Fakultät für Informatik und Mathematik
Lehrstuhl für Programmierung

Analyse des Parallelisierungspotentials sequenzieller Programme durch Kombination von statischer und dynamischer Analyse

Bachelorarbeit

Autor:
Andreas Johannes Wilhelm
August 2011

Aufgabensteller:
Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung, Universität Passau

Betreuer:
Dr. Armin Größlinger, Universität Passau
Dr. Tobias Schüle, Siemens AG

Andreas Johannes Wilhelm:
Analyse des Parallelisierungspotentials sequenzieller Programme durch Kombination von statischer und dynamischer Analyse, Bachelorarbeit, Universität Passau, 2011

Zusammenfassung

Der Großteil bestehender Software ist sequenziell, weshalb die Vorteile aktueller Multicore-Plattformen oft ungenutzt bleiben. Eine manuelle Parallelisierung ist jedoch in vielen Fällen sehr aufwändig und die zu erwartenden Leistungssteigerungen schwer vorhersagbar.

In dieser Arbeit wird ein Verfahren zur Ermittlung von Parallelisierungspotential sequenzieller Programme vorgestellt. Durch den Einsatz verschiedener Analysen und Heuristiken werden günstige Stellen für grobgranulare Parallelität ermittelt. Die Ergebnisse unterstützen Softwareentwickler bei der Parallelisierung bestehender Anwendungen. Eine Besonderheit des gewählten Ansatzes ist die Kombination aus dynamischer Ermittlung von Ablaufinformationen und interprozeduraler Abhängigkeitsanalyse.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Der Einzug von Mehrkernprozessoren in alle Arten von Computersystemen stellt die Software-Entwicklung vor große Herausforderungen. Ein Großteil bestehender Software ist sequenziell und konnte bisher regelmäßig von steigenden Taktfrequenzen profitieren. Aufgrund physikalischer Grenzen stagnieren allerdings die Taktfrequenzen seit ca. 5 Jahren. Hardwarehersteller nutzen die weiterhin steigende Anzahl an Transistoren je Chip aus, um mehrere Rechenkerne in die Prozessoren zu integrieren. Die Laufzeit sequenzieller Software wird dadurch nicht verbessert. Um dies zu ändern, müssen die Anwendungen parallelisiert werden, was in den meisten Fällen ein sehr aufwändiger Vorgang ist.

Es bestehen grundsätzlich zwei Möglichkeiten für die Parallelisierung von Software. Einerseits können Entwickler durch explizite Anweisungen im Quelltext bestimmen, welche Teile eines Programms parallel ausgeführt werden sollen. Durch den Einsatz von Synchronisationsmechanismen lassen sich parallele Abläufe sehr genau koordinieren. Im Allgemeinen ist es jedoch schwierig, die passenden Stellen für Parallelität innerhalb einer bestehenden Anwendung zu ermitteln. Außerdem ist im Voraus nicht bekannt, wie hoch der Aufwand für die Parallelisierung ist und welcher Laufzeitgewinn erwartet werden kann.

Die Alternative dazu ist eine implizite Parallelisierung, die diese Aufgabe dem verwendeten Compiler überträgt. Verschiedene Analysen sollen sichere Möglichkeiten für Parallelität entdecken. Der Programmierer schreibt weiterhin sequenziellen Code, den der Compiler automatisch in parallel ausführbare Software übersetzt. Auch wenn diese Variante sehr komfortabel erscheint, so liefern die verwendeten Analysen oftmals nur wenig Möglichkeiten für Parallelität. Für sichere Aussagen müssen die Compiler konservativ vorgehen, d. h. sie dürfen bei unsicheren Aussagen nicht von möglicher Parallelität ausgehen. Gerade im Falle irregulärer Anwendungen führt implizite Parallelisierung deshalb zu unbefriedigenden Ergebnissen.

In dieser Arbeit wird ein Verfahren beschrieben, welches die Vorteile der impliziten und expliziten Parallelisierung vereint. Durch statische sowie dynamische Analysen werden dem Benutzer zielgerichtet Möglichkeiten für Parallelität in sequenziellen Anwendungen aufgezeigt. Das Ergebnis ist eine Abschätzung des potentiellen Laufzeitgewinns durch parallele Ausführung mit Berücksichtigung des dafür benötigten Aufwands. Dadurch kann mit Hilfe von Sprachkonstrukten bestehender Programmierschnittstellen, wie OpenMP, eine manuelle Parallelisierung durchgeführt werden.

Das Verfahren beschränkt sich hierbei auf grobgranulare Tasks in Form von Prozeduren, in denen gewöhnlich eine große Anzahl an Instruktionen verarbeitet werden. Mehrkernprozessoren profitieren von solchen Tasks mehr als von feingranularer Parallelität, da weniger Kommunikations- und Synchronisationsaufwand notwendig ist. Ein weiterer Vorteil besteht darin, dass die Struktur der Benutzerprogramme nur wenig verändert werden muss. Eine andere Alternative zur Aufteilung in grobgranulare Tasks ist die Parallelisierung von Schleifen. Deren Ausführungszeiten machen häufig einen großen Anteil der Programmlaufzeit aus. Gerade im Bereich von parallelisierenden Compilern konnten bereits große Fortschritte erreicht werden [Ban94, FL11]. Im Rahmen dieser Arbeit werden Schleifen nicht explizit untersucht. Das Ziel ist es, die Relevanz von Task-Parallelität auf Prozedurebene sowie die Nützlichkeit der verwendeten Analysen zu bewerten.

1.1 Statische und dynamische Analysen

Ein besonderes Merkmal dieser Arbeit besteht in der Art und Weise der Ermittlung von Parallelisierungspotential. Es werden sowohl statische als auch dynamische Analysemethoden eingesetzt. Bei statischen Analysen erfolgt eine formale Untersuchung des Quelltextes von Programmen während der Übersetzung. Es gibt viele Einsatzmöglichkeiten für solche Analysen, wie Syntaxprüfungen, Kontroll- oder Datenflussanalysen. Die Resultate sind sehr detailliert und für sämtliche Aufrufkontexte gültig. Damit sind die Analysen für Optimierungen von Programmen gut geeignet. Allerdings ist es im Allgemeinen nicht möglich, für alle Situationen exakte Aussagen zu treffen [Lan92]. Mögliche Ursachen sind die Verwendung von Zeigern, rekursive Datenstrukturen oder späte Bindung innerhalb von Programmen.

Dynamische Analysen werden vor allem in Profiling-Werkzeugen benutzt. Die Auswertungen erfolgen hier zur Laufzeit der Programme. In den meisten Fällen werden dazu an bestimmten stellen Instruktionen in den Quelltext eingefügt. Während der Ausführung können dadurch Informationen an die Analyseanwendung weitergeben werden. Bei der dynamischen Analyse entspricht das Ergebnis dem, was tatsächlich während der Ausführung aufgetreten ist. Es muss allerdings beachtet werden, dass die Resultate hierbei nicht allgemeingültig sind. Jeder Ablauf kann aufgrund abweichender Eingabedaten unterschiedlich sein.

Das in dieser Arbeit beschriebene Verfahren extrahiert zunächst mit Hilfe von dynamischer Analyse notwendige Ablaufinformationen der Benutzerprogramme. Diese Informationen werden zur Bestimmung von Ausführungszeiten und Aufrufstellen der Prozeduren benötigt. Der Rest der

Auswertung basiert ausschließlich auf statischen Analysetechniken.

Bestehende Abhängigkeiten zwischen Prozeduren können einer möglichen Parallelisierung im Wege stehen, indem sie eine bestimmte Ausführungsreihenfolge erzwingen. Die Herausforderung bei der Suche nach Parallelisierungspotential besteht somit in der Bestimmung von Abhängigkeiten zwischen verschiedenen Prozeduraufrufen. Eine rein statische Abhängigkeitsanalyse kann allerdings, wie vorhin erwähnt, in manchen Fällen keine exakten Aussagen treffen. In diesen Situationen zeigt der verwendete Ansatz ein spekulatives Verhalten. Es wird also davon ausgegangen, dass nur diejenigen Abhängigkeiten existieren, welche durch die Analysen gefunden wurden. Der Benutzer hat somit nach erfolgter Parallelisierung die Aufgabe, die Gültigkeit der ermittelten Lösung zu überprüfen.

1.2 Bestehende Ansätze

Es existieren viele bestehende Arbeiten zur Parallelisierung von sequenziellen Programmen. Parallelisierende Compiler, wie der SUIF Compiler der Universität Stanford, führen ebenfalls statische Abhängigkeitsanalysen durch [Ama95]. Da diese aber auf einer feingranularen Ebene von Instruktionen arbeiten, sind sie ebenso wenig mit diesem Ansatz vergleichbar wie spekulative Ausführungsumgebungen. Hierbei werden Ergebnisse von Abhängigkeitsanalysen verwendet, um parallele Tasks zu bilden und diese im Anschluss auszuführen. Treten während der Laufzeit dennoch Zugriffskonflikte auf, so nimmt das System die bereits ausgeführten Operationen zurück und wiederholt den Ablauf in sequenzieller Reihenfolge. Die Ausführungsumgebungen können sowohl durch Hardware als auch durch Software bereit gestellt werden [SCZM05, OM08].

Mit den kommerziell verfügbaren Werkzeugen Intel Parallel Advisor und Prism von CriticalBlue existieren zwei Ansätze, die dem Verfahren dieser Arbeit sehr nahe kommen [Int11, Cri11]. Beide bieten dem Benutzer eine Auswertung von Parallelisierungspotential bezüglich grobkörniger Parallelität auf Prozedurebene. Bei der Verwendung des Parallel Advisors können mit Hilfe von Profiling-Informationen sogenannte “Hotspots”, also Stellen mit einem großen Anteil an der Ausführungszeit, identifiziert werden. Durch manuelle Annotationen im Quelltext hat der Benutzer die Möglichkeit, beliebige Bereiche für eine parallele Ausführung vorzumerken. Anschließend berechnet die Anwendung mit Hilfe der protokollierten Ablaufinformationen die resultierende Laufzeiteinsparung sowie entstehende Konflikte.

Bei Prism von CriticalBlue wird ebenfalls das zu analysierende Programm vorab instrumentiert und ausgeführt. Der Hauptunterschied zum Parallel Advisor besteht in der Auswahl von Tasks. Anstatt Annotationen im Quelltext anzugeben, können einzelne Prozeduren zur parallelen Ausführung ausgewählt werden. Anschließend kann ein Scheduler die resultierenden Ausführungszeiten ermitteln. Falls bei der Abarbeitung Abhängigkeiten entdeckt werden, muss der Benutzer entscheiden, ob die sequenzielle Ausführungsreihenfolge verwendet werden soll oder die Konflikte ignoriert werden können. Damit besteht allerdings die Verpflichtung, in der späteren Implementierung die Konflikte tatsächlich zu beseitigen. Eine Besonderheit der Anwendung ist die Unterstützung der POSIX Thread Schnittstelle. Sollten die zu analysierende Anwendungen bereits parallel ablaufende Berechnungen enthalten, so berücksichtigt Prism dies bei der Auswertung.

Im Gegensatz zu dem Verfahren dieser Arbeit verwenden Prism und der Parallel Advisor dynamische Analysen zur Bestimmung von Abhängigkeiten zwischen Prozeduraufrufen. Dadurch sind die Resultate abhängig von der konkreten Ausführung. Eine parallele Umsetzung kann nur für die Ausführung mit den selben Eingabedaten der Analyse als sicher eingestuft werden. Zusätzlich ist der Benutzer bei der Verwendung beider Werkzeuge verpflichtet, die zu untersuchenden Aufrufstellen selbst festzulegen. Die Abhängigkeiten werden nur auf diesen festgelegten Stellen analysiert. Das Verfahren dieser Arbeit identifiziert hingegen sämtliche Aufrufe der Benutzeranwendung und bewertet sie automatisch hinsichtlich Parallelisierungspotential.

1.3 Überblick

Im folgenden Kapitel werden wichtige Grundlagen erläutert, welche innerhalb der Arbeit Verwendung finden. Eine wichtige Aufgabe besteht in der Entwicklung eines Analysewerkzeugs zur Umsetzung der vorgestellten Konzepte. Diese werden in Kapitel 3 präsentiert. Anschließend erfolgt eine Übersicht über die Schwerpunkte der Implementierung, welche mit Hilfe des LLVM-Frameworks (*Low Level Virtual Machine*) realisiert wurde. Die Analysen werden anhand eines Mergesort-Algorithmus sowie mehrerer Testprogramme der SPEC2006-Benchmarks geprüft. Die Resultate sind in Kapitel 5 aufgeführt. Im letzten Kapitel wird schließlich eine Zusammenfassung der wichtigsten Erkenntnisse sowie ein Ausblick auf mögliche Erweiterungen gegeben.

KAPITEL 2

Grundlagen

Für das in dieser Arbeit beschriebene Verfahren zur Analyse von Parallelisierungspotential werden verschiedene Techniken und Modelle aus dem Compilerbau verwendet. Die entsprechenden Konzepte stammen dabei zum Großteil aus dem Bereich maschinenunabhängiger Optimierung und automatischer Parallelisierung. Die Analyseaufgaben können durch die Verwendung allgemein akzeptierter Lösungsverfahren effizient bearbeitet werden. Zudem existieren geeignete Repräsentationen der Problemstellungen, was zu einer besseren Übersichtlichkeit führt.

Die Auswahl der Themen in den folgenden Abschnitten wurde bezüglich der Relevanz der hier beschriebenen Arbeit gewählt. Zu Beginn der Abschnitte werden nicht erläuterte Begriffe explizit erwähnt und können bei Bedarf in der Literatur nachgelesen werden [ASU86, Wol95, Muc97].

2.1 Aufrufgraph

Der Aufrufgraph einer Anwendung beschreibt die Aufrufhierarchie der enthaltenen Prozeduren. Ein solcher Graph besteht aus einer Menge von Knoten und Kanten, für die folgendes gilt:

1. Zu jeder Prozedur des Programms gibt es einen Knoten.
2. Es gibt einen Knoten zu jeder Aufrufstelle, d.h. eine Stelle im Programm, an der eine Prozedur aufgerufen wird.
3. Wenn eine Aufrufstelle c eine Prozedur p aufrufen kann, gibt es eine Kante Knoten c zum Knoten p.

Bei der Definition fällt zunächst auf, dass mehrere Kanten vom Knoten für eine Prozedur q zum Knoten für eine Prozedur p vorhanden sein können. Dies ist genau dann der Fall, wenn q

mehrere Aufrufstellen besitzt, welche p aufrufen können. Somit handelt es sich bei einem Aufrufgraph um einen Multigraph. Diese Tatsache spielt bei der Ermittlung von Abhängigkeiten eine wichtige Rolle (siehe Abschnitt 3.5).

In Programmiersprachen wie C und C++ ist es möglich, Prozeduren mit Hilfe von Funktionszeigern oder durch dynamische Bindung aufzurufen. Die Aufrufziele können dabei im Allgemeinen erst zur Laufzeit bestimmt werden. Einfache statische Analysen müssen bei jeder Aufrufstelle davon ausgehen, dass alle erreichbaren Prozeduren für den Aufruf in Frage kommen. Jede Möglichkeit entspricht dabei einer zusätzlichen Kante im Aufrufgraph. Durch interprozedurale Analyse (siehe Abschnitt 2.5) lässt sich eine Annäherung an einen konkreten Aufrufgraphen erreichen. Jedoch ist es auch dadurch nicht möglich externe Prozeduraufrufe aufzulösen. Ein konservativer Ansatz führt somit auch hier zu einer Überapproximation von möglichen Aufrufzielen. Das bedeutet, dass mehr Kanten in dem Aufrufgraph auftreten können, als tatsächlich vorhanden sind.

Eine Alternative zur statischen Berechnung von Aufrufgraphen besteht in der Ermittlung der Knoten und Kanten zur Laufzeit des Programms. Jede aufgerufene Prozedur wird als Knoten im Graph vermerkt. Die Kanten ergeben sich durch die Aufrufreihenfolge. Zusätzlich ist es möglich, Informationen bezüglich der Ausführung (Anzahl Aufrufe, Ausführungszeiten, usw.) in den Aufrufgraph einfließen zu lassen. Ein dynamischer Aufrufgraph ist im Allgemeinen nicht mit einem statisch ermittelten identisch. Es kann beispielsweise bei der dynamischen Ausführung passieren, dass bestimmte Prozeduren gar nicht aufgerufen werden. Damit ist die Prozedur nicht im dynamischen Aufrufgraph vorhanden.

In dieser Arbeit spielen dynamische Aufrufgraphen eine entscheidende Rolle bei der Analyse des Parallelisierungspotentials. Anhand der Aufrufhierarchie werden Paare von Prozeduren ausgewählt, welche für die Bewertung in Frage kommen. Außerdem enthält der Aufrufgraph Informationen über den zeitlichen Ablauf der zu untersuchenden Anwendung. Diese Informationen werden ebenfalls für die Auswertung des Potentials benötigt.

2.2 Def-Use- und Use-Def-Ketten

Im folgenden Abschnitt wird vorausgesetzt, dass die Begriffe Datenflussanalysen und Datenflussgraphen bekannt sind.

Compiler benötigen häufig präzise Informationen über die Verwendung von Variablenwerten an bestimmten Punkten des Programms. Es ist vor allem bei Abhängigkeitsanalysen wichtig zu wissen, welche Verwendungen von Variablen Abhängigkeiten erzeugen. Sogenannte *Def-Use*-Ketten verbinden Definitionen (*Def*) einer Variablen mit sämtlichen Verwendungen (*Use*), die über den Datenflussgraph erreichbar sind. *Use-Def*-Ketten verbinden Verwendungen von Variablen mit den vorausgehenden Definitionen.

Listing 2.1 soll das Konzept von *Def-Use*- und *Use-Def*-Ketten deutlich machen. Bei Ersteren lassen sich fünf Definitionen betrachten. Interessant sind hierbei diejenigen in den Zeilen 2 und 4. Aufgrund der zweiten `if-else`-Anweisung erreicht die Definition aus Zeile 4 (im Gegensatz zu

der aus Zeile 2) nicht die Verwendung in Zeile 7. Dieser Effekt tritt auch bei der *Use-Def*-Kette durch die Verwendung von x in Zeile 7 auf. Die zugehörige Definition kann hier im Beispiel lediglich die von Zeile 2 sein.

```
1  if (z > 1)       // Use: z
2     x = 1;        // Def: x
3  else
4     x = 2;        // Def: x
5
6  if (z > 2)       // Use: z
7     y = x + 1;    // Def: y   Use: x
8  else
9     y = x - 3;    // Def: y   Use: x
10
11 z = x + y;       // Def: z   Use: x, y
```

Listing 2.1: *Def-Use*- und *Use-Def*-Ketten

In Abbildung 2.1 sind die *Def-Use*-Ketten zu dem Beispiel eingetragen. Die gestrichelten Linien zeigen jeweils von einer Definition zu den Verwendungen. Jede Linie besitzt den Namen der definierten Variable als Beschriftung. Die entsprechenden *Use-Def*-Ketten sind bis auf die Richtung der Pfeile identisch.

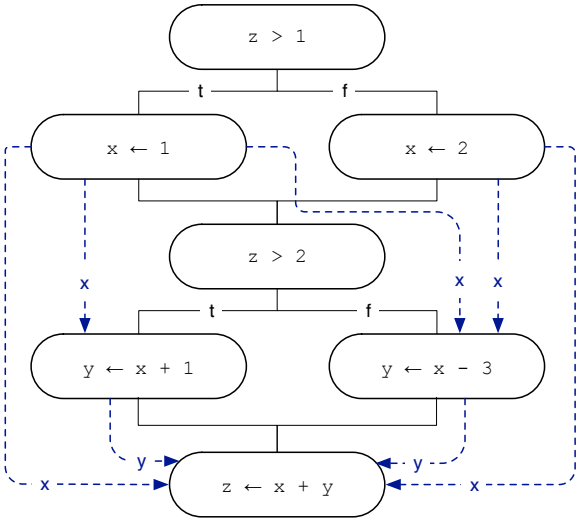

Abbildung 2.1: *Def-Use*-Ketten zu Listing 2.1

Def-Use-Ketten werden bei der Suche nach Abhängigkeiten zwischen Prozeduraufrufen verwendet. Durch sie wird ein möglicher Datenfluss zwischen den Aufrufstellen erkannt. Falls beispielsweise der Rückgabewert einer Prozedur eine Variable definiert, welche anschließend an eine andere Prozedur als Argument übergeben wird, so kann eine Datenabhängigkeit existieren (siehe Abschnitt 2.4).

2.3 Dominatoren und Postdominatoren

Im folgenden Abschnitt wird vorausgesetzt, dass die Begriffe Kontrollflussgraph und Basisblock bekannt sind. Basisblöcke sind hierbei die Knoten von Kontrollflussgraphen. Die Kanten entsprechen den Kontrollflusspfaden zwischen Basisblöcken.

Definition 1. *Ein Knoten p dominiert einen anderen Knoten q, falls jeder Pfad im Flussgraph vom Eingangsknoten hin zu q durch den Knoten p verläuft.*

In der Literatur wird eine solche Beziehung als p *dom* q notiert. Falls p *dom* q, so kann jeder Pfad $s : start \rightarrow q$ in die beiden Pfade $s_{prefix} : start \rightarrow p$ und $s_{suffix} : p \rightarrow q$ aufgeteilt werden. Da ein Pfad vom Eingangsknoten zu einem Knoten q immer durch q selbst führt, gilt stets q *dom* q.

Dominatorinformationen werden häufig in Form von Bäumen (Dominatorbaum) dargestellt. Die Möglichkeit ergibt sich aus der Eigenschaft, dass jeder Knoten einen unmittelbaren Dominator besitzt. Dieser ist der letzte Dominator des Basisblocks auf einem beliebigen Pfad beginnend beim Eingangsknoten.

In Abbildung 2.2 ist ein Flussgraph einer Prozedur gezeigt. Es existiert eine *if-else*-Struktur am Basisblock a, sowie zwei Schleifen. Der entsprechende Dominatorbaum in Abbildung 2.3 besitzt als Wurzelknoten den Knoten a. Dieser dominiert sämtliche anderen Knoten. Das bedeutet, dass der entsprechende Basisblock stets am Anfang ausgeführt wird. Man beachte den Basisblock h, welcher zwar am Ende der Aufrufhierarchie steht, aufgrund der unterschiedlichen Pfade über f und g jedoch nur von Knoten a dominiert wird.

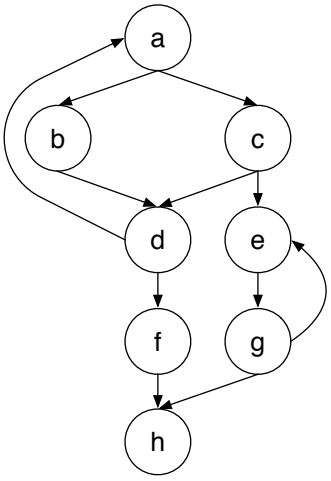

Abbildung 2.2: Beispiel für einen Flussgraph

Damit die Definition von Dominatoren angewendet werden kann, wird ein eindeutiger Einstiegsknoten eines Flussgraph vorausgesetzt. Ein ähnliches Konzept, welches in dieser Arbeit benötigt wird, ist die Postdominanz. Es besitzt eine ähnliche Voraussetzung, der Flussgraph muss allerdings in diesem Fall genau einen Ausgangsknoten enthalten.

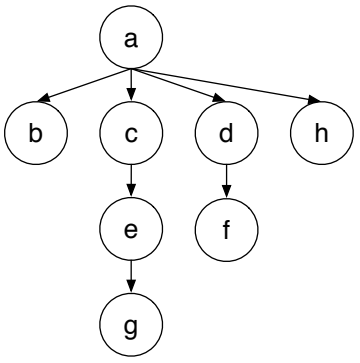

Abbildung 2.3: Dominatorbaum zu Flussgraph aus Abbildung 2.2

Definition 2. *Ein Knoten q eines Flussgraph postdominiert einen Knoten p des selben Graph, wenn jeder Pfad von p zum Ausgang über q führt.*

Die Notation für Postdominanz ist *q pdom p*. Folgende Definition ist somit ebenfalls gültig: Falls *q pdom p*, so kann ein Pfad $p \rightarrow exit$ in die beiden Pfade $p \rightarrow q$ und $q \rightarrow exit$ aufgeteilt werden. Jeder Basisblock postdominiert sich selbst.

Auch Postdominatoren können in Form von Bäumen dargestellt werden, da jeder Knoten von exakt einem Knoten postdominiert wird. Abbildung 2.4 zeigt den Postdominatorbaum zu dem Flussgraph in Abbildung 2.2. Sämtliche Knoten werden von h postdominiert, da dies der Ausgangsknoten des Flussgraphen ist. Nur Knoten g postdominiert zusätzlich einen weiteren Basisblock. Im Fall der Knoten b und d ist die Rückwärtskante von d nach a im Flussgraph die Ursache dafür, dass sie keinen weiteren Knoten postdominieren. Es existieren deshalb mehrere Pfade zum Ausgangsknoten (z. B. $b \rightarrow d \rightarrow f \rightarrow h$ sowie $b \rightarrow d \rightarrow a \rightarrow c \rightarrow e \rightarrow g \rightarrow h$).

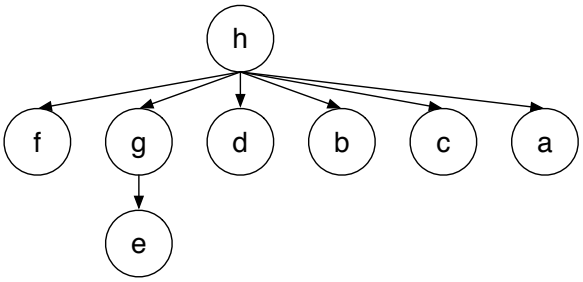

Abbildung 2.4: Postdominatorbaum zu Flussgraph aus Abbildung 2.2

Dominator- und Postdominator-Informationen werden bei der Bewertung des Parallelisierungspotentials eingesetzt. Falls ein Prozeduraufruf a einen anderen Prozeduraufruf b dominiert, so ist sichergestellt, dass a vor b ausgeführt werden muss. Falls zusätzlich b den Aufruf a postdominiert, dann werden die Prozeduraufrufe stets gemeinsam ausgeführt. Die Analyse bewertet in dem Fall das Potential der Aufrufstellen höher, da eine parallele Ausführung wahrscheinlicher ist als bei Aufrufstellen in unterschiedlichen Kontrollpfaden.

2.4 Formen von Abhängigkeiten

Die Ermittlung von Abhängigkeiten zwischen einzelnen Instruktionen oder grobgranularen Operationen (z. B. Prozeduren) in einem Programm ist essenziell für die Parallelisierung. Es existieren zwei Klassen von Abhängigkeiten: Kontrollflussabhängigkeiten (Kontrollabhängigkeiten) und Datenabhängigkeiten.

Kontrollabhängigkeiten Es besteht eine Kontrollabhängigkeit zwischen zwei Ausdrücken a_1 und a_2, falls das Ergebnis von a_1 Einfluss auf die Ausführungshäufigkeit von a_2 hat. Eine solche Art von Abhängigkeit tritt häufig bei der Verwendung von Kontrollstrukturen auf. In der `if-else`-Anweisung

```
if (c) then a else b
```

sind `a` und `b` kontrollabhängig von `c`. Kontrollabhängigkeiten können auch bei Schleifen entstehen, wobei der Schleifenrumpf von der Schleifenbedingungen abhängig ist:

```
while(c) do a
```

In dem gezeigten Beispiel ist Anweisung `a` von `c` kontrollabhängig.

Datenabhängigkeiten Datenabhängigkeiten entstehen, falls mehrere Operationen auf gemeinsame Daten zugreifen. Kommen die Operationen ohne gemeinsame Zugriffe aus, so kann die Aufrufreihenfolge beliebig sein. Das ist auch dann der Fall, wenn ausschließlich Lesevorgänge ausgeführt werden. Eine ganz andere Situation entsteht, falls mindestens eine Instruktion Daten ändert, welche von anderen Operationen verwendet werden. Hier kann eine andere Reihenfolge der Operationen zu unterschiedlichen Ergebnissen führen. Es existieren drei Arten solcher Datenabhängigkeiten:

1. Eine **echte Abhängigkeit** *(True Dependence)* besteht, falls Daten nach einer Schreiboperation gelesen werden. Der Lesevorgang hängt von der vorherigen Modifikation ab und kann daher nicht ohne weiteres parallel zu dieser ausgeführt werden.

2. Die **umgekehrte Abhängigkeit** *(Anti Dependence)* beschreibt eine Schreiboperation nachdem die zugehörigen Daten gelesen wurden. Auch hier kann eine parallele Ausführung der betroffenen Operationen andere Ergebnisse als die sequenzielle Reihenfolge liefern. Dabei ist es möglich, dass die Modifikation der Daten vor dem Lesevorgang geschieht, wodurch andere Werte gelesen werden. Eine solche Art von Datenabhängigkeit lässt sich häufig durch die Verwendung zusätzlicher Speicherbereiche umgehen.

3. Eine **Ausgabeabhängigkeit** *(Output Dependence)* tritt bei einem Paar von Schreiboperationen auf die selben Daten auf. Je nach Reihenfolge der Schreiboperationen können die Speicherbereiche verschiedene Werte enthalten. Auch bei dieser Abhängigkeit kann der Einsatz zusätzlicher Speicherbereiche eine Lösung bieten.

Abhängigkeitsgraph Eine übliche Repräsentation von Abhängigkeiten aus der Compilertechnik sind sogenannten Abhängigkeitsgraphen. In einem solchen Graphen stellen Operationen (In-

struktionen oder grobgranulare Codeabschnitte) die Knoten dar. Abhängigkeiten zwischen den Operationen werden als gerichtete Kanten in den Graphen eingetragen. Dabei erhält jede Kante eine Kennzeichnung mit dem entsprechenden Typ der Abhängigkeit. Durch den Codeabschnitt aus Listing 2.2 sollen die erwähnten Abhängigkeiten anhand eines Beispiels aufgezeigt werden.

```
1 a = b + c;
2 c = 0;
3 if (a <= 10) {
4     d = b * e;
5     e = d + 1;
6 }
7 e = 2;
```

Listing 2.2: Abhängigkeiten

Der Ausdruck in Zeile 1 liest den Wert von Variable *c*, welche in Zeile 2 überschrieben wird. Das erzeugt eine umgekehrte Abhängigkeit zwischen den beiden Instruktionen. Die Abfrage in Zeile 3 liest den Wert von Variable *a*. Diese wurde vorher in Zeile 1 definiert, sodass eine echte Abhängigkeit entsteht. Zwischen den Anweisungen in Zeile 3 und Zeile 4 besteht eine Kontrollabhängigkeit, da letztere nur bei wahrer Bedingung ausgeführt wird. Die Ausgabeabhängigkeit zwischen den Instruktionen aus Zeile 5 und Zeile 7 besteht aufgrund der Schreiboperationen bezüglich derselben Variable (e).

Der vollständige Abhängigkeitsgraph ist in Abbildung 2.5 zu sehen. Die Kantenbezeichnungen haben folgende Bedeutung: U = Umgekehrte Abhängigkeit, A = Ausgabeabhängigkeit, K = Kontrollabhängigkeit. Falls eine Kante keine Bezeichnung besitzt, so handelt es sich um eine echte Abhängigkeit.

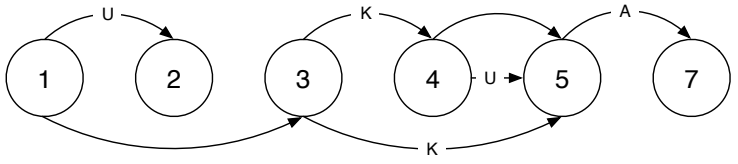

Abbildung 2.5: Abhängigkeitsgraph

Die verschiedenen Arten von Abhängigkeiten zwischen Prozeduraufrufstellen sind in der Analyseanwendung das wichtigste Kriterium zur Berechnung des Potentials von Parallelität. Für die Auswertung werden Abhängigkeitsgraphen gebildet. Durch Berücksichtigung der Transitivität werden damit Abhängigkeiten zwischen verschiedenen Prozeduraufrufstellen erkannt.

2.5 Interprozedurale Analyse

In diesem Abschnitt wird vorausgesetzt, dass der Begriff *stark zusammenhängende Komponenten* aus der Graphentheorie bekannt ist.

Definition 3. *Intraprozedurale Analyse beschreibt die Auswertung einer einzelnen Prozedur, ohne Beachtung von Aufrufkontexten.*

Gerade im Bereich von automatischer Parallelisierung auf Befehls- und Schleifenebene sind Abhängigkeitsanalysen ein unverzichtbarer Bestandteil. Viele bestehende Arbeiten zeigen die Möglichkeiten, die sich dadurch für parallelisierende Compiler ergeben [DRV00].

Hinsichtlich des Schwerpunkts dieser Arbeit, der grobgranularen Parallelisierung auf Prozedurebene, bestehen ähnliche Herausforderungen im Hinblick auf die Abhängigkeitsanalyse. Durch die Verwendung von Zeigern, Referenzen oder globalen Variablen muss eine int**ra**prozedurale Analyse von jedem Prozeduraufruf annehmen, dass dabei der Zustand aller für sie sichtbarer Variablen verändert wird. Aufrufziele von Funktionszeigern stellen eine zusätzliche Schwierigkeit dar. Auch hier können intraprozedurale Techniken häufig keine präzisen Aussagen treffen. Um eine möglichst genaue statische Analyse zu erreichen, muss int**er**prozedural vorgegangen werden.

Eine interprozedurale Analyse arbeitet über ein gesamtes Programm hinweg, wobei Informationen von Aufrufern zu den aufgerufenen Prozeduren und zurück fließen. Im Hinblick auf die Suche nach Abhängigkeiten zwischen Prozeduren ist vor allem die interprozedurale Zeigeranalyse wichtig. Informationen über Ziele von Zeigern und Werte von formalen Parametern oder globalen Variablen fließen hin zur aufgerufenen Prozedur. Innerhalb dieser können die auftretenden Abhängigkeiten nun genauer bestimmt werden. Dieser Schritt heißt *Top-Down*-Phase. Die Auswirkungen von Prozeduren, wie eine Wertänderung von Referenzparametern, werden im Gegenzug an die aufrufenden Stellen zurück propagiert. Daher heißt dieser Schritt *Bottom-Up*-Phase. Das Verfahren kann zwar für exakte Auswertungen sehr aufwändig sein, die Abhängigkeiten werden aber um einiges genauer bestimmt als bei der rein intraprozeduralen Analyse [Muc97].

In dieser Arbeit wird interprozedurale Analyse als Teil der Abhängigkeitsanalyse eingesetzt. Es können damit die Auswirkungen von Prozeduraufrufen ermittelt werden, um mögliche Parallelität zu entdecken.

2.5.1 Zeigeranalyse

Sprachen wie C und C++ unterstützen den Einsatz von Zeigern für den Zugriff auf Speicheradressen. Um die Auswirkungen von Zeigerargumenten bei Prozeduraufrufen berücksichtigen zu können, ist der Einsatz von interprozeduraler Analyse notwendig. Das Codebeispiel in Listing 2.3 soll die Herausforderungen solcher Zeigeranalysen deutlich machen.

```
1 int global = 5;
2
3 void f (int *p) {
4   *p = *p + 1;
5   global = 0;
```

```
6  }
7
8  void g (int *q) {
9    *q = *q - global;
10 }
11
12 int main (void) {
13   int a = 10;
14   int *b = &a;
15   void (*fptr) (int *q) = g;
16
17   f(&a);
18   (*fptr)(b);
19
20   return 0;
21 }
```

Listing 2.3: Beispielprogramm zu Zeigeranalyse

Es bestehen zwei Abhängigkeiten zwischen dem Aufruf von Prozedur `f` in Zeile 17 und dem folgenden Aufruf von Prozedur `g` (durch Funktionszeiger `fptr`). Einerseits wird in beiden Fällen auf den Wert der Variable `a` zugegriffen. Andererseits verändert `f` den Wert der globalen Variable `global`, welche anschließend von `g` gelesen wird. Damit die Analyse diese Abhängigkeiten erkennen kann, muss sie folgende Probleme lösen:

- **Indirekte Prozeduraufrufe**
  Funktionszeiger wie in Zeile 15 müssen aufgelöst werden können. Zu indirekten Aufrufen zählen auch virtuelle Methoden wie in C++.
- **Interprozedurale Zeigeranalyse**
  Die *Bottom-Up*-Phase, ausgehend von den Prozeduren `f` und `g`, muss dem Aufrufer (`main`) die Schreiboperation auf das Argument bekannt geben. Um die Abhängigkeit der beiden aufgerufenen Prozeduren zu erkennen, muss anschließend erkannt werden, dass der Zeiger `b` auf die Adresse `&a` zeigt.
- **Globale Variablen**
  Eine gute interprozedurale Analyse muss erkennen, dass die beiden aufgerufenen Prozeduren auf die globale Variable `global` zugreifen. Dadurch entsteht eine echte Abhängigkeit.

2.5.2 Kontextsensitivität

Eine Prozedur kann sich abhängig von den Kontextinformationen der Aufrufstelle (Argumente sowie globale Variablen) unterschiedlich verhalten. Falls eine interprozedurale Analyse den Kontext der Aufrufe berücksichtigt, so spricht man von Kontextsensitivität. Dadurch lassen sich präzisere Resultate erzielen, da Auswirkungen des konkreten Verhaltens eines Aufrufs betrachtet werden. Es existieren unterschiedliche Möglichkeiten, um kontextsensitive Analysen zu realisieren. Im Folgenden werden zwei Techniken erläutert: eine Analyse auf Klonbasis und eine mit Zusammenfassungen.

Bei der kontextsensitiven Analyse auf Klonbasis werden für sämtliche Aufrufstellen Kopien der aufgerufenen Prozeduren erstellt. Die Analyse erkennt somit bei jeder Prozedur, von welcher Aufrufstelle diese aufgerufen wurde. Dadurch entsteht zum einen eine große Anzahl an unterschiedlichen Kopien (eine pro Kontext). Zum anderen kann Rekursion schlecht aufgelöst werden, weil die verschiedenen Iterationen durch eine einzige Kopie einer Aufrufstelle stammen.

Im Falle der kontextsensitiven Analyse mit Zusammenfassung können ebenfalls Prozeduren geklont werden.[1] Hier erfolgt allerdings vorher eine Analyse, die das Verhalten von Prozeduren bezüglich verschiedener Aufrufkontexte zusammenfasst. Diese Analyse besteht aus zwei unterschiedlichen Phasen:

1. Die **Bottom-Up-Phase** verwendet lokale Informationen über die Prozedur um Informationen über das Verhalten zu den Aufrufern zu propagieren. Das Resultat der Phase ist eine Zusammenfassung der Auswirkungen der Prozeduren für alle Aufrufkontexte.

2. In der **Top-Down-Phase** werden Aufruferinformationen zu den aufgerufenen Prozeduren propagiert. Damit können Ergebnisse bezüglich dieser Prozeduren berechnet werden.

Tritt bei den zu analysierenden Programmen Rekursion auf, so können im Allgemeinen unendlich viele Aufrufpfade entstehen. Um dennoch eine effektive Berechnung durchführen zu können, wird eine Fixpunktlösung mit Hilfe stark zusammenhängender Komponenten ermittelt.

2.5.3 Flussinsensitivität

Eine flussinsensitive Analyse ignoriert den Kontrollfluss eines Programms. Sie nimmt prinzipiell an, dass die Operationen in beliebiger Reihenfolge ausgeführt werden. Zur Verdeutlichung zeigt Listing 2.4 zunächst eine Menge aufeinanderfolgender Zuweisungsinstruktionen.

```
1 Object *a = new Object(); // erstelltes Objekt: h
2 Object *b = new Object(); // erstelltes Objekt: i
3 Object *c = new Object(); // erstelltes Objekt: j
4 a = b;
5 b = c;
6 c = a;
```

Listing 2.4: Beispielprogramm zu Flussinsensitivität

Man sieht, dass nach Zeile 4 der Zeiger `a` nun auf i zeigt. Nach Zeile 5 zeigt `b` auf j und nach Zeile 6 zeigt der Zeiger `c` ebenfalls auf i. Diese Erkenntnisse berücksichtigen den Kontrollfluss der Instruktionen. So wird nach jeder Instruktion berechnet, auf welche Ziele jede Variable zeigen kann. Es ist möglich, dass bestimmte Ziele nach einer Anweisung nicht mehr erreichbar sind. Zeile 4 "zerstört"beispielsweise das Ziel h.

Die flussinsensitive Analyse nimmt keine Rücksicht auf die Reihenfolge der Instruktionen. Es wird lediglich berechnet, auf welche Ziele eine Variable möglicherweise zeigen kann. Eine "Vernichtung"von Zielen, wie oben erwähnt, findet dabei nicht statt. Fussinsensitive Analysen sind somit weniger präzise als flusssensitive.

[1] *Inlining* ist auch möglich.

KAPITEL 3

Analyse des Parallelisierungspotentials

Dieses Kapitel beschreibt das in dieser Arbeit verwendete Verfahren zur Ermittlung von Parallelisierungspotential. Dazu erfolgt zunächst eine Übersicht des grundsätzlichen Aufbaus mit einer Beschreibung der notwendigen Komponenten. Anschließend wird erläutert, wie die verschiedenen Analysen und Heuristiken eingesetzt werden. Am Ende des Kapitels wird das Vorgehen der verwendeten Abhängigkeitsanalysen beschrieben, welche die zentralen Elemente zur Bewertung von Parallelisierungspotential sind.

3.1 Konzept und Vorgehensweise

Dieser Abschnitt erläutert den grundlegenden Aufbau der Anwendung. Es wird gezeigt, welche Komponenten für die Analyse notwendig sind und wie diese interagieren. Ein besonderes Augenmerk beim Entwurf liegt auf einfacher Erweiterbarkeit und Austauschbarkeit. Gerade bei Analysewerkzeugen besteht der Wunsch nach einer universell einsetzbaren Lösung. Erweiterungen bezüglich der Funktionsvielfalt, wie die Unterstützung von Schleifen als parallele Tasks oder zusätzlichen Programmiersprachen, sollen einfach zu realisieren sein. Die gesamte Prozesskette besteht aus mehreren Schritten, vom Einlesen der Benutzeranwendung in Form von Quelltext-Dateien bis hin zur Ausgabe des Analyse-Ergebnisses.

In Abbildung 3.1 ist der Prozessablauf der Anwendung zu sehen. Graue Rechtecke stellen Verarbeitungseinheiten dar, welche unterschiedliche Berechnungen durchführen. Die weißen Dateisymbole stehen für Ein- oder Ausgabedateien, die von den Verarbeitungseinheiten gelesen oder erstellt werden. Der Inhalt des gestrichelten Rechtecks beschreibt die internen Komponenten des Systems. Die Pfeile deuten auf die vorgegebene Bearbeitungsreihenfolge hin.

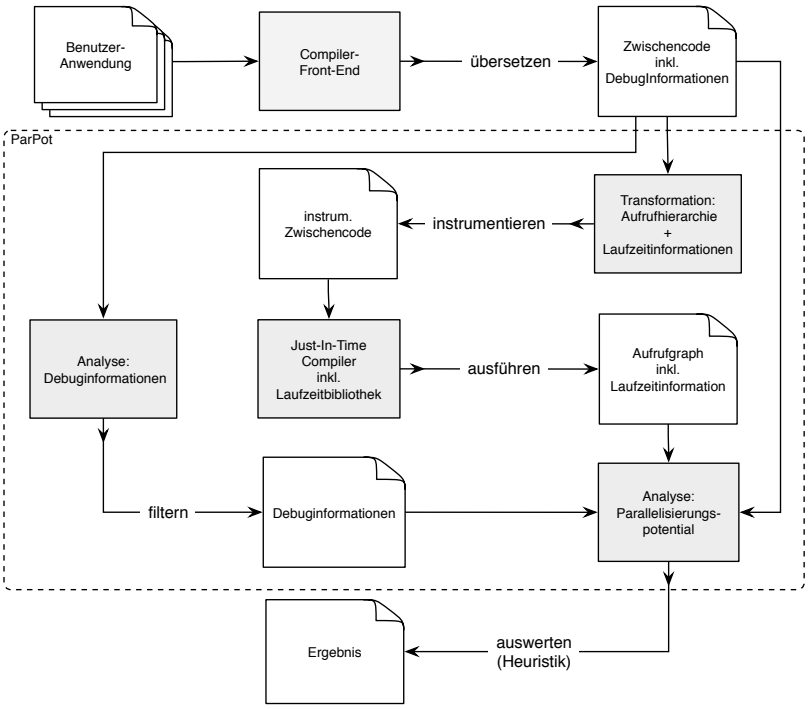

Abbildung 3.1: Prozessablauf der Analyseanwendung

Das Verfahren dieser Arbeit lässt sich in vier aufeinanderfolgende Bearbeitungsschritte einteilen:

1. **Erzeugung von Zwischencode**
   Für eine einheitliche, sprachunabhängige Lösung wird der Quelltext der Benutzeranwendung in Zwischencode übersetzt. Sämtliche Analysen und Transformationen arbeiten ausschließlich auf dieser Repräsentation.

2. **Instrumentierung**
   In das übersetzte Benutzerprogramm werden Anweisungen eingefügt, welche das Verfahren für die Ermittlung von dynamischen Ablaufinformationen benötigt.

3. **Ausführung**
   Durch Ausführung des instrumentierten Codes können die Ablaufinformationen extrahiert werden.

4. **Analyse**
   Im letzten Schritt werden die verschiedenen Analysen und Heuristiken zur Bewertung des Parallelisierungspotentials angewendet. Das Ergebnis dieser Phase ist eine Liste mit den potentiell günstigsten Stellen für Parallelität.

In den folgenden Abschnitten werden die verschiedenen Prozessschritte einzeln betrachtet, um die unterschiedlichen Vorgänge näher zu erläutern.

3.2 Erzeugung von Zwischencode

Eine Anforderung an das zu entwickelnde System ist die vollständige Unterstützung der Programmiersprachen C und C++. Für einen allgemeinen Einsatz muss der gesamte Sprachumfang abgedeckt werden. Diese Aufgabe ist alles andere als trivial. Vor allem C++ bietet aufgrund der unterstützten Programmierparadigmen eine große Anzahl verschiedener Sprachkonstrukte. Objektorientierung und die Existenz von Zeigern sind nur einige Herausforderungen für die Abhängigkeitsanalyse.

Um solche sprachabhängigen Merkmale unter Berücksichtigung von Sonderfällen zu unterstützen, und noch viel wichtiger, eine sprachunabhängige Lösung zu entwerfen, ist Abstraktion notwendig. Die Analyseverfahren dürfen nicht auf dem Ursprungscode arbeiten, sondern müssen einen generell gültigen Zwischencode verwenden. Im Compilerbau kommt dafür meist eine Form von Drei-Adress-Code zum Einsatz. Dieser besteht aus einer Sequenz von elementaren Programmoperationen wie der Addition von zwei Zahlen. Der Zwischencode wird durch ein Compiler-Front-End erstellt, welches auch statische Prüfungen[1] durchführt.

Abbildung 3.2 zeigt den Teil des Systems, in welchem der Anwendungscode in den Zwischencode übersetzt wird. Das Compiler-Front-End muss die Sprachen C und C++ unterstützen. Außerdem soll der Zwischencode mächtig genug sein, um sämtliche Sprachkonstrukte effizient darstellen zu können. Die Aufteilung des Quelltextes der Benutzeranwendungen auf mehrere Dateien ist die Regel. Damit Transformationen und Analysen dennoch einen einfachen Zugriff erhalten, muss das Front-End den gesamten Zwischencode in einheitlicher Form erstellen.

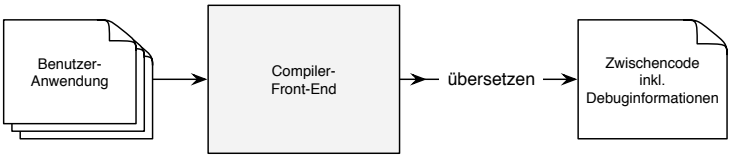

Abbildung 3.2: Erzeugung von Zwischencode

Beim Übersetzen des Ausgangscodes werden unter anderem neue Bezeichnungen für Prozeduren sowie Variablen eingeführt. Der Benutzer des Analysewerkzeugs kann anhand dieser Bezeichner im Allgemeinen keinen Rückschluss auf die entsprechenden Stellen im Ausgangscode ziehen. Das resultierende Analyse-Ergebnis wäre somit nur eingeschränkt nützlich. Eine Lösung für dieses Problem sind Debuginformationen. Dabei handelt es sich um Metadaten, welche das Front-End in den Zwischencode einfügt. Sie enthalten Hinweise auf die korrespondierenden Originalstellen und gegebenenfalls Typinformationen.

3.3 Ermittlung von Ablaufinformationen

Das verwendete Analyseverfahren muss selbständig potentielle Prozeduraufrufstellen identifizieren, die auf Abhängigkeiten zu überprüfen sind. Das Ergebnis muss anschließend hinsichtlich

[1] Konkret sind das zwei Arten von Prüfungen: syntaktische Prüfungen und Typprüfungen.

der Relevanz für eine Parallelisierung bewertet werden. Beide Aufgaben erfordern Informationen über den Ablauf der zu untersuchenden Anwendung. Im Gegensatz zur Abhängigkeitsanalyse werden die Ablaufinformationen allerdings nicht durch ein statisches Vorgehen ermittelt. Für die Aussagekraft der Ergebnisse spielt es eine große Rolle, dass der untersuchte Programmablauf mit der wahren Ausführung übereinstimmt. Nur wenn dies der Fall ist, können Laufzeiteinsparungen durch eine Parallelisierung der Benutzeranwendung erwartet werden.

Durch statische Analyse erreicht man lediglich eine Annäherung an die tatsächliche Aufrufhierarchie. In vielen Fällen (z. B. aufgrund Aufruf von externen Bibliotheken) ist das konkrete Sprungziel einer Aufrufstelle erst zur Laufzeit bekannt. Für die Bewertung von Parallelisierungspotential ist außerdem die Häufigkeit von Aufrufen einer Prozedur essentiell. Auch hier ist eine statische Ermittlung nicht ohne Weiteres möglich, da Aufrufhäufigkeiten beispielsweise von Eingabedaten abhängig sein können. Das wohl wichtigste Kriterium für eine mögliche Parallelisierung sind die Laufzeiten von Prozeduren. Nur wenn sie einen bestimmten Anteil der Gesamtlaufzeit der Anwendung ausmachen, können Einsparungen durch Parallelität erreicht werden (Stichwort: Amdahls Gesetz). Statische Techniken, wie die Verwendung von Instruktionszähler, sind für die Bestimmung der Laufzeiten nicht geeignet. Die tatsächlichen Zeiten können zum einen abhängig von der verwendeten Hardware sein, zum anderen können auch hier externe Aufrufe die Resultate beeinflussen.

Aufgrund der erwähnten Probleme erfolgt die Ermittlung des Aufrufgraphen (siehe Abschnitt 2.1) sowie der Ausführungszeiten der Prozeduren zur Laufzeit. Bei diesem Vorgehen müssen jedoch Einschränkungen berücksichtigt werden. Die verwendeten Eingabedaten sowie der Ausführungskontext haben Einfluss auf den Verlauf der Anwendung. Die ermittelten Ablaufinformationen können sich abhängig von der konkreten Ausführung stark unterscheiden. Es ist daher sehr wichtig, eine möglichst repräsentative Programmausführung zu gewährleisten.

Um den Ablauf einer Anwendung zur Laufzeit ermitteln zu können, muss an bestimmten Stellen ein Informationsaustausch mit der Analyse erfolgen. Dafür werden die entsprechenden Stellen des erzeugten Zwischencodes mit Anweisungen ergänzt (Instrumentierung, siehe Abschnitt 4.3.1). Dabei handelt es sich um Aufrufe einer Laufzeit-Bibliothek. Die Bibliothek muss anhand der Aufrufzeitpunkte den Ablauf der Anwendung untersuchen und benötigte Informationen protokollieren. Wichtig sind vor allem Details über die verschiedenen Aufrufstellen von Prozeduren. Dazu gehört neben dem Namen der Prozedur auch die gesamte Ausführungszeit. Abbildung 3.3 zeigt den Vorgang.

Nachdem die Ausführung beendet wurde, müssen die Ablaufinformationen für die weitere Verarbeitung zur Verfügung stehen. Die Bibliothek bereitet dafür die protokollierten Daten auf und schreibt sie in einem geeigneten Format in eine Ausgabedatei.

3.4 Analyse und Heuristiken

Der letzte Schritt des Verfahrens ist die eigentliche Analyse des Parallelisierungspotentials. Hier werden die gesammelten Informationen bezüglich des Programmablaufs verwendet, um geeignete Paare von Prozeduren für die Abhängigkeitsanalyse zu identifizieren. Erst nachdem die

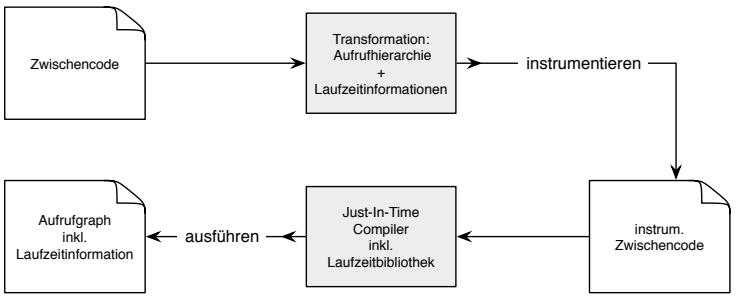

Abbildung 3.3: Instrumentierung des Zwischencodes

Abhängigkeiten vollständig ermittelt wurden, lässt sich eine Aussage hinsichtlich der möglichen Parallelisierung von Prozeduraufrufen treffen. Abbildung 3.4 zeigt, dass die Analysetechniken Informationen über Aufrufgraph, Zwischencode und Debug-Metadaten benötigen.

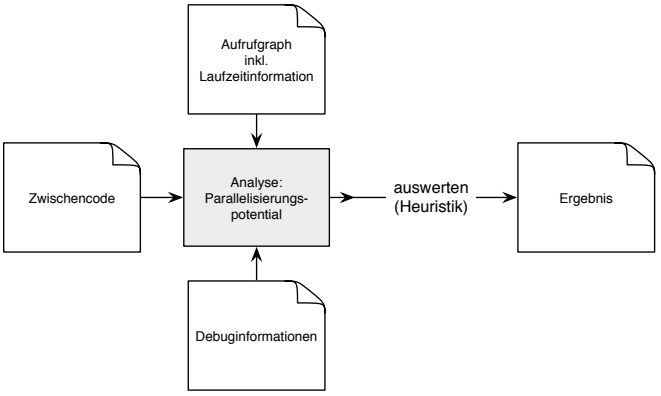

Abbildung 3.4: Analyse des Parallelisierungspotential

In diesem Abschnitt werden neben dem grundsätzlichen Vorgehen zur Auswahl der Prozeduren auch die verwendeten Heuristiken für die Bewertung erläutert.

3.4.1 Auswahl der Prozedurpaare

Komplexe Prozeduren rufen häufig eine Vielzahl von Unterprozeduren auf, wobei manche davon unabhängig zu den anderen ausgeführt werden können. Das trifft dann zu, wenn keine Daten- oder Kontrollabhängigkeiten zwischen ihnen existieren. Die Ausführungsreihenfolge der entsprechenden Prozeduraufrufe unterliegt somit nur den Regeln der imperativen Programmiersprachen. Diese bieten bei sequenziellen Programmen keine Möglichkeit, um eine beliebige Reihenfolge zu erlauben.

Durch den Einsatz von Bibliotheken wie Cilk oder OpenMP können Entwickler mit relativ einfachen Mitteln *fork-join*-Parallelität in ihrer Anwendungen verwenden. Damit lassen sich Prozeduren als eigenständige Tasks parallel zu anderen Berechnungen ausführen. Die Ergebnis-

se der einzelnen Tasks können schließlich per Synchronisationsmechanismen zu fest definierten Zeitpunkten verwendet werden.

Diese Arbeit soll das soeben beschriebene Vorgehen unterstützen. Das Ziel ist es, günstige Prozeduren für *fork-join*-Parallelität zu identifizieren. Dazu werden unter anderem Abhängigkeiten zwischen verschiedenen Aufrufstellen innerhalb einzelner Prozeduren betrachtet. Das System verwendet den dynamischen Aufrufgraph, um ausgehend von der Einstiegsprozedur, sämtliche Paare von Aufrufoperationen zu ermitteln und eine Bewertung des Parallelisierungspotential abzugeben.

Definition 4. *Sei $\mathcal{P}$ die Menge aller Prozeduren einer Anwendung, $\mathcal{A}_p$ die Menge der Prozduraufrufstellen einer Prozedur $p \in \mathcal{P}$ und $<_p \subset \mathcal{A}_p \times \mathcal{A}_p$ die "kleiner"-Relation bezüglich der Codeposition von zwei Aufrufstellen in p. Dann wird die Menge aller zu untersuchender Paare von Aufrufstellen für die Prozedur p definiert durch*

$$\mathcal{G}_p := <_p$$

Für jedes Paar aus $\mathcal{G}_p$ berechnet eine Abhängigkeitsanalyse die spezifischen Abhängigkeiten zwischen den zugehörigen Prozeduraufrufen (siehe Abschnitt 3.5). Die Abhängigkeiten werden als Kanten zwischen den Aufrufstellen in einen Abhängigkeitsgraph eingefügt. Es existiert je ein Graph für jede Prozedur der Benutzeranwendung. Durch ein iteratives Vorgehen werden sämtliche Knoten des Aufrufgraphen untersucht, sodass eine vollständige Analyse sämtlicher Aufrufstellen erfolgt. Jede Prozedur wird dabei höchstens ein Mal untersucht, um der Gefahr einer Verklemmung bei Rekursion entgegenzuwirken. Abbildung 3.5 verdeutlicht die Vorgehensweise zur Auswahl der zu analysierenden Prozeduraufrufe. Die gestrichelten Rechtecke zeigen die Mengen von Aufrufstellen, welche je einen eigenen Aufrufgraph bilden.

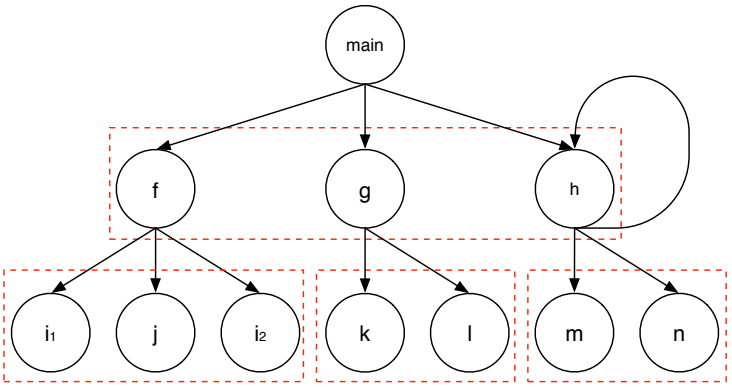

Abbildung 3.5: Auswahl der Prozeduraufrufe zur Analyse

Die Reihenfolge der Aufrufe wird gemäß Tiefensuchordnung abgelesen. Damit ergeben sich folgende Mengen von zu untersuchenden Paaren:

$$\mathcal{G}_{main} = \{(f, g), (f, h), (g, h)\}$$

$$\mathcal{G}_f = \{(i_1, j), (i_1, i_2), (j, i_2)\}$$

$\mathcal{G}_g = \{(k, l)\}$

$\mathcal{G}_h = \{(m, n), (m, h), (n, h)\}$

In dem gezeigten Beispiel existiert ein Zyklus durch den Aufruf $h \to h$. Wie man anhand der Menge $\mathcal{G}_h$ sehen kann, erfolgt der Aufruf durch die letzte Aufrufstelle innerhalb der Prozedur h. Die Menge aller Paare von Aufrufstellen für die Abhängigkeitsanalyse besteht aus $\cup_{p \in \mathcal{P}} \mathcal{G}_p$.

3.4.2 Heuristiken

Nachdem die Abhängigkeitsanalyse für alle Prozeduren der Anwendung einen Abhängigkeitsgraph erstellt hat (siehe Abschnitt 3.5), kann das Parallelisierungspotential berechnet werden. Ein wichtiges Kriterium ist dabei die erreichbare Laufzeiteinsparung.

Definition 5. *Seien $\mathcal{P}$, $\mathcal{A}_p$ und $\mathcal{G}_p$ wie in Definition 4. Außerdem soll die Abbildung $\tau : \mathcal{A}_p \to \mathbb{R}$ die Ausführungszeit einer Aufrufstelle $a \in \mathcal{A}_p$ liefern. Dann ist $\tau_{max} : \mathcal{A}_p \times \mathcal{A}_p \to \mathbb{R}$ die maximale parallele Laufzeiteinsparung eines Aufrufstellenpaars $g \in \mathcal{G}_p$ mit*

$$\tau_{max}(g) = \sum_{a \in \{first(g),\ second(g)\}} \tau(a) - \max_{a \in \{first(g),\ second(g)\}} (\tau(a))$$

Für die maximale Laufzeiteinsparung wird davon ausgegangen, dass sämtliche Aufrufstellen in einem eigenen Task ausgeführt werden. Die Summe der Ausführungszeiten beschreibt hierbei die sequenzielle Laufzeit der Tasks. Die maximale Einsparung ist diese sequenzielle Zeit abzüglich der Zeit für den längsten Task.

Die Ergebnisse der Anwendung sollen dem Benutzer einen guten Überblick über Aufrufstellen von Prozeduren mit hohem Potential für Parallelisierung bieten. Neben der maximalen Laufzeiteinsparung müssen dazu die Abhängigkeiten bewertet werden. Zum einen zählt hier natürlich die Anzahl. Zum anderen sind Abhängigkeiten unterschiedlich gravierend im Bezug auf Parallelität. Echte Abhängigkeiten sind beispielsweise ein größeres Hindernis für Parallelität als umgekehrte oder Ausgabeabhängigkeiten. Daher bewertet die Analyse jede Abhängigkeiten mit einem spezifischen Faktor $c_t \in \mathbb{R}$ je Abhängigkeitstyp t.

Die Anzahl von Abhängigkeiten spielt noch eine weitere Rolle für den praktischen Einsatz des Verfahrens. Für die Bewertung einer Abhängigkeit muss berücksichtigt werden, wie viele zusätzliche Abhängigkeiten zu der betroffenen Aufrufstellen existieren. Einzelne Abhängigkeiten sind umso gravierender einzustufen, je kleiner die Gesamtzahl der Abhängigkeiten für die entsprechende Aufrufstelle ist. Die reine Anzahl sollte somit nicht linear in die Berechnung des Parallelisierungspotentials einfließen. Oder anders ausgedrückt, ob zwei oder drei Abhängigkeiten existieren macht einen größeren Unterschied für die Parallelisierung als wenn zwölf oder dreizehn vorhanden sind. Aus diesem Grund wird anstatt einem linearen Faktor die Funktion $e^{-x/\lambda}$ verwendet, wobei λ eine Linearitätskonstante ist. Abbildung 3.6 zeigt die Funktion für $\lambda = 10$.

Definition 6. *Seien $\mathcal{P}$, $\mathcal{A}_p$, $\mathcal{G}_p$ sowie τ_{max} wie in Definition 5. Zusätzlich sei $\mathcal{D}_g$ die Menge an Abhängigkeiten innerhalb des Analysepaars $g \in \mathcal{G}_p$ und $f : \mathcal{D}_g \to \mathbb{R}$ liefert für eine gegebene Abhängigkeit $d \in \mathcal{D}_g$ den Wert c_t, welcher beschreibt, wie gravierend der zugrundeliegende*

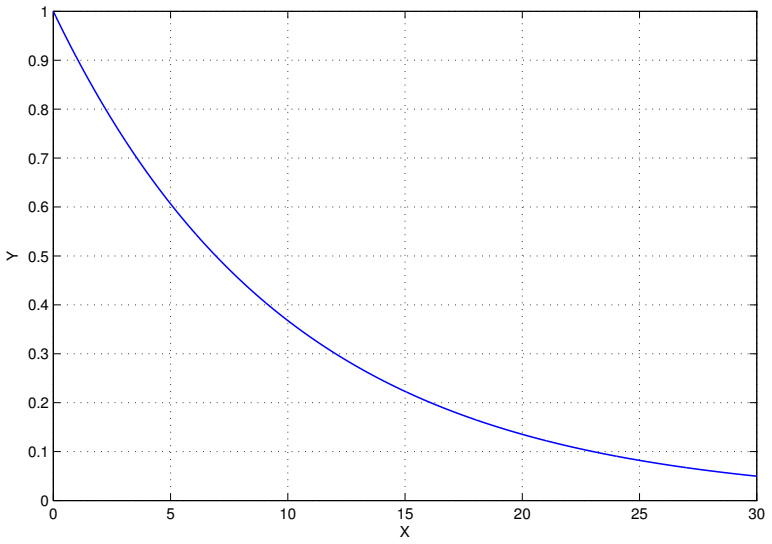

Abbildung 3.6: Funktion $e^{-x/\lambda}$ mit $\lambda = 10$

Abhängigkeitstyp t ist. Das Parallelisierungspotential $\eta : \mathcal{A}_p \times \mathcal{A}_p \to \mathbb{R}$ eines Analysepaars g wird definiert durch

$$\eta(g) = \frac{\tau_{max}(g)}{e^{(\sum_{d \in \mathcal{D}_g} f(d))/\lambda}}$$

Für die Berechnung des Parallelisierungspotentials wird die maximale Laufzeiteinsparung als linearer Faktor betrachtet. Das gilt nicht für die Abhängigkeiten, welche gewichtet über die oben gezeigte e-Funktion in die Berechnung einfließen.

Zur Präsentation der Analyseergebnisse wird die Liste an Paaren von Aufrufstellen mit der maximal zu erwartenden Laufzeiteinsparung (τ_{max}) sowie den ermittelten Abhängigkeiten ausgegeben. Die Liste wird sortiert nach dem hier beschriebenen Parallelisierungspotential (η).

3.5 Suche nach Abhängigkeiten

Die Existenz von Kontroll- oder Datenabhängigkeiten kann eine parallele Ausführung mehrerer Prozeduraufrufe ungültig machen. Für die Ermittlung des Parallelisierungspotentials ist es daher sehr wichtig, die vorhandenen Abhängigkeiten möglichst genau zu identifizieren. Der in dieser Arbeit verwendete Ansatz unterscheidet zwischen drei verschiedenen Arten von Abhängigkeiten, welche in der Anwendung auftreten können: Datenabhängigkeiten durch Argumente, durch globale Variablen oder Kontrollabhängigkeiten. Im restlichen Teil dieses Abschnitts werden die Abhängigkeitsarten mit Hilfe von Codebeispielen beschrieben und die verwendeten Suchstrategien präsentiert.

3.5.1 Datenabhängigkeiten durch Argumente

Die Verwendung aktueller Parameter kann zu Datenabhängigkeiten zwischen mehreren Prozeduren führen. Im allgemeinen Fall muss der Datenfluss zwischen den Prozeduraufrufen betrachtet werden. Falls ein Pfad mehrere Aufrufinstruktionen enthält und mindestens ein schreibender Datenzugriff existiert, so besteht eine Abhängigkeit. Das Codebeispiel aus Listing 3.1 soll mögliche Fälle von Datenabhängigkeiten durch Argumente deutlich machen.

```
1  void f (int &p) {
2     p = p + 1;          // Schreib- und Leseoperation
3  }
4
5  int g (int &p) {
6     return p + 1;       // Leseoperation
7  }
8
9  void h (int &p) {
10    f(p);               // ind. Schreib- und Leseoperation
11 }
12
13 int main (void) {
14    int a = 0;
15
16    f(a);
17    int b = g(a);
18    h(a);
19    h(b);
20
21    return 0;
22 }
```

Listing 3.1: Abhängigkeiten durch formale Parameter

In dem Beispiel werden ausschließlich Referenzen zur Parameterübergabe verwendet. Damit soll die Lesbarkeit des Quelltextes gesteigert werden. Der Einsatz von Zeigern würde keinen Einfluss auf die Erkenntnisse haben. Die Prozedur `f` führt eine Lese- sowie Schreiboperation auf die übergebene Referenz aus, `g` liest dagegen nur den Wert von `p`. Die Prozedur `h` ruft `f` auf und beinhaltet somit dieselben Operationen.

Abbildung 3.7 zeigt den Abhängigkeitsgraph für die Prozedur `main`. Es werden lediglich Datenabhängigkeiten zwischen Prozeduraufrufen betrachtet. Prinzipiell existiert zusätzlich eine Ausgabeabhängigkeit zwischen den Anweisungen in Zeile 16 und Zeile 18, da beide Aufrufe schreibend auf Variable `a` zugreifen. Allerdings besteht bezüglich derselben Variable eine echte Abhängigkeit zwischen den Instruktionen. Diese ist hinsichtlich Parallelität als gravierender einzustufen. Die Ausgabeabhängigkeit kann daher ignoriert werden. Bei den Instruktionen in Zeile 17 und Zeile 19 besteht die Datenabhängigkeit über über eine zusätzliche Variable (`b`). Um dies zu erkennen muss der Datenfluss verfolgt werden.

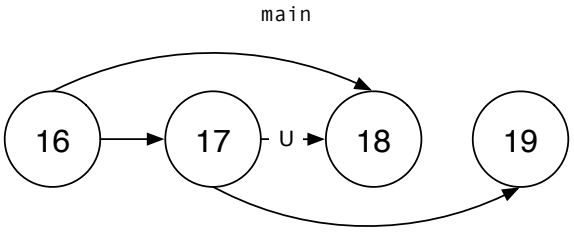

Abbildung 3.7: Abhängigkeitsgraph zu Listing 3.1

Die Erkennung von Datenabhängigkeiten erfolgt in zwei grundlegenden Schritten. Durch interprozedurale Analyse wird geprüft, ob Argumente innerhalb eines Prozeduraufrufs gelesen oder verändert werden. Im zweiten Schritt werden mit Hilfe vorhandener *Def-Use*-Informationen Pfade des Datenflussgraphen untersucht. Sollte ein Pfad von einem Aufruf einer Prozedur zu einem anderen führen, so besteht eine Abhängigkeit. Der entsprechende Abhängigkeitstyp kann durch das Ergebnis der Auswertungen bestimmt werden.

Die interprozedurale Analyse ist essenziell für die Suche nach Abhängigkeiten bei Programmen, welche Zeiger einsetzen. Sie erfolgt in drei aufeinanderfolgenden Phasen. In einer **lokalen Phase** werden je Prozedur sämtliche Zugriffe auf verwendete Speicherobjekte geprüft. Hier ist vor allem die Art des Zugriffs von Interesse. In der sogenannten **Bottom-Up-Phase** werden anschließend Analyseergebnisse von aufgerufenen Prozeduren (der vorherigen Phase) an die Aufrufer hoch-propagiert. Dort erfolgt jeweils eine Verfeinerung von Zugriffsinformationen. Um Verklemmungen durch rekursive Aufrufe entgegenzuwirken, werden vorhandene Zyklen durch die Berücksichtigung *stark zusammenhängender Komponenten* innerhalb des Aufrufgraphen berücksichtigt. In einer **Top-Down-Phase** werden zuletzt Informationen bezüglich Speicherbereiche der Zeigerargumente an die aufgerufenen Prozeduren weitergereicht. Damit lassen sich gegebenenfalls Zugriffsinformationen zusammenfassen. Dies ist beispielsweise der Fall, wenn beim Aufruf einer Prozedur mehrere Zeigeparameter auf ein und denselben Speicherbereich verweisen (Aliasing). Eine genauere Erläuterung der Vorgehensweise erfolgt in Abschnitt 4.2.

3.5.2 Datenabhängigkeiten durch globale Variablen

Ähnlich zum Datenzugriff über formale Parameter können auch Schreiboperationen auf globale Variablen Abhängigkeiten zwischen Prozeduren erzeugen. Das Codebeispiel in Listing 3.2 veranschaulicht die Problemstellung.

```
1 int global_1 = 0;
2 int global_2 = 0;
3
4 void f (void) {
5    gobal_1 = 1;                // Schreiboperation (GV1)
6 }
```

```
7
8  int g (void) {
9     return global_1 + global_2;// Leseoperation (GV1, GV2)
10 }
11
12 void h (void) {
13    f();                        // ind. Aufruf von f
14    global_2 = 2;               // Schreiboperation (GV2)
15 }
16
17 int main (void) {
18    f();
19    g();
20    h();
21
22    return 0;
23 }
```

Listing 3.2: Beispielprogramm - Abhängigkeiten durch globale Variablen

Der Aufruf von Prozedur `h` in Zeile 20 führt einen Schreibzugriff auf die globalen Variablen `global_1` (indirekt über `f`) und `global_2` aus. Daraus resultiert eine Schreibabhängigkeit zum Aufruf von `f` in Zeile 18 sowie eine umgekehrte Abhängigkeit zu dem vorhergehenden Aufruf von `g`. Abbildung 3.8 zeigt den zugehörigen Abhängigkeitsgraph.

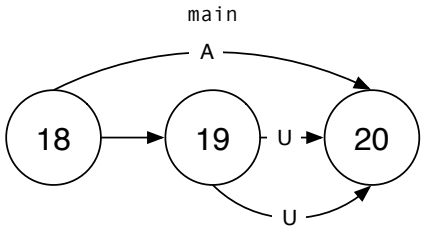

Abbildung 3.8: Abhängigkeitsgraph zu dem Beispiel aus Listing 3.2

Eine Suche nach Datenabhängigkeiten, welche durch den Zugriff auf globale Variablen bestehen, erfolgt auf dieselbe Art und Weise wie im Falle von aktuellen Parametern. Mit Hilfe von *Def-Use*-Informationen lassen sich Abhängigkeiten durch den Datenfluss finden. Die interprozedurale Analyse erfolgt auch hier in drei Phasen. In der lokalen Phase werden sämtliche Instruktionen einer Prozedur untersucht, ob Werte globaler Variablen gelesen oder geschrieben werden. Die Bottom-Up-Phase propagiert Zugriffsinformationen aufgerufener Prozeduren hoch an die Aufrufer. Zusätzlich hat eine Top-Down-Phase die Aufgabe, Ergebnisse von Alias-Analysen an die aufgerufenen Prozeduren weiterzugeben. Gegebenenfalls können dadurch Zugriffsinformationen zusammengefasst werden.

3.5.3 Kontrollabhängigkeiten

Wie im Abschnitt 2.4 beschrieben, besteht eine Kontrollabhängigkeit, falls die Resultate von Prozeduren die Aufrufhäufigkeit anderer Prozeduren beeinflussen. Es müssen somit Kontrollstrukturen in der Benutzeranwendung existieren, damit diese Art von Abhängigkeit auftreten kann. Das Codebeispiel in Listing 3.3 zeigt, dass Kontrollabhängigkeiten durch verschiedene Konstellationen entstehen können.

```
1  int f (const int &r) {
2     return r + 1;  // Leseoperation
3  }
4
5  void g (int &r) {
6     r = r + 1;     // Lese- und Schreiboperation
7  }
8
9  int main (void) {
10
11    int a = f(1);
12    int b = 0;
13
14    if (a >= 10)
15       g(b);
16
17    for (int i = 0; i < f(a); ++i)
18       g(b);
19
20    return 0;
21 }
```

Listing 3.3: Beisprogramm - Kontrollabhänigkeiten

In der `main` Prozedur existiert eine *if*-Anweisung sowie eine *for*-Schleife. Das Resultat des Prozeduraufrufs in Zeile 11 wird in eine lokale Variable `a` geschrieben, welche in der Bedingung der `if`-Anweisung gelesen wird. In dem zugehörigen Rumpf wird die Prozedur `g`, bedingt durch das Resultat von `f`, ausgeführt. Somit sind die beiden Prozeduren kontrollabhängig. Wie man sieht, kann es zur Auswertung von Kontrollabhängigkeiten erforderlich sein, den Datenfluss zu berücksichtigen. Der Abhängigkeitsgraph in Abbildung 3.9 zeigt die beiden Kontrollabhängigkeiten. Anders als in den beiden vorherigen Beispielen wird in diesem Graph zusätzlich ein Knoten für eine Anweisung hinzugefügt, die kein Prozeduraufruf ist (Knoten für Zeile 14). Das ist hier notwendig, um die zugehörige Kontrollabhängigkeit darstellen zu können.

Im Gegensatz zu den Datenabhängigkeiten kann bei der Analyse von Kontrollabhängigkeiten intraprozedural vorgegangen werden. Da die zu untersuchenden Paare von Prozeduraufrufen innerhalb der selben Prozedur (Aufrufer) liegen müssen, ist ein prozedurübergreifendes Vorgehen nicht notwendig. Ausgehend von einer Aufrufstelle werden anhand der *Def-Use*-Kette sämtliche Verwendungen ermittelt, die Bedingungsanweisungen einer Kontrollstruktur sind. Falls sich in

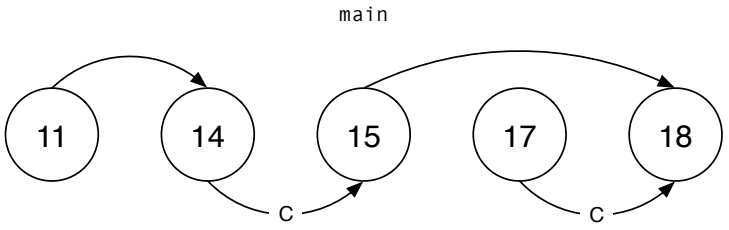

Abbildung 3.9: Abhängigkeitsgraphen zu Beispiel aus Listing 3.3

den zugehörigen Rümpfen ein weiterer Prozeduraufruf findet, so ist dieser kontrollabhängig zur Ausgangsanweisung. Treten in einem Rumpf Definitionsanweisungen auf, dann müssen nachfolgende Verwendungen der definierten Variablen ebenfalls überprüft werden.

KAPITEL 4

Implementierung

Das in dieser Arbeit entwickelte Werkzeug benötigt zur Analyse von Benutzerprogrammen mehrere Komponenten. Neben einem Compiler-Front-End für die Übersetzung in Zwischencode, müssen verschiedene Optionen zur statischen Analyse existieren. Das Compiler-Framework LLVM erfüllt einen Großteil dieser Anforderungen und wurde deshalb als Grundlage für die Implementierung ausgewählt. Es bietet umfangreiche Werkzeuge und Verfahren zur Erstellung von Analysen. Zusätzlich existiert eine Lösung für die interprozedurale Analyse, welche aufwändige Zeiger-Berechnungen durchführt (siehe Abschnitt 4.2).

Im Rest des Kapitels werden zunächst die Besonderheiten von LLVM sowie der Datenstrukturanalyse vorgestellt. Anschließend folgt eine Übersicht auf zentrale Stellen der Umsetzung.

4.1 LLVM - Low Level Virtual Machine

LLVM ist ein freies Compiler-Framework für die Übersetzung, Optimierung und Ausführung von Programmen. Die Entwicklung von LLVM wurde im Jahr 2000 an der Universität Illinois begonnen und hat das Ziel, Programmanalysen und Transformationen in allen Compiler-Phasen zu unterstützen [Lat02].

Für LLVM gibt es eine große Anzahl von Compiler-Front-Ends und Codegeneratoren (statisch sowie Just-In-Time), sodass viele Programmiersprachen und Prozessorarchitekturen unterstützt werden. Beispiele für solche Architekturen sind x86 (32- sowie 64-Bit), PowerPC und ARM. Bezüglich der unterstützten Sprachen zählen C, C++, Java, Objective-C und FORTRAN zu den bekanntesten. Ein großer Vorteil beim Einsatz von LLVM ist der verwendete Zwischencode (*Intermediate Representation*, LLVM-IR), durch den sprach- und plattformunabhängige Optimierungen ermöglicht werden.

Die Architektur von LLVM besitzt einen modularen Aufbau mit verschiedenen Bibliotheken für alle Compiler-Phasen. Diese können verwendet werden, um eigene Compiler mit Hilfe des LLVM-Frameworks zu erstellen. Außerdem dienen die Bibliotheken zur Erstellung von Analysen und Transformationen.

4.1.1 Repräsentation des Zwischencodes

Der Zwischencode wird von LLVM für sämtliche Compiler-Phasen genutzt. Er dient als Eingabe für die Code-Erzeugung sowie als interne Repräsentation bei Optimierungen oder Analysen. LLVM-IR kann in drei gleichwertigen Formen dargestellt werden, die leicht ineinander transformierbar sind. Eine Möglichkeit ist speicherinterner Compilercode für die Verwendung innerhalb von Bibliotheken. Eine andere sind Bitcode-Dateien, welche zum Beispiel für die Ausführung durch einen Just-In-Time-Compiler benutzt werden können. Die dritte Möglichkeit ist die Darstellung in textueller Form, ähnlich zu einem Drei-Adress-Code.

LLVM-IR bietet eine unbeschränkte Menge an virtuellen Registern. Die Repräsentation von Instruktionen, welche auf die Register zugreifen, erfolgt in statischer Einzelzuweisungsform (*Static Single-Assignment*, SSA) [ASU86]. Jedes virtuelle Register wird somit in genau einer Instruktion beschrieben und jede Verwendung von Registern wird von ihrer Definition dominiert. Für den Fall, dass einzelne Variablen in mehreren Kontrollflusspfaden definiert sind, beinhaltet die Instruktionsmenge von LLVM-IR explizite Phi(ϕ)-Operationen. Damit können die verschiedenen Definitionen innerhalb einer Instruktion miteinander kombiniert werden. Die SSA-Form erzeugt durch ihre Eigenschaften *def-use*-Ketten. Das ermöglicht eine einfache Datenflussanalyse innerhalb der Anwendung.

LLVM-Zwischencode bietet zwei verschiedene Instruktionen für den Aufruf von Prozeduren. Bei Prozeduren, welche keine Ausnahmebehandlung verwenden,[1] kommen sogenannte `call`-Instruktionen zum Einsatz. Sie erwarten einen typisierten Funktionszeiger sowie eine Liste von Argumenten. Die Ausführung des Programms wird nach der aufgerufenen Prozedur in demselben Basisblock fortgesetzt.

Zur Behandlung von Ausnahmen werden Prozeduren mit Hilfe der Instruktion `invoke` aufgerufen. Diese erwartet neben den Parametern, welche auch die `call`-Instruktion bietet, zusätzlich noch zwei Basisblöcke. Einer davon wird ausgeführt, falls der Aufruf erfolgreich war. Der zweite Basisblock stellt die Ausnahmebehandlung dar und wird im Falle einer Ausnahme aufgerufen. In jedem Fall wird der aktuelle Basisblock nach dem Aufruf der Prozedur verlassen.

4.1.2 Analysen und Transformationen

Ein wichtiger Bestandteil von LLVM ist die vorgegebene Architektur von Analysen und Transformationen (*Passes*). Durch solche *Passes* werden Auswertungen und Optimierungen auf Zwischencode ausgeführt. Außerdem können Analyse-Ergebnisse über vorgegebene Methoden ausgegeben werden. Eine große Unterstützung ist die modulare Repräsentation von Benutzeranwendungen. Für jedes Konstrukt (Prozeduren, Basisblöcke, Instruktionen, Variablen usw.) exis-

[1] Prozeduren ohne Ausnahmebehandlung werden mit dem Zusatz `nounwind` gekennzeichnet.

tiert eine eigene Klasse mit den entsprechenden Zugriffsmethoden. Die Klasse `Module` stellt die höchste Ebene in einem Programm dar. Die Informationen stammen entweder aus einer Übersetzungseinheit des Ursprungsprogramms oder aus mehreren Einheiten, welche durch einen Linker gebunden wurden. Die `Module`-Klasse enthält eine Liste mit sämtlichen Prozeduren der Anwendung. Diese werden durch die Klasse `Function` repräsentiert und beinhalten wiederum eine Liste von Basisblöcken der Prozedur (Klasse `BasicBlock`). Jeder Basisblock enthält eine Liste der enthaltenen Instruktionen (Klasse `Instruction`).

In LLVM sind *Passes* Klassen, welche von bestimmten Unterklassen der Klasse `Pass` erben. Jede Unterklasse ist für eine bestimmte Art von Analysen vorgesehen. So wird beispielsweise die Klasse `FunctionPass` für jede Prozedur der Anwendung einzeln aufgerufen, um intraprozedurale Analysen oder Transformationen auszuführen. In der Implementierung dieser Analyseanwendung kommen ausschließlich Analysen und Transformationen vor, welche von der Klasse `ModulePass` erben. Diese sind für interprozedurale Analysen über die gesamte Benutzeranwendung bestimmt.

Die folgende Liste bietet eine Übersicht über bestehende Analysen, welche zur Realisierung der Anwendung verwendet wurden:

- **Datenstrukturanalyse** (*Data Structure Analysis)*
  Interprozedurale Zeigeranalyse für die Suche nach Abhängigkeiten (siehe Abschnitt 4.2).
- **Erzeugung des Dominatorbaums** (*Dominator Tree Construction)*
  Diese Analyse erstellt einen Dominatorbaum anhand des Kontrollflussgraphen. Damit stehen Dominator-Informationen für die Analyse zur Verfügung.
- **Erzeugung des Postdominatorbaums** (*Post-Dominator Tree Construction*)
  Erzeugung des Postdominatorbaums anhand statischer Kontrollfluss-Informationen zur Verwendung von Postdominator-Informationen.

In Abschnitt 4.3 werden die Analysen und Transformationen vorgestellt, welche für die Anwendung zusätzlich erstellt werden mussten.

4.2 Datenstrukturanalyse

Die statische Suche nach Abhängigkeiten kann ohne interprozedurale Analysen keine Aussagen zum Verhalten von Prozeduraufrufen machen. Im Basisumfang von LLVM ist kein Werkzeug vorhanden, das kontextsensitive, interprozedurale Analysen bietet. Für diesen Zweck wurde jedoch die Datenstrukturanalyse (*Data Structure Analysis*, DSA) entwickelt, welche sich als separates Projekt (*Pool-Allocation*) in die LLVM-Umgebung integrieren lässt [Lat05].[2]

DSA ist kontextsensitiv, feldsensitiv, flussinsensitiv und unterstützt sämtliche Spracheigenschaften von C. Dazu zählen unsichere Typkonversionen, Funktionszeiger, Rekursion und Sprunganweisungen. Feldsensitivität bedeutet, dass die Zeigeranalyse bezüglich verschiedener Felder von Datenstrukturen unterscheidet. Flussinsensitivität sagt aus, dass die Reihenfolge der Instruktionen nicht beachtet wird. Der verwendete Algorithmus untersucht in einem inkrementellen

[2] Die aktuelle Version der Datenstrukturanalyse ist lediglich in LLVM 2.7 einsetzbar.

Verfahren den Aufrufgraphen nach Zugriffsinstruktionen auf Speicherobjekte. Es wird stets zwischen vollständigen und unvollständigen Informationen unterschieden. Nur wenn die Analyse sicher bestimmen kann, dass keine zusätzlichen Zugriffe auf die Objekte existieren, kann es diese als vollständig einstufen. Falls die Speicherobjekte von mehreren Prozeduren verwendet werden, so kann die Kennzeichnung erst nach interprozeduraler Analyse erfolgen. Durch die Unterscheidung zwischen vollständigen und unvollständigen Informationen werden konservative Aussagen von DSA ermöglicht.

4.2.1 Repräsentation

Intern erzeugt die Datenstrukturanalyse für jede Prozedur der Benutzeranwendung einen Graph (*DSGraph*). Die Knoten (*DSNode*) eines *DSGraph* repräsentieren Speicherobjekte (Heap, Stack), auf welche die Prozedur zugreift. Ein Knoten beinhaltet den Typ des Objekts, die enthaltenen Felder der Datenstruktur (falls Typ nicht skalar) sowie ein spezifisches Kennzeichen. Das Kennzeichen sagt aus, ob es sich bei dem Knoten um ein Objekt auf dem Heap- oder Stack-Speicher handelt bzw. ob es ein globales Objekt (gobale Variable oder Prozedur) ist. Außerdem wird gekennzeichnet, welche Art von Zugriff (lesen, schreiben) auf das Objekt innerhalb der Prozedur erfolgt und ob das Objekt vollständig gemäß der vorherigen Beschreibung ist.

Kanten werden in den Graphen zwischen Feldern (falls vorhanden) von *DSNodes* eingetragen. Sie beschreiben einen Verweis zwischen den beiden Speicherobjekten. Falls beispielsweise ein Element a einer Liste auf ein Datenobjekt d verweist, so existiert zwischen den Knoten von a und d eine Kante. Zusätzlich zu den Kanten zwischen Knoten wird für jedes (zeigerkompatible) virtuelle Register einer Prozedur der Verweis auf den entsprechenden *DSNode*-Knoten im Graph gespeichert.

Um die Darstellung von Graphen der Datenstrukturanalyse zu verdeutlichen, zeigt Listing 4.1 die Prozedur `addList()`. Diese hat die Aufgabe, einen neuen Patienten (`Patient *data`) an eine verkettete Liste (`ListT *list`) anzuhängen. Hierzu wird bis zum letzten Element iteriert, ein neues Element der Liste auf dem Heapspeicher angelegt und dieses mit dem bisher letzten Element verbunden.

```
1  typedef struct List { Patient *data; List *next} ListT;
2
3  void addList (ListT *list, Patient *data) {
4     ListT *b = NULL, *nlist;
5     while(list != NULL) {
6        b = list;
7        list = list->next;
8     }
9     nlist = new ListT;
10    nlist->data = data;
11    nlist->next = NULL;
12    b->next = nlist;
13 }
```

Listing 4.1: Beispielprogramm zur Repräsentation von DSA

Der zugehörige *DSGraph* des Beispielcodes ist in Abbildung 4.1 ersichtlich. Graue Rechtecke stellen *DSNode*-Knoten mit den zugehörigen Feldattributen dar, Typinformationen und Kennzeichen sind nicht gezeigt. Kleine weiße Rechtecke zeigen virtuelle Register (lokale Variablen), die Ellipsen sind formale Parameter.

Am Ende der Konstruktionsphase des Graphen existieren zwei Knoten, einer für das neu erstellte Listen-Speicherobjekt, der andere für die übergebenen Patientendaten (ohne Feldinformationen). Aufgrund der Flussinsensitivität erscheinen zusätzlich gelesene Listenelemente nicht als *DSNodes* im Graph. Die Analyse erkennt jedoch, dass `list`, `b` und `nlist` auf denselben Knoten zeigen. Die Informationen zu dem Speicherobjekt von `Patient` erhält die Analyse durch die Auswertung der aufrufenden Prozedur.

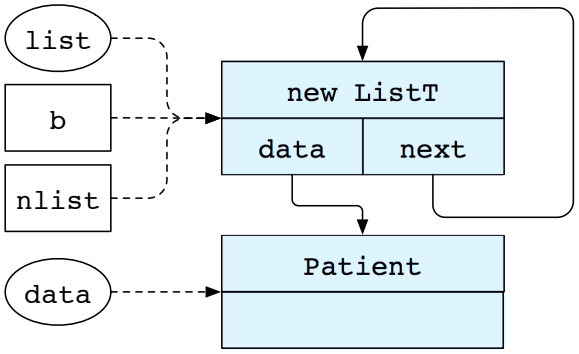

Abbildung 4.1: *DSGraph* zur Prozedur `addList()` aus Listing 4.1

Für die zu implementierende Anwendung liefern die Ergebnisse der Datenstrukturanalyse die Menge an Speicherstellen, auf welche durch einen Prozeduraufruf zugegriffen wurde. Durch Bildung der Schnittmenge von Resultaten mehrerer Aufrufstellen können dadurch Abhängigkeiten gefunden werden.

4.2.2 Konstruktion des Graphen

Datenstruktur-Graphen werden in drei Phasen konstruiert und verfeinert. Die erste Phase erstellt für jede Prozedur der Anwendung einen Graphen, wobei ausschließlich intraprozedurale Auswertungen erfolgen. Im zweiten Schritt vervollständigt eine sogenannte "Bottom-Up"-Phase fehlende Informationen aufgrund von Prozeduraufrufen. Dazu integriert die Analyse Graph-Informationen von aufgerufenen Prozeduren in den *DSGraph* der Aufrufer. Die abschließende "Top-Down"-Phase ergänzt unvollständige Informationen aufgrund aktueller Parameter, indem die Graphen der Aufrufer mit denen der aufgerufenen Prozeduren vereinigt werden. Im Folgenden werden die einzelnen Schritte näher spezifiziert.

Lokale Analyse Das Ziel dieser Analysephase ist die Berechnung eines lokalen *DSGraph* für jede Prozedur, ohne Berücksichtigung von Informationen über aufzurufende Prozeduren oder eigene Aufrufer. Es ist der einzige Schritt, bei dem der Zwischencode betrachtet wird. Die restlichen beiden Phasen arbeiten ausschließlich auf Datenstruktur-Graphen.

Für jedes virtuelle Register wird zu Beginn ein leerer Knoten (mit entsprechendem Typ) im Graph erzeugt. Jedes Register erhält einen Verweis auf den zugehörigen *DSNode*-Knoten. Anschließend iteriert die Analyse über sämtliche Instruktionen der Prozedur. Je nach Art der Instruktion werden zusätzliche Knoten erzeugt, bestehende modifiziert oder mehrere zusammengefasst. Allokationen erzeugen beispielsweise neue Knoten, bei Zuweisungen wird der Knoten des virtuellen Registers mit dem des Zuweisungsoperanden zusammengeführt.

Am Schluss der lokalen Analyse berechnet das Verfahren für jeden Knoten, ob er vollständig ist. *DSNodes* für Argumente oder globale Variablen können hier noch nicht vollständig sein, da die Kontextinformationen von Aufrufen fehlen.

"Bottom-Up"-Phase Diese Phase verfeinert die bestehenden Graphen, indem sie Informationen von aufgerufenen Prozeduren zu den jeweiligen Aufrufern hoch-propagiert. Dazu wird für jede Prozedur der vorhandene *DSGraph* geklont und in den Graph des Aufrufers eingefügt. Die Knoten zu den formalen Argumenten der aufgerufenen Prozedur werden hierbei mit denen der aktuellen Argumente an den Aufrufstellen verbunden. Beim Klonen sind Stack-Objekte ausgenommen, da sie für den Aufrufer nicht erreichbar sind. Am Ende erfolgt erneut eine Prüfung auf Vollständigkeit der Knoten, aufgrund der neu gewonnen Informationen.

Eine wichtige Technik der "Bottom-Up"-Phase ist die Suche nach stark zusammenhängenden Komponenten (*Strongly Connected Components*, SCC) [ASU86]. Dabei kommt eine modifizierte Version von Tarjan's Algorithmus zum Einsatz, der hier auch indirekte Aufrufe (z. B. über Funktionszeiger) auflösen kann [Sed88]. Es ist damit möglich, Rekursionen innerhalb der Anwendung zu erkennen, um eine effektive Analyse zu gewährleisten.

"Top-Down"-Phase Die letzte Phase ist ähnlich zum "Bottom-Up"-Schritt, da ebenfalls *DSGraph*-Informationen zusammengeführt werden. Hier propagieren die Aufrufer die Graphen allerdings zu den aufgerufenen Prozeduren. Damit können Knoten von mehreren Argumenten vereint werden, falls diese auf dieselben Speicherobjekte zeigen. Außerdem ist es in diesem Schritt möglich, die Vollständigkeit der Konten zu Argumenten zu setzen. Falls noch immer unvollständige Knoten vorhanden sind, so müssen diese durch externe Aufrufe entstanden sein.

4.3 Umsetung (Architektur)

In diesem Abschnitt werden die wichtigsten Analysen und Transformationen präsentiert, welche für den Ablauf der Anwendung erstellt wurden. Zur Ermittlung der Ablaufinformationen wird dabei die Instrumentierung des Zwischencodes, die zugehörige Laufzeitbibliothek sowie die Umsetzung des Lesevorgangs erläutert. Der letzte Punkt beschreibt schließlich die Umsetzung der Analyse von Parallelisierungspotential.

4.3.1 Instrumentierung des Zwischencodes

Zur Ermittlung der Ablaufinformationen, wie in Abschnitt 3.5 beschrieben, muss der vorhandene Zwischencode erweitert werden. Nach der Ausführung der Anwendung sollen Informationen

bezüglich Aufrufhierarchie sowie Laufzeiten von Prozeduren in Form einer externen Datei vorliegen. Dazu rufen die hinzugefügten Anweisungen Methoden einer Laufzeitbibliothek (siehe Abschnitt 4.3.2) auf, welche die übergebenen Informationen protokollieren und schließlich abspeichern.

Anweisungen zur Erzeugung des Aufrufgraphen Der (dynamische) Aufrufgraph einer Anwendung soll die Hierarchie der Prozeduraufrufe einer Ausführung abbilden. Zur Ermittlung ist es wichtig, dass die Laufzeitbibliothek über jeden Aufruf benachrichtigt wird. Außerdem muss ihr mitgeteilt werden, dass sie die gesammelten Daten in eine externe Datei schreiben kann, sobald die Anwendung beendet ist und damit keine weiteren Aufrufe mehr erfolgen.

Grundsätzlich wäre es zur Berücksichtigung aller Prozeduraufrufe ausreichend, den Aufruf einer Bibliotheksmethode vor jeder vorhandenen Aufrufinstruktion im Zwischencode einzufügen. Der Methode müssen zur Identifikation zwei Argumente übergeben werden: eine eindeutige Nummer der Aufrufstelle sowie der Name der aufgerufenen Prozedur.[3] Allerdings treten dabei zwei Schwierigkeiten auf, die den Aufbau des Aufrufgraphen verhindern. Zum einen lassen sich externe Prozeduren erst zur Laufzeit bestimmen. Dadurch kann der Name der Prozedur nicht durch die Instrumentierung in den Zwischencode eingefügt werden. Das zweite Problem entsteht durch rekursive Aufrufe von Prozeduren. Die Laufzeitbibliothek kann in dem Fall nicht bestimmen, von welcher Instanz der Prozedur ein konkreter Aufruf stammt.

Zur Vermeidung der genannten Schwierigkeiten werden drei unterschiedliche Methodenaufrufe der Laufzeitbibliothek verwendet. Ein Aufruf (`llvm_call($Aufrufstelle, $Prozedur)`) findet, wie vorhin erwähnt, vor jeder Aufrufinstruktion statt. Damit die Bibliothek externe Prozeduren zu der eben verwendeten Aufrufstelle zuordnen kann, wird eine zusätzliche Methode (`llvm_called($Prozedur)`) am Anfang jeder Prozedur aufgerufen. Der Aufruf der letzten Bibliotheksmethode (`llvm_callEnd($Aufrufstelle, $Prozedur)`) erfolgt jeweils nach einer Aufrufstelle.[4] Ähnlich wie bei geklammerten Ausdrücken ist es damit möglich, die genaue Verschachtlung der Prozeduraufrufe zu rekonstruieren. Abbildung 4.2 zeigt ein Beispiel für instrumentierten (vereinfachten) Zwischencode.

In dem Zwischencode sind zwei Aspekte besonders hervorzuheben. Prozedur `A` ruft die virtuelle Prozedur `V` auf. Dies kann erst zur Laufzeit bestimmt werden, indem nach dem Bibliotheksaufruf `llvm_call(A2, ext)` der Aufruf `llvm_called(V)` folgt. Der Name der Prozedur wird der Aufrufstelle zugeordnet. Der andere Aspekt betrifft die Rekursion `A`→`V`→`A`. Die Analyse muss entscheiden können, von welcher konkreten Instanz der Prozedur `A` ein Prozeduraufruf stattgefunden hat. Dies ermöglicht der Methodenaufruf `llvm_callEnd(V1, A)` in `V`, durch den erkannt wird, dass die zweite Instanz von `A` verlassen wurde.

Die Laufzeitbibliothek muss die protokollierten Informationen zum Ende der Anwendung konsolidieren und in eine Ausgabedatei schreiben. Dazu erfolgt zu Beginn der Einstiegsprozedur ein zusätzlicher Aufruf der Bibliothek (`llvm_build_and_write()`). Die entsprechende Methode initiiert den Konsolidierungsvorgang über die Funktion `int atexit ( void ( *function`

[3] In LLVM Zwischencode besitzt jede Prozedur einen eindeutigen Namen.
[4] Im Falle von `invoke`-Instruktionen können zwei Basisblöcke als Nachfolger existieren.

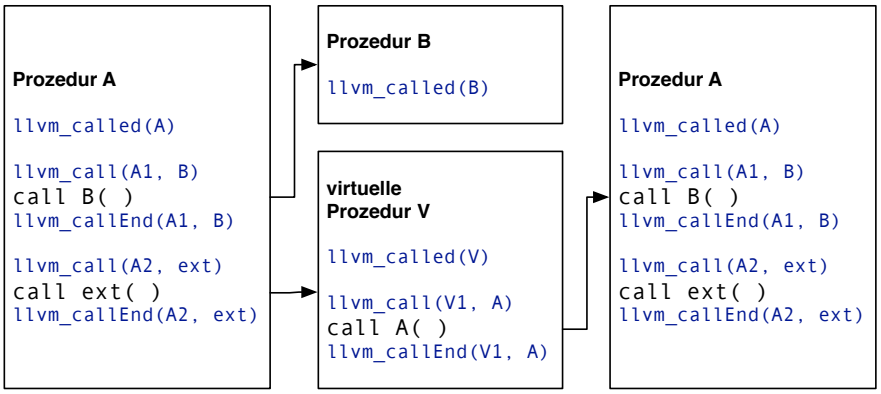

Abbildung 4.2: Instrumentierung zum Aufbau des Aufrufgraphen

`) (void) )` der Standardbibliothek von C. Dadurch startet der Vorgang sicher am Schluss der Programmausführung.

Laufzeitmessung Leider bietet LLVM keine Möglichkeit, um Laufzeitmessungen durch dynamische Programmausführung zu erhalten. Es können lediglich Aufrufhäufigkeiten von Kontrollpfaden ermittelt werden. Es ist somit eine Eigenimplementierung der Laufzeitmessung erforderlich, wobei keine zusätzlichen Anweisungen im Zwischencode eingfügt werden müssen. Mit den vorhandenen Bibliotheksaufrufen hat die Analyse die Information, welche Prozeduren an welchen Stellen aufgerufen werden. Damit ist es Aufgabe der Analysebibliothek, die aktuellen Zeitpunkte vom System zu ermitteln. Dies muss jeweils beim Einstieg sowie beim Verlassen der Prozedur geschehen. In der Implementierung werden zur Laufzeitmessungen "Performance Counter" mit Hilfe der PAPI-Schnittstelle (*Performance Application Programming Interface*) abgefragt.

4.3.2 Laufzeitbibliothek

Sobald der Zwischencode durch Bibliotheksaufrufe ergänzt wurde, kann er für die Ermittlung von Ablaufinformationen übersetzt und auf der Zielarchitektur ausgeführt werden. Die Bibliothek hat nun zunächst die Aufgabe, übergebene Informationen zu den Aufrufen in eine vorgegebene Datenstruktur zu schreiben. Für jede Aufrufstelle werden folgende Informationen verwaltet:

- Eine eindeutige **ID** zur Identifikation der Aufrufstelle innerhalb der Anwendung.
- Den **Namen** der zugehörigen Prozedur.
- Einen **Zähler**, welcher die Aufrufhäufigkeit durch die Aufrufstelle beschreibt.
- Die Summe der **Ausführungszeiten** aller Aufrufe durch die Aufrufstelle.
- Ein Kennzeichen, das angibt ob eine **Messung** der Ausführungszeit momentan für den Prozeduraufruf aktiv ist.

Jede Instanz der Datenstruktur hat einen Zeiger auf die nächste Aufrufstelle innerhalb derselben Prozedur. Außerdem existiert ein Zeiger auf die besuchten Aufrufstellen der aufgerufenen Prozedur in Form der verketteten Liste. Um die Hierarchie berücksichtigen zu können, enthält eine globale Variable stets einen Zeiger auf die aktuell besuchte Aufrufstelle.

Zur besseren Erläuterung zeigt die Abbildung 4.3 ein Beispiel für eine befüllte Datenstruktur der Laufzeitbibliothek. Die Einstiegsprozedur (`main`) hat drei Aufrufstellen (`f`, `g` und `h`). Die zugehörige Struktur besitzt somit einen Zeiger auf die verkettete Liste dieser Aufrufstellen. Die aufgerufenen Prozeduren der Aufrufstellen von `f` und `g` besitzen, im Gegensatz zu der Prozedur von Aufrufstelle `h`, weitere Aufrufinstruktionen.

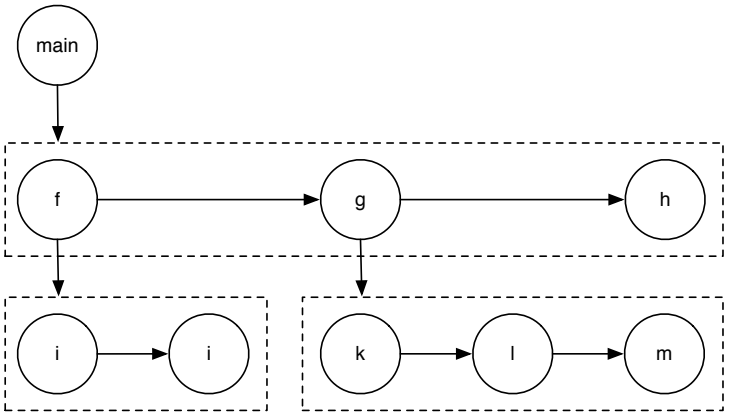

Abbildung 4.3: Beispiel zur Datenstruktur der Laufzeitbibliothek

Wie in Abschnitt 4.3.1 erwähnt, werden die Ablaufinformationen durch vier Methodenaufrufe innerhalb der Anwendungen ermittelt. Die folgende Übersicht erläutert deren Aufgaben:

1. **Bevorstehender Prozeduraufruf** (`llvm_call($Aufrufstelle, $Prozedur)`)
   Falls noch keine Instanz der Datenstruktur für die betroffene Aufrufstelle existiert, wird eine neue erzeugt. Es ist möglich, dass der Name der Prozedur zu diesem Zeitpunkt noch nicht bekannt ist (siehe nächster Punkt). Die ID der Aufrufstelle wird eingetragen, die Anzahl der Aufrufe inkrementiert, das Kennzeichen zur Messung auf "wahr" gesetzt und der Zähler durch eine PAPI-Methode gestartet. Außerdem wird die verkettete Liste der Aufrufstellen des Aufrufers erweitert und der globale Zeiger auf die aktuelle Aufrufstelle gesetzt.

2. **Prozedur wurde aufgerufen** (`llvm_called($Prozedur)`)
   Hier wird der Prozedurname der entsprechenden Datenstruktur-Instanz geändert, falls dieser durch späte Bindung noch nicht beim Aufruf bekannt war.

3. **Prozeduraufruf wurde beendet** (`llvm_callEnd($Aufrufstelle, $Prozedur)`)
   Die Zeitmessung wird beendet und der Mittelwert kann durch die bereits ermittelten (falls vorhandenen) Daten berechnet werden. Zusätzlich wird der globale Zeiger auf die vorherige Aufrufstelle gesetzt. Dadurch kann die Hierarchie erzeugt werden, selbst wenn Rekursion vorliegt.

4. **Programm wird beendet** (`llvm_build_and_write()`)
   Falls noch Zeitmessungen aktiv sind, werden diese beendet und wie im Schritt zuvor für die Summenbildung verwendet. Zuletzt konsolidiert die Bibliothek die Daten und schreibt sie in eine Ausgabedatei. Als Format wird *dot* des *Graphviz* Paket verwendet. Damit erreicht man neben einer standardisierten Struktur auch eine graphische Ausgabemöglichkeit.

4.3.3 Einlesen der Ablaufinformationen

Damit die Ablaufinformationen für die Analyseanwendung zur Verfügung stehen, müssen sie aus der erstellten externen Datei gelesen werden. Für diesen Zweck wurde eine LLVM-Analyse (*ModulePass)* erstellt, die eine Schnittstelle für den Rest der Analyseanwendung bereitstellt. Abbildung 4.4 zeigt das Klassendiagramm der Analyse. Zur besseren Übersichtlichkeit enthält das Diagramm lediglich einen Teil der Attribute und Methoden.

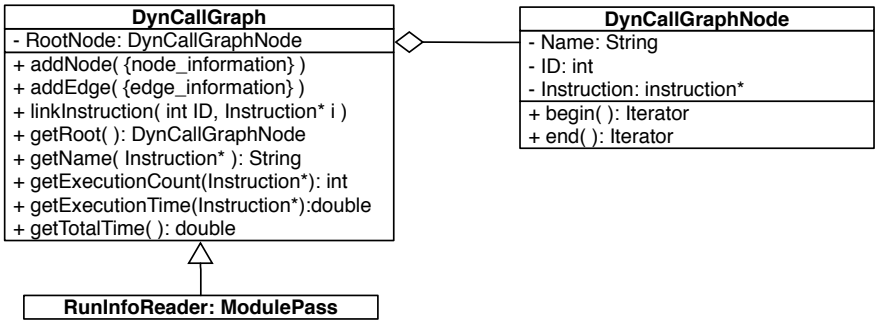

Abbildung 4.4: UML-Klassendiagramm der Import-Analyse für Ablaufinformationen

Die Klasse `DynCallGraph` repräsentiert den Aufrufgraphen innerhalb der Anwendung mit Instanzen der Klasse `DynCallGraphNode` als Knoten. Zu Beginn der Analyse werden die Daten von der Datei gelesen und über die Methoden `addNode()` sowie `addEdge()` zum Graphen hinzugefügt. Für jede Aufrufstelle wird durch die Methode `linkInstruction()` ein Zeiger auf die Instruktion des geladenen LLVM-Moduls mit dem eingefügten Knoten verknüpft. Dieser Schritt ist sehr wichtig, da hier die Verbindung zwischen dem dynamischen Aufruf und der statischen Informationen der LLVM-Analyse erstellt wird.

Durch die Methode `getRoot()` der Analyse erhält man den Knoten für die Einstiegsprozedur. Über Iteratoren ist es möglich, sämtliche Aufrufstellen eines Knotens zu ermitteln. Die Ablaufinformationen je Aufrufinstruktion können aber auch direkt über die Analyse-Schnittstelle abgerufen werden.

4.3.4 Analyse des Parallelisierungspotentials

Nach dem Einlesen der Ablaufinformationen kann das Parallelisierungspotential von Prozeduraufrufen ausgewertet werden. Für diesen Zweck wurde eine LLVM-Analyse entwickelt, die notwendige Komponenten (Datenstrukturanalyse, Ablauf- und Debug-Informationen) lädt und damit die Berechnungen durchführt. Um interprozedurale Auswertungen zu ermöglichen, erbt die

Analyseklasse von der Klasse `ModulePass`. Dadurch erhält man Zugriffe auf Informationen über sämtliche Prozeduren, Basisblöcke und Instruktionen einer Benutzeranwendung.

Der Code in Algorithmus 1 beschreibt den Ablauf der Analyse. Dabei sind zunächst die beiden (überschriebenen) Methoden `run_on_module` und `print` zu beachten, da diese die zeitliche Reihenfolge vorgeben. Erstere wird von der LLVM-Architektur zu Beginn der Analyseanwendung ausgeführt. Ein Aufruf der Methode `print` erfolgt, falls der Benutzer explizit eine Ausgabe der Ergebnisse anfordert.

Algorithmus 1 ParPotPass

```
 1: procedure run_on_module
 2: root_node ← root node of callgraph
 3: depGraph ← analyzeDependencies(root_node)
 4: runInfo ← getRunInformation( )
 5: result ← calculatePotential(depGraph, runInfo)
 6: end procedure

 7: procedure print
 8: dbgInfo ← getDebugInformation( )
 9: printResult(result, dbgInfo)
10: end procedure

11: function analyzeDependencies (node): DependenceGraph
12: depGraph_node ← NULL
13: for all set of analyzable call sets cSets_node do
14:    add checkControlFlow(set) to depGraph_node
15:    add checkAliasAnalysis(set) to depGraph_node
16:    add checkGlobals(set) to depGraph_node
17:    add checkDominators(set) to depGraph_node
18: end for
19: for all child of called procedures Calls_node do
20:    add analyzeDependencies(child) to depGraph_node
21: end for
22: return depGraph_node
23: end function
```

Im ersten Schritt werden Abhängigkeiten und Dominatorbeziehungen zwischen Prozeduraufrufstellen berechnet und für die spätere Auswertung in Abhängigkeitsgraphen eintragen. Um sämtliche Prozeduren für die Analyse in Betracht zu ziehen, erfolgt eine Iteration über den Aufrufgraph mittels Tiefensuche, beginnend bei der Einstiegsprozedur. Diese wird in Zeile 2 durch einen Aufruf von `getRoot()` der Klasse `RunInfoReader` (siehe Abschnitt 4.3.3) ermittelt. Anschließend startet die Methode `analyzeDependencies()` in Zeile 3 die rekursive Analyse von Aufrufstellen. Für jedes Paar (Ausdruck $cSets_{node}$ in Zeile 13) werden 4 verschiedene Analysemethoden angewendet:

1. **Kontrollabhängigkeiten suchen** (`checkControlFlow()`)
   In dieser Methode werden Kontrollabhängigkeiten zwischen den übergebenen Aufrufstellen gesucht. Dazu verfolgt die Analyse den Datenfluss von Rückgabewerten und Zeigerargumenten über *Def-Use*-Ketten. Ob ein Prozeduraufruf die Werte eines formalen Parameters ändern kann, wird mit Hilfe von interprozeduraler Analyse (Datenstrukturanalyse) ermittelt. Falls auf dem Pfad zwischen den Aufrufinstruktionen Kontrollstrukturen vorkommen, so wird eine Kontrollabhängigkeit in den Abhängigkeitsgraph der aufrufenden Prozedur eingefügt.

2. **Interprozedurale Zeigeranalyse** (`checkAliasAnalysis()`)
   Mit Hilfe der Datenstrukturanalyse wird ein Paar von Aufrufstellen bezüglich Zugriff auf gemeinsame Speicherbereiche durch Zeigerargumente geprüft. Dabei liefert die Analyse für jede Prozedur einen Graph mit einer Menge an Speicherobjekten (*DSNodes*), auf die innerhalb der Prozedur zugegriffen wird. Zu jedem Objekt ist bekannt, ob eine Modifikation (*mod*) oder eine Leseoperation (*ref*) stattfindet. Falls die Schnittmenge von Paaren nicht leer ist, so besteht (bei mindestens einem schreibenden Zugriff) eine Datenabhängigkeit zwischen den entsprechenden Prozeduraufrufstellen. Diese wird in den Abhängigkeitsgraph eingetragen.

3. **Datenabhängigkeit durch globale Variablen suchen** (`checkGlobals()`)
   Die Suche nach Datenabhängigkeiten durch globale Variablen erfolgt ebenfalls durch ein interprozedurales Verfahren. Für eine Prozedur werden sämtliche Instruktionen nach Zugriffen auf globale Variablen überprüft. Hier unterscheidet auch diese Analyse zwischen Lese- und Schreiboperationen.

   Bei Aufrufinstruktionen werden rekursiv die zugehörigen Prozeduren untersucht und die Ergebnisse hoch-propagiert. Sollte es sich um einen externen Aufruf handeln, so wird zunächst versucht, den Namen der Prozedur über den dynamischen Aufrufgraphen zu ermitteln. Falls dies nicht klappt, so handelt es sich um einen unauflösbaren Aufruf einer externen Bibliothek. Es wird hier davon ausgegangen, dass keine Abhängigkeit bezüglich globaler Variablen vorliegt. Wenn der Name der aufgerufenen Prozedur ermittelt werden konnte, so wird die entsprechende Referenz aus der modulweiten Prozedurliste ermittelt.

   Das Ergebnis dieser Analyse ist eine Menge an Zugriffsinformationen bezüglich globaler Variablen. Am Ende werden, wie bei der Zeigeranalyse, die Schnittmengen verglichen und gegebenenfalls der Abhängigkeitsgraph ergänzt.

4. **Dominatorbeziehungen prüfen** (`checkDominators()`
   Neben den reinen Abhängigkeiten spielt noch ein weiteres Kriterium eine Rolle für die Parallelisierung von Aufrufstelen. Falls zwei Aufrufinstruktionen in einer Ausführung der übergeordneten Prozedur nicht beide verwendet werden, so erzielt eine *Fork-Join*-Parallelisierung keinen Laufzeitgewinn. Dieser Fall kann eintreten, wenn sich die Instruktionen in unterschiedlichen Kontrollpfaden befinden. Die Methode `checkDominators()` untersucht für sämtliche Paare von Aufrufstellen deren Dominator- bzw. Postdominatorbeziehung. Wenn Instruktion a die Instruktion b dominiert und gleichzeitig b die Instruktion a postdominiert, so müssen sie stets gemeinsam ausgeführt werden. Andernfalls liegen die

Prozeduren auf unterschiedlichen Kontrollpfaden. Das sieht die Anwendung als Indiz für geringeres Parallelisierungspotential.

Nachdem die Abhängigkeiten ermittelt wurden, berechnet die Analyse das Parallelisierungspotential. Für jedes Paar werden die Abhängigkeiten der Graphen sowie die Ablaufinformationen durch die geladene Analyse betrachtet und anhand der Formel für η (siehe Abschnitt 3.4) berechnet. Die Anwendung fügt jedes Ergebnis in eine Liste ein, die absteigend nach dem ermittelten Wert des Parallelisierungspotentials sortiert wird. Die in der Implementierung benutzten Werte für c_t (t = Abhängigkeitstyp) lauten:

$c_{Echte\ Abh.} = 3$

$c_{Ausgabeabh.} = 1$

$c_{Umgekehrte\ Abh.} = 1$

$c_{Kontrollabh.} = 2$

$c_{Keine\ Dominatorbez.} = 1$

Am Ende der Analyse erfolgt die Ausgabe der Ergebnisliste, wobei die Bezeichner des Zwischencodes durch diejenigen der Debuginformationen ausgetauscht werden. Man muss allerdings beachten, dass die Anwendung lediglich ein spekulatives Ergebnis liefert. Die Informationen über Vollständigkeit der Datenstrukturanalyse werden nicht zur Auswertung verwendet. Außerdem nimmt die Analyse an, dass externe Bibliotheksmethoden keinen Zugriff auf Speicherobjekte innerhalb der Benutzeranwendung haben. Um konservative Analyse-Ergebnisse liefern zu können, müssen auch Lese- und Schreiboperationen auf externe Dateien berücksichtigt werden. Dies ist momentan nicht implementiert.

KAPITEL 5

Experimentelle Ergebnisse

Der wichtigste Aspekt dieser Arbeit ist die Unterstützung von Software-Entwicklern, durch Hinweise auf mögliche Parallelität in ihren sequenziellen Programmen. Im Gegensatz zu parallelisierenden Compilern lassen sich die Ergebnisse nicht allein durch quantitative Merkmale, wie die parallele Ausführungszeit, bewerten. Hohe Genauigkeit der Analysen sowie eine einfache Interpretierbarkeit der Resultate sind entscheidendere Merkmale für die Qualität der Anwendung. Eine exakte Messung dieser Kriterien kann, selbst mit intelligenten Heuristiken, nicht allgemeingültig sein. Oft spielen semantische Aspekte eine wichtige Rolle, welche die statische Analyse nicht berücksichtigen kann.

Die Bewertung der Resultate muss somit durch manuelle Prüfungen anhand der Quelltexte erfolgen. Dabei soll kontrolliert werden, ob die ermittelten Abhängigkeiten bezüglich Typ und Umfang zutreffen. Nicht berücksichtigte Abhängigkeiten zählen selbstverständlich auch zu Ungenauigkeiten der Analyse. Der berechnete Wert der maximalen Laufzeitersparnis kann durch prototypische Parallelisierung evaluiert werden.

Der erzeugte Zwischencode der Beispielanwendungen wurde mit Hilfe des vorhandenen Just-In-Time Compilers (JIT) ausgeführt. Dabei wurde die dynamische Kompilierung ("lazy compilation") deaktiviert, um die Laufzeitmessungen nicht durch unnötige Kompilierzeiten zu manipulieren.

Im Folgenden werden die Ergebnisse der im Rahmen dieser Arbeit durchgeführten Tests vorgestellt. Abschnitt 5.1 zeigt einen bekannten Mergesort-Algorithmus, welcher durch die Resultate parallelisiert wird. In Abschnitt 5.2 wird das Werkzeug für die Auswertung von Programmen der SPEC2006-Benchmarks verwendet.

5.1 Fallstudie: Mergesort

Für den Test der Analyseanwendung wurde zunächst ein Mergesort-Algorithmus verwendet [CLRS09]. Dieser basiert auf dem "Teile-und-Herrsche"-Paradigma und bietet daher eine gute Möglichkeit für den Einsatz von grobkörniger Parallelität.

5.1.1 Erläuterung des Algorithmus

Bei einem Mergesort-Algorithmus liegt der entscheidende Schritt bei dem Zusammenführen (merge) von zwei sortierten Teilsequenzen. Wie in Abbildung 5.1 zu sehen, arbeitet der hier verwendete Algorithmus auf zwei Teilsequenzen eines Arrays T. Die linke Sequenz $T[l_1..r_1]$ mit der Länge $n_1 = r_1 - l_1 + 1$ und die rechte Sequenz $T[l_2..r_2]$ mit der Länge $n_2 = r_2 - l_2 + 1$ werden kombiniert in einen Teil eines anderen Arrays $A[l_3..r_3]$ geschrieben.

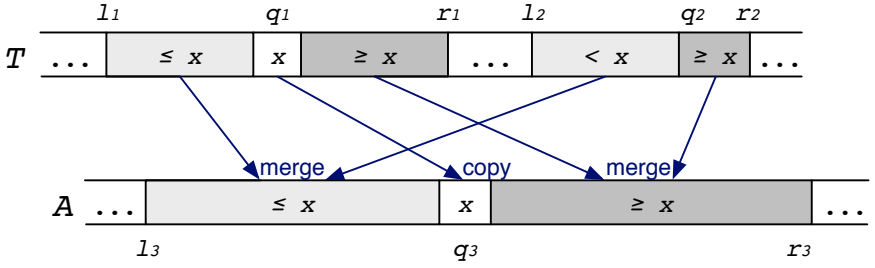

Abbildung 5.1: Mergesort-Algorithmus

Der Algorithmus ermittelt zunächst das mittlere Element $x = T[q_1]$ der Teilsequenz $T[l_1..r_1]$, wobei $q_1 = \lfloor (l_1 + r_1)/2 \rfloor$. Da die Teilsequenzen sortiert sind, stellt q_1 den Median von $T[l_1..r_1]$ dar. Anschließend erfolgt die Berechnung des Index q_2 der Teilsequenz $T[l_2..r_2]$, sodass beim Einfügen von x zwischen $T[q_2 - 1]$ und $T[q_2]$ die Sortierung erhalten bleibt. Diese Berechnung erfolgt durch binäre Suche. Schließlich werden die beiden Teilsequenzen $T[l_1..r_1]$ und $T[l_2..r_2]$zusammengeführt und in das Array $A[l_3..r_3]$ eingefügt. Dabei werden folgende Schritte durchgeführt:

1. Setze $q_3 = l_3 + (q_1 - l_1) + (q_2 - l_2)$.
2. Kopiere x in $A[q_3]$.
3. Führe rekursiv die Teilsequenzen $T[l_1..q_1 - 1]$ und $T[l_2..q_2 - 1]$ zusammen und speichere das Ergebnis in $A[l_3..q_3 - 1]$.
4. Führe rekursiv die Teilsequenzen $T[q_1 + 1..r_1]$ und $T[q_2..r_2]$ zusammen und speichere das Ergebnis in $A[q_3 + 1..r_3]$.

Der Basisfall tritt ein, wenn $n_1 = n_2 = 0$, da hier nichts mehr zusammengeführt werden kann.

Um die Auswertungen der Analyseanwendung verstehen zu können, wird in Listing 5.1 der Quelltext für die Prozedur `merge` gezeigt. Es erfolgt hier die beschriebene Sortierung und Zusammenführung der beiden Teilsequenzen aus T in A. Die Methode `exchange` tauscht die Werte der beiden übergebenen Speicheradressen aus. `binarySearch` führt die vorhin erwähnte binäre Suche nach dem Wert $T[q_1]$ im Array T zwischen l_1 und l_2 durch. Beide Methoden werden hier an dieser Stelle nicht gezeigt.

```
1  void merge(int *T, int l1, int r1, int l2, int r2,
2           int *A, int l3) {
3      int n1 = r1 - l1 + 1;
4      int n2 = r2 - l2 + 1;
5      if (n1 < n2) {          // stelle sicher, dass n1 >= n2
6          exchange(&r1, &r2);
7          exchange(&l1, &l2);
8          exchange(&n1, &n2);
9      }
10     if (n1 == 0)         // Basisfall
11       return;
12     else {
13         int q1 = floor((l1 + r1) / 2);
14         int q2 = binarySearch(T[q1], T, l2, r2);
15         int q3 = l3 + (q1 - l1) + (q2 - l2);
16         A[q3] = T[q1];
17         merge(T, l1, q1-1, l2, q2-1, A, l3);
18         merge(T, q1+1, r1, q2, r2, A, q3+1);
19     }
20 }
```

Listing 5.1: Mergesort - Prozedur `merge`

Die verwendete `mergeSort` Prozedur arbeitet auf zwei Arrays A und B, die als Zeigerargumente übergeben werden. Array A ist dabei das unsortierte Array als Eingabe. In Array B wird das sortierte Ergebnis geschrieben. Der Quelltext ist in Listing 5.2 ersichtlich.

```
1  void mergeSort(int *A, int l, int r, int *B, int s) {
2    int n = r - l + 1;
3    if (n == 1)
4     B[s] = A[l];
5    else {
6      int T[n];
7      int q = floor((l+r)/2);
8      int v = q - l + 1;
9      mergeSort(A, l, q, T, 1);
10     mergeSort(A, q+1, r, T, v + 1);
11     merge(T, 1, v, v+1, n, B, s);
12   }
13 }
```

Listing 5.2: Mergesort - Prozedur `mergeSort`

5.1.2 Ergebnisse der Analyse

Der Algorithmus wurde zum Test des Analysewerkzeugs mit einem ausreichend großen, absteigend sortierten Array aufgerufen. Dabei wurde die in Tabelle 5.1 gezeigte Ergebnisliste zurückgeliefert. Jeder Eintrag ist darin wie folgt aufgebaut:

1. **Aufrufer** Die erste Zeile zeigt in der linken Spalte den Namen der Prozedur, welche die Aufrufstellen enthält. Rechts erscheint die Ursprungsdatei der Prozedur. Das Verzeichnis und der Name wurden aus Debuginformationen ermittelt.

2. **Aufrufstellenpaar** In der zweiten Zeile erscheinen die beiden Aufrufstellen, welche analysiert wurden. Neben der Zeilennummer innerhalb der Datei hat der Benutzer auch Informationen über den Namen der aufgerufenen Prozeduren sowie die berechnete maximale Laufzeitersparnis bei paralleler Ausführung im Bezug auf die Gesamtlaufzeit der Benutzeranwendung.

3. **Abhängigkeiten** Die folgenden Zeilen listen vorhandene Abhängigkeiten zwischen den Aufrufstellen auf. Für jedes Speicherobjekt, das für die Abhängigkeit verantwortlich ist, existiert eine Zeile in der Liste. Die linke Spalte enthält den Namen der betreffenden Speicherobjekte. Im Falle von Kontrollabhängigkeiten zeigt die Analyse lediglich die Ausgangsvariable. Abhängigkeiten durch globale Variablen werden mit *(G)* gekennzeichnet. In der rechten Spalte erscheinen die gefundenen Typen von Abhängigkeiten (T = Echte Abhängigkeit; O = Ausgabeabhängigkeit; A = Umgekehrte Abhängigkeit; C = Kontrollabhängigkeit).

1	Prozedur: `mergeSort`	*../mergesort/mergesort.cpp*
	66: `mergeSort` - 67: `mergeSort`	max: [43.95 %]
	`B` → `B`	O
2	Prozedur: `mergeSort`	*../mergesort/mergesort.cpp*
	66: `mergeSort` - 68: `merge`	max: [4.63 %]
	`B` → `A`	T
3	Prozedur: `mergeSort`	*../mergesort/mergesort.cpp*
	67: `mergeSort` - 68: `merge`	max: [4.63 %]
	`B` → `A`	T
4	Prozedur: `merge`	*../mergesort/mergesort.cpp*
	53: `merge` - 54: `merge`	max: [2.315 %]
	`A` → `A`	O

Tabelle 5.1: Ergebnisse für mergeSort

Anhand der Ergebnisse wird deutlich, dass das meiste Parallelisierungspotential in den beiden rekursiven Aufrufen von `mergeSort` (Zeilen 9 und 10) liegt. Die Analyse unterscheidet nicht zwischen Zugriffen auf verschiedene Teile eines Arrays. Sie erkennt jedoch, dass der Wert des formalen Parameters `B` geschrieben, aber nicht gelesen wird. Die entdeckte Ausgabeabhängigkeit in Eintrag 1 kann ignoriert werden, da bei genauem Hinsehen beim ersten Aufruf von `mergeSort`

auf den Bereich $[l..q]$ und beim zweiten auf den Bereich $[q+1..r]$ zugegriffen wird.

Der zweite und dritte Eintrag der Ergebnisliste bezieht sich jeweils auf einen Aufruf der Prozedur `mergeSort` mit dem folgenden Aufruf der Prozedur `merge` in Zeile 11. Die Analyse erkennt korrekterweise eine echte Abhängigkeit durch die Übergabe von `T` an die beiden Prozeduren. `mergeSort` ändert in Zeile 4 den Wert des entsprechenden Arguments `B` und `merge` liest wiederum den Wert seines Arguments `A`. Beim vierten Eintrag wird ähnlich zu dem ersten eine Ausgabeabhängigkeit innerhalb der rekursiven Aufrufe von `merge` entdeckt.

5.1.3 Parallelisierung

Das Analysewerkzeug sagt eine potentielle Laufzeitersparnis von 43.95 % voraus, falls die beiden Aufrufe von `mergeSort` parallelisiert werden. Bei der Berechnung dieses Wertes wird die Summe der Laufzeit aller Aufrufe gewertet. Zur Kontrolle dürfen daher nur die jeweils ersten Aufrufe der Rekursion parallel zueinander ausgeführt werden. Da diese Lösung nicht von mehr als zwei Tasks profitiert würde man in der Praxis an dieser Stelle mehrere Rekursionsebenen für eine Parallelisierung betrachten. Denkbar wäre ein bestimmter Schwellwert der Arraygröße, bis zu dem die beiden Aufrufe parallel zueinander ausgeführt werden.

Die Parallelisierung erfolgte mit Hilfe von OpenMP, wobei der Aufruf von `mergeSort` in Zeile 9 innerhalb eines eigenen Tasks ausgeführt wurde. Durch eine Synchronisationsbarriere wurde sichergestellt, dass die Ergebnisse vor dem Aufruf von `merge` in Zeile 11 vollständig berechnet wurden. Die parallele Ausführung erzielte eine Laufzeitersparnis von 41.56 %. Die Abweichung zum berechneten Wert ist verständlich, da der Aufwand für Taskerzeugung und Synchronisation nicht berücksichtigt wird. Das Ergebnis zeigt dennoch, dass die Berechnung der Laufzeiten praxisgerechte Ergebnisse liefert.

5.2 Fallstudie: SPEC2006

Für die Auswertung der Analyse-Ergebnisse wurden zusätzlich verschiedene Programme der SPEC2006 Benchmark-Sammlung verwendet. Die Testumgebung von LLVM bietet für deren Ausführung bereits passende Makefiles. Diese mussten erweitert werden, damit neben den herkömmlichen Bitcode-Dateien auch Versionen inklusive Debuginformationen bereit stehen. Zusätzlich sorgen die Makefiles für die Einhaltung der notwendigen Schritte der Analyseanwendung. Dazu gehört neben der Instrumentierung auch die Ausführung der Anwendung zur Ermittlung von Ablaufinformationen.

Aufgrund drei verschiedener Fehlerfälle konnten nicht alle Tests erfolgreich abgeschlossen werden. Fehlende Unterstützung von Inline-Assembler-Befehlen des JIT-Compilers und Ausnahmen durch die Datenstrukturanalyse waren die häufigsten Ursachen. In zwei Fällen (403.gcc und 483.xalancbmk) kam es zum Stillstand der Analyse wegen Speicherengpässen.

Die Anylse-Ergebnisse der Testanwendungen zeigten teilweise große (potentielle) Laufzeiteinsparungen durch Parallelisierung einzelner Aufrufstellenpaare. Abbildung 5.2 stellt den Anteil der Zeitersparnis im Bezug auf die Gesamtlaufzeit dar. Hierbei wurde jeweils das Paar mit dem

maximalen Wert verwendet. Testfall 462.libquantum besitzt mit über 43% die größte theoretische Laufzeiteinsparung. Das bedeutet, dass zwei Prozeduraufrufe innerhalb derselben Prozedur existieren, wobei derjenige mit der kürzeren Laufzeit 43% der Gesamtlaufzeit der Anwendung benötigt.

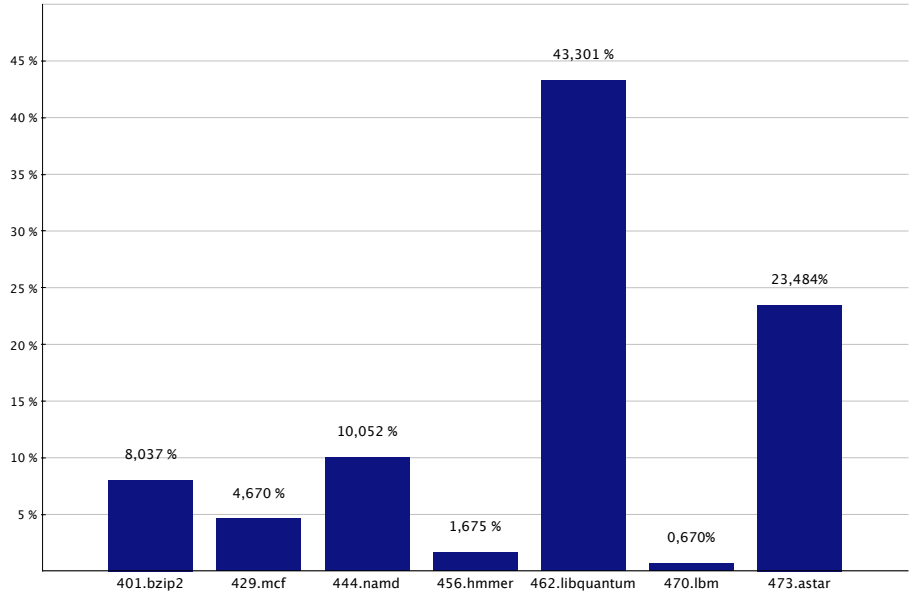

Abbildung 5.2: Maximale Laufzeiteinsparung durch ein Aufrufstellen-Paar (SPEC2006)

Im Folgenden werden zwei Testfälle näher betrachtet, um verschiedene Erkenntnisse aus den experimentellen Resultaten zu erläutern. Dazu werden Einträge aus der Ergebnisliste und (wenn nötig) zugehörige Codeabschnitte aus den SPEC2006-Benchmarks präsentiert.

5.2.1 401.bzip2

Der Benchmark testet die Geschwindigkeit einer Kompression und Dekompression von Daten. Tabelle 5.2 zeigt einen Teil der Ergebnisliste mit den Einträgen 1, 2 (höchstes kalkuliertes Parallelisierungspotential) und 11.
Die Tabelle zeigt einen deutlichen Unterschied der maximalen Laufzeiteinsparung zwischen dem ersten Eintrag (8.04%) und den folgenden Paaren (max. 0.40%). Die Analyse entdeckt zwischen den Aufrufen von `spec_compress` und `spec_uncompress` lediglich eine Ausgabeabhängigkeit aufgrund einer globalen Variable. In Abbildung 5.3 sieht man die Aufrufhierarchie, welche zu der Abhängigkeit durch die Variable `exitValue` führt. Wie im Abschnitt 2.4 erwähnt, kann ein Einsatz von temporären Variablen diese Art von Abhängigkeit aufheben.

Die beiden Aufrufstellen sind aber aufgrund einer anderen Abhängigkeit nicht parallelisierbar. Der Benchmark arbeitet bei der Kompression mit externen Dateien, welche in `spec_compress` geschrieben und in `spec_uncompress` gelesen werden. Die Analyseanwendung bietet jedoch keine Möglichkeit, Abhängigkeiten aufgrund von Dateizugriffen festzustellen.

1	Prozedur: `main`	*401.bzip2/src/spec.c*
	333: `spec_compress` - 353: `spec_uncompress`	max: [8.04 %]
	`(G) exitValue` → `(G) exitValue`	O
2	Prozedur: `BZ2_bzCompress`	*401.bzip2/src/bzlib.c*
	481: `handle_compress` - 503: `handle_compress`	max: [0.40 %]
	`s` → `s`	T
⋮		
11	Prozedur: `compressStream`	*401.bzip2/src/bzip2.c*
	463: `spec_fread` - 470: `BZ2_bzWriteClose64`	max: [0.06 %]
	`spec_fd` → `spec_fd`	T
	`nIbuf`	C
⋮		

Tabelle 5.2: Ergebnisse für 401.bzip2 (Auszug)

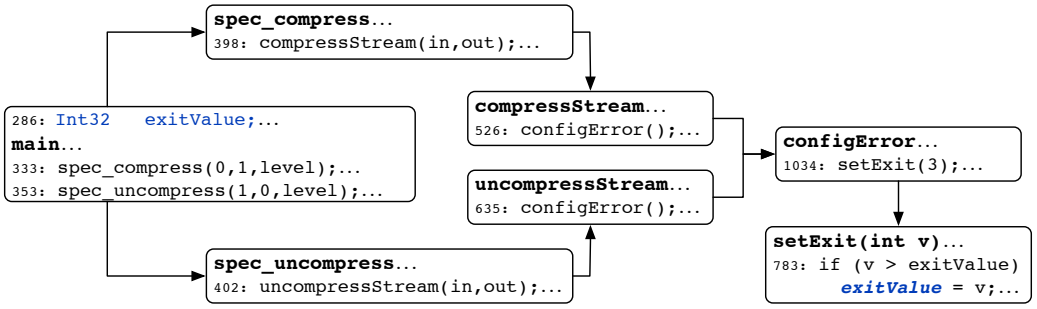

Abbildung 5.3: Ausgabeabhängigkeit bei 401.bzip2

Eintrag 2 der Ergebnisliste deutet auf eine echte Abhängigkeit zwischen zwei Aufrufstellen der Prozedur `BZ2_bzCompress` hin. Bei näherer Betrachtung (siehe Abbildung 5.4) fällt zunächst auf, dass der Wert von Variable `s` an mehreren Stellen der Aufrufkette verändert wird. In der gezeigten Prozedur `handle_compress` passiert das in Zeile 440 durch den Aufruf von `copy_input_until_stop` sowie in Zeile 444. Zudem entspricht der Name der Variable, welche die Abhängigkeit erzeugt (`s`), nicht derjenigen aus den Aufrufstellen (`strm`). Für eine komfortablere Verwendung der Analyseanwendung sollten diese beiden Punkte in einer späteren Version berücksichtigt werden.

Der dritte Eintrag von Tabelle 5.2 bringt durch die maximale Laufzeiteinsparung von 0.06% fast keinen Gewinn durch Parallelisierung. Allerdings werden durch das Beispiel zwei Dinge hervorgehoben. Zum einen gibt es an dieser Stelle zwei Speicherobjekte, die Abhängigkeiten hervorrufen. Zum anderen wird durch die Variable `nIbuf` eine Kontrollabhängigkeit erläutert.

Abbildung 5.5 präsentiert die Kontrollabhängigkeit durch `nIBuf`. Diese besteht zwischen dem Aufruf von `spec_fread` (siehe Makro `fread`) und `BZ2_bzWriteClose64`. Der Rückgabewert,

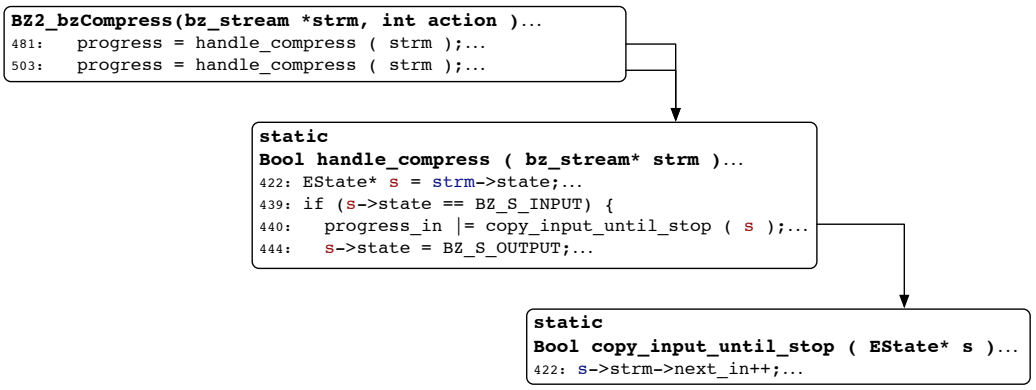

Abbildung 5.4: Echte Abhängigkeit bei 401.bzip2

der in Variable `nIbuf` gespeichert wird, erzeugt eine Kontrollabhängigkeit zum Prozeduraufruf von `BZ2_bzWrite` in Zeile 465. Hier wird wiederum schreibend auf den Zeiger `bzf` zugegriffen, was eine Datenabhängigkeit zum Aufruf in Zeile 470 erzeugt. Wie man sieht, muss hier die implementierte Auswertung von Kontrollabhängigkeiten neben der Verfolgung der *Def-Use*-Kette auch die Datenstrukturanalyse einsetzen. Ansonsten würde die Analyse nicht erkennen, dass der Aufruf von `BZ2_bzWrite` den dereferenzierten Wert von `bzf` modifiziert.

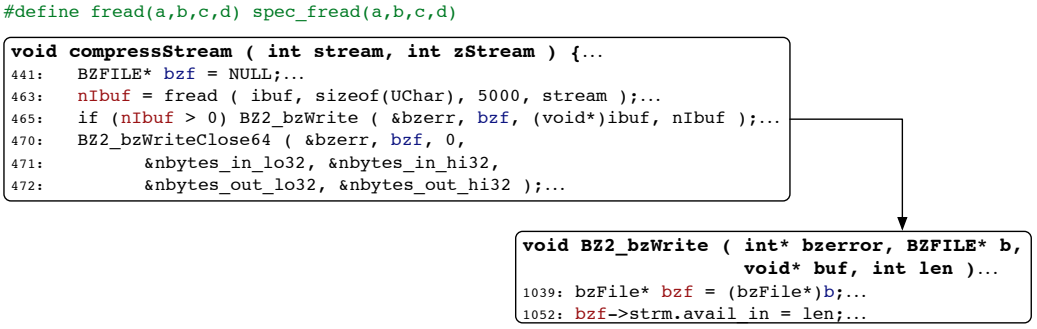

Abbildung 5.5: Kontrollabhängigkeit bei 401.bzip2

5.2.2 462.libquantum

Bei diesem Benchmark wurden die höchsten Ergebnisse bezüglich Laufzeitersparnis erzielt. Wie in Tabelle 5.3 zu sehen, hat der erste Eintrag der Liste eine maximale Ersparnis von 43.30%. Die Einträge 6 und 7 werden aufgelistet, um die Auswirkung der Bewertungsheuristik aufzuzeigen.

Fast die gesamte Laufzeit des Testprogramms wird in der Prozedur `mul_mod_n` benötigt. Eine Aufteilung der Aufrufe von `muln` und `muln_inv` wäre eine große Ersparnis hinsichtlich der Laufzeit. Jedoch besteht eine echte Abhängigkeit durch den formalen Parameter `reg`. Der Wert wird über die indirekten Aufrufe der Prozedur `quantum_toffoli` in Zeile 96 gelesen und in

1	Prozedur: `mul_mod_n`	*462.libquantum/src/omuln.c*
	77: `muln` - 81: `muln_inv`	max: [43.30 %]
	`reg` → `reg`	T
⋮		
6	Prozedur: `quantum_bmeasure`	*462.libquantum/src/measure.c*
	72: `quantum_frand` - 80: `quantum_prob_inline`	max: [6.90 %]
	`r` → `r`	C
7	Prozedur: `main`	*462.libquantum/src/shor.c*
	100: `quantum_exp_mod_n` - 104 `quantum_bmeasure`	max: [8.37 %]
	`qr` → `qr`	T
⋮		

Tabelle 5.3: Ergebnisse für 462.libquantum (Auszug)

Zeile 97 verändert. Ein beispielhafter Auszug der Aufrufhierarchie ist in Abbildung 5.6 gezeigt.

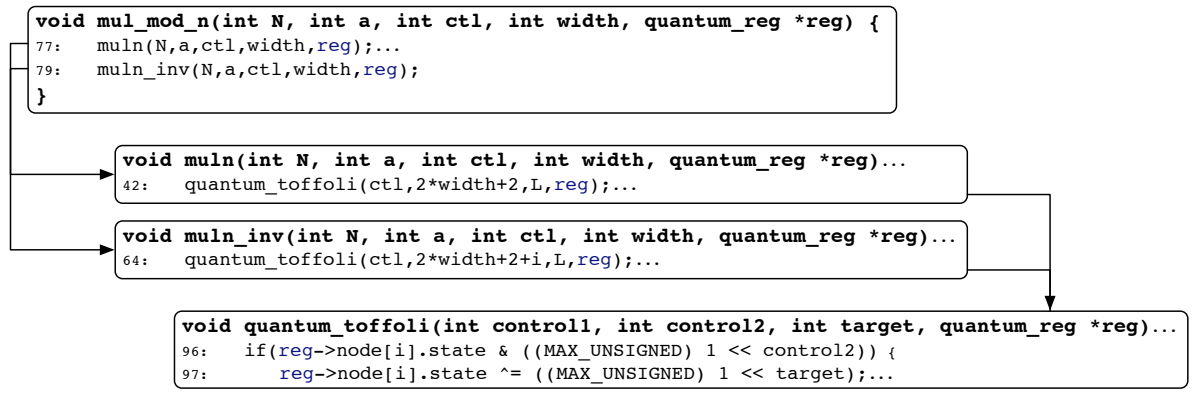

Abbildung 5.6: Echte Abhängigkeit bei 462.libquantum

Die restlichen Einträge der Ergebnistabelle weisen bis zu Ergebnis 5 ebenfalls echte Abhängigkeiten mit ähnlichen Konstellationen auf. Eine Parallelisierung ist ohne Wissen über die Struktur und Semantik der Anwendung schwer möglich.

In der verwendeten Analyseanwendung werden echte Abhängigkeiten für die Parallelisierung als gravierender betrachtet als Ausgabe- oder Kontrollabhängigkeiten (siehe Abschnitt 4.3.4). Daraus resultiert die Ordnung der Einträge 6 und 7 in der Tabelle 5.3. Obwohl der Eintrag 7 eine höhere maximale Laufzeitersparnis aufweist, wird er aufgrund der echten Abhängigkeit schlechter bewertet.

5.3 Zusammenfassung

Obwohl die manuelle Auswertung von Analyse-Ergebnissen der Anwendung aufwändig ist, wurden viele Rückschlüsse auf den praktischen Nutzen gezogen. Die Testläufe zeigten auch mögliche Stellen für Verbesserungen auf. Diese könnten in einer folgenden Version berücksichtigt werden.

Die statische Analyse erreicht in den Tests eine hohe Genauigkeit bei der Ermittlung von Abhängigkeiten, die über eine Vielzahl von Hierarchiestufen des Aufrufgraphen entdeckt werden konnten. Für die manuelle Prüfung der Ergebnisse wäre es in diesen Fällen sehr hilfreich, wenn exakte Kontextinformationen über verwendete Variablen und Argumente der Aufrufkette verfügbar wären. Beim Zugriff auf Arrays muss beachtet werden, dass die Analyse keine Unterscheidung zwischen einzelnen Feldern ermöglicht. Außerdem kann es aufgrund des spekulativen Vorgehens vorkommen, dass Abhängigkeiten in manchen Situationen nicht erkannt werden. In den Testläufen konnten solche Fälle nur bei Benchmark 401.bzip2 nachgewiesen werden. Hier besteht eine echte Abhängigkeit aufgrund von Dateizugriffen. Dieser Typ von Abhängigkeit wird von der Analyse nicht unterstützt.

Hinsichtlich Parallelisierung konnte nur bei dem Mergesort-Algorithmus eine Umsetzung durch *Fork-Join*-Technik durchgeführt werden. Die ermittelte Laufzeiteinsparung entsprach dem Wert des Analyseresultats. Bei den Benchmarks der SPEC2006-Sammlung wurden keine geeigneten Stellen für eine einfache Parallelisierung von Prozeduraufrufen entdeckt. Obwohl die Werte der maximalen Laufzeiteinsparung für eine Ausführung durch mehrere Tasks sprechen, kann man die vorliegenden Abhängigkeiten nicht ohne Weiteres ignorieren.

KAPITEL 6

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Verfahren zur Ermittlung des Parallelisierungspotential sequenzieller Programme vorgestellt. Durch den Einsatz verschiedener Analysen und Heuristiken werden günstige Stellen auf Prozedurebene ermittelt, die den Softwareentwickler bei der Parallelisierung bestehender Anwendungen unterstützen. Dazu wird der mögliche Laufzeitgewinn sowie der notwendige Aufwand in Form von Abhängigkeiten im Voraus berechnet und ausgegeben.

Eine Besonderheit des gewählten Ansatzes ist die Kombination aus dynamischer Ermittlung von Ablaufinformationen und statischer Abhängigkeitsanalyse. Die meisten existierenden Werkzeuge, wie der Intel Parallel Advisor oder CriticalBlue Prism, protokollieren Speicherzugriffe während der Laufzeit. Sämtliche Analysen erfolgen mit Hilfe dieser dynamischen Informationen. Das zu erwartende Ergebnis ist somit abhängig von dem konkreten Aufrufkontext. Außerdem erwarten die genannten Programme vom Benutzer, dass dieser die zu untersuchenden Prozeduren manuell bestimmt. Das hier präsentierte Verfahren nutzt hingegen interprozedurale Analysetechniken, um Abhängigkeiten zwischen Prozeduraufrufen festzustellen. Für sämtliche Paare von Aufrufstellen einer Prozedur werden sowohl Daten- als auch Kontrollabhängigkeiten gesucht. Anschließend berechnet eine Heuristik das resultierende Parallelisierungspotential für die Paare und hebt besonders günstige Stellen für eine Parallelisierung hervor.

Im Zuge dieser Arbeit wurde ein Werkzeug entwickelt, das auf dem LLVM Compiler-Framework aufbaut. Sämtliche Berechnungen arbeiten auf einer Zwischencode-Darstellung von LLVM. Prinzipiell können sie daher unabhängig von der verwendeten Programmiersprache eingesetzt werden. Ein wichtiger Bestandteil des Werkzeugs ist die Datenstrukturanalyse, welche als zusätzliches Werkzeug in die LLVM-Umgebung eingebunden werden kann. Sie bietet kontextsensitive, feldsensitive sowie interprozedurale Möglichkeiten für aufwändige Zeiger-Analysen. Da allein durch die Datenstrukturanalyse nicht sämtliche relevanten Abhängigkeiten entdeckt werden können,

wurden zusätzlich zwei weitere Analysen erstellt. Diese erkennen Kontrollabhängigkeiten und Datenzugriffe auf globale Variablen. Im Gegensatz zur Datenstrukturanalyse bieten die interprozeduralen Analysen, welche selbst erstellt wurden, eine Auflösung von später Bindung. Die Namen der Prozeduren, welche nicht zur Übersetzungszeit bestimmt werden konnten, können durch die dynamischen Ablaufinformationen ermittelt werden.

Es konnte durch experimentelle Tests gezeigt werden, dass das Werkzeug Abhängigkeiten über mehrere Hierarchiestufen des Aufrufgraphen bestimmen kann. Bei der Analyse eines Mergesort-Algorithmus wurde aufgrund der vielversprechenden Resultate eine Parallelisierung durchgeführt. Dabei konnte gezeigt werden, dass die Anwedung tatsächlich parallel abläuft und die berechnete Laufzeiteinsparung zutrifft. Auch bei der Ausführung der SPEC2006-Benchmarks wurden zum Teil hohe Laufzeiteinsparungen durch Parallelisierung von Prozeduraufrufen entdeckt. Aufgrund der gefundenen Abhängigkeiten konnte jedoch keine einfache Parallelisierung durchgeführt werden. Allerdings wurde deutlich, dass das erläuterte Verfahren gerade bei großen oder unbekannten Softwaresystemen sehr nützlich ist. Der Benutzer kann ohne viel Aufwand günstige Möglichkeiten für Parallelität auf Prozedurebene in seinen Anwendungen finden.

Die Resultate der Analysen ermöglichen meist eine gute Lokalisierung der gefundenen Abhängigkeiten. In manchen Fällen wurden jedoch Abhängigkeiten über mehrere Hierarchiestufen, Kontrollstrukturen und Definitionen hinweg entdeckt. Dadurch waren die exakten Ursachen von Abhängigkeiten schwer ersichtlich. Eine lohnende Erweiterung wäre somit die Anzeige von Aufrufkontexten, um Abhängigkeiten detailliert nachverfolgen zu können. Eine weitere Schwierigkeit bei der Auswertung von Ergebnissen entsteht aufgrund der fehlenden C++-Unterstützung der Datenstrukturanalyse. Es lassen sich zwar auch hier die Abhängigkeiten ermitteln, jedoch liefert die DSA bei Zugriffen auf Objektattribute nur die jeweilige Instanz als Speicherobjekt zurück. Eine exakte Lokalisierung der Ursache ist somit schwer möglich.

Der Ansatz zur Unterstützung von Entwicklern kann prinzipiell mit jeder Art von grobgranularer Task-Parallelität erweitert werden. Es liegt somit auf der Hand, Schleifen als eigenständige Tasks zu betrachten. Einerseits existieren viele Anwendungsgebiete, bei denen die Ausführung von Schleifen einen großen Anteil der Gesamtlaufzeit ausmachen. Andererseits hat die Forschung in dem Bereich bereits große Fortschritte gemacht. Arbeiten zu dem Thema existieren mittlerweile auch für das LLVM-Framework [Gro11].

Abschließend kann man betonen, dass statische Analysen in der Lage sind, den Entwicklern bei der Parallelisierung zu helfen. Um jedoch das volle Potential von Multicore- und zukünftigen Manycore-Prozessoren ausnutzen zu können, muss Parallelität von Beginn an beachtet werden. In jeder Entwicklungsphase von Software können Restriktionen für mögliche Parallelität geschaffen werden. Manche Restriktionen können durch Werkzeuge, wie das in dieser Arbeit entwickelte, entdeckt und entfernt werden. Ein besserer Ansatz ist jedoch die Berücksichtigung von Parallelität bereits in frühen Phasen der Software-Entwicklung.

Literaturverzeichnis

[Ama95] S. P. Amarasinghe. The SUIF Compiler for Scalable Parallel Machines. In *Parallel Processing for Scientific Computing*, 1995.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban94] Utpal Banerjee. *Loop Parallelization*. Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[Cri11] CriticalBlue. Criticalblue Prism. http://www.criticalblue.com/, 2011.

[DRV00] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[FL11] Paul Feautrier and Christian Lengauer. The Polyhedron Model. In *Encyclopedia of Parallel Computing*. Springer-Verlag, 2011.

[Gro11] Tobias Christian Grosser. Enabling Polyhedral Optimizations in LLVM. Diplomarbeit, Universität Passau, 2011.

[Int11] Intel. Intel Parallel Advisor 2011. http://software.intel.com/en-us/articles/intel-parallel-advisor/, 2011.

[Lan92] William Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, 1992.

[Lat02] Christopher Arthur Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[Lat05] Christopher Arthur Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, 2005.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[OM08] Cosmin E. Oancea and Alan Mycroft. Languages and Compilers for Parallel Computing. chapter Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS), pages 156–171. Springer-Verlag, 2008.

[SCZM05] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede Approach to Thread-Level Speculation. *ACM Trans. Comput. Syst.*, 23:253–300, August 2005.

[Sed88] Robert Sedgewick. *Algorithms, 2nd Edition.* Addison-Wesley, 1988.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wortwörtlich oder sinngemäß übernommenen wurden, als solche gekennzeichnet sind, sowie dass ich diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 8. August 2011 *Andreas Johannes Wilhelm*