Entwurf und Implementierung eines AMD64-Backends für LibFirm

Robin Redeker

18. Juli 2012

Zusammenfassung

In dieser Arbeit wurde ein AMD64-Backend für die LibFirm Compilersuite entworfen und implementiert. Es werden einfache C-Programme ohne Gleitkommawerte, wie zum Beispiel das 8-Damen-Problem aus der LibFirm Testsuite korrekt übersetzt. Die Performance des ausgegeben Codes liegt unter der vom GNU C Compiler auf höchster Optimierungsstufe, allerdings über der niedrigsten Optimierungsstufe. Die Performance lässt sich noch weiter steigern durch eine bessere Ausnutzung der Adressierungsmodi und einer verbesserten Registerallokationsstrategie.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Seit AMD eine Spezifikation für eine Erweiterung des x86 Instruktions-Sets mit dem Namen AMD64[4] veröffentlicht hat, haben viele Prozessorhersteller diese implementiert. Darunter AMD selbst, Intel, VIA und andere.

Durch die weite Verbreitung der AMD64-Architektur haben immer mehr Systeme die Möglichkeit von den dort vorgenommenen Verbesserungen zu profitieren. Insbesondere bietet die AMD64-Architektur einem größeren Adressraum und mehr Register. Hier bieten sich neue Möglichkeiten für die Implementierung von architekturspezifischen Optimierungen in LibFirm. Ein Backend für AMD64 bietet LibFirm zudem die Möglichkeit Optimierungen mit anderen Compilern für diese weit verbreitete Architektur zu vergleichen.

1.2 Ziel

Diese Arbeit besteht aus Entwurf und Implementierung eines rudimentären Compiler-Backends für die LibFirm-Compilersuite. Einfache C Programme, die sich auf Integer- und Steuerflussoperationen beschränken, sollen damit ausführbar sein. Ziel ist es eine Implementierung des 8-Damen Problems fehlerfrei auf der AMD64-Architektur auszuführen.

Anschließend wird mit einfachen Benchmarks die Laufzeit des Compilats in Relation zum GNU C Compiler ermittelt.

Um den Rahmen dieser Arbeit nicht zu überschreiten werden Gleitkommaoperationen nicht umgesetzt.

1.3 Struktur der Arbeit

Nach dieser Einleitung geht es weiter mit dem Kapitel 2 Grundlagen, in dem kurz auf LibFirm und die AMD64 Erweiterung eingegangen wird. Zudem wird im Abschnitt 2.3 die Struktur des LibFirm-Backends dargestellt. Das Kapitel 3 Implementierung befasst sich danach mit Themen wie das Vorgehen bei der Implementierung selbst und der Codeauswahl. Abschließend werden im Kapitel 4 Fazit die Benchmarkergebnisse vorgestellt und ein Ausblick auf weitere Implementierungsaufgaben gegeben.

Kapitel 2

Grundlagen

2.1 AMD64

Dieses Kapitel stellt die AMD64-Architektur vor. Da die AMD64-Architektur eine rückwärtskompatible 64-Bit Erweiterung der weit verbreiteten x86-Architektur ist, werden hier insbesondere die Erweiterungen vorgestellt. Die x86-Architektur wurde ihrerseits bereits durch die IA32-Architektur schon einmal auf 32-Bit erweitert. Was diese Architektur auszeichnet ist die große Verbreitung. So finden sich einem Großteil der PCs heutzutage x86 kompatible CPUs wieder. Zunehmend besitzen diese nun auch die Erweiterungen der AMD64-Architektur. Auf eine umfassende Beschreibung der x86 bzw. IA32-Architektur muss an dieser Stelle aus Umfangsgründen verzichtet werden. Für Details der IA32-Architektur sei deshalb auf [5] verwiesen.

2.1.1 Features

Hier sind die wichtigsten Features der AMD64-Architektur aufgelistet, auf die sich auch der Rest dieses Kapitels beziehen wird:

- Register Erweiterungen: Es wurden 8 zusätzliche GPRs (General Purpose Register) eingeführt, was dann insgesamt 16 GPRs ergibt. Zusätzlich sind die 16 GPRs 64-Bit breit. Dazu kommen noch 8 zusätzliche YMM/XMM Register, welche vor allem von den SSE (Streaming SIMD Extensions Instructions) Instruktionen verwendet werden.
- Einheitlicher Registerzugriff: In der x86-Architektur war es nicht möglich auf das unterste Byte aller GPRs zuzugreifen. Hier wurde die Möglichkeit nachgerüstet das untere Byte aller GPRs adressieren zu können.

- Long Mode: Mit einem neuen Betriebsmodus bietet die AMD64-Architektur bis zu 64-Bit breite virtuelle Adressen an. So wurde auch der Instruction Pointer (das RIP Register) auf 64-Bit vergrößert. Dieser neue Betriebsmodus bietet zudem einen flachen Adressraum, bei der die segmentbezogene Adressierung der x86-Architektur nicht mehr verwendet wird.

2.1.2 Betriebsmodi

Im folgenden werden die Betriebsmodi der AMD64-Architektur beschrieben. Aufbauend auf der x86-Architektur, welche ihrerseits schon mit verschiedenen Betriebsmodi kam, wurde die Architektur um einen sogenannten *Long Mode* erweitert. Der übernommene *Legacy Mode* der x86-Architektur ist unterteilt in die folgenden drei Modi:

- Protected Mode: Es werden in diesem Modus 16- und 32-Bit Programme unterstützt, auf den Speicher kann segmentiert zugegriffen werden, optionales Paging wird unterstützt, Privilegien werden geprüft und es können bis zu 4 Gb Arbeitsspeicher angesprochen werden.

- Virtual-8086 Mode: In diesem Modus können Real Mode Programme als Protected Mode Prozesse laufen.

- Real Mode: Es können 16-Bit Programme ausgeführt werden. Der Speicher wird Segmentbasiert adressiert, es gibt kein Paging oder Speicherschutz und es kann nur auf 1 MB Arbeitsspeicher zugegriffen werden.

Prozessoren, welche die AMD64-Architektur implementieren, befinden sich nach dem Booten grundsätzlich im *Legacy Mode*, und sind damit kompatibel mit der 32-Bit x86-Architektur.

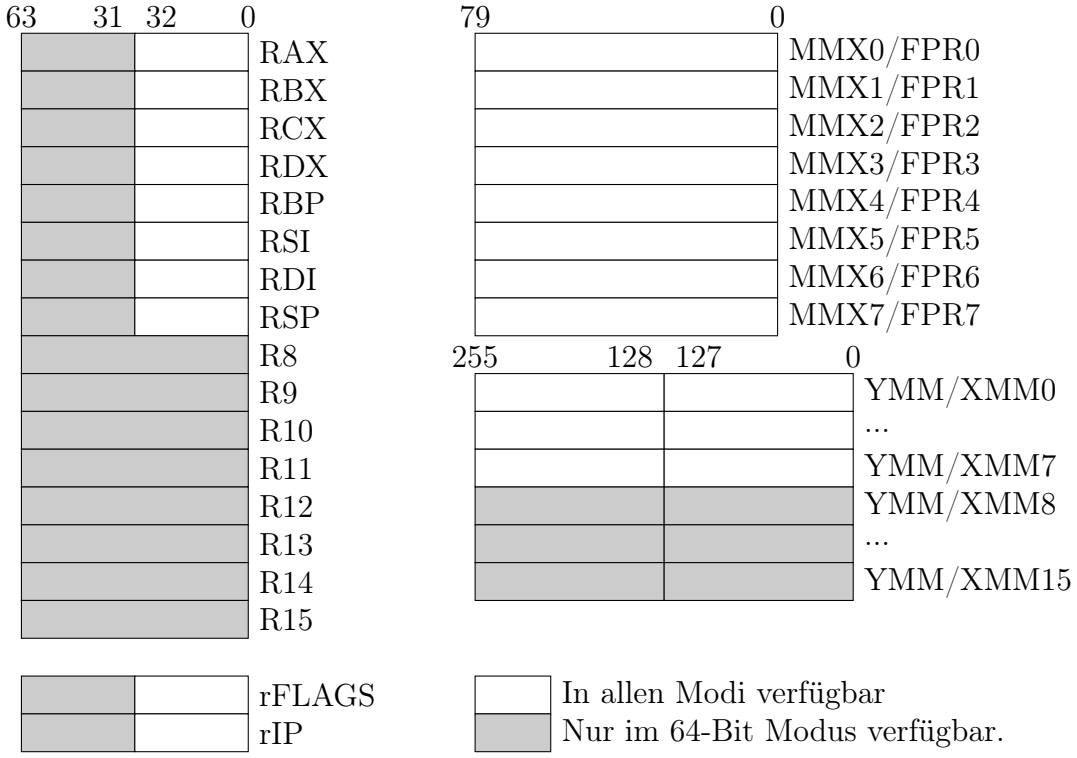
Aufbauend auf dem *Legacy Mode* wurde ein *Long Mode* für 64-Bit eingeführt. Dieser ist seinerseits in folgende Modi unterteilt:

- 64-Bit Mode: Dieser Modus bietet Zugriff auf einen virtuellen 64-Bit Adressraum. Dies erfordert allerdings auch ein 64-Bit Betriebssystem mit entsprechender Tool Chain. In diesem Modus sind alle Register 64-Bit breit und es gibt 8 zusätzliche GPRs (General Purpose Register), 8 zusätzliche YMM/XMM Register sowie eine gleichförmige Adressierung dieser Register. Mehr hierzu in Abschnitt 2.1.3 .

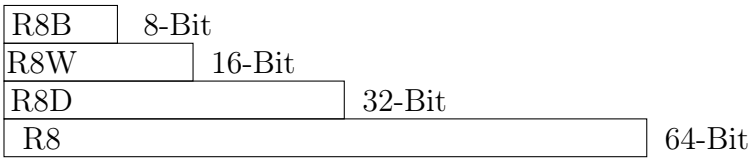
- Compatibility Mode: Dieser Modus bietet Binärkompatibilität für 16- und 32-Bit Programme. Die 16- und 32-Bit Programme können hier ohne erneutes Kompilieren ausgeführt werden.

2.1.3 Registersatz

In der folgenden Abbildung sind die Grau unterlegten Teile der Register nur im 64-Bit Modus verfügbar. Im Legacy Mode sind weiterhin nur die bisherigen x86 Register verfügbar:



Neu im 64-Bit Mode hinzu gekommen ist die einheitliche Adressierung der unteren 8-Bit, 16-Bit, 32-Bit und 64-Bit der General Purpose Register. Im folgenden Beispiel sind die 4 Adressierungs-Namen für das R8-Register veranschaulicht:



Dieses Schema gilt für die Register R8 bis R15. Bei den 32-Bit x86 Registern EAX, EBX, ECX, EDX, ESI, EDI, EBP und ESP kamen die 64-Bit Versionen hinzu, die statt mit einem E jeweils mit einem R beginnen. Zusätzlich kommt die Möglichkeit hinzu die unteren 8-Bit der 32-Bit x86 Register ESI, EDI, EBP und ESP zu Adressieren, jeweils unter dem Namen SIL, DIL, BPL und SPL.

Adressierungsmodi

Die x86-Architektur hat zwei verschiedene Arten von Pointern. *Near Pointer* und *Far Pointer*. Die *Effective Address* eines *Near Pointer* bezog sich dabei auf das aktuelle Speichersegment. Wogegen man bei einem *Far Pointer* über den *Selector* das Speichersegment angeben kann.

Neu mit dem 64-Bit Long Mode ist, dass die klassische x86 Adressierung über einen *Far Pointer* entfällt, und stattdessen der gesamte Speicherbereich als ein Datensegment behandelt wird.

Da aufgrund der durchgängigen Adressierung des gesamten Speichers durch die *Near Pointer* die Segmentierung in den Hintergrund rückt, sind im 64-Bit Modus die Segmentregister DS, ES und SS deaktiviert. CS, FS und GS sind nach wie vor nutzbar, dienen jedoch hauptsächlich als Zusatzregister für Adressberechnungen.

2.1.4 Calling Conventions

Da beim AMD64 mehr GPRs zur Verfügung stehen, wurden die Calling Conventions dementsprechend angepasst. Bei einem Aufruf werden die Argumente für eine aufgerufene Funktion erst einmal in Registern untergebracht. Wenn mehr Argumente übergeben werden müssen, wird auf den Stack zurück gegriffen.

Registerzuteilung

Die Register RBP, RBX und R12 bis R15 gehören dem Aufrufer, das bedeutet die aufgerufene Funktion darf diese Register nicht verändern, oder muss sie nach einem Aufruf speichern und beim Zurückkehren wiederherstellen. Die restlichen Register stehen allesamt der aufgerufenen Funktion zur Verfügung. Dank der zusätzlichen 8 GPRs können für die Argumentübergabe nun auch Register genutzt werden. Die AMD64 ABI sieht daher vor, dass die ersten Argumente auf bestimmte Register verteilt werden und die verbleibenden Argumente wie bei x86 auf den Stack gelegt werden. Sowohl die meisten Ganzzahldatentypen können mittels Registern übergeben werden, wie auch Gleitkommadatentypen, welche in den Vektorregistern YMM/XMM0 bis YMM/XMM7 hinterlegt werden. Die Rückgabe von Ergebnissen findet in den meisten Fällen mittels des RAX Registers (für Integers) oder des YMM/XMM0 Registers (für Gleitkomma) statt. Welche Datentypen

in welche Register gelegt werden wird in den AMD64 Calling Conventions [9] im Detail beschrieben.

Beispiel

Das folgende Beispiel soll verdeutlichen wie Argumente bei AMD64 im einfachen Fall übergeben werden. Auch wird kurz gezeigt wie der Prolog und der Epilog normalerweise aussieht. Hierbei läuft das Programm im beschriebenen *Long Mode* der AMD64-Architektur, und hat somit Zugriff auf die zusätzlichen Register. Diese werden von den AMD64 Calling Conventions für die Übergabe der ersten Argumente einer Funktion genutzt. Hiermit entfällt für die meisten Argumente die Ablage auf dem Stack bei x86.

Ein einfaches C-Programm 2.1 dient hier als Beispiel. Es werden lokale Variablen mit verschiedenen Typen an eine zweite Funktion übergeben, welche diese dann addiert und einen Gleitkommawert zurück gibt.

Listing 2.1: C Beispiel für AMD64 Calling Conventions

```
1  float test(long a, int b, int c, float x, double y) {
2      return a + b + c + x + y;
3  }
4
5  int main(int argc, char*argv[]) {
6      long   a = 10;
7      int    b = 11,
8             c = 12;
9      float  x = 1.1;
10     double y = 1.2;
11     return test(a, b, c, x, y);
12 }
```

Die Funktion 2.1 wurde mit dem GNU C Compiler (Version 4.6.2) in GNU Assembler übersetzt. Der Assemblercode 2.2 der *main*-Funktion ist mit erläuternden Kommentaren versehen.

Listing 2.2: GNU Assembler Ausschnitt für die main Funktion

```
1  # Prolog:
2  pushq %rbp              # Sichern von %rbp
3  movq  %rsp, %rbp        # Anlegen des Stack-Frames
4  subq  $48, %rsp         # Reservieren des Stack-Frames
5
6  movq  $10, -8(%rbp)     # a = 10 (64-Bit Integer)
7  movl  $11, -12(%rbp)    # b = 11 (32-Bit Integer)
8  movl  $12, -16(%rbp)    # c = 12 (32-Bit Integer)
9  movl  $0x3f8ccccd, %eax
```

```
10 movl  %eax, -20(%rbp)     # x = 1.1 (32-Bit Gleitkomma)
11 movabsq  $4608083138725491507, %rax
12 movq  %rax, -32(%rbp)     # y = 1.2 (64-Bit Gleitkomma)
13
14 # Aufruf von test:
15 movsd -32(%rbp), %xmm1    # 2. Gleitkommaargument: %xmm1 = y
16 movss -20(%rbp), %xmm0    # 1. Gleitkommaargument: %xmm0 = x
17 movl  -16(%rbp), %edx     # 3. Integerargument:    %edx = c
18 movl  -12(%rbp), %ecx     # b laden
19 movq  -8(%rbp), %rax      # a laden
20 movl  %ecx, %esi          # 2. Integerargument:    %esi = b
21 movq  %rax, %rdi          # 1. Integerargument:    %rdi = a
22 call  test
```

In den Zeilen 2-4 ist der Prolog zu sehen in dem der Stack-Frame für die lokalen Variablen aufgebaut wird. Dort werden dann in den Zeilen 6-12 die lokalen Variablen initialisiert. Ab Zeile 15 beginnt dann der Funktionsaufruf. Da die beiden Gleitkommavariablen verschieden groß sind, werden in den Zeilen 15-16 verschiedene *mov*-Befehle eingesetzt. Die *%xmmN*-Register sind mit einer Größe von 128-Bit dabei groß genug die jeweiligen Gleitkommawerte aufzunehmen. In den Zeilen 20 und 21 ist zu sehen, wie für 32-Bit Integerwerte das *%esi*-Register benutzt wird, wogegen für den 64-Bit Integerwert das *%rdi*-Register benutzt wird. Hierbei handelt es sich beim *%esi*-Register um die unteren 32-Bit des *%rsi*-Registers.

Es folgt nun der kommentierte Assemblercode für die *test*-Funktion 2.3:

Listing 2.3: GNU Assembler Ausschnitt für die test Funktion

```
1  # Prologs:
2  pushq %rbp              # Aufrufer muss %rbp sichern
3  # subq $48, %rsp        # Da test keine anderen Funktionen
4                          # aufruft muss der Stack-Frame nicht
5                          # reserviert werden!
6
7  movq  %rsp, %rbp        # Anlegen des neuen Stack-Frames
8  movq  %rdi, -8(%rbp)    # a = %rdi
9  movl  %esi, -12(%rbp)   # b = %esi
10 movl  %edx, -16(%rbp)   # c = %edx
11 movss %xmm0, -20(%rbp)  # x = %xmm0
12 movsd %xmm1, -32(%rbp)  # y = %xmm1
13
14 # ...
15 # Additions-Anweisungen weggelassen
16 # Ergebnis in: %xmm0
17
18 # Epilog:
19 # movq %rbp, %rsp  # Unnoetig, da %rsp nicht veraendert wurde
20 popq  %rbp          # Wiederherstellen von %rbp
21 ret                 # Rueckkehr, Gleitkomma-Ergebnis in %xmm0
```

Hier ist gut zu sehen, wie in der Zeile 3 die Reservierung des Stack-Frames ausgelassen

wurde. Da die *test*-Funktion keine weiteren Funktionen aufruft, muss sie den Stack-Pointer in *%rsp* nicht verändern, und spart somit jeweils im Prolog wie auch im Epilog in Zeile 19 eine Instruktion. In den Zeilen 7-12 ist zu sehen, wie die lokalen Variablen mittels der Argumente in den Register initialisiert werden. Die folgenden Berechnungen wurden in Zeile 15 weggelassen, da diese ohne Bedeutung für das Beispiel sind. Wichtig ist jedoch, dass die Berechnungen den errechneten Gleitkommawert gleich in Register *%xmm0* hinterlegen. Dieses Register ist nach den AMD64 Calling Conventions das Rückgabewert-Register für Gleitkommazahlen. Wenn die Funktion einen Integerwert zurück geben wollte, würde dafür das Register *%rax* (64-Bit Integer) oder *%eax* (32-Bit Integer) herangezogen.

2.2 LibFirm

LibFirm ist eine C Bibliothek, entwickelt für die Forschung an neuen Compiler-Optimierungen. Wie viele andere moderne Compiler verwendet LibFirm eine SSA (Static Single Assignment) basierte Zwischensprache. Diese Zwischensprache mit dem Namen Firm hat eine Graphform, in welcher Datenfluß- und Steuerflussabhängigkeiten dargestellt sind.

Nach einem Abschnitt über die SSA-Form wird die Zwischensprache Firm näher angesehen. Zum Schluss wird dann näher auf die softwaretechnische Struktur und die Phasen innerhalb des LibFirm-Backends eingegangen..

2.2.1 SSA-Form - Static Single Assignment

Die SSA-Form ist eine 1988 von Rosen, Wegman und Zadeck [10] veröffentlichte Programmrepräsentation. Moderne Compiler wie GCC (Gnu Compiler Collection), LLVM und LibFirm verwenden die SSA-Form heutzutage für vielfältige Optimierungen.

Zur Erläuterung des zentralen Konzepts hinter der SSA-Form ist im Listing 2.4 eine C Funktion gegeben, welche sich in ihrer ursprünglichen Form befindet. Variablen wie y werden hier häufiger neue Werte zugewiesen.

Listing 2.4: Eine C-Funktion, welche sich nicht in SSA-Form befindet.

```
1  int foo (int a, int b) {
2      int x = a * b;
3      int y = 0;
4
5      if (a > 0) {
6          y = x - a;
7          y = y * 2;
8      } else {
9          y = x - b;
10     }
11
12     int res = y;
13
14     printf ("Calculated %d\n", res);
15     return res;
16 }
```

In der SSA-Form wird nun für jede Zuweisung **statisch** ein neuer Variablenname erzeugt. Das heißt, dass in der SSA-Form jeder Variable nur einmal ein Wert zugewiesen wird. Dies sei exemplarisch im Listing 2.5 dargestellt. Somit kann anhand des Variablennamens das Statement, welches den Wert in dieser Variablen berechnet hat, eindeutig identifiziert werden. Hierbei sei angemerkt, dass die Namen **statisch** vergeben sind, dies bedeutet allerdings, dass zur Laufzeit des Programms, zum Beispiel in Schleifen, durchaus

häufiger Zuweisungen stattfinden können.

Listing 2.5: Die C Funktion in SSA-Form.

```
1  int foo (int a_0, int b_0) {
2      int x_0 = a_0 * b_0;
3      int y_0 = 0;
4
5      if (a_0 > 0) {
6          y_1 = x_0 - a_0;
7          y_2 = y_1 * 2;
8      } else {
9          y_3 = x_0 - b;
10     }
11
12     int res_0 = Φ(y_2, y_3);
13
14     printf ("Calculated %d\n", res_0)
15     return res_0
16 }
```

Die if-Schleife in Zeile 5 im Listing 2.4 hat den Effekt, dass y von zwei verschiedenen Grundblöcken berechnet wird. Da stellt sich die Frage, was man nach der Transformation in die SSA-Form an *res* in Zeile 12 zuweisen soll. Hier muss *res* abhängig vom vorhergehend durchlaufenen Grundblock der jeweils richtige Wert zugewiesen werden. Hierzu führt die SSA-Form die Φ-Funktion ein. Die Φ-Funktion hat die Aufgabe abhängig vom Steuerfluss, bzw. vom vorher durchlaufenen Grundblock, die richtige Variable auszuwählen. Dies ist in Zeile 12 im Listing 2.5 zu sehen. Hier gibt Φ abhängig vom ausgeführten Grundblock die entsprechend berechnete Variable an die Zuweisung weiter. Für die Umsetzung der Funktionalität der Φ-Funktion ist der Codegenerator zuständig. Er muss entsprechenden Code erzeugen, welcher die Weitergabe der zuvor berechneten Variablen regelt.

Für die Beschreibung der Steuerflussabhängigkeiten setzen Compiler häufig einen Steuerflussabhängigkeits-Graphen ein. Allerdings ist die Berechnung der SSA-Form und des Graphen sehr aufwändig, und auch die resultierenden Datenstrukturen sind platzintensiv. Hier zu gibt es unter anderen sowohl in [6] als auch in [1] Algorithmen, welche die SSA-Form und den Graphen mit praxistauglichem Aufwand berechnen. Letztere Arbeit bezieht sich hierbei direkt auf die Zwischendarstellung Firm, welche im nächsten Abschnitt näher eingeführt wird.

Es sei noch angemerkt, dass die SSA-Form und der Steuerflussabhängigkeits-Graph nicht für ein ganzes Progamm berechnet wird, sondern üblicherweise jeweils pro Funktion. Es gibt jedoch in der aktuellen Forschung schon Ansätze die SSA-Form auf mehrere Funktionen auszudehnen, hierzu sei auf [11] verwiesen.

2.2.2 Zwischencode Repräsentation

Firm ist eine am Lehrstuhl für Programmierparadigmen des Karlsruhe Instituts für Technologie entwickelte Zwischendarstellung für Compiler. Sie wird im technischen Report von Braun, Buchwald und Zwinkau [1], sowie in dem Tutorial von Lindenmaier [8] vorgestellt und beschrieben. Firm ist eine graphenbasierte Zwischendarstellung des Programms, bei der sich der Zwischencode in SSA-Form befindet.

LibFirm arbeitet wärend des gesamten Übersetzungsvorgangs mit dieser Zwischendarstellung. Erst die letzte Phase des jeweiligen architekturabhängigen Backends erzeugt linearen Assembler code.

Ein Beispiel für einen Firm Graphen ist in Abbildung 2.1 zu finden. Für das Beispiel wurde der Code aus Listing 2.4 mit dem zum LibFirm gehörenden C Übersetzer *cparser* übersetzt und der Graph vor Eintritt in das Backend ausgegeben. Zur Visualisierung wurde das *yComp*-Tool eingesetzt.

Bei LibFirm werden die Funktionen jeweils einzeln in einen Graphen mit Knoten, welche für (Grund)Blöcke, Operationen und Werte stehen, übersetzt. Jeder dieser Graphen enthält dabei einen *Start Block*, einen *End Block* und eine variable Anzahl an anderen (Grund)Blöcken. Werte und Operationen werden über Verweise auf den Block-Knoten den Blöcken zugeordnet. Diese werden auch grafisch in *yComp* direkt innerhalb diser Blöcke dargestellt. Knoten (und damit auch Blöcke) sind mit gerichteten Abhängigkeitskanten verbunden, wobei die Farbe der Kante von der Art der Abhängigkeit abhängt. Blaue Kanten stehen für Speicherabhängigkeiten, schwarze Kanten für Datenabhängigkeiten und die roten Kanten für Steuerabhängigkeiten. Die roten Kanten dienen auch dazu die benötigten Steuerflussabhängigkeiten zwischen den Grundblöcken zu repräsentieren. Zwischen den Blöcken gibt es, außer den roten Kanten, keine sichtbaren Kanten. Datenabhängigkeiten zwischen den Blöcken werden in *yComp* nur über einen grünen Kreis angedeutet und nur bei Bedarf dargestellt. In den Abbildungen zu den Übersetzungsphasen sind daher diese Kanten nicht sichtbar.

Knoten repräsentieren in Firm eine Operation die einen Wert zurück gibt. Wenn mehrere Werte zurückgegeben werden müssen, dann werden diese in Tupel zusammengefasst. Um aus diesen wieder einzelne Daten zu selektieren gibt es sogenannte *Proj*-Knoten. Firm stellt verschiedenste Knoten-Klassen bereit, wie zum Beispiel *Add* zum addieren oder *Cmp* zum vergleichen von Werten. Mit diesen Knoten-Klassen lassen sich problemlos C Programme darstellen. Es sei hier angemerkt, dass der Firm Graph ersteinmal nur recht abstrakte Abhängigkeiten darstellt und die ursprüngliche Reihenfolge der Anweisungen im C Programm nur über die Abhängigkeiten festgehalten werden können. Da erst eine spätere Phase im Backend eine lineare Ordnung den Grundblöcken wieder herstellt,

kann es sein, dass der Code für die einzelnen Anweisungen dann in anderer Reihenfolge ausgegeben wird. Eine korrektes Ergebnis wird hierbei über die Abhängigkeiten sicher gestellt.

Für weitere Details möchte ich nocheinmal auf den Report von Braun, Buchwald und Zwinkau [1] verweisen, welcher neben einer detailierteren Beschreibung von Firm auch einen Algorithmus zum Aufbau des Firm Graphen bereit stellt.

2.3 Struktur des LibFirm-Backends

Im folgenden Abschnitt wird nun das LibFirm-Backend näher beschrieben. Zuerst soll auf die verschiedenen Übersetzungs-Phasen eingegangen werden, gefolgt von einer Beschreibung der Softwarearchitektur des Backends.

2.3.1 Übersetzungs-Phasen

Das Backend überführt in mehreren Phasen den architekturunabhängigen Graphen in architekturabhängigen Assemblercode. Diese Phasen stehen hierbei abstrakt für die Erledigung einer bestimmten Aufgabe im Backend und können dabei mehrere Pässe über den Firm Graphen beinhalten. Diese Pässe sammeln die notwendigen Informationen und führen die jeweils notwendigen Transformationen durch. Herausheben möchte ich hier nur die wichtigsten Phasen des Backends. Dies sind die Phasen *Codeauswahl*, *Befehlsanordnung*, *Registerallokation* und die *Codeausgabe*. Nicht weiter betrachtet werden Verifizierungs-Phasen und kleinere oft architekturabhängige Transformationen, wie zum Beispiel das Ersetzen von Gleitkommaarithmetik mit Integerarithmetik für Architekturen die kein Gleitkomma bereit stellen. Die Verifizierungs-Phasen stellen sicher, dass bestimmte softwaretechnisch relevante Invarianten eingehalten werden und dienen zur frühen Fehlererkennung bei der Softwareentwicklung von LibFirm.

Als anschauliches Beispiel für die Veränderungen des Graphen nach jeder Phase soll die Abbildung 2.1 dienen, welche das Beispiel aus Listing 2.4 als Firm-Graphen darstellt.

Codeauswahl

Die Codeauswahl-Phase übersetzt die Firm Knoten-Klassen ausgehend von ihren abstrakten Operationen in architekturspezifische Knoten. Manche Knoten-Klassen bleiben erhalten, wie zum Beispiel *Proj* Knoten, und haben nach wie vor die selbe Semantik. Andere Knoten, wie zum Beispiel der *Cmp* Knoten aus dem Graphen in Abbildung 2.1 wird transformiert in einen *amd64_ Cmp* Knoten, zu sehen in Abbildung 2.2. Vor dieser Phase

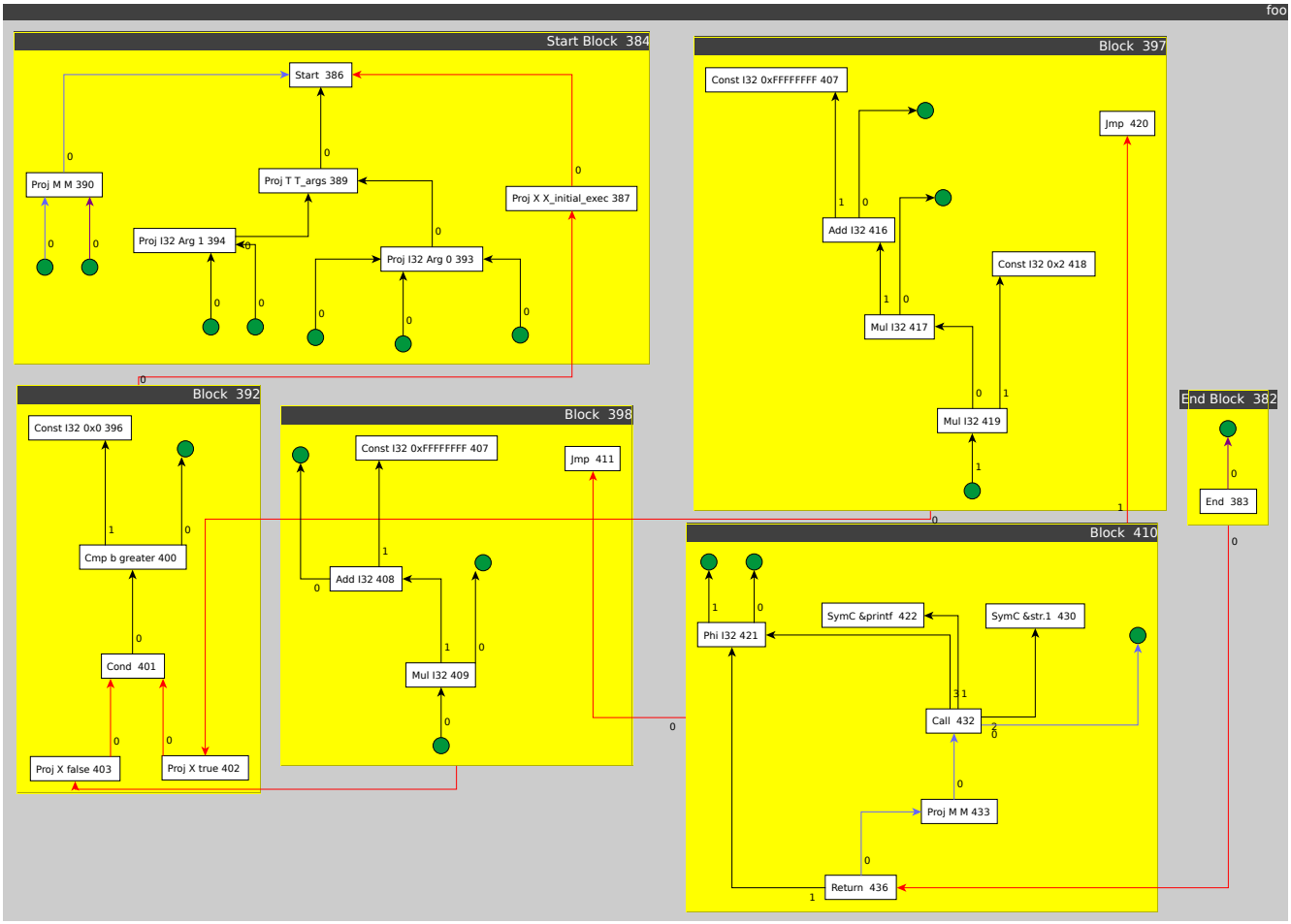


Abbildung 2.1: Das Beispiel aus Listing 2.4 als Firm Graphen.

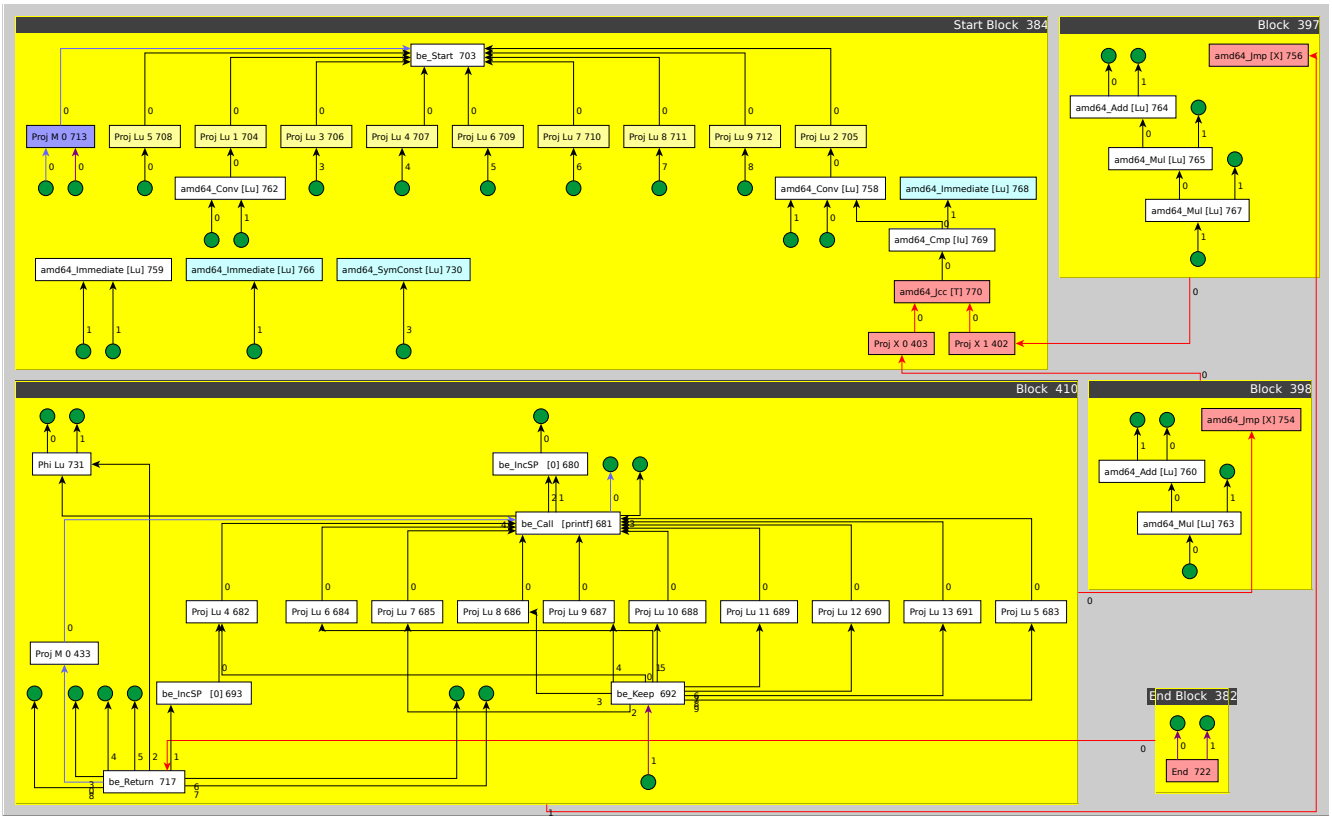


Abbildung 2.2: Der Graph aus Abbildung 2.1 nach der Codeauswahl.

sind vom Middleend auch noch andere Transformationen und Optimierungen vorgenommen worden, so kann es sein, dass Knoten von einem Block in einen anderen verschoben wurden, oder dass ganze Blöcke entfallen.

Befehlsanordnung

In dieser Phase werden die Knoten unter Berücksichtigung ihrer Abhängigkeiten untereinander wieder in eine lineare Ordnung gebracht. Dieser lineare Ablauf wurde mit pinken Kanten in Abbildung 2.3 visualisiert und legt die Ausführungsreihenfolge der Instruktionen fest. Diese Information ist insbesondere wichtig für die nachfolgenden Phasen wie die *Registerallokation* und natürlich die *Codeausgabe*.

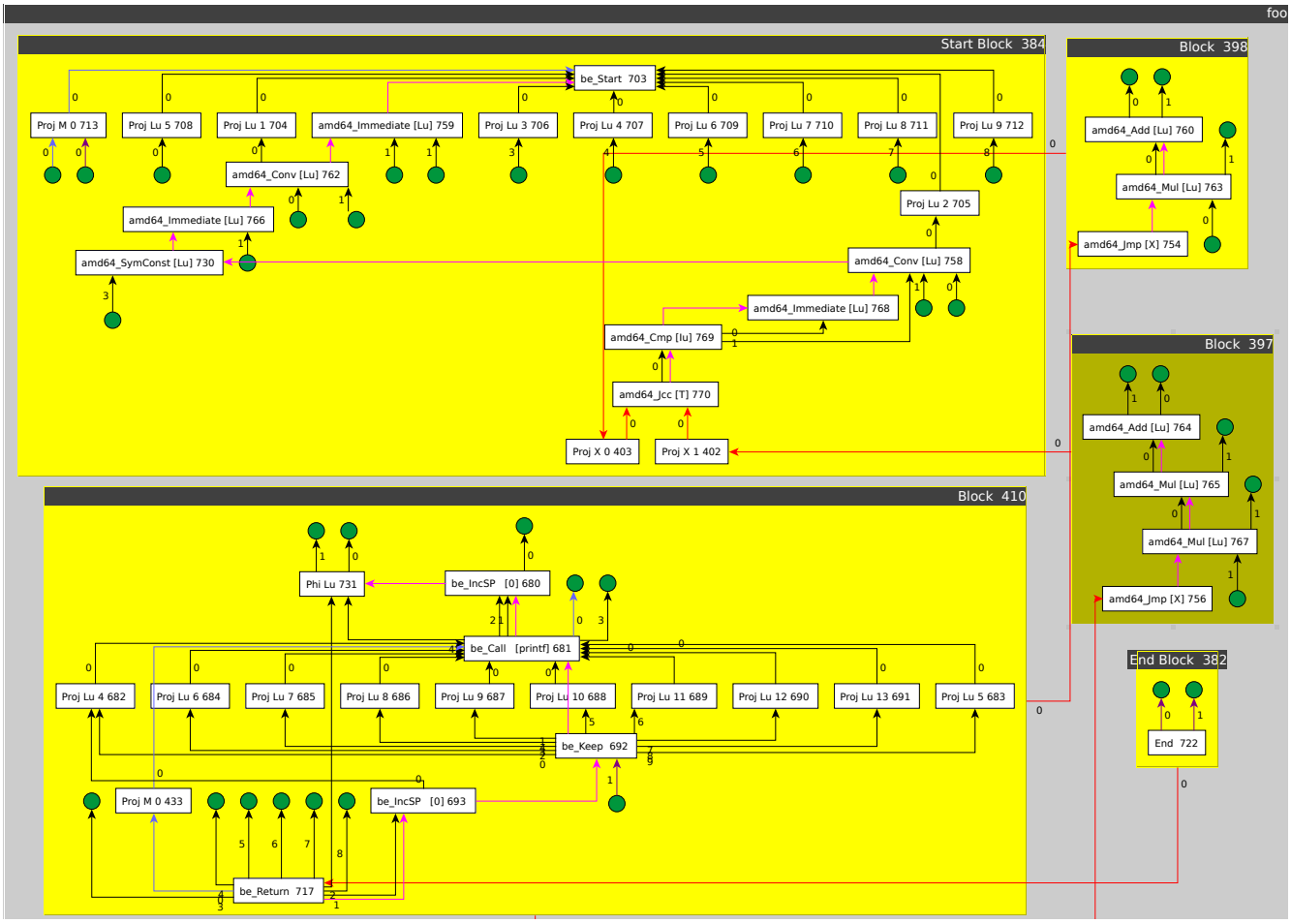


Abbildung 2.3: Der Graph aus Abbildung 2.2 nach der Befehlsanordnung.

Registerallokation

Nachdem die Ausführungsreihenfolge der Instruktionen festgelegt ist folgt die Registerallokations-Phase. In dieser werden die von den Daten und Instruktionen zu benutzenden Register der Ziel-Architektur vergeben. Hierbei muss auf vielfältige architekturspezifische Sonderfälle geachtet werden. Manche Instruktionen benötigen bestimmte Register für die Übergabe und Rückgabe von Daten, wie die Divisions-Instruktionen *div* oder *mod* bei AMD64.

Die Registerallokation ist hierbei in zwei Phasen aufgeteilt, in der ersten wird auf Grund des errechneten Registerdrucks sogenannter *Spillingcode* erzeugt. Dieser *Spillingcode* sorgt hierbei dafür, dass Werte die nicht in die verfügbaren Register passen vorübergehend in den Speicher geschrieben werden. Der erzeugte *Spillingcode* führt hierbei zu einer Transformation des Graphen.

In der darauf folgenden Phase werden die Register den jeweiligen Werten und Operationen zugeteilt. Hierfür werden die Register für die *Codeausgabe* in den Knotenattributen vermerkt.

Für detailierte Informationen über die Registerallokation für Code in SSA Form sei hierbei auf [2] und [7] verwiesen.

Codeausgabe

Die letzte Phase im Backend nutzt die von der Befehlsanordnungs- und Registerallokations-Phase geleistete Arbeit und setzt die Instruktions-Knoten um in Assembler für die jeweilige Zielarchitektur. Im Listing 2.6 ist die AMD64 GNU Assemblerausgabe des um die Registerzuteilunginformation erweiterten Graphen aus der Abbildung 2.3 zu sehen. In den Kommentaren hinter den Assembleranweisungen vermerkt LibFirm den Knoten aus dem diese Instruktion hervorging. Es ist ersichtlich, wie zum Beispiel in Zeile 5, dass ein Instruktions-Knoten mehrere Assembleranweisungen ergeben kann. Anhand der Sprungmarken in den Zeilen 2, 17, 22 und 26 können die verschiedenen Blöcke aus der Abbildung 2.3 identifiziert werden.

In dieser Phase kann sehr aus der in der Befehlsanordnungs-Phase vorgenommenen Ordnung profitiert werden, wenn wie in Zeile 16 Sprunganweisungen weggelassen werden können.

Listing 2.6: Von der Codeausgabe des AMD64-Backends erzeugter Assembler für die Funktion in Listing 2.4.

```
1  foo:
2  .L384: sub $8, %rsp                  /* be_Start T[703:5]  */
3         mov %r14, (%rsp)              /* amd64_Store M[862:135]  */
4         mov $-1, %rax                 /* amd64_Immediate Lu[759:41]  */
5         shl $32, %rsi                 /* amd64_Conv Lu[762:44]  */
6         sar $32, %rsi                 /* amd64_Conv Lu[762:44]  */
7         mov %rsi, %rdx                /* amd64_Conv Lu[762:44]  */
8         mov $2, %rcx                  /* amd64_Immediate Lu[766:48] */
9         mov $.Lstr.1, %r8             /* amd64_SymConst Lu[730:12] */
10        shl $32, %rdi                 /* amd64_Conv Lu[758:40]  */
11        sar $32, %rdi                 /* amd64_Conv Lu[758:40]  */
12        mov %rdi, %rsi                /* amd64_Conv Lu[758:40]  */
13        mov $0, %rdi                  /* amd64_Immediate Lu[768:50] */
14        cmp %rdi, %rsi                /* amd64_Cmp Iu[769:51] */
15        jle .L398                     /* Proj X[403:53] */
16        /* fallthrough to .L397 */    /* Proj X[402:54] */
17 .L397: add %rdx, %rax                /* amd64_Add Lu[764:46] */
18        mov %r8, %rdi                 /* be_Copy Lu[851:131]  */
19        mul %rsi                      /* amd64_Mul Lu[765:47] */
20        mul %rcx                      /* amd64_Mul Lu[767:49]  */
21        jmp .L410                     /* amd64_Jmp X[756:38]  */
22 .L398: add %rsi, %rax                /* amd64_Add Lu[760:42] */
23        mov %r8, %rdi                 /* be_Copy Lu[852:132]  */
24        mul %rdx                      /* amd64_Mul Lu[763:45] */
25        /* fallthrough to .L410 */    /* amd64_Jmp X[754:36]  */
26 .L410: mov %rax, %rsi                /* be_Copy Lu[774:55]  */
27        mov %rax, %r14                /* be_Copy Lu[853:133]  */
28        xor %rax, %rax                /* be_Call T[681:14] */
29        call printf                   /* be_Call T[681:14] */
30        mov %r14, %rax                /* be_Copy Lu[854:134]  */
31        mov (%rsp), %r14              /* amd64_Load T[863:136]  */
32        add $8, %rsp                  /* be_Return X[717:24] */
33        ret                           /* be_Return X[717:24] */
```

2.3.2 Backend Modularisierung

Für die Codeausgabe unterstützt LibFirm verschiedene CPU-Architekturen. Darunter befinden sich zum Zeitpunkt der Anfertigung dieser Arbeit ein Backend für die SPARC-, ARM-, AMD64- und IA32-Architektur. Bei dem IA32-Backend wurde insbesondere auch viel Entwicklungsarbeit in die Optimierung der Codeausgabe gesteckt. So werden die

sehr architekturspezifischen Adressierungsmodi der IA32-Architektur unterstützt und für Adressberechnungen und direkten Zugriff auf Operanden benutzt.

Das Backend von LibFirm ist für die Unterstützung von mehreren verschiedenen Architekturen in einen generischen und einen spezifischen Teil unterteilt. Den Code für das generische Backend ist im LibFirm Projekt Repository im Unterverzeichnis *libfirm/ir/be* zu finden. Dort sind generische Funktionalitäten untergebracht wie das Durchlaufen des Firm-Graphen, Support-Routinen für die Assemblerausgabe, Registerallokations-Algorithmen mit Spillingcode-Erzeugung, Code-Anordnung (Scheduling) und architekturunabhängige Optimierungen.

Desweiteren befinden sich in dem Unterverzeichnis auch die architekturspezifischen Backend-Implementierungen. So ist z.b. IA32 in *libfirm/ir/be/ia32* zu finden. Das AMD64-Backend befindet sich in *libfirm/ir/be/amd64*. Die Backend-Implementierungen sind auf mehrere Dateien verteilt, die unter den Backends einheitlich benannt sind. Für neue Backend-Implementierungen existiert auch ein sogenanntes *TEMPLATE* verzeichnis, welches bereits Prototypen für die Dateien mitliefert. Im folgenden seien die Dateien prototypisch anhand des AMD64-Backends erläutert:

- bearch_amd64.[ch]
  Die Hauptdatei für ein Backend. Hier wird das generische Backend aus dem Überverzeichnis initialisiert und architekturspezifische Konstanten gesetzt. Auch wird hier ein Teil der Calling Conventions implementiert, wie zum Beispiel die Vergabe von Registern für Funktionsargumente.

- amd64_transform.[ch]
  Dieser Teil des Backends ersetzt Firm Knoten teilweise durch architekturspezifische Knoten. Dies kann sowohl eine 1:1 Umsetzung sein, als auch eine N:M, wenn Firm-Operationen nicht direkt umgesetzt werden können. Zum Beispiel im IA32-Backend werden viele Transformationen bereits unter Beachtung der optimalen Nutzung der Adressierungsmodi vorgenommen.

- amd64_emitter.[ch]
  Der in *amd64_ transform.[ch]* erzeugte Graph mit den architekturspezifischen Knoten wird hier übersetzt in Assembler. Dieser Teil kümmert sich unter anderem auch darum, dass die generischen Block-Scheduling Routinen aufgerufen werden, welche wieder eine lineare Ausführungsstruktur in den Firm-Graphen bringt.

- amd64_new_nodes.[ch], amd64_nodes_attr.h In diesen Dateien werden neue Knoten für die architekturspezifische Version des Firm-Graphen definiert, welche sich

nicht mit dem *amd64_spec.pl* Script automatisch generieren lassen. Auch können hier weitere Attribute für Knoten definiert werden, um Informationen aus früheren Transformations-Phasen an spätere durch zu reichen.

- amd64_spec.pl Dieses in Perl verfasste Script, welches mit Hilfe von weiteren Perl Scripts im generischen Backend ausgewertet wird, stellt eine abstrakte Beschreibung der Architektur dar. Es kann hier unter anderem beschrieben werden, welche Registerklassen und Register eine Architektur bietet und was für architekturspezifische Knoten es gibt. Einzelne Knoten-Beschreibungen können hier auch angeben, welche Eingabe- und Ausgabe-Register ein Knoten benötigt. Auch können hier Assembler-Templates angegeben werden, welche eine explizite Implementierung in *amd64_emitter.[ch]* unnötig machen.

  Dieses Script generiert beim Compilieren von LibFirm folgende Dateien:

  - gen_amd64_emitter.[ch] Enthält die aus den Assembler-Templates erzeugten Assemblerausgabe-Routinen für die in *amd64_spec.pl* definierten Knoten.
  - gen_amd64_new_nodes.c.inl/.h Enthält Definitionen und Konstruktoren für die in *amd64_spec.pl* definierten Knoten.
  - gen_amd64_regalloc_if.[ch] Enthält die Registerinformationen aus *amd64_spec.pl* in Form von Datenstrukturen, welche an den generischen Registerallokator im generischen Backend weitergegeben werden.

Kapitel 3

Implementierung

Im folgenden wird auf die Implementierung des AMD64 Compiler-Backends für LibFirm eingegangen. Hierbei werde ich zuerst auf die vorhanden Mittel eingehen und die Vorgehensweise erläutern. Im darauf folgenden Abschnitt wird die Codeauswahl des AMD64 Compiler-Backends näher erläutert. Abschließend wird auf besondere Problematiken, die wärend des Implementierungsprozesses auftraten eingegangen.

3.1 Ausgangsbasis

Hier wird auf vorhandenes Material eingegangen, sowie auf die Zielsetzung unter welcher das AMD64-Backend entwickelt wurde.

3.1.1 Ziel

Um den Rahmen der Arbeit einzuhalten wurde als Ziel die Ausführung des 8-Damen-Problems aus der LibFirm Testsuite gewählt. Hierzu muss das Backend den vom cparser erzeugten Firm Graphen in passenden AMD64-Assembler umsetzen. Für die Ausführung des 8-Damen-Problems werden nur die Integer- und Kontrollfluss-Operationen von C benötigt. Dies ermöglichte das Auslassen von Gleitkommaoperationen. Auch wurde nicht gefordert, dass C Strukturen direkt an Funktionen übergeben werden können. Damit konnte ein Teil der komplexität der Calling Conventions weggelassen werden.

3.1.2 Vorhandenes

Für die Umsetzung des AMD64-Backends wurde als Referenz für die AMD64 Architektur das von AMD bereitgestellte Handbuch [3] und [4] verwendet. Für die Implementierung

der Calling Conventions wurde die System V ABI Erweiterung für AMD64 verwendet [9].

Desweiteren konnte die Assemblerausgabe des vorhandenen GNU C Compilers verwendet werden um die aus den Referenzen gewonnenen Informationen und Annahmen zu überprüfen. Dieser wurde auch zum Testen und Vergleichen des vom LibFirm AMD64-Backends erzeugten Programmcodes verwendet.

LibFirm besaß bei Aufnahme dieser Arbeit bereits Compiler-Backends für ARM, Sparc und IA32. Hier konnte jederzeit auf Code-Beispiele für die interne LibFirm-Backend API zurückgegriffen werden, was die Einarbeitung und Umsetzung des AMD64-Backends erleichterte. Die nahe Verwandschaft mit der IA32-Architektur bot zusätzliche Referenzpunkte bei der Implementierung.

3.1.3 Vorgehen

Die Abstammung von der IA32-Architektur legte nahe bei dieser Arbeit vom IA32 Backend von LibFirm auszugehen. Softwaretechnisch wäre es am sinnvollsten gewesen gemeinsame Funktionalität in ein x86-Backend auszulagern und diese sowohl für das IA32- als auch das AMD64-Backend zu nutzen. Profitiert hätte hiervon die softwarearchitektonische Gestaltung des IA32-Backends wie auch zukünftige auf der x86-Architektur aufbauende Architekturen, wie eben AMD64. Eine gemeinsame Logik für die komplexen Adressierungsmodi der IA32-Architektur hätte für das AMD64-Backend genutzt werden können.

Allerdings hätte ein solches Projekt einen wesentlich größeren Arbeitsaufwand mit sich gezogen und hätte den Rahmen dieser Studienarbeit überschritten. Ein Kopieren des IA32-Backends und Umschreiben wurde auch in Betracht gezogen, jedoch hätten die bereits vorhandenen Optimierungen und Adressmodi-Operationen nachvollzogen werden müssen. Da jedoch zu Anfang dieser Arbeit beim Author noch nicht genügend Erfahrung im Umgang mit der IA32-Architektur und entsprechenden Compiler-Backends vorhanden war, wurde ein anderer Weg gewählt.

Zur Umsetzung des AMD64-Backends wurde das vorhandene Template-Backend benutzt und nach und nach erweitert. Hierdurch war eine Konzentration auf Einzelaspekte der Implementierung möglich und es konnte wärend der Implementierung nach und nach Erfahrung mit der LibFirm-Backend API gewonnen werden.

3.2 Codeauswahl

In diesem Abschnitt wird auf die Übersetzung von Firm Knoten in AMD64-Knoten und den späteren Assembler betrachtet. Da der erstere Vorgang, die Transformation der Firm

Knoten in AMD64-Knoten, eigentlich eine von der Coderausgabe unabhängige Phase ist, würde sich eine getrennte Betrachtung anbieten. Da allerdings die Transformations-Phase im AMD64-Backend relativ trivial ausfällt, verglichen mit anderen Backends, wird hier sowohl die Transformation als auch die letztendliche Codeausgabe in einem Schritt betrachtet.

Da eine umfassende Betrachtung der Übersetzungen aller Firm-Operationen den Rahmen dieser Arbeit überschreiten würde, werden in den folgenden Abschnitten nur wenige Operationen exemplarisch betrachtet.

3.2.1 Arithmetik

Dieser Abschnitt greift beispielhaft die Arithmetik-Operationen Add, Sub, Or, Eor und And heraus und deren Übersetzung darstellen. Für die Übersetzung dieser Firm Knoten wurden in *amd64_spec.pl* die AMD64 Spezifischen Knoten mit gleichem Namen erzeugt. Die Transformation bestand aus einer 1:1 Ersetzung der Firm Knoten durch die gleichnamigen AMD64-Knoten. Weitere Informationen mussten nicht an die Knoten gehängt werden.

Zu erwähnen gilt hier die Behandlung der Operanden vor der eigentlichen Berechnung. AMD64 erfordert, wie auch IA32, dass die Ausgabe des Ergebnisses dieser arithmetischen Operationen in das Register des zweiten Operanden erfolgt. Da jedoch der Registerallokator hierfür keine spezielle Logik bereit hielt, musste sich die Codeausgabe des AMD64-Backends damit befassen.

Hier sei noch angemerkt, dass der erzeugte Assemblercode nur 64-Bit Arithmetik durchführt und hier inkompatibel mit einigen Randbedingungen der C-Arithmetik ist. Dies ist insbesondere für den nächsten Abschnitt von bedeutung, wenn die Load- und Store-Operationen betrachtet werden.

Exemplarisch seien die Operanden der jeweiligen Operationen in den Registern *%r13* und *%r14* gespeichert. Die Ausgage des Ergebnisses wurde vom Registerallokator für das Register *%rcx* vorgesehen. Die Codeausgabe hat diese Knoten dann wie folgt übersetzt:

And	`mov %r13, %rcx` `and %r14, %rcx`
Or	`mov %r13, %rcx` `or %r14, %rcx`
Eor	`mov %r13, %rcx` `xor %r14, %rcx`
Add	`mov %r13, %rcx` `add %r14, %rcx`

Sub	`mov %r13, %rcx`
	`neg %r14`
	`add %r14, %rcx`
	`neg %r14`

Bei der *Sub* Operation sei angemerkt, dass hier statt einer Subtraktions-Operation, welche die AMD64 Plattform bietet, eine Addition mit dem Negativen wert des zweiten Operanden vorgenommen wurde. Diese Vereinfachung wurde vorgenommen, da die Implementierung der Subtraktions-Operation eine kompliziertere Behandlung der Operanden notwendig gemacht hätte. Für den Fall, dass das Ausgaberegister das selbe ist wie einer der beiden Operanden, wäre hier ein temporäres Register für die Vertauschung notwendig gewesen, welches der Allokator jedoch nicht zugewiesen hat. An diser Stelle wäre noch Potential für Optimierungen.

3.2.2 Load & Store

Hier wird die Übersetzung der Load- und Store-Operationen näher betrachtet. Diese wurden als Beispiel herangezogen, da hier der einheitliche Zugriff auf Register deutlich wird, den die AMD64-Architektur bietet. So kann ein Zugriff auf die unteren 8, 16, 32 oder 64 Bit von jedem der 16 GPRs (General Purpose Register) erfolgen. Um dies auszunutzen speichert das AMD64-Backend in der Transformationsphase die Größe des Zugriffs, bzw. die Größe des Datums das gespeichert oder geladen werden soll. Die Codeausgabe selektiert bei der Ausgabe des jeweiligen Load oder Store Knotens dann die Registersuffixe abhängig von der vorher gespeicherten Größe des Datums.

In den folgenden Beispielen wird angenommen, dass das zu speichernde Datum im Register *%r14* liegt oder, dass das zu ladende Datum in das Register *%r14* geladen werden soll. Die Speicheradresse auf welche die Load- bzw. Store-Operation zugreift liegt dabei im Register *%rax*.

	Unsigned	Signed
8-Bit Store	`mov %r14b, (%rax)`	(siehe Unsigned)
16-Bit Store	`mov %r14w, (%rax)`	(siehe Unsigned)
32-Bit Store	`mov %r14d, (%rax)`	(siehe Unsigned)
64-Bit Store	`mov %r14, (%rax)`	(siehe Unsigned)

Bei der Load-Operation gibt es noch einen Sonderfall zu beachten. Da der ausgegebene Assemblercode nur 64-Bit Arithmetik beherrscht, müssen Daten mit Vorzeichen nach dem Ausführen von 8-Bit, 16-Bit und 32-Bit Load-Operationen mit einer Vorzeichenerweiterung in das Zielregister geschrieben werden. Hierfür bietet die AMD64-Architektur die *movsx*-Operation an.

	Unsigned	Signed
8-Bit Load	`mov (%rax), %r14b`	`mov (%rax), %r14b` `movsx %r14b, %r14`
16-Bit Load	`mov (%rax), %r14w`	`mov (%rax), %r14w` `movsx %r14w, %r14`
32-Bit Load	`mov (%rax), %r14d`	`mov (%rax), %r14d` `movsxd %r14d, %r14`
64-Bit Load	`mov (%rax), %r14`	(siehe Unsigned)

3.3 Probleme

Dieser Abschnitt behandelt die bei der Implementierung aufgetretenen Probleme. Die größte Schwierigkeit bei der Umsetzung des AMD64-Backends war die Einarbeitung in die LibFirm-Backend API. Dies lag hauptsächlich an der nicht vorhandenen Dokumentation über die interne Architektur des generischen LibFirm Backends. Durch Einarbeitung in die Backends anderer Architekturen und den generischen Backendcode selbst konnte man sich mit der Implementierung eines neuen Backends allerdings gut vertraut machen.

Desweiteren traten diverse Probleme wärend der Implementierung auf, welche teilweise geringfügige Eingriffe in das generische Backend notwendig machten. Auf diese will ich in den nächsten zwei Abschnitten näher eingehen.

3.3.1 Switch Jump Table Codegeneration

Die Firm-Knoten mit der Klasse Switch werden mit Hilfe des generischen Backends typischerweise übersetzt in eine Sprungtabelle. Das bedeutet, dass für die verschiedenen durchnummerierten Fälle einer Switch-Anweisung eine Tabelle existiert, über welche die

Fall-Nummer auf eine Sprungadresse abgebildet wird. Hierbei ging das generische Backend davon aus, dass die Sprungadressen immer 4-Bytes groß sind, was auf den bisher existierenden 32-Bit-Architekturen der Fall war. Die AMD64-Architektur arbeitet allerdings mit 8-Byte Adressen, wofür das generische Backend erweitert werden musste. Die Erweiterung wurde in der Form vorgenommen, dass sich die Größe der in der Sprungtabelle gespeicherten Adressen dynamisch nach der Architektur richtet.

3.3.2 Stackframe Handling

Im Falle des AMD64-Backends war noch eine Erweiterung des generischen Backends notwendig, weil die Transformation und Übersetzung der Start- und Return-Knoten auch den Stackframe mit aufbauen. Hier übernimmt nun bei Bedarf das generische Backend das verändern des Stackpointer-Bias für den Zugriff auf lokale Variablen.

Kapitel 4

Fazit

Die Implementierung eines AMD64-Backends ist abgeschlossen und wie der nächste Abschnitt zeigen wird, lassen sich einfache C-Programme korrekt übersetzen. Der darauf folgende Abschnitt gibt einen Überblick sowohl über die fertige als auch über die noch offene Funktionalität. Abschließen wird dieses Kapitel und diese Arbeit dann mit einer kurzen Aufzählung der noch möglichen Optimierungen und einer Zusammenfassung zum Abschluss kommen.

4.1 Benchmarks

Da die Implementierung des AMD64-Backends nicht vollständig genug ist um den SPEC Benchmark laufen zu lassen, wurden für ein Vergleich der Ausführungsgeschwindigkeit des ausgegebenen Codes des AMD64-Backends spezielle Test-Programme herangezogen.

Dieser Abschnitt ist in zwei Teile unterteilt: Einmal die Messungen selbst, bei dem auf die Methodik der Benchmark-Messungen eingegangen wird und die Ergebnisse dargestellt werden. Im zweiten Teil werden die Ergebnisse dann kurz bewertet und eingeordnet.

Das Hauptziel der Benchmarks ist die Einordnung der erreichten Performanz der Ausgabe des AMD64-Backends. Zwar wurde das AMD64-Backend nicht unter dem Gesichtpunkt einer möglichst guten Performanz entwickelt, aber diese Einordnung kann mögliche Probleme in der Implementierung aufdecken. Auch kann die Einordnung zeigen, wieviel Raum im Bezug auf den GNU C Compiler noch für Optimierungen bleibt.

Alle Benchmarks wurden auf einem 64-Bit Debian GNU/Linux ausgeführt. Die Hardware bestand aus einer “Intel Core i3 CPU M 370” CPU mit einer Taktfrequenz von 2.40 GHz und 3 Gb Arbeitsspeicher.

4.1.1 Messungen

Die ersten zwei Programme sind einmal eine Quick-Sort Implementierung, dann eine Implementierung des N-Damen-Problems und beim dritten handelt es sich um einen rekursiven Algorithmus zur Berechnung der n'ten Zahl der Fibonacci-Reihe. Auf die Programme, Testmethode und Ergebnisse gehen die folgenden drei Abschnitte nun näher ein. Davon wird der erste Abschnitt über Quick-Sort näher auf die Methodik und Darstellung der Ergebnisse eingehen.

Verglichen wurde der GNU C Compiler für die AMD64-Architektur in 2 verschiedenen Optimerungsstufen mit dem in dieser Arbeit entwickelten AMD64-Backend von LibFirm. Die zwei Optimierungsstufen des GNU C Compilers waren einmal -O0, welche jegliche Art von Optimierung weitestgehen abstellt. Sowie -O3, welche bewirkt, dass alle möglichen Optimierungen durchgeführt werden. Bei LibFirm wurde das voreingestellte Optimierungs-Level 1 verwendet.

Um die Varianz, welche durch nebenläufige Prozesse und anderen verwaltenden Abläufen im Betriebssystem hervorgerufen wird, der Durchlaufzeiten eines Benchmarks gering zu halten, wurden die Problemgrößen der einzelnen Benchmarks angepasst. Die Problemgrößen wurden so gewählt, dass ein Benchmark mindestens eine Sekunde zum durchlaufen brauchte. Von 4 aufeinanderfolgenden Abläufen wurde dann die beste (kürzeste) Zeit genommen.

QuickSort

Die vorliegende Quick-Sort Implementierung kommt aus der Test-Suite von LibFirm. Das Benchmark-Programm erstellt zuerst einen Array gefüllt mit Zufallszahlen, welcher daraufhin mit einer rekursiven Implementierung des Quick-Sort Algorithmus implementiert wird. Das Programm kommt mit einem eigenen LKG (Linear Kongruenz Generator, bzw. Pseudo-Zufallszahlen-Generator) und benutzt die Standard C Bibliothek nur für die Ein- und Ausgabe. Die Größe des Arrays konnte über die Kommandozeile an das Benchmark-Programm übergeben werden. Um eine Ausführungszeit von mindestens einer Sekunde im besten Fall zu erreichen wurde eine Arraygröße von 7000000 Elementen gewählt. Die Ergebnisse des Benchmarks sind in der folgenden Tabelle dargestellt:

	Zeit (Sek.)	libFirm	gcc -O0	gcc -O3
libFirm	2.18	–	-6%	-53%
gcc -O0	2.05	7%	–	-50%
gcc -O3	1.03	112%	98%	–

In der Zeilenbeschriftung in der linken Spalte ist angegeben welches Compilat ausgeführt wurde. In der nächsten Spalte mit der Überschrift "Zeit" steht die beste Ausführungszeit von 4 direkt aufeinander folgenden Ausführungen des entsprechenden Compilats. Die Zeilen sind nach der Ausführungszeit absteigend sortiert. Weitere Spalten listen noch einmal den jeweiligen Compiler auf und die zugehörigen Zellen geben einen Prozentualen vergleichswert zwischen den Compilern wieder. Die Werte oberhalb der Diagonalen geben an, um wieviel Prozent der Compiler in der jeweiligen Zeile langsamer war. Die Werte unterhalb der Diagonalen geben an, umwieviel der jeweilige Compiler schneller war.

In der hier gegebenen Tabelle lässt sich erkennen, dass das Compilat des AMD64-Backends am längsten gebraucht hat (2.18 Sekunden) den Quick-Sort Benchmark auszuführen. Damit ist es das langsamste Compilat des Benchmarks und 6% langsamer als das unoptimierte GCC Compilat und 53% langsamer als das optimierte GCC Compilat.

Queens

Beim N-Damen-Problem wurde auch auf die LibFirm-Testsuite zurück gegriffen. Hier wurde die Problemgröße vom klassischen 8-Damen-Problem auf das 13-Damen-Problem angehoben, damit der Benchmark mindestens eine Sekunde Ausführungszeit erreicht. Die Ergebnisse sind in folgender Tabelle festgehalten:

	Zeit (Sek.)	gcc -O0	libFirm	gcc -O3
gcc -O0	5.11	–	-43%	-77%
libFirm	2.91	75%	–	-61%
gcc -O3	1.15	344%	154%	–

Fibonacci

Für die Berechnung der n'ten Fibonacci Zahl wurde ein eigenes Benchmark-Programm entwickelt, welches einen rekursiven Algorithmus benutzt zum Berechnen. Für den Benchmark wurde die 41. Fibonacci-Zahl berechnet. Die Ergebnisse des Benchmarks sind in folgender Tabelle festgehalten:

	Zeit (Sek.)	gcc -O0	libFirm	gcc -O3
gcc -O0	3.1	–	-25%	-61%
libFirm	2.33	33%	–	-48%
gcc -O3	1.2	158%	94%	–

4.1.2 Bewertung

Die Benchmarks haben gezeigt, dass sich die Performanz, des vom AMD64-Backend erzeugten Codes, mit der Performanz der GCC-Ausgaben vergleichen lässt. Die Ausführungszeiten lagen entweder im vom GCC aufgespannten Bereich, oder im Falle des Queens-Benchmarks nahe dran. Das bedeutet, dass es keine fundamentalen Performanz-Probleme mit dem erzeugten Code des AMD64-Backends gibt. Es zeigt jedoch auch, dass noch einiger Raum für Optimierungen bleibt. Näheres hierzu im Abschnitt 4.2.2 Optimierungen.

4.2 Stand der Implementierung

Wie die Benchmarks gezeigt haben, ist das AMD64-Backend in der Lage einfache C-Programme auszuführen. Insbesondere auch das ursprünglich angesetzte Ziel des 8-Damen Beispiels aus der LibFirm Testsuite. Die Ganzzahlarithmetik von C wird korrekt in AMD64-Assemblercode übersetzt und alle Firm-Knoten-Klassen werden transformiert und in AMD64-Assembler übersetzt. Zusammen mit einer Implementierung der entsprechenden Calling Conventions der AMD64-Architektur sind damit einfache C-Programme korrekt übersetzbar.

4.2.1 Offenes

Noch offen sind Gleitkommaoperationen und eine vollständigere Implementierung der Calling Conventions, z.B. um die Übergabe von C-Structs und Unions zu ermöglichen. Eine Überarbeitung der Ganzzahlarithmetik muss auch vorgenommen werden, da die Arithmetik-Operationen mit 64-Bit Zahlen arbeiten, hier sind Probleme mit der Korrekheit der C-Übersetung zu erwarten.

Auch geht die momentane Implementierung davon aus, dass keine Frame-Pointer erzeugt werden. Der Frame-Pointer ist ein speziell reserviertes Register (üblicherweise *%rbp*), welches zur Addressierung der lokalen Variablen verwendet wird. Daher müssen Programme zur Zeit mit `-fomit-frame-pointer` übersetzt werden. Siehe hierzu auch die Calling Convention Beispiele 2.1.4, bei denen der Frame-Pointer in *%rbp* erzeugt wird.

4.2.2 Optimierungen

Wie die Benchmarks auch gezeigt haben, ist noch einiger Spielraum für Optimierungen. Hier fällt bei der Betrachtung des erzeugten Assemblers auf, dass viele der von der AMD64-Architektur angebotenen Adressierungsmodi ungenutzt sind. Wie im IA32-Backend können durch eine intelligentere Ausnutzung der Adressierungsmodi überflüssige Adressberechnungen vermieden werden, welche insbesondere bei Arrayzugriffen anfallen.

Zudem könnte die Registerallokation verbessert werden, da derzeit der Registerallokator davon ausgeht, dass Arithmetikoperationen beliebige Eingabe- und Ausgaberegister haben können. Allerdings haben viele AMD64-Operationen nur 2 Operanden (welche je nach Operation im Speicher oder in Registern liegen können) und legen das Ergebnis der Operation am Speicherort des zweiten Operanden ab. Ein verbesserter Registerallokator könnte dies von vornherein in betracht ziehen und die Register gleich passend vergeben.

Auch die Ganzzahlarithmetik kann durch relativ einfache Verbesserungen noch beschleunigt werden. So ist beispielsweise die Übersetzung des Firm Knotens für die Subtraktion noch sehr ineffizient. Hier wird eine Subtraktion in 2 Negations-Operationen und eine Addition übersetzt.

4.2.3 Zusammenfassung

In dieser Arbeit wurde ein AMD64-Backend für die LibFirm Compilersuite entworfen und implementiert. Es werden einfache C-Programme ohne Gleitkommawerte, wie zum Beispiel das 8-Damen-Problem aus der LibFirm Testsuite korrekt übersetzt. Die Performance des ausgegeben Codes liegt unter der vom GNU C Compiler auf höchster Optimierungsstufe, allerdings über der niedrigsten Optimierungsstufe. Die Performance lässt sich noch weiter steigern durch eine bessere Ausnutzung der Adressierungsmodi und einer verbesserten Registerallokationsstrategie.

Kapitel 5

Anhang

Literaturverzeichnis

[1] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.

[2] Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch. Ssa-based register allocation with pbqp. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 42–61. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19861-8_4.

[3] AMD Corporation. *AMD64 Architecture Programmer's Manual: Volume 1: Application Programming*, September 2007.

[4] AMD Corporation. *AMD64 Architecture Programmer's Manual: Volume 2: System Programming*, September 2007.

[5] Intel Corporation. *Intel® 64 and IA-32: Architectures Software Developer's Manual*, December 2011.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[7] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In Alan Mycroft Andreas Zeller, editor, *Compiler Construction 2006*, volume 3923. Springer, March 2006.

[8] Götz Lindenmaier. libfirm – a library for compiler optimization research implementing firm. Technical Report 2002-5, Sep 2002.

[9] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*, September 2010.

[10] 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, pages 12–27, New York, NY, USA, 1988. ACM.

[11] Stefan Staiger, Gunther Vogel, Steffen Keul, and Eduard Wiebe. Interprocedural static single assignment form. In *In Proceedings of the 14 th Working Conference on Reverse Engineering (WCRE)*, 2007.