Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

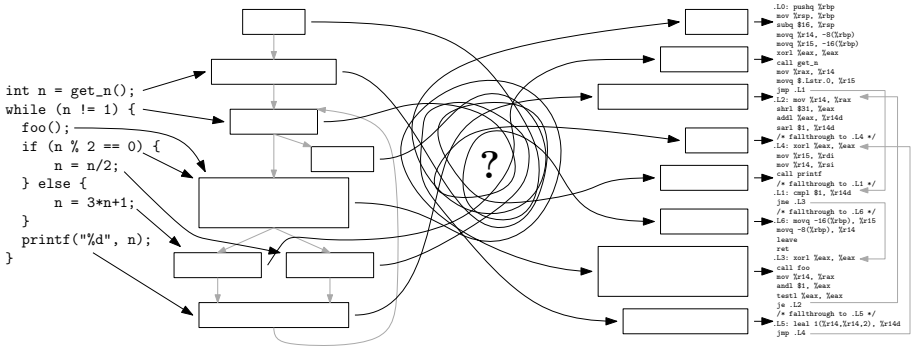
Lehrstuhl Prof. Dr.-Ing. Snelting

Optimierung der Grundblockanordnung

Bachelorarbeit von

Christoph Breisacher

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Andreas Fried
Abgabedatum:	18. April 2019

Zusammenfassung

Bei der Grundblockanordnung müssen die Grundblöcke eines Programmes, für dessen Übersetzung in die Maschinensprache, in eine schnell und effizient ausführbare Reihenfolge gebracht werden. Diese Arbeit stellt einen Ansatz zur Verbesserung der Grundblockanordnung in libFirm vor. Im Gegensatz zu klassischen Ansätzen bezieht dieser nicht nur die Anzahl der Fall-throughs, sondern auch alle weiteren Sprünge des Programmes mit ein. Im Vergleich zur bisherigen Implementierung der Grundblockanordnung in libFirm, erzeugt der in dieser Arbeit implementierte Ansatz bis zu 7,6 % schnellere Programme.

With basic block ordering, the basic blocks of a program must be brought into a fast and efficiently executable sequence, to allow for its translation into machine language. This thesis presents a approach for improving the basic block ordering of libFirm. Contrary to classical approaches not only the number of fall-throughs, but also all other jumps of the program are considered. Compared to the existing implementation of basic block ordering in libFirm, the approach implemented in this thesis generates up to 7.6 % faster programs.

Inhaltsverzeichnis

1 Einführung **7**

2 Grundlagen und Verwandte Arbeiten **9**
2.1 Compiler 9
2.2 Kontrollflussgraphen 10
2.3 Programmausführung 11
2.4 Grundblockanordnung 12
2.5 Verwandte Arbeiten 14
2.6 libFirm 16

3 Entwurf und Implementierung **17**
3.1 Grundblockanordnung in libFirm 17
3.2 ExtTSP 19
3.3 ExtTSP-Algorithmus 22
3.4 Implementierung 24
3.4.1 Voraussetzungen 25
3.4.2 Optimierungen 26

4 Evaluation **29**
4.1 Testbedingungen 29
4.2 Auswirkungen auf erzeugte Programme 30
4.3 Auswirkungen auf die Compilerlaufzeit 33
4.4 Auswirkungen des Parameters K 34

5 Fazit und Ausblick **39**

1 Einführung

Seit ihrer Entstehung in den 1940er Jahren haben Computer Einzug in unsere Welt gehalten. Mit den Jahren gewannen sie stetig an Bedeutung. Es gibt heutzutage kaum noch einen Lebensbereich, der nicht direkt oder zumindest indirekt von ihnen abhängig ist. Hand in Hand mit ihrer wachsenden Verbreitung, stiegen allerdings auch die Anforderungen an Leistungsfähigkeit und Effizienz der Computer. Dies führte über die Jahre zu enormen Fortschritten in der Computertechnik. Mit jeder Generation von Prozessoren wurden diese schneller, effizienter und somit auch komplexer. Moderne Prozessoren sind daher mittlerweile so komplex, dass es sehr schwierig ist, effiziente Programme für sie zu schreiben. Deshalb ist es heute üblich, Programme nicht mehr in der Maschinensprache des Prozessors selbst, sondern in einer einfacheren Hochsprache, zu formulieren. Damit diese Programme von einem Prozessor ausgeführt werden können, ist eine Übersetzung in die Maschinensprache des Prozessors notwendig. Diese Übersetzung wird in der Regel durch ein Computerprogramm übernommen, den sogenannten „Compiler“. Dessen Aufgabe, die Abbildung von Hochsprachen-Programmen auf die komplexe und doch beschränkt ausdrucksfähige Maschinensprache, ist schwierig. In vielen Fällen beinhaltet sie sogar besonders komplexe Teilprobleme aus einer Klasse von Problemen, die als NP-vollständig bezeichnet werden. So hat beispielsweise die Reihenfolge, in der die einzelnen Maschinensprachenbefehle in einem Programm stehen, Auswirkungen auf Geschwindigkeit und Effizienz, mit der das Programm durch den Prozessor ausgeführt werden kann. Für die Übersetzung zerteilen viele Compiler das übersetzte Programm in kleinere Abschnitte, welche als Grundblöcke bezeichnet werden. Diese können, in gewissem Umfang, getrennt voneinander übersetzt werden und finden sich auch in der Übersetzung - dem Maschinensprachenprogramm - wieder. Wie diese Grundblöcke im erzeugten Programm angeordnet sind, ist für die Funktionalität des Programmes unbeachtlich. Maßgeblich ist diese Anordnung jedoch für die Effizienz mit der das Programm durch den Prozessor ausgeführt werden kann. Hierfür sind die Programmsprünge, welche den korrekten Ablauf des Programmes zwischen den einzelnen Grundblöcken gewährleisten, verantwortlich. Diese können die Ausführungsgeschwindigkeit des Programmes beeinträchtigen, falls sie nicht vorhergesehen werden können, oder zu weit entfernten Grundblöcken springen. Denn in diesen Fällen benötigt der Prozessor Zeit, um erneut seine volle Geschwindigkeit zu erreichen, oder zu warten bis der angesprungene Programmteil aus dem langsamen Speicher geladen wurde. Daher ist es für das effiziente Anordnen der Grundblöcke essenziell die Sprünge zwischen diesen genauer zu betrachten. Ziel dieser Arbeit ist es, sich

mit dem Finden jener Grundblockanordnungen zu beschäftigen, die eine möglichst effiziente Ausführung durch Prozessoren erlauben.

libFirm ist eine am Karlsruher Institut für Technologie entwickelte Programmbibliothek, zum Schreiben von Compilern. Sie stellt Funktionalitäten bereit, die von vielen Compilern benötigt werden. Unter anderem enthält sie auch einen Algorithmus zum Finden von Grundblockanordnungen. Da dieser bislang recht simpel ist, besteht an dieser Stelle offenbar noch Optimierungspotential. In dieser Arbeit wird versucht, dieses Optimierungspotential zu nutzen und die Grundblockanordnung in libFirm zu verbessern. Hierfür wird ein neuer Algorithmus, aus dem Gebiet der „Profile-guided Optimizations“, vorgeschlagen und in libFirm implementiert.

In Kapitel 2 werden die, zum Verständnis dieser Arbeit nötigen, Grundlagen erläutert und sich mit verwandten Arbeiten zu diesem Thema auseinandergesetzt. Des Weiteren wird das Problem der Grundblockanordnung motiviert und genauer betrachtet. Kapitel 3 stellt den im Rahmen dieser Arbeit implementierten Ansatz zur Optimierung der Grundblockanordnung vor. Zunächst wird dabei auf die bisherige Implementierung eingegangen, um im Anschluss den neuen Ansatz zu motivieren und erklären. Abschließend wird dessen Implementierung beschrieben. Daraufhin wird der implementierte Ansatz in Kapitel 4 evaluiert, indem zum einen untersucht wird, welche Auswirkungen der neue Algorithmus auf die Performanz kompilierter Programme hat und zum anderen die Auswirkungen auf die libFirm-Bibliothek analysiert werden. In einem Schlussfazit wird in Kapitel 5 die Frage geklärt, ob sich die Implementierung des, eigentlich nicht für den Einsatz in Compilern konzipierten, Algorithmus gelohnt hat.

2 Grundlagen und Verwandte Arbeiten

Dieses Kapitel führt in die zum Verständnis dieser Arbeit nötigen Grundlagen ein. Behandelt werden unter anderem Compiler, Kontrollflussgraphen und die Programmbibliothek libFirm. Außerdem wird das Problem der Grundblockanordnung motiviert und auf verwandte Arbeiten eingegangen. Die ersten beiden Abschnitte orientieren sich dabei an den Erläuterungen [1] von Cooper und Torczon.

2.1 Compiler

Ein *Compiler* ist ein Computerprogramm, welches Programme von einer Quellsprache in eine Zielsprache übersetzt. Typische Quellsprachen sind zum Beispiel `C`, `C++` und `Java`. Die Zielsprache ist klassischerweise der Befehlssatz eines bestimmten Prozessors.

Moderne optimierende Compiler bestehen in der Regel aus drei Komponenten: *Front-End*, *Middle-End* und *Back-End*. Das *Front-End* legt den Fokus auf das Einlesen und Verstehen des Quellprogrammes. Darüber hinaus übersetzt es das Programm in eine Zwischenrepräsentation (ZR), die zur weiteren Verarbeitung des Programmes innerhalb des Compilers dient. Das *Middle-End*, auch Optimierer genannt, transformiert die vom Front-End erzeugte ZR und gibt eine semantisch äquivalente ZR an das Back-End weiter. Mittels verschiedener Transformationen wird dabei versucht, das Programm zu vereinfachen, zu verkleinern, zu beschleunigen oder anderweitig zu verbessern. Zuletzt übernimmt das *Back-End* die Abbildung der optimierten ZR auf die Zielsprache.

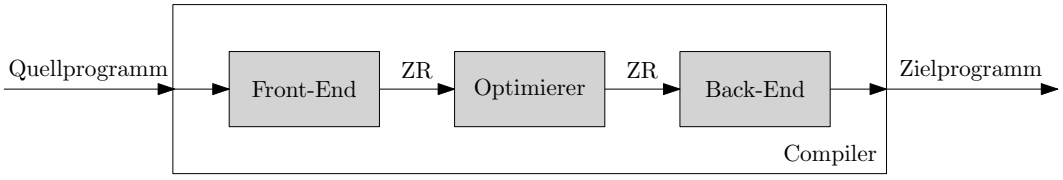


Abbildung 2.1: Aufbau eines optimierenden Compilers. [1]

Abhängig von den gewählten Quell- und Zielsprachen, sowie den speziellen Transformationen, die ein Compiler anwendet, wird eine Vielzahl verschiedener ZR eingesetzt. Beispielsweise werden oft Bäume zur Repräsentation der grammatikalischen Struktur des Quellprogrammes genutzt. Deren unflexible Struktur erschwert allerdings die Darstellung anderer Eigenschaften der Programme. Deshalb werden oft weitere allgemeinere Graphen als ZR genutzt. Ein solcher Graph wird in Abschnitt 2.2 näher betrachtet.

2.2 Kontrollflussgraphen

In der Regel besitzen Programme keinen linearen *Kontrollfluss*. Das bedeutet, sie bestehen nicht bloß aus einer Folge von Instruktionen, die linear abgearbeitet werden kann. Stattdessen enthalten Programme alternative Ausführungsstränge, Schleifen und Sprünge und weisen daher gerade einen nichtlinearen Kontrollfluss auf. In Compilern wird der Kontrollfluss eines Programmes häufig durch Kontrollflussgraphen dargestellt.

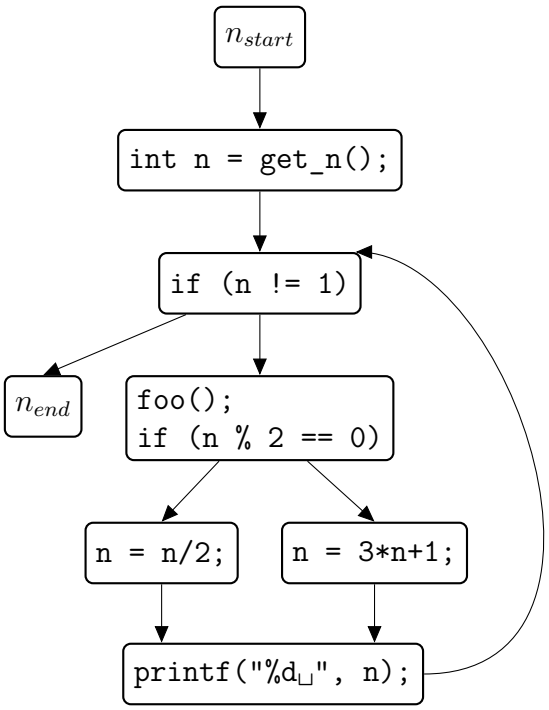
Ein *Grundblock* stellt die einfachste Einheit des Kontrollflusses in einem Programm dar. Er besteht aus einer maximal langen Folge von Instruktionen, die stets in derselben Reihenfolge ausgeführt wird. Grundblöcke müssen immer komplett von der ersten bis zur letzten Instruktion sequenziell ausgeführt werden.

Der Kontrollfluss zwischen den Grundblöcken eines Programmes wird mittels eines *Kontrollflussgraphen* modelliert. Ein Kontrollflussgraph ist ein gerichteter Graph $G = (N, E)$. Dabei entspricht jeder Knoten $n \in N$ einem Grundblock. Jede Kante $e = (n_i, n_j) \in E$ entspricht einem möglichen Übergang des Kontrollflusses von Block n_i zu Block n_j. Jeder Kontrollflussgraph enthält einen Startknoten n_{start} und einen Endknoten n_{end}. Diese markieren den Eintritts- und Austrittspunkt des Kontrollflusses für den modellierten Programmausschnitt.

Abbildung 2.2 zeigt beispielhaft ein `C`-Programm und dessen Darstellung als Kontrollflussgraph. Nur der Grundblock, bestehend aus den Zeilen 3 und 4, enthält mehr als eine Instruktion. Die Zeilen 1 und 2 bilden keinen gemeinsamen Grundblock, da Zeile 2 den Kontrollfluss auch aus Zeile 9 erhält. Andernfalls würde der gemeinsame Grundblock nicht immer von Anfang bis Ende durchlaufen.

```
1  int n = get_n();
2  while (n != 1) {
3      foo();
4      if (n % 2 == 0) {
5          n = n/2;
6      } else {
7          n = 3*n+1;
8      }
9      printf("%d␣", n);
10 }
```

a Programmcode



b Kontrollflussgraph

Abbildung 2.2: Ein Beispielprogramm mit zugehörigem Kontrollflussgraph.

2.3 Programmausführung

Auf Prozessoren ausführbare Programme können von Hand geschrieben oder mit einem Compiler erzeugt werden (siehe Abschnitt 2.1). Sie bestehen aus einer Folge von Maschinenbefehlen aus dem Befehlssatz des entsprechenden Prozessors. Die Befehle eines Befehlssatzes können dabei in zwei Klassen unterteilt werden: normale Befehle und Sprungbefehle, die den Kontrollfluss durch Verzweigungen und Unterprogrammaufrufe beeinflussen. Dieser Abschnitt orientiert sich im Folgenden an Drepper [2].

Bei der Ausführung eines Programmes durch einen Prozessor werden die einzelnen Maschinenbefehle ausgeführt. Um dies zu beschleunigen werden *Pipelines* eingesetzt. Dabei wird die Ausführung eines Befehls in mehrere Stufen unterteilt. Sobald ein Befehl eine Stufe durchlaufen hat, beginnt parallel bereits der folgende Befehl diese Stufe. Die maximale Ausführungsgeschwindigkeit wird erreicht, wenn sich in jeder Pipeline-Stufe je ein Befehl befindet. Derartige Pipelines können aus über 20 Stufen bestehen. Mit steigender Pipeline-Länge steigt auch die Zeit, die benötigt wird, um erneut die maximale Geschwindigkeit zu erreichen, falls der Strom an Befehlen abbricht (Pipeline-Stall). Dies geschieht, wenn der nächste Befehl nicht korrekt vorhergesagt werden kann, oder es zu lange dauert den nächsten Befehl zu laden.

Letzterem wird versucht, mittels Caches und spezielleren *Instruktionscaches* entgegenzuwirken. Caches sind schnelle Speicher, welche dem langsamen Hauptspeicher vorgeschaltet sind. In einem Instruktionscache werden Befehle aus dem Hauptspeicher zwischengespeichert, die vorraussichtlich bald vom Prozessor benötigt werden. Dies ist möglich, da Programmcode normalerweise hohe räumliche und zeitliche Lokalitäten aufweist. So wird beispielsweise Code in einer Schleife häufig mehrmals ausgeführt und nach einem Befehl werden in der Regel die im Speicher folgenden Befehle ausgeführt. Ist ein benötigter Befehl noch nicht im Cache vorhanden (Cache-Miss), muss er zeitaufwändig aus dem langsameren Speicher geladen werden. Dabei wird direkt ein größerer Speicherbereich in den Cache geladen. Wird derselbe oder einer der mitgeladenen Befehle daraufhin benötigt, liegt er bereits im schnellen Cache-Speicher.

Programmcode, der nur aus normalen Befehlen besteht, hat den Vorteil, dass er mittels Caches effizient geladen und ausgeführt werden kann. Bei Sprungbefehlen ist dies dagegen nicht der Fall. Das Sprungziel mancher bedingter Sprünge kann nicht statisch ermittelt werden. Teilweise ist nicht sicher, ob ein Sprung überhaupt ausgeführt wird. Außerdem verursachen viele Sprünge Cache-Misses. Diese Probleme verursachen Pipeline-Stalls und beeinträchtigen so die Performanz. Aus diesem Grund setzen moderne Prozessoren hoch spezialisierte *Sprungvorhersage* (engl. branch prediction) ein, um Sprungziele bereits früh vorherzusagen und somit die entsprechenden Instruktionen in den Cache laden zu können. Obwohl dies auf modernen Prozessoren sehr gut funktioniert, haben Sprungbefehle aber auch heute noch negative Auswirkungen auf die Performanz.

2.4 Grundblockanordnung

Soll ein Compiler-Back-End Programmcode für die Ausführung auf einem Prozessor erzeugen, muss es die ZR des Programmes in eine Folge von Maschinenbefehlen übersetzen. Bevor dabei die eigentlichen Maschinenbefehle generiert werden können, muss das Programm in eine speichergerechte lineare Form transformiert werden. Zusätzlich ist es ein grundlegendes Ziel des Compilers, möglichst effizienten, schnell ausführbaren Programmcode zu erzeugen. Für Programme, die keine Sprungbefehle beinhalten, ist diese Abbildung trivial. Enthält das Programm allerdings Sprungbefehle gewinnt das Problem an Komplexität.

Die Grundblöcke eines Programmes können theoretisch in beliebiger[1] Reihenfolge im Speicher angeordnet werden. Sprungbefehle am Ende der Grundblöcke gewährleisten den korrekten Kontrollfluss. Das Problem der *Grundblockanordnung* ist es, die

[1] In der Realität wird in vielen Fällen erwartet, dass der Startblock/Einsprungpunkt an erster Stelle steht.

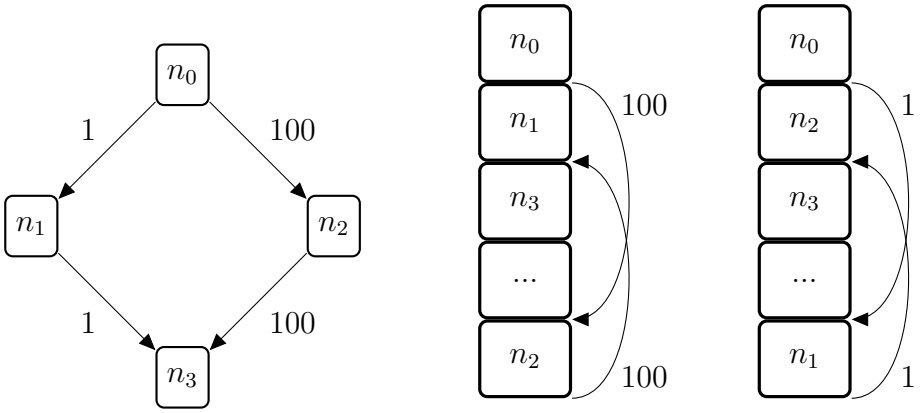
Grundblöcke in eine möglichst effiziente und schnell ausführbare Reihenfolge zu bringen. Im Folgenden wieder nach Torczon [1].

Da Sprungbefehle auf vielen Prozessoren die Performanz beeinträchtigen (siehe Abschnitt 2.3), ist es sinnvoll diese zu vermeiden. Hat ein Grundblock n_1 nur einen Nachfolger n_2 im Kontrollflussgraphen, kann dieser direkt anschließend im Speicher abgelegt werden. Der Sprung (n_1, n_2) kann dann komplett ausgelassen werden. Hat ein Grundblock mehrere Nachfolger, ist es nur möglich einen Nachfolger direkt anzuschließen. Die anderen Nachfolger müssen durch bedingte Sprungbefehle erreicht werden. Der Übergang zum direkt angeschlossenen Nachfolger wird auch als „Fall-through branch“ (dt. durchfallender Sprung) bezeichnet. Ein Fall-through beeinträchtigt auf den meisten Prozessoren die Performanz weniger als ein ausgeführter Sprung. Hat ein Grundblock mehrere Vorgänger im Kontrollflussgraphen muss entschieden werden, ob und von welchem Vorgänger durchgefallen wird. Des Weiteren können positive Cache-Effekte erzielt werden, wenn Blöcke, die gemeinsam ausgeführt werden, nahe im Speicher liegen.

Versuche, die Grundblockanordnung zu optimieren, beruhen implizit auf der Erkenntnis, dass manche Sprünge ein asymmetrisches Verhalten aufweisen: bei der Programmausführung werden manche Sprünge meist genommen, andere wiederum fast nie. Sprünge mit zwei Sprungzielen springen teilweise fast immer zum selben Ziel. Abbildung 2.3a zeigt einen Ausschnitt des Kontrollflussgraphen eines Programmes. Für jede Kante ist angegeben wie häufig sie zur Laufzeit ausgeführt wird. Der Sprung (n_0, n_2) wird 100 mal öfter als der Sprung (n_0, n_1) ausgeführt. Folglich sollte der Sprung (n_0, n_2) als Fall-through gesetzt werden. Hätten die Sprünge (n_0, n_1) und (n_0, n_2) ähnliche Ausführungshäufigkeiten, würde die Grundblockanordnung keine große Auswirkung auf den Programmausschnitt haben.

Zwei verschiedene Grundblockanordnungen für den Programmausschnitt sind in den Abbildungen 2.3b und 2.3c zu sehen. Die „langsame“ Anordnung setzt den Sprung (n_0, n_1) als Fall-through und (n_0, n_2) als bedingten Sprung. Die „schnelle“ Anordnung setzt die Sprünge genau umgekehrt. Falls nun auf dem Zielsystem Fall-throughs schneller als bedingte Sprünge sind, nutzt die „schnelle“ Grundblockanordnung den schnelleren Fall-through hundertmal öfter.

Laut Newell unt Pupyrev [3] beruht der verbreitetste Ansatz zur Grundblockanordnung darauf, häufig gemeinsam ausgeführte Grundblöcke eng beieinander in den Speicher zu legen. Ziel ist es, die Grundblöcke so anzuordnen, dass der wahrscheinlichste Nachfolger eines Blockes zum Fall-through wird, wie im Beispiel oben veranschaulicht wurde. Diese Strategie reduziert die Zahl der tatsächlich ausgeführten Sprünge, den benötigten Platz im Instruktionscache und entlastet die Sprungvorhersage. Formeller kann die Problemstellung dieses Ansatzes zur Grundblockanordnung nach ihnen wie folgt definiert werden:



a Kontrollflussgraph mit Ausführungshäufigkeiten der Sprünge

b Langsame Anordnung

c Schnelle Anordnung

Abbildung 2.3: Ausschnitt eines Kontrollflussgraphen und zwei mögliche Anordnungen der Grundblöcke. [1]

Gegeben ein gerichteter Kontrollflussgraph bestehend aus Grundblöcken und den Häufigkeiten der Sprünge zwischen den Blöcken. Finde eine Anordnung der Blöcke so, dass die Zahl der Fall-throughs maximiert wird.

In dieser Formulierung ist das Problem äquivalent zum Maximum Traveling Salesman Problem (TSP), einem NP-vollständigen Optimierungsproblem. Eine der ersten Arbeiten, die das Problem der Grundblockanordnung auf das Finden eines Pfades auf einem Graphen reduzierten, ist von Boesch und Gimpel [4]. Auf Grund der Einfachheit dieses Modells und der guten Ergebnisse in der Praxis sind TSP-basierte Ansätze zur Grundblockanordnung sehr verbreitet [3]. Im Folgenden wird dieser Ansatz (die Zahl der Fall-throughs zu maximieren) deshalb als „TSP-Ansatz“ bezeichnet.

2.5 Verwandte Arbeiten

Ein Großteil der Arbeiten, die sich mit dem Problem der Grundblockanordnung und verwandten Themen beschäftigen, stammen aus dem Gebiet der „Profile-guided Optimizations“ (PGO). Bei diesen werden Programme mittels unterschiedlicher Methoden zur Laufzeit vermessen, um Profile zu erstellen. Die gewonnenen Profile werden dann durch Compiler und sogenannte „Binary Optimizer“ verwendet, um eine optimierte Version des Programmes zu erzeugen.
Diese Arbeit unterscheidet sich jedoch von den folgenden Arbeiten aus dem Gebiet der PGO, da sie auf den Einsatz von Profilen verzichtet. Statt Profilen werden statische Analysen verwendet, die ohne Testläufe von einem Compiler durchgeführt

werden können.

Die Arbeit von Pettis und Hansen [5] ist eine der ersten, die sich direkt mit der Anordnung von Grundblöcken beschäftigten. In ihrer Arbeit stellen sie gleich zwei Algorithmen für den TSP-Ansatz (Abschnitt 2.4) zur Anordnung von Grundblöcken vor. Dabei werden Ketten aus Grundblöcken gebildet, die häufig nacheinander ausgeführt werden.
Im Laufe der Zeit wurden zahlreiche Arbeiten veröffentlicht, welche Erweiterungen und Variationen dieses Ansatzes vorstellen. Ein Beispiel ist die Arbeit „Software Trace Cache“ [6] von Ramirez, Larriba-Pey, Navarro, Valero und Torrellas. Sie findet unter anderem in der GCC[2] Verwendung und versucht wichtigen von unwichtigem Programmcode zu trennen. Hierzu werden besonders oft ausgeführte Programmteile in einen reservierten Teil des Instruktionscaches gelegt.

Einen anderen Ansatz [7] verfolgen beispielsweise Young, Johnson, Karger und Smith. Sie reduzieren das Problem der Grundblockanordnung auf eine gerichtete Version des Traveling Salesman Problems (DTSP). Unter der Verwendung von DTSP-Lösern erzielen sie gute Ergebnisse, benötigen dafür allerdings viel Rechenzeit.

Liu, Zhang, Liang, Yang und Cheng nähern sich dem Problem mithilfe neuronaler Netze [8]. Sie finden typische Strukturen in Kontrollflussgraphen, für welche anschließend ein neuronales Netz die Kantenhäufigkeiten schätzt. Auf Basis dieser Informationen wählen sie für jede Struktur ein Ergebnis aus einer Menge vordefinierter Grundblockanordnungen.

Neben der Anordnung von Grundblöcken, die häufig auf einzelne Funktionen beschränkt ist, beschäftigen sich einige Arbeiten auch mit der Anordnung ganzer Funktionen im Programm. Eine der ersten dieser Arbeiten ist auch die Arbeit [5] von Pettis und Hansen. Sie beschreiben einen Algorithmus, der die Funktionen eines Programmes nach einer „closest is best“-Strategie anordnet.

Ein neuerer Ansatz [9] zur Funktionsanordnung stammt von Ottoni und Maher. Sie stellen einen auf einem gerichteten Aufrufgraphen basierenden Ansatz vor. In ihrer Arbeit vergleichen sie den klassischen Ansatz von Pettis und Hansen mit ihrem eigenen und können hierbei eine Verbesserung feststellen.

Eine weitere mit dem Thema der Grundblockanordnung verwandte Arbeit [10] ist die „Hot-Cold Optimierung“ von Cohn und Lowney. Bei dieser werden die „kalten“ Grundblöcke einer Funktion, die nicht oder nur selten durchlaufen werden, von der restlichen Funktion separiert. Auf diese Weise rückt der „heiße“ Programmcode der Funktionen näher zusammen, was positive Cache-Effekte zur Folge hat. Die beschriebene Optimierung ist laut Newell und Pupyrev [3] komplementär zu der in

[2]GNU Compiler Collection, `https://gcc.gnu.org/`.

dieser Arbeit implementierten Optimierung der Grundblockanordnung und kann daher eingesetzt werden, um die Ergebnisse dieser Arbeit noch weiter zu optimieren. Eingesetzt wird sie unter anderem in der GCC.

2.6 libFirm

libFirm[3] ist eine in `C` geschriebene Programmbibliothek zum Compilerbau. Sie implementiert ein Compiler-Middle- und -Back-End. Diese ermöglichen zahlreiche Optimierungen sowie das Generieren von Assemblercode für verschiedene Prozessorarchitekturen wie `x86`, `x86_64` und `SPARC`. Mithilfe von libFirm ist es möglich, sich bei der Erstellung eines Compilers auf die Programmierung eines Front-Ends zu beschränken. Die weiteren Aufgaben des Compilers können durch libFirm übernommen werden, das unabhängig von der Quellsprache komplexe Optimierungen und die Codegenerierung kapselt.

Die von libFirm verwendete Zwischenrepräsentation ist das Graph-basierte Firm. Firm steht für „Fiasco's Intermediate Representation Mesh“, was auf seine ursprüngliche Herkunft, dem Fiasco `Sather-K` Compiler, zurückzuführen ist. Später wurde Firm von diesem in die libFirm-Bibliothek abgespalten.

Programme werden in Firm als Menge von Funktionen repräsentiert, die jeweils als Graph modelliert werden. Jede Instruktion entspricht dabei einem Knoten. Die Kanten des Graphen modellieren Abhängigkeiten, beispielsweise für den Daten- und Kontrollfluss. Dies hat zur Folge, dass Kontrollflusskanten nicht wie üblich anzeigen, wohin die Kontrolle aus einem Grundblock fließen kann, sondern woher ein Grundblock den Kontrollfluss erhalten kann.

Eine ausführlichere Einführung in die Firm-Zwischenrepräsentation bieten die Arbeiten [11] und [12].

Um libFirm verwenden zu können, wird ein Front-End benötigt, welches libFirms Middle- und Back-End zu einem kompletten Compiler vervollständigt (siehe Abschnitt 2.1). Zu den für libFirm verfügbaren Front-Ends gehören unter anderem:

bytecode2firm[4] Ein `Java`-Bytecode Front-End.
firm-bf[5] Ein Front-End für die esoterische Programmiersprache `Brainfuck`.
cparser[6] Ein zur GCC kompatibles `C`-Front-End mit voller `C99`-Unterstützung.

[3] Webseite des libFirm-Projekts: `https://pp.ipd.kit.edu/firm/`.
[4] `https://github.com/libfirm/bytecode2firm`.
[5] `https://github.com/libfirm/firm-bf`.
[6] `https://github.com/libfirm/cparser/`.

3 Entwurf und Implementierung

Dieses Kapitel stellt den, im Rahmen dieser Arbeit implementierten, Ansatz zur Optimierung der Grundblockanordnung in libFirm vor. Zunächst wird dabei auf die bisherige Implementierung eingegangen. Anschließend wird der neue Ansatz motiviert, erklärt und zuletzt auch dessen Implementierung beschrieben.

3.1 Grundblockanordnung in libFirm

Wie in Abschnitt 2.6 bereits erläutert, unterstützt libFirm die Codegenerierung für verschiedene Prozessorarchitekturen. Das interne Back-End ist modular aufgebaut und besitzt einzelne Module für die unterstützten Architekturen. Konkret sind dies:

- `AMD64`
- `ARM`
- `IA-32`
- `MIPS`
- `RISC-V`
- `SPARC`

Diese implementieren eine Schnittstelle über die sie aufgerufen werden können, um Code zu generieren. Da libFirm Programme als Firm-Graphen einzelner Funktionen repräsentiert, findet die Codegenerierung für jede Funktion getrennt statt. Dabei ist der erste Schritt eines jeden Back-Ends die Zerteilung des Firm-Graphen der Funktion in eine geordnete Liste der enthaltenen Grundblöcke. Anschließend wird der Code für die Funktion generiert, wobei die Grundblöcke, in der zuvor festgelegten Reihenfolge, verarbeitet werden. Die Abbildung des Firm-Graphen auf die geordnete Liste von Grundblöcken - die Grundblockanordnung (siehe Abschnitt 2.4) - erfolgt durch eine Komponente die in libFirm als „Block-Scheduler“ bezeichnet wird.

Der Block-Scheduler ist nicht Bestandteil der einzelnen Back-End-Module. Stattdessen ist er Teil des allgemeineren Back-Ends und kann durch die einzelnen Module verwendet werden. So ist es möglich, die Grundblockanordnung unabhängig von der konkreten Zielarchitektur zu implementieren.

Die Eingabe des Block-Schedulers, der Funktions-Firm-Graph, stellt auf oberster Ebene einen Kontrollflussgraphen dar. Er enthält Knoten, die die Grundblöcke der Funktion repräsentieren, und Kanten zur Repräsentation der Kontrollflussabhängigkeiten. Wie in Abschnitt 2.6 erwähnt, sind diese Kontrollflusskanten im Vergleich zu den Kanten eines normalen Kontrollflussgraphen exakt umgekehrt. Die Grundblock-Knoten enthalten Subgraphen zur Repräsentation der Instruktionen, Speicherzustände und weiteren Elementen von Firm. Außerdem enthält der Funktionsgraph immer einen Start- und einen End-Grundblock. Allerdings enthält nur der Startblock Instruktions-Knoten. Der Endblock wird vorallem dafür benötigt, über den Graphen iterieren zu können.

Die Ausgabe des Block-Schedulers ist eine geordnete Liste der Grundblock-Knoten des Funktions-Firm-Graphen. Diese enthält nicht den Endblock, da für diesen kein Programmcode generiert werden muss. Außerdem ist erforderlich, dass die generierte Grundblockanordnung mit dem Startblock der Funktion beginnt.

libFirms bisherige Implementierung des Block-Schedulers verfolgt den verbreiteten in Abschnitt 2.4 beschriebenen TSP-Ansatz: sie versucht die Zahl der Fall-throughs zu maximieren und verwendet dabei geschätzte Ausführungshäufigkeiten der Grundblöcke und Kontrollflusskanten. Der konkret verwendete Algorithmus geht wie folgt vor:

1. Für alle Blöcke, die nur einen Nachfolger haben: Setze den ausgehenden Sprung als Fall-through, wenn dieser die höchste Häufigkeit unter den beim Nachfolger eingehenden Kontrollflusskanten hat. (Betrachte zuerst häufiger ausgeführte Sprünge)

2. Für alle Schleifen: Setzte bis zu einen Sprung der die Schleife verlässt als Fall-through. (Betrachte Sprünge zu seltener ausgeführten Blöcken zuerst)

3. Setzte alle übrigen Sprünge als Fall-through, wenn deren Start- und Zielblock noch keinen aus-/eingehenden Fall-through haben. (Betrachte häufiger ausgeführte Sprünge zuerst)

4. Breche entstandene Zyklen aus Fall-throughs auf.

5. Sammle alle Ketten aus durch Fall-throughs verbundenen Blöcken. Blöcke ohne ein- oder ausgehende Fall-throughs sind Ketten der Länge 1.

6. Bilde eine Kette aus Fall-throughs ausgehend vom Startblock. Bis alle Blöcke Teil der Kette sind: Hänge an das Ende der Kette, die Kette des direkten Nachfolgers mit der höchsten Sprunghäufigkeit an. Falls alle Nachfolger bereits Teil der Kette sind, hänge eine beliebige andere Kette an.

Dies ist ein recht simpler greedy Algorithmus, was den Schluss nahelegt, dass an dieser Stelle noch Optimierungspotential bestehen könnte. Ein komplexerer Algorithmus könnte möglicherweise bessere Grundblockanordnungen produzieren. Verstärkt wird diese Vermutung noch durch die Beobachtung, dass der Algorithmus sich Kombinationsmöglichkeiten verbaut, indem er Zyklen zunächst konstruiert, die später jedoch wieder aufgebrochen werden müssen.

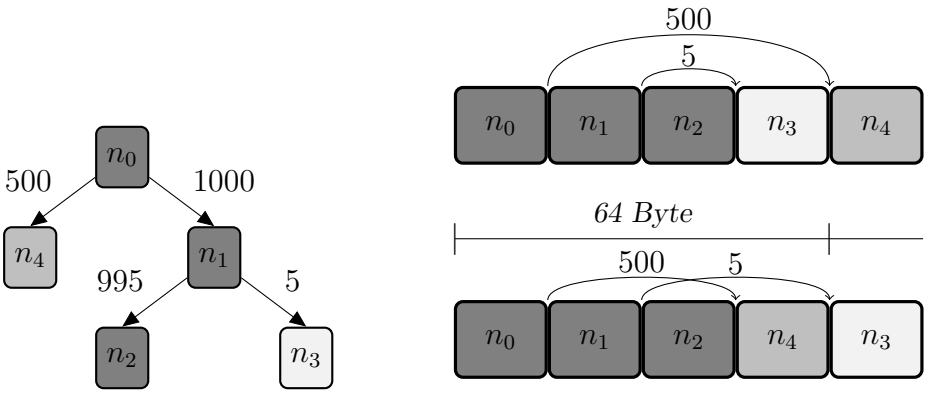
Die nächsten Abschnitte beschreiben den in dieser Arbeit implementierten Algorithmus, der eine Alternative zur bisherigen Implementierung des libFirm Block-Schedulers darstellt.

3.2 ExtTSP

In ihrer Arbeit „Improved Basic Block Reordering“ [3] regen Newell und Pupyrev an, dass der verbreitete TSP-Ansatz (Abschnitt 2.4) zur Grundblockanordnung ihrer Ansicht nach nicht ausreicht, um gute Grundblockanordnungen zu konstruieren. Genauer bemängeln sie, dass das bloße Maximieren der Fall-throughs negative Auswirkungen auf das Caching-Verhalten der Programme haben kann. Außerdem zeigen sie auf, dass sich Programmbeispiele finden lassen, die unter den Anforderungen des TSP-Ansatzes mehrere optimale Lösungen besitzen.

Ein solches Beispiel zeigt Abbildung 3.1 auf der nächsten Seite. Für den mit Ausführungshäufigkeiten annotierten Kontrollflussgraphen in Abbildung 3.1a erreichen zwei verschiedene Grundblockanordnungen (dargestellt in Abbildung 3.1b) die maximale Anzahl von Fall-throughs. Die beiden Anordnungen unterscheiden sich allerdings in der Anzahl der Instruktionscachezeilen, die sie während einer typischen Ausführung benötigen. Dies führt dazu, dass das untere Beispiel vermutlich eine höhere Performanz aufweisen wird, da sich die häufigsten Sprünge innerhalb einer einzelnen Cachezeile bewegen. Es zeigt sich also, dass auch Sprünge die keine Fall-throughs werden, durch einen Algorithmus beachtet werden sollten.

Dass es generell nicht immer vorteilhaft ist, die Zahl der Fall-throughs zu maximieren, zeigt ein weiteres Beispiel aus ihrer Arbeit. Für den Kontrollflussgraphen in Abbildung 3.2a muss eine Grundblockanordnung mit der maximalen Anzahl an Fall-throughs die Fall-through-Ketten $n_0 \to n_1 \to n_3 \to n_4$ und $n_5 \to n_6 \to n_2$ enthalten. Die obere der beiden Anordnungen in Abbildung 3.2b zeigt eine solche. Die untere Anordnung hingegen, enthält nicht die maximale Anzahl Fall-throughs, da sie den Sprung $n_6 \to n_2$ nicht als Fall-through setzt. Ein Blick auf die Cachenutzung der beiden Alternativen weist die untere Anordnung als performanter aus, da sie die häufig ausgeführten Grundblöcke auf zwei Cachezeilen konzentriert. Die obere Anordnung verteilt diese dagegen auf drei Cachezeilen. Die untere Anordnung ist



a Kontrollflussgraph **b** Grundblockanordnungen

Abbildung 3.1: Ein Kontrollflussgraph und zwei mögliche Anordnungen der Grundblöcke. Beide haben die maximale Anzahl an Fall-throughs, aber sie nutzen den Instruktionscache unterschiedlich effektiv aus. Alle Grundblöcke haben dieselbe Größe. Die Blockfarbe symbolisiert die Ausführungshäufigkeit. [3]

also besser, obwohl sie die Zahl der Fall-throughs nicht maximiert.

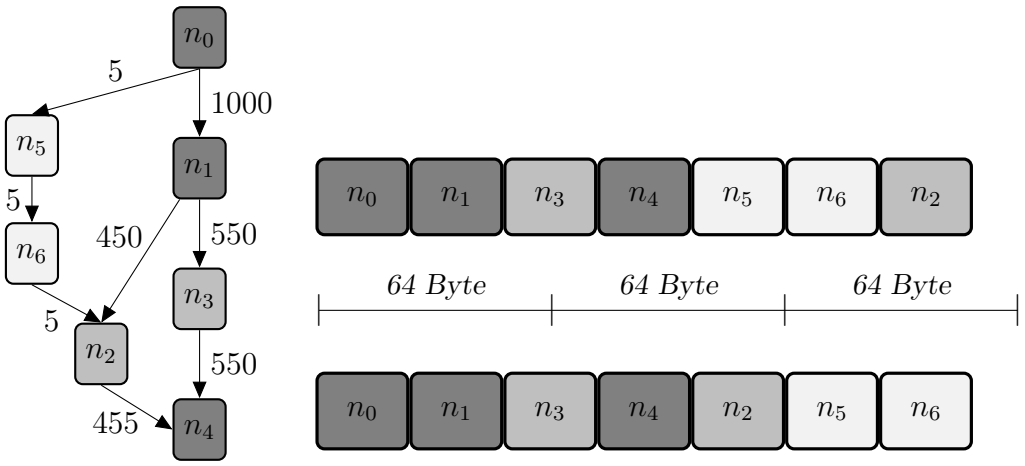
Des Weiteren entwerfen Newell und Pupyrev in ihrer Arbeit Score-Funktionen, die Grundblockanordnungen auf einen reellen Wert abbilden. Dieser Wert soll die zu erwartende Performanz eines mit der Anordnung erzeugten Programmes widerspiegeln. Dabei nehmen sie an, dass die Ausführungszeit einzelner Grundblöcke nicht von der gewählten Anordnung abhängt. Demnach beeinflusst die Anordnung nur die Sprünge, welche wiederum negative Auswirkungen wie Pipeline-Stalls haben können (siehe Abschnitt 2.3). Zusätzlich zu den Sprüngen, lassen sie auch deren Längen in ihre Funktionen mit einfließen. Auf diese Weise können die negativen Cache-Effekte langer Sprünge einbezogen werden und Fall-throughs als Sprünge der Länge 0 dargestellt werden.

Dem TSP-Ansatz ordnen sie die folgende Score-Funktion zu:

$$\text{TSP} = \sum_{(s,t)\in E} w(s,t) \cdot \begin{cases} 1 & \text{für } len(s,t) = 0, \\ 0 & \text{für } len(s,t) > 0. \end{cases}$$

Dabei ist $len(s,t)$ die Länge und $w(s,t)$ die Ausführungshäufigkeit des Sprunges beziehungsweise der Kontrollflusskante (s,t). E entspricht der Menge der Kanten des Kontrollflussgraphen. Eine optimale Grundblockanordnung im Sinne des TSP-Ansatzes entspricht hier einem maximalen TSP-Score.

Um eine Score-Funktion zu erhalten, die das Problem der Grundblockanordnung besser modelliert und die beobachteten Ungenauigkeiten des TSP-Ansatzes beachtet,



a Kontrollflussgraph **b** Grundblockanordnungen

Abbildung 3.2: Ein Kontrollflussgraph und zwei mögliche Anordnungen der Grundblöcke. Die untere Anordnung nutzt den Instruktionscache optimaler, obwohl sie nicht die maximale Anzahl Fall-throughs enthält. Alle Grundblöcke haben dieselbe Größe. Die Blockfarbe symbolisiert die Ausführungshäufigkeit. [3]

gehen Newell und Pupyrev folgendermaßen vor: sie definieren eine allgemeine Score-Funktion, die unter anderem Sprunglänge, Richtung und Art (bedingt/unbedingt) mit variablen Koeffizienten enthält. Anschließend führen sie aufwendige Experimente mit verschiedenen Grundblockanordnungen durch, um Daten über den Zusammenhang der einzelnen Sprungeigenschaften mit der Performanz zu erhalten. Schließlich wenden sie ein komplexes Machine-Learning-Verfahren an, um aus den Daten passende Koeffizienten für ihre Score-Funktion zu generieren.

Die erhaltene optimierte Score-Funktion, die sie Extended TSP (ExtTSP) nennen, ist folgende:

$$\text{ExtTSP} = \sum_{(s,t)\in E} w(s,t) \cdot \begin{cases} 1 & \text{für } len(s,t) = 0, \\ 0{,}1 \cdot \left(1 - \frac{len(s,t)}{1024}\right) & \text{für } 0 < len(s,t) \leq 1024 \text{ und } s < t, \\ 0{,}1 \cdot \left(1 - \frac{len(s,t)}{640}\right) & \text{für } 0 < len(s,t) \leq 640 \text{ und } s > t, \\ 0 & \text{sonst.} \end{cases}$$

Interessanterweise ähnelt ExtTSP dem klassischen TSP-Modell, denn der dominante Faktor ist ebenfalls die Zahl der Fall-throughs. Der Hauptunterschied liegt darin, dass ExtTSP auch längere Sprünge betrachtet. Der Einfluss dieser ist jedoch signifikant kleiner und nimmt linear in der Länge des Sprungs ab. Weiterhin scheinen im Hinblick auf die Performanz Vorwärtssprünge wichtiger als Rückwärtssprünge zu sein. Der Unterschied zwischen bedingten und unbedingten Sprüngen war dagegen so gering, dass er nicht in die Funktion mit aufgenommen wurde. Insgesamt stellen Newell und

Pupyrev eine hohe Korrelation zwischen dem ExtTSP-Score und der Performanz einer Grundblockblockanordnung fest.

Da die Autoren keine Aussage hierzu treffen ist unklar, in wie weit die erhaltenen ExtTSP-Koeffizienten von der Maschine abhängen, auf der zu Beginn die Daten gesammelt wurden. Es wäre also beispielsweise möglich, dass die optimalen Koeffizienten stark von der Cache-Struktur der Maschine abhängen.

Das ExtTSP-Problem, das sich analog zum TSP-Ansatz formulieren lässt, ist auch ein NP-vollständiges Optimierungsproblem [3]. Im nächsten Abschnitt wird der von Newell und Pupyrev vorgestellte Algorithmus für das ExtTSP-Problem erläutert.

3.3 ExtTSP-Algorithmus

Listing 3.1 zeigt den vorgeschlagenen Algorithmus zur Lösung des ExtTSP-Problems. Im Folgenden wird dieser als „ExtTSP-Algorithmus“ oder kurz Algorithmus bezeichnet und analog zur Originalarbeit [3] beschrieben.

Der Algorithmus findet eine optimierte Grundblockanordnung für einzelne Funktionen eines Programmes. Als Eingabe erwartet er einen gewichteten Kontrollflussgraphen $G = (N, E, w)$. N und E sind dabei analog zu Abschnitt 2.2 definiert. $w(s, t)$ ist die Funktion, die jeder Kontrollflusskante $(s, t) \in E$ eine Ausführungshäufigkeit zuordnet. Des Weiteren wird eine Funktion `size`(x) erwartet, die jedem Grundblock seine Größe in Bytes zuordnet. Diese wird benötigt, um die Längen der Sprünge einer Grundblockanordnung zu berechnen. Ziel des Algorithmus ist es, eine Grundblockanordnung $(g_1, g_2, \ldots, g_{|N|})$ der Blöcke in N zu finden, die mit dem ebenfalls gegebenen Startblock $n^* \in N$ beginnt und einen möglichst hohen ExtTSP-Score aufweist.

Grundsätzlich ist der Algorithmus greedy und arbeitet mit Ketten von Grundblöcken. Zu Beginn befindet sich dabei jeder Grundblock in einer eigenen Kette. Dann werden iterativ Paare von Ketten verschmolzen, um den erreichten ExtTSP-Score zu erhöhen. In jeder Iteration werden die zwei Ketten miteinander verschmolzen, deren Kombination die größte Steigerung des ExtTSP-Scores hervorruft. Ist nur noch eine Kette übrig stoppt der Algorithmus. Die verbliebene Kette enthält alle Grundblöcke und stellt die gewonnene Grundblockanordnung dar (siehe `ReorderBasicBlocks()` in Listing 3.1).

Listing 3.1: ExtTSP-Algorithmus [3]

1 **Eingabe**: Kontrollflussgraph $G = (N, E, w)$, Startblock $n^* \in N$, size(x)
2 **Ausgabe**: Grundblockanordnung $(n^* = g_1, g_2, \ldots, g_{|N|})$
3
4 **Function** ReorderBasicBlocks$(\ldots)$
5 `/* initial chain creation */`
6 **for** $n \in N$ **do**
7 $Chains \leftarrow Chains \cup (n)$;
8 `/* chain merging */`
9 **while** $|Chains| > 1$ **do**
10 **for** $c_i, c_j \in Chains$ **do**
11 $gain[c_i, c_j] \leftarrow$ ComputeMergeGain(c_i, c_j);
12 `/* find best pair of chains */`
13 $src, dst \leftarrow \arg\max_{i,j} gain[c_i, c_j]$;
14 `/* merge the pair and update chains */`
15 $Chains \leftarrow Chains \cup$ Merge$(src, dst) \setminus \{src, dst\}$;
16 **return** *ordering given by the remaining chain*;
17
18
19 **Function** ComputeMergeGain(src, dst)
20 `/* try all ways to split chain` src `*/`
21 **for** $i = 1$ **to** size(src) **do**
22 `/* break the chain at index` i `*/`
23 $s_1 \leftarrow src[1 : i]$;
24 $s_2 \leftarrow src[i + 1 :$ size$(src)]$;
25 `/* try all valid ways to concatenate */`
26 $score_i \leftarrow \max \begin{cases} \text{ExtTSP}(s_1, s_2, dst) \text{ if } n^* \notin dst \\ \text{ExtTSP}(s_1, dst, s_2) \text{ if } n^* \notin dst \\ \text{ExtTSP}(s_2, s_1, dst) \text{ if } n^* \notin s_1, dst \\ \text{ExtTSP}(s_2, dst, s_1) \text{ if } n^* \notin s_1, dst \\ \text{ExtTSP}(dst, s_1, s_2) \text{ if } n^* \notin src \\ \text{ExtTSP}(dst, s_2, s_1) \text{ if } n^* \notin src \end{cases}$
27 `/* return the gain of merging chains` src `and` dst `*/`
28 **return** $\max_i score_i - \text{ExtTSP}(src) - \text{ExtTSP}(dst)$;

Ein wichtiger Bestandteil des Algorithmus ist die Vorgehensweise beim Verschmelzen zweier Ketten. Dies kann in der Funktion `ComputeMergeGain()` genauer betrachtet werden, welche für zwei Ketten die Steigerung des ExtTSP-Scores durch eine Verschmelzung berechnet. Um zwei Ketten *src* und *dst* zu verschmelzen wird zunächst die Kette *src* an der Stelle i in zwei kleinere Ketten s_1 und s_2 geteilt. Anschließend werden für die Ketten s_1, s_2 und *dst* alle 6 möglichen Kombinationen, sie zu einer langen Kette zusammenzusetzen, betrachtet. Dabei werden die Kombinationen verworfen, die den Startblock n^* nicht am Beginn der Kette belassen. Nachdem *src* an allen möglichen Stellen i zerschnitten und die jeweiligen Rekombinationen mit *dst* betrachtet wurden, wird jene betrachtete Kette als Ergebnis der Verschmelzung gewählt, welche den höchsten ExtTSP-Score erreicht hat.

Diese komplexe Rekombination wurde gewählt, um den Suchraum verglichen mit simpler Konkatenation zu vergrößern. So können Lösungen erreicht werden, die der greedy Algorithmus sonst nicht finden würde.

Die Laufzeit einer einfachen Implementierung des ExtTSP-Algorithmus liegt in $\mathcal{O}(|N|^5)$: in der Funktion `ReorderBasicBlocks` werden $|N|$ Verschmelzungen von Ketten vorgenommen. Dabei müssen jeweils bis zu $|N|^2$ Paare von Ketten betrachtet werden. Für jedes dieser Paare benötigt die Funktion `ComputeMergeGain` $\mathcal{O}(|N|^2)$ Schritte, um den Verschmelzungsgewinn zu berechnen. In Unterabschnitt 3.4.2 wird beschrieben wie diese Laufzeit noch reduziert werden kann.

3.4 Implementierung

Der im vorstehenden Kapitel beschriebene ExtTSP-Algorithmus wurde im Rahmen dieser Arbeit komplett in die libFirm-Bibliothek integriert. Da diese zuvor nur auf einen „Block-Scheduler“ (siehe Abschnitt 3.1) ausgelegt war, musste sie zunächst vorbereitet werden. Hierfür wurde eine neue Kommandozeilenoption hinzugefügt. Unter Verwendung des „cparser“-Front-Ends ist diese beispielsweise als

```
-mblock-scheduler={normal, exttsp, random}
```

verfügbar. So ist es dem Nutzer des Compilers möglich, den verwendeten Algorithmus zur Grundblockanordnung auszuwählen. `normal` entspricht dabei der bisherigen Implementierung die in Abschnitt 3.1 erläutert wurde. `exttsp` entspricht dem ExtTSP-Algorithmus. `random` wählt einen zu Testzwecken in dieser Arbeit implementierten Block-Scheduler, der pseudozufällige Grundblockanordnungen generiert. Die Standardeinstellung ist dabei momentan in allen Fällen `normal`.

3.4.1 Voraussetzungen

Damit der ExtTSP-Algorithmus in libFirm implementiert werden konnte, mussten die hierfür benötigten Eingabedaten bereitgestellt werden.

Für den Kontrollflussgraphen war dies leicht möglich, da die verwendete Zwischenrepräsentation Firm, wie in Abschnitt 3.1 erklärt, diesen bereits enthält. Für die weiteren benötigten Daten gestaltete sich dies schwieriger.

Der ursprüngliche Einsatzzweck, für welchen die Autoren den ExtTSP-Algorithmus entwickelten, ist in dem Binary Optimizer „BOLT“ [13]. Dies ist ein Programm, das mittels Profile-Guided-Optimization bereits kompilierte Programme beschleunigen soll. Hierfür werden zur Laufzeit beispielsweise die Ausführungshäufigkeiten von Grundblöcken und Sprüngen gemessen. Auch die Größen der einzelnen Grundblöcke, wie sie durch den Algorithmus benötigt werden, sind in diesem Anwendungsfall natürlich vorhanden. Da solche Daten in libFirm nicht zur Verfügung stehen, müssen diese auf andere Art ermittelt werden.

Im Fall der Ausführungshäufigkeiten bot libFirm hierfür bereits eine Lösung: mittels eines Linearen Gleichungssystems ist es in der Lage, geschätzte Ausführungshäufigkeiten für die Grundblöcke und Kontrollflusskanten eines Firm-Graphen zu generieren. Dabei wird jeder Kontrollflusskante eine Wahrscheinlichkeit zugewiesen. Im Fall von Schleifenkanten wird hier von einer 90-prozentigen Wahrscheinlichkeit ausgegangen, dass der Kontrollfluss diese nicht verlässt. Für if-Anweisungen werden alle Kontrollflusspfade als gleich wahrscheinlich angenommen. Des Weiteren ist die Summe der Wahrscheinlichkeiten aller ausgehenden Kontrollflusskanten eines Blockes Eins. So ergibt sich die Ausführungshäufigkeit eines Grundblockes als die Summe aller Ausführungshäufigkeiten seiner Kontrollflussvorgänger, jeweils skaliert um die Wahrscheinlichkeit der passenden Kontrollflusskante.

Für die benötigten Größen der Grundblöcke bot libFirm noch keine passende Lösung. Deshalb wurde im Rahmen dieser Arbeit eine Größenschätzung für Instruktionen implementiert. Mithilfe dieser konnten dann geschätzte Größen für ganze Grundblöcke gewonnen werden. Da die Länge einer Instruktion in Maschinensprache von der Prozessorarchitektur abhängt, musste die Größenschätzung für die verschiedenen Back-End-Module separat implementiert werden. Für manche Architekturen war dies trivial möglich, da sie feste Instruktionsgrößen aufweisen. Ein Beispiel hierfür ist SPARC, auf welcher alle Instruktionen 4 Byte groß sind. Andere Architekturen wie beispielsweise IA-32 (x86) erlauben variable Instruktionslängen, wodurch die Schätzung der endgültigen Länge enorm an Komplexität gewinnt. Für diese wurden einfache Abschätzungen implementiert, die sich beispielsweise an der Existenz von Immediate-Argumenten der Instruktionen orientieren. In durchgeführten Tests lieferten die Größenschätzungen auf diesen Architekturen passable Ergebnisse.

Die für IA-32 implementierte Größenschätzung funktioniert beispielsweise wie folgt:

```
1  if Node is IncSP: // Veränderung des Stackpointers
2      if Offset(Node) == 0:
3          return 0 Byte
4      else
5          return 3 Byte
6  estimate = 2 Byte // Schätze 2 Bytes für normale Befehle
7  for all Predecessors(Node) p:
8      if p is Immediate: // Node hängt von Immediate-Wert ab
9          estimate += 4 Byte
10         break
11 if Node has ConstAttribute: // Zusätzliche kleine Konstante
12     estimate += 2 Byte
13 return estimate
```

Eine der zentralen Fragen dieser Arbeit ist, ob der vorgestellte ExtTSP-Algorithmus auch ohne exakte Profiling-Daten und Grundblockgrößen noch gute Ergebnisse liefern kann und sich somit für den Einsatz in Compilern eignet.

3.4.2 Optimierungen

In ihrer Arbeit [3] zeigen Newell und Pupyrev einige Möglichkeiten auf, wie die schlechte Laufzeit ihres Algorithmus von $\mathcal{O}(|N|^5)$ noch verbessert werden kann. Diese wurden alle in libFirm implementiert und werden im Folgenden beschrieben.

Das Verschmelzen zweier Ketten $c1$ und $c2$ durch den Algorithmus kann nur dann einen Gewinn im ExtTSP-Score hervorrufen, wenn ein Sprung zwischen $c1$ und $c2$ existiert. Dies liegt daran, dass ExtTSP nur Sprünge betrachtet. So muss das aufwendige `ComputeMergeGain` nur für adjazente Ketten berechnet werden und die Zahl der Kettenpaare, die zum Verschmelzen in Frage kommen, ist nach oben durch die Zahl der Kanten $|E|$ des Kontrollflussgraphen beschränkt.

Weiterhin lassen sich die Ergebnisse von `ComputeMergeGain` zwischenspeichern und wiederverwenden, da jeder berechnete „Merge-Gain“ nur von zwei Ketten abhängt. Nimmt in einer Iteration keine dieser beiden am Verschmelzen teil, kann das Ergebnis aus der vorherigen Iteration wiederverwendet werden. Zusammen mit der vorherigen Beobachtung ergibt sich so eine Beschränkung des Algorithmus durch

$$\mathcal{O}\left(\sum_{c} size(c) \cdot degree(c)\right)$$

wobei die Summe über alle Ketten c gebildet wird, die an einer Verschmelzung teilnehmen. Dabei ist $size(c)$ die Zahl der Grundblöcke in c und $degree(c)$ die Zahl der Sprünge, die c betreten oder verlassen. Im Worst-Case ergibt dies eine Laufzeit in $\mathcal{O}(|N|^2 \cdot |E|)$, was auf realen Kontrollflussgraphen $\mathcal{O}(|N|^3)$ entspricht.

Da sie im Laufe ihrer Experimente auf Beispiele für Funktionen mit extrem vielen Grundblöcken stießen, führen Newell und Pupyrev zudem einen positiven Schwellwert K ein. Besteht eine Kette aus mehr als K Grundblöcken, wird sie dann in `ComputeMergeGain` nicht mehr zerteilt, sondern bloß noch mit anderen Ketten konkateniert. So liegt die Laufzeit für sehr große Funktionen nicht mehr im kubischen Bereich, sondern nur noch in $\mathcal{O}(K \cdot |N|^2)$, wobei sie $K = 128$ als Schwellwert wählen.

Unabhängig von den Verbesserungsvorschlägen von Newell und Pupyrev wurde `ComputeMergeGain` in dieser Arbeit noch weiter optimiert. So reicht es zum einen, die erste der sechs Kombinationsmöglichkeiten ExtTSP(s_1, s_2, dst) lediglich einmal statt in der Schleife zu prüfen, da diese sich für die verschieden Trennpunkte i nicht unterscheidet. Zum anderen kann die fünfte Kombination ExtTSP(dst, s_1, s_2) komplett gestrichen werden, da diese der ersten Kombination entspricht, wenn `ComputeMergeGain` mit dst als erstem und src als zweitem Argument aufgerufen wird.

Des Weiteren wurde die Implementierung des $\texttt{arg}\max_{i,j} gain[c_i, c_j]$ in der Funktion `ReorderBasicBlocks` als Priority-Queue zur weiteren Performanzsteigerung in Betracht gezogen. Verschiedene Tests zeigten allerdings, dass die Laufzeit der Maximumssuche verglichen mit der Berechnung der „Merge-Gains“ vernachlässigbar klein ist.

Sonstiges

Insgesamt benötigte die Implementierung des ExtTSP-Algorithmus und der nötigen Erweiterungen von libFirm knapp über 700 Zeilen `C`-Quellcode.

4 Evaluation

Dieses Kapitel evaluiert den in dieser Arbeit implementierten Ansatz zur Grundblockanordnung. Hierfür wird einerseits untersucht, welche Auswirkungen der neue Algorithmus auf die Performanz kompilierter Programme hat und andererseits die Auswirkungen auf die libFirm-Bibliothek analysiert.

4.1 Testbedingungen

Alle Messungen und Benchmarks wurden auf einem dedizierten Testsystem mit den folgenden Spezifikationen durchgeführt:

Prozessor	Intel® Core™ i7-3770 @ 8 x 3.40GHz
Hauptspeicher	15 GiB
Betriebssystem	Ubuntu 18.04.2 LTS (64-bit)
Linux Kernel	4.15.0-46-generic

In allen Fällen wurde dabei das libFirm-Front-End „cparser“ in der Version 1.22.1[1] verwendet. Dieses wurde immer mit der höchsten Optimierungsstufe `-O3` ausgeführt. Kombiniert mit diesem wurde libFirm in der Version 1.22[2] eingesetzt. Zur Auswahl des verwendeten Algorithmus zur Grundblockanordnung wurde die neu eingeführte Kommandozeilenoption `-mblock-scheduler` mit entsprechendem Argument genutzt. Für alle Tests mit dem AMD64-Back-End benötigte libFirm das zusätzliche Argument `-fPIC`.

Im Folgenden wird libFirms bestehende Implementierung der Grundblockanordnung als „Normal“ bezeichnet. Die neue Implementierung wird „ExtTSP“ genannt.

[1] Der verwendete cparser Commit ist `217ee4a08ab7c09a8666977c7e2963edc9a8776d`.
[2] Der verwendete libFirm Commit ist `e7745ca2880f142f5e3f1cbabf7c62d327f42eb2`.

4.2 Auswirkungen auf erzeugte Programme

Um zu untersuchen wie sich die ExtTSP-Grundblockanordnung auf kompilierte Programme auswirkt, wurde der SPEC CPU2000-Benchmark[3] verwendet. Dies ist eine standardisierte Sammlung verschiedener Benchmark-Programme der Standard Performance Evaluation Corporation (SPEC). Sie besteht aus 26 `C`, `C++` und `Fortran` Testprogrammen, die zum Testen der Performanz von Prozessoren und Compilern eingesetzt werden. Da aktuell keine libFirm-Front-Ends für die Sprachen `C++` und `Fortran` existieren, wurden im Folgenden lediglich die 15 `C`-Benchmarks verwendet. Diese unterteilen sich wiederum in 11 Integer- und 4 Floating-point-Benchmarks.

Die Benchmarks wurden mithilfe der SPEC-Suite kompiliert, 35 Mal ausgeführt und dabei auf ihre Ausführungszeit vermessen. Um die Auswirkungen von ExtTSP betrachten zu können, wurde dies jeweils für Normal und ExtTSP durchgeführt. Da das verwendete Testsystem außerdem sowohl AMD64-, als auch IA-32-Programme ausführen kann, wurden die beschriebenen Tests für beide Back-Ends durchgeführt.

Das Ergebnis für jedes der Back-Ends zeigen die Tabellen 4.1 und 4.2. Die angegebenen Laufzeiten entsprechen dem arithmetischen Mittel der Laufzeiten in Sekunden, über alle 35 Iterationen des Benchmarks. Zu jedem Laufzeitmittel ist die Standardabweichung σ angegebenen. Die Spalte „Speedup“ gibt an, wie viel Prozent der Benchmark im Mittel schneller bzw. langsamer bei der Verwendung von ExtTSP war. Zur Überprüfung der Signifikanz dieser Unterschiede wurde ein t-Test für alle Benchmarks durchgeführt. Dieser schätzt die Wahrscheinlichkeit, dass die gemessenen Unterschiede zwischen Normal und ExtTSP lediglich auf zu wenige Messungen zurückzuführen sind. Ist das Ergebnis des t-Test höher als das Signifikanzniveau von 1 %, wird der gemessene Unterschied üblicherweise als nicht signifikant betrachtet. Einen weiteren Signifikanztest zeigt die Spalte Δ/σ. Sie gibt die Differenz der gemittelten Laufzeiten, relativ zur maximalen Standardabweichung der beiden Messungen an. Ist dieser Wert kleiner als $\pm 200\,\%$ und die Differenz damit geringer als die doppelte Standardabweichung, wird diese nicht als signifikant betrachtet. Zur Veranschaulichung der Signifikanz sind fehlschlagende Signifikanztests und die betroffenen Speedups in den Tabellen rot dargestellt.

Tabelle 4.1 zeigt die Ergebnisse des Benchmarks mit dem IA-32-Back-End. Von den mit ExtTSP kompilierten Programmen sind 5 von 15 signifikant schneller als die mit Normal kompilierten. 2 von 15 Programmen sind mit ExtTSP signifikant langsamer. Für 8 der 15 Programme lassen sich keine signifikanten Unterschiede zwischen den beiden Algorithmen erkennen. Davon erzeugen ExtTSP und Normal bei 5 von diesen vergleichbar schnelle Programme. Die restlichen 3 Programme weisen Unterschiede zwischen den Algorithmen auf, die im Vergleich zur Standard-

[3] `https://www.spec.org/cpu2000/`.

abweichung zu gering sind. Von den 4 Floating-point-Programmen zeigte keines signifikante Unterschiede. Im Fall von `177.mesa` und `183.equake` scheint dies auf die hohe Standardabweichung zurückführbar zu sein. Ihre Messungen enthielten mehrere starke Ausreißer. Am meisten von ExtTSP profitiert der Benchmark `175.vpr` mit einem Speedup von 3,9 %. Das im Vergleich zu Normal schlechteste Ergebnis liefert `253.perlbmk` mit einem negativen Speedup von −5,9 %. Dieser zeigte, trotz niedriger Standardabweichung und signifikanten Auswirkungen, zwischen verschiedenen Benchmarkläufen um mehrere Prozent schwankende Ergebnisse. Im Mittel, über die Benchmarks für die signifikante Unterschiede gemessen werden konnten, scheint ExtTSP mit 0,34 % Speedup leicht schnellere IA-32-Programme zu erzeugen.

Tabelle 4.1: Ergebnisse des SPEC CPU2000-Benchmark mit dem IA-32-Back-End. Die Laufzeiten entsprechen dem arithmetischen Mittel der Ausführungszeiten in Sekunden über 35 Iterationen. σ ist die jeweilige Standardabweichung. Die Spalte t-Test schätzt die Wahrscheinlichkeit, dass gemessene Unterschiede bloß auf zu wenige Messungen zurrückzuführen sind. Δ/σ ist die Differenz der durchschnittlichen Laufzeiten, relativ zur maximalen Standardabweichung der beiden Messungen. Speedup gibt die Steigerung der Ausführungsgeschwindigkeit von ExtTSP gegenüber Normal an. Die Farbe rot markiert das Fehlen von Signifikanz.

	Normal		ExtTSP				
Benchmark	Laufzeit	σ	Laufzeit	σ	t-Test	Δ/σ	Speedup
164.gzip	64,77	0,10	65,21	0,13	0,00 %	-324,1 %	-0,665 %
175.vpr	48,50	0,14	46,67	0,12	0,00 %	1313,0 %	3,906 %
176.gcc	22,70	0,05	22,23	0,06	0,00 %	848,2 %	2,109 %
181.mcf	24,15	0,09	24,22	0,12	1,39 %	-54,4 %	-0,267 %
186.crafty	27,71	0,06	27,34	0,05	0,00 %	604,5 %	1,335 %
197.parser	58,94	0,12	58,33	0,12	0,00 %	485,8 %	1,040 %
253.perlbmk	57,57	0,18	61,21	0,26	0,00 %	-1417,4 %	-5,952 %
254.gap	26,29	0,11	26,05	0,11	0,00 %	207,3 %	0,912 %
255.vortex	41,59	0,19	41,55	0,16	38,21 %	19,4 %	0,088 %
256.bzip2	49,21	0,14	49,33	0,13	0,02 %	-89,9 %	-0,253 %
300.twolf	64,03	0,24	64,35	0,28	0,00 %	-114,3 %	-0,496 %
177.mesa	63,89	0,21	64,47	2,00	9,29 %	-29,0 %	-0,900 %
179.art	26,05	0,45	26,06	0,38	86,07 %	-3,9 %	-0,067 %
183.equake	25,78	0,11	26,16	2,03	27,88 %	-18,5 %	-1,432 %
188.ammp	87,43	0,14	87,71	0,27	0,00 %	-105,3 %	-0,318 %
Geom. Mittel							-0,085 %
Geom. Mittel (signifikant)							0,341 %

Tabelle 4.2 zeigt die Ergebnisse des Benchmarks mit dem AMD64-Back-End. Von den 15 Benchmark-Programmen sind 8 signifikant schneller, wenn sie mit ExtTSP kompiliert wurden. Nur eines der 15 Programme ist mit ExtTSP signifikant langsamer als mit Normal. Für 6 Programme lassen sich keine signifikanten Unterschiede zwischen den beiden Algorithmen feststellen. Dabei erzeugen ExtTSP und Normal für `177.mesa` vergleichbar schnelle Programme. Die negativen und positiven Auswirkungen die ExtTSP auf die übrigen 5 Programme hat, sind im Vergleich zur Standardabweichung zu gering. Im Fall von `181.mcf` ist dies trotz des hohen negativen Speedups von −4,3 % der Fall, da auch die Standardabweichung sehr hoch ist. Diese hohen Werte werden durch starke Ausreißer in der ExtTSP-Messung verursacht. Eine Berechnung des Speedups auf Basis der Minima der beiden Messungen (33,64 s und 33.76 s) ergibt lediglich einen negativen Speedup von −0.28 %. Am meisten von ExtTSP profitiert hat der Benchmark `179.art` mit 7,6 % Speedup. Den größten signifikanten negativen Effekt hat ExtTSP auf den Benchmark `256.bzip2`, der im Vergleich zu Normal 1,3 % langsamer ist. Im Mittel, über alle Benchmarks für die signifikante Unterschiede gemessen werden konnten, scheint ExtTSP 2,16 % schnellere AMD64-Programme zu erzeugen.

Tabelle 4.2: Ergebnisse des SPEC CPU2000-Benchmark mit dem AMD64-Back-End. Die Bedeutung der verschiedenen Spalten wird in der Beschreibung von Tabelle 4.1 erläutert.

	Normal		ExtTSP				
Benchmark	Laufzeit	σ	Laufzeit	σ	t-Test	Δ/σ	Speedup
164.gzip	71,01	0,11	70,66	0,15	0,00 %	236,3 %	0,497 %
175.vpr	43,55	0,14	43,21	0,11	0,00 %	244,1 %	0,806 %
176.gcc	27,00	0,04	26,37	0,05	0,00 %	1164,2 %	2,386 %
181.mcf	34,17	0,39	35,73	0,96	0,00 %	-163,2 %	-4,378 %
186.crafty	27,57	0,05	27,50	0,05	0,00 %	126,7 %	0,242 %
197.parser	69,90	0,53	69,03	0,14	0,00 %	162,7 %	1,259 %
253.perlbmk	72,49	0,33	71,45	0,35	0,00 %	299,8 %	1,457 %
254.gap	28,25	0,09	27,58	0,13	0,00 %	516,2 %	2,428 %
255.vortex	42,38	0,11	40,78	0,16	0,00 %	1030,1 %	3,941 %
256.bzip2	51,95	0,13	52,66	0,10	0,00 %	-566,2 %	-1,346 %
300.twolf	74,92	0,34	75,56	0,28	0,00 %	-188,4 %	-0,846 %
177.mesa	27,94	0,14	27,90	0,07	17,24 %	25,8 %	0,132 %
179.art	27,16	0,10	25,24	0,06	0,00 %	1845,2 %	7,616 %
183.equake	20,63	0,17	20,99	0,30	0,00 %	-119,6 %	-1,711 %
188.ammp	57,37	0,16	56,27	0,15	0,00 %	683,3 %	1,954 %
Geom. Mittel							0,928 %
Geom. Mittel (signifikant)							2,166 %

Insgesamt sind die Ergebnisse mit den beiden Back-Ends also sehr unterschiedlich. Es gibt Benchmarks, die mit beiden Back-Ends signifikant von ExtTSP profitieren (`175.vpr`, `176.gcc`, `254.gap`). Des Weiteren gibt es 5 Benchmarks, die mit einem der Back-Ends signifikant von ExtTSP profitieren, jedoch mit dem anderen nicht das Signifikanzniveau erreichen. Benchmarks, die mit beiden Back-Ends signifikant negativ durch ExtTSP beeinflusst werden, gibt es nicht. Der Benchmark `256.bzip2` wird mit AMD64 durch ExtTSP langsamer, aber mit IA-32 zeigt er keine signifikanten Veränderungen. Weitere 4 Benchmarks zeigen mit keinem der Back-Ends signifikante Unterschiede zwischen Normal und ExtTSP. Die beiden Benchmarks `164.gzip` und `253.perlbmk` zeigen mit beiden Back-Ends unterschiedliche signifikante Änderungen durch ExtTSP: sie werden im Vergleich zu Normal mit IA-32 langsamer, mit AMD64 jedoch schneller.

Woher diese Unterschiede im Detail stammen ist unklar. Es ist zu vermuten, dass sie in den unterschiedlichen Implementierungen der Back-Ends begründet sind. Diese implementieren verschiedene Optimierungen und verwenden beispielsweise unterschiedliche Befehlssatzerweiterungen für Floating-point-Berechnungen.

Eine Gemeinsamkeit der beiden Back-Ends im Bezug auf ExtTSP konnte mit dem Programm `perf stat` beobachtet werden: für alle Programme des SPEC CPU2000-Benchmarks stieg die Zahl der ausgeführten Sprünge, wenn mit ExtTSP kompiliert wurde. Gleichzeitig sank jedoch der Anteil der falsch vorhergesagten Sprünge (engl. branch misses) im Vergleich zu Normal. Außerdem sank auch die Zahl der Instruktionscache-Misses für einige der Benchmarks. Eine Korrelation mit den erreichten positiven und negativen Speedups konnte allerdings nicht festgestellt werden.

Zusammengefasst scheinen viele Programme von der Grundblockanordnung mit dem ExtTSP-Algorithmus zu profitieren. Die Performanz einiger Programme leidet jedoch auch im Vergleich zu Normal. Besonders in Kombination mit dem AMD64-Back-End scheint sich der Einsatz von ExtTSP mit Speedups von bis zu 7,6 % zu lohnen.

4.3 Auswirkungen auf die Compilerlaufzeit

Um zu untersuchen, welche Auswirkungen der Einsatz des ExtTSP-Algorithmus auf die Laufzeit von libFirm hat, wurde wie im vorherigen Abschnitt der SPEC CPU2000-Benchmark verwendet. Alle Benchmarks wurden mithilfe der SPEC-Suite 5 Mal mit dem AMD64-Back-End kompiliert. Dies wurde sowohl für Normal, als auch für ExtTSP durchgeführt. Dabei wurde für jede kompilierte Funktion gemessen, wie viel Zeit der verwendete Algorithmus zur Grundblockanordnung benötigt. Außerdem wurde die Ausführungszeit des gesamten Back-Ends, sowie der

einzelnen Komponenten, gemessen.

Tabelle 4.3 zeigt die für jeden Benchmark zusammengefassten Ergebnisse. Die Spalte „Blöcke“ enthält die Summe der Grundblöcke des Benchmarks. „F.“ ist die Zahl der Funktionen und „$\max_F$“ die Zahl der Grundblöcke der größten Funktion. Sowohl für Normal als auch für ExtTSP ist die Gesamtlaufzeit t der Grundblockanordnung in Mikrosekunden angegebenen. Die Spalte „$\frac{t_e}{t_n}$“ gibt an, um welchen Faktor ExtTSP langsamer als Normal ist. Des Weiteren gibt die Spalte „Anteil“ für beide Algorithmen an, welchen Anteil sie an der Gesamtlaufzeit des Back-Ends haben. Der Anteil ist rot markiert, wenn die Grundblockanordnung, im Vergleich zu den anderen Komponenten des Back-Ends, den größten Anteil an der Gesamtlaufzeit ausmacht. Die Tabelle ist nach der Laufzeit t_e der ExtTSP-Grundblockanordnung sortiert.

Wie zu erwarten, ist ExtTSP deutlich langsamer als Normal. Im kleinsten Fall, auf dem Benchmark `183.equake`, ist ExtTSP um den Faktor 18 langsamer als Normal. Der größte Unterschied besteht bei `177.mesa`. Für diesen ist ExtTSP 278 mal langsamer als Normal. Es fällt auf, dass der Unterschied mit steigender Grundblock- und Funktionszahl größer wird. Außerdem scheinen gerade sehr große Funktionen den Unterschied stark zu beeinflussen. Dies zeigt sich daran, dass die Benchmarks mit dem größten Unterschied zwischen den Algorithmen (`186.crafty`, `177.mesa`, `253.perlbmk`, `176.gcc`) auch die Benchmarks mit den größten maximalen Funktionen sind.

Im Vergleich mit der Gesamtlaufzeit des Back-Ends liegt Normal in fast allen Fällen unter einem Anteil von 1 %. ExtTSP hingegen beansprucht auf den Benchmarks zwischen 8,8 % und 58,8 %. Für 10 der 15 Benchmarks ist ExtTSP dabei nicht die Back-End-Komponente mit dem höchsten Anteil an der Laufzeit. Im Fall der restlichen 5 Benchmarks, macht die Grundblockanordnung durch ExtTSP den größten Anteil an der Back-End-Laufzeit aus. Ob dieser Aufwand im Hinblick auf die erreichten Speedups vertretbar ist, muss abgewogen werden.

4.4 Auswirkungen des Parameters K

Da der ExtTSP-Algorithmus, wie bereits in Unterabschnitt 3.4.2 erläutert, einen Parameter K besitzt, wurden dessen Auswirkungen auf die Qualität des Ergebnisses und die Compilerlaufzeit untersucht. Der Parameter ist ein Schwellwert und legt fest, bis zu welcher Länge die Grundblockketten beim Verschmelzen komplex rekombiniert werden. Hat eine Kette eine Länge größer K, wird sie nicht mehr zerteilt und bloß noch mit anderen Ketten konkateniert. Hierdurch soll die Laufzeit für Funktionen mit sehr vielen Blöcken reduziert werden. Für ihren Anwendungsfall wählen Newell und Pupyrev [3] einen Schwellwert von K = 128. Für die vorherigen Messungen

Tabelle 4.3: Die Ergebnisse des Compiler-Laufzeitvergleichs zwischen NORMAL und EXTTSP. „Blöcke“ gibt die Anzahl der Grundblöcke an. „F.“ ist die Zahl der Funktionen und „$\max_F$“ die Zahl der Grundblöcke der größten Funktion. Die Zeiten sind die akkumulierten Laufzeiten der Algorithmen in Mikrosekunden. „$\frac{t_e}{t_n}$“ gibt an, um welchen Faktor EXTTSP langsamer als NORMAL ist. Die prozentualen Anteile beider Algorithmen stellen deren Anteil an der Gesamtlaufzeit des Back-Ends dar. Diese sind rot, falls der Anteil im Vergleich zu den anderen Back-End-Komponenten am größten ist. Die Tabelle ist nach der Spalte t_e sortiert.

				NORMAL		EXTTSP		
Benchmark	Blöcke	F.	$\max_F$	Zeit t_n	Anteil	Zeit t_e	Anteil	$\frac{t_e}{t_n}$
183.equake	465	27	141	940	0,5 %	17222	8,8 %	18
179.art	641	26	116	1093	0,7 %	38446	18,4 %	35
181.mcf	580	26	165	958	1,0 %	75134	42,3 %	78
188.ammp	4178	179	309	7304	0,6 %	170148	11,9 %	23
164.gzip	2338	69	143	3653	0,5 %	189143	21,1 %	52
256.bzip2	1952	74	171	3397	0,7 %	261997	32,5 %	77
175.vpr	5717	175	215	9417	0,5 %	842236	31,9 %	89
300.twolf	8201	190	336	14417	0,5 %	1346220	31,2 %	93
186.crafty	6905	109	1188	13013	0,4 %	1936279	34,0 %	149
197.parser	11715	323	262	19413	0,6 %	2206867	40,8 %	114
254.gap	33827	849	704	54785	0,6 %	3373236	26,2 %	62
255.vortex	27420	923	586	48137	0,4 %	4802215	25,6 %	100
177.mesa	29884	1039	1060	42536	0,5 %	11817224	58,8 %	278
253.perlbmk	50142	1020	2081	87998	0,6 %	21119232	57,6 %	240
176.gcc	105630	2052	1253	178120	0,5 %	34305086	49,8 %	193

wurde dieser Wert ebenfalls verwendet.

Um die Auswirkungen der Wahl von K zu untersuchen, wurde das Programm `176.gcc` aus dem SPEC CPU2000-Benchmark verwendet. Für dieses benötigte der ExtTSP-Algorithmus in Abschnitt 4.3 die meiste Zeit. Außerdem enthält es, unter den Programmen des Benchmarks, die größte Anzahl von Grundblöcken und mindestens eine sehr große Funktion, bestehend aus 1253 Grundblöcken.

Der Benchmark wurde mithilfe der SPEC-Suite mit verschiedenen Wahlen des K-Parameters und dem AMD64-Back-End, kompiliert und mit jeweils 10 Iterationen auf seine Ausführungszeit vermessen. Außerdem wurden, wie im vorherigen Abschnitt, die Laufzeit der Grundblockanordnung und der einzelnen Back-End-Komponenten gemessen. Zusätzlich wurde er zum Vergleich auch mit dem Normal-Algorithmus und 10 Iterationen kompiliert.

Tabelle 4.4 veranschaulicht die Ergebnisse der beschriebenen Tests. Die Spalte „Laufzeit“ gibt jeweils die gesamte Laufzeit der Grundblockanordnung in Sekunden an. Der „Back-End-Anteil“ ist, analog zu den Anteilen im vorherigen Abschnitt, der Anteil der Grundblockanordnung, an der Gesamtlaufzeit des Back-Ends. Diese sind wieder rot, wenn die Grundblockanordnung den größten Anteil an der Laufzeit hat. Ebenfalls analog zum vorherigen Abschnitt ist $\frac{t_e}{t_n}$ der Faktor, um den der ExtTSP-Algorithmus langsamer als Normal ist. Die Spalte „Speedup“ gibt, analog zum Abschnitt 4.2, den Speedup des von ExtTSP erzeugten Benchmarks gegenüber Normal an. Bis auf den Fall $K = 2$ waren diese immer signifikant und die Standardabweichungen waren vernachlässigbar klein.

Wie zu erwarten zeigt sich, dass durch die Wahl kleinerer Werte für K die Laufzeit tatsächlich reduziert werden kann. Schon bei einer Halbierung des Wertes von 128 auf 64, halbiert sich auch die Laufzeit. Der Anteil der Grundblockanordnung an der Back-End-Laufzeit, fällt ebenfalls um die Hälfte und ist so nicht mehr der größte Back-End-Anteil. Gleichermaßen sinkt der Faktor um den ExtTSP langsamer als Normal ist.

Außerdem ist erkennbar, dass die Höhe des Schwellwertes die Qualität der gefundenen Grundblockanordnungen direkt beeinflusst: Je höher der für K gewählte Wert, desto höher ist in den meisten Fällen auch der Speedup. Die Fälle in denen ein höheres K zu einem schlechteren Ergebnis führt, weisen darauf hin, dass der ExtTSP-Score (siehe Abschnitt 3.2) kein perfektes Maß für die Performanz einer Grundblockanordnung ist: Durch den höheren Schwellwert K findet der Algorithmus Grundblockanordnungen, mit höheren ExtTSP-Scores, die, den geringeren Speedups zufolge, jedoch nicht performanter sind. Die Wahl eines $K > 128$ mehr als verdoppelt die Laufzeit jeweils in den Schritten zu 256 und 512. Allerdings scheint sich dieser Mehraufwand auch auszuzahlen, da der erreichte Speedup noch einmal weiter auf 3,6 % und 3,9 % steigt und sich damit mehr als verdoppelt. Der letzte Schritt von $K = 1024$

auf $K = 2048$, scheint, durch die Fehlerhafitgkeit des ExtTSP-Scores, erneut unvorteilhaft beeinflusst zu werden. In diesem fällt der Speedup um 1 % ab, obwohl nun auch die Ketten in der größten Funktion des Benchmarks, aufgebrochen werden können.

Tabelle 4.4: Ergebnisse des `176.gcc`-Benchmark, mit verschiedenen Wahlen für den Parameter K des ExtTSP-Algorithmus. Die Laufzeiten geben die gesamte Laufzeit der Grundblockanordnung auf dem Benchmark in Sekunden an. Die Spalte „Back-End-Anteil“ gibt den Anteil der Grundblockanordnung, an der Gesamtlaufzeit des Back-Ends, an. Dieser Wert ist rot, wenn die Grundblockanordnung den größten Anteil an der Laufzeit des Back-Ends ausmacht. $\frac{t_e}{t_n}$ gibt den Faktor, um den ExtTSP langsamer als Normal ist, an. Die Spalte „Speedup“ gibt den Speedup des Benchmarks gegenüber Normal an.

K	Laufzeit [s]	Back-End-Anteil	$\frac{t_e}{t_n}$	Speedup
1	4,36	11,2 %	25	-1,052 %
2	4,68	11,9 %	27	0,191 %
4	5,01	12,7 %	28	1,163 %
8	5,46	13,6 %	31	0,432 %
16	6,25	15,3 %	35	0,740 %
32	7,35	17,5 %	42	1,418 %
64	12,22	26,1 %	69	1,856 %
128	34,45	49,9 %	195	1,733 %
256	87,88	71,8 %	498	3,601 %
512	267,31	88,5 %	1515	3,917 %
1024	420,66	92,4 %	2384	3,914 %
2048	436,87	92,7 %	2476	2,927 %

5 Fazit und Ausblick

In dieser Arbeit wurde ein Ansatz zur Verbesserung der Grundblockanordnung der libFirm-Bibliothek vorgestellt und in diese integriert. Hierfür wurde ein Algorithmus, aus dem Gebiet der Profile-guided Optimizations, auf das Problem der Grundblockanordnung in einem Compiler angewendet. Anschließend wurde dieser mit der bisherigen Implementierung verglichen und dessen Auswirkungen auf libFirm untersucht.

Die Evaluation zeigte, dass die Wirksamkeit des neuen Ansatzes sowohl von der verwendeten Zielarchitektur, als auch den konkreten kompilierten Programmen abhängt. Für die IA-32-Architektur konnte so festgestellt werden, dass manche Programme durch den neuen Ansatz bis zu 3,9 % schneller wurden. Jedoch gab es hier auch Programme, welche deutlich an Geschwindigkeit verloren haben. Schlussendlich konnte für viele Programme kein signifikanter Unterschied zwischen der bisherigen Implementierung und dem neuen Ansatz festgestellt werden, wodurch dieser für IA-32 daher im Mittel nur leicht schnellere Programme erzeugte. Hingegen konnte mit dem AMD64-Back-End für die Mehrzahl der untersuchten Programme ein signifikanter Speedup durch den neuen Ansatz, festgestellt werden. Dabei wurden Speedups von bis zu 7,6 % und im Mittel über 2,1 % erreicht. Trotzdessen konnte keine durchgängige Verbesserung für alle getesteten Programme beobachtet werden. Für einen Teil der Programme konnte auch mit AMD64 kein signifikanter Unterschied, im Vergleich zur bisherigen Implementierung, gemessen werden.

Weiterhin konnte beobachtet werden, dass der neue Ansatz aufgrund seiner Komplexität deutlich mehr Zeit zum Finden einer Grundblockanordnung benötigt. Für sehr große Programme stieg die Laufzeit dabei so weit an, dass sie die Gesamtlaufzeit des Back-Ends dominierte. Wie daraufhin jedoch gezeigt wurde, lässt sich die Laufzeit durch entsprechende Parameterwahl, auf Kosten der Ergebnisqualität, beeinflussen und so auf ein akzeptables Maß reduzieren.

Zusammengefasst lässt sich also positiv feststellen, dass das Ziel dieser Arbeit erreicht und ein neuer Ansatz zur Grundblockanordnung implementiert werden konnte, der für einige Programme tatsächlich bessere Ergebnisse erzeugt. Vor allem in Kombination mit dem AMD64-Back-End, scheint sich der Aufwand gelohnt zu haben, da auf diesem, mit im Schnitt über 2 % Speedup, eine deutliche Verbesserung erreicht wurde. Ebenfalls konnte die Fragestellung, ob der eingesetzte ExtTSP-Algorithmus auch ohne exakte Profiling-Daten noch gute Ergebnisse erzielt, mit Ja beantwortet

werden.

Die weitere Verwendung der Implementierung in libFirm kann als Ergänzung zur bisherigen Implementierung klar empfohlen werden, da viele Programme deutlich von der neuen Grundblockanordnung profitieren. Ob sie allerdings als Standardeinstellung verwendetet werden sollte, ist gerade im Fall von IA-32 abzuwägen. Im Fall des AMD64-Back-Ends scheint die ExtTSP-Grundblockanordnung jedoch ein potenter Ersatz für die bisherige Implementierung zu sein. Durch die Wahl eines niedrigeren K-Parameters, wie beispielsweise 64, können die Laufzeiten in einem akzeptablen Rahmen gehalten und dennoch deutliche Speedups, gegenüber der bisherigen Implementierung, erreicht werden. Eine variable Wahl des Parameters, in Abhängigkeit des gewählten Optimierungslevels, ist ebenfalls denkbar.

Zukünftig könnten die Auswirkungen, der durch Schätzungen ersetzten Profiling-Daten, noch weiter analysiert werden. So wäre beispielsweise interessant, ob der ExtTSP-Algorithmus auf Architekturen mit fester Instruktionsgröße, wie SPARC und MIPS, von der exakten Grundblockgrößenschätzung profitiert und bessere Ergebnisse erzielt. Außerdem könnten die implementierten Größenschätzungen noch verbessert werden.

Des Weiteren besteht auch bei der Schätzung der Ausführungshäufigkeiten noch Optimierungspotential. Die bisherige Implementierung in libFirm betrachtet beispielsweise noch nicht, ob bedingte Sprünge zu einer Ausnahmebehandlung oder ähnlichem gehören. Von einer solchen Optimierung würden auch andere, der in libFirm implementierten Optimierungen, profitieren. Der Ansatz [14] von Wu und Larus, welcher unter anderem auch in der GCC verwendet wird, liese sich beispielsweise leicht in libFirms bestehende Implementierung integrieren.

Darüber hinaus gibt es noch zahlreiche weitere Optimierungsansätze, die sich ebenfalls mit der Anordnung von Programmcode beschäftigen. Exemplarisch wäre die Erforschung des Zusammenspiels aus Grundblockanordnung und der, in Abschnitt 2.5 bereits erwähnten, „Hot-Cold“-Optimierung interessant. Ein weiterer vielversprchender Forschungsansatz wäre, die Grundblockanordnung, auch über die Grenzen von Funktionen hinweg, einzusetzen.

Literaturverzeichnis

[1] L. Torczon and K. Cooper, *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2nd ed., 2011.

[2] U. Drepper, "What every programmer should know about memory." `http://people.redhat.com/drepper/cpumemory.pdf`, 2007.

[3] A. Newell and S. Pupyrev, "Improved basic block reordering," *arXiv preprint arXiv:1809.04676*, 2018.

[4] F. T. Boesch and J. F. Gimpel, "Covering points of a digraph with point-disjoint paths and its application to code optimization," *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 192–198, 1977.

[5] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *ACM SIGPLAN Notices*, vol. 25, pp. 16–27, ACM, 1990.

[6] A. Ramirez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *Proceedings of the 13th international conference on Supercomputing*, pp. 119–126, ACM, 1999.

[7] C. Young, D. S. Johnson, M. D. Smith, and D. R. Karger, "Near-optimal intraprocedural branch alignment," *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 183–193, 1997.

[8] X. Liu, J. Zhang, K. Liang, Y. Yang, and X. Cheng, "Basic-block reordering using neural networks," in *Proceedings of SMART Workshop*, p. 12, 2007.

[9] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pp. 233–244, IEEE Press, 2017.

[10] R. Cohn and P. G. Lowney, "Hot cold optimization of large Windows/NT applications," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pp. 80–89, IEEE, 1996.

[11] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[12] G. Lindenmaier, "libfirm – a library for compiler optimization research implementing firm," Tech. Rep. 2002-5, Sept. 2002.

[13] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: a practical binary optimizer for data centers and beyond," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 2–14, IEEE Press, 2019.

[14] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th annual international symposium on Microarchitecture*, pp. 1–11, ACM, 1994.

Erklärung

Hiermit erkläre ich, Christoph Breisacher, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift