Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

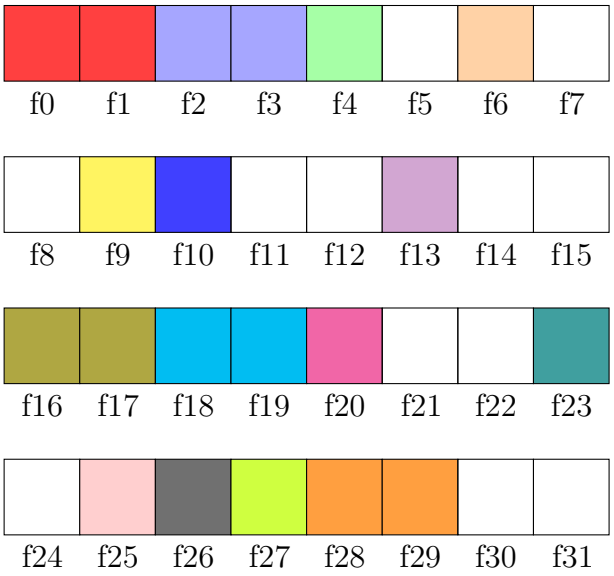
Lehrstuhl Prof. Dr.-Ing. Snelting

SSA-basierte Doppelregisterallokation

Bachelorarbeit von

Johannes Bucher

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M.Sc. Andreas Fried

Bearbeitungszeit: 28. November 2017 – 26. März 2018

Zusammenfassung

Registerallokation ist eine wichtige Aufgabe während dem Kompilieren von Programmen, um den Variablen Register zuzuteilen. Ein häufig genutzter Ansatz für die Registerallokation ist die Übertragung auf das Problem der Graphfärbung. Arbeitet die Registerallokation auf einer Zwischenrepräsentation in SSA-Form, ergeben sich Vereinfachungen für die genutzten Algorithmen.

Die SPARC-Architektur bietet die Möglichkeit, in einem Paar von benachbarten 32-Bit-Registern eine Gleitkommazahl doppelter Genauigkeit zu speichern, die 64 Bit Speicherplatz benötigt. In dieser Arbeit wird die Auswirkung der Verwendung solcher Doppelregister auf die Registerallokation in einem SSA-basierten Registerallokator anhand der Registerallokation in libFirm untersucht. Die bestehende Implementierung wird angepasst, um Doppelregister zu unterstützen. Die abschließende Bewertung zeigt, dass die Laufzeit von mit libFirm für die SPARC-Architektur übersetzten Programmen, die viele Gleitkommaberechnungen doppelter Genauigkeit durchführen, dadurch erheblich reduziert wird.

Inhaltsverzeichnis

1. Einführung **7**

2. Grundlagen und verwandte Arbeiten **9**
2.1. SSA-Form . . . 9
2.2. libFirm . . . 10
2.3. SPARC-Architektur . . . 12
2.3.1. Überblick . . . 12
2.3.2. Floating-Point-Register . . . 12
2.4. Registerallokation . . . 14
2.5. Verwandte Arbeiten . . . 16

3. Entwurf und Implementierung **17**
3.1. Ausgangssituation in libFirm . . . 17
3.2. Repräsentation von Doppelregistern . . . 18
3.3. Spilling . . . 19
3.3.1. Workset . . . 20
3.3.2. Ablauf . . . 20
3.3.3. Anpassung für Doppelregister . . . 21
3.4. Registerzuteilung . . . 23
3.4.1. Registereinschränkungen verarbeiten . . . 23
3.4.2. Schreib- und Lesezugriffe speichern . . . 27
3.4.3. Färben . . . 27
3.5. Auflösung von Permutationsknoten . . . 30
3.5.1. Permutationsknoten . . . 30
3.5.2. Umwandlung in Kopien . . . 31
3.5.3. Aufteilung in Einzelregister . . . 33
3.5.4. RegSplit- und RegJoin-Knoten . . . 34

4. Evaluation **37**
4.1. Testumgebung . . . 37
4.2. Korrektheit . . . 38
4.3. Performanz . . . 39

5. Fazit und Ausblick **41**
5.1. Fazit . . . 41

5.2. Ausblick 41
5.2.1. Verringerung der eingefügten Permutationsknoten 41
5.2.2. Kopienminimierung 42
5.2.3. Gleitkommazahlen vierfacher Genauigkeit 42

A. Anhang **49**

1. Einführung

Zahlreiche moderne Prozessorarchitekturen nutzen eine Wortbreite von 64 Bit. Je nach Anwendungsfeld sind aber nach wie vor Prozessoren mit 32-Bit-Architekturen verbreitet. Eine solche Architektur ist die SPARC-Architektur (*Scalable Processor Architecture*). Diese von dem Unternehmen Sun Microsystems entwickelte Architektur wurde erstmals im Jahr 1986 vorgestellt. Aufgrund der Skalierbarkeit der Architektur kommen SPARC-Prozessoren in unterschiedlichsten Anwendungsgebieten zum Einsatz, beispielsweise in Supercomputern und in Systemen, die für den Einsatz im Weltraum konzipiert sind [1]. Eine Gleitkommazahl einfacher Genauigkeit hat auf der SPARC V8-Architektur eine Breite von 32 Bit. Dennoch ist es wünschenswert, auch auf einer 32-Bit-Architektur Berechnungen auf Daten von höherer Genauigkeit, insbesondere Gleitkommazahlen mit doppelter Genauigkeit, durchzuführen. Ein Datenwort für eine Gleitkommazahl doppelter Genauigkeit umfasst 64 Bit. Prozessoren der SPARC-Architektur bringen bereits die nötigen Voraussetzungen mit, um mit solchen Datenwörtern umzugehen. Dazu gehört neben den nötigen Prozessorinstruktionen auch der Ansatz, ein 64-Bit-Datenwort in zwei Registern abzuspeichern. Diese Zusammenfassung zweier 32-Bit-Register zur Speicherung eines 64-Bit-Datenwortes wird mit dem Begriff Doppelregister bezeichnet.

Die Programmierung von Anwendungen erfolgt meist in höheren Programmiersprachen und die Verwaltung der Register des Prozessors wird nicht vom Programmierer übernommen. Im Maschinencode, der auf dem Prozessor ausgeführt wird, muss jedoch eine Zuteilung von Daten des Programms auf die Register enthalten sein. Die Aufgabe, den Variablen im Programm Register zuzuteilen, die Registerallokation, wird daher vom Compiler übernommen.

Ein Compiler kann in mehrere Phasen unterteilt werden. Das Front-End ist zuständig für das Einlesen der Quellsprache und der semantischen und syntaktischen Analyse des Codes. Das Ergebnis wird dem Middle-End übergeben, das zahlreiche Optimierungen vornimmt. Anschließend übernimmt das Back-End die Aufgabe, prozessorspezifischen Maschinencode zu generieren, wobei auch weitere der Zielarchitektur angepasste Optimierungen vorgenommen werden können. Die Registerallokation ist Teil des Back-Ends. Eine Zwischenrepräsentation stellt das Austauschformat zwischen den Phasen des Compilers dar. Durch Nutzung einer Zwischenrepräsentation wird der Einsatz verschiedener Back-Ends für verschiedene Architekturen unter Beibehaltung der Funktionen des Front- und Middle-Ends vereinfacht.

libFirm ist eine am Karlsruher Institut für Technologie entwickelte Bibliothek für die graphbasierte Zwischenrepräsentation Firm. Eine besondere Eigenschaft von Firm ist die SSA-Form, die verschiedene Optimierungen und Algorithmen erheblich vereinfacht und beschleunigt.

libFirm beinhaltet bereits die Möglichkeit, aus der Zwischenrepräsentation Maschinencode für die SPARC-Prozessorarchitektur zu erzeugen. Es ist bisher jedoch kein geeigneter Registerallokator enthalten, der mit Doppelregistern umgehen kann. Programme, die für die SPARC-Architektur kompiliert werden sollen und 64 Bit breite Gleitkommazahlen verwenden, müssen deshalb bisher auf langsamere, in Software emulierte Gleitkommaarithmetik zurückgreifen.

In der vorliegenden Arbeit wird eine Erweiterung der bestehenden Registerallokation von libFirm für die Allokation von Doppelregistern unter Ausnutzung der SSA-Form der Zwischenrepräsentation entwickelt und bewertet.

Die zum Verständnis der Aufgabenstellung und gewählten Implementierung nötigen Grundlagen werden in Kapitel 2 eingeführt. Kapitel 3 erläutert die gewählte Implementierung der Doppelregisterallokation. Anschließend wird in Kapitel 4 die Implementierung hinsichtlich korrekter Funktionsweise und Laufzeitverhalten evaluiert. Kapitel 5 fasst die Ergebnisse dieser Arbeit kurz zusammen und gibt einen Ausblick auf mögliche zukünftige Verbesserungen.

2. Grundlagen und verwandte Arbeiten

In diesem Kapitel werden einige Grundlagen erklärt, die zum Verständnis der Implementierung nötig sind. Zu Beginn wird auf Zwischenrepräsentationen in SSA-Form eingegangen und eine kurze Einführung zu libFirm gegeben. Anschließend werden die Eigenschaften und Anforderungen der SPARC-Architektur bezüglich der Verwendung von Doppelregistern dargelegt. Nach einer Einführung in die Registerallokation mittels Graphfärbung werden die Vorteile der SSA-Form auf die Registerallokation erläutert. Abschließend werden verwandte Arbeiten zur Doppelregisterallokation genannt.

2.1. SSA-Form

Beim Übersetzen eines Programms in Quellsprache wird der vom Programmierer geschriebene Code üblicherweise in verschiedenen Phasen in eine Zwischenrepräsentation, genannt *intermediate representation (IR)* transformiert. Diese Abstraktionsebene dient dann als Basis für weitere Anpassungen und Optimierungen, die dadurch unabhängig von der Quellsprache durchgeführt werden können. Zum Ende hin wird aus der Zwischenrepräsentation der Code in der gewünschten Maschinensprache generiert. Die Benutzung einer Zwischenrepräsentation im Compiler erlaubt somit eine einfache Trennung von Front-End und Back-End des Compilers. Dadurch können etwa Front-Ends für verschiedene Sprachen mit dem gleichen Back-End eines Compilers kombiniert werden. Optimierungen können einfach im gemeinsamen Middle-End implementiert werden und stehen den verschiedenen Front- und Back-Ends zur Verfügung.

Eine besondere Eigenschaft für Zwischenrepräsentationen ist die SSA-Form. SSA steht für *Static Single Assignment* und bezeichnet die Eigenschaft, dass jeder Variable nur einmalig ein Wert zugewiesen wird. Jede weitere Zuweisung an eine Variable hat zufolge, dass in der SSA-Form eine neue Instanz für diese Variable eingeführt werden muss. Jede Zuweisung an eine Variable wird als Definition eines neuen Wertes bezeichnet. In SSA-Form repräsentiert eine Variable somit exakt einen Wert. Operationen

definieren diese Werte (*def*) und hängen von bereits definierten Werten ab. Dies wird als *use* bezeichnet. Liegt die Zwischenrepräsentation eines Programms in SSA-Form vor, sind zahlreiche Optimierungen und Algorithmen, die auf der Zwischenrepräsentation ausgeführt werden, einfacher zu realisieren [2, 3]. In Abschnitt 2.4 wird gesondert auf die Auswirkungen und Vorteile der SSA-Form für die Registerallokation eingegangen.

Listing 2.1: Ein Codeausschnitt, der sich nicht in SSA-Form befindet.

```
a = some_input();
if (a < 20) {
    b = 1;
} else {
    b = 0;
}
do_stuff(b);
b = 2;
```

Listing 2.2: Der gleiche Codeausschnitt in SSA-Form.

```
a = some_input();
if (a < 20) {
    b1 = 1;
} else {
    b2 = 0;
}
b3 = φ(b1, b2)
do_stuff(b3);
b4 = 2;
```

Listing 2.1 zeigt ein Programm, bei dem der Inhalt einer Variablen (`b`) vom Kontrollfluss des Programms abhängt: Je nach Inhalt der Variable `a` wird `b` ein anderer Wert zugewiesen. In SSA-Form müssen deshalb weitere Variablen eingeführt werden, sodass jeder Variable nur einmalig ein Wert zugewiesen wird. Listing 2.2 verdeutlicht dies. Bei der Zusammenführung des Kontrollflusses nach der if-else-Anweisung muss für weitere Zugriffe auf die Variable `b` eine ϕ-Funktion eingefügt werden. ϕ-Funktionen erhalten als Parameter die Variable aus jedem möglichen Kontrollflusspfad. Je nach tatsächlich eingeschlagenem Pfad gibt die ϕ-Funktion den Wert der entsprechenden Variablen zurück [2, 3]. Die ϕ-Funktion ist keine reale Instruktion und wird nicht im erzeugten Code in Maschinensprache ausgegeben.

Zahlreiche moderne Compiler, darunter GCC (GNU Compiler Collection) und LLVM nutzen zumindest in Teilen interne Zwischenrepräsentationen in SSA-Form [4, 5].

2.2. libFirm

libFirm ist eine C-Bibliothek, welche die am Karlsruher Institut für Technologie entwickelte Zwischenrepräsentation Firm realisiert. Bei Firm handelt es sich um eine vollständig graphbasierte Zwischenrepräsentation, die sich während des gesamten Übersetzungsvorgangs bis zur Codegenerierung in SSA-Form befindet. Die Zwischenrepräsentation in Form eines gerichteten Graphen beinhaltet dabei sowohl

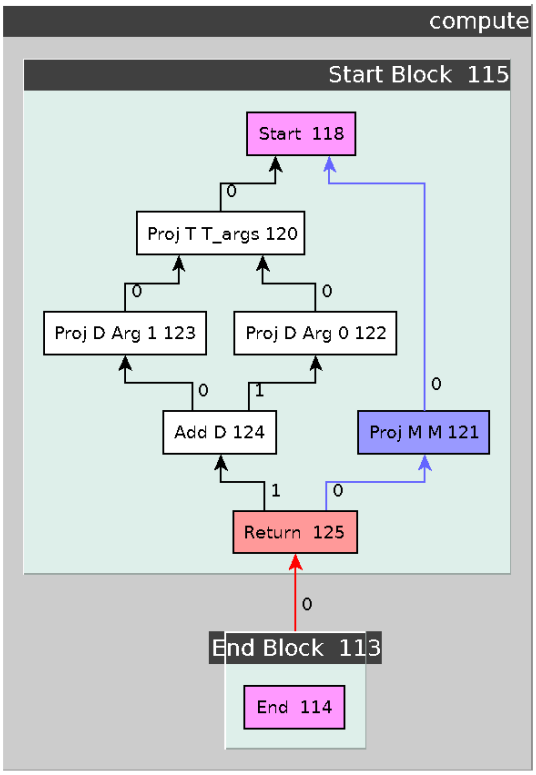


Abbildung 2.1.: Beispielhafte Visualisierung eines FIRM-Graphen für eine Funktion `compute`, die zwei Gleitkommazahlen doppelter Genauigkeit als Parameter erhält und die Summe dieser zurückgibt.

Datenabhängigkeiten als auch den Steuerfluss des Programms. Die Knoten des FIRM-Graphen repräsentieren Operationen. Datenabhängigkeiten werden als Kanten zu den Operationen modelliert, die die benötigten Werte definieren. Anhand der Datenabhängigkeiten kann das *Scheduling* später die Schedule-Reihenfolge festlegen, in der die Operationen ausgeführt werden. Jede Operation ist zudem Teil eines Grundblocks, in dieser Arbeit kurz als *Block* bezeichnet [6, 7]. Abbildung 2.1 zeigt ein einfaches Beispiel für einen Ausschnitt eines FIRM-Graphen.

Es existieren verschiedene Front-Ends, welche die libFIRM-Bibliothek nutzen, unter anderem für die Sprachen Java und C [7]. Das C-Front-End (*cparser*) diente in dieser Arbeit als Grundlage um die entwickelte Funktionalität zu testen.

2.3. SPARC-Architektur

2.3.1. Überblick

Die SPARC-Architektur (SPARC: *Scalable Processor Architecture*) ist eine ab dem Jahr 1985 von dem Unternehmen Sun Microsystems entwickelte Architektur für Mikroprozessoren. Die im Zusammenhang dieser Arbeit betrachtete Version 8 der Architektur wurde 1990 veröffentlicht. Im Folgenden wird mit der SPARC-Architektur implizit Version 8 der Architektur bezeichnet.

Die SPARC-Architektur ist eine 32-Bit-Architektur und gehört zu den RISC-Architekturen. RISC steht für *Reduced Instruction Set Computer* und bedeutet, dass der Befehlssatz der Architektur aus wenigen, einfachen Instruktionen besteht. Des Weiteren handelt es sich bei der SPARC-Architektur um eine Load-Store-Architektur, das heißt die Instruktionen arbeiten ausschließlich mit Operanden, die in den Registern des Prozessors vorliegen. Zum Lesen aus und Schreiben in den Hauptspeicher existieren zusätzlich eigene Lade- und Speicherinstruktionen. Aufgrund dieses Designs verfügen SPARC-Prozessoren, wie für RISC-Architekturen üblich, über eine große Anzahl an Registern im Vergleich zu Architekturen, die komplexere Instruktionen realisieren, sogenannten CISC-Architekturen (*Complex Instruction Set Computer*).

Prozessoren, die die SPARC-Architektur implementieren, verfügen über eine Integer Unit (IU) für Ganzzahlberechnungen und über eine Floating Point Unit (FPU) für Gleitkommaberechnungen. Je nach Implementierung werden diese noch durch einen Coprozessor (CP) ergänzt. Dementsprechend existieren die Registerklassen der General-Purpose-Register und der Floating-Point-Register, zusätzlich sind einige Statusregister vorhanden. Eine Besonderheit der SPARC-Architektur ist das sogenannte Register Window, das aus der großen Menge der General-Purpose-Register jeder Funktion auf Assemblerebene einen Ausschnitt (32 Register) zur Verfügung stellt [8].

2.3.2. Floating-Point-Register

Die Floating Point Unit (FPU) der SPARC-Architektur verfügt über 32 Floating-Point-Register, die von allen Floating-Point-Instruktionen verwendet werden können. Im Gegensatz zu den General-Purpose-Registern wird hier kein Register Window verwendet, alle Funktionen greifen auf die gleichen Register zu. Die Floating-Point-Register müssen daher vom Aufrufer einer Funktion gesichert und danach wiederhergestellt werden (*caller save*). Die Floating-Point-Register tragen Indizes von Null bis 31 und werden dementsprechend mit *f0 – f31* bezeichnet [8].

Eine weitere Besonderheit der SPARC-Architektur ist die Handhabung von Gleitkommazahlen höherer Genauigkeit. Neben den 32 Bit breiten *single precision* Worten bietet SPARC auch *double precision* (64 Bit Breite) und *quad precision* (128 Bit Breite) Worte für Gleitkommazahlen doppelter und vierfacher Genauigkeit. Diese werden ebenfalls in den Floating-Point-Registern gespeichert. Dabei belegt ein Wort mehrere Register: ein Wort doppelter Genauigkeit zwei, ein Wort vierfacher Genauigkeit vier Register [8].

Definition 1. *Ein Doppelregister bezeichnet ein Paar aus unmittelbar aufeinanderfolgenden Registern, in dem ein Wert gespeichert ist, der die Kapazitäten der beiden Einzelregister benötigt. In diesem Zusammenhang wird hier auch von der Registerbreite gesprochen. Diese beträgt für Doppelregister zwei.*

64 Bit breite Gleitkommazahlen werden in Doppelregistern gespeichert. Es können somit bis zu 16 Worte doppelter Genauigkeit und bis zu acht Worte vierfacher Genauigkeit in den Floating-Point-Registern gespeichert werden. Insbesondere ist auch eine gemischte Belegung der Register aus Worten einfacher, doppelter und vierfacher Genauigkeit erlaubt [8]. Aus diesem Grund ist die SPARC-Architektur zentral für diese Arbeit, da sie die Möglichkeit der Doppelregister bereitstellt. Die Codegenerierung für SPARC-Prozessoren von Code, der 64 Bit breite Gleitkommazahlen nutzt, musste bisher auf eine softwareseitige Emulation der Floating Point Unit zurückgreifen.

Die Registerallokation von Vierfachregistern für Worte vierfacher Genauigkeit wurde im Zusammenhang dieser Arbeit nicht implementiert. Aktuell sind keine SPARC-Prozessoren verfügbar, die diese Funktionalität in Hardware bereitstellen. In Abschnitt 5.2.3 wird auf diesen Punkt näher eingegangen.

Register	Registerinhalt	Ausrichtung an Registerindex	Registerindex
FD-0	s:exp[10:0]:fraction[51:32]	0 mod 2	r
FD-1	fraction[31:0]	1 mod 2	r+1
FQ-0	s:exp[14:0]:fraction[111:96]	0 mod 4	r
FQ-1	fraction[95:64]	1 mod 4	r+1
FQ-2	fraction[63:32]	2 mod 4	r+2
FQ-3	fraction[31:0]	3 mod 4	r+3

Tabelle 2.1.: Platzierung von Gleitkommazahlen doppelter und vierfacher Genauigkeit in der SPARC V8-Architektur [8].

Um ein Doppelregister zu verwenden, genügt es, der jeweiligen Instruktion eines der Floating-Point-Register zuzuteilen. Die Instruktion geht davon aus, dass in diesem angegebenen und dem unmittelbar darauffolgenden Register der Wert vorliegt. Dabei wird gefordert, dass ein Doppelregister immer aus einem Register mit geradem Index und dem direkt darauffolgenden Register besteht [8]. Tabelle 2.1 gibt für Doppelregister und Vierfachregister an, wie diese in den Floating-Point-Registern der SPARC-Architektur platziert werden dürfen.

2.4. Registerallokation

Einer der Schritte, die vom Back-End eines Compilers vorgenommen werden, ist die Registerallokation. Diese ist für die Zuteilung von Registern des Prozessors an einzelne Werte in der Zwischenrepräsentation verantwortlich. Da im Allgemeinen die Menge der Variablen und damit der Werte innerhalb eines Programms größer ist als die zur Verfügung stehenden Register des Prozessors, müssen einige Werte in den Hauptspeicher ausgelagert werden. Die Registerallokation versucht eine gültige Zuteilung von Registern an eine Teilmenge der Werte vorzunehmen. Da der Zugriff auf Werte im Hauptspeicher erheblich langsamer ist als der Registerzugriff, ist eine gute Zuteilung von besonderem Interesse.

Ein verbreiteter Ansatz um die Registerallokation durchzuführen ist die Übertragung auf das Problem der Graphfärbung. Diese Transformation wurde bereits 1981 von Chaitin gezeigt [9]. Bei der Graphfärbung wird jedem Knoten in einem Graph eine Farbe zugeteilt. Zwei adjazente Knoten dürfen dabei nicht dieselbe Farbe haben. Die minimale Anzahl Farben die zum korrekten Färben eines Graphen G benötigt wird, heißt chromatische Zahl $\chi(G)$. Kann ein Graph mit k Farben korrekt gefärbt werden, heißt er k-färbbar [10].

Setzt man bei der Registerallokation Graphfärbung ein, wird ein sogenannter Interferenzgraph genutzt. Jeder Wert im Programm entspricht einem Knoten in diesem ungerichteten Graphen. Können zwei Werte nicht das gleiche Register nutzen, wird eine Kante zwischen diesen im Interferenzgraphen eingefügt. Diese Interferenz tritt immer dann auf, wenn zwei Werte gleichzeitig lebendig sind. Ein Wert wird an einem Punkt im Programm als lebendig bezeichnet, wenn er bereits definiert wurde und zu einem späteren Zeitpunkt auf diesen Wert zugegriffen wird. Die chromatische Zahl des Interferenzgraphen gibt somit an, wie viele Register benötigt werden, um alle gleichzeitig lebendigen Werte abzuspeichern. Die Anzahl an benötigten Registern zu einem Zeitpunkt wird als *Registerdruck* bezeichnet. Ist dieser größer als die zur Verfügung stehenden Register, müssen einige Werte in den Hauptspeicher geschrieben werden. Dies wird als *Spilling* bezeichnet [9, 11].

Die Registerallokation mittels Graphfärbung kann im Allgemeinen in mehrere Phasen eingeteilt werden. Nach dem Aufbau des Interferenzgraphen wird eine *Coalescing*-Phase durchgeführt. Diese versucht unnötige Kopien von Registern zu entfernen und somit den Registerdruck zu senken. Daraufhin wird die eigentliche Graphfärbung vorgenommen, welche den Werten Register zuweist. In dieser Arbeit wird dieser Schritt als Registerzuteilung bezeichnet, wohingegen mit dem Begriff Registerallokation der gesamte mehrschrittige Prozess bezeichnet wird. Stellt sich bei der Färbung heraus, dass die Anzahl an Registern nicht ausreicht, der Graph also nicht k-färbbar ist mit k der Registeranzahl, muss die Spilling-Phase ausgeführt werden. Danach kann wieder mit dem Erstellen des Interferenzgraphen begonnen werden. Dieser Ablauf ist in Abbildung 2.2 schematisch dargestellt.

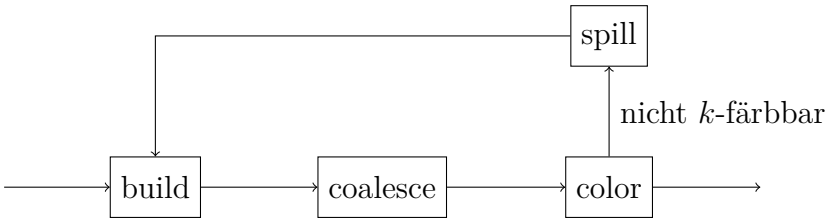


Abbildung 2.2.: Klassischer iterativer Ablauf von Registerallokation unter Anwendung von Graphfärbung [11].

Ist die Zwischenrepräsentation in SSA-Form, hat der Interferenzgraph nützliche Eigenschaften, welche die Registerallokation vereinfachen. Ein solcher Interferenzgraph ist chordal. Ein Graph G heißt chordal, wenn die Länge jedes Kreises in G maximal drei beträgt. Chordale Graphen besitzen ein sogenanntes *Perfektes Eliminationsschema (PES)*. Durch ein perfektes Eliminationsschema wird eine Reihenfolge vorgegeben, in der Knoten aus dem Graph entfernt werden müssen, sodass gilt: für jeden entfernten Knoten bilden die Nachbarn des Knoten eine Clique. Des Weiteren ist in chordalen Graphen die Größe der größten Clique gleich der chromatischen Zahl $\chi(G)$. Werte im Programm, die gleichzeitig lebendig sind, entsprechen Cliquen im Interferenzgraphen. Die Anzahl an zur Verfügung stehenden Registern n ist bekannt und durch Spilling kann sichergestellt werden, dass an jedem Punkt im Programm maximal n Werte gleichzeitig lebendig sind. Ist dies gegeben, ist die chromatische Zahl des Interferenzgraphen G beschränkt: $\chi(G) \leq n$. Der Graph ist somit n-färbbar. Nach dem Färben ist daher kein weiteres Spilling nötig und es genügt, das Spilling einmalig durchzuführen [11]. Coalescing zur Entfernung überflüssiger Kopien kann durch Neufärbung des Graphen so ausgeführt werden, dass die chordale Eigenschaft des Graphen erhalten bleibt und kein weiteres Spilling vonnöten ist [12]. Die Registerallokation lässt sich daher zu dem in Abbildung 2.3 gezeigten Ablauf vereinfachen [11].

Abbildung 2.3.: Ablauf von Registerallokation unter Anwendung von Graphfärbung für Zwischenrepräsentationen in SSA-Form. Der Interferenzgraph ist hierbei chordal [11].

2.5. Verwandte Arbeiten

Die Arbeit von Sebastian Hack, Daniel Grund und Gerhard Goos [11] zeigt die hier beschriebenen Eigenschaften für Interferenzgraphen von Zwischenrepräsentationen in SSA-Form und führt den in Abbildung 2.3 gezeigten Ablauf der Registerallokation ein.

Der Schritt der Kopienminimierung, der einen Teil der Coalescing-Phase bildet, wird in [12] von Sebastian Hack und Gerhard Goos eingeführt und die Algorithmen, die dazu in libFirm zum Tragen kommen, vorgestellt.

Andere Autoren untersuchten bereits die Registerallokation für Architekturen, die besondere Strukturen wie Doppelregister bereitstellen. So stellen Briggs, Cooper und Torczon den Ansatz vor, mehrere Kanten im Interferenzgraphen einzufügen, um Doppelregister und darüber hinausgehende Bedingungen zu modellieren [13]. Auch Brian Nickerson behandelt Anpassungen an die Interferenz zwischen Variablen um Bedingungen wie Doppelregister im Interferenzgraph modellieren zu können [14]. Ein weiterer Ansatz besteht darin, die Knoten im Interferenzgraphen je nach Breite des benötigten Registers unterschiedlich zu gewichten, wie von Michael Smith und Glenn Holloway untersucht wurde [15].

Allen diesen Ansätzen ist gemein, dass sie auf dem nicht-SSA-basierten, iterativen Modell wie in Abbildung 2.2 gezeigt aufsetzen. Die im letzten Abschnitt genannten Vorteile der SSA-basierten Registerallokation entfallen somit, insbesondere die Möglichkeit, das Spilling einmalig durchzuführen und anschließend die Färbung vorzunehmen. Dies stellt für die vorliegende Arbeit eine Grundvoraussetzung dar.

3. Entwurf und Implementierung

In diesem Kapitel werden anhand der bestehenden Implementierung in libFirm die Anpassungen erklärt, die eine Verwendung von Doppelregistern in der Registerallokation ermöglichen. Zunächst wird auf die Repräsentation von Doppelregistern eingegangen, anschließend werden die Implementierungen in den einzelnen Schritten der Registerallokation beschrieben. Sowohl das Spilling als auch die Registerzuteilung erfordern Anpassungen für Doppelregister. Der an die Registerzuteilung anschließende Schritt der Kopienminimierung wurde in dieser Arbeit nicht bearbeitet. Da die Kopienminimierung in libFirm optional ist und eine Optimierung darstellt, liefert die Registerallokation auch ohne diesen Schritt korrekte Ergebnisse. In Abschnitt 5.2.2 wird dieser Umstand näher betrachtet. Zuletzt wird darauf eingegangen, welche Anpassungen an die Auflösung von Permutationsknoten nötig waren. Diese Auflösung ist nötig, da Permutationsknoten keine Instruktionen der Prozessorarchitektur darstellen.

3.1. Ausgangssituation in libFIRM

Das libFirm-Back-End verfügt bereits über eine funktionsfähige Registerallokation. Da libFirm wie bereits erwähnt eine Zwischenrepräsentation in SSA-Form nutzt, folgt die Registerallokation dem in Abschnitt 2.4 beschriebenen Ablauf.

Auch enthält libFirm ein Back-End für die SPARC-Architektur. Die Codegenerierung für SPARC ist darin vollständig implementiert. Das schließt auch die Funktionalität der Doppelregister mit ein: SPARC-Instruktionen, die auf Gleitkommazahlen doppelter Genauigkeit operieren, beispielsweise `faddd` zur Addition zweier 64-Bit-Gleitkommazahlen, können vom Back-End in Maschinensprache ausgegeben werden.

Die Funktionalität der Registerallokation und die Implementierungen der zugehörigen Algorithmen sind bisher jedoch nicht fähig, mit Doppelregistern umzugehen. Im Folgenden wird erläutert, welche Anpassungen und Erweiterungen implementiert werden mussten, um die Registerallokation auch für Doppelregister zu ermöglichen.

3.2. Repräsentation von Doppelregistern

In libFirm existieren verschiedene Datentypen, um Register und spezielle Anforderungen an Register einer Operation abzubilden.

Der Datentyp `struct arch_register_t` repräsentiert ein einzelnes Register der Prozessorarchitektur. In diesem Datentyp ist unter anderem die Registerklasse und der Index des Registers innerhalb der Registerklasse gespeichert. Neben den Registern gibt es auch Registeranforderungen. Diese werden durch den Datentyp `struct arch_register_req_t` modelliert. Listing 3.1 zeigt die Definition dieses Datentyps. Eine Registeranforderung umfasst ebenfalls die Registerklasse und kann beispielsweise eine Einschränkung auf bestimmte Register beschreiben. Dies ist zum Beispiel für Prozessorinstruktionen nötig, welche die Operanden immer in bestimmten Registern erwarten oder das Ergebnis der Operation immer in ein festgelegtes Register schreiben. In der Registeranforderung wird auch die benötigte Breite des Registers gespeichert. Für Doppelregister muss dieser Wert auf zwei gesetzt werden. Die Codegenerierung des SPARC-Back-Ends unterscheidet anhand dieses Parameters, ob Gleitkommazahlen einfacher oder doppelter Genauigkeit verwendet werden sollen und wählt auf diese Weise die korrekten Prozessorinstruktionen aus. Eine Instanz des Knoten-Datentyps (`struct ir_node`) für Knoten im Firm-Graphen enthält Registeranforderungen aller Werte, die aus den Registern gelesen oder in Register geschrieben werden. Für Werte, die von einem Knoten definiert und demzufolge in Register geschrieben werden, werden zusätzlich die zugewiesenen Register im Knoten gespeichert.

Im Normalfall wird bereits bei der Erstellung des Knotens die Registeranforderung gesetzt, die Information über die benötigte Registerbreite ist also vor der Registerallokation im Knoten gegeben. Es bietet sich daher an, diese bestehende Repräsentation auch bei der Implementierung der Registerallokation für Doppelregister einzusetzen. Die Registerallokation von libFirm wurde deshalb dahingehend angepasst, anhand dieser bestehenden Repräsentation Doppelregister zu identifizieren. Einem Wert wird dazu nach wie vor eine einzelne Instanz des Register-Datentyps zugeteilt. Durch die in der Registeranforderung angegebene Breite wird für Doppelregister implizit auch die zweite Hälfte an den Wert zugewiesen.

Listing 3.1: Der Datentyp für Registeranforderungen im libFirm-Quellcode [16]. Über den Parameter `width` kann die Breite des erforderlichen Registers gesetzt werden.

```
typedef struct arch_register_req_t arch_register_req_t;
struct arch_register_req_t {
    /** The register class this constraint belongs to. */
    const arch_register_class_t *cls;
    /** allowed register bitset
     * (in case of wide-values this is only about the first
```

```
     * register). NULL if all registers are allowed. */
    const unsigned              *limited;
    /** Bitmask of ins which should use the same register. */
    unsigned                    should_be_same;
    /** Bitmask of ins which shall use a different register */
    unsigned                    must_be_different;
    /** Specifies how many sequential registers are required */
    unsigned char               width;
    /** ignore this input/output while allocating registers */
    bool                        ignore : 1;
    /** The instructions modifies the value in the register in
     * an unknown way, the value has to be copied if
     * it is needed afterwards. */
    bool                        kills_value : 1;
};
```

Eine Prozessorarchitektur verfügt meist über mehrere Registerklassen. Bei der SPARC-Architektur sind dies hauptsächlich die Klasse der General-Purpose-Register und die Klasse der Floating-Point-Register. Um zu bestimmen ob eine Registerklasse Doppelregister unterstützt wurde der Datentyp `arch_register_class_t`, der im libFirm-Back-End eine Registerklasse repräsentiert, erweitert. Der Datentyp wurde dazu um das Feld `bool double_registers_allowed` ergänzt. Dieses Feld kann dann in der Definition der jeweiligen Zielarchitektur gesetzt werden.

3.3. Spilling

Der erste Schritt des in libFirm implementierten SSA-basierten Registerallokators ist das Spilling. Ziel dieser Phase ist es, wie in Abschnitt 2.4 kurz beschrieben, dafür zu sorgen, dass zu jedem Zeitpunkt in einem Programm die Menge der gleichzeitig lebendigen Werte maximal der in einer Registerklasse vorhandenen Registeranzahl entspricht. Dies wird durch Einfügen von *spills* und *reloads* erreicht, wodurch Werte in den Speicher geschrieben werden und zu einem späteren Zeitpunkt wieder aus dem Speicher geladen werden. Dadurch kann der Registerdruck verkleinert werden, da weniger Werte gleichzeitig lebendig sind.

Die Implementierung in libFirm nutzt hierfür den Algorithmus von Bélády. Sind mehr Werte lebendig als Register vorhanden sind, muss entschieden werden, welche der Werte in den Speicher ausgelagert werden. Béládys Algorithmus wählt hierzu die Werte, deren nächste Benutzung am weitesten von der aktuellen Position entfernt ist. Die Entfernung wird von der Anzahl Instruktionen zwischen den beiden Punkten im Programm bestimmt.

3.3.1. Workset

Der Datentyp **struct** workset_t bildet eine zentrale Datenstruktur im Spilling-Algorithmus. Listing 3.2 enthält die Definition des Datentyps. Innerhalb eines Worksets wird eine Liste von Knoten mit dem Zeitpunkt der nächsten Benutzung des von ihnen definierten Wertes mithilfe des Datentyps **struct** loc_t gespeichert. Diese Liste repräsentiert die Menge der zu einem Zeitpunkt lebendigen Werte.

Listing 3.2: Für Worksets genutzte Datentypen [16].

```
typedef struct loc_t {
    ir_node  *node;
    /** A use time (see beuses.h). */
    unsigned  time;
    /** value was already spilled on this path */
    bool      spilled;
} loc_t;

typedef struct workset_t {
    /** current length */
    unsigned len;
    /** array of the values/distances in this working set */
    loc_t    vals[];
} workset_t;
```

Um gemäß dem Algorithmus von Bélády die Werte, deren Distanz zur nächsten Benutzung am größten ist, einfach zu erhalten, können die Knoten in einem Workset mittels workset_sort(workset) nach dieser Distanz sortiert werden.

3.3.2. Ablauf

Der Spilling-Algorithmus in libFirm geht blockweise vor. Für jeden Block im Graph wird dazu die Funktion process_block aufgerufen, die das Spilling für diesen Block durchführt. Dabei erfolgt die Traversierung des Kontrollflussgraphen in reverse postorder. Wenn ein Block besucht wird, ist dadurch sichergestellt, dass die Blöcke, welche im Kontrollfluss vor dem aktuell betrachteten Block liegen, bereits besucht wurden. Zu einem Block werden darüber hinaus je ein Workset für den Anfang des Blocks (Start-Workset) und das Ende des Blocks (Ende-Workset) abgespeichert. Hat ein Block keine Vorgänger, wird als Start-Workset ein leeres Workset gewählt. Hat ein Block exakt einen Vorgängerblock, wird dessen Ende-Workset als Start-Workset des aktuellen Blocks gesetzt. Wenn ein Block mehrere Vorgänger besitzt, wird versucht, aus den Ende-Worksets der Vorgängerblöcke gemäß Bélády die besten Werte in das

Start-Workset zu übernehmen. Für Schleifen wird zusätzlich bestimmt, welche Werte im Schleifenrumpf benötigt werden. Wird ein Wert innerhalb einer Schleife aus dem Hauptspeicher geladen, muss dies im schlechtesten Falle in jedem Durchlauf erfolgen. Dies verschlechtert die Performanz des Programms erheblich, daher wird an dieser Stelle versucht, innerhalb der Schleife möglichst wenige Werte zu laden.

Nachdem das Start-Workset festgelegt wurde, wird anhand der bereits vor dem Spilling festgelegten Schedule-Reihenfolge über die Knoten im Block iteriert. Jeder Zugriff auf einen Wert (use) durch den Knoten und jeder Wert, welchen der Knoten definiert, wird gespeichert. Zum Ausführungszeitpunkt der Instruktion, die durch diesen Knoten repräsentiert wird, müssen die Operanden der Instruktion in Registern vorliegen. Werte, die von der Instruktion definiert werden, dürfen keine anderen, noch lebendigen Werte in den Registern überschreiben.

Das Spilling stellt zum einen sicher, dass Werte, die von einer Instruktion gelesen werden, vor der Ausführung der Instruktion in den Registern vorliegen. Ist dies nicht der Fall und Werte wurden zuvor ausgelagert, müssen entsprechende reload-Knoten eingefügt werden, wodurch die Werte aus dem Speicher wieder in die Register geschrieben werden. Da im Allgemeinen nicht genug Register frei sind, um die geladenen Werte aufzunehmen, müssen gegebenenfalls andere Werte zuvor ausgelagert werden. Schreibt die Instruktion Werte in Register und es sind nicht ausreichend Register unbelegt, müssen zudem weitere Werte vor Ausführung der Instruktion ausgelagert werden. Dadurch wird verhindert, dass Werte ungewollt überschrieben werden, die lebendig sind, das heißt zu einem späteren Zeitpunkt noch benötigt werden.

Nachdem das Spilling innerhalb eines Blocks beendet wurde, wird das letzte Workset als Ende-Workset des Blocks abgespeichert, um für im Kontrollfluss nachfolgende Blöcke bei der Erstellung des Start-Worksets zur Verfügung zu stehen.

Nachdem alle Blöcke auf diese Weise abgearbeitet wurden, müssen in einem letzten Schritt noch die Blockgrenzen zwischen Blöcken angepasst werden, da durch die blockweise Durchführung des Spilling nicht immer das Start- und Ende-Workset korrekt zueinander passen und weitere Auslagerungen oder reload-Vorgänge eingefügt werden müssen.

3.3.3. Anpassung für Doppelregister

Der beschriebene Algorithmus muss angepasst werden, um Doppelregisterallokation korrekt zu implementieren. Zentrale Anpassung hierfür ist die Entkopplung der Anzahl Knoten in einem Workset und der von diesen Knoten benötigten Registeranzahl. In der bisherigen Implementierung wurde die Anzahl der Knoten in einem Workset mit

der durch die Registerklasse zur Verfügung gestellten Registeranzahl verglichen, um die Anzahl an benötigten Auslagerungen zu bestimmen. Von dieser bereits definierten Länge eines Worksets wird nun die *genutzte Länge* unterschieden, welche sich durch Summieren aller benötigten Registerbreiten der im Workset enthaltenen Knoten ergibt. Erlaubt eine Registerklasse sowohl Einzel- als auch Doppelregister, entspricht nicht die Anzahl der gleichzeitig lebendigen Werte, sondern die genutzte Länge eines Worksets dem Registerdruck. Für Registerklassen, die nur Einzelregister unterstützen, entspricht die genutzte Länge immer der Länge des Worksets.

Wenn Operanden einer Instruktion ausgelagert wurden und vor der Instruktion wieder in die Register geladen werden, müssen, sofern keine Register unbelegt sind, an deren Stelle andere Werte ausgelagert werden. Werden alle Register als Einzelregister genutzt, bedingt das Laden eines Wertes aus dem Speicher somit höchstens das Auslagern eines anderen Wertes in den Speicher. Wenn eine Registerklasse Register mit einer maximalen Breite von w erlaubt, können durch das Laden eines Wertes, der eine Registerbreite von w benötigt, bis zu w Auslagerungen anderer Werte nötig sein. Die tatsächliche Anzahl hängt im Einzelnen davon ab, welche Registerbreiten diejenigen Werte benötigen, die zuerst ausgelagert werden. Nach dem Algorithmus von Bélády sind dies die Werte, deren nächste Benutzung am weitesten entfernt liegt. Um Werte in Doppelregister ($w = 2$) zu laden müssen somit bis zu zwei Werte ausgelagert werden.

Beispiel 1. *Ein Workset liege sortiert vor, das heißt die Knoten, deren Werte die größte Distanz zur nächsten Benutzung besitzen, befinden sich am Ende der Liste. Alle aktuell lebendigen Werte, also genau diejenigen, welche im Workset enthalten sind, belegen Einzelregister. Das Workset enthalte außerdem die maximal mögliche Anzahl Knoten, alle Register sind somit belegt. Die aktuell betrachtete Instruktion liest zwei Werte aus je einem Doppelregister. Beide Werte sind aktuell nicht lebendig und müssen daher aus dem Speicher geladen werden.*

Da keine Register frei sind, müssen andere Werte ausgelagert werden. Dazu werden aus dem Workset solange Knoten entfernt, bis der Platz ausreicht, um die zu ladenden Werte aufzunehmen. Im diesem Beispiel müssen also vier Werte ausgelagert werden, bevor die zwei angeforderten Werte in die Register geschrieben werden können.

Die Anpassungen am Quelltext des Spilling unterstützen generell Register, die aus mehreren einzelnen Registern bestehen. Die Register sind nicht auf Einzel- und Doppelregister beschränkt, das Spilling funktioniert für beliebige Registerbreiten. Durch das Spilling wird lediglich sichergestellt, dass zu jedem Zeitpunkt alle momentan lebendigen Werte in den Registern Platz finden. Die zusätzliche Bedingung für Doppelregister, das Ausrichten an geraden Indizes, wird hier nicht beachtet sondern muss von der nachfolgenden Registerzuteilung vorgenommen werden. In Abschnitt 3.4.1 wird auf diesen Umstand näher eingegangen.

3.4. Registerzuteilung

Das vorangegangene Spilling hat an den erforderlichen Stellen einen Teil der Werte ausgelagert. Dadurch wurde sichergestellt, dass an jedem Punkt im Programm alle zu diesem Zeitpunkt lebendigen Werte in die Register passen. Im nächsten Schritt können nun die Register an die Knoten zugeteilt werden. Dabei wird der Graph der Zwischenrepräsentation in mehreren Iterationen traversiert und die nachfolgend beschriebenen Schritte jeweils für jeden Block durchgeführt.

3.4.1. Registereinschränkungen verarbeiten

Zunächst müssen für Knoten, die Einschränkungen an die Register stellen, Vorarbeiten durchgeführt werden. Solche Einschränkungen sind beispielsweise ein festes Ausgaberegister einer Instruktion. Im betrachteten Block wird entsprechend der Schedule-Reihenfolge für jeden Knoten geprüft, ob dieser Einschränkungen an die Register stellt. Ist dies nicht der Fall, kann der nächste Knoten betrachtet werden.

Andernfalls wird vor dem betrachteten Knoten ein sogenannter Permutationsknoten eingefügt. Durch den Permutationsknoten können die Register aller zu diesem Zeitpunkt lebendigen Werte getauscht werden. Eine Instruktion, die Einschränkungen auf bestimmte Register trifft, kann die benötigten Register somit auf jeden Fall nutzen. Sollten beispielsweise Register, in die eine solche Instruktion schreibt, bereits durch andere Werte belegt sein, können mithilfe des Permutationsknotens die bereits enthaltenen Werte in andere Register verschoben werden.

Neben Instruktionen, die bestimmte Ein- oder Ausgaberegister erwarten, wird jede Instruktion, die in Doppelregister schreibt, als Instruktion mit Einschränkungen angesehen. Dies ist nötig, da ab einem bestimmten Registerdruck Situationen auftreten können, die eine Permutation erforderlich machen. Sobald der Registerdruck mehr als die Hälfte der insgesamt zur Verfügung stehenden Register beträgt, besteht die Möglichkeit, dass für eine Instruktion, die in ein Doppelregister schreibt, kein solches mehr zur Verfügung steht. Das vorangegangene Spilling hat lediglich dafür gesorgt, dass die Anzahl an unbelegten Registern ausreicht, um den zu schreibenden Wert unterzubringen. Die unbelegten Register können jedoch so angeordnet sein, dass keine gültige Position für ein Doppelregister vorhanden ist.

Beispiel 2. *Betrachtet wird eine Instruktion, die einen Wert in ein Doppelregister schreibt. In der zugehörigen Registerklasse seien n Einzelregister enthalten. Während der Instruktion seien exakt $\frac{n}{2}$ weitere Werte einfacher Genauigkeit lebendig, die bereits Register zugeteilt bekommen haben. Dabei sei jedem Wert ein Register mit geradem Index zugeordnet. Für den neuen von der Instruktion definierten Wert kann kein*

Register gefunden werden, da nur Register mit ungeradem Index unbelegt sind, wie Abbildung 3.1 verdeutlicht.

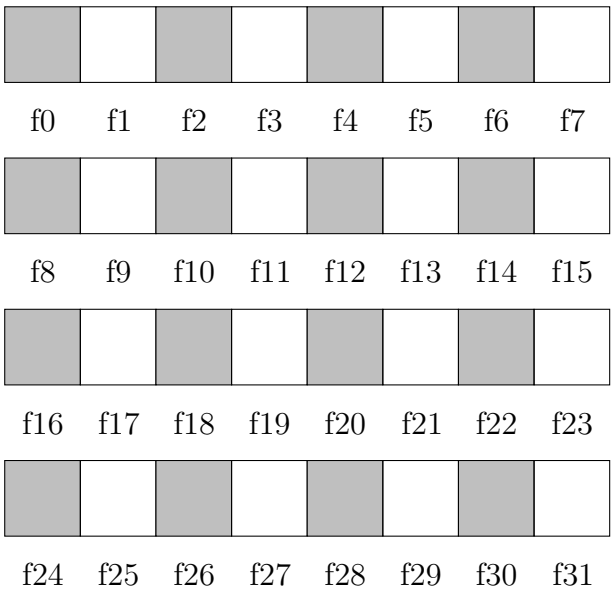


Abbildung 3.1.: Situation ohne Permutationsknoten. Die zum Zeitpunkt der Instruktion belegten Register sind grau hinterlegt. Für eine Instruktion, die einen weiteren Wert in einem Doppelregister definiert, ist hier kein passendes Register verfügbar, obwohl der Registerdruck nur die Hälfte der Registeranzahl beträgt.

Durch Einfügen eines Permutationsknotens besitzen die während der Instruktion lebendigen Werte zunächst wieder kein zugeteiltes Register. Abbildung 3.2 zeigt für diese Werte zusammen mit dem von der Instruktion definierten Wert eine neu bestimmte Registerzuteilung.

Nun können auch den zu schreibenden Werten der Instruktion Register zugeteilt werden. Alle anderen während der Instruktion lebendigen Werte werden durch den Permutationsknoten definiert. Auch diesen Werten wird jeweils ein Register zugewiesen. Für die Zuteilung der Register wird an dieser Stelle ein bipartites Matching verwendet. Als bipartites Matching wird eine Menge von Kanten eines ungerichteten Graphen, dessen Knoten in zwei Partitionen eingeteilt sind, bezeichnet. Dabei besitzen keine zwei Kanten des Matchings einen gemeinsamen Endknoten [10]. Eine der beiden Partitionen wird durch die erwähnten Werte gebildet, denen Register zugeteilt werden sollen. Die einzelnen Register der Registerklasse stellen die zweite Partition dar. Eine Kante zwischen zwei Knoten wird genau dann in den Graphen eingefügt, wenn der entsprechende Wert in dem entsprechenden Register stehen

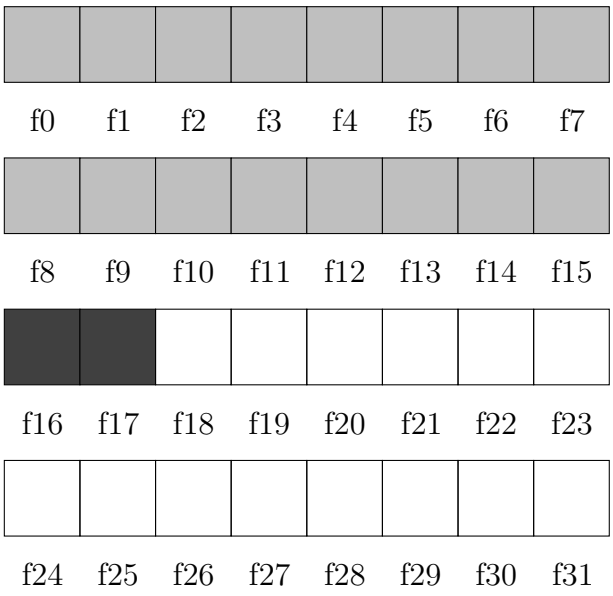


Abbildung 3.2.: Nach Einfügen eines Permutationsknotens vor der betroffenen Instruktion ist der Registerdruck zum Ausführungszeitpunkt der Instruktion weiterhin gleich, durch die Neuzuteilung der ersten 16 Register an die Werte kann aber für den neuen Wert ein Doppelregister (hier *f16*) gefunden werden.

darf. Schreibt die Instruktion den Wert immer in ein festgelegtes Register, wird dementsprechend nur eine Kante zwischen diesem Wert und den Registern eingefügt. Werte, die in beliebigen Registern stehen dürfen, beispielsweise die Ausgaben des Permutationsknotens, sind Endknoten mehrerer Kanten zu Registern. Das gefundene Matching ist dann eine korrekte Registerzuteilung an die Werte.

Benötigt ein Wert ein Doppelregister, dürfen nur zu Registern mit geradem Index Kanten im Graph eingefügt werden. Dies missachtet aber die Eigenschaft des Doppelregisters als Paar von aufeinanderfolgenden Einzelregistern. Ein gültiges Matching kann so gewählt sein, dass ein eigentlich als obere Hälfte eines Doppelregisters gewähltes Register mit ungeradem Index einem anderen Wert zugeteilt wird, wie in Abbildung 3.3 gezeigt.

Für Registerklassen, die Doppelregister erlauben, wird deshalb kein bipartites Matching genutzt. Stattdessen werden den Werten die Register zugeteilt indem das erste unbelegte Register per linearer Suche über die Liste der Register gesucht wird. Benötigt ein Wert ein Doppelregister, wird entsprechend das erste unbelegte Register mit geradem Index gesucht, dessen unmittelbar darauffolgendes Register ebenfalls unbelegt ist. Da auf diese Weise die Register von vorn gefüllt werden und allen lebendigen Werten durch den Permutationsknoten ebenfalls an dieser Stelle Register

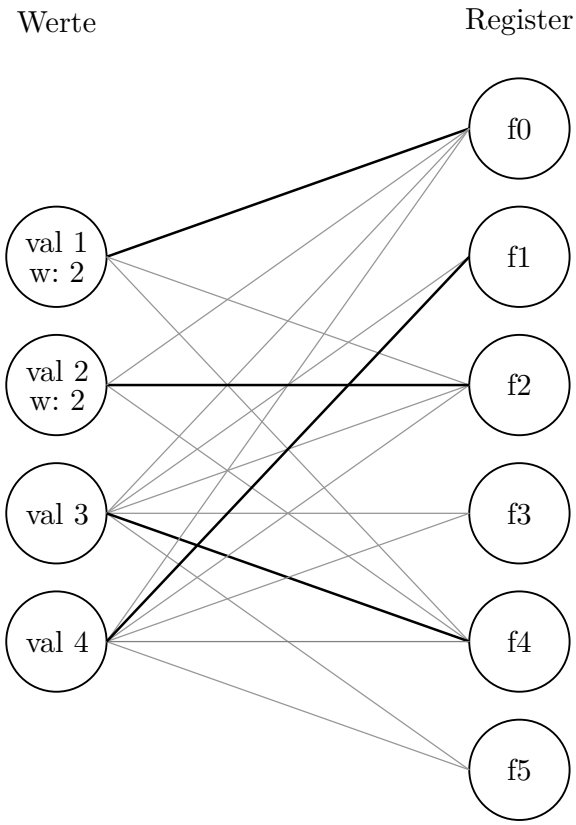


Abbildung 3.3.: Den vier Werten sollen Register zugeordnet werden. Die Werte eins und zwei benötigen Doppelregister. Die dicker hervorgehobenen Kanten bilden ein gültiges Matching. Register *f1* wird jedoch implizit sowohl Wert eins als auch Wert vier zugeteilt, da Wert eins als Doppelregister *f0* und *f1* erhält. Das Matching bildet daher keine gültige Registerzuteilung.

zugewiesen werden, wird immer eine Möglichkeit für ein Doppelregister gefunden.

Die Klasse der Floating-Point-Register der SPARC-Architektur ist die einzige Registerklasse in libFirm, die Doppelregister in der hier behandelten Form unterstützt. Die Instruktionen der SPARC-Architektur, die in Floating-Point-Register schreiben, stellen in den meisten Fällen keine besonderen Anforderungen an die Register. Wenige Ausnahmen bilden einzelne Instruktionen, die Werte auf bestimmte Register einschränken. Da diesen vor den anderen durch den Permutationsknoten lebendigen Werten Register zugewiesen werden ist die Zuteilung auf diese Weise durchführbar.

3.4.2. Schreib- und Lesezugriffe speichern

Vor dem eigentlichen Zuteilen von Registern an diejenigen Werte, die nicht bereits durch den vorherigen Schritt abgedeckt wurden, werden zunächst Informationen über die Definitionen und Benutzungen der Werte innerhalb eines Blocks erstellt.

Nach dem Betrachten der am Ende des Blocks lebendigen Werte wird die Schedule-Reihenfolge im Block rückwärts abgelaufen. In einer doppelt verketteten Liste wird jeder Schreib- und Lesezugriff auf einen Wert als Instanz des Datentyps **struct** border_t gespeichert. Zuletzt werden noch die zu Beginn des Blocks lebendigen Werte hinzugefügt. Aufgrund der SSA-Form existiert für jeden Wert genau eine Definition und somit genau ein Schreibzugriff. Für Werte, die über die Blockgrenze hinweg in den Block hinein lebendig sind, wird ebenfalls eine Definition in die Liste eingefügt.

3.4.3. Färben

Im abschließenden Schritt der Registerzuteilung können nun Register an die verbleibenden Werte zugeteilt werden, was dem Färben der einzelnen Knoten im Interferenzgraphen entspricht. Dies wird blockweise durchgeführt, wobei die im vorigen Schritt für jeden Block bestimmte Liste an Definitionen und Benutzungen der Werte genutzt wird.

Indirekt enthält diese Liste die Intervalle in denen Werte lebendig sind. Diese werden durch die Positionen der Definitionen und Benutzungen gebildet. Die Liste gibt gleichzeitig eine Reihenfolge vor, in welcher den Werten die Register zugeteilt werden können. Die Liste wird dazu in umgekehrter Reihenfolge als bei der Erstellung durchlaufen. Somit werden die Definitionen und Benutzungen der Werte von vorne nach hinten abgelaufen. Dies entspricht der umgekehrten Reihenfolge eines perfekten Eliminationsschemas des Interferenzgraphen. Somit wird eine korrekte Färbung erzeugt [11].

Für jede Definition, die betrachtet wird, muss dem Wert ein Register zugeteilt werden. In einigen Fällen wurde dem Wert bereits ein Register zugeteilt. Dies ist zum einen dann der Fall, wenn die eigentliche Definition des Wertes in einem Vorgängerblock stattfand und der Wert über die Blockgrenze hinweg in den aktuell betrachteten Block hinein lebendig ist. Des Weiteren wurden für Instruktionen, die Registereinschränkungen haben, bereits die Register zugeteilt (siehe Abschnitt 3.4.1). Insbesondere wurden dabei auch solche Instruktionen eingeschlossen, die Werte in Doppelregistern definieren. Dennoch erfordert auch dieser Schritt der Registerzuteilung eine Anpassung an Doppelregister, da nicht alle Knoten aus der Zwischenrepräsentation abgedeckt wurden. Knoten, welche die Anwendung einer ϕ-Funktion darstellen,

wurden bei der Verarbeitung der Registereinschränkungen nicht mit einbezogen. Diesen müssen hier noch Register zugeteilt werden. Unabhängig davon, ob bereits ein Register in einem früheren Schritt zugeteilt wurde oder dies erst jetzt stattfindet, wird das entsprechende Register als belegt markiert. Für Doppelregister werden beide Einzelregisterbestandteile markiert.

Jede Benutzung eines Wertes in der Liste entspricht dem Ende des Intervalls, in welchem dieser Wert lebendig ist. Da anschließend der Wert das Register nicht mehr belegt, wird es an dieser Stelle wieder freigegeben, indem es als verfügbar markiert wird. Analog werden auch hier beide Einzelregister eines Doppelregisters als verfügbar markiert. Werte, die zu einem späteren Zeitpunkt im Block definiert werden, können die Register dann wieder nutzen.

Bei der Zuteilung eines Registers an einen Wert muss ein passendes unbelegtes Register gesucht werden. Hierfür wird eine einfache lineare Suche über die Register durchgeführt, bis ein unbelegtes Register gefunden wird. Dabei beginnt die Suche immer am Register mit Index Null. Während für Einzelregister immer das erste freie Register genutzt werden kann, muss für Doppelregister zum einen beachtet werden, dass das nachfolgende Register, welches den hinteren Teil des Registerpaares bildet, ebenfalls unbelegt ist. Außerdem darf für Doppelregister nur ein Register mit geradem Index gewählt werden. Diese Suche wird von der Funktion **int** get_next_free_reg übernommen, die als Parameter die Liste der Register sowie die Breite eines angeforderten Registers erhält.

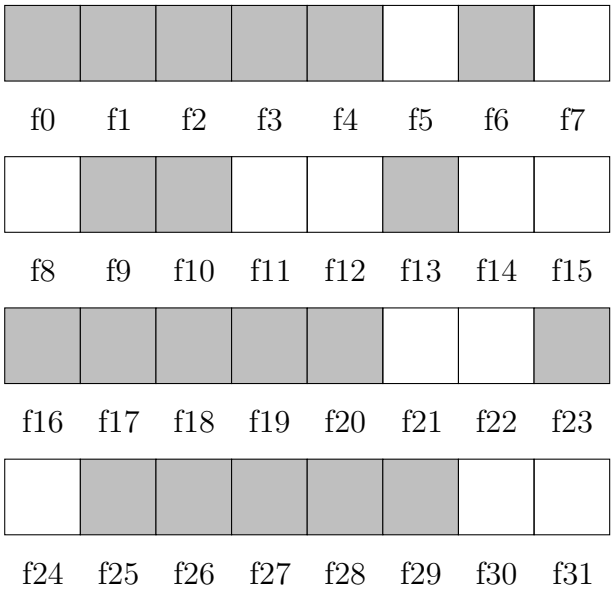


Abbildung 3.4.: Ausgangssituation für Beispiel 3. Von lebendigen Werten belegte Register sind grau eingefärbt.

Beispiel 3. *Ein Teil der verfügbaren Register der 32 Register umfassenden Register-*

klasse sei wie in Abbildung 3.4 belegt. Die Suche nach einem freien Einzelregister liefert die Position fünf zurück, da dies das erste unbelegte Register ist.

Wird ein Doppelregister benötigt, muss ein geeignetes Paar an Einzelregistern gefunden werden und der Index des unteren Registers zurückgegeben werden. Im in Abbildung 3.4 dargestellten Zustand ist die erste Position an der zwei aufeinanderfolgende Register unbelegt sind, Position elf. Da aber für das Doppelregister ein gerader Index benötigt wird, ist das Ergebnis der Suche Position 14. Anschließend stellt sich die Registerbelegung wie in Abbildung 3.5 dar.

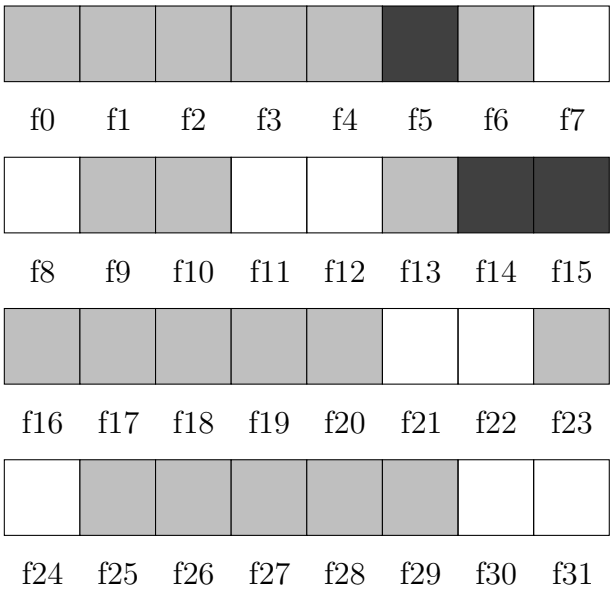


Abbildung 3.5.: Die belegten Register, nachdem zunächst ein Einzel- und anschließend ein Doppelregister zugewiesen wurden.

Im Normalfall ist die Registerbelegung allerdings nicht so verteilt, wie Abbildung 3.4 vermuten lässt. Durch das Einfügen der Permutationsknoten zuvor für Instruktionen, die Werte in Doppelregistern definieren, werden allen zu diesem Zeitpunkt lebendigen Werten aus der gleichen Registerklasse die Register neu zugewiesen. Dabei werden diese von vorne der Reihe nach ausgewählt und es kommen nicht mehrere Lücken zustande, sondern die Belegung stellt sich wie in Abbildung 3.2 dar. Somit kann auch in diesem Schritt eine Position für ein Doppelregister gefunden werden, wenn der Registerdruck hoch ist.

3.5. Auflösung von Permutationsknoten

3.5.1. Permutationsknoten

Permutationsknoten sind ein Konstrukt in der graphbasierten Zwischenrepräsentation, um die Register einer Menge von Werten neu zuzuordnen. Sie werden unter anderem im in Abschnitt 3.4.1 geschilderten Verarbeiten von Registereinschränkungen eingefügt. Da ein Permutationsknoten keiner existierenden Instruktion einer Prozessorarchitektur entspricht, muss dieser in eine Folge von Kopien umgewandelt werden. Mit welchen Prozessorinstruktionen die Registerkopien realisiert werden, hängt von der Zielarchitektur ab.

Abbildung 3.6 zeigt ein Beispiel eines Permutationsknotens.

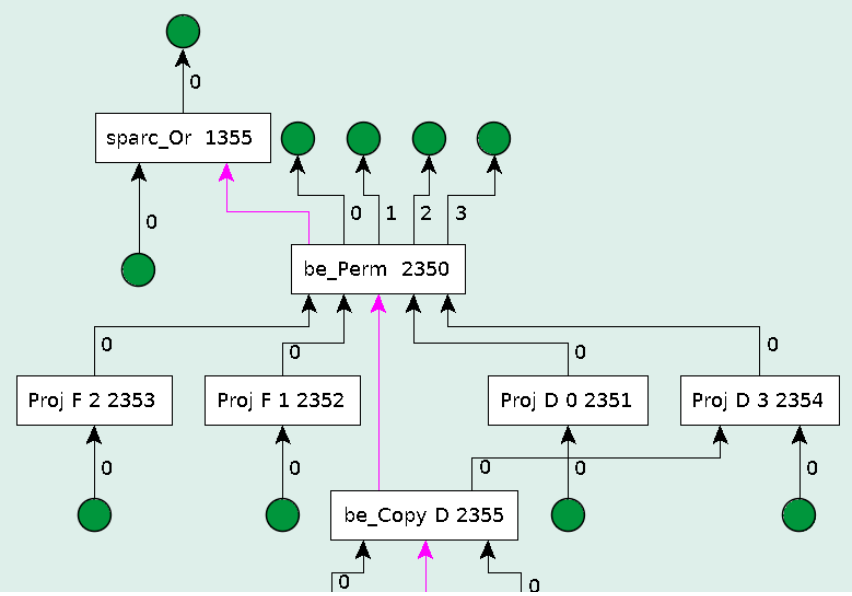


Abbildung 3.6.: Ein Permutationsknoten, welcher die Register für vier Werte tauscht. Die Proj-Knoten dienen dazu, auf einen einzelnen Wert aus dem Ergebnistupel eines Knotens zuzugreifen. Die Buchstaben F und D hinter dem Knotentyp geben an, ob es sich um einen Wert einfacher (*float*) oder doppelter Genauigkeit (*double*) handelt.

3.5.2. Umwandlung in Kopien

Um die Permutationsknoten zu entfernen wird der Graph der Zwischenrepräsentation einmal komplett abgelaufen und die Transformation für jeden Permutationsknoten durchgeführt. Im Folgenden wird zunächst die Auflösung für Permutationsknoten ohne Berücksichtigung von Doppelregistern erklärt. Anschließend wird begründet, warum für Doppelregister eine Anpassung nötig ist und diese vorgestellt.

Zu Beginn werden die Ein- und Ausgaberegister des Permutationsknotens betrachtet und daraus eine Liste der Registerpaare erstellt. Ein Registerpaar bezeichnet in diesem Kontext nicht ein Doppelregister, sondern ein Paar aus dem Ein- und dem Ausgaberegister für einen Wert am Permutationsknoten. Da bei der Erstellung der Permutationsknoten nicht darauf geachtet wurde, welche Register als Ausgaberegister gewählt werden, besteht ein Teil dieser Registerpaare unter Umständen aus demselben Ein- und Ausgaberegister. Solche Paare erfordern keine Kopie des Wertes und werden daher im weiteren Verlauf der Transformation nicht mehr beachtet. Für den speziellen Fall, dass dies für alle Registerpaare eines Permutationsknotens zutrifft, wird der gesamte Permutationsknoten aus dem Graphen entfernt. Um im nächsten Schritt die Paare in der richtigen Reihenfolge bearbeiten zu können, werden außerdem alle Eingaberegister in einer Liste abgelegt und eine Zuordnung von Ausgaberegister zu dem jeweiligen Registerpaar mit diesem Ausgaberegister abgespeichert.

Um die Permutationsknoten korrekt auflösen zu können müssen die Abhängigkeiten zwischen den Registerpaaren beachtet werden. Wird ein Register sowohl als Eingaberegister für einen Wert, als auch Ausgaberegister für einen anderen Wert genutzt, muss zuerst für das Registerpaar mit dem betroffenen Register als Eingaberegister eine Kopie erstellt werden. Anschließend kann die Kopie für das andere Registerpaar eingefügt werden. Die Abhängigkeiten zwischen den Registerpaaren bedingen also eine Reihenfolge der Kopien. Abbildung 3.7 zeigt eine Visualisierung der Abhängigkeiten für einen Permutationsknoten in Form eines gerichteten Graphen. Ein Knoten repräsentiert eine Kopie, eine Kante von Knoten v zu Knoten u bedeutet, dass die Kopie v von u abhängt, und u somit vor v ausgeführt werden muss. Jedes Register kann Teil von maximal zwei Kopien sein: einmal als Quell- und einmal als Zielregister. Eine Kopie kann daher nur Teil von höchstens zwei Abhängigkeiten sein. Der Abhängigkeitsgraph hat also maximal Eingangsgrad eins und maximalen Ausgangsgrad eins. Der Graph kann somit nur aus einfachen Ketten und Zyklen bestehen. Bilden die Abhängigkeiten innerhalb eines Permutationsknotens mehrere Ketten oder Kreise, liegen diese in unterschiedlichen schwachen Zusammenhangskomponenten.

Bei der in libFirm implementierten Auflösung in Kopien werden zunächst die Ketten in Kopien transformiert. Anhand der anfangs erstellten Listen wird das Ende der Kette identifiziert und für jedes Registerpaar in der Kette eine Kopie eingefügt. Um einen Zyklus aufzulösen, muss erst ein beliebiges Register, welches als Eingaberegister

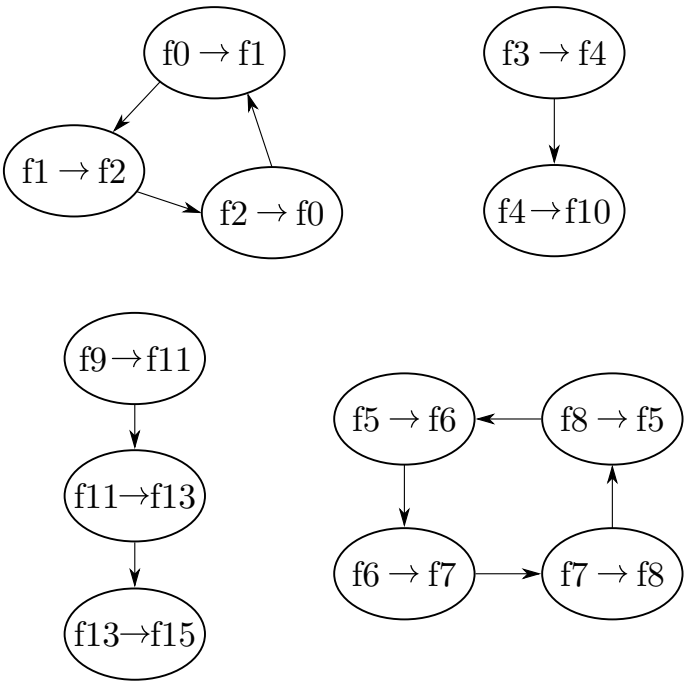


Abbildung 3.7.: Abhängigkeiten innerhalb eines Permutationsknotens, der die Register für zwölf Werte tauscht. Die Abhängigkeiten zwischen den Kopien bilden zwei Zyklen und zwei Ketten. In den Knoten ist das Quell- und Zielregister der Kopie angegeben.

im Zyklus enthalten ist, temporär in ein weiteres, freies Register kopiert werden. Anschließend können die verbleibenden Elemente des Zyklus analog zu den Ketten in Kopien umgewandelt werden. Im letzten Schritt kann dann der in ein temporäres Register kopierte Wert in das eigentliche Zielregister kopiert werden. Wurde vor dem Zyklus bereits eine Kette abgearbeitet, kann das erste Eingaberegister der Kette direkt als freies Register bei der Zyklusauflösung genutzt werden. Andernfalls muss ein freies Register in der Registerklasse gesucht werden. Ist kein Register frei, kann der Zyklus nicht in Kopien aufgelöst werden. In diesem Fall wird der Permutationsknoten in mehrere kleine Permutationsknoten umgewandelt, die jeweils Transpositionen zweier Register darstellen. Solche Transpositionen werden für die SPARC-Architektur dann über ein reserviertes Register umgesetzt.

Werden Werte doppelter Genauigkeit genutzt, können zum einen Permutationsknoten entstehen, deren Register alle Doppelregister sind. In diesem Fall kann die bestehende Implementierung genutzt werden. Zum anderen können Permutationsknoten auftreten, die sowohl Werte in Einzelregistern, als auch Werte in Doppelregistern tauschen. Dabei ist es möglich, dass sich die Menge der Einzel- und die der Doppelregister überschneiden: Da bei der Erstellung des Permutationsknotens den betroffenen Werten komplett neue Register zugeteilt werden, können Register, die vor dem

Permutationsknoten Bestandteil eines Doppelregisters waren, nach der Ausführung als Einzelregister fungieren und umgekehrt. Dadurch ergeben sich komplexere Abhängigkeiten für die Reihenfolge der Kopien. Ein einzelnes Register ist zwar nach wie vor Teil maximal zweier Kopien, da aber Kopien von Doppelregistern zwei Einzelregister kopieren, kann eine Kopie Teil von bis zu vier Abhängigkeiten sein. Für den Abhängigkeitsgraphen ergeben sich so komplexere Möglichkeiten, wie Abbildung 3.8 veranschaulicht. Die Auflösung mit dem bisherigen Algorithmus funktioniert in solchen Fällen nicht mehr.

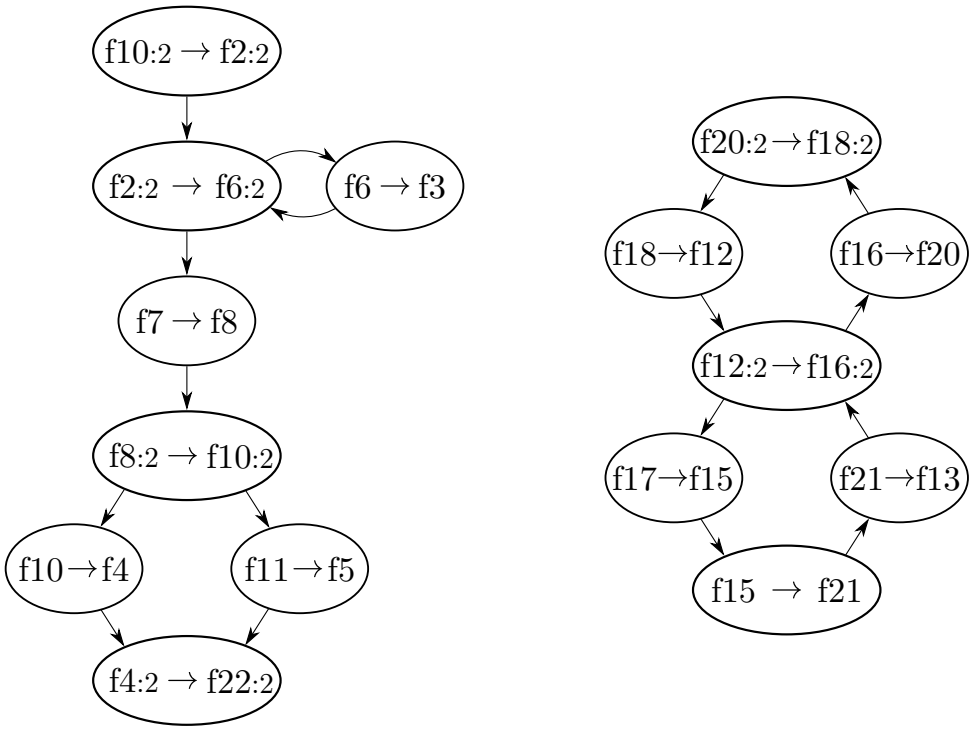


Abbildung 3.8.: Abhängigkeiten innerhalb eines Permutationsknotens, der sowohl Werte in Einzel- als auch in Doppelregistern tauscht. Der Ein- und Ausgangsgrad der Knoten kann hier je maximal zwei betragen. So sind auch komplexere Abhängigkeitsgraphen möglich.

Um für Permutationsknoten mit gemischten Registerbreiten eine korrekte Auflösung in Kopien zu erzielen, wurde der im folgenden Abschnitt beschriebene Ansatz der Aufteilung von Doppelregistern in Einzelregister umgesetzt.

3.5.3. Aufteilung in Einzelregister

Ein Permutationsknoten, welcher sowohl Doppel- als auch Einzelregister enthält, wird zunächst durch einen neuen Permutationsknoten ersetzt, der nur Werte in

Einzelregistern permutiert. Zu jedem Wert in einem Doppelregister erhält der neue Knoten zwei getrennte Werte mit den entsprechenden Ein- und Ausgaberegistern als Einzelregister. Für die beiden Hälften des Doppelregisters ist je ein Registerpaar im neuen Permutationsknoten enthalten. Jedes der beiden Einzelregister kann nur Teil von maximal zwei Abhängigkeiten zwischen den Kopien sein. Dadurch besteht der Abhängigkeitsgraph nur aus einfachen Ketten und Zyklen und die Auflösung des Permutationsknotens erfolgt wie oben beschrieben.

Für jeden Wert, der eigentlich in einem Doppelregister gespeichert ist, werden zwei Kopien eingefügt. Die zwei Hälften werden getrennt kopiert. Möglich ist dies, da eine Kopie eines Doppelregisters zu einem späteren Zeitpunkt ohnehin als zwei Kopien der beteiligten Einzelregister realisiert wird. Der Grund hierfür ist, dass auf der SPARC-Architektur die Kopie eines Doppelregisters durch zwei `fmovs`-Maschinenbefehle realisiert werden muss, wodurch die beiden Hälften eines Doppelregisters getrennt kopiert werden [8].

3.5.4. RegSplit- und RegJoin-Knoten

Um die Aufteilung eines Doppelregisters in zwei Einzelregister im Kontext eines Permutationsknotens zu modellieren und gleichzeitig vor und nach dem Permutationsknoten die Werte korrekt in Doppelregistern zu halten, werden die neuen Knotentypen *RegSplit* und *RegJoin* eingeführt. Da sie erst zum Zeitpunkt der Auflösung des Permutationsknotens in den Firm-Graphen eingefügt werden, stehen die aufzuteilenden Register bereits fest. Der RegSplit-Knoten wird unmittelbar vor dem Permutationsknoten eingefügt. Er erhält als Eingabe einen Wert in einem Doppelregister und generiert als Ausgabe zwei Werte in den Hälften dieses Doppelregisters. Diese dienen dann als Werte, die durch den Permutationsknoten neue Register erhalten. Unmittelbar nach dem Permutationsknoten wird entsprechend ein RegJoin-Knoten eingefügt, der diese beiden Werte als Eingabe erhält und wieder als einen Wert im entsprechenden Doppelregister bereitstellt. Abbildung 3.9 zeigt RegSplit- und RegJoin-Knoten anhand eines Beispiels im Firm-Graphen.

Da die RegSplit- und RegJoin-Knoten nur dazu dienen, die korrekte Transformation der Permutationsknoten in Kopien zu ermöglichen, werden diese bei der späteren Codegenerierung der Maschinensprache nicht berücksichtigt.

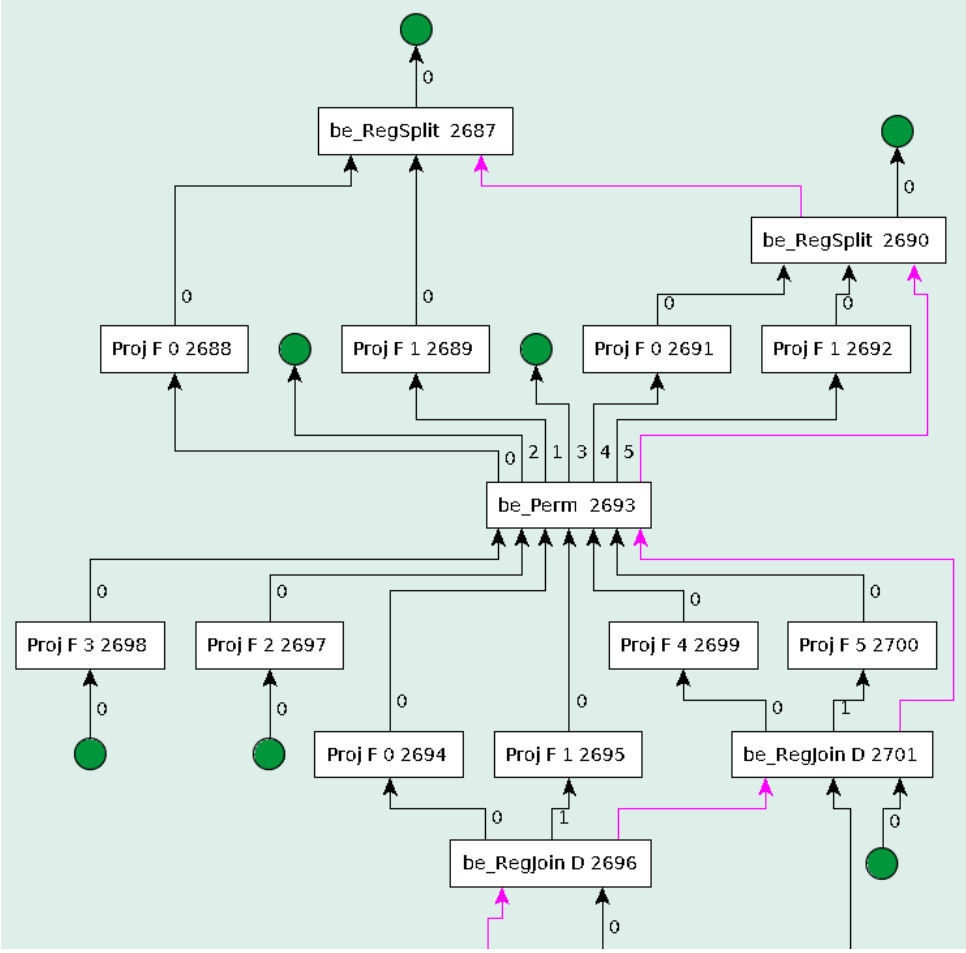


Abbildung 3.9.: Der hier gezeigte Permutationsknoten wurde für den in Abbildung 3.6 abgebildeten Permutationsknoten eingesetzt, durch welchen ursprünglich die Register von vier Werten getauscht werden. Zwei der Werte benötigen Doppelregister, für diese wurden RegSplit- und RegJoin-Knoten eingefügt. Der neue Permutationsknoten tauscht so die Register von sechs Werten in Einzelregistern.

4. Evaluation

4.1. Testumgebung

Während und nach der Entwicklung der vorgestellten Implementierung wurde diese durch Testen auf korrekte Funktionsweise hin evaluiert. Um die Funktionalität der Doppelregister zu testen, müssen Programme für die SPARC V8-Architektur kompiliert werden. Kompiliert wurden zu diesem Zweck verschiedene C-Programme mit `cparser`, dem C-Front-End von libFirm. Anschließend können die Programme mit dem Emulator QEMU ausgeführt werden, ohne einen physischen SPARC-Prozessor zu benötigen.

Da die Kopienminimerung nicht für Doppelregister angepasst wurde, muss diese ausgeschaltet werden, wenn Programme für SPARC kompiliert werden sollen und diese Variablen doppelter Genauigkeit (Variablen des Typs **double** in C) enthalten. Ein beispielhafter Aufruf von cparser für das Programm `test.c` sieht damit wie folgt aus:

Listing 4.1: Beispielhafter Aufruf von `cparser` um das Programm `test.c` für SPARC zu kompilieren

```
cparser --target=sparc-leon-linux-gnu -m32 \
        -mra-chordal-co-algo=none -static test.c
```

Betriebssystem	Ubuntu 17.10
Linux-Kernel	4.13
cparser	1.22.1
QEMU	2.10.1

Tabelle 4.1.: Umgebung unter der Testprogramme kompiliert und ausgeführt wurden um die Funktionalität zu testen.

Um die Performanz der Implementierung zu testen wurde für mehrere Programme jeweils die Laufzeit unter Benutzung von Doppelregistern (*hard-float*) mit der Laufzeit einer Emulation der Gleitkommaarithmetik in Ganzzahlarithmetik (*soft-float*) verglichen. Da Laufzeitmessungen in Emulationslösungen wie QEMU keine belastbaren

Ergebnisse liefern, wurde für diesen Teil der Evaluation eine Hardware-Implementierung der SPARC-Architektur genutzt. Eingesetzt wurde dazu ein LEON3-Prozessor, der auf einem Virtex-7 FPGA implementiert ist. LEON3 ist ein Prozessor der LEON-Familie, die ursprünglich von der Europäischen Weltraumorganisation ESA und später von Gaisler Research entwickelt wurde und die SPARC-Architektur in Version 8 implementiert [1].

FPGA	ProDesign FM-XC7V2000T-R2 (Virtex-7)
Prozessor	LEON3
Taktrate	100 MHz
Betriebssystem	OctoPOS

Tabelle 4.2.: Eckdaten der Testumgebung für die Performanztests.

4.2. Korrektheit

Die *firm-testsuite* ist eine Sammlung von zahlreichen, meist kleinen Testprogrammen für das libFirm-Projekt. Diese wurde genutzt, um die Funktionsweise der Implementierung zu testen. Der Großteil der Testfälle lieferte das erwartete, eine geringe Anzahl von zwölf Testprogrammen jedoch ein abweichendes Ergebnis. Die Hälfte der Fälle mit abweichendem Ergebnis ist auf die Verwendung der Emulation mit QEMU zurückzuführen. Wenn das entsprechende Testprogramm in der soft-float-Variante kompiliert wurde, trat das gleiche Fehlerbild auf. Die verbleibenden von der Erwartung abweichenden Fälle lieferten leicht falsche[1] Werte für die Ergebnisse der in den Programmen durchgeführten Gleitkommaberechnungen. Hier konnte die Fehlerursache nicht geklärt werden. In libFirm wird nach verschiedenen Phasen im Back-End ein *verify*-Schritt ausgeführt, um Fehler zu erkennen. In keinem Fall wies dieser Schritt auf Fehler in der Registerallokation hin. Eine Fehlerursache wurde aufgrund der komplexen Firm-Graphen nicht gefunden.

Damit die korrekte Funktionsweise der Registerallokation auch unter hohem Registerdruck überprüft werden kann, wurde ein sehr einfaches Testprogramm (siehe Listing A.1) geschrieben. Das Programm verwendet eine hohe Zahl an geschachtelten Schleifen, wobei die Schleifenvariablen im innersten Schleifenrumpf verwendet werden. Die Berechnung im innersten Schleifenrumpf erfordert das Vorliegen aller Schleifenvariablen in den Registern. Zusätzlich wird durch die Mischung der Datentypen **float** und **double** für die Schleifenvariablen die Erstellung von Permutationsknoten mit

[1] Im Falle des fbench-Programms beispielsweise ergibt sich im schlechtesten Falle eine Abweichung an der vierten Nachkommastelle.

gemischten Registerbreiten provoziert. Dadurch konnte die korrekte Auflösung der Permutationsknoten wie in Abschnitt 3.5 erläutert überprüft werden.

4.3. Performanz

Für vier Programme wurden Laufzeitmessungen auf der in Abschnitt 4.1 beschriebenen SPARC-Plattform durchgeführt. Alle Testprogramme wurden zunächst als hard-float-Variante und anschließend in der soft-float-Variante kompiliert und ausgeführt. Die soft-float-Variante wurde dabei mit aktivierter Kopienminimerung kompiliert. Dabei wurde jedes Programm in beiden Varianten 20 mal ausgeführt und jeweils die Laufzeit gemessen. Die erste Messung wurde jeweils nicht mit in die Auswertung einbezogen um den Einfluss von „kalten“ Caches zu Beginn zu verringern.

Drei der Testprogramme (n-body.c, spectral-norm.c und partial-sums.c) sind aus dem *Computer Language Benchmarks Game* [17]. Als vierten Benchmark wurde das Programm fbench.c ausgewählt [18]. Alle diese Programme sind auch Bestandteil der firm-testsuite und führen zum Großteil Gleitkommaberechnungen durch. Die Programme wurden leicht angepasst, indem etwaige Ausgaben auf die Konsole entfernt wurden, da diese die Laufzeit aufgrund der langsamen FPGA-Konsole verfälschen. Tabelle 4.3 zeigt eine statistische Auswertung der Messergebnisse.

Benchmark	$\bar{x}$ $[\mu s]$	s_x^2 $[\mu s^2]$	s_x $[\mu s]$
n-body.c (hard-float)	1 276 781,395	46 344,794	215,278
n-body.c (soft-float)	4 038 255,658	222,446	14,915
spectral-norm.c (hard-float)	1 364 589,395	0,322	0,567
spectral-norm.c (soft-float)	16 269 401,342	75,113	8,667
partial-sums.c (hard-float)	1 018 823,421	1,146	1,071
partial-sums.c (soft-float)	4 097 156,816	38,839	6,232
fbench.c (hard-float)	840 449,158	1,696	1,302
fbench.c (soft-float)	3 050 809,500	49,972	7,069

Tabelle 4.3.: Ergebnisse der Laufzeitmessungen. Angegeben sind das statistische Mittel $\bar{x}$, die empirische Varianz s_x^2 und die Standardabweichung s_x der Stichprobe. Alle Werte sind auf 3 Nachkommastellen gerundet.

Für jedes der Testprogramme konnte eine klare Laufzeitverkürzung der hard-float-Variante gegenüber der soft-float-Variante erzielt werden, wie Tabelle 4.4 zeigt. Die geringe Varianz der Messergebnisse unterstreicht dieses Ergebnis. Da in der hard-float-Variante die Kopienminimierung ausgeschaltet ist, enthalten die Programme in Maschinensprache eine sehr große Menge an Registerkopien. Dennoch überwiegt

Benchmark	mittlere Laufzeit $\bar{x}$ in μs hard-float	soft-float	Speedup
n-body.c	1 276 781,395	4 038 255,658	3,163
spectral-norm.c	1 364 589,395	16 269 401,342	11,923
partial-sums.c	1 018 823,421	4 097 156,816	4,021
fbench.c	840 449,158	3 050 809,500	3,630

Tabelle 4.4.: Direkter Vergleich der jeweiligen hard- und soft-float-Varianten. Die Spalte Speedup gibt den Faktor an, um den die hard-float-Variante gegenüber der soft-float-Variante schneller war.

die negative Auswirkung der Emulation der Gleitkommaarithmetik in der soft-float-Variante auf die Laufzeit. Mit einer für Doppelregister angepassten Kopienminimerung sollte der Laufzeitvorteil der hard-float-Variante weiter steigen.

5. Fazit und Ausblick

5.1. Fazit

Die vorgestellte Implementierung einer Registerallokation mit Doppelregisterunterstützung ermöglicht es, für Programme, die Gleitkommazahlen doppelter Genauigkeit verwenden, die Floating Point Unit von Prozessoren der SPARC V8-Architektur voll auszunutzen. Dabei mussten keine neuen Algorithmen für die Registerallokation entwickelt werden, sondern es gelang, die in libFirm bereits bestehenden so anzupassen, dass die Funktionalität erfüllt ist. Es ist somit nicht mehr nötig, auf die Emulation der Gleitkommaarithmetik mittels der Compileroption −msoft−float zurückzugreifen.

Wie die Untersuchungen in Kapitel 4 gezeigt haben, funktioniert die Registerallokation für Doppelregister und liefert erhebliche Verringerungen der Laufzeit der untersuchten Programme. Für diese konnte je nach Programm ein Laufzeitvorteil um den Faktor ≈ 3 bis 12 erreicht werden. Im nächsten Abschnitt wird auf zukünftige Verbesserungen eingegangen, wodurch dieser Gewinn weiter steigen dürfte.

5.2. Ausblick

5.2.1. Verringerung der eingefügten Permutationsknoten

Während der Registerzuteilung werden vor Instruktionen, die Werte in Doppelregistern definieren, Permutationsknoten eingefügt. Die Anzahl der Permutationsknoten im Firm-Graphen kann daher besonders in Programmen, die viele Gleitkommaberechnungen ausführen, sehr groß werden. Es ist allerdings nicht notwendig, vor jeder solchen Instruktion einen Permutationsknoten einzufügen. Beträgt der Registerdruck zum Zeitpunkt der Instruktion weniger als die Hälfte der gesamten Zahl an Registern, kann auf jeden Fall eine Position für ein Doppelregister gefunden werden. Abbildung 3.1 zeigt die ungünstigste Registerzuteilung bei einem Registerdruck, der exakt der Hälfte der Gesamtregisterzahl entspricht. Sobald ein Register weniger belegt ist,

sind mindestens drei aufeinanderfolgende Register unbelegt. In diesem Abschnitt kann ein Doppelregister platziert werden. Die Anpassung, Permutationsknoten nur einzufügen, wenn der Registerdruck mindestens die Hälfte der Gesamtregisterzahl beträgt, stellt eine einfache Optimierung der Implementierung dar. Durch die daraus resultierende geringere Anzahl an Registerkopien ist eine Verringerung der Laufzeit der hard-float-Variante zu erwarten.

5.2.2. Kopienminimierung

Die Kopienminimierung ist ein Schritt, der zum Ende der SSA-basierten Registerallokation ausgeführt wird und die Anzahl an Kopien von Werten in andere Register reduziert. Im Gegensatz zum klassischen Ablauf der Registerallokation (siehe Abbildung 2.2), bei dem durch diesen Schritt potentiell weitere Auslagerungen von Werten nötig werden, stellt die Kopienminimierung in dem in libFirm angewandten Ansatz (Abbildung 2.3) eher eine optionale Optimierung dar. Auch ohne diese liefert die Registerallokation korrekte Ergebnisse. In [12] wird die in libFirm implementierte Kopienminimierung vorgestellt.

Aus Zeitgründen konnte die Kopienminimierung nicht für Doppelregister angepasst werden und muss deshalb beim Kompilieren mit SPARC als Zielarchitektur mit der Option −mra−chordal−co−algo=none deaktiviert werden, sofern Variablen doppelter Genauigkeit im Programm verwendet werden. Die Kopienminimierung könnte die Anzahl der durch einen Permutationsknoten erzeugten Registerkopien verringern, indem Ein- und Ausgaberegister wenn möglich gleichgesetzt werden. Dadurch ließe sich eine weitere Verbesserung der Laufzeit gegenüber der soft-float-Variante von Programmen erzielen.

5.2.3. Gleitkommazahlen vierfacher Genauigkeit

In Abschnitt 2.3.2 wurden neben den Doppelregistern für Werte doppelter Genauigkeit auch Vierfachregister erwähnt, in denen Werte vierfacher Genauigkeit abgelegt werden können. Auch diese Vierfachregister sind Teil der SPARC V8-Architektur. Dennoch beschränkte sich diese Arbeit auf die Betrachtung und Implementierung der Doppelregister. Dies hat den einfachen Grund, dass aktuell keine SPARC-Prozessoren existieren, in deren Floating Point Unit die nötige Hardware für Gleitkommaarithmetik auf Gleitkommazahlen vierfacher Genauigkeit (128 Bit) implementiert ist. Daher muss diese Arithmetik softwareseitig umgesetzt werden und es gibt keinen Bedarf für Vierfachregister.

Einige Teile der im Rahmen dieser Arbeit entwickelten Implementierung sind dennoch

darauf ausgelegt, nicht nur Doppelregister, sondern beliebige Registerbreiten zu unterstützen. Insbesondere das in Abschnitt 3.3 beschriebene Spilling funktioniert für beliebige Registerbreiten. Sollten in Zukunft SPARC-Prozessoren erscheinen, deren Floating Point Unit Gleitkommazahlen vierfacher Genauigkeit unterstützt, oder allgemein Prozessoren, die Register mit größeren Registerbreiten bereitstellen, halten sich die nötigen Anpassungen an die Implementierung der Registerallokation von libFirm in Grenzen.

Literaturverzeichnis

[1] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 409–415, 2002.

[2] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, (New York, NY, USA), pp. 12–27, ACM, 1988.

[3] J. Aycock and N. Horspool, "Simple generation of static single-assignment form," in *Compiler Construction* (D. A. Watt, ed.), (Berlin, Heidelberg), pp. 110–125, Springer Berlin Heidelberg, 2000.

[4] "GNU compiler collection (GCC) internals." `https://gcc.gnu.org/onlinedocs/gccint/`. Letzter Zugriff am 23.03.2018.

[5] "The LLVM compiler infrastructure." `https://llvm.org`. Letzter Zugriff am 23.03.2018.

[6] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[7] "Firm – optimization and machine code generation." `https://pp.ipd.kit.edu/firm`. Letzter Zugriff am 23.03.2018.

[8] SPARC International Inc., *The SPARC Architecture Manual*, 1992. Version 8, Rev. SAV080SI9308.

[9] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47 – 57, 1981.

[10] A. Gibbons, *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[11] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in SSA-form,"

in *Compiler Construction* (A. Mycroft and A. Zeller, eds.), (Berlin, Heidelberg), pp. 247–262, Springer Berlin Heidelberg, 2006.

[12] S. Hack and G. Goos, "Copy coalescing by graph recoloring," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 227–237, ACM, 2008.

[13] P. Briggs, K. D. Cooper, and L. Torczon, "Coloring register pairs," *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 3–13, Mar. 1992.

[14] B. R. Nickerson, "Graph coloring register allocation for processors with multi-register operands," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, (New York, NY, USA), pp. 40–52, ACM, 1990.

[15] M. D. Smith and G. Holloway, "Graph-coloring register allocation for irregular architectures," tech. rep., 2000.

[16] "libFIRM Quelltext. revision 8948846ef." `http://pp.ipd.kit.edu/git/libfirm/`.

[17] I. Gouy, "The computer language benchmarks game." `http://benchmarksgame.alioth.debian.org/`. Letzter Zugriff am 23.03.2018.

[18] J. Walker, "John walker's floating point benchmark." `https://www.fourmilab.ch/fbench/`. Letzter Zugriff am 24.03.2018.

Erklärung

Hiermit erkläre ich, Johannes Bucher, 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

A. Anhang

Listing A.1: Testprogramm um hohen Registerdruck zu erzeugen.

```
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
  unsigned int max = 3;
  double result = 0;

  for (double a = 0; a < max; a++) {
    for (float b = 0; b < max; b++) {
      for (double c = 0; c < max; c++) {
        for (float d = 0; d < max; d++) {
          for (double e = 0; e < max; e++) {
            for (float f = 0; f < max; f++) {
              for (double g = 0; g < max; g++) {
                for (float h = 0; h < max; h++) {
                  for (double i = 0; i < max; i++) {
                    for (float j = 0; j < max; j++) {
                      for (double k = 0; k < max; k++) {
                        for (float l = 0; l < max; l++) {
                          for (double m = 0; m < max; m++) {
                            for (float n = 0; n < max; n++) {
                              result = (n+m-l+k-j+i-h+g+f-e+d*c-b*a);
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  printf("result: %f\n", result);
}
```