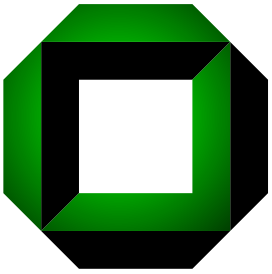


Universität Karlsruhe (TH)

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

Entwurf und Implementierung eines SSA-basierten x86-Backends

Studienarbeit von Christian Würdig

September 2006

Betreuer:
Dipl.-Inform. Sebastian Hack
Dipl.-Inform. Michael Beck
Dipl.-Inform. Rubino Geiß
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Kurzfassung

Diese Studienarbeit beschreibt den Entwurf und die Implementierung eines Backends für Prozessoren der *x86* Familie, ausgehend von der graphbasierten SSA-Darstellung Firm. Die Aufgaben des Backends sind die Codegenerierung und Parametrisierung des Registerzuteilers. Es wird gezeigt, wie die Eigenschaften der Zielplattform modelliert und anschließend in die entsprechenden Befehle umgesetzt werden. Das Backend wurde in die bestehende Infrastruktur integriert und das existierende Rahmenwerk erweitert, um die Unterstützung weiterer Prozessorfamilien zu erleichtern.

Inhaltsverzeichnis

1 Einleitung

Das Backend eines Übersetzers ist für die Überführung der vom Frontend erzeugten Zwischendarstellung in die Maschinenbefehle verantwortlich.

Ziel dieser Arbeit ist es, ein Modul für den am Institut entwickelten Übersetzer zu erstellen. Ausgangsbasis ist ein bereits existierendes Backend-Rahmenwerk mit Registerzuteiler und Befehlsanordner, das definierte Schnittstellen für die Codegenerierung bereit stellt. Neben der möglichst nahtlosen Integration des Moduls in das bestehende System, ist auch darauf zu achten, dass die Erweiterung des Übersetzters um weitere Backends für andere Zielplattformen problemlos zu realisieren ist.

In Kapitel 2 werden zunächst die Zielplattform *x86* als auch die Zwischensprache FIRM auf ihre Besonderheiten im Hinblick auf die Codegenerierung betrachtet. Es werden dabei speziell jene Eigenschaften isoliert, die sich als problematisch in der Überführung gestalten. Anschließend gibt Kapitel 3 einen kurzen Überblick über die verschiedenen Codegenerierungstechniken, deren Vor- und Nachteile sowie deren Umsetzungen in verwandten Arbeiten. Zusätzlich wird der gewählte Lösungsansatz und das konkrete Softwaredesign vorgestellt. In Kapitel 4 wird erläutert, wie die in Kapitel 2 identifizierten Probleme gelöst und in der Implementierung umgesetzt wurden. Kapitel 5 enthält eine Vergleich der verschiedenen architekturspezifischen Optimierung im Hinblick ihrer Auswirkungen auf die Registerzuteilung (Registerdruck, Ein- und Auslagerungscode). Die Arbeit wird in Kapitel 6 mit der Zusammenfassung und einem Ausblick abgeschlossen.

2 Grundlagen

2.1 Die *x86* Architektur

Unter der Bezeichnung *x86* versteht man die Prozessorfamilie von Intel, die 1978 mit dem *8086* ihren Anfang nahm und heute als *Pentium V / Core* (Intel) bzw. *Athlon XP* (AMD) die wohl weitverbreiteste Prozessorfamilie im Endbenutzer PC Markt darstellt. Auch wenn sich die einzelnen Prozessoren heute stark von ihren Vorgängern unterscheiden, insbesondere was die Leistungsfähigkeit angeht, so haben sie doch alle eine gewisse Menge von Befehlen und Eigenschaften gemeinsam, denn ein Programm, das für den *286* übersetzt wurde soll ja auch heute noch lauffähig sein. Für den Backend-Entwurf gibt es dabei mehrere wesentliche Dinge zu beachten.

Die augenscheinlichste Einschränkung ist die Anzahl der Register: es gibt lediglich acht 32-Bit Register, die sogenannten Allzweck (general purpose) Register, die ein Programm direkt verwenden kann[1]: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` und `ESP`. `ESP` enthält den Kellerpegel zur Adressierung des Prozedurkellers und steht nicht zur allgemeinen Verfügung. Es bleiben also maximal sieben zuteilbare Register übrig.

Als nächstes ist zu bemerken, dass die meisten Befehle als Zwei-Adresscode Befehle ausgelegt sind, d.h. dass einer der Operanden auch immer gleichzeitig das Ziel darstellt, in dem das Ergebnis der Operation abgelegt wird. Dies hat natürlich Auswirkungen auf den Registerzuteiler, da so nahezu jeder Befehl zusätzlichen Einschränkungen unterworfen ist. Besonders nachteilig wirkt sich die Zwei-Adresscode Eigenschaft aus, wenn der überschriebene Operand hinterher noch einmal benötigt wird. Dann muss unter Umständen eine extra Kopieroperation eingefügt werden.

Zudem erwarten einige Befehle einen oder beide Operanden in bestimmten Registern oder legen das Ergebnis in fest definierten Registern ab. Auch dies hat wiederum Auswirkungen auf die Modellierung der Befehlsanforderungen.

Die vierte Besonderheit ist beim Umgang mit Gleitkommazahlen zu bemerken – es gibt keine explizit adressierbaren Gleitkommaregister. Die FPU ist als Kellermaschine ausgelegt und verfügt über einen festen Satz an sogenannten *Kellerregistern* `ST0` bis `ST7`. Dabei werden `ST0` und ein beliebiges anderes Kellerregister als Operanden verwendet und das Ergebnis wird wieder in `ST0` abgelegt. Bei dieser Architektur ist die klassische Registerzuteilung offensichtlich

[1] es gibt meist noch interne Register, die von der CPU z.B. zum Ablegen von Zwischenergebnissen verwendet werden, diese können vom Übersetzer allerdings nicht verwendet werden

nicht direkt anwendbar.

Des Weiteren sind die Aufrufkonventionen von Funktionen korrekt abzubilden, die festlegen wie (Reihenfolge) und wo (Register oder Keller) Funktionsparameter und die Resultate übergeben bzw. zurückgegeben werden müssen, und wer die Aufrufschachtel hinterher wieder abbaut (der Aufrufer oder der Aufgerufene). Bei *x86* haben sich dabei drei verschiedene Aufrufkonventionen etabliert:

Pascal die Paramter werden von links nach rechts übergeben und die aufgerufene Funktion baut die Aufrufschachtel ab

cdecl die Parameter werden von rechts nach links übergeben und die aufrufende Funktion baut die Aufrufschachtel ab

stdcall wie *cdecl*, aber die aufgerufene Funktion baut die Aufrufschachtel ab

Das Ergebnis wird dabei immer in `EAX`, `EDX` bzw. `STO` übermittelt.

Es gibt noch eine weitere Variante – die Übergabe von Parametern in Registern, in der Literatur und in manchen Übersetzern auch *fastcall* genannt. Dies ist allerdings kaum standardisiert und abhängig vom gewählten Übersetzer[2]. Um die Kompatibilität zu gewährleisten sollte diese Variante nur bei nicht nach außen sichtbaren Funktionen gewählt werden.

Die Beschreibung der oben genannten Aufrufkonventionen ist allerdings nicht vollständig, denn je nach Betriebssytem, Prozessortyp und Programmiersprache gibt es weitere Abwandlungen der genannten Varianten. So werden z.B. bei 64-Bit Systemen unter Windows maximal vier Parameter in Registern übergeben und zusätzlich müssen vom Aufrufer noch 32 Byte auf dem Keller reserviert werden, auch wenn die aufgerufene Funktion keine Parameter hat. Die aufgerufene Funktion kann beliebige Werte in diesem Bereich ablegen. Unter Linux hingegen werden bis zu 14 Parameter in Registern übergeben und es wird ein spezieller 120 Byte großer Bereich auf dem Keller reserviert, in dem die aufgerufene Funktion beliebige Werte ablegen kann. Funktionsaufrufe können den Inhalt dieses Bereichs allerdings zerstören. Der Übersetzer muss also darauf achten, dass keine Werte, die nach Funktionsaufrufen noch benötigt werden in diesem Bereich abgelegt werden. Für eine umfassende Auflistung der verschiedenen Konventionen der verschiedenen Übersetzter und Betriebssystem sein an dieser Stelle auf [1] verwiesen.

Als letzte Eigenschaft ist noch zu erwähnen, dass die *x86* Prozessoren keine klassischen *Load-Store*-Architekturen darstellen. Die meisten Befehle können einen ihrer Operanden selbst direkt aus dem Speicher laden und gegebenfalls auch das Ergebnis wieder in den Speicher schreiben. Zudem werden komplexe Adressierungspfade unterstützt, die in Intelsyntax[3] wie in Abb. 2.1 dargestellt werden.

Diese Kombination aus komplexer Adressberechnung und implizitem Laden bzw. Speichern wird auch als *Addressmode* bezeichnet und ermöglicht es, z.B.

[2] viele Übersetzer ermöglichen es dem Programmierer sogar die zu verwendenden Register pro Funktion zu spezifizieren

[3] Es gibt noch die AT&T Syntax, vgl. http://sig9.com/articles/att-syntax/

```
[base + index * scale + disp]

base:  Basisregister
index: Indexregister
scale: 0, 1, 2, 4, 8 als Indexskalierung
disp:  32-Bit Konstante als Offset
```

Abbildung 2.1: Adressierungspfad in Intelsyntax

Zugriffe auf ein Element einer lokalen Reihung in einem kompakten Befehl darzustellen: Das *Basisregister* ist der Kellerpegel, im *Indexregister* steht der Index des Elementes, als *scale* wird die Elementgröße gewählt und *disp* gibt die Anfangsadresse der Reihung relativ zum Anfang des Kellers an.

2.2 Die Zwischendarstellung FIRM

FIRM [2] ist eine graphbasierte SSA-Darstellung [3], die am IPD Goos als Zwischendarstellung verwendet wird. Ein Programm besteht aus einer Ansammlung von Graphen, wobei für jede Funktion ein Graph existiert. Jeder Graph ist in Grundblöcke eingeteilt, die durch Blockknoten dargestellt werden. Die Grundblockoperationen werden mittels Operationsknoten abgebildet, die durch Kanten mit ihren jeweiligen Grundblockknoten verbunden sind.

Die Kanten zwischen zwei Operationen stellen in FIRM *Datenabhängigkeiten* dar. Diese Entscheidung wurde zum Einen aus algorithmischen (viele Analysen und Optimierungen basieren auf Datenabhängigkeit) und zum Anderen aus implementierungstechnischen Gründen getroffen, da bei dieser Modellierung eine Operation lediglich ihre benötigten Operanden kennen muss. Deren Anzahl ist meist konstant und die Zeiger auf die Vorgänger können so beim Anlegen des Knotens statisch alloziert werden. In der Datenflussdarstellung müsste die Operation zudem wissen, von welchen anderen Operationen sie benutzt wird – eine Information die sich durch die verschiedenen Optimierungen häufig ändert.

Aufgrund der Eigenschaft der statischen Einmalzuweisung ist es zulässig, dass FIRM die lokalen Variablen einer Funktion nur durch die Ergebnisse, der auf den Definitionen der Variablen vorgenommenen Operationen darstellt. Zwei Ausnahmen davon stellen zum Einen Variablen dar, von denen die Adresse genommen wird, und zum Anderen, im Speicher allozierte Variablen, wie Reihungen und Strukturen. In beiden Fällen werden die Variablen als Datenbehälter modelliert, auf die mit expliziten Lade- und Speicheroperationen zugegriffen wird. Diese Datenbehälter teilen den Speicher in Sektionen ein und werden als symbolische Speicherwerte dargestellt. Jede Operation, die auf dem Speicher arbeitet, erhält zusätzlich zu ihren normalen Operanden einen solchen Speicherwert als Eingabe und produziert als zusätzliches Resultat einen Speicherwert als Ausgabe, der den, unter Umständen veränderten Speicher darstellt. Der Fluss dieser Speicherwerte im Graphen wird über Abhängigkeitskanten modelliert, wodurch auch gleichzeitig eine partielle Anordnung der Speicheroperationen erfolgt. So wird verhindert, dass, z.B. durch eine Optimierung, eine Ladeoperation

vor einer Speicheroperation ausgeführt wird, die auf den gleichen Datenbehälter zugreift.

In FIRM repräsentiert jeder Knoten genau einen Wert. Durch die vorgestellte Art der Modellierung des Speicherflusses gibt es aber Knoten, die mehrere Werte definieren. Um die FIRM-Semantik zu erhalten, liefern entsprechende Operationen weiterhin nur einen Wert zurück, der allerdings ein *Tupel* ist, bestehend aus den Einzelresultaten. Die gewünschten Ergebnisse lassen sich dann mittels *Proj*-Knoten aus diesem Ergebnistupel extrahieren. Diese Darstellung muss vom Backend, z.B. bei der Modellierung der Registeranforderungen der Operationen gesondert behandelt werden, denn es werden nicht der eigentlichen Operation n Register zugeteilt, sondern jedem *Proj*-Knoten wird je ein Register zugeteilt.

Ein weiterer Knotentyp, der gesondert behandelt werden muss, ist der ϕ-Knoten, der durch die Darstellung des Programms in SSA-Form notwendig wird. Diese Knoten bewirken, dass in Abhängigkeit vom Steuerfluss die jeweils gültige Definition eines Wertes weiter verwendet wird. Da ϕ-Funktionen aber keine Entprechungen in Prozessorbefehlen haben, weil sie ein konzeptionelles Konstrukt sind, müssen sie bei der Codeerzeugung abgebaut werden. Dies geschieht in diesem Fall durch den Registerzuteiler, so dass das Backend diese Knoten ignorieren kann.

Da FIRM als Quell- und Zielsprachen unabhängige Repräsentation entwickelt wurde, enthält sie insbesondere keine komplexen Operationen. Dies hat zur Folge, dass einzelne komplexe Befehle eines Prozessors nicht einem Knoten, sondern einem Teilgraphen entsprechen. So entspricht z.B. das im vorhergehenden Abschnitt genannte Beispiel des Adressierungspfades (siehe Abb. 2.1) dem Graphen in Abb. 2.2.

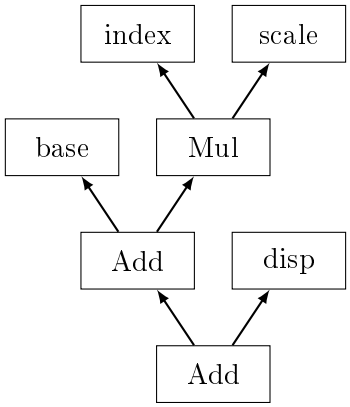


Abbildung 2.2: *x86* Adressierungspfad als FIRM Graph

Dies hat wiederum Auswirkungen auf die Registerzuteilung. Wenn der Wert explizit aus dem Speicher geladen wird, verbraucht er ein zuteilbares Register, während bei der Adressierungspfad Variante lediglich ein, nach aussen nicht sichtbares Register verwendet wird (in irgendein Register muss der Wert zur Berechnung geladen werden), dass dem Registerzuteiler auch sonst nicht zur Verfügung steht.

Um also die Prozessorarchitektur möglichst gut auszunutzen und den Druck

auf die Register zu reduzieren, soll das Backend solche komplexen Befehle selbstverständlich unterstützen und muss dementsprechende Muster erkennen.

Des Weiteren ist zu bemerken, dass in FIRM das Konzept des Prozedurkellers nicht explizit verankert ist. Da Backend muss sich um das Reservieren, Verwalten der Zugriffe und wieder Freigeben des Kellerspeichers selbst kümmern.

3 Codeerzeugung

An dieser Stellen folgen einige Definitionen, um die in dieser Arbeit verwendeten Begrifflichkeiten zu klären.

Definition 3.1 (Befehlsauswahl, Codegenerierung, Codeauswahl). Die *Befehlsauswahl* bestimmt für alle Operationen der Zwischensprache die konkreten Maschinenbefehle und wandelt das Quellprogramm in das Maschinenprogramm um.

Definition 3.2 (Befehlsanordnung, Scheduling). Die *Befehlsanordnung* bestimmt die Ausführungsreihenfolge der Befehle eines Grundblocks und legt die Anordnung der Grundblöcke fest.

Definition 3.3 (Betriebsmittelzuteilung). Die *Betriebsmittelzuteilung* ist hauptsächlich für die Registerzuteilung und die Verwaltung des Speichers für lokale Variablen und Funktionsparameter zuständig.

Definition 3.4 (Codeerzeugung). Unter Codeerzeugung versteht man den Prozess der Abbildung der Zwischensprache auf den symbolischen Maschinencode (Assemblercode) unter Beachtung der Ressourcenbeschränkungen. Dieser Prozess besteht im wesentlichen aus drei Aufgaben: *Befehlsauswahl*, *Befehlsanordnung* (*Scheduling*) und *Betriebsmittelzuteilung*

Da die Registerzuteilung und die Befehlsanordnung bereits implementiert sind, sind die Hauptaufgaben des zu erstellenden Backends zum Einen die Implementierung der Schnittstelle des Registerzuteilers und zum Anderen die Befehlsauswahl.

Für die Befehlsauswahl wurden im Laufe der Jahre verschiedene Algorithmen und Vorgehensweisen entwickelt, die jeweils verschiedene Vor- und Nachteile haben. Dabei wird unterschieden in die *klassische Befehlsauswahl* (3.1), der *Makroexpansion* und die *Entscheidungstabellen* zugeordnet werden, die Befehlsauswahl mit *Termersetzung* (3.2), auf der Verfahren wie *BUPM* und *BURS* basieren, sowie die *Graphersetzung* (3.3) auf die z.B. das PBQP-Verfahren zurückgreift.

Die folgenden Abschnitte zur Beschreibung der einzelnen Verfahren dienen lediglich der Erläuterung der grundlegenden Prinzipien und sollen keine formalen und vollständigen Abhandlungen darstellen, da dies den Rahmen der Arbeit sprengen würde. Weitergehende Information und ausführlichere Referenzen finden sich in der Diplomarbeit von Hannes Jakschitsch [4].

3.1 Klassische Befehlsauswahl

Die Verfahren der klassischen Befehlsauswahl, in der Literatur auch als *interpretierende Befehlsauswahl* bekannt, sind die am einfachsten zu implementierenden. Das Eingabeprogramm, das in der Zwischendarstellung vorliegt, wird dabei von einer abstrakten Maschine ausgeführt, die alle nicht statisch berechenbaren Teile, z.B. Operationen die von erst zur Laufzeit bekannten Werten abhängen, als konkreten Maschinencode ausgibt. Das bekannteste Beispiel für statische Berechenbarkeit ist die Konstantenfaltung. Übersetzer, die mit nur einem Durchlauf über die Zwischendarstellung arbeiten, betreiben in dieser Maschinensimulation auch noch Registerzuteilung sowie alle Zielmaschinen spezifischen Optimierungen (algebraische Vereinfachungen, Kurzauswertungen boolscher Ausdrücke etc.). Dies erhöht natürlich die Komplexität des Verfahrens unter Umständen beträchtlich. Moderne Übersetzter führen diese Optimierungen jedoch in der Regel schon vorher auf der Zwischensprache aus, so dass sich nur noch um die Auswahl der Befehle gekümmert werden muss.

Die zwei Hauptverfahren, die nach diesem Schema arbeiten, sind die Makroexpansion und die Entscheidungstabellen. Bei der Makroexpansion wird im Prinzip für jede Operation der Zwischensprache eine Funktion (oder auch Makro) aufgerufen, die, unter Auswertung der Argumente und der Nebenbedingungen, den entsprechenden Maschinencode ausgibt. Bei der Variante der Entscheidungstabelle existiert für einen Befehl auf Zwischensprachebene eine Tabelle, die für alle Varianten des Befehls, den jeweils auszugebenden Maschinencode enthält.

3.2 Befehlsauswahl mit Termersetzung

Termersetzungsverfahren verfolgen Idee der Trennung der Algorithmen zur Codegenerierung und der Beschreibung der Zielmaschine. Damit soll vermieden werden, den Codegenerator für jede Zielarchitektur neu schreiben zu müssen, sondern im Idealfall ist lediglich eine Zielmaschinenbeschreibung zu erstellen, aus der der Codegenerator generiert wird. Der klassische Ansatz arbeitet auf Zwischensprachen und Zielmaschinenbeschreibung als Baumdarstellung. Die Zwischensprache und die Befehle der Zielmaschine werden als Grammatik in Präfixform spezifiziert und anschließend wird der, durch ein konkretes Programm gegebene, Ausdrucksbaum mittels der Maschinengrammatik zerteilt.

Dies ist ein (unvollständiges) Beispiel einer Grammatik einer Zwischensprache:

```
Expr ::= Op Expr Expr | const | Var
Op   ::= + | - | *
```

Und dies ein Beispiel einer Maschinengrammatik:

```
reg -> var,        30, {print 'LD  $R, $1';}
reg -> +(reg reg),  5, {print 'ADD $R, $1, $2';}
reg -> +(reg var), 35, {print 'ADD $R, $1, $2';}
reg -> *(reg reg), 10, {print 'MUL $R, $1, $2';}
```

Bei den Produktionen der Maschinengrammatik definiert das Nichtterminal der linken Seite den Ressourcentyp des gelieferten Ergebnisses (Register, Speicher) und die rechte Seite beschreibt das Befehlsmuster. An die Regeln werden zusätzlich noch ein Kostenmaß, das bei mehrdeutigen Anwendungen als Entscheidungshilfe dient (normalerweise wird die kostengünstigste Anwendung bevorzugt) und eine Aktion, die bei der endgültigen Anwendung der Regel ausgeführt wird, annotiert.

Die Ausführung dieser Aktionen in Postfixreihenfolge ergeben das zugehörige Maschinenprogramm. Diese Art der Termersetzung, das Finden und Überdecken, ist das einfachste Verfahren und wurde z.B. in CGSS [5] als *LR*-Zerteiler implementiert.

Da einfache Überdeckung aber meist unbefriedigende Codequalität liefert, wurde der Ansatz erweitert zu richtigen Termersetzungssystemen, im folgenden TES genannt, die auch Baumtransformationen erlauben. Damit läßt sich z.B. das Distributivgesetz als Termersetzungsregel wie folgt definieren:

```
*(A +(B C)) -> +(*(A B) *(A C))
```

Diese Art der Regelspezifizierung erlaubt eine relativ kompakte und elegante Spezifikation des Ersetzungssystems, hat aber den Nachteil, dass die Termersetzung auf einem kompletten Baum, aufgrund der in den Regeln enthaltenen Variablen (in diesem Beispiel `A`, `B` und `C`), nicht mehr effizient durchführbar ist. Die Lösung ist die Überführung des TES in ein Grundtermersetzungssystem GTES, in dem die Ersetzungsregeln keine Variablen mehr enthalten. Dies geschieht, indem alle Regeln $l \rightarrow r$, die Variablen enthalten, durch eine Menge von Regeln ersetzt werden, die durch alle *benötigten* Substitution der Variablen mittels Grundtermen (Terme ohne Variablen) entstanden ist. Allerdings ist die Konstruktion eines vollständigen GTES aus einem TES nur semi-entscheidbar, d.h. die entsprechenden Algorithmen terminieren nur, falls das GTES existiert. Das Problem hierbei ist, dass die eigentlich unendliche Menge von Substitutionen so eingeschränkt werden muss, dass die entstehende endliche Menge das gleiche leistet, wie die unendliche Menge.

Dieser Weg, die Spezifikation als Termersetzungssystem zu erlauben und daraus automatisch ein Grundtermersetzungssystem zu generieren, ist in BEG (Backendgenerator) implementiert, der auf der Arbeit von Emmelmann [6] beruht. Für das resultierende Grundtermersetzungssystem kann dann z.B. das Bottom Up Pattern Matching, auch BUPM genannt, eingesetzt werden, das eine effektive Befehlsauswahl ermöglicht. Dabei werden, im Baum von unten nach oben gehend, alle passenden Muster gefunden und anschließend wird von oben herab die kostengünstigste Abdeckung selektiert.

Da, wie bereits erwähnt, die automatische Erzeugung der Grundtermersetzungssysteme nicht immer funktioniert, wurden die Bottom Up Rewrite Systeme, kurz BURS, entwickelt, die direkt mit einer Spezifikation als TES arbeiten. Diese Systeme finden *alle* möglichen Überdeckungen, wobei es exponentiell viele „sinnvolle" gibt[1] und anschließend erfolgt die Suche nach dem globalen Optimum bezüglich der Kosten. Dieses Problem ist im Allgemeinen NP-hart. Zur

[1] eine genauere Analyse findet sich in [7]

Reduzierung der Suchkosten wurde u.a. auf die A*-Suche zurückgergriffen [8], trotzdem sind die resultierenden Codegeneratoren relativ langsam und deutlich komplexer, als auf Grundtermersetzungssystemen basierende.

3.3 Befehlsauswahl mit Graphersetzung

Codeerzeugung mittels Termersetzung hat den Nachteil, dass sie Zwischensprachen in Baumdarstellungen erfordert. Die Zwischensprachen moderner Übersetzer sind heute jedoch in Graphform, da sich nur so die Verwendung gemeinsam genutzter Teilausdrücke richtig modellieren läßt. Der erste Ansatz war, die bekannten Termersetzungsverfahren auf Graphen zu erweitern, indem man den Graph in Bäume aufbricht. Dies führt aber dazu, dass sich z.B. gemeinsam verwendete Teilausdrücke nicht mehr erkennen lassen, es also unter Umständen zu einem Informationsverlust führt, was zu einer schlechteren Codequalität führen kann.

Es gibt auch eine Erweiterung des BURS Ansatzes auf DAGs (Directed Acyclic Graphs, gerichtete nicht zyklische Graphen) durch Boesler [9], die allerdings auch nur Baummuster finden kann. Dieses Verfahren wurde im CGGG von Boesler [8] implementiert, es zeigen sich dabei jedoch die Grenzen des Verfahrens. So ist es z.B. nicht möglich, eine Regel anzugeben, die das IF-THEN-ELSE Muster aus Abbildung 3.1 findet und mittels einer Kopie und eines Conditional Move Befehls überdeckt.

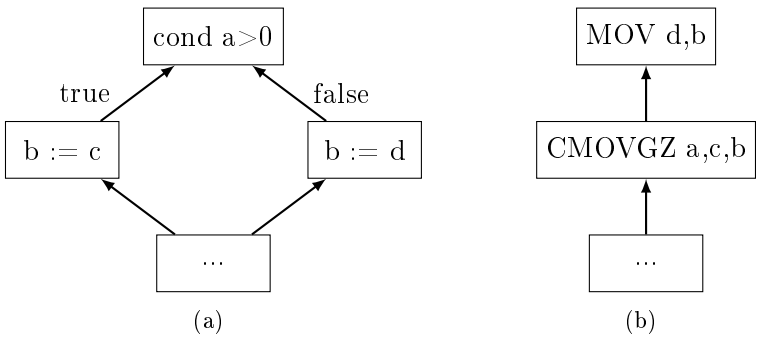


Abbildung 3.1: IF-THEN-ELSE Sequenz (a) und die mögliche Implementierung mittels Conditional Moves (b)

Ein Graphersetzungsverfahren, das auf DAGs operiert, stellt die von Eckstein, König und Scholz vorgeschlagene Abbildung der Codeerzeugung aus SSA-Graphen auf PBQP dar [10]. Dieses Verfahren wurde in Behind von Hannes Jakschitsch [4] implementiert und erlaubt die Spezifikation von Graphmustern.

3.4 Analyse der Verfahren und Lösungsansatz

3.4.1 Analyse der existierenden Verfahren

Wie bereits erwähnt, wurden Term- und Graphersetzungsverfahren entwickelt, um die verwendeten Algorithmen zur Codeerzeugung und die Spezifikation der Prozessoren zu trennen. Im Idealfall implementiert man somit einmal die Algorithmen und muss dann nur noch für jede Zielarchitektur die Spezifikation erstellen, was bei der Unterstützung mehrerer Prozessorfamilien durch einen Übersetzer einiges an Zeit sparen kann. In der Praxis haben die verschiedenen Architekturen allerdings viele spezielle Eigenschaften, die sich in anderen Prozessoren nicht wiederfinden. Die vollständige Abdeckung dieser Ausnahmen mittels einer Spezifikation erhöht deren Komplexität beträchtlich. Damit wiederum wird auch die Erstellung der Spezifikation anfälliger für Fehler. Daher ist die Ausnutzung bestimmter Eigenschaften entweder nicht möglich, oder die Implementierungen bieten Schnittstellen an, die zur Anbindung handgeschriebener Optimierungen dienen. Dies verwischt allerdings wieder die Abgrenzung zwischen verwendeten Algorithmen und der Maschinenbeschreibung. Insbesondere die Aufrufkonventionen für Funktionen müssen für jede Architektur von Hand implementiert werden, da es zu viele verschiedene Varianten gibt, die teilweise sogar vom verwendeten Betriebssystem abhängen, um eine generische Spezifikation dafür zu entwickeln.

Zudem ist bekannt, dass die Phasen Befehlsauswahl, Befehlsanordnung und Registerzuteilung nicht unabhängig voneinander sind, sondern sich gegenseitig beeinflussen. Dies ist mit vollständig automatisch generierten Generatoren nicht implementierbar bzw. die Implementierungen müßten sehr viele Schnittstellen an unterschiedlichen Punkten bereitstellen, so dass man Gefahr läuft, das komplette System total zu zerstückeln.

Eine weitere Beobachtung ist, dass die automatischen Suchverfahren eine Prinzip bedingte hohe Laufzeit besitzen, da sie alle möglichen Überdeckungen finden müssen. So hatte z.B. der BURS basierte CGGG für größere Graphen (ca. 1000 Knoten) eine Laufzeit von mehreren Stunden.

Zum Makroexpansionsverfahren ist zu bemerken, dass es in der Literatur häufig als unpraktikabel und Beispiel für schlechte Codeerzeugung abgetan wird. Dies ist jedoch nicht ganz gerechtfertigt, denn genau genommen wenden eigentliche alle im letzten Abschnitt vorgestellten Verfahren diese Technik an. Die Überdeckungsmuster, die eine Ausgabe produzieren, werden dabei i.d.R. mit einer Vorlage versehen, die von den Codegeneratoren zur Laufzeit für jeden Befehl ausgewertet wird (Einsetzen der konkreten Register, Konstanten, etc.). Anschließend wir der so entstandene Maschinenbefehl ausgegeben.

Die Schlussfolgerung aus all diesen Beobachtungen ist, dass es offensichtlich besser ist, nicht ein einzelnes Verfahren zu forcieren und damit nicht nur dessen Vor- und Nachteile in Kauf zu nehmen, sondern eine Kombination der verschiedenen Systeme zur Implementierung zu nutzen.

3.4.2 Der verwendete Lösungsansatz

Diese Arbeit beruht auf einem Rahmenwerk, das für die verschiedenen Phasen Schnittstellen bereitstellt, in die sich die Codeerzeugung einhängen kann. Die erste Phase beginnt mit der Festlegung der Aufrufkonventionen und der Anordnung der Variablen auf dem Keller. Dies erfolgt zum Teil durch einen generischen Mechanismus, der die benötigten Informationen (Ausrichtung der Variablen auf dem Keller, Kellerrichtung, Register für Kellerpegel und Aufrufschachtel, etc.) per Rückruffunktionen vom Backend erfährt, und zum Teil manuell, indem dem Backend die Möglichkeit gegeben wird, an den entsprechenden Stellen die nötigen Änderungen vorzunehmen.

Anschließend erfolgt der Umbau des allgemeinen FIRM-Graphen in einen Zielplattform abhängigen Maschinengraphen. Dazu registriert das Backend die Architektur spezifischen Knoten als FIRM-Knoten, was in FIRM explizit vorgesehen ist. Damit hat man den großen Vorteil, dass der resultierende Zielgraph mit vorhandenen Werkzeugen der FIRM-Bibliothek, wie z.B. Funktionen zum Ablaufen und Ausgeben von Graphen, bearbeitet werden kann. Da die benötigten Konstruktorfunktionen für die Knoten und auch die Registrierung in FIRM für die meisten Knoten gleichartig sind, wurde, auch im Hinblick auf die Erstellung von Codegeneratoren für andere Plattformen, eine einfache Spezifikationssprache entwickelt (siehe Kapitel A), die die Beschreibung der Knoten mit ihren Eigenschaften, z.B. Registeranforderungen, umfaßt. Hieraus werden die Konstruktoren, die Registrierungsaufrufe sowie die Schnittstelle zum Registerzuteiler generiert. Anschließend erfolgt eine Zielplattform spezifische Optimierungsphase, was beim *x86* im Wesentlichen die Realisierung des Addressmode umfaßt.

Nach der Graphtransformation und -optimierung erfolgt die Befehlsanordnung und die Registerzuteilung. Abschließend wird der endgültige Maschinencode ausgegeben. Die Ausgabe erfolgt per Makroexpansionsverfahren und ist teilweise aus der Spezifikation generiert. Diese ermöglicht es, für einfache Befehle eine Schablone anzugeben, woraus eine Ausgabefunktion generiert wird, die die Platzhalter für Register, Konstanten, etc. entsprechend ersetzt und die resultierende Zeichenkette ausgibt. Für komplexere Knoten oder Muster, wie sie z.B. bei `switch` Anweisungen entstehen, wird die Ausgabe von Hand implementiert. In der Ausgabephase wird der finale Graph entlang der Befehlsanordnungskanten abgelaufen und für jeden Knoten die entsprechende Ausgabefunktion aufgerufen. Insbesondere werden in dieser Phase nur noch Gucklochoptimierungen durchgeführt.

Offensichtlich wird die zentrale Phase des Backends, die Befehlsauswahl, gleich zu Beginn mit der Graphtransformation und -optimierung des FIRM-Graphen in den Backendgraphen durchgeführt. Dabei wird der Graph nur zweimal durchlaufen, womit sich eine Komplexität $O(E)$, mit E = Anzahl der Kanten im FIRM-Graph, für die Auswahl ergibt. Dies bedingt eine statische, nicht kostengesteuerte Entscheidung für einen bestimmten Befehl, was im Allgemeinen zu einer nicht optimalen Überdeckung führt. Zu beobachten ist aber, dass aktuelle Prozessoren, abgesehen von SIMD-Befehlen, keine besonders komplexen Instruktionen besitzen, die ein größeres Muster im Graphen abdecken würden.

Ein `Add` wird zu einem `Add`, ein `Sub` zu einem `Sub`, ein `Mul` zu einem `Mul`, usw. Des Weiteren sind die Optimierungen so implementiert, dass ein komplexer Befehl immer eingesetzt wird, wenn es möglich ist. Dies führt wiederum dazu, dass eine statische Befehlsauswahl zu nicht viel schlechteren Ergebnissen führt, als z.B. BURS, dabei aber wesentlich schneller ist.

4 Die Implementierung

4.1 Modellierung der Registeranforderungen

Um eine korrekte Zuteilung der Register zu ermöglichen, muss für die Backendknoten eine Schnittstelle implementiert werden, über die der Registerzuteiler die benötigten Informationen abfragen kann. Die Grundidee dabei ist, dass, ausgehend vom SSA Ansatz, jedem Knoten maximal ein Register zugeteilt wird. Bei Knoten, die Tupel von Werten produzieren werden daher die entsprechenden `Proj`-Knoten zur Zuteilung herangezogen anstatt die Knoten selbst.

Zusätzlich kann jeder Knoten noch Einschränkungen für Ein- und Ausgangswerte besitzen. So liegt z.B. bei *x86* das Ergebnis der Ganzzahl Division immer im Registerpaar `EDX`:`EAX`. Bei Schiebe- und Rotieroperationen muss der zweite Quelloperanden in `CL` (`ECX`) liegen. Dies läßt sich realisieren, indem für jeden Ein- und Ausgang die Menge der zulässigen Register angegeben wird. Auch die Eigenschaft des Zwei-Adresscodes muss sich in den Registeranforderungen widerspiegeln. Dazu gibt es die Möglichkeit, dem Registerzuteiler einen Gleichfärbewunsch mitzuteilen. Es kann z.B. angegeben werden, dass der Ausgang das gleiche Register bekommen soll wie einer der Eingänge.

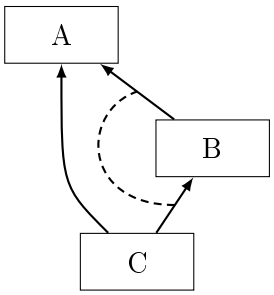


Abbildung 4.1: Beispiel eines nicht erfüllbaren Gleichfärbewunsches

Wie in Abbildung 4.1 zu sehen ist, lässt sich dieser Wunsch nicht immer erfüllen. In diesem Beispiel existiert ein Gleichfärbewunsch, dargestellt durch die gestrichelte Kante, für das Ergebnis von B mit dem Ergebnis von A. Da das Ergebnis von A und B aber nach deren Berechnung noch einmal von C benötigt wird, kann der Registerzuteiler den beiden Knoten nicht das selbe Register zuteilen, sondern er wird für A und B unterschiedliche Register verwenden. Um die Zwei-Adresscode-Semantik zu erfüllen, muss nach der Registerzuteilung zwischen A und B noch eine Kopie des Ergebnisregisters von A in das Ergeb-

nisregister von B eingefügt werden.

Ein weiteres Problem ist, dass es Operationen wie die Multiplikation gibt, die in FIRM nur einen Wert produzieren, beim *x86* aber zwei Register (`EAX` und `EDX`) beschreiben. Die in dieser Arbeit verwendete Lösung besteht darin, dass der entsprechende Maschinenbefehl ein Tupel von Werten zurückliefert, in dem der erste Wert das gewünschte Ergebnis darstellt und die restlichen Werte (für jedes zusätzlich benötigte Register einer) Pseudoergebnisse. Damit der Registerzuteiler erkennt, dass zusätzliche Register benötigt werden, müssen alle Werte dieses Tupels herausprojiziert und verwendet werden. Dazu wird für jeden Wert ein `Proj`-Knoten angelegt, der als Ausgangsbeschränkung das entsprechende Register besitzt. Die Knoten, die vorher den Wert der FIRM-Operation benutzt haben, bekommen den `Proj` als Eingang zugewiesen, der das eigentliche Ergebnis der Maschinenoperation darstellt. Um eine Benutzung der anderen Werte zu erzwingen, wird ein `Keep`-Knoten eingefügt, der beliebig viele Werte verwenden kann und keinen Wert produziert. In Abbildung 4.2 ist am Beispiel der Multiplikation gezeigt, wie sich das der FIRM-Operation entsprechende Muster (a) im *x86* Graphen (b) darstellt.

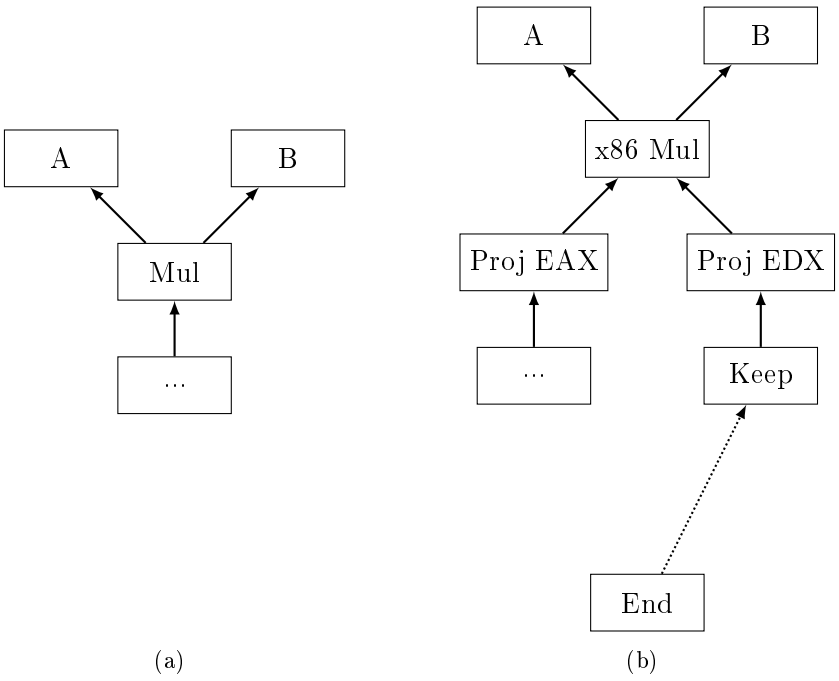


Abbildung 4.2: Multiplikation in FIRM (a) und das erzeugte Muster im *x86* Graphen (b)

Der `Keep`-Knoten ist über eine besondere Kante (gepunktet) mit dem `End`-Knoten verbunden, um ihn am Leben zu erhalten, denn in FIRM werden Knoten, die von keinem anderen Knoten verwendet werden, als toter Code interpretiert und ignoriert.

4.2 Reservierung von Kellerspeicher und Anordnung der Variablen

Die Abbildung der Reservierung von Kellerspeicher und Anordnung der Variablen auf dem Keller erfolgt ebenfalls in einer generischen Phase für die eine Schnittstelle zu implementieren ist. Darüber werden die notwendigen Informationen, in welche Richtung der Keller wächst (nach unten[1] oder oben[2]), Ausrichtung (8, 16, 32, 64 Bit) und zu verwendende Register für Kellerpegel und Schachtelzeiger, erfragt. Anschließend werden, basierend auf diesen Informationen, die entsprechenden Knoten zum Reservieren von Speicher für lokale Variablen und ausgelagerte Register sowie Funktionsparamter in den Graphen eingefügt, ebenso wie die entsprechenden Knoten zur Freigabe dieses Speichers.

Des Weiteren werden Knoten, die dynamische Speicherreservierung auf dem Keller benötigen[3], in die entsprechenden Backendknoten umgewandelt.

4.3 Aufrufkonventionen

Zur Unterstützung der Aufrufkonventionen gibt es eine generische Phase, die allerdings nur einen Teil der benötigten Strukturen einfügt. So werden Knoten zum Laden der Funktionsparameter aus dem Keller eingefügt, ebenso wie Speicheroperationen zum Ablegen von Parametern für aufgerufene Funktionen entsprechend ihrer Konvention (Parameter von links nach rechts bzw. rechts nach links, Aufrufer baut die Aufrufschachtel ab oder der Aufgerufene, etc.).

Zudem werden Muster eingefügt, die dafür sorgen, dass der Registerzuteiler automatisch alle caller-save[4] Register vor Funktionsaufruf sichert und hinterher wieder herstellt, falls sie benötigt werden. Dementsprechende Muster werden auch für alle callee-save[5] Register erzeugt, so dass alle zerstörten Register am Anfang einer Funktion gesichert und am Ende wieder hergestellt werden.

Darüber hinausgehenden Aufrufkonventionen, wie „Ein Wert muss in Allzweck- und Gleitkomma-Registern übergeben werden" oder „Parameter müssen auf dem Keller und in Registern übergeben werden", sind vom Backend selbst zu implementiert, da es zu viele Möglichkeiten gibt, um eine vollautomatische Phase dafür zu erstellen.

4.4 Unterstützung der Gleitkomma-Arithmetik

Die Unterstützung für Gleitkomma-Arithmetik erfolgt mittels der SSE und SSE2[6] Befehle. Das hat den Vorteil, dass die entsprechenden Gleitkommabefehle ganz analog den Integerbefehlen behandelt werden können, ohne die Kellerarchitektur der *x87* Gleitkomma-Einheit modellieren zu müssen. Zudem besitzen

[1] fast alle Prozessoren
[2] z.B. PA-RISC
[3] in C z.B. `alloca` oder VLAs (Reihungen variabler Länge)
[4] müssen vom Aufrufer einer Funktion gesichert werden
[5] müssen von der aufgerufenen Funktion gesichert werden
[6] Operationen mit `double` sind erst seit SSE2 enthalten

SSE Befehle eine geringere Latenz und können ohne Probleme mit MMX Befehlen gemischt werden, da sie über echte eigene Register verfügen, während die MMX Register auf den *x87* Registerkeller abgebildet werden.

Im transformierten Graphen werden die Gleitkommabefehle als eigene Knoten dargestellt, denn der Datentyp ist, im Gegensatz zu FIRM, nicht als Attribut des Knotens erfaßt. Auf diese Weise können die Registeranforderungen statisch vom Knotenkonstruktor erzeugt werden und müssen nicht explizit bei der Graphtransformation gesetzt werden. Des Weiteren spart dies eine Fallunterscheidung bei der Ausgabe der Befehle.

4.5 Unterstützung des Addressmode

Wie bereits in Kapitel 2 erwähnt, können fast alle Befehle einen ihrer Operanden direkt aus dem Speicher laden und das Ergebnis dort wieder ablegen. Für eine „gute" Codeerzeugung ist eine Unterstützung dieser Eigenschaft essentiell, da sich so explizite Adressberechnungen sowie Lade- und Speicherbefehle sparen lassen. Zudem lässt sich damit auch der Registerdruck senken, weil keine extra Register für den geladenen bzw. zu speichernden Wert benötigt werden.

Bei der Modellierung ist zu beachten, dass die Befehle, die Addressmode benutzen, `Load` und gegebenenfalls `Store` Knoten im FIRM Graphen ersetzen und somit den Zustand des Speichers auch verändern und weitergeben müssen, denn der transformierte Graph soll das gleiche Programm repräsentieren wie der originale Graph. Es gibt hierbei mehrere Wege, die Addressmode Unterstützung in FIRM abzubilden.

Eine Möglichkeit ist, einen `AddressMode` Knoten einzuführen, der die Adressberechnung und Lade- bzw. Speicheroperation darstellt. Dies würde allerdings die Ausgabe verkomplizieren, da so jedesmal überprüft werden müßte, ob das Argument ein solcher `AddressMode` Knoten ist und dann den entsprechenden Speicherzugriff ausgeben. Zudem wäre die Darstellung inkonsistent, denn auch Lade- und Speicherbefehle beherrschen die verschiedenen Adressierungspfade. Man müßte also zusätzlich spezielle Addressmode-Lade- und Speicherbefehle einführen, woraus wiederum mehr Fallunterscheidungen bei der Ausgabe resultieren. Die zweite Variante ist, für jeden Befehl einen entsprechenden Addressmode Befehl einzuführen, der zusätzlich noch Argumente für Basis- sowie Indexregister und die Speicherkante besitzt. Damit verdoppelt sich allerdings die Anzahl der Knoten in der Spezifikation und erhöht die Komplexität, da normale und Addressmode Knoten unterschiedlich behandelt werden müssen.

In dieser Arbeit ist eine Verallgemeinerung der zuletzt beschriebenen Variante implementiert, indem die Unterscheidung zwischen Addressmode Operation und nicht Addressmode Operation weggelassen wird. Alle Knoten, die Addressmode beherrschen, besitzen je einen Eingang für Basis- und Indexregister, einen Eingang für jedes eigentliche Argument und einen Eingang für den symbolischen Speicherwert. Dementsprechend produzieren die Knoten auch zwei Werte – das eigentliche Resultat der Operation und den neuen symbolischen Speicherwert.

Der Addressmode-Optimierer ist als allgemeine Funktion implementiert, die

auch mehrfach auf einen Graphen angewendet werden kann, z.B. einmal nach der initialen Transformation und einmal nach dem Einfügen des Aus- und Einlagerungscodes des Registerzuteilers. Es werden zwei verschiedene Aktionen durchgeführt: das Zusammenfalten von Operationen zur Adressberechnung und das Verschmelzen von Lade- und Speicheroperationen mit Operationen.

4.5.1 Zusammenfalten von Adressberechnungen

Adressberechnungen werden durch Additionen identifiziert, bzw., als Ausnahme, Subtraktionen mit Konstanten, die sich als Addition mit negativen Konstanten darstellen lassen. Dabei werden mögliche Adressberechnungen durch folgende Muster identifiziert:

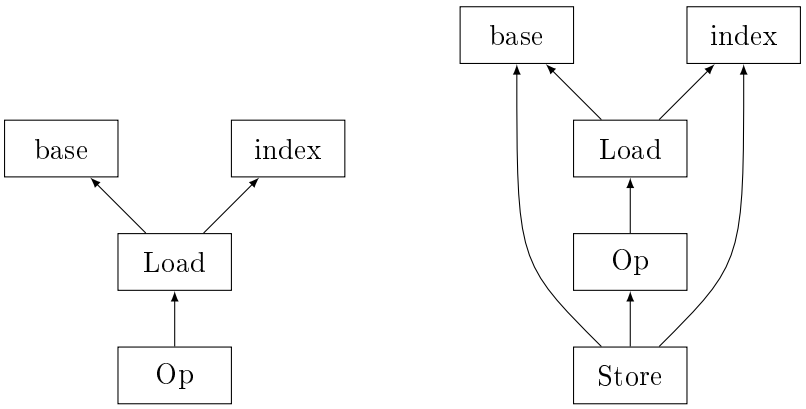
- Subtraktion mit einer Konstanten (Offsetberechnung)
- Addition mit einer Konstanten (Offsetberechnung)
- Addition mit einem `Shl`, das `1`, `2` oder `3` als konstante Shiftweite hat (Indexberechnung mit Skalierung)
- alle anderen Additionen (Basis- und Indexberechnung)

Zur Abbildung dieser Muster gibt es den Maschinenknoten `Lea`. Dieser stellt eine explizite Berechnung des Adressierungspfads dar und ist als Befehl auf *x86* Prozessoren verfügbar. Bei der Umwandlung wird versucht, bereits vorher enstandene `Lea`-Knoten mit dem aktuellen zu verschmelzen. Dies gelingt nur, wenn die Additon einen Teil der Adressberechnung (Basis- oder Indexberechnung) darstellt, den das vorgehende `Lea` noch nicht berechnet, denn es kann nur ein Basis- und ein Indexregister verwendet werden.

Auf diese Art werden auch Additionen umgewandelt, die keine Adressberechnungen sind. Da `Lea` die Addition ebenfalls durchführt, bleibt die Semantik allerdings erhalten. Zudem `Lea` hat den Vorteil, dass es nicht die Zwei-Adresscode Semantik aufweist, d.h. das Zielregister muss nicht zwangsläufig einem der Quellregister entsprechen. Damit sind weniger Knoten im Graphen enthalten, die Registerbeschränkungen aufweisen, was den Freiheitsgrad der Zuteilung erhöht. Somit ist diese Transformation für den Registerzuteiler sogar von Vorteil.

4.5.2 Verschmelzen von Operationen mit Lade- und Speicherbefehlen

Beim Verschmelzen ist zwischen zwei Typen von Addressmodi zu unterscheiden: dem Addressmode als Quelloperand und Addressmode als Zieloperand. Während ersterer dem Hineinziehen eines Ladebefehls entspricht (siehe Abbildung 4.3(a)), ist im zweiten Fall ein komplexeres Muster zu erkennen, nämlich ein Lade und ein Speicherbefehl, die die gleiche Adresse benutzen (siehe Abbildung 4.3(b)). Außerdem ist zu beachten, dass der Ladebefehl nur von einer Operation benutzt wird und zwischen der Operation und dem Ladebefel kein Speicherbefehl liegt.



(a) Quell-Addressmode (b) Ziel-Addressmode

Abbildung 4.3: Die verschiedenen Addressmodi

5 Messergebnisse

Alle Messungen wurden mit dem C-Teil der SPEC Benchmark Sammlung SPEC CPU2000 durchgeführt. Dies erfasst die folgenden Programme: 164.gzip, 175.vpr, 176.gcc, 177.mesa, 179.art, 181.mcf, 183.equake, 186.crafty, 188.ammp, 197.parser, 253.perlbmk, 254.gap, 255.vortex, 256.bzip2 und 300.twolf.

In Tabelle 5.1 erfolgt als erstes ein Überblick über die Entwicklung der Knotenanzahl in den einzelnen Phasen des Backends. Die Zahlen ergeben sich aus der Summe der einzelnen Benchmarks.

Phase	Knoten	Befehle	Load	Reload	Store	Spill
Anfangsgraphen	1.753.827	1.002.199	148.896	0	44.714	0
Kelleraufbau	3.822.890	1.299.115	165.974	0	187.755	0
Befehlsauswahl	3.450.780	899.458	156.466	0	186.224	0
Registerzuteilung	4.132.271	1.239.985	156.466	230.023	186.224	63.294
Finale Graphen	4.464.460	1.324.025	386.870	0	249.514	0

Tabelle 5.1: Knotenstatistik der einzelnen Phasen

Die Spalte „Knoten" erfasst die Gesamtzahl aller Knoten aller Graphen. Als „Befehle" wurden alle Knoten gezählt, die zu einer Ausgabe eines Assemblerbefehls führen. Der Term „Befehl" wird auch im Folgenden in dieser Weise verwendet.

Zu beobachten ist, dass zu Beginn lediglich knapp die Hälfte aller Knoten Befehle sind und am Ende sogar nur ein Viertel. Dies liegt hauptsächlich an der Modellierung von multiplen Rückgabewerten mittels `Proj`-Knoten, die zu keiner Anweisung führen (siehe auch Kapitel 2). Der Großteil dieser Knoten wird in der Kelleraufbauphase eingefügt. Hier werden Register, die vom Aufrufer einer Funktion zu sichern sind, als Rückgabewerte des Funktionsaufrufes dargestellt. So kann der Registerzuteiler erkennen, welche Register auszulagern sind, bevor die Funktion aufgerufen wird.

Weiterhin sind extra Zahlen für Lade- und Speicheroperationen erfasst. Diese sind noch einmal in normale Befehle und vom Registerzuteiler eingefügten Ein- (Reload) und Auslagerungscode (Spill) unterteilt. Man kann erkennen, dass die Anzahl der Speicheroperationen in der Kelleraufbauphase um ca. 75% ansteigt. In dieser Phase wird für jeden Funktionsparameter eine Speicheroperation erzeugt, die den Wert auf dem Keller ablegt. Am Ende sind also nur gut ein

Sechstel aller Speichoperationen normale Operationen, ca. ein Viertel Registerauslagerungen und mehr als die Hälfte sind zur Parameterübergabe notwendig. Bei den Ladeoperationen hingegen entsprechen fast 60% Registereinlagerungen und nur 40% normalen Ladebefehlen. Alles in allem erhöht sich der Anteil von Speicheroperationen an den Befehle durch den Registerzuteiler um ca. 20% von einem Drittel auf mehr als die Hälfte.

Ebenfalls zu bemerken ist, dass die Befehlsauswahlphase die Anzahl der Knoten um ca. 10% verringert. Zum Einen ist das auf den Addressmode Optimierer zurückzuführen, der Adressberechnungsmuster zu einzelnen Befehlen verschmilzt. Zum Anderen auf die Eigenschaft des *x86*, dass fast jede Anweisung eine beliebige 32-Bit Konstante direkt als Operand verwenden kann. Diese Konstanten werden als Attribut in dem jeweiligen Knoten aufgenommen und stellen damit keinen Befehl mehr dar. Möglich ist diese Vereinigung bei ca. 90% aller Konstanten, wobei etwa 10% aller Befehle Konstanten entsprechen.

Benchmark	Befehle ges.	Lea	Befehle Lea	Verschmolzen
164.gzip	10.080	1.201	1.934	681
175.vpr	41.747	6.265	7.175	4.735
176.gcc	422.868	47.612	54.776	41.385
177.mesa	147.998	22.547	26.912	16.743
179.art	3.263	370	398	203
181.mcf	2.769	460	525	356
183.equake	4.443	593	804	349
186.crafty	55.829	9.985	16.252	8.637
188.ammp	34.642	4.229	4.850	3.451
197.parser	27.815	3.394	4.353	2.718
253.perlbmk	175.196	14.909	16.073	11.119
254.gap	167.231	19.511	25.194	12.881
255.vortex	159.039	30.676	32.021	27.525
256.bzip2	7.760	865	1.277	463
300.twolf	63.345	8.598	9.486	6.296
Summe	1.324.025	171.215	201.887	137.541

Tabelle 5.2: Statistik zu Adressberechnungsmustern

Tabelle 5.2 gibt einen Überblick über die Befehle für Adressberechnungsmuster. In der ersten Spalte sind die einzelnen Benchmarks aufgeführt und in der zweiten, als Referenz, die Anzahl aller Befehle in diesem Benchmark nach der letzten Phase. Die dritte Spalte enthält die Anzahl der erzeugten `Lea`-Knoten und die vierte, die Anzahl der dazu verwendeten Befehle. Als letzte Spalte ist die Anzahl der `Lea`-Knoten aufgeführt, die mit einer Speicheroperation verschmolzen wurden und somit keine expliziten Assemblerbefehle mehr darstellen.

Im Schnitt werden der Tabelle zu Folge ca. 1,18 Operation pro `Lea` ver-

wendet. In Zusammenhang mit den knapp 140.000 verschmolzenen `Lea`s ergibt dies mehr als 160.000 Anweisungen, die im Endeffekt mit Speicheroperationen zusammengeführt wurden. Es entsprechen also mehr als 12% aller Anweisungen Adressberechnungen, die sich zu 90% aus Additionen, 6% Multiplikationen (Shifts) und 4% Konstanten zusammensetzen.

Die Unterstützung des Addressmodes spart nicht nur nur Operationen ein, sondern senkt auch den Registerdruck, wie in Abbildung 5.1 zu sehen. Dort ist die Entwicklung des Registerdrucks der Allzweckregister einmal mit und einmal ohne Unterstützung des Addressmodes abgebildet.

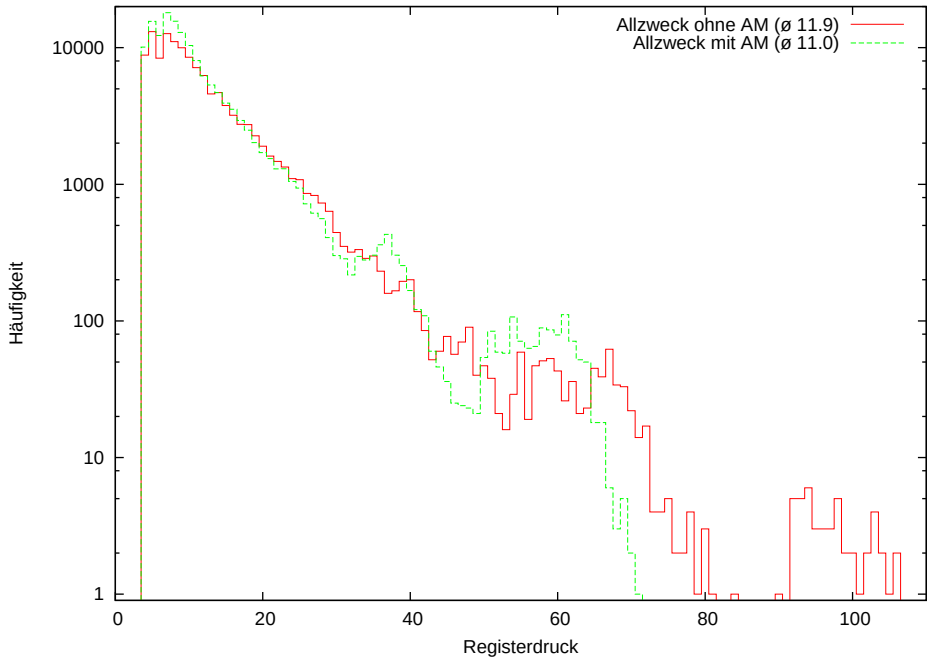


Abbildung 5.1: Registerdruck der Allzweckregister mit und ohne Addressmode

Dazu wurde der Druck an jedem Knoten gemessen und eine Häufigkeitsverteilung aufgestellt. Die Häufigkeit ist zwecks besserer Übersicht logarithmisch aufgetragen. Zu beobachten ist, dass die Häufigkeit in bestimmten Bereichen, z.B. zwischen 50 und 65, mit Addressmode ansteigt, aber bei höheren Werten hingegen abnimmt. Der maximale Registerdruck sinkt von 106 auf 70 und der durchschnittliche um ca. 8% von 11.9 auf 11.0. Bei den Gleitkommaregistern reduziert sich der durchschnittliche Registerdruck sogar um fast 12%.

Ein weiteres Augenmerk liegt auf der Übersetzungszeit für ein Programm. In Abbildung 5.2 sind die absoluten Laufzeiten der vier Hauptphasen des Backends für einzelne Graphgrößen angegeben.

Zu erkennen ist, dass die Laufzeit der einzelnen Phasen linear wächst, wobei es allerdings zwei sprunghafte Anstiege gibt: Einmal zwischen 500 und 1500 Knoten und einmal zwischen 4500 und 6000 Knoten. Dies liegt vermutlich daran, dass der Speicherverbrauch eine Grenze überschreitet, an der mehr Aufwand zur Speicherverwaltung notwendig ist. Am stärksten wird davon die Laufzeit des Registerzuteilers beeinflusst.

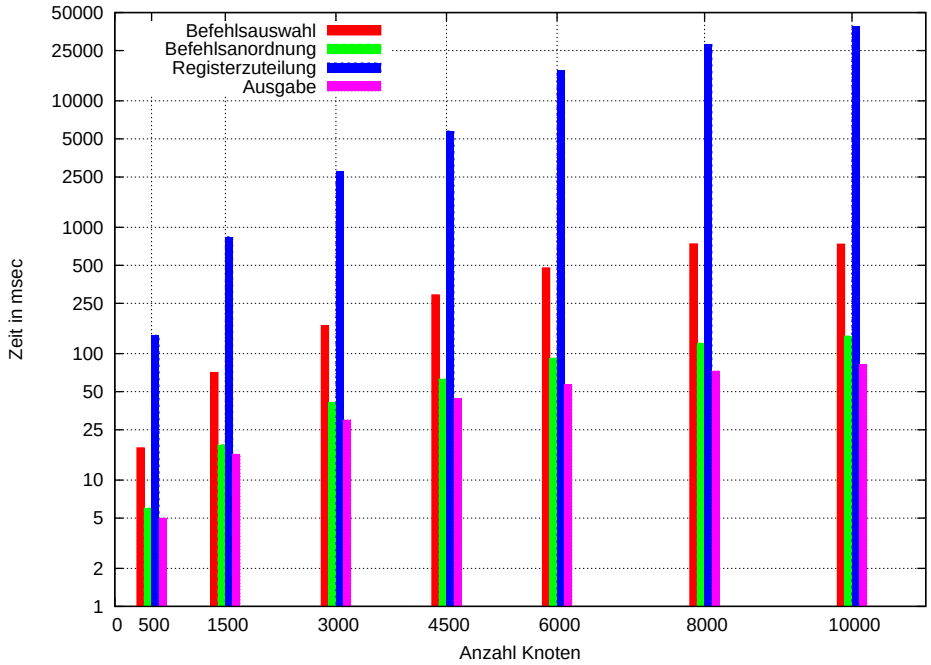


Abbildung 5.2: Absolute Laufzeit der einzelnen Backendphasen

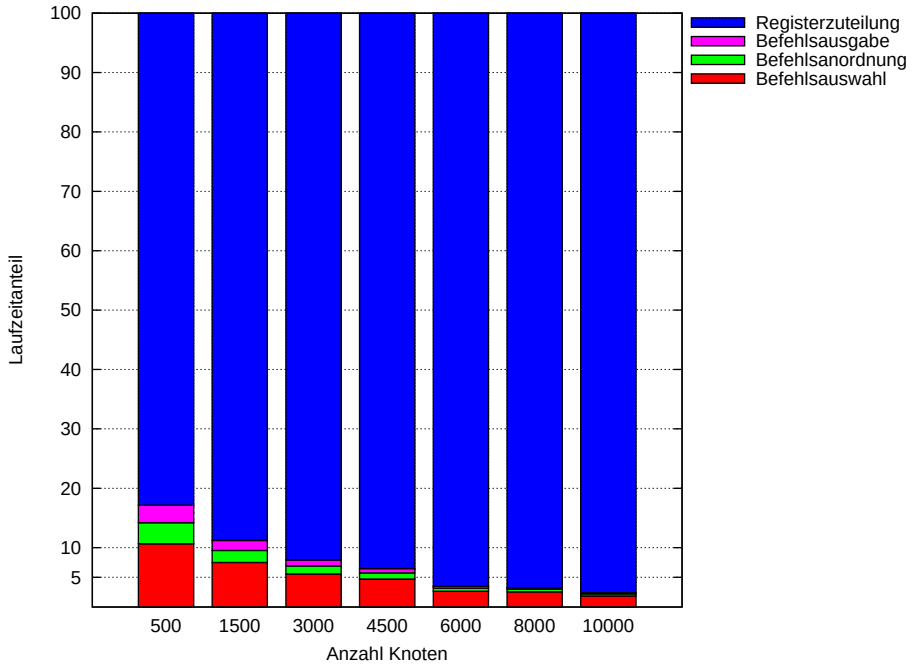


Abbildung 5.3: Anteil der einzelnen Phasen an der Backendlaufzeit

Dessen Laufzeit steigt daher mit einem Proportionaliätsfaktor größer als eins, wie in Abbildung 5.3 zu sehen ist. Dort ist der Anteil der einzelnen Phasen an der Laufzeit des Backends dargestellt.

Der Anteil des Registerzuteilers steigt dabei von ca. 83% bei 500 Knoten auf 97% bei 10000 Knoten.

6 Zusammenfassung und Ausblick

Die Zeit für den Entwurf und die Implementierung des *x86* Backends betrug ca. drei Monate. Dabei wurde ein Rahmenwerk geschaffen, dass als Vorlage für weitere Backends dienen kann. Wie ein durchgeführtes Blockpraktikum belegt, ist es mittels dieser Vorlage möglich, innerhalb von drei Wochen ein lauffähiges Modul zu erstellen. So wurden im Rahmen des Praktikums ein ARM, ein PowerPC und ein MIPS-DLX Backend prototypisch implementiert. Bei Prozessoren die spezielle Eigenschaften wie SIMD Befehle besitzen ist u.U. etwas mehr Aufwand zur Nutzung und Ausreizung der entsprechenden Architekturen nötig.

Im Hinblick auf das *x86* Backend sind noch einige Verbesserungen und Optimierungen offen:

- So ist eine Unterstützung der klassischen Gleitkommaeinheit wünschenswert, um die Programme auch auf nicht SSE1/2 fähigen Rechnern laufen lassen zu können.
- Weiterhin fehlt Unterstützung für Programme, die mit 64Bit Werten[1] arbeiten. Dazu ist noch eine Phase nötig, die die entsprechenden 64Bit Operationen in 32Bit Operationen aufteilt.

Eine weitere sinnvolle Erweiterung ist die Nutzung der SIMD Eigenschaft der SSE Einheit. Das Problem hierbei ist, dass die Graphmuster, die SIMD Befehlen entsprechen, so komplex sind, dass eine von Hand implementierte Suche nicht sinnvoll ist. Abhilfe können an dieser Stelle Graphersetzungswerkzeuge wie GrGen [11] schaffen. Kombiniert man diese mit einem Ansatz, wie ihn Enno Hofmann in seiner Diplomarbeit [12] vorgestellt und implementiert hat, in dem die zu findenden Graphmuster aus einer Spezifikation[2] erzeugt werden, so lässt sich die Suche nach den SIMD Mustern automatisieren.

[1] z.B. `long long` in C

[2] wie sie in vielen Prozessorhandbüchern schon zu finden ist

A Die Spezifikationssprache

Die Spezifikation erfolgt in Perl, d.h. die Beschreibung der Knoten und Registeranforderungen stellen Initialisierungen von Perl Datenstrukturen, wie Hashes und Arrays, dar. Dies hat den Vorteil, dass kein extra Parser für die Spezifikation geschrieben werden muss, sondern sie direkt in die Generatorskripte, die auch in Perl geschrieben sind, eingebunden werden kann. Dazu müssen folgende Variablen ausgefüllt werden:

`$arch` - Architekturkürzel (z.B. ia32, sparc, mips, ppc, ...)

`$comment_string` - die Zeichenkette, die einen Kommentar in der Assemblerausgabe einleitet; wird benutzt um Code und Kommentare formatiert auszugeben

`$additional_opcodes` - Falls zusätzlich opcodes registriert werden sollen (außer den spezifizierten), dann kann hier die Anzahl der zusätzlich benötigten opcodes angeben.

`%reg_classes` - Beschreibung der Registerklassen

`%nodes` - Beschreibung der Backendknoten

A.1 Die Architektur `$arch`

Das hier angegebene Architekturkürzel wird für die Namen der generierten Dateien und Funktionen verwendet, um Namenskonflikte zu vermeiden. Im Allgemeinen sollten alle nach außen sichtbaren Funktionen, Variablen, Typen, etc. dieses Kürzel beinhalten. Bsp.:

```
$arch = "ia32";
```

A.2 Die Variablen `$comment_string` und `$additional_opcodes`

Die Variable `$comment_string` enthält die Zeichenkette, die einen Kommentar im Assemblertext einleitet. Sie wird dazu verwendet, im Ausgabemakro Code und Kommentar formatiert auszugeben. Bsp.:

```
$comment_string = "/*";
```

Die Variable `$additional_opcodes` ist optional. Mann kann damit bei Bedarf zusätzliche opcodes in FIRM reservieren, falls man spezielle Knoten anlegen will, die sich nicht mit dem Generator erzeugen lassen.

A.3 Die Registerklassen `%reg_classes`

Diese Variable ist ein Hash, mit einem Namen für jede Registerklasse als Schlüssel, der auf ein Array zeigt, dass die Registerbeschreibungen enthält. Jedes dieser Register wird durch einen Hash beschrieben, wobei jeder dieser Hashes zwei Schlüssel enthält – `name` zeigt auf den Registernamen und `type` zeigt auf den Registertyp. Als Registername kann eine beliebige Zeichenkette bestehen aus `A-Z`, `a-z`, `0-9` und `_` angegeben werden. Der Registertyp kann mit Hilfe der folgenden vier Flags definiert werden:

- `0` - kein spezieller Registertyp
- `1` - caller-save Register, d.h. dieses Register muss vom Aufrufer einer Funktion gesichert werden
- `2` - callee-save Register, d.h. dieses Register muss von der aufgerufenen Funktion gesichert werden
- `4` - ignore Register, d.h. dieses Register wird bei der Registerzuteilung nicht berücksichtigt und muss ggfs. von Hand gesetzt werden (sinnvoll z.B. beim Register für Kellerpegel)

Wenn mehrere dieser Flags gesetzt sein sollen, dann sind die entsprechende Werte zu addieren. Der letzte Eintrag in dem Array mit den Registerbeschreibungen ist ein besonderer Hash, der als Schlüssel `mode` besitzt. Dieser zeigt auf den größten FIRMMode, den ein Register dieser Klasse aufnehmen kann.
Beispiel, die Beschreibung der `ia32` Registerklassen `gp` (General Purpose) und `fp` (Floating Point):

```
%reg_classes = (
  "gp" => [
            { "name" => "eax", "type" => 1 },
            { "name" => "ebx", "type" => 2 },
            { "name" => "ecx", "type" => 1 },
            { "name" => "edx", "type" => 1 },
            { "name" => "esi", "type" => 2 },
            { "name" => "edi", "type" => 2 },
            { "name" => "ebp", "type" => 2 },
            { "name" => "esp", "type" => 6 }, # callee-save & ignore
            { "name" => "xxx", "type" => 6 }, # dummy register
            { "mode" => "mode_P" }
          ],
  "fp" => [
            { "name" => "xmm0", "type" => 1 },
            { "name" => "xmm1", "type" => 1 },
            { "name" => "xmm2", "type" => 1 },
```

```
            { "name" => "xmm3", "type" => 1 },
            { "name" => "xmm4", "type" => 1 },
            { "name" => "xmm5", "type" => 1 },
            { "name" => "xmm6", "type" => 1 },
            { "name" => "xmm7", "type" => 1 },
            { "name" => "xxxx", "type" => 6 },  # dummy register
            { "mode" => "mode_D" }
          ]
);
```

A.4 Die Knoten %nodes

Die Backendknoten sind in der Hash Variablen `%nodes` beschrieben. Jeder Schlüssel entspricht einem Knotennamen und zeigt auf die zugehörige Knotenbeschreibung. Der endgültige Knotenname wird aus dem Architekturkürzel und dem hier angegebenen Namen gelbildet. Bsp:

```
$arch = "ia32";

%nodes = (

"Add" => {
   ...
}

);
```

Der resultierende Knoten heißt `ia32_Add`.
Die Knotenbeschreibung ist wiederum ein Hash, für den die folgenden Schlüssel definiert sind:

`op_flags` Die hier angebenen Flags entsprechen den `irop_flags` aus FIRM. Sollen mehrere Flags gesetzt sein, so sind sie mit `|` zu verbinden. Folgende Werte sind definiert:

- `N` - `irop_flag_none`
- `L` - `irop_flag_labeled`
- `C` - `irop_flag_commutative`
- `X` - `irop_flag_cfopcode`
- `I` - `irop_flag_ip_cfopcode`
- `F` - `irop_flag_fragile`
- `Y` - `irop_flag_forking`
- `H` - `irop_flag_highlevel`
- `c` - `irop_flag_constlike`
- `K` - `irop_flag_keep`

Dieser Eintrag ist optional und kann auch entfallen, dann wird als Standard `N` verwendet.
Bsp.: `op_flags => "L|X"`

`irn_flags` Mit diesen Flags kann man die Registerallokation des jeweiligen Knotens steuern. Es können ebenfalls mehrere Flags gesetzt sein, die dann entsprechend mit `|` verbunden werden. Folgende Werte sind definiert:

- `N` - `dont_spill`, d.h. der Wert dieses Knotens kann nicht ausgelagert werden, falls der Registerdruck zu hoch wird
- `R` - `rematerializable`, d.h. es ist u.U. billiger den Wert neu zu berechnen, anstatt ihn auszulagern
- `I` - `ignore`, d.h. dieser Knoten soll vom Registerallokator nicht beachtet werden, die Registerzuteilung muss dann von Hand geschehen

Der Eintrag ist optional und kann entfallen, dann werden keine besonderen Flags gesetzt.
Bsp: `irn_flags => "R"`

`arity` Die `arity` gibt die Anzahl der Eingänge des Knotens an. Folgende Werte können angeben werden:

- `0,1,2,...` - ein beliebige Dezimalzahl größer oder gleich 0, jeder dieser Knoten hat dann die hier festgelegte Zahl an Eingängen
- `variable` - die Anzahl der Eingänge pro Knoten ist verschieden, aber nach der Konstruktion fest (z.B. bei `Call` Knoten)
- `dynamic` - die Anzahl der Eingänge kann auch nach der Knotenkonstuktion noch variieren
- `any` - die Stelligkeit ist nicht näher festgelegt

Dieser Eintrag ist optional und kann entfallen, dann wird die Stelligkeit aus der Angabe der Registeranforderungen für die Eingänge berechnet. Man muss ihn also nur setzen, wenn man explizit eine andere Stelligkeit angeben will.
Bsp.: `arity => "variable"`

`state` Hiermit wird angegeben, in welchem Zustand sich Knoten dieses Typs befinden. Folgende Werte sind definiert:

- `floats` - Knoten diesen Typs können beliebig zwischen Blöcken verschoben werden
- `pinned` - Knoten diesen Typs können nicht aus dem Block heraus verschoben werden, in dem sie erzeugt wurden
- `exc_pinned` - Knoten diesen Typs können nicht aus dem Block heraus verschoben werden, in dem sie erzeugt wurden, falls sie eine Ausnahme erzeugen können

`mem_pinned` - Knoten diesen Typs können nicht aus dem Block heraus verschoben werden, in dem sie erzeugt wurden, falls sie eine Ausnahmeerzeugen können oder auf Speicher zugreifen

Dieser Eintrag ist optional und kann entfallen, dann wird als Standard `floats` verwendet.
Bsp.: `state => "pinned"`

`reg_req` Dieser Wert dient der Beschreibung der Registeranforderungen für die Knoteneinund -ausgänge. Die Angabe erfolgt als Hash mit zwei Schlüsseln – `in` für die Eingänge und `out` für die Ausgänge. Jeder dieser beiden Schlüssel zeigt auf ein Array mit Strings, die die Anforderungen für den jeweiligen Ein-/Ausgang beschreiben. Aus der Anzahl der Strings im Feld für `in` kann auch die Stelligkeit (siehe `arity`) berechnet werden. Es gilt die folgende Syntax:

`none` - Dies bedeutet, dass keine Register zugeteilt werden sollen (z.B. bei Ein-/Ausgängen von Speicherkanten).

`Name einer Registerklasse` - Wenn der Name einer der in `%reg_classes` definierten Registerklassen angegeben wird, dann kann jedes Register dieser Klasse verwendet werden.

`Name eines oder mehrere Register einer Klasse` - Man kann die möglichen Register auch aufzählen (getrennt durch Leerzeichen), dann sind nur die erwähnten Register zulässig.
Bsp.: `"eax edx"`
Es ist auch möglich die Register zu negieren, dann heißt das, das alle bis auf die negierten Register zulässig sind.
Bsp.: `"!ebx !ecx"`
Das Mischen von Registern verschiedener Klassen ist nicht zulässig. Das Mischen von negierten Registern und nicht negierten Registern ist ebenfalls nicht zulässig.
`"eax xmm0"` → falsch
`"!eax edx"` → falsch

`in_rX` - Nur bei den Ausgabeanforderungen zulässig. Damit kann man den Wunsch ausdrücken, dass der durch `X` (`X` $>= 1$) angegebene Eingang das gleiche Register „zugewiesen" bekommen soll wie der aktuelle Ausgang. Dies kann vom Registerallokator allerdings nicht immer erfüllt werden, so dass ggfs. hinterher von Hand eine Kopie eingfügt werden muss.
Bsp.: `"in_r1"`
Es ist auch möglich diese Angabe zu negieren und damit ungleiche Register zu fordern. Im Gegensatz zur Gleichfärbung ist dies dann nicht nur ein Wunsch, sondern eine Forderung, die auch immer sichergestellt wird.
Bsp.: `"!in_r2"`
Es ist möglich normale und negierte Angabe zu mischen, allerdings

ist nur eine normale und/oder negierte Angabe pro Ausgang zulässig.
`"in_r1 !in_r2"` → richtig
`"in_r1 in_r2"` → falsch
`"!in_r1 !in_r2"` → falsch
Ebenfalls möglich ist das Msichen mit der Angabe von Registernamen, so dass sich z.B. die Anforderung: „Das Ergebnis liegt immer in Register `eax`. Der erste Operand sollte ebenfalls in `eax` sein, es darf aber auf keinen Fall der zweite Operand in `eax` liegen." folgendermaßen realisieren läßt:
`"eax in_r1 !in_r2"`

Die Angabe der Registeranforderungen ist optional und kann entfallen, allerdings muss dann die Stelligkeit spezifiziert werden. Zu bemerken sei noch, dass Eingängen keine Register zugeteilt werden, sondern die Angabe der Anforderungen dient der Steuerung der Zuteilung der Register zu den Ausgängen und ggfs. dem Einfügen von Kopien.
Bsp.: `"in" => [ "gp", "gp" ], "out" => [ "in_r1" ] }`

`comment` Optionaler C-Kommentar, der vor den erzeugten Knotenkonstruktor eingefügt wird.
Bsp.: `comment => "Store(ptr, val, mem) = ST ptr,val"`

`rd_constructor` Hiermit kann optional der Quellcode des Knotenkonstruktors angegeben werden, falls nicht der Standardkonstruktor verwendet werden soll. Die Signatur wird fogendermaßen generiert:

```
ir_node *new_rd_<Knotenname>(dbg_info *db, ir_graph *irg, ir_node *block,
                             ir_node *op1 ..., ir_node *opX, ir_mode *mode)
```

`op1` bis `opX` stehen für die Operanden, die der Knoten verwendet, wobei `X` die Anzahl der Eingänge ist. Der Standardkonstruktor sieht wie folgt aus (`Y` steht für die Anzahl der Ausgänge):

```
ir_node *new_rd_<Knotenname>(dbg_info *db, ir_graph *irg, ir_node *block,
                             ir_node *op1, ..., ir_node *opX,
                             ir_mode *mode)
{
  ir_node *res;
  ir_node *in[X];
  int flags = 0;
  static const $arch_register_req_t *_in_req_<Knotenname>[] =
  {
    <Knotenname>_reg_req_in_0,
    ...
    <Knotenname>_reg_req_in_(X-1),
  };
  static const $arch_register_req_t *_out_req_<Knotenname>[] =
  {
    <Knotenname>_reg_req_out_0,
    ...
    <Knotenname>_reg_req_out_(Y-1),
  };

  if (!op_<Knotenname>) {
```

```
    assert(0);
    return NULL;
  }

  in[0]   = op1;
  ...
  in[X-1] = opX;

  /* create node */
  res = new_ir_node(db, irg, block, op_<Knotenname>, mode, X, in);

  /* init node attributes */
  init_$arch_attributes(res, flags, _in_req_<Knotenname>, _out_req_<Knotenname>, Y);

  /* optimize node */
  res = optimize_node(res);
  irn_vrfy_irg(res, irg);

  return res;
}
```

Wenn explizit kein Konstruktor erzeugt werden soll, dann muss der Schlüssel auf `NONE` gesetzt werden:
`rd_constructor => "NONE"`
Beispiel für die Angabe eines eigenen Konstruktors:

```
"rd_constructor" =>
"  if (!op_ia32_Return) assert(0);
  return new_ir_node(db, irg, block, op_ia32_Return, mode_X, n, in);
"
```

`args` Wenn kein Standardkonstruktor verwendet wird, dann kann kann man an dieser Stelle auch andere Argumente für die Funktionssignatur des Konstruktors angeben. Die Angabe erfolg als Array von Hashes, wobei jeder Hash zwei Schlüssel enthält – `type` für den Typ des Arguments und `name` für den Variablennamen. Zu beachten ist, dass die ersten drei Parameter trotzdem automatisch als `dbg_info *db`, `ir_graph *irg` und `ir_node *block` generiert werden. D.h. alle hier angegebenen Paramter werden zusätzlich zu diesen drei erzeugt.
Beispiel passend zum obigen selbst definierten Konstruktor:

```
"args" => [
            { "type" => "int",        "name" => "n" },
            { "type" => "ir_node **", "name" => "in" }
          ]
```

`emit` Mit diesem Schlüssel läßt sich automatisch eine Ausgabefunktion erzeugen. Spezifiziert werden kann sie mittels normalem C-Code und Ausgabemakros. Die Makros sind durch einen Punkt `.` am Anfang der Zeile gekennzeichnet und werden durch eine `printf` Sequenz ersetzt. Daher ist auch Syntax der Makros an `printf` angelehnt. Abgesehen von den üblichen Formatangaben sind folgende Platzhalter definiert:

`%Sx` - Gibt das Register von Eingang `x` aus (`x` >= 1)

`%Dx` - Gibt das Register von Ausgang `x` aus (`x` >= 1)

`%Ax` - Gibt den Namen des FIRM Knotens an Eingang `x` aus (`x` >= 1)

`%M` - Gibt einen String in Abhängigkeit des Modes des Knotens aus

`%name_einer_funktion` - Es wird das Resultat der angegebenen Funktion als Parameter an `printf` übergeben. Diese Funktion bekommt den Knoten als Eingabe und muss einen String zurückliefern. Das Makro `. add %ia32_add_params` wird also in folgenden Code umgewandelt: `printf("%s", ia32_add_params(n))`

Optional kann noch eine Zahl vor dem Punkt angegeben werden, die die Einrücktiefe der `printf` Sequenzen im generierten Code entspricht. Die erzeugte Ausgabefunktion sieht folgendermaßen aus:

```
void emit_<Knotenname>(ir_node *n, emit_env_t *env) {
  FILE *F = env->out;

  Code aus "emit" mit ersetzten Makros
}
```

Beispiel einer einfachen Addition:
`"emit" => '. add %S1, %S2, %D1 \t /* Add(%A1, %A2) */'`
Komplexeres Beispiel (Min-Knoten auf `ia32`):

```
"emit" =>
'2. cmp %S1, %S2\t\t\t/* prepare Min(%A1, %A2) */
  if (mode_is_signed(get_irn_mode(n))) {
2.  cmovg %D1, %S2\t\t\t/* %S1 is greater %S2 */
  }
  else {
2.  cmova %D1, %S2, %D1\t\t\t/* %S1 is above %S2 */
  }
'
```

`attr` Wenn man zusätzlich noch Attribute an den Standardkonmstruktor übergeben will, dann kann hier eine mit Komma separierte Liste von Variablen angeben werden, die an die Argumentliste des Standardkonstruktors angefügt wird. Bsp.:
`"attr" => "int a_min, int a_max"`

`init_attr` Wenn man zusätzliche Attribute zum Standardkonstruktor hinzugefügt hat, dann wird hier die Initialisierung der Attribute durchgeführt. Es wird automatisch eine Variablendeklaration erzeugt und die Knotenattribute des neu erzeugten Knotens an diese zugewiesen:
`$arch_attr_t *attr = get_$arch_attr(res);`
Diese Variable kann zur Initialisierung verwendet werden.

Beispiel passend zur vorherigen Spezifikation zusätzlicher Attribute:
`"init_attr" => "  attr->min = a_min; attr->max = a_max;"`

`cmp_attr` An dieser Stelle kann der Quellcode zum vergleichen von Attributen zweier Knoten des gleichen Typs angegeben werden. Dies ist notwendig, da FIRM beim Erzeugen der Knoten bereits CSE (Common Subexpression Elimination) betreibt. Dazu wird der neue Knoten mit allen anderen verglichen. Wenn sich heraus stellt, dass er zu einem anderen gleich ist, dann wird der alte Knoten verwendet und kein neuer erzeugt. Zwei Knoten sind in FIRM gleich, wenn sie den gleichen opcode besitzen (also von der gleichen Art sind), die selben Vorgänger besitzen und u.a. die Attributvergleichsfunktion die Gleichheit der Attribute bestätigt. Bei Konstanten treffen die beiden ersten Eigenschaften offensichtlich immer zu, denn sie besitzen keinen Vorgänger. Daher muss die Attributvergleichsfunktion implementiert werden, da ansonsten alle Konstanten bei der Umwandlung in Backend spezifische Konstanten gleich werden würden.
Die erzeugte Funktion sieht folgendermaßen aus:

```
static int cmp_attr_<Knotenname>(ir_node *a, ir_node *b) {
  <arch>_attr_t *attr_a = get_<arch>_attr(a);
  <arch>_attr_t *attr_b = get_<arch>_attr(b);

  Code aus "cmp_attr"
}
```

Es kann also im C-Code auf die Attribute der beiden zu vergleichenden Knoten mittels `attr_a` und `attr_b` zugegriffen werden.

A.5 Generierte Dateien und Funktionen

Es gibt drei Perlskripte, die aus der Spezifikation, die Knotenkonstruktoren, die Strukturen für die Registerallokation und Ausgabefunktionen generieren.

`generate_new_opcodes.pl` - Dieses Skript erzeugt die folgenden Dateien:

`gen_<arch>_new_nodes.c.inl` - Enthält die folgenden Funktionen:
`void <arch>_create_opcodes(void)` – Initialisiert und registriert die opcodes in FIRM. Diese Funktion muss einmal bei der Initialisierung aufgerufen werden.
`ir_node *new_rd_<Knotenname>(...)` – die Konstruktorfunktion für jeden Knoten
`ir_op *get_op_<Knotenname>(void)` – Liefert `ir_op` für jeden Knoten zurück
`int is_<Knotenname>(const ir_node *n)` – Liefert `1`, wenn der übergebene Knoten vom Typ `<Knotenname>` ist und `0` sonst
`int is_<arch>_irn(const ir_node *node)` – Liefert `1`, wenn der übergebene Knoten zu den eigenen gehört und `0` sonst

`gen_<arch>_new_nodes.h` - Enthält die Prototypen für die oben genannten Funktionen.

`generate_regalloc_if.pl` - Dieses Skript erzeugt die folgenden Dateien:

`gen_<arch>_regalloc_if.c` - Enthält die Definitionen der Registeranforderungsstrukturen, die Definition der Registerklassen, sowie eine Funktion zum Initialisieren der Registerstruktur
`void <arch>_register_init(void *isa_ptr)`, die einmalig bei der Initialisierung aufgerufen werden muss.

`gen_<arch>_regalloc_if.h` - Enthält die Deklarationen obiger Strukturen und Funktionen sowie einige Definitionen für Registernamen und Klassennamen.

`gen_<arch>_regalloc_if_t.h` - Enthält Definitionen für den internen Gebrauch und sollte nie eingebunden werden.

`generate_emitter.pl` - Dieses Skript erzeugt die folgenden Dateien:

`gen_<arch>_emitter.c` - Enthält die Ausgabefunktionen für jeden Knoten (falls definiert).

`gen_<arch>_emitter.h` - Enthält die Prototypen der Ausgabefunktionen.

Literaturverzeichnis

[1] Agner Fog. *Calling Conventions for different C++ compilers and operating Systems*, May 2005. available at http://www.agner.org/assem/calling_conventions.pdf.

[2] Götz Lindenmaier, Michael Beck, Boris Boesler, and Rubino Geiß. Firm, an Intermediate Language for Compiler Research. Technical Report 2005-8, Dept. of Computer Science, University of Karlsruhe (TH), March 2005. available at http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/8.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[4] Hannes Jakschitsch. Befehlsauswahl auf SSA-Graphen. Master's thesis, IPD Goos, November 2004. available at http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf.

[5] H.-St. Jansohn and R. Landwehr. CGSS: Ein System zur automatischen Erzeugung von Code-Generatoren. Master's thesis, Institut für Informatik II, 1980.

[6] Helmut Emmelmann. *Codeselektion mit regulär gesteuerter Termersetzung*. Oldenbourg, 1994. ISBN 3-486-23252-5.

[7] Boris Boesler. A Modification to BURS in Codegeneration. Technical Report 2003-12, Universität Karlsruhe (TH), Fakultät für Informatik, 6 2003. available at http://www.info.uni-karlsruhe.de/~boesler/papers/burs-tech-report.pdf.

[8] Boris Boesler. *Codeerzeugung mit Graphersetzung und Lösungsgraphen*. PhD thesis, Universität Karlsruhe, Aachen, Februar 2005.

[9] Boris Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe (TH), IPD, June 1998. available at http://www.info.uni-karlsruhe.de/~boesler/thesis.ps.gz.

[10] Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection based on SSA-Graphs. In *SCOPES*, pages 49–65, 2003.

[11] Rubino Geiß, Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *ICGT 2006*. Springer, 2006. preliminary version to be submitted to ICGT 2006, available at http://www.info.uni-karlsruhe.de/papers/grgen_icgt2006.pdf.

[12] Enno Hofmann. Regelerzeugung zur maschinenabhängigen Codeoptimierung. Master's thesis, IPD Goos, November 2004. available at http://www.info.uni-karlsruhe.de/papers/da_hofmann.pdf.