aufgeführt. Das Dispatcherverhalten für per Eingangsdaten getriggerter Ausführungen wird ebenso wie für die korrekte Ausführung notwendige, zusätzlich einzuführende Abhängigkeiten beschrieben.

Tabelle 8.1: Mögliche Abhängigkeits-Kombinationen innerhalb von Schleifen

Fall	Körper abhängig von Index	Abbruchbedingung abh. von Körper	Schleifengetragene Abh. vorhanden
1	nein	nein	nein
2	nein	nein	ja
3	nein	ja	nein
4	nein	ja	ja
5	ja	nein	nein
6	ja	nein	ja
7	ja	ja	nein
8	ja	ja	ja

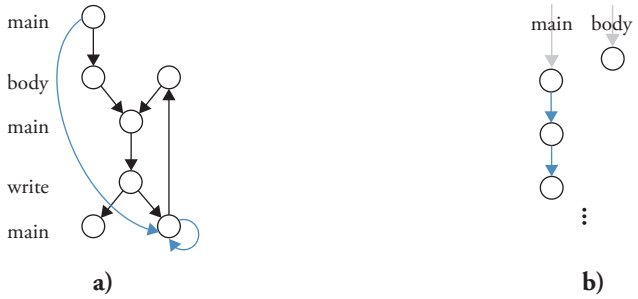


Abbildung 8.17: Datenflusskanten im Function-Slice-Graph für den Fall 1 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

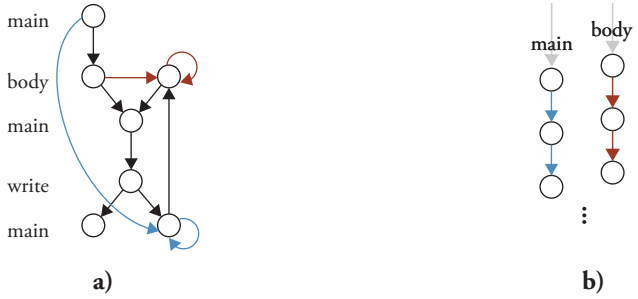


Abbildung 8.18: Datenflusskanten im Function-Slice-Graph für den Fall 2 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

Fall 1 Abbildung 8.17a) zeigt die auftretenden Datenflusskanten, in diesem Fall ist dies nur der Datenfluss der Indexvariablen von einer Ausführungsinstanz zur nächsten. Knoten werden zur Ausführung gebracht, wenn alle eingehenden Datenabhängigkeiten erfüllt sind. Der Schleifenkörper (body) hat entweder keine eingehenden Datenflusskanten, oder aber Abhängigkeiten, die vor dem ersten Betreten der Schleife erfüllt wurden. In diesen beiden Fällen wird dieser Knoten unabhängig, und möglicherweise vor der Schleife ausgeführt. Da keine Abhängigkeit von der Indexvariablen zum Schleifenkörper besteht, wird der Schleifenkörper nur einmal und nicht wiederkehrend aufgerufen. Hingegen wird der Indexraum eigenständig durchlaufen, bis die Abbruchbedingung erfüllt ist. Ein möglicher resultierender zeitlicher Ablauf ist in Abbildung 8.17b) dargestellt.

Fall 2 Wie zuvor in Fall 1 wird der Indexraum nebenläufig und unabhängig vom Schleifenkörper durchlaufen. Sowohl die erste Instanz der Indexiteration als auch des Schleifenkörpers produziert Eingangsdaten für die nächste Instanz des jeweils selben Knotens. Somit triggert die erste Ausführung seine eigenen folgenden Ausführungsinstanzen. Für den Schleifenkörper gibt es keine Abbruchbedingung, so dass eine Endlosschleife entsteht. Analog zu Fall 1 sind die Darstellungen in Abbildung 8.18 aufgeführt.

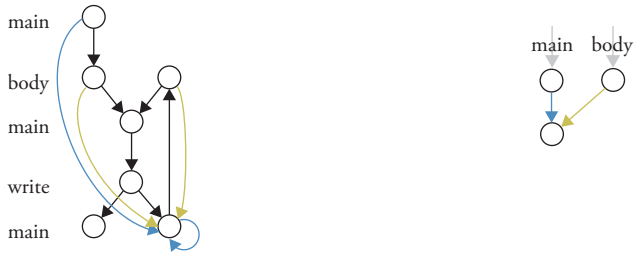


Abbildung 8.19: Datenflusskanten im Function-Slice-Graph für den Fall 3 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

Abbildung 8.20: Datenflusskanten im Function-Slice-Graph für den Fall 4 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

Fall 3 Die Inkrementierung bzw. Dekrementierung des Schleifenindex ist bei dieser Konstellation erst nach Beendigung des Schleifenkörpers möglich. Dieser wird jedoch wie in Fall 1 nur einmal initial ausgeführt. Es kommt daher in diesem Fall zu keinem Schleifenverhalten (vgl. Abbildung 8.19).

Fall 4 Die erste Schleifenausführung initiert zwar die weiteren (siehe Abbildung 8.20), jedoch erfolgt keine Rückkopplung der Abbruchbedingung auf den Schleifenkörper. Auch hier entstünde eine Endlosschleife für den Schleifenkörper. Das Durchschreiten des Indexraumes startet erst nach Beendigung der Schleifenkörperinstanz, läuft dann jedoch bis die Abbruchbedingung erfüllt ist.

Fall 5 Dieser Fall wird in Abbildung 8.21 dargestellt. Jeder Indexwert triggert den zugehörigen Schleifenkörper und nebenläufig den Schritt zum nächsten Indexwert. Wird die Abbruchbedingung erfüllt, werden die Daten für weitere Schritte nicht mehr geliefert. Somit endet die Ausführung wie im sequentiellen Fall. Die Ausführung entspricht derer eines maximalen Unrollings. Da keine Abhängigkeiten zwischen den Instanzen des Schleifenkörpers vorhanden sind, können diese auch parallel ausgeführt werden (siehe Abbildung 8.25b))

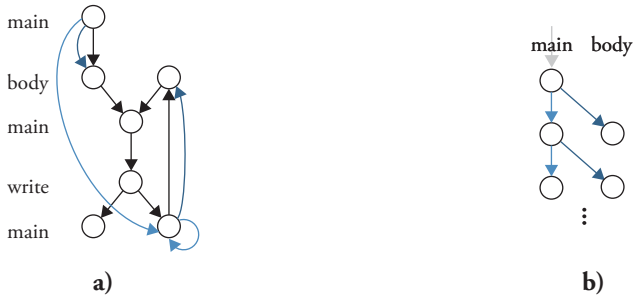


Abbildung 8.21: Datenflusskanten im Function-Slice-Graph für den Fall 5 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

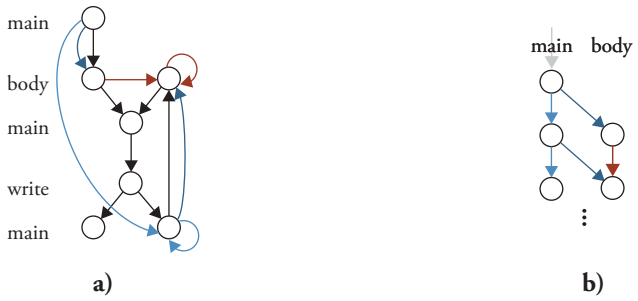


Abbildung 8.22: Datenflusskanten im Function-Slice-Graph für den Fall 6 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

Fall 6 Wie Fall 5, jedoch verhindert die zusätzliche schleifengetragene Abhängigkeit eine parallele Ausführung der einzelenen Schleifenkörperinstanzen sowohl theoretisch als auch in der Realisierung. Die Indexiteration kann jedoch bereits parallel zum Schleifenkörper erfolgen (vgl. Abbildungen 8.22 und 8.25c)).

Fall 7 Der Index triggert die Ausführung des Schleifenkörpers, dieser wiederum triggert den nächsten Indexschritt. Die Ausführung entspricht somit dem sequentiellen Ablauf (siehe Abbildung 8.23)

Fall 8 Entspricht Fall 7, die zusätzliche schleifengetragene Abhängigkeit hat keine Auswirkung auf die Parallelisierbarkeit (Abbildung 8.24).

Die Fälle 5–8 werden im Dispatchermodell inhärent korrekt, und mit der maximal möglichen Parallelität umgesetzt. Für die Fälle 1–4 ist es erforderlich künstlich zusätzliche Datenkanten einzufügen. Das fehlende mehrfache Instanziieren des Schleifenkörpers kann durch Einfügen einer Indexabhängigkeit innerhalb des Schleifenkörpers, bzw. einer Kontrollabhängigkeitskante als Pseudodatenfluss, erreicht werden. Das Verhalten gleicht dann dem in den Fällen 5–8.

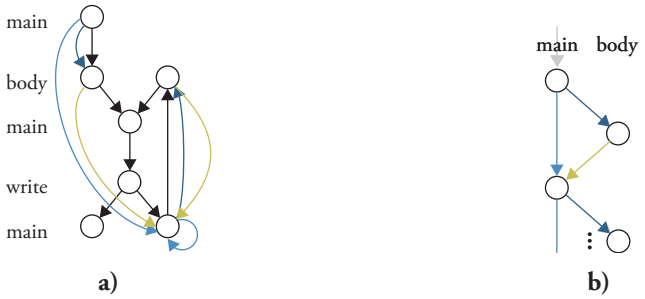


Abbildung 8.23: Datenflusskanten im Function-Slice-Graph für den Fall 7 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

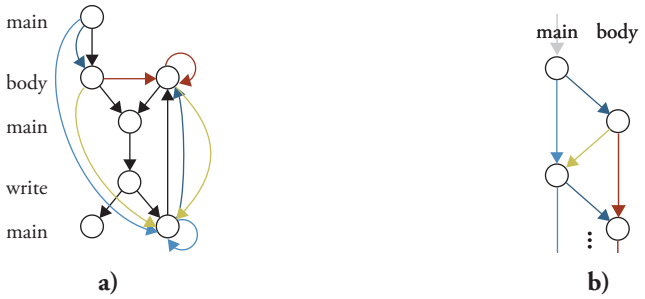


Abbildung 8.24: Datenflusskanten im Function-Slice-Graph für den Fall 8 (a), Resultierender zeitlicher Ablauf im Dispatcher (b)

Die zuvor beschriebenen Fälle beziehen sich auf diejenigen Schleifen, die nur einen Schleifenkörperknoten aufweisen, bzw. bei Schleifen mit umfangreicheren Körpern auf all diejenigen Knoten, welche keine Dateneingangskanten von anderen ebenfalls zum selben Schleifenkörper (eine Ausnahme bilden schleifengetragene Abhängigkeiten) gehörenden Knoten besitzen. Jeder dieser Knoten triggert die Ausführung weiterer Knoten innerhalb des Schleifenkörpers. Mehrere voneinander datenunabhängige Knoten in einem Schleifenkörper lassen sich per Loop Distribution parallelisieren. Das hier beschriebene Dispatchermodell setzt dieses Verfahren inhärent um. Mehrere voneinander datenabhängige Einheiten bilden Datenflussstrukturen. Sobald der erste Knoten zur Ausführung gebracht wurde, triggert dieser die Ausführung der davon abhängenden Knoten, vormals Folgeknoten im Kontrollfluss. Somit wird auch in diesem Fall eine, dem Pipelining entsprechende Parallelität umgesetzt. Solange keine schleifengetragenen Abhängigkeiten vorhanden sind, werden Schleifenkörpersequenzen parallel ausgeführt.

Eine zusätzliche Maßnahme ist bei allen Programmeinheiten, die in mehreren Instanzen, also mehrfach, ausgeführt werden, notwendig. Die Instanzen sind zu erzeugen sowie diese im Programmablauf korrekt zuzuordnen. Von Programmeinheiten produzierte Daten wurden, wie zuvor beschrieben, um das Ziel des Datenflusses angereichert. Solange keine Zielinstanz in der Warteschlange vorhanden ist, muss sie logischerweise erzeugt werden. Wenn jedoch bereits eine Instanz vorhanden ist, darf ein Eingangsdatum

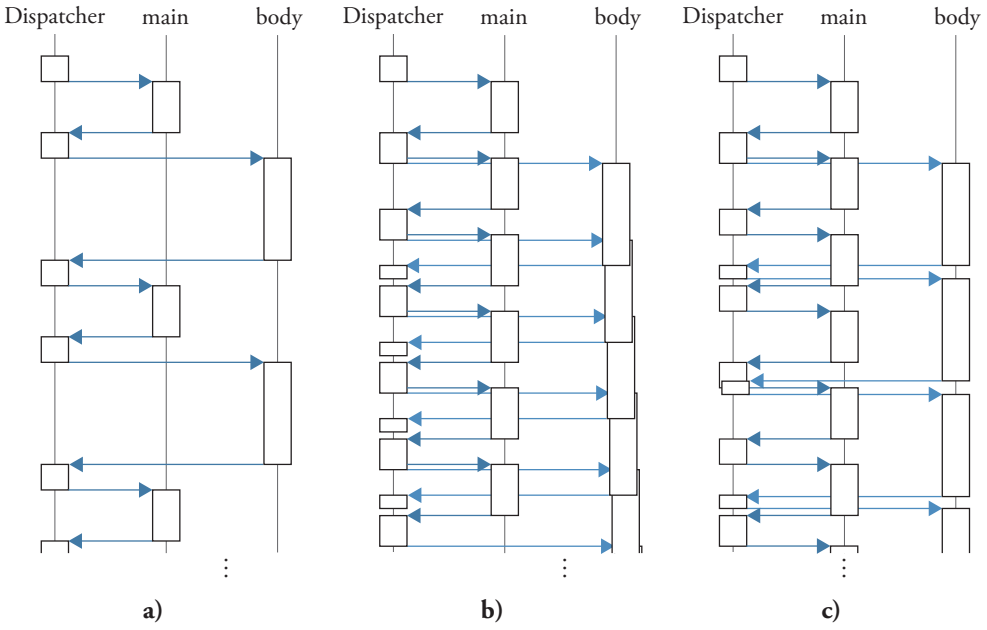


Abbildung 8.25: Parallele Ausführung eines Programms nach Überführung in das Dispatchermodell für exemplarische Fälle: a) sequentielle Ausführung, b) Ausführung wie sie durch den Fall 5 beschrieben wird, c) Ausführung wie sie im 6. Fall auftritt

nicht überschrieben werden. Stattdessen ist ein Duplikat der Instanz zu erzeugen und in diesem das entsprechende Eingangsdatum zu überschreiben. Dadurch wird sichergestellt, dass Daten, die bei allen Instanzen identisch sind, auch alle erreichen. Bei mehreren Daten, möglicherweise aus unterschiedlichen Quellen, die sich in den Instanzen unterscheiden, müssen diese zusammengeführt werden. Dabei muss nicht nur die Zieleinheit des Datenflusses mit den Daten mitgeführt werden, sondern auch ein Instanzindentifikator, wie z. B. die Indexvariable. Daten ohne Instanzidentifikator werden an alle in der Warteschlange vorhandenen Instanzen übergeben.

8.4.5 Ergebnisse für eine Beispiel-Implementierung

Eine Bewertung des präsentierten Dispatcher-Modells kann anhand von Laufzeituntersuchungen durchgeführt werden. Die zuvor verwendeten Beispielanwendungen bieten entweder wenig Optimierungspotential, oder erfordern, wie im Falle des MPEG-2 Beispiels, aufgrund der Komplexität manuelle Eingriffe. Daher soll hier ein konstruiertes Beispiel mit einem der zuvor behandelten Fälle des Schleifenverhaltens im Dispatcher verwendet werden. Durch die Verwendung einer Testanwendung lassen sich Parameter, welche Einfluss auf die Laufzeit haben, gezielt variieren, und das Verhalten des Gesamtsystems in deren Abhängigkeit bewerten.

Als Basis für die Datenerhebung wird eine Anwendung, welche im Schleifenkörper eine Summe über W Ganzzahlwerte berechnet, realisiert. Die folgenden Messwerte

```
Hauptprogramm() {
 limit = 100
 W     = 5000
 Für n = 1 ... limit
   Funktionsaufruf body(n,W)
}

body(n,W) {
  Summe=0
  Für m = n ... n+W
    Summe = Summe + m
  Ergebnis: Summe
}
```

Listing 8.3: Pseudocode der sequentiellen Anwendung zur Laufzeitmessung

entstammen einer Ausführung auf einem Intel Core i7 (2600, 3,4 GHz) mit 4 Kernen (durch Hyperthreading 8 mögliche Threads). Die Ausführung der Knoten basiert auf einem durch die Java Klasse *Executors* zur Verfügung gestellten *newFixedThreadPool* variabler Größe. Zusätzlich wird der Dispatcher in einem eigenen Thread ausgeführt.

Die sequentielle Programmbeschreibung (Pseudocode in Lisiting 8.3) wurde in das Function-Slice-Modell überführt. Die konkreten Function-Slices und deren Datenflussabhängigkeiten sind in Abbildung 8.26 dargestellt. Die Kombination auftretender Datenflusskanten im Function-Slice-Graph entspricht dem zuvor beschriebenen Fall 5. Das Function-Slice des Schleifenkörpers (`body`) wird innerhalb einer Schleife aufgerufen, so dass es in der durch die Schleifengrenze L_{limit} definierten Anzahl instanziiert wird. Neben der Zahl der ausgeführten Threads N_T und dem Arbeitsaufwand im Schleifenkörper W kann die Zahl der Durchläufe der Schleife im Hauptprogramm N_I variiert werden.

Der Graph in Abbildung 8.27 gibt die Ausführungsdauer des vollständigen Programms (verteilt auf 4 Threads) in Abhängigkeit von L_{limit} wieder. Wie zu erwarten, zeigt sich hier eine lineare Abhängigkeit. Die Laufzeitmessungen wurden jeweils 200 mal durchgeführt. Bei der Ausführung kommt es zu Streuungen in den Ergebnissen. Die Standardabweichung der Messwerte wird in den Graphen durch die Fehlerbalken dargestellt. Die Ausführung findet in einer Laufzeitumgebung statt, und nicht nativ auf dem Prozessor. Je nach Aktivität anderer auf dem System laufender Prozesse, variiert der Verwaltungsaufwand (durch Scheduling im Betriebssystem).

Ausgangsbasis für die in dieser Arbeit behandelten Analysen und Transformationen ist eine sequentielle Programmbeschreibung. Deren Ausführung stellt die Referenz für die Laufzeit des Gesamtsystems dar. Der Vergleich der Implementierung im Dispatcher-Modell zu der rein sequentiellen Variante ist in Abbildung 8.28 zu sehen. Im sequentiellen Fall ist der Arbeitsaufwand des Gesamtsystems linear abhängig von W. Durch die Parallelisierung mittels des Dispatcher-Modells verringert sich die Ausführungszeit ab einem Arbeitsaufwand W von ca. 60000. Für kleine W ist durch den zusätzlichen Verwaltungsaufwand der Nutzen jedoch aufgehoben. Daher spielt die Granularitätsbetrachtung, wie in Kapitel 5 bereits dargelegt, für ein geeignetes Modell eine große Rolle.

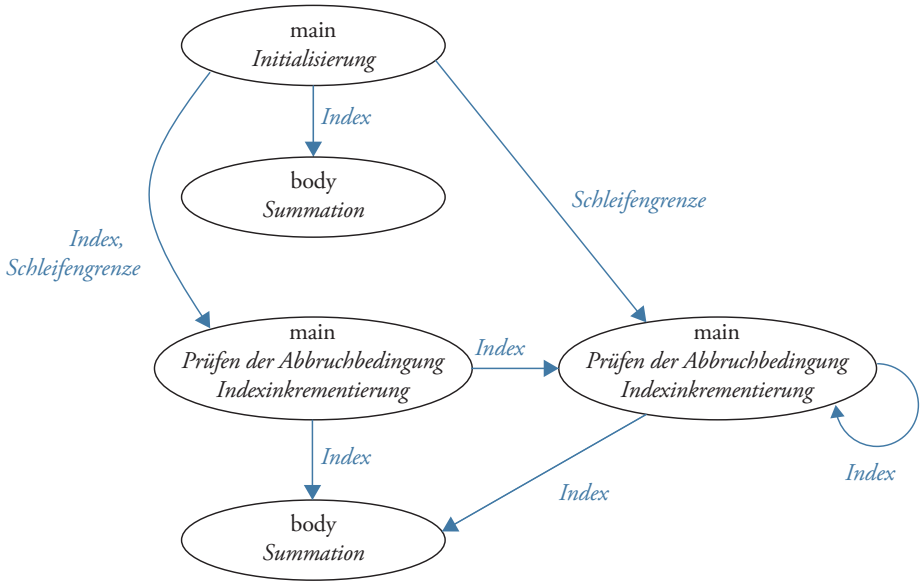


Abbildung 8.26: Function-Slices und Datenflusskanten der Implementierung zur Laufzeituntersuchung

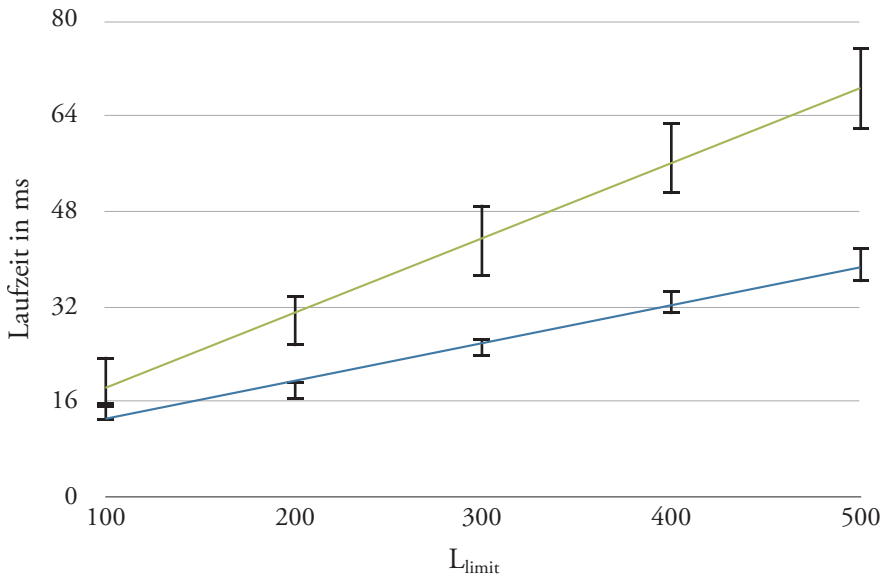


Abbildung 8.27: Laufzeit der Dispatcher-Implementierung in Abhängigkeit von der Schleifengrenze L_{limit} für unterschiedlichen Rechenaufwand im Schleifenkörper (blau: $W = 5000$, grün: $W = 500000$)

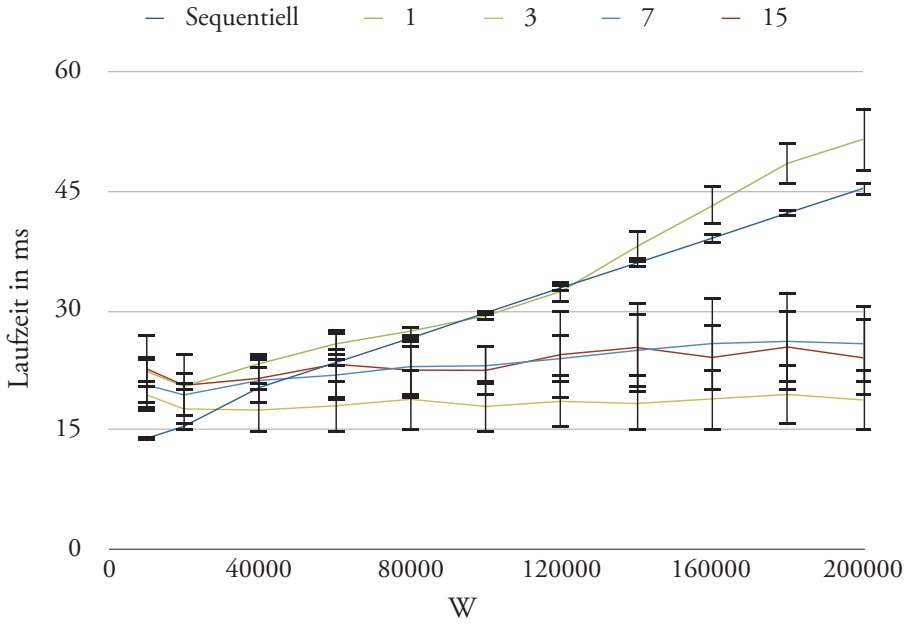


Abbildung 8.28: Laufzeit in Abhängigkeit von dem Arbeitsaufwand W. Für die rein sequentielle Implementierung und die transformierte Variante (Ausführung in 1, 3, 7 und 15 Threads)

Für eine Ausführung von nur einem Thread zur Bearbeitung der Function-Slices zeigt sich der Mehraufwand des Dispatcher-Modells im Vergleich zu der Referenz.

In Abbildung 8.28 ist bereits zu sehen, dass eine Ausführung in 4 Threads (1 Thread für den Dispatcher, 3 für den Thread-Pool) zu den geringsten Ausführungszeiten führt. In der Betrachtung der Laufzeit in Abhängigkeit der Thread-Pool-Größe (Abbildung 8.29) zeigen sich im Bereich von $N_T = 3$ bis $N_T = 5$ die kürzesten Ausführungszeiten. Bei nur einem Thread für die Ausführung kann Parallelität nicht ausgenutzt werden, der Overhead der Transformation führt zu vergleichsweise großen Laufzeiten. Eine wachsende Anzahl an Threads über 5 hinaus bringt keinen weiteren Nutzen.

8.5 Weitere Auswirkungen einer geänderten Ausführungs-Reihenfolge

Ein Programm mit einer geänderte Ausführungsreihenfolge wird solange korrekte Ergebnisse produzieren, solange alle Abhängigkeiten erfüllt sind. Jedes Programm produziert Ausgaben. Wenn jedes Teilergebnis korrekt berechnet wird, muss jedoch darüber hinaus sichergestellt werden, dass die Ausgaben auch in der korrekten Reihenfolge erfolgen. Dies entspricht im wesentlichen einer Eingabeabhängigkeit.

Neben den bekannten und offensichtlichen Abhängigkeiten wie sie beschrieben wurden, können jedoch auch versteckte Abhängigkeiten z. B. in C-Bibliotheken existieren.

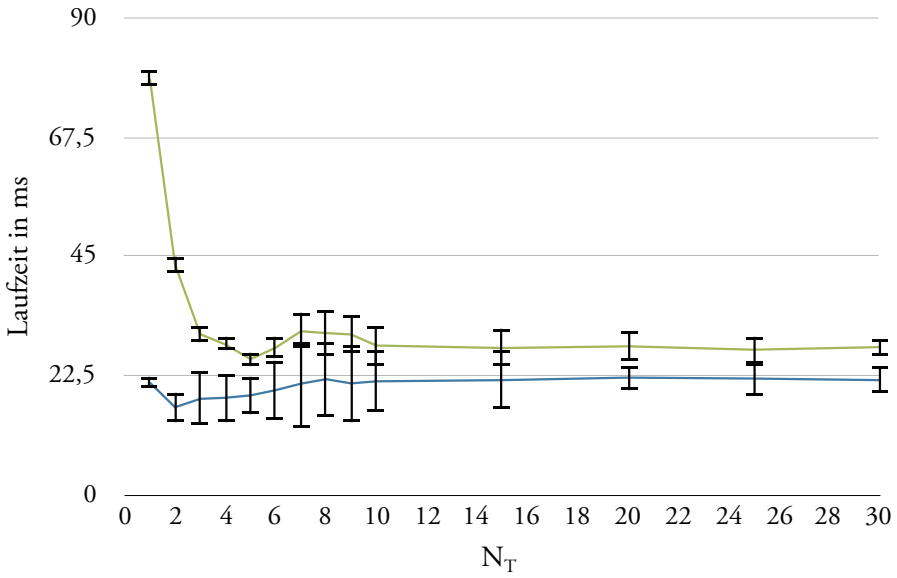


Abbildung 8.29: Laufzeit der Dispatcher-Implementierung in Abhängigkeit von der Anzahl der Threads im Thread-Pool (N_T) (grün: $W = 6000$, blau: $W = 600000$)

Ein Beispiel stellen die Funktionen `open`, `read` bzw. `write` und `close` dar. `open` liefert einen File-Deskriptor zurück. Die entsprechende Variable wird mit dem Öffnen einer Datei einmal geschrieben. Die anderen Low-Level-Datei-I/O-Funktionen greifen nur lesend auf den Wert zu. Nach den zuvor beschriebenen Transformationen könnten alle Aufrufe von `read` oder `close` sofort ausgeführt werden. Ein sofortiger Aufruf von `close` würde jedoch alle weiteren I/O-Aktivitäten verhindern. An dieser Stelle wird der Eingriff eines Entwicklers notwendig. Im Sinne eines rechnergestützten Verfahrens können entsprechende Abhängigkeitskanten im Profil ergänzt werden. Die in Kapitel 7 gezeigte Visualisierung kann in diesem Fall als Hilfsmittel dienen.

Kapitel 9

Schlussbetrachtung

9.1 Resümee

In dieser Arbeit wurden Komponenten für eine automatisierte Parallelisierung sequentieller Programme betrachtet. Deren Zusammenspiel wurde zu Beginn als Konzept skizziert.

Für die sich aus der Intention der Arbeit ergebenden Schritte, konnte auf in der Literatur beschriebene Verfahren zurückgegriffen werden. Das ursprünglich die Forschung initiierende Modell der Function-Slices weist für den späteren Verlauf wesentliche Eigenschaften auf. Die Anforderungen an das Modell stellten eine vielversprechende Randbedingung für die Untersuchung und Ausnutzung von Parallelität dar. Die teils bekannten Maßnahmen zur Parallelisierung wurden auf die Besonderheiten in diesem Modell hin untersucht. Dabei wurden Schwächen dieses Modells offensichtlich. Daher ist auch die Rückbesinnung auf Blöcke (Basisblöcke bzw. Subblöcke) für diese Schritte relevant. Das Problem der Kombination zu Einheiten mit einer idealen Granularität konnte auf diese Weise nicht erreicht werden. Hier lohnt sich ein Ausblick auf zukünftige Forschungsaktivitäten.

Die Umsetzung der zuvor beschriebenen Teilschritte kann in einem technischen System realisiert werden. Ein möglicher Workflow ist in Abbildung 9.1 abgebildet. Dabei bilden sich die Schritte

1. Quellcode übersetzen in Zwischencode (IR), Zerlegung in Blöcke
2. Bestimmung der statischen Abhängigkeiten
3. Bestimmung der dynamischen Abhängigkeiten (Laufzeitumgebung, RTE) und Bildung von Statistiken über mehrere Durchläufe
4. Bildung der Programmeinheiten
5. Transformation in Threads; Codekapselung
6. Ausführen des geänderten Programms

heraus. Die Übersetzung in den Zwischencode und mögliche Ansatzpunkte für eine Transformation der Basisblöcke in Subblöcke an Unterprogrammaufrufstellen ist im Detail in Abbildung 9.2 zu sehen. Als Eingabe dienen C-Quellcode-Dateien (.c) und C-Header-Dateien (.h). Das LLVM-Compilerfrontend übersetzt diese in Bitcode (.bc) oder

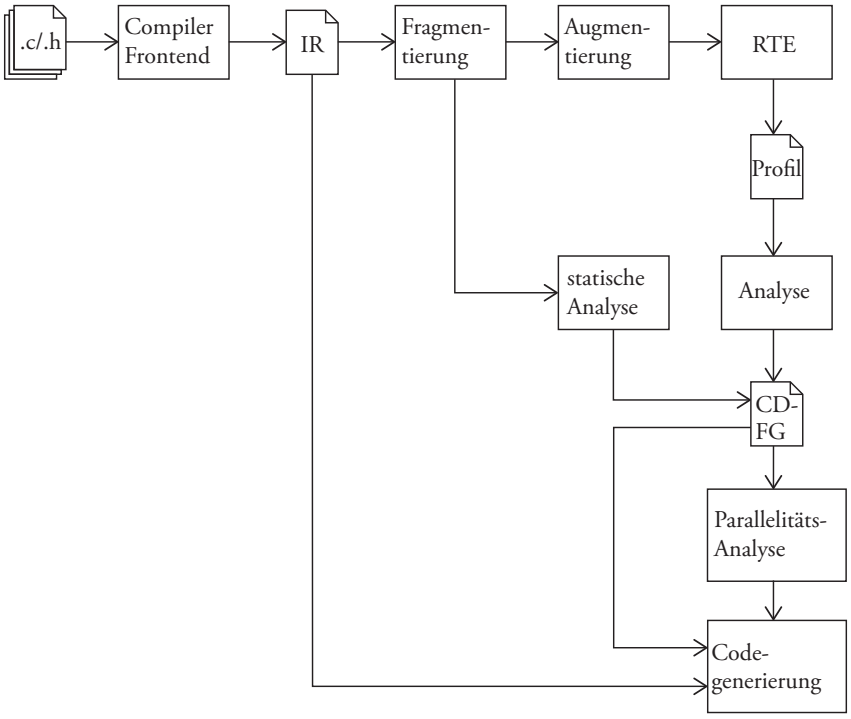


Abbildung 9.1: Übersicht über ein mögliches System

in LLVM-Zwischencode (.ll). Der Zwischencode kann mittels des LLVM-Assemblers (`llvm-as`) in Bitcode überführt werden. Für jede C-Quellcode-Datei entsteht so eine Bitcode-Datei. Der Linker (`llvm-link`) verknüpft diese zu einem Gesamt-Bitcode. Über den Disassembler (`llvm-dis`) lässt sich eine vollständige Zwischenrepräsentation des Programmsystems erzeugen. Diese bietet die Ausgangsbasis für die Fragmentierung, bzw. Analyse. Eine Schwierigkeit stellt dabei die Kombination dynamischer und statischer Analysen auf Basis der transformierten Zwischenrepräsentation dar. Das in der Arbeit verwendete LLILA-System entstammt der laufenden Forschung und weist an einigen Punkten unzuverlässiges Verhalten auf, insbesondere konnte es nicht den Entwicklungen des LLVM der letzten Jahre folgen. Unter anderem die Rückverfolgung von Werten zu Unterprogrammen bzw. deren Rückgabewerten ist teilweise unvollständig. Für die vorliegende Arbeit war dies jedoch nicht relevant, da die dynamische Analyse nicht Teil der Arbeit, sondern nur ein Hilfsmittel darstellte.

Es wurden alternative Fragmentierungskonzepte aufgeführt. Das Function-Slice-Modell weist durch seine Konzeption und den daraus resultieren Eigenschaften neue Möglichkeiten der Restrukturierung auf. Die existierenden Mängel des Modells können jedoch umgangen werden, indem Verzweigungsknoten eingeführt werden bzw. durch Analysen, welche nicht auf dem reinen Kontroll- und Datenflussgraph basieren, sondern auch das zeitliche Verhalten berücksichtigen.

Die Modelleigenschaften führten darüber hinaus zu einem Vergleich mit Schaltungsstrukturen. Eine für die Analyse hilfreiche Visualisierung kann davon profitieren. So

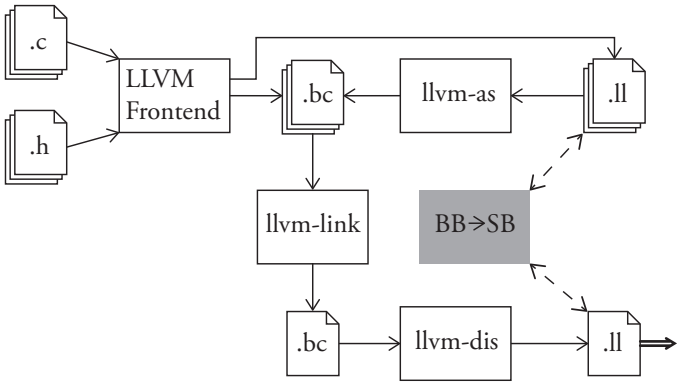


Abbildung 9.2: Compiler-Komponenten zur Erzeugung des Zwischencodes und Transformationskomponente für die Zerlegung von Basisblöcken in Subblöcke

wurde ein neuer Ansatz zur Softwarevisualisierung präsentiert. Die Ergebnisse zeigen, dass dieses Verfahren zur reinen Softwarevisualisierung Schwächen aufweist. Bei großen bzw. komplexen Systemen, wie sie gerade bei für eine Parallelisierung prädestinierten Anwendungen auftreten, ergibt sich keine übersichtliche Darstellung. Hier wären mehrere hierarchische Ebenen wünschenswert. Auch wenn mit Function-Slice-Clustern ein Weg in diese Richtung eingeschlagen wurde, könnten alternative Modelle mit gleichen Eigenschaften zu einer besseren Gliederung führen.

Die Visualisierung mittels Schaltsystemlayouts ist im Vergleich zu einfachen Graph-Layout-Algorithmen sehr performant. Zudem konnten geeignete Maßnahmen zur Komplexitätsreduktion angewandt werden. Aus der Visualisierung als Schaltung wurden Konzepte deutlich, wie etwa die Kombination von mehreren Flüssen zu einer Leitung, welche verzweigt. Für Datenflüsse wäre es darüber hinaus möglich verschiedene Datenobjekte in eine Leitung zu bündeln und diese in der Betrachtungsumgebung bei Bedarf zu expandieren, bzw. weitere Informationen über Datenobjekte, statistische Verteilung des Betretens oder der transportierten Datenvolumina anzuhängen und über Selektion abzurufen. Die einzige Voraussetzung dafür ist, dass die Analysen entsprechende Daten liefern. Bei komplexen Problemstellungen bleibt diese Form der Visualisierung nutzbar, lässt jedoch keine offensichtlichen Rückschlüsse auf Optimierungspotential zu. Vielmehr lassen sich interaktiv Abhängigkeiten innerhalb von Programmsystemen verfolgen, indem Verbindungen ausgewählt, und in der Darstellung hervorgehoben werden. Auf Basis der Erkenntnisse, die durch überschaubare Programme geringerer Komplexität gewonnen wurden, können jedoch automatisierte Analysen vergleichbare Strukturen auch in komplexen Graphen finden. Insgesamt bietet sich eine gut strukturierte einfache Handhabung komplexer Graphen. Gerade für eine Kontrollflussbetrachtung eignet sich die eingeführte Hierarchie, möglicherweise erweiterbar um zusätzliche Stufen. Die Erkenntnisse über Eigenschaften der Function-Slices und die Transformation in Netzlisten wurden für die Transformation in eine Schaltungsstruktur erneut aufgegriffen.

In Kapitel 8 wurden Analysen zu Parallelität in sequentiellen Programmen bzw. in der Function-Slice-Darstellung betrachtet. Dabei wurden unterschiedliche Konzepte auf

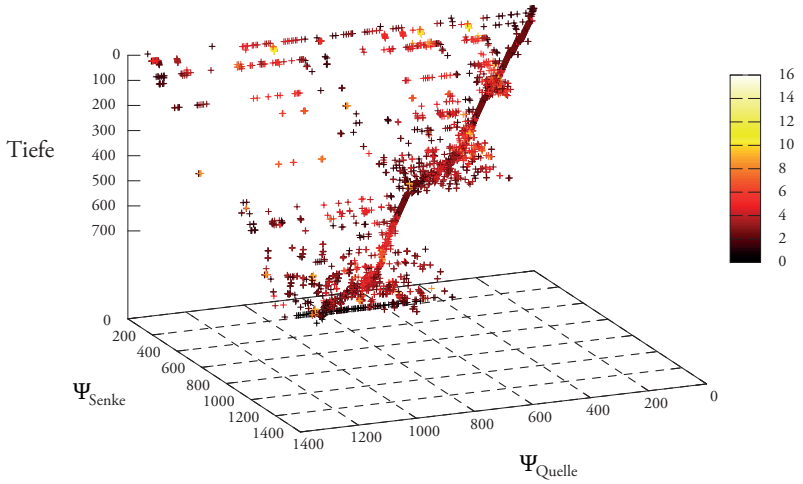


Abbildung 9.3: Datenflussaktivität des MPEG-2-Dekoders; Jeder Datenfluss wird durch einen Punkt mit repräsentiert (Position ergibt sich aus der Quellen-Senken-Kombination und der Tiefe im Function-Slice-Graph), das transportierte Datenvolumen wird durch die Farbe dargestellt [Gre14]

dieses Modell übertragen. Es konnte gezeigt werden, welche Art der Parallelität unter welchen Voraussetzungen nutzbar ist. Dies setzt voraus, dass die Ausführung des Programms entsprechend angepasst wird. Daher wurden diesbezüglich Konzepte entwickelt. Das Dispatchermodell setzt Function-Slices in Threads um. Hierbei wurde auf erforderliche Transformationen eingegangen, welche eine parallele Ausführung erst möglich machen können. Zudem wurde eine Untersuchung des Verhaltens bei Kontrollflussstrukturen durchgeführt. Es konnte gezeigt werden, dass das Dispatchermodell inhärent die zuvor beschriebenen Parallelitätsarten umsetzen kann.

9.2 Ausblick

Eine Abkehr vom Function-Slice-Modell und Besinnung auf Subblöcke geht, wie gezeigt, mit einer für die Parallelisierung ungeeigneten Granularität einher. Daher wären alternative Konzepte zur Gruppierung denkbar. Ein Flaschenhals bei der parallelen Ausführung von Teilen eines Programms stellt der Datenaustausch zwischen den einzelnen Instanzen dar. Daher wäre eine Untersuchung möglicher Fragmentierung mit dem Ziel, den Datenfluss zwischen Fragmenten zu minimieren, interessant. Zugleich darf dabei jedoch der Workload der Fragmente nicht außer Acht gelassen werden. Dieser ist im Idealfall gleichmäßig auf alle Fragmente verteilt. Für die Zerlegung von Graphen gibt es Graphpartitionierungsalgoritmen wie z. B. den Kernighan-Lin-Algorithmus.

Der beschriebene Ansatz zur Transformation in eine logische Schaltung nutzt die aus der Visualisierung ersichtlichen Eigenschaften aus. Dabei wird auf die Umsetzung der Datentransformationsanweisungen des ursprünglichen Programms nicht eingegangen. Hier konnte gezeigt werden, unter welchen Umständen bzw. mit welchen zusätzlichen

Maßnahmen diese Umsetzung möglich ist. Die Folge dieser Transformation ist jedoch, dass mehr Speicher vorhanden sein muss, da nicht mehr eine zentrale Realisierung, sondern dezentrale Kopien erforderlich sind. Der Gewinn ist gegenüber dem zusätzlichen Hardwareaufwand für jedes Problem separat zu bewerten. Hinzu kommt, dass Quelltexte mit Unterprogrammen der Wiederverwendung von Code dient. Auch ähnliche oder identische Aufgaben werden als separate Einheiten in Hardware realisiert. Je nach Kontext ergeben sich andere Abhängigkeiten, realisiert als andere Verschaltung, und somit duplizierte funktionale Blöcke in der Hardware-Struktur.

Eine mögliche Abhilfe stellt die dynamische, partielle Rekonfiguration dar. Dies ist die Möglichkeit während des laufenden Betriebs Teil-Konfigurationen in ein FPGA zu laden. Somit werden Teile der Hardware ausgetauscht, ohne dass die verbleibenden Teile unterbrochen werden. Der Einsatz der partiellen Rekonfiguration ermöglicht Designs auf kleineren Geräten unterzubringen oder den Energieverbrauch zu senken. Abbildung 9.3 zeigt die Aktivität des MPEG-2-Dekoders. Es fällt auf, dass die Ausführung einen unstetigen Verlauf aufweist. Es wechseln sich sequentielle Folgen und ausgedehntere, parallelisierbare Gruppen ab. Aufgrund dieser Kaskadierung lässt sich prognostizieren, welche Bereiche zu bestimmten Zeiten nicht benötigt werden und somit nicht als aktive Hardware vorgehalten werden müssen.

Literatur

[ACE+12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even u. a. „Par4all: From convex array regions to heterogeneous computing“. In: *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*. 2012.

[AI88] Arvind und Robert A. Iannucci. „Two fundamental issues in multiprocessing“. In: *Parallel Computing in Science and Engineering*. Hrsg. von Rüdiger Dierstein, Dieter Müller-Wichards und Hans-Martin Wacker. Bd. 295. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1988, S. 61–88. ISBN: 9783540189237.

[Amd67] Gene M Amdahl. „Validity of the single processor approach to achieving large scale computing capabilities“. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, S. 483–485.

[Ami12] Mehdi Amini. „Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators“. Diss. Ecole Nationale Supérieure des Mines de Paris, 2012.

[BB01] B. Bhattacharya und S.S. Bhattacharyya. „Parameterized dataflow modeling for DSP systems“. In: *Signal Processing, IEEE Transactions on* 49.10 (2001-10), S. 2408–2421. ISSN: 1053-587X.

[BBC+11] Anna Beletska, Wlodzimierz Bielecki, Albert Cohen u. a. „Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations“. In: *Parallel Computing* 37.8 (2011). Follow-on of ISPDC'2009 and HeteroPar'2009, S. 479–497. ISSN: 0167-8191.

[BBK+08] Günther Bengel, Christian Baun, Marcel Kunze und Karl-Uwe Stucky. *Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicoreprozessoren, Multiprozessoren, Cluster und Grid (German Edition)*. Vieweg+Teubner Verlag, 2008. ISBN: 3834803944.

[BHC97] Sanjeev Banerjia, William A Havanki und Thomas M Conte. „Treegion scheduling for highly parallel processors“. In: *Euro-Par'97 Parallel Processing*. Springer, 1997, S. 1074–1078.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul u. a. „Cilk: An Efficient Multithreaded Runtime System“. In: *Journal of Parallel and Distributed Computing* 37.1 (1996), S. 55–69. ISSN: 0743-7315.

[BL93] J.T. Buck und E.A. Lee. „Scheduling dynamic dataflow graphs with bounded memory using the token flow model“. In: *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*. Bd. 1. 1993-04, 429–432 Vol.1.

[BRS+07] Chris Bennett, Jody Ryall, Leo Spalteholz und Amy Gooch. „The Aesthetics of Graph Visualization“. In: *Proceedings of the Third Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging*. Computational Aesthetics'07. Alberta, Canada: Eurographics Association, 2007, S. 57–64. ISBN: 9783905673432.

[BS06] Saisanthosh Balakrishnan und Gurindar S. Sohi. „Program Demultiplexing: Data-flow Based Speculative Parallelization of Methods in Sequential Programs“. In: *SIGARCH Comput. Archit. News* 34.2 (2006-05), S. 302–313. ISSN: 0163-5964.

[CBK+14] Simone Campanoni, Kevin Brownell, Svilen Kanev u. a. „HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs“. In: *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press. 2014, S. 217–228.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen u. a. „Efficiently Computing Static Single Assignment Form and the Control Dependence Graph“. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991-10), S. 451–490. ISSN: 0164-0925.

[CGP07] Lei Chai, Qi Gao und Dhabaleswar K. Panda. „Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System“. In: *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*. CCGRID '07. Washington, DC, USA: IEEE Computer Society, 2007, S. 471–478. ISBN: 0769528333.

[CHM+11] Daniel Cordes, Andreas Heinig, Peter Marwedel und Arindam Mallik. „Automatic extraction of pipeline parallelism for embedded software using linear programming“. In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE. 2011, S. 699–706.

[CJH+12a] Simone Campanoni, Timothy M Jones, Glenn Holloway u. a. „HELIX: Making the extraction of thread-level parallelism mainstream“. In: *IEEE Micro* 4 (2012), S. 8–18.

[CJH+12b] Simone Campanoni, Timothy Jones, Glenn Holloway u. a. „HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing“. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, Kalifornien: ACM, 2012, S. 84–93. ISBN: 9781450312066.

[CJH+12c] Simone Campanoni, Timothy Jones, Glenn Holloway u. a. „The HELIX project: overview and directions“. In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, S. 277–282.

[Col95] Jean-François Collard. „Automatic parallelization of while-loops using speculative execution“. In: *International Journal of Parallel Programming* 23.2 (1995), S. 191–219.

[Col97] Jean-Francois Collard. *Automatic Parallelization of Higher-Order Languages in the Polytope Model*. 1997.

[DBM+09] C. Dave, Hansang Bae, Seung-Jai Min u. a. „Cetus: A Source-to-Source Compiler Infrastructure for Multicores“. In: *Computer* 42.12 (2009-12), S. 36–42. ISSN: 0018-9162.

[DeM79] Tom DeMarco. *Structured analysis and system specification*. Yourdon Press, 1979.

[Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007. ISBN: 3540465049.

[DIN 66001] Norm DIN 66001 Dezember 1983. *Informationsverarbeitung; Sinnbilder und ihre Anwendung*

[DIN 66261] Norm DIN 66261 November 1985. *Informationsverarbeitung; Sinnbilder für Struktogramme nach Nassi-Shneiderman*

[DM01] Oliver Diessel und George Milne. „Hardware compiler realising concurrent processes in reconfigurable logic“. In: *Computers and Digital Techniques, IEE Proceedings-*. Bd. 148. 45. IET. 2001, S. 152–162.

[Dud13] Bibliographisches Institut, Hrsg. *Duden*. "[Online; Stand: 2013-01-19]". 2013. URL: http : / / www . duden . de / node / 680160 / revisions/1215086/view.

[DV95] Alani Darte und Frederic Vivien. „A classification of nested loops parallelization algorithms“. In: *Emerging Technologies and Factory Automation, 1995. ETFA'95, Proceedings., 1995 INRIA/IEEE Symposium on*. Bd. 1. IEEE. 1995, S. 217–234.

[EG95] DJV Evans und Andrzej Gościński. *Automatic Identification of Parallel Units and Synchronisation Points in Programs*. Technical report TR C95/08. Deakin University. School of Computing und Mathematics, 1995.

[eXCite] *eXCite - Y Explorations*. "[Online; Stand: 2015-10-06]". URL: %7Bhttp://www.yxi.com%7D.

[Fea96] Paul Feautrier. „Automatic parallelization in the polytope model“. In: *The Data Parallel Programming Model*. Springer, 1996, S. 79–103.

[Fis81] J. A. Fisher. „Trace Scheduling: A Technique for Global Microcode Compaction“. In: *IEEE Trans. Comput.* 30.7 (1981-07), S. 478–490. ISSN: 0018-9340.

[GDG+03] Sumit Gupta, Nikil Dutt, Rajesh Gupta und Alex Nicolau. „SPARK: A high-level synthesis framework for applying parallelizing compiler transformations“. In: *VLSI Design, 2003. Proceedings. 16th International Conference on*. IEEE. 2003, S. 461–466.

[GK83] Daniel D Gajski und Robert H Kuhn. „Guest editors' introduction: New VLSI tools“. In: *Computer* 16.12 (1983), S. 11–14.

[GN47] Herman H. Goldstine und John von Neumann. „Planning and coding of problems for an electronic computing instrument“. In: *report prepared for the U.S. Army Ord. Dept.* I (1947).

[Gra] *Graphviz - Graph Visualization Software*. "[Online; Stand: 2015-08-04]". URL: http://www.graphviz.org.

[Gre07] Carsten Gremzow. „Compiled low-level virtual instruction set simulation and profiling for code partitioning and ASIP-synthesis in hardware/-software co-design“. In: *Proceedings of the 2007 summer computer simulation conference*. SCSC. San Diego, California: Society for Computer Simulation International, 2007, S. 741–748. ISBN: 1565553160.

[Gre14] Carsten Gremzow. „Verhaltensbasierter Digitalentwurf: Reflexive Verhaltenssynthese auf Basis virtueller Instruktionssatz-Architekturen“. Habilitationsschrift. Berlin: Technische Universität Berlin, 2014.

[Gri04] Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures*. Universität Passau, 2004.

[Gri96] Martin Griebl. „The mechanical parallelization of loop nests containing while loops“. Diss. Universität Passau, 1996.

[GTA06] Michael I. Gordon, William Thies und Saman Amarasinghe. „Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs“. In: *SIGARCH Comput. Archit. News* 34.5 (2006-10), S. 151–162. ISSN: 0163-5964.

[Gus88] John L Gustafson. „Reevaluating Amdahl's law“. In: *Communications of the ACM* 31.5 (1988), S. 532–533.

[Hai59] Lois M. Haibt. „A Program to Draw Multilevel Flow Charts“. In: *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*. IRE-AIEE-ACM '59 (Western). San Francisco, California: ACM, 1959, S. 131–137.

[HAM+05] Mary W Hall, Saman P Amarasinghe, Brian R Murphy u. a. „Interprocedural parallelization analysis in SUIF“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.4 (2005), S. 662–731.

[HGK06] Christian Hammer, Martin Grimme und Jens Krinke. „Dynamic Path Conditions in Dependence Graphs“. In: *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '06. Charleston, South Carolina: ACM, 2006, 58–67. ISBN: 1595931961.

[Hin01] Michael Hind. „Pointer analysis: Haven't we solved this problem yet?“ In: *PASTE'01*. ACM Press, 2001, S. 54–61.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen u. a. „The Superblock: An Effective Technique for VLIW and Superscalar Compilation“. In: *J. Supercomput.* 7.1-2 (1993), S. 229–248. ISSN: 0920-8542.

[HMM00] I. Herman, G. Melancon und M.S. Marshall. „Graph visualization and navigation in information visualization: A survey“. In: *Visualization and Computer Graphics, IEEE Transactions on* 6.1 (2000-01), S. 24–43. ISSN: 1077-2626.

[HP00] Michael Hind und Anthony Pioli. „Which Pointer Analysis Should I Use?“ In: *SIGSOFT Softw. Eng. Notes* 25.5 (2000-08), S. 113–123. ISSN: 0163-5948.

[HPR88] Susan Horwitz, Jan Prins und Thomas Reps. „On the adequacy of program dependence graphs for representing programs“. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, S. 146–157.

[HPR89] Susan Horwitz, Jan Prins und Thomas Reps. „Integrating Noninterfering Versions of Programs“. In: *ACM Trans. Program. Lang. Syst.* 11.3 (1989-07), S. 345–387. ISSN: 0164-0925.

[HR92] Susan Horwitz und Thomas Reps. „The Use of Program Dependence Graphs in Software Engineering“. In: *Proceedings of the 14th International Conference on Software Engineering*. ICSE '92. Melbourne, Australia: ACM, 1992, S. 392–411. ISBN: 0897915046.

[HRB88] S. Horwitz, T. Reps und D. Binkley. „Interprocedural Slicing Using Dependence Graphs“. In: *SIGPLAN Not.* 23.7 (1988-06), S. 35–46. ISSN: 0362-1340.

[Kah74] G. Kahn. „The semantics of a simple language for parallel programming“. In: *Information processing*. Hrsg. von J. L. Rosenfeld. Stockholm, Schweden: North-Holland, Amsterdam, 1974-08, S. 471–475.

[KB96] Byung-Kyoo Kang und James M. Bieman. „Using Design Cohesion to Visualize, Quantify, and Restructure Software“. In: *The 8th International Conference on Software Engineering and Knowledge Engineering, SEKE '96, Lake Tahoe, Nevada, USA, June 10-12, 1996*. Knowledge Systems Institute, 1996, S. 222–229. ISBN: 0964169932.

[KIB15] *KIB - Lines of Code*. 2015. URL: https : / / docs . google . com / spreadsheet / ccc ? key = 0Aqe2P9sYhZ2ndElxbjVTcnV2bHFOW % 20mUwSkt2bjZLdVE&usp=drive_web#gid=5%5Cnewline (besucht am 22. 06. 2015).

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua u. a. „Dependence Graphs and Compiler Optimizations“. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '81. Williamsburg, Virginia: ACM, 1981, S. 207–218. ISBN: 089791029X.

[Kog81] Peter M. Kogge. *Architecture of Pipelined Computers*. Mcgraw-Hill (Tx), 1981. ISBN: 0070352372.

[Kri03] Jens Krinke. „Advanced slicing of sequential and concurrent programs“. In: *Ausgezeichnete Informatikdissertationen 2003*. 2003, S. 101–110. URL: http://subs.emis.de/LNI/Dissertation/Dissertation4/article14.html.

[Kri04] Jens Krinke. „Advanced Slicing of Sequential and Concurrent Programs“. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, S. 464–468. ISBN: 0769522130.

[Kuc77] David J. Kuck. „A Survey of Parallel Machine Organization and Programming“. In: *ACM Comput. Surv.* 9.1 (1977-03), S. 29–59. ISSN: 0360-0300.

[KW01] Michael Kaufmann und Dorothea Wagner, Hrsg. *Drawing graphs - methods and models*. Berlin New York: Springer, 2001. ISBN: 3540420622.

[LA92] R.M. Lee und V.H. Allan. „Advanced software pipelining and the program dependence graph“. In: *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*. 1992-12, S. 208–211.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth u. a. „Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation“. In: *SIGPLAN Not.* 40.6 (2005-06), S. 190–200. ISSN: 0362-1340. DOI: 10.1145/1064978.1065034.

[LD98] Arun Lakhotia und Jean-Christophe Deprez. „Restructuring programs by tucking statements into functions“. In: *Information and Software Technology* 40.11–12 (1998), S. 677–689. ISSN: 0950-5849.

[Len93] Christian Lengauer. „Loop parallelization in the polytope model“. In: *CONCUR'93*. Springer. 1993, S. 398–416.

[Lie06] Jens Lienig. *Layoutsynthese elektronischer Schaltungen - Grundlegende Algorithmen für die Entwurfsautomatisierung* -. 1. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2006. ISBN: 9783540299424.

[Lig09] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009. ISBN: 9783827420565.

[LLH+13] Shuangchen Li, Yongpan Liu, Xiaobo Sharon Hu u. a. „Optimal partition with block-level parallelization in c-to-rtl synthesis for streaming applications". In: *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. IEEE. 2013, S. 225–230.

[LPC11] Feng Li, Antoniu Pop und Albert Cohen. „Advances in Parallel-Stage Decoupled Software Pipelining Leveraging Loop Distribution, Stream-Computing and the SSA Form". In: *WIR 2011: Workshop on Intermediate Representations*. Florent Bouchez, Sebastian Hack und Eelco Visser. 2011, pp–29.

[LPC12] Feng Li, Antoniu Pop und Albert Cohen. „Automatic extraction of coarse-grained data-flow threads from imperative programs". In: *IEEE Micro* 4 (2012), S. 19–31.

[LZ04] C.-H. Lung und M. Zaman. „Using Clustering Techniques to Restructure Programs". In: *Proc. of the Int'l Conf. on Software Engineering Research and Practice*. Las Vegas, NV, 2004, S. 853–858.

[McC04] Shawn McCloud. „Catapult c synthesis-based design flow: Speeding implementation and increasing flexibility". In: *White Paper, Mentor Graphics* (2004).

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen u. a. „Effective Compiler Support for Predicated Execution Using the Hyperblock". In: *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, S. 45–54. ISBN: 0818631759.

[Mor13] John Paul Morrison. „Flow-based programming". In: *Journal of Application Developers' News*. 2013.

[Net04] Nicholas Nethercote. „Dynamic binary analysis and instrumentation". PhD Thesis. University of Cambridge, 2004.

[Neu10] Andreas Neustifter. „Efficient Profiling in the LLVM Compiler Infrastructure". Diplomarbeit. Karlsplatz 13, A-1040 Wien: Technische Universität Wien, 2010. URL: http://llvm.org/pubs/2010-04-NeustifterProfiling.html.

[NS73] I. Nassi und B. Shneiderman. „Flowchart Techniques for Structured Programming". In: *SIGPLAN Not.* 8.8 (1973-08), S. 12–26. ISSN: 0362-1340.

[OO84] Karl J. Ottenstein und Linda M. Ottenstein. „The Program Dependence Graph in a Software Development Environment“. In: *SIGPLAN Not.* 19.5 (1984-04), S. 177–184. ISSN: 0362-1340.

[ORS+05] G. Ottoni, R. Rangan, A. Stoler und D.I. August. „Automatic thread extraction with decoupled software pipelining“. In: *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*. 2005-11, 12 ff.

[PC90] Andy Podgurski und Lori A. Clarke. „A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance“. In: *IEEE Trans. Software Eng.* 16.9 (1990), S. 965–979.

[PCJ96] Helen C. Purchase, Robert F. Cohen und Murray James. „Validating Graph Drawing Aesthetics“. In: *Proceedings of the Symposium on Graph Drawing*. GD '95. London, UK, UK: Springer-Verlag, 1996, S. 435–446. ISBN: 3540607234.

[PGV08] Graham D. Price, John Giacomoni und Manish Vachharajani. „Visualizing Potential Parallelism in Sequential Programs“. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, S. 82–90. ISBN: 9781605582825.

[PJP+15] S. Prema, R. Jehadeesan, B.K. Panigrahi und S.A.V. Satya Murty. „Dependency analysis and loop transformation characteristics of auto-parallelizers“. In: *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*. 2015-02, S. 1–6.

[Pro15] LLVM Project, Hrsg. *LLVM Language Reference Manual — LLVM 3.7 documentation*. "[Online; Stand: 2015-04-14 16:26 Uhr]". 2015. URL: http://llvm.org/docs/LangRef.html#instruction-reference.

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee u. a. „A New Foundation for Control-Dependence and Slicing for Modern Program Structures“. In: *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Hrsg. von Shmuel Sagiv. Bd. 3444. Lecture Notes in Computer Science. Springer, 2005, S. 77–93. ISBN: 3540254358.

[Rhi01] Arjen van Rhijn. „Morpheus: A tool for the generation of schematics of extracted transistor level circuits with parasitic components“. Magisterarb. Delft: University of Technology, 2001.

[ROR+08] Easwaran Raman, Guilherme Ottoni, Arun Raman u. a. „Parallel-stage decoupled software pipelining“. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2008, S. 114–123.

[RQS12] Chris Rupp, Stefan Queins und die SOPHISTen. *UML 2 glasklar - Praxiswissen für die UML-Modellierung*. 4., aktualisierte und erweiterte Auflage. M: Carl Hanser Verlag GmbH Co KG, 2012. ISBN: 9783446431973.

[RR12] Thomas Rauber und Gudula Rünger. *Parallele Programmierung*. 3. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2012. ISBN: 9783642136047.

[RRB13] Sk. Riazur Raheman, Amiya Kumar Rath und M Hima Bindu. „An Overview of Program Slicing and its Different Approaches". In: *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)* 3 (11 2013-11). ISSN: 2277 128X.

[RS89] Jagannathan Ramanujam und Ponnuswamy Sadayappan. „A methodology for parallelizing programs for multicomputers and complex memory multiprocessors". In: *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM. 1989, S. 637–646.

[RVD07] Sean Rul, Hans Vandierendonck und Koen De Bosschere. „Function Level Parallelism Driven by Data Dependencies". In: *SIGARCH Comput. Archit. News* 35.1 (2007-03), S. 55–62. ISSN: 0163-5964.

[RVD10] Sean Rul, Hans Vandierendonck und Koen De Bosschere. „A profile-based tool for finding pipeline parallelism in sequential programs". In: *Parallel Computing* 36.9 (2010), S. 531–551.

[RVV+04] Ram Rangan, Neil Vachharajani, Manish Vachharajani und David I. August. „Decoupled Software Pipelining with the Synchronization Array". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, S. 177–188. ISBN: 0769522297.

[Sar89] Vivek Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. 1. Aufl. London: Pitman [u.a.], 1989. ISBN: 0273088025.

[Sco58] A. E. Scott. „Automatic Preparation of Flow Chart Listings". In: *J. ACM* 5.1 (1958-01), S. 57–66. ISSN: 0004-5411.

[SHZ+12] Kevin Streit, Clemens Hammacher, Andreas Zeller und Sebastian Hack. „Sambamba: A runtime system for online adaptive parallelization". In: *Compiler Construction*. Springer. 2012, S. 240–243.

[Tan06] Andrew S. Tanenbaum. *Computerarchitektur - Strukturen, Konzepte, Grundlagen*. 5. überarbeitete Auflage. München: Pearson Studium, 2006. ISBN: 9783827371515.

[TCA07] William Thies, Vikram Chandrasekhar und Saman Amarasinghe. „A practical approach to exploiting coarse-grained pipeline parallelism in c programs". In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE. 2007, S. 356–369.

[TF10] Georgios Tournavitis und Björn Franke. „Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information“. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM. 2010, S. 377–388.

[TH07] Jürgen Teich und Christian Haubelt. *Digitale Hardware/Software-Systeme - Synthese und Optimierung*. 2., erweiterte Auflage. Berlin Heidelberg New York: Springer-Verlag, 2007. ISBN: 9783540468240.

[Tip94] Frank Tip. *A Survey of Program Slicing Techniques.* Techn. Ber. Amsterdam, The Netherlands, The Netherlands, 1994.

[Tit03] Peter Tittmann. *Graphentheorie - eine anwendungsorientierte Einführung ; mit zahlreichen Beispielen und 80 Aufgaben*. 1. Auflage. München; Wien: Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2003. ISBN: 9783446223431.

[TKA02] William Thies, Michal Karczmarek und Saman Amarasinghe. „StreamIt: A Language for Streaming Applications“. In: *International Conference on Compiler Construction*. Grenoble, France, 2002-04. URL: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf.

[TWF+09] Georgios Tournavitis, Zheng Wang, Björn Franke und Michael FP O'Boyle. „Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping“. In: *ACM Sigplan Notices*. Bd. 44. 6. ACM. 2009, S. 177–187.

[ULS+08] Jeffrey D. Ullman, Monica S. Lam, Ravi Sethi und Alfred V. Aho. *Compiler - Prinzipien, Techniken und Werkzeuge*. 2. Auflage. München: Pearson Deutschland GmbH, 2008. ISBN: 9783827370976.

[VRD10] Hans Vandierendonck, Sean Rul und Koen De Bosschere. „The Paralax infrastructure: automatic parallelization with a helping hand“. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM. 2010, S. 389–400.

[Wal91] David W. Wall. „Limits of Instruction-level Parallelism“. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, S. 176–188. ISBN: 0897913809.

[Wei79] Mark David Weiser. „Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method“. Diss. Ann Arbor, MI, USA, 1979.

[WFW+94] Robert P Wilson, Robert S French, Christopher S Wilson u. a. „SUIF: An infrastructure for research on parallelizing and optimizing compilers“. In: *ACM Sigplan Notices* 29.12 (1994), S. 31–37.

[Win03] Sabine Winkler. *Cover your world! – Codecoverage mit Clover*. 2003-04. URL: http : / / www . oio . de / testcoverage - clover . htm (besucht am 17. 02. 2015).

[Wir98] N. Wirth. „Hardware compilation: translating programs into circuits“. In: *Computer* 31.6 (1998-06), S. 25–31. ISSN: 0018-9162.

[WPC+02] Colin Ware, Helen Purchase, Linda Colpoys und Matthew McGill. „Cognitive Measurements of Graph Aesthetics“. In: *Information Visualization* 1.2 (2002-06), S. 103–110. ISSN: 1473-8716.

[WTF+14] Zheng Wang, Georgios Tournavitis, Björn Franke und Michael F. P. O'boyle. „Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping“. In: *ACM Trans. Archit. Code Optim.* 11.1 (2014-02), 2:1–2:26. ISSN: 1544-3566.

[WWB+08] Cheng Wang, Youfeng Wu, Edson Borin u. a. „New Slicing Algorithms for Parallelizing Single-Threaded Programs“. In: *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA)*. 2008.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang u. a. „A Brief Survey of Program Slicing“. In: *SIGSOFT Softw. Eng. Notes* 30.2 (2005-03), S. 1–36. ISSN: 0163-5948.

[YC12] J.M. Ye und Tianzhou Chen. „Exploring Potential Parallelism of Sequential Programs with Superblock Reordering“. In: *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. 2012-06, S. 9–16.

Betreute Abschlussarbeiten

[Eis14] Jan-Magnus Eisenhuth. „Dynamische Visualisierung von Laufzeitprofilen komplexer Softwaresysteme“. Bachelor-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2014.

[Fie10] Thomas Fiedler. „Entwurf, Implementierung und Charakterisierung einer Kommunikation über PCI-Express für ein Detektorauslese- und -kalibrationssystem“. Bachelor-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2010.

[Fle11] Tim Fleschenberg. „Anbindung eines FPGA-Entwicklungsboards über PCIe an ein Linux-System“. Bachelor-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2011.

[Kob10] Katrin Kobas. „Rekonfigurierbarkeit von Field-Programmable-Gate-Arrays (FPGAs)“. Bachelor-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2010.

[Sal14] Mohamed Saleh-Attia. „Statistische Auswertung von Laufzeitprofilen komplexer Softwaresysteme“. Bachelor-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2014.

[Sch14] Martin Schröder. „Visualisierung von Kontroll- und Datenflussabhängigkeiten in komplexen Softwaresystemen“. Master-Thesis. Rainer-Gruenter-Str. 21, 42119 Wuppertal: Bergische Universität Wuppertal, 2014.

Eigene Publikationen

[PBG+15] Nils Potthoff, Christoph Brandau, Carsten Gremzow und Dietmar Tutsch. „Data and Control Flow Visualization by Transforming Software into Schematic Diagrams“. In: *The 15th Middle Eastern Simulation and Modelling Multiconference (MESM) - The 5th GAMEON-ARABIA Conference*. Best-Paper-Award. Eurosis-ETI, 2015, S. 20–27. ISBN: 9789077381878.

[PGB+15a] Nils Potthoff, Carsten Gremzow, Christoph Brandau und Dietmar Tutsch. „Function Slices: A Model to Extract Parallelism from Sequential Applications“. In: *ESM'2015 - The 2015 European Simulation and Modeling Conference*. Eurosis-ETI, 2015.

[PGB+15b] Nils Potthoff, Carsten Gremzow, Christoph Brandau und Dietmar Tutsch. „Performance Increase by Software Decomposition with Characteristics of Combinational Logic“. In: *Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3GPCIC) 2015*. 2015.

[PGB+16] Nils Potthoff, Carsten Gremzow, Christoph Brandau und Dietmar Tutsch. „A New Approach to Mapping Software to Coprocessor Circuits“. In: *2016 IEEE International Conference on Consumer Electronics*. IEEE. Las Vegas, USA: IEEE, 2016.