Universität Siegen
Institut für Praktische Informatik
Naturwissenschaftlich-Technische Fakultät

Diplomarbeit

Linear-Scan-Registerallokation im OCaml-Native-Code-Compiler

Marcell Fischbach

3. November 2011

Gutachter: Privatdozent Dr. Kurt Sieber
Dipl. Inf. Benedikt Meurer

Inhaltsverzeichnis

1 Einleitung **4**

2 Einordnung in das Projekt **7**

3 Der Linear-Scan-Algorithmus **9**
3.1 Der grundlegende Algorithmus 9
3.2 Einordnung in den Compileprozess 11
3.3 Erzeugung der Intervalle 16
3.3.1 Intervallstruktur 16
3.3.2 Der Instruktionsbaum 18
3.3.3 Erzeugung der Intervalle 20
3.3.4 Fixe Intervalle 22
3.3.5 Zerstörte Register 23
3.3.6 Alternative Intervalldarstellung 24
3.4 Der Linear-Scan-Algorithmus 25
3.4.1 Abarbeiten der Intervalle 26
3.4.2 Zuweisung eines freien physikalischen Registers 27
3.4.3 Zuweisung bei blockierten Registern 29
3.4.4 Gespillte Pseudoregister 30
3.4.5 Lösen von ungültigen Instruktionsargumenten 31

4 Die native *OCaml-Toplevel* **33**
4.1 Der Workflow 34
4.1.1 Der Bytecode-Workflow 34
4.1.2 Der native Workflow 36
4.2 Motivation 37
4.3 Opcode und ASM-Emitter 38
4.3.1 Operandendarstellung 40
4.3.2 Label und Symbole 42
4.3.3 Relocations 42
4.4 Linken und Ausführen 44

4.4.1 Allozieren des Speichers . . . 45
4.4.2 Globale Symbole . . . 45
4.4.3 Globale Relocation . . . 46
4.4.4 Ausführen . . . 46

5 Performancevergleich **48**
5.1 Compiletime . . . 50
5.1.1 Pathologischer Fall . . . 51
5.2 Runtime . . . 52
5.3 Native OCaml Toplevel . . . 53

6 Fazit **55**

Literatur **56**

Erklärung **58**

Kapitel 1

Einleitung

Zwei konkurrierende Entwicklungsziele des Compilerbaus bestehen zum Einen darin, einen Compiler zu entwickeln, welcher möglichst schnell arbeitet und zum Anderen darin, Maschinencode zu erzeugen, welcher möglichst schnell ausgeführt werden kann. In Offlinesituationen, in denen Compilieren und Ausführen getrennt voneinander erfolgen, besteht die oberste Prämisse darin, möglichst optimalen und damit schnellen Code zu erzeugen. Die Laufzeit des Compilers ist in solchen Situationen von untergeordneter Bedeutung. In Onlinesituationen jedoch, wo ein Programm in Form von Quelltext compiliert und direkt im Anschluss ausgeführt wird, trägt die Compilezeit effektiv zu der Laufzeit des Programms bei. In solchen Situationen ist die Laufzeit des Compilers von erheblich größerer Bedeutung.

Die Registerallokationsphase eines Compilers trägt Maßgeblich zu der Erzeugung von leistungsfähigem Code bei, denn in dieser Phase wird die Entscheidung getroffen, welche der Werte des Programms sich zu welcher Zeit in Registern befinden. Da Register die performantesten Speicher eines Computers sind, ist eine geschickte Nutzung maßgeblich für die Qualität des entstehenden Codes von Bedeutung. Jedoch ist diese Phase auch die, welche am längsten läuft. In den gängigen Compilern, wie z.B. dem GNU `gcc` oder dem *Java HotSpot*™*Server Compiler*, basiert der verwendete Algorithmus auf dem *Graph-Coloring-Algorithmus* [Cha82], welcher 1982 von Gregory Chaitin vorgestellt wurde. Hierbei wird versucht, die Knoten eines Graphen beliebiger Komplexität mit `N` unterschiedlichen Farben zu färben, sodass keine zwei durch eine Kante miteinander verbundene Knoten die gleiche Farbe bekommen. Der Wert `N` gibt die Anzahl der verfügbaren CPU-Register an. Die Knoten des Graphen werden durch alle Variablen, Zwischenergebnisse etc. gebildet und Kanten entstehen immer dann, wenn zwei dieser Werte in einer Instruktion benötigt werden. Dieser Algorithmus hat jedoch quadratische Laufzeit, was ihn mit steigender Programmkomplexität stetig unattraktiver

macht.

Die funktionale Programmiersprache *OCaml*, welche im Wesentlichen den Quelltext in Bytecode compiliert, welcher interpretiert ausgeführt werden kann, besitzt seit 1995 ebenfalls einen Compiler, welcher den Quelltext in nativen Code compilieren kann. Auch hier basiert der Registerallokator auf dem *Graph-Coloring-Algorithmus*. Gegenstand dieser Diplomarbeit besteht in der Implementierung des *Linear-Scan-Algorithmus* für den *OCaml-Native-Code-Compiler*. Der *Linear-Scan-Algorithmus* stellt einen alternativen Registerallokator mit linearer Laufzeit dar. Dieser Algorithmus basiert auf einer Liste von Livetime-Intervallen statt auf einem Graphen. Der grundlegende Algorithmus wird in Kapitel 3.1 erleutert. Der Ablauf unterteilt sich in zwei Bereiche. Kapitel 3.3 erläutert den Aufbau und die Erzeugung der Intervalle, während Kapitel 3.4 die Implementierung des *Linear-Scan-Algorithmus*, wie er im Rahmen dieser Diplomarbeit entstanden ist, beschreibt.

Die Toplevel von *OCaml* stellt die oben genannte Onlinesituation dar, mit der Quelltext mit einem Aufruf compiliert und ausgeführt werden kann. Wie bei dem *OCaml-Compiler* gibt es auch bei der *OCaml-Toplevel* eine bytecode und eine native Variante. Die Ausführungszeit von nativem Code ist verglichen mit der Ausführungszeit von interpretiertem Bytecode erheblich schneller. Jedoch basiert der Workflow des nativen Compilers auf dem Vorhandensein einer Toolchain, mit welcher die Maschinencodeerzeugung sowie das Linken durchgeführt werden kann. Dies bringt jedoch Probleme mit sich, denn zum Einen besteht die Möglichkeit, dass diese Toolchain auf dem System nicht vorhanden ist und zum Anderen führt diese Nutzung zu hohen Latenzen, da immer wieder Zwischenschritte auf eine Festplatte gespeichert werden müssen. In Kapitel 4 wird eine Proof-Of-Concept-Implementierung für die *i386-Architektur* vorgestellt, welche ohne eine externe Toolchain auskommt. Die Maschinencodeerzeugung, d.h. das Generieren von Prozessoropcodes und das Linken, werden direkt in den *OCaml-Native-Code-Compiler* integriert. Dazu wird in Kapitel 4.1 ausführlich beschrieben, wie der Workflow der Codegenerierung aufgebaut ist und welche Probleme damit verbunden sind. Kapitel 4.3 beschreibt den Aufbau und die Funktionsweise, mit der die Prozessoropcodes erzeugt werden und Kapitel 4.4 erläutert das Zusammenspiel von Implementierungen in *OCaml* und *C* für die Allozierung von ausführbarem Speicher und das Ausführen des erzeugten Codes.

In Kapitel 5 werden die Implementierungen, welche im Rahmen dieser Diplomarbeit entstanden sind, in den direkten Vergleich zu den bisherigen Implementierungen gestellt. Diese Vergleiche gliedern sich in drei Unterbereiche. In Kapitel 5.1 wird die Compiletime des *OCaml-Native-Code-Compilers* mit *Graph-Coloring* und mit *Linear-Scan* verglichen, während sich Kapitel 5.2 mit der Ausführungszeit des compilierten Codes befasst. Als letztes wird

in Kapitel 5.3 die Performance der nativen Toplevel, zum Einen mit der bisherigen Variante und zum Anderen mit der Variante dieser Arbeit, in den direkten Vergleich zu der Bytecode Variante gestellt.

Kapitel 2

Einordnung in das Projekt

Die erste Implementierung der funktionalen Programmiersprache *Caml* wurde von 1987 bis 1992 von dem französischen *Institut national de recherche en informatique et en automatique* (INRIA) entwickelt. In dieser Implementierung wurde der *Caml-Code* durch den *Caml-Compiler* in LLM3 Code gewandelt, welcher wiederum in der virtuellen Machine des *Le Lisp-Systems*[1] ausgeführt werden konnte.

Durch ein Redesign von Xavier Leroy in 1990 und 1991 wird der *Caml-Code* durch den *Caml-Compiler* in eine eigene Bytecoderepräsentation gewandelt, welche durch einen Bytecodeinterpreter ausgeführt werden kann. Die Implementierung des Bytecodeinterpreters erfolgte in *C*. Diese Loslösung von dem *Le Lisp-System* erbrachte eine erhebliche Portabilitätssteigerung. Diese Version von *Caml* wurde als *Caml Light* veröffentlicht.

Im Jahr 1995 wurde die Version *Caml Special Light* durch Xavier Leroy veröffentlicht. Eine der wichtigsten Änderungen zu *Caml Light* bestand in der Implementierung eines *Native-Code-Compilers*. Der *Caml-Code* wird nicht mehr in Bytecode übersetzt um dann von einem Interpreter ausgeführt zu werden, sondern es wird direkt nativer, ausführbarer Code erzeugt.

Im Jahr 1996 wurde *Caml* durch die Arbeit der Entwickler Didier Rémy und Jérôme Vouillon um objektorientierte Konzepte erweitert [Rem02], was zu der Sprache *Objective Caml* führte, welche später in *OCaml* umbenannt wurde.

Die Entwicklungen dieser Arbeit beziehen sich auf die *OCaml-Version* `3.12.1`, welche die aktuelle Version zu dem Entwicklungszeitpunkt darstellte. Der komplette Quellcode, inklusive einiger nützlicher Bibliotheken, ist auf der Webseite[2] des *OCaml-Projekts* frei zum Download erhältlich. Die-

[1] http://christian.jullien.free.fr/lelisp - Ebenfalls von INRIA entwickelt

[2] http://http://caml.inria.fr/

ser Quellcode gliedert sich in diverse Verzeichnisse, von denen die für diese Arbeit wichtigsten kurz vorgestellt werden sollen.

- `asmcomp`: Der Order `asmcomp` enthält den Quelltext für das Backend des *Native-Code-Compilers*. Diverse Optimierungen, der Registerallokator und der Assemblercode-Emitter sind hier angesiedelt. Ebenfalls befinden sich in diesem Verzeichnis Unterverzeichnisse für die diversen unterstützten Architekturen, wie *amd64*, *i386*, *PPC* etc. Diese Verzeichnisse wiederum beinhalten architekturspezifische Implementierungen für den Assemblercode-Emitter, die Prozessor-Spezifikation, die verfügbaren CPU-Register, Callingconventions...

- `asmrun`: Enthält den Quelltext der Laufzeitbibliothek, nativ compilierter Programme, sowie entsprechende Prozeduren der nativen Toplevel zum Laden, Linken und Ausführen nativ compilierter Programme.

- `bytecomp`: Der Ordner `bytecomp` enthält den Quelltext für das Backend des Bytecode-Compilers.

- `byterun`: In dem Verzeichnis `byterun` befindet sich der Interpreter, geschrieben in *C*, für die Ausführung des Bytecodes.

- `parsing, typing`: Das Verzeichnis `parsing` enthält den Parser für die Transformation von *OCaml-Code* in einen abstrakten Syntaxbaum und das Verzeichnis `typing` enhält das *OCaml-Typsystem*.

- `toplevel`: In diesem Ordner befindet sich der Quelltext der *OCaml-Toplevel*. Sowohl für die Bytecode-Variante als auch für die native Variante.

- `testsuite`: Hier ist eine Testbench für *OCaml* zu finden, welche aus Testprogrammen diverser Kategorien unterschiedlicher Testgebiete besteht.

Die Implementierung des *Linear-Scan-Algorithmus* ist Teil des *OCaml-Native-Code-Compilers* und ist somit Teil von `asmcomp`. Die Implementierung ist in den Dateien `asmcomp/interval.ml` sowie `asmcomp/linscan.ml` zu finden. Die Implementierungen für die Erzeugungen des Native-Codes sind in `asmcomp/i386/emit.mlp` sowie in `toplevel/opttoploop.ml` und `asmrun/natdynlink.c` zu finden.

Kapitel 3

Der Linear-Scan-Algorithmus

Die im Rahmen dieser Diplomarbeit entstandene Implementierung des *Linear-Scan-Algorithmus* folgt im Wesentlichen dem Algorithmus von M. Polette und V. Sarkar [PS99], obwohl einige Änderungen aufgrund der gesteigerten Codequalität von Traub et al. [THS98] übernommen wurden. Des Weiteren wurde aus sprachgebundenen Implementierungsgründen an diversen Stellen von dem originalen Algorithmus abgewichen.

- Der ursprüngliche Algorithmus von Polette und Sarkar sieht Intervalle als in sich abgeschlossen. Um auf die natürliche Struktur des Codes einzugehen, werden Lücken in den Intervallen berücksichtigt.
- Um große Teile des bestehenden *OCaml-Codes* nutzen zu können, werden die Intervalle nicht aus der linearisierten Form erzeugt, sondern in einem rekursiven Scan über den Instruktionsbaum[1] in Depth-First-Order
- Die `active`-Liste wird in umgekehrter Reihenfolge gehalten, um Geschwindigkeitsvorteile von *OCaml-Listen* zu nutzen, da bei diesen ein Zugriff auf das erste Element direkt erfolgen kann, während die Liste vollständig durchlaufen werden muss, um das letzte Element zu erhalten.

3.1 Der grundlegende Algorithmus

Auch wenn die tatsächliche Implementierung in einigen Punkten von dem grundlegenden Algorithmus abweicht, so ist die generelle Struktur identisch.

[1] Dieser Instruktionsbaum ist eine Zwischen-Code Darstellung im Native-Code-Compiler

Im Folgenden soll der grundsätzliche Ablauf des Algorithmus kurz vorgestellt werden. Abbildung 3.1 zeigt den Algorithmus in einer Pseudocodedarstellung. Die verwendete Codekonvention enstspricht derjenigen Konvention, die über das gesamte Dokument hinweg verwendet wird. Namen von Variablen und Funktionen werden `mit fester Breite` dargestellt, Schlüsselwörter werden in **fett** dargestellt. Grundsätzlich werden alle Implementierungsdetails weggelassen.

```
linear_scan_register_allocation
  active ← {}
  for each live interval i, in order of increasing start point
    expire_old_intervals(i)
    if length(active) = R then
      spill_at_interval(i)
    else
      register[i] ← a register removed from pool of free registers
      add i to active, sorted by increasing end point
  end for

expire_old_invervals(i)
  foreach inverval j in active, in order of increasing end point
    if endpoint[j] ≥ startpoint[i] then
      return
    remove j from active
    add register[j] to pool of free registers
  end for

split_at_interval(i)
  spill ← last interval in active
  if endpoint[spill] > endpoint[i] then
    register[i] ← register[spill]
    location[spill] ← new stack location
    remove spill from active
    add i to active, sorted by increasing end point
  else
    location[i] ← new stack location
```

Abbildung 3.1: Der grundlegende *Linear-Scan-Registerallokationsalgorithmus*

Da der Algorithmus nichts über den Aufbau und die Gewinnung der Intervalle aussagt, wird angenommen, dass diese bereits in einem vorangegangen

Schritt ermittelt wurden.

Das grundlegende Konstrukt des *Linear-Scan-Algorithmus* bildet die Liste `active`. In ihr werden die Intervalle in chronologisch absteigender Reihenfolge gehalten, die zu jedem Zeitpunkt einem physikalischen Register zugewiesen sind. Die Größe dieser Liste hat dadurch eine natürliche Obergrenze, nämlich die Anzahl der vorhandenen physikalischen Register. In einem linearen Durchlauf über alle Intervalle wird versucht diesen ein Register zuzuweisen. Der Startzeitpunkt des aktuellen Intervalls spiegelt den augenblicklichen Zeitpunkt der Registervergabe wider.

Alle Intervalle, die sich in `active` befinden, die vor diesem Zeitpunkt bereits abgelaufen sind, können aus `active` entfernt werden, da diese nach diesem Zeitpunkt nicht mehr benötigt werde. Wenn nach den Entfernungen aus `active` noch Platz für ein weiteres Intervall ist, so kann dem aktuellen Intervall ein zu diesem Zeitpunkt nicht vergebenes Register zugewiesen werden und es kann in `active` aufgenommen werden.

Ist kein Platz für ein weiteres Register in `active`, sind also zu diesem Zeitpunkt alle Register vergeben, so muss ein Intervall gespillt werden, d.h. es muss diesem Intervall ein Slot auf dem Stack zugewiesen werden. Das Intervall, dessen Endzeitpunkt am weitesten in der Zukunft liegt, ist der Kandidat, der gespillt werden muss. Dieser Kandidat kann entweder das aktuell betrachtete Intervall oder das letzte Intervall in `active` sein. Ist es das aktuelle Intervall, so wird diesem ein Stackslot zugewiesen. Befindet sich der Kandidat in `active`, so muss dieser aus `active` entfernt werden. Das Register, welches diesem Intervall zugewiesen ist, wird an das aktuelle Intervall übertragen und dem Kandidaten wird ein Stackslot zugewiesen. Im Anschluss kann nun das aktuelle Intervall in `active` eingefügt werden.

Wenn alle Intervalle durchlaufen sind, so wurde jedem Intervall entweder ein Register oder ein Platz auf dem Stack zugewiesen und der Algorithmus ist beendet. Wie sich jedoch herausstellt, ist diese Vergabe der Register nicht immer gültig. So ist es z.B. auf den *Intel-Architekturen i386* und *amd64* nicht gültig, in einer Instruktion zwei Speicheroperanden zu verwenden. Der Registerallokator kann jedoch nicht gewährleisten, dass dies nicht vorkommt. Somit müssen gegebenenfalls im Anschluss noch Codeveränderungen durchgeführt werden, welche diese Konflikte auflösen.

3.2 Einordnung in den Compileprozess

Der Compileprozess des *OCaml-Native-Code-Compilers* besteht aus einer Folge von mehreren Prozessschritten. Diese Schritte beginnen mit dem Parsen des zu compilierenden Quellcodes, welcher aus einer `*.ml`-Datei stammt,

oder, im Falle der interaktiven Shell der *OCaml-Toplevel*, auch direkt aus der Eingabe gelesen werden kann. Der geparste Quellcode wird durch den Parser (siehe dazu die Datei `parsing/parsetree.mli` der *OCaml-Quellen*) in einen abstrakten Syntaxbaum (AST) transformiert. Abbildung 3.2 zeigt einen abstrakten Syntaxbaum für den Ausdruck `let x=1 in x+3`.

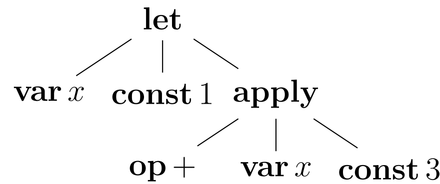

Abbildung 3.2: Abstrakter Syntaxbaum

In dem nächsten Schritt des Compilers werden Typannotationen berechnet (siehe dazu die Datei `typing/typedtree.ml`) um daraus den getypten Syntaxbaum zu erzeugen, wie er in Abbildung 3.3 dargestellt ist.

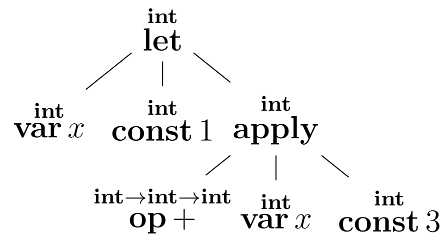

Abbildung 3.3: Getypter Syntaxbaum

Aus diesem getypten Syntaxbaum erzeugt der Compiler, inspiriert durch das ungetypte call-by-value λ-Kalkül, die sogenannte Lambda-Repräsentation (siehe dazu die Datei `bytecomp/lambda.ml`) wie in Abbildung 3.4 dargestellt.

```
(let (x/1038 1) (+ x/1038 3))
```

Abbildung 3.4: Lambda-Repräsentation

Diese Lambda-Repräsentation wird im Anschluss optimiert, indem Transformationen durchgeführt werden, welche die Lambda-Bäume in kleinere oder bessere Lambda-Bäume umwandeln (siehe dazu die Datei `bytecomp/simplif.ml`). Diese verbesserte Lambda-Darstellung wird weiter verarbeitet, indem sie in eine Variante der Lambda-Darstellung (siehe dazu die Datei `asmcom/clambda.ml`)

mit expliziten Closures und expliziten direct/indirect function calls (siehe dazu die Datei `asmcom/closure.ml`) transformiert wird. Diese wiederum wird in eine äquivalente Darstellung in dem internen Dialekt *C*-- umgewandelt (siehe dazu die Dateien `asmcomp/cmm.ml` und `asmcomp/cmmgen.ml`). Abbildung 3.5 zeigt die *C*---Repräsentation für den Ausdruck `let abs x = if x < 0 then -x else x`.

```
(function camlAbs__abs_1030 (x/1031: addr)
 (if (< x/1031 1) (- 2 x/1031) x/1031))
```

Abbildung 3.5: *C*---Repräsentation

Aus diesem *C*-- Dialekt werden in der Selektion (siehe dazu die Datei `asmcomp/selection.ml`) architekturspezifische Hardwareinstruktionen er– zeugt. Die Struktur dieser Instruktionen ist in der Datei `asmcomp/mach.ml` beschrieben. Die Struktur dieser Repräsentation basiert immer noch auf einem Baum. Erst in dem vorletzten Schritt, der Linearisierung, wird diese Baumstruktur in eine lineare Darstellung transformiert. Abbildung 3.6 zeigt die Hardwareinstruktionen für den Beispielausdruck.

```
camlAbs__abs_1030(R/0[%eax])
  x/8 := R/0[%eax]
  if x/8 <s 1 then
    I/9 := 2
    I/10 := I/9
    I/10 := I/10 - x/8
    R/0[%eax] := I/10
    return R/0[%eax]
  else
    R/0[%eax] := x/8
    return R/0[%eax]
  endif
```

Abbildung 3.6: Hardwareinstruktionen

Im nachfolgenden Schritt Allocation-Combining (siehe dazu die Datei `asmcomp/comballoc.ml`) werden alle Heap-Allokationen, welche in einem Basicblock auftreten, zu einer Allokation zusammengefasst. Dies hält zum Einen den Code kleiner und zum Anderen ist dadurch eine Effizienzsteigerung in der Ausführung zu erwarten.

In der Liveness-Analyse wird, wie der Name bereits beschreibt, die Live– ness der Pseudoregister ermittelt.

Im folgenden Schritt werden Pseudoregister ermittelt, welche während Instruktionen live sind, die potentiell alle Register zerstören (auf das Problem der zerstörten Register wird in Kapitel 3.3.5 näher eingegangen). Diese Instruktionen sind z.B. `Icall_ind`, `Icall_imm` (*OCaml-Calls*), `Iextcall` (*C-Calls*) und `Itrywith` (bei dem werfen von Exceptions). Da diese Instruktionen alle Register überschreiben, müssen alle Pseudoregister, die während der Ausführung dieser Instruktionen live sind, auf dem Stack liegen, damit ihre Werte nicht verloren gehen. In dem Spill-Schritt (siehe dazu die Datei `asmcomp/spill.ml`) wird für jedes dieser Pseudoregister ein weiteres Pseudoregister angelegt, welches als gespillt markiert wird. In der Registerallokationsphase wird diesem neuen Pseudoregister ein entsprechender Platz auf dem Stack zugewiesen. Nun kann vor der Überschreibung eine `Ispill`-Instruktion eingefügt werden, welche eine Zuweisung von dem originalen Pseudoregister an das neue durchführt, wodurch dieser Wert auf den Stack ausgelagert wird. Nach der überschreibenden Instruktion wird eine `Ireload`-Instruktion eingefügt, welche den Wert vom Stack zurück in das Pseudoregister lädt. Abbildung 3.7 zeigt ein kurzes Beispiel, wie das Pseudoregister `reg0` in Zeile `2` gespillt wird, um nach einem Call in Zeile `4` wieder zurück in das Pseudoregister `reg0` geladen zu werden.

```
1: Imov 10, reg0
2: Ispill reg0, reg1
3: Icall
4: Ireload reg1, reg0
```

Abbildung 3.7: Spilling eines Pseudoregister

Im nächsten Schritt, dem Split-Schritt (siehe dazu die Datei `asmcomp/split.ml`), werden Live-Range-Splittings durchgeführt, indem neue Pseudoregister eingeführt werden, in welche die gespillten Pseudoregister geladen werden. Abbildung 3.8 zeigt das gleiche Beispiel wie oben, jedoch wird in Zeile `4` der Wert nun in ein neues Register geladen. Ab diesem Zeitpunkt wird nun das Pseudoregister `reg2` an Stelle von `reg0` verwendet.

Der nächste Schritt ist der zentrale Punkt dieser Arbeit, die Registerallokation. Hier wird versucht, jedem der Pseudoregister ein physikalisches Register zuzuweisen. Die bisherige Implementierung basiert auf dem *Graph-Coloring-Algorithmus* nach Chaitin. Dieser Algorithmus ist in zwei Teile unterteilt. Der Interferenzgraph wird in `asmcomp/interf.ml` aufgebaut und die Färbung des Graphen, also die Registervergabe und Zuweisung, findet in `asmcomp/coloring.ml` statt.

```
1: Imov 10, reg0
2: Ispill reg0, reg1
3: Icall
4: Ireload reg1, reg2
```

Abbildung 3.8: Live-Range-Splitting durch neues Pseudoregister nach `Ireload`

Wie bereits angesprochen, basieren all diese Schritte auf einem Instruktionsbaum. In dem nun folgenden Schritt, der Linearisierung (siehe dazu die Datei `asmcomp/linearize.ml`), wird aus diesem Baum eine lineare Folge von Instruktionen erzeugt. Abbidlung 3.9 zeigt diese linearisierte Darstellung für den Beispielausdruck.

```
camlAbs__abs_1030:
  if x/8[%eax] >=s 1 goto L100
  I/9[%ebx] := 2
  I/10[%ebx] := I/10[%ebx] - x/8[%eax]
  R/0[%eax] := I/10[%ebx]
  return R/0[%eax]
  L100:
  return R/0[%eax]
```

Abbildung 3.9: Linearisierte Darstellung

In dem nun letzten Schritt, dem Emitting (siehe dazu die Datei `asmcomp/emit.ml`), wird für jede der Instruktionen der linearen Darstellung eine äquivalente Darstellung in Form von Mnemonics einer textuellen Assemblersprache in eine Datei geschrieben. Diese Datei kann im Anschluss von einem Assembler-Compiler in Maschinencode umgewandelt werden. Dieser Schritt ist plattformspezifisch[2], sodass für jede der unterstützten Plattformen eine eigene Implementierung des Emitters existiert.

Für die Erzeugung der Intervalle aus den Pseudoinstruktionen eignet sich die lineare Darstellung des Programms am besten. Dies ist jedoch aus zwei Gründen nicht möglich:

1. Der Linearisierungsschritt erwartet, dass den Pseudoregistern des übergebenen Codes bereits physikalische Register zugewiesen wurden. Diese

[2] Es ist auch betriebssystemspezifisch insofern, dass für *Windows-Betriebssysteme* immer auch noch eine NT-Variante existiert. Die NT-Variante verwendet die *Intel-Syntax*, während die anderen Varianten die *AT&T-Syntax* verwenden.

physikalischen Register werden dort benötigt, um unnötige Instruktionen bereits an dieser Stelle zu entfernen. Solche Instruktionen sind beispielsweise Move-Instruktionen, bei denen Quelle und Ziel identisch sind (z.B. `mov %eax, %eax`). Dies ist jedoch nicht möglich, wenn die Registerallokation erst nach der Linearisierung durchgeführt wird.

2. Nach der Registerallokation muss evtl. zusätzlicher Code eingefügt werden, um illegale Instruktionsargumente zu korrigieren. Solche illegalen Argumentkombinationen können z.B. auf der *i386-Architektur* Stack-Stack-Moves sein. Die bestehenden Routinen für das Code-Reloading, die genau diese Problemstellung lösen, arbeiten jedoch auf dem Instruktionsbaum und nicht auf der linearen Darstellung.

Aus den oben genannten Gründen ist es sinnvoll, die *Linear-Scan-Register–allokation* in den gleichen Prozessschritt zu integrieren, wie dies in der *Graph-Coloring* Variante der Fall ist. Anderenfalls müssten diverse Funktionen, welche bereits bestehen, erneut implementiert werden. Da solche Reimplementierungen zum Einen fehleranfällig sind und zum Anderen der Wartungsaufwand stark ansteigt, ist dies zu vermeiden.

3.3 Erzeugung der Intervalle

Auch der *Linear-Scan-Algorithmus* besteht aus zwei Teilschritten. In diesem Kapitel wird der erste Schritt beschrieben, in dem aus den Instruktionen die Liveintervalle erzeugt werden. Im zweiten Schritt wird mit Hilfe dieser Liveintervalle eine Vergabe von physikalischen Registern an Pseudoregister durchgeführt (dies wird in Kapitel 3.4 beschrieben). In der Datei `asmcomp/interval.ml` ist die Implementierung der Erzeugung der Liveintervalle zu finden.

3.3.1 Intervallstruktur

Für jedes Pseudoregister wird während der Erzeugung der Intervalle eine Instanz eines Intervalls `interval` wie in Abbildung 3.10 angelegt. Jedes Intervall enthält eine Referenz auf das Pseudoregister, welches später während der Registervergabe modifiziert werden kann. Darüber hinaus enthält ein Intervall den Definitionsbereich, der durch `begin` und `end` gegeben ist. `Begin` gibt während der Ausführung des *Linear-Scan-Algorithmus* den jeweils aktuellen Zeitpunkt des Intervalls an und `end` wird genutzt, um die Intervalle in `active` entsprechend zu sortieren. Die `ranges`, welche die lückenlosen Abschnitte eines Intervalls bilden, sind ebenfalls über dem Definitionsbereich `begin` und

`end` definiert. Mathematisch gesehen, ist dieser durch `[begin, end]` definiert.

struct *range*
  `begin` : *int*
  `end` : *int*

struct *interval*
  `reg` : *pseudoregister*
  `begin` : *int*
  `end` : *int*
  `ranges` : *range list*

Abbildung 3.10: Datenstruktur der Intervalle

Eine Range beschreibt den Bereich eines Pseudoregisters, beginnend bei der Instruktion, in der es als Zieloperand auftritt, und endend bei der letzten Instruktion, in der es als Quelloperand auftritt, bevor es erneut als Zielop–erand auftretend überschrieben wird. Innerhalb einer Range kann ein Pseudoregister somit exakt einmal geschrieben aber beliebig häufig gelesen werden.

In Kapitel 3.3.2 wird beschrieben, wie aus den Instruktionen die Intervalle erzeugt werden. Bei diesem Prozess wird jeder Instruktion ein Index zugewiesen, welcher die Grundlage für `[begin, end]` in den Ranges und in den Intervallen bildet.

Jeder dieser Indizes ist in zwei Slots unterteilt, den `ARG`-Slot und den `RES`-Slot. Im Weiteren dieser Arbeit wird auch $\texttt{instr}_{RES}$ oder $\texttt{instr}_{ARG}$ geschrieben, um den Bezug zu einer bestimmten Instruktion herzustellen. Tritt ein Pseudoregister als Ziel einer Instruktion auf, sodass eine neue Range erzeugt werden muss, wird diese in dem entsprechenden `RES`-Slot beginnen. Wenn eine Range endet, indem ihr zugehöriges Pseudoregister als Argument einer Instruktion auftritt, so verweist ihr Ende auf den `ARG`-Slot.

```
10: x := 10;
20: y := 2 * x;
```

Abbildung 3.11: Codeausschnitt für Ranges

Der in Abbildung 3.11 gezeigte Codeausschnitt führt, für die Pseudoregister `x` und `y`, zu den in Abbildung 3.12 gezeigten Intervallen.

So könnten die Pseudoregister `x` und `y` das gleiche physikalische Register zugewiesen bekommen, da es zu keiner Überlappung der beiden Ranges kommt.

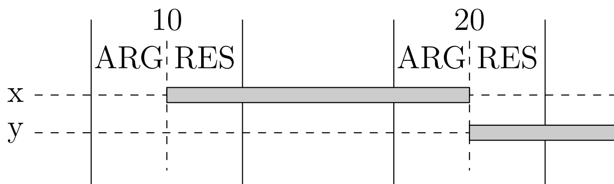

Abbildung 3.12: Einfache Ranges

Nach einer Lücke im Intervall, welche durch Basicblocks entstanden ist, in denen das Pseudoregister nicht live ist, tritt das Pseudoregister nicht zuerst als Ziel einer Instruktion auf und in der letzten Instruktion, vor einer solchen Lücke, wird es nicht zwangsläufig als Argument einer Instruktion verwendet. An den Grenzen solcher Lücken enden Ranges somit auf dem `RES`-Slot und sie beginnen auf dem `ARG`-Slot. Dies erweckt den Anschein, als wäre aus einem Intervall ein Bereich entfernt worden. Abbildung 3.13 zeigt dies schematisch.

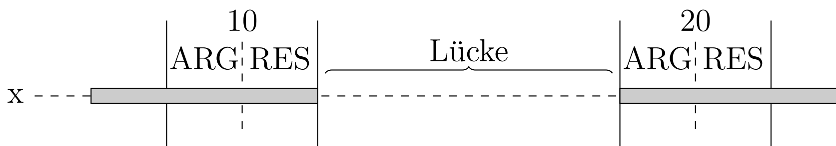

Abbildung 3.13: Range mit einer Lücke

3.3.2 Der Instruktionsbaum

Wie bereits zuvor erwähnt, ist es, aufgrund des Ablaufs und der Struktur des `OCaml-Codes`, nicht möglich, die Intervalle in einem Pass über den linearisierten Code zu erzeugen. Die Eingabe der Intervallerzeugung ist ein Instruk–tionsbaum des zu compilierenden Programms und dieser muss in der gleichen Weise traversiert werden, wie dies später während der Linearisierungsphase erfolgt, sodass die relative Ordnung der Instruktionen zueinander erhalten bleibt.

Jede Instruktion besteht aus einer Instruktionsbeschreibung, den Pseudoregistern, welche in dieser Instruktion verwendet werden und einer nachfolgenden Instruktion. Dies bildet eine einfach verkettete Liste von Instruktionen, welche eine lineare Instruktionsfolge bildet. Jede Instruktionsfolge endet auf einer definierten Instruktion mit der `Iend`-Beschreibung. Die vollständige Beschreibung des Instruktionsbaums ist in der Datei `asmcomp/mach.ml` zu finden; Abbildung 3.14 zeigt eine vereinfachte Darstellung.

Der Typ der Instruktion wird durch die `description` spezifiziert. Jeder

struct *instruction*
`description` : *instruction_description*
`arg, res, live` : *pseudoregister list*
`next` : *instruction*

Abbildung 3.14: Beschreibung des Instruktionsbaums

Instruktionstyp kann wiederum beliebig viele *Subinstruktionen* haben, wodurch sich eine Baumstruktur bildet. So hat z.B. der Typ `Iifthenelse` jeweils für den `then`-Teil und den `else`-Teil eine *Subinstruktion* und damit eine *Subinstruktionsfolge*. Die Instruktionstypen, welche für die Baumstruktur des Instruktionsbaums verantwortlich sind, sind in Abbildung 3.15 angegeben.

enum *instruction_description*
`Iend`
`Iifthenelse` : *instruction* * *instruction*
`Iswitch` : *instruction list*
`Iloop` : *instruction*
`Icatch` : *instruction* * *instruction*
`Itrywith` : *instruction* * *instruction*

Abbildung 3.15: Instruktionsbeschreibung

- `Iifthenelse` - Hier existieren zwei Teilfolgen, jeweils eine für den Then-Block und den Else-Block.
- `Iswitch` - Hier existieren beliebe Teilfolgen. Für jeden Case-Block eine.
- `Iloop` - Es existiert genau eine Teilfolge für den Loop-Körper.
- `Icatch` - Es existiert eine Teilfolge für den ausgeführten Code und eine für den Handler, welcher im Fehlerfall ausgeführt wird.
- `Itrywith` - Analog zu `Icatch`.

Alle Teilfolgen dieser fünf Instruktionen haben in sich eine Ordnung, so wird z.B. bei `Iifthenelse`-Konstrukten erst der Then-Block untersucht und dann der Else-Block. Diese Ordnung wird bei der Traversierung des Baums in Depth-First-Order beachtet und beibehalten. Alle Instruktionen werden, beginnend bei `1`, in der Reihenfolge, in der sie besucht werden, durchnummeriert. Diese Werte beschreiben den Index der Instruktion und bilden den Definitionsbreich der Intervalle. Abbildung 3.16 zeigt den Ablauf, wie der

Instruktionsbaum traversiert wird. Die in diesem Code verwendete Funk–tion `update_interval` ist für die Erzeugung und das Update der Intervalle zuständig und wird in Kapitel 3.3.3 beschrieben.

```
eval_instruction(instruction)
  foreach live in instruction.live
    update_interval live
  end for
  switch type of instruction
    case Iend:
      return
    case Iifthenelse:
      eval_instruction then
      eval_instruction else
    case Iswitch:
      for each case
        eval_instruction case
      end for
    case Iloop:
      eval_instruction body
    case Icatch:
      eval_instruction body
      eval_instruction handler
    case Itrywith:
      eval_instruction body
      eval_instruction handler
  end switch
  eval_instruction next
```

Abbildung 3.16: Traversieren des Syntaxbaums in Depth-First-Order

3.3.3 Erzeugung der Intervalle

Wie der Name Liveintervall bereits besagt, basieren die Ranges der Intervalle auf der Liveness der Pseudoregister, die sie repräsentieren. Die Live–nessanalyse wird, wie in Kapitel 3.2 beschrieben, vor der Registerallokation durchgeführt und die Livnessinformationen stehen bereits zur Verfügung. Der *OCaml-Compiler* unterscheidet zwischen drei Arten, auf denen ein Pseudoregister live sein kann.

- `arg` - Das Pseudoregister ist Argument der Instruktion.
- `res` - Das Pseudoregister ist Ergebnis (Resultat) der Instruktion.
- `live` - Das Pseudoregister ist weder Argument noch Ergebnis, jedoch trägt es Informationen, die in späteren Instruktionen als Argument benötigt werden.

Die Liveness eines Pseudoregisters [ALSU08] besagt, dass es einen Wert führt, welcher zu einem späteren Zeitpunkt unverändert benötigt wird. Da `arg` ebenfalls auf diese Definition zutrifft, ist `arg` lediglich ein Sonderfall von `live`, insofern, dass die Information des Pseudoregister in der aktuellen Instruktion benötigt wird. Obwohl `arg` und `live` vom Gedankengang ähnlich sind, müssen sie dennoch gesondert betrachtet werden, da die Grenzen der Ranges auf unterschiedliche Slots fallen. Eine Range eines Pseudoregisters beschreibt eine nicht unterbrochene Folge von Instruktionen, mit den Indizes `[begin, end]`, in denen dieses live ist. Eine Unterbrechung der Liveness einer solchen Folge von Instruktionen hat zur Folge, dass eine Range beendet wird und eine neue Range erzeugt wird. Für jedes Pseudoregister aus `arg`, `live` und `res` einer Instruktion werden folgende Unterscheidungen durchgeführt:

1. Wenn für das Pseudoregister noch kein Intervall existiert, wird eins angelegt. Ein solches Pseudoregister wird also zum ersten mal im Code verwendet und muss somit als `res` einer Instruktion auftreten. Die Grenzen des Intervalls, und somit auch die Grenzen der initialen Range, werden mit [$\mathtt{instr}_{RES}$, $\mathtt{instr}_{RES}$] initialisiert.

2. Bei bereits bestehenden Intervallen muss entschieden werden, ob die letzte Range auf die aktuelle Instruktion erweitert werden kann, oder ob, aufgrund einer Unterbrechung der Liveness, eine neue Range angelegt werden muss. Bei Erweiterung muss nur das Ende der letzten Range und des Intervalls angepasst werden.

    - Tritt das Pseudoregister als `arg` auf, so wird das Ende dieser Range auf den `ARG`-Slot der aktuellen Instruktion gesetzt. Die Range und das Intervall wird also auf [..., $\mathtt{instr}_{ARG}$][3] geändert.
    - Tritt das Pseudoregister jedoch als `res` oder `live` auf, so wird das Ende der Range auf den `RES`-Slot der aktuellen Instruktion gesetzt. Die Range und das Intervall werden also auf [..., $\mathtt{instr}_{RES}$] geändert.

[3]Die ...-Schreibweise besagt, dass diese Grenze der Range oder des Intervalls unverändert übernommen wird.

Bei der Erzeugung einer neuen Range muss sowohl der Anfang als auch das Ende gesetzt werden. Das Ende der Range und des Intervalls wird nach den gleichen Kriterien wie bei der Erweiterung gesetzt. Der Anfang wird folgendermaßen gesetzt:

- Tritt das Pseudoregister als `arg` oder `live` auf, so wird der Anfang der neuen Range auf den `ARG`-Slot der aktuellen Instruktion gesetzt. Die Range wird also auf [instr$_{ARG}$, ...] initialisiert.
- Tritt das Pseudoregister als `res` auf, so wird der Anfang der neuen Range auf den `RES`-Slot der aktuellen Instruktion gesetzt. Die Range wird also auf [instr$_{RES}$, ...] initialisiert.

3.3.4 Fixe Intervalle

Wenn *OCaml-Code* übersetzt wird, so werden während des Compileprozesses für alle Variablen, Zwischenergebnisse, etc. des *OCaml-Codes* Pseudoregister angelegt. Diesen Pseudoregistern wurde bis zu diesem Zeitpunkt noch kein physikalisches Register zugewiesen. Darin besteht die Aufgabe des Registerallokators. Diverse Operationen verlangen jedoch, dass ihre Argumente in definierten physikalischen Registern stehen. So muss z.B. auf der *i386-Architektur* bei `Idiv` und `Imod` der erste Operand in `eax` und der zweite in `ecx` stehen, oder bei `Iextcall` (*C-Calls*) besagt das *System V ABI für die x86-Architektur* [San97], welches von nahezu jedem Betriebssystem[4] implementiert wird, welches auf einem *x86* Prozessor läuft, dass der Rückgabewert eines solchen Calls in Register `eax` steht.

Um zu gewährleisten, dass diese Werte auch wirklich in diesen Registern stehen, oder dass diese Werte aus den richtigen Registern gelesen werden, gibt es die fixen Pseudoregister. Diese bilden ein festes Mapping von Pseudoregistern auf physikalische Register. Ist es also nötig, dass zu einer Instruktion der Wert eines Pseudoregisters in einem bestimmten physikalischen Register steht, so werden bereits in der Selektion `Imov`-Instruktionen eingefügt um die korrekten Zuweisungen zwischen den variablen und den fixen Pseudoregistern durchzuführen. Abbildung 3.17 zeigt, wie diese Zuweisungen bei dem Ausdruck `x = 10 / 3` stattfinden.

Obwohl den fixen Pseudoregistern bereits ein Register zugewiesen ist, werden für diese dennoch Intervalle angelegt, denn sie bilden Ranges, in denen andere Pseudoregister nicht das gleiche physikalische Register zugewiesen bekommen dürfen. Dies spielt im nächsten Kapitel 3.3.5 eine entscheidende Rolle.

[4]Mit Ausnahme von *Win32*, welches ein anderes ABI implementiert

```
Imov 10, i0
Imov i0, eax
Imov 3, i1
Imov i1, ecx
Idiv ecx        // eax = 3, edx = 1
Imov eax, x
```

Abbildung 3.17: Instruktionsfolge für `x = 10 / 3`

3.3.5 Zerstörte Register

So wie einige Instruktionen verlangen, dass ihre Operanden in definierten Registern stehen, so existieren Instruktionen, die die Werte von bestimmten Registern überschreiben. So werden z.B. auf der *i386*-Architektur bei der Instruktion `Idiv` die beiden Register `eax` und `edx` überschrieben [Int11]. Für Funktionsaufrufe legen die Calling-Conventions fest, welche Register überschrieben werden. Dies wird bei `Iextcall` (*C-Calls*) durch das *System V ABI* und bei *OCaml-Calls* durch die *OCaml-Calling-Conventions* spezifiziert. Da die Letztgenannten jedoch keine Callee-Save Register kennen, werden alle Register als überschrieben angenommen. Die genauen Informationen, welche Register von welcher Instruktion überschrieben werden, ist architekturabhängig. Die genauen Angaben sind in der Datei `asmcomp/proc.ml` zu finden.

Ein Registerallokator muss den Aspekt berücksichtigen, dass Pseudoregistern keine physikalischen Register zugewiesen werden, welche durch solche Instruktionen überschrieben werden. Um dies mit den Mitteln des *Linear-Scan-Algorithmus* zu lösen, werden punktuelle Ranges in den fixen Intervallen eingebaut. Bei jeder Instruktion, welche Register zerstört, werden in den entsprechenden fixen Intervallen eine Range eingeführt, deren `begin` und `end` auf den `RES`-Slot der Instruktion verweist. Somit ergibt sich eine Range `[instr`$_{RES}$`, instr`$_{RES}$`]`. Der Algorithmus, welcher in Kapitel 3.4 beschrieben wird, berücksichtigt die fixen Intervalle in der Form, dass keinem Intervall ein physikalisches Register zugewiesen wird, welches eine Überlappung mit dem zugehörigen fixen Intervall hat.

Da ein *OCaml-Call* (kein *C-Call*) keine Callee-Save Register kennt und somit auf allen Plattformen alle Register zerstört, kann es keine Intervalle geben, denen ein physikalisches Register zugewiesen wird, welches sich über einen solchen Call erstreckt. Alle Pseudoregister dieser Intervalle müssen gespillt werden. Da dies jedoch nicht die Arbeit eines Registerallokators benötigt, werden solche Präallokationen bereits, wie in Kapitel 3.2 beschrieben, in der Spill- und Split-Phase durchgeführt. Da diese Pseudoregister dort

bereits als gespillt markiert wurden, obliegt es dem Registerallokator nur noch einen Stackslot zuzuweisen.

3.3.6 Alternative Intervalldarstellung

Diese Darstellung der Intervalle, wie sie sich nun final durchgesetzt hat, ist eine von mehreren Lösungen, welche während der Implementierung entstanden sind. Von diesen entstandenen Intervalldarstellungen soll an dieser Stelle eine näher beschrieben werden, denn sie hat sich über die meiste Zeit der Entwicklung als praktikabel und effizient durchgesetzt. Letztlich hat sie jedoch eine Besonderheit der *ARM-Architektur* als problematisch entlarvt.

Diese Darstellung der Intervalle wurde durch die Arbeit [Wim04] von Christian Wimmer an dem *Java HotSpot*™ *Client Compiler* inspiriert. Hier wird das Intervallende nicht inklusiv `[begin, end]` sondern exklusiv `[begin, end[` betrachtet. Dafür entfällt jedoch die Trennung der Indizes in zwei Slots. Betrachtet man also den Codeausschnitt, wie in Abbildung 3.18, so führte dies zu den Intervallen, wie Abbildung 3.19 gezeigt.

```
10: x := 10;
20: y := 2 * x;
```

Abbildung 3.18: Codeausschnitt für Ranges

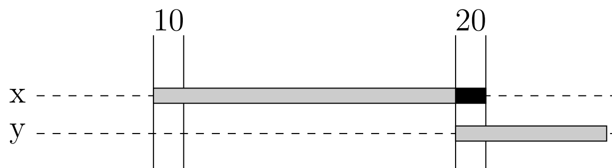

Abbildung 3.19: Alternative Darstellung der Intervallen mit exklusivem Intervallende

Die Darstellung des Intervallendes in schwarz soll anzeigen, dass dieses Intervallende exklusiv ist. Auch hier kann den beiden Pseudoregistern `x` und `y` das gleiche physikalische Register zugewiesen werden. Nun existiert jedoch die Besonderheit der *ARM-Architektur* für die Instruktion `mul Rd, Rm, Rs`, dass der Operand `Rm` nicht durch das gleiche Register gegeben werden kann, wie das Ergebnis `Rd`. Um dies durch den Registerallokator zu gewährleisten, wird während der Selektion das Pseudoregister für `Rm` ebenfalls in die Ergebnisliste `res` aufgenommen. Das Intervall für `Rm` hat nun zwei Ranges[5]. Eine

[5]Effektiv ist dies natürlich nur eine Range

Range für die Funktionalität als Operand `[x, y[` und eine für die Funktionalität als zweites Ergebnis `[y, y[`. Das Intervall für `Rd` führt zu einem Intervall `[y, z[`. Aufgrund der exklusiven Intervalldarstellung existiert keine Überlappung dieser Intervalle. Abbildung 3.20 zeigt diese Situation schematisch.

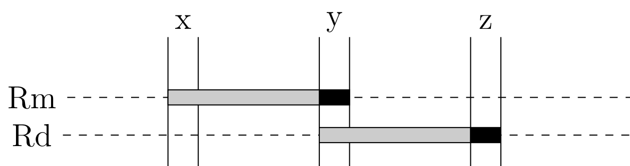

Abbildung 3.20: Problematische Intervalle einer `mul`-Instruktion auf der *ARM-Architektur*

Da es zu keiner Überlappung kommt, können `Rd` und `Rm` das gleiche physikalische Register zugewiesen bekommen. Jedoch ist dies keine gültige Instruktion der *ARM-Architektur*. Dieses Problem führte letztlich zu der geänderten Intervalldarstellung, wie sie nun existiert.

3.4 Der Linear-Scan-Algorithmus

Nachdem in Kapitel 3.3 beschrieben wurde, wie aus dem `OCaml-Code` Intervalle erzeugt werden, soll in diesem Kapitel nun beschrieben werden, wie diese Intervalle genutzt werden, um mit ihnen eine Zuweisung von Pseudoregistern zu physikalischen Registern zu finden. In der Datei `asmcomp/linscan.ml` ist die Implementierung des hier beschriebenen Algorithmus zu finden. Die im Folgenden beschriebenen Zuweisungen eines physikalischen Registers an ein Intervall sind gleichbedeutend mit der Zuweisung eines physikalischen Registers an das Pseudoregister, welches durch das Intervall repräsentiert wird.

Die Ausführung des *Linear-Scan-Algorithmus* erfolgt, wie der Name bereits sagt, streng linear. In einem linearen Pass über alle Intervalle wird diesen ein physikalisches Register oder ein Stackslot zugewiesen.

Die Intervalle, welche die Eingabe für den Algorithmus bilden, werden zu Beginn in chronologischer Reihenfolge aufsteigend nach Index `begin`[6] sortiert. In jedem Schritt wird ein Intervall bearbeitet, wobei der Index `begin` des Intervalls den aktuellen Index darstellt. Durch die Sortierung ergibt sich ein monotones Ansteigen des aktuellen Indexes. Dies ist wichtig, da ein Intervall, welches einmal abgelaufen ist, nicht wieder aktiv werden kann, wobei ein

[6]Die Indizes `begin` und `end` sind die `begin`- und `end`-Werte der `interval` Struktur wie in Kapitel 3.3.1 beschrieben.

Intervall, dessen Index `end` kleiner ist als der aktuelle Index, als abgelaufen gilt.

3.4.1 Abarbeiten der Intervalle

Neben der Liste der Intervalle wird in dieser Implementierung mit drei weiteren Intervalllisten gearbeitet. Diese drei Intervalllisten sind die folgenden:

- `active` - Sie enthält alle Intervalle, die eine Überschneidung mit dem aktuellen Index haben.
- `inactive` - Sie enthält alle Intervalle, die noch nicht abgelaufen sind, aber keine Überschneidung mit dem aktuellen Index haben. Sie haben also über dem aktuellen Index eine Lücke.
- `fixed` - Sie enthält alle noch nicht abgelaufenen fixen Intervalle aus Kapitel 3.3.4.

Jedes Pseudoregister gehört einer Registerklasse an. Diese Klassen sind üblicherweise:

- *Integer*
- *Float*

Eine Ausnahme bildet hier die *ARM-Architektur*, welche nur eine Integer-Registerklasse kennt. Floats werden durch paarweise Verwendung von Integer-Registern ermöglicht.

Die drei Intervalllisten `active`, `inactive` und `fixed` werden separat für alle Registerklassen gehalten. Im Folgenden wird nicht explizit auf die Nutzung der Intervalllisten in unterschiedlichen Klassen hingewiesen, sondern es werden immer die Intervalllisten zu der entsprechenden Klasse des aktuell bearbeiteten Intervalls betrachtet.

Zu Beginn der Abarbeitung sind die beiden Listen `active` und `inactive` unbefüllt, während die Liste `fixed` mit allen fixen Intervallen vorbelegt ist. Für jedes Intervall der Eingabe wird die Funktion `handle_interval` aus Abbildung 3.21 ausgeführt, welche für das Intervall ein geeignetes physikalisches Register oder einen geeigneten Stackslot findet.

In jedem Schritt werden alle Intervalle aus `active`, `inactive` und `fixed` entfernt, welche zu dem aktuellen Index bereits abgelaufen sind. Diese Intervalle spielen für die weitere Registervergabe keine Rolle mehr, denn alle zukünftigen Intervalle können keine Überschneidung mehr mit diesen Intervallen haben. Sie können somit entfernt werden.

Des Weiteren wird für alle Intervalle aus `active` geprüft, ob sie mit dem aktuellen Index eine Überschneidung haben. Wenn zu einem Intervall keine Überschneidung vorliegt, so ist das physikalische Register, welches diesem Intervall zugewiesen ist, für den aktuellen Index noch für weitere Zuweisungen frei. Das Intervall wird aus `active` entfernt und in `inactive` eingefügt.

Wie in `active` wird auch in `inactive` für alle Intervalle geprüft, ob sie eine Überschneidung mit dem aktuellen Index haben. Wenn für ein Intervall eine solche Überschneidung vorliegt, so steht das physikalische Register, welches diesem Intervall zugewiesen ist, für weitere Zuweisungen nicht mehr zur Verfügung. Somit wird dieses Intervall aus `inactive` wieder entfernt und erneut in `active` aufgenommen.

Nach diesen Vorbereitungen beinhaltet `active` exakt die Intervalle, denen zu dem aktuellen Index ein physikalisches Register zugewiesen ist und `inactive` alle nicht abgelaufenen Intervalle, die zum aktuellen Index nicht live sind.

Im Anschluss wird mit der Funktion `try_allocate_free_register` versucht, dem Intervall ein physikalisches Register zuzuweisen. Ist dies nicht möglich, existiert also zu dem aktuellen Index kein weiteres freies physikalisches Register mehr, so wird mit der Funktion `allocate_blocked_register` entweder versucht ein physikalisches Register frei zu machen oder einen Stack–slot zu wählen. Diese beiden Funktionen werden in den Kapiteln 3.4.2 und 3.4.3 beschrieben.

3.4.2 Zuweisung eines freien physikalischen Registers

Die Funktion `try_allocate_free_register` versucht dem aktuellen Intervall ein physikalisches Register zuzuweisen, welches zu dem aktuellen Index keinem anderen Intervall zugewiesen ist. Zusätzlich muss geprüft werden, ob es zwischen dem aktuellen Intervall und den Intervallen, denen bereits ein Register zugewiesen wurde, eine Überschneidung gibt. Die physikalischen Register, die diesen überlappenden Intervallen zugewiesen sind, müssen natürlich aus dem Pool der potentiellen physikalischen Register ausgeschlossen werden. Abbildung 3.22 zeigt den Ablauf, wie diese Zuweisung durchgeführt wird.

Um ein freies physikalisches Register zu finden, wird als erstes eine Liste `free_registers` erzeugt, welche alle physikalischen Register enthält, die generell zur Vergabe vorhanden sind. Es werden nun sukzessive alle Register ausgeschlossen, die nicht mehr zur Vergabe bereit stehen.

Alle Intervalle, welche sich in der Liste `active` befinden, haben zu dem aktuellen Index ein physikalisches Register zugewiesen. Diese Register stehen für das aktuelle Intervall nicht mehr zur Verfügung und müssen demnach ausgeschlossen werden. Sie werden aus der Liste `free_registers` entfernt.

```
handle_interval interval
  current_pos = interval.begin
  for each i in active do
    if i is expired then
      remove i from active
    else if i is not live on current_pos then
      move i to inactive
  end for
  for each i in inactive do
    if i is live on current_pos then
      move i to active
  end for
  for each i in fixed do
    if i is expired then
      remove i from fixed
  end for

  if try_allocate_free_register failed then
    allocate_blocked_register
```

Abbildung 3.21: Abarbeiten der Intervalle

Die Liste `inactive` beinhaltet alle nicht abgelaufenen Intervalle, denen bereits ein physikalisches Register zugewiesen wurde, die aber keine Überschneidung mit dem aktuellen Zeitpunkt besitzen. Bei diesen Intervallen muss explizit überprüft werden, ob es eine Überschneidung mit dem aktuellen Intervall gibt. Wenn Überschneidungen vorliegen, so stehen die diesen Intervallen zugewiesenen physikalischen Register nicht zur Verfügung und müssen ausgeschlossen werden. Auch diese Register werden aus der `free_registers` Liste enfernt.

Mit den Intervallen in der Liste `fixed` verhält es sich genauso wie mit den Intervallen aus der Liste `inactive`. Sie enthält alle vorallozierten Intervalle und die punktuellen Intervalle, die durch registerzerstörende Instruktionen entstanden sind, wie dies in Kapitel 3.3.5 beschrieben wurde. Es muss getestet werden, ob eine Überschneidung mit dem aktuellen Intervall vorliegt. Wenn eine solche Überschneidung vorliegt, muss das entsprechende physikalische Register aus dem Pool potentieller Register entfernt werden.

Nachdem nun alle physikalischen Register aus der Liste `free_registers` ausgeschlossen wurden, die aufgrund von Überlappungen mit dem aktuellen Intervall nicht zur Verfügung stehen, kann eines der noch verbleibenden

```
try_allocate_free_register interval
  free_registers = all physical registers
  for each i in active do
    remove i.reg from free_registers
  end for
  for each i in inactive ∪ fixed do
    if i overlapping interval then
      remove i.reg from free_register
  end for
  if free_registers is empty then
    fail
  else
    interval.reg = first register from free_registers
    add interval to active
```

Abbildung 3.22: Zuweisung eines freien Registers

physikalischen Register gewählt werden. Ist die Liste free_registers leer, steht kein freies Register mehr zur Verfügung und es muss auf einem anderen Weg versucht werden, eine Zuweisung zu erzeugen. Dies wird in Kapitel 3.4.3 beschrieben.

Ist die Liste free_registers nicht leer, so wird das erste freie physikalische Register aus dieser Liste dem aktuellen Intervall zugewiesen. Das aktuelle Intervall kann nun in die Liste active aufgenommen werden. Die Liste active ist nach dem Endzeitpunkt ihrer Intervalle sortiert. Somit muss sie von vorne beginnend durchlaufen werden, um die korrekte Position zu finden, in welcher das aktuelle Intervall eingefügt werden kann.

3.4.3 Zuweisung bei blockierten Registern

Wenn kein weiteres freies Register für das aktuelle Intervall gefunden werden kann, so muss das Pseudoregister eines Intervalls gespillt werden. Die Methode allocate_blocked_register aus Abbildung 3.23 trifft genau die Entscheidungm, welches Intervall zum spillen ausgewählt wird.

Die Entscheidung, welches Intervall gespillt wird, ist leicht getroffen. Das Intervall, welches als letztes endet, wird zum spillen ausgewählt. Die Liste active ist chronologisch nach Endzeitpunkt ihrer Intervalle sortiert, was das Auffinden des letzten Intervalls sehr schnell gestaltet. Dieses letzte Intervall latest_active aus active wird nun mit dem aktuellen Intervall verglichen. Endet das Intervall latest_active später als das aktuelle Intervall, d.h. der

```
allocate_blocked_register interval
  latest_active = the last interval within active
  if latest_active.end > interval.end then
    interval.reg = latest_active.reg
    remove latest_active from active
    add interval to active
    assign a stackslot to latest_active
  else
    assign a stackslot to interval
```

Abbildung 3.23: Ermittlung des zu spillenden Registers

Endindex von `latest_active` ist größer als der Endindex des aktuellen Intervalls, so wird das physikalische Register, welches `latest_active` zugewiesen ist, auf das aktuelle Intervall übertragen. Im Anschluss wird das Intervall `latest_active` aus der Liste `active` entfernt um Platz für das aktuelle Intervall zu machen, welches in `active` an der richtigen Stelle eingefügt wird. Als letztes wird dem aus `active` entfernten Intervall `latest_active` ein Stackslot zugewiesen, in dem der Wert während der Programmausführung zu finden sein wird.

Endet das aktuelle Intervall später als `latest_active`, so muss dem aktuellen Intervall ein entsprechender Stackslot zugewiesen werden.

3.4.4 Gespillte Pseudoregister

Der Wert eines Pseudoregister kann sich während der Ausführung an zwei unterschiedlichen Stellen befinden. Zum Einen kann er sich in einem physikalischen Register befinden und zum Anderen kann er im Arbeitsspeicher, genauer auf dem Stack, liegen. Den Pseudoregistern, welche während der Registerallokation gespillt wurden, wurde ein sogenannter Stackslot zugewiesen. Ein solcher Slot ist ein Integerwert und beschreibt einen Offset bezogen zu dem Stackpointer, z.B. dem Register `esp` auf *i386-Architekturen* oder `rsp` auf *amd64-Architekturen*.

Die Anzahl `num_stack_slots` der benötigten Stackslots, also die Anzahl der gespillten Pseudoregister innerhalb einer Funktion, gibt die Größe[7] des benötigten Stackbereichs an. Beim Eintreten in eine Funktion wird der Stackpointer um die benötigte Größe dekrementiert[8]. Die folgenden Angaben be-

[7]Ein Stackslot umfasst auf 32Bit Architekturen 4 Byte auf 64Bit Architekturen 8Byte. Ein Stackslot für Floats umfasst jedoch immer 8 Byte.

[8]Der Stack wächst von oben nach unten deswegen wird der benötigte Speicherbereich

ziehen sich auf die *amd64-Architektur*. Der Bereich zwischen `rsp` und `rsp + num_stack_slots * 8` ist unbelegt und kann für die gespillten Pseudoregister verwendet werden. Ein Pseudoregister, dem der Stackslot `m` mit `n = 8*m` zugewiesen wurde, kann nun mit dem Speicherzugriff `n(%rsp)`[9] angesprochen und in diversen Instruktionen als Speicheroperand verwendet werden. An jeder Stelle, an welcher der Scope der Funktion verlassen wird, wird der Stackpointer um den zuvor dekrementierten Wert wieder inkrementiert. Der Stackpointer beinhaltet dann wieder den Wert, den er vor dem Eintritt in die Funktion führte. Der Aufbau eines solchen Stackslotkonstrukts ist in Abbildung 3.24 zu sehen.

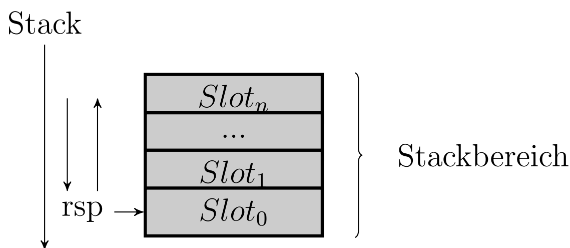

Abbildung 3.24: Einfacher Aufbau von Stackslots

Da bei rekursiven Funktionsaufrufen der Scope der aufrufenden Funktion nicht verlassen wird, wächst der Stackbereich bei jeder Rekursionstiefe um den benötigten Speicherbereich. Wenn die Rekursion zu einem Ende kommt, wird der Stackpoint in jedem Scope, der verlassen wird, um den entsprechenden Wert wieder inkrementiert und der Stack wird somit wieder in seinen ursprünglichen Zustand zurückgesetzt. Dies ist in Abbildung 3.25 zu sehen.

3.4.5 Lösen von ungültigen Instruktionsargumenten

Der *Linear-Scan-Algorithmus* arbeitet auf einer unabhängigen Liste von Liveintervallen, die nichts über die Instruktionen, aus denen sie entstanden sind, aussagen. Somit kann keine Logik angewendet werden, die bestimmten Pseudoregistern bestimmte physikalische Register zuweist, oder die besagt: Pseudoregister `X` darf gespillt werden, Pseudoregister `Y` muss in einem Register stehen. Aus diesem Grund können Operandenkonstellationen auftreten, die auf bestimmten Architekturen nicht erlaubt sind. So ist es z.B. in den *amd64-* oder *i386-Architekturen* für viele Instruktionen gestattet, dass ein Operand im Speicher stehen darf, aufgrund des Aufbaus des Instruktionssets

---

von dem Stackpointer subtrahiert.

[9] `n(%rsp)` in *AT&T-Syntax* und `[rsp + n]` in *Intel-Syntax*.

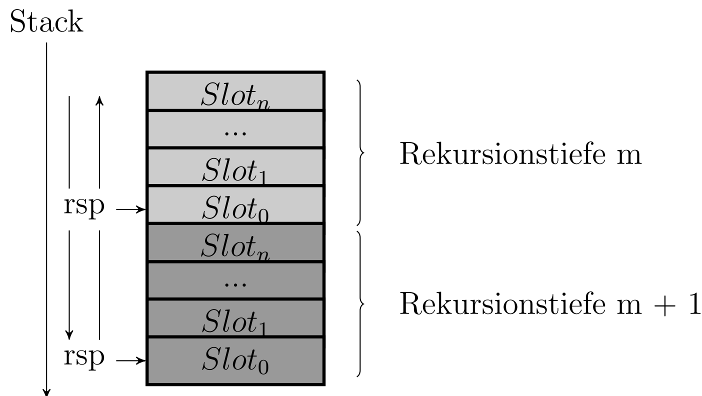

Abbildung 3.25: Aufbau der Stackslots bei rekursiven Aufrufen

jedoch niemals *mehr* als einer. Bei Load/Store-Architekturen wie *MIPS*, *PPC* oder *ARM* dürfen Speicheroperanden ausschließlich in `load`- und `store`-Instruktionen auftreten dürfen. Jedoch kann genau dies, wie grade erwähnt, nicht garantiert werden.

Für dieses Problem bietet der Code des *OCaml-Compilers* bereits die Lösung, denn vor dem gleichen Problem steht auch die bestehende Registerallokation, die auf einem *Graph-Coloring-Algorithmus* basiert.

Nach der Registerallokationsphase wird ein Reload des Codes durchgeführt, welcher solche ungültigen Stellen auffindet und entsprechenden Code einfügt um diese Probleme zu lösen. Durch diese Veränderungen an dem Code sind nun aber auch die zugewiesenen Register verloren gegangen, wodurch die Registerallokation erneut durchgeführt werden muss.

Diese Schleife läuft so lange, bis entweder kein weiterer Code mehr eingefügt werden muss, wodurch dann auch die Registerzuweisung nicht verloren geht, oder bis eine maximale Obergrenze an Durchläufen erreicht ist. In letzterem Fall wird der Compiler den Compileprozess abbrechen.

Kapitel 4

Die native *OCaml-Toplevel*

Wer mit *OCaml* programmieren möchte, dem stehen zur Ausführung seines Codes mehrere Möglichkeiten und Tools zur Verfügung:

- `ocamlc` - Der *OCaml-Compiler*
- `ocamlrun` - Der *OCaml-Bytecode-Interpreter*
- `ocamlopt` - Der *OCaml-Native-Code-Compiler*
- `ocaml` - Die *OCaml-Toplevel*
- `ocamlnat` - Die native *OCaml-Toplevel*

Seit der Veröffentlichung von *Caml Light* 1991 besteht eine der Möglichkeiten der Ausführung von *OCaml-Code* darin, den *OCaml-Compiler* `ocamlc` dazu zu nutzen, den Code in Bytecode zu compilieren. Dieser kann dann durch den Bytecode-Interpreter `ocamlrun` ausgeführt werden.

Da die Ausführung, bzw. die Interpretation des Bytecodes nicht sehr effizient ist, wurde mit der Einführung von *Caml Special Light* 1992 eine Alternative angeboten. Der *OCaml-Native-Code-Compiler* `ocamlopt` kann dazu genutzt werden, den *OCaml-Code* in ein natives Binary zu compilieren, welches ohne die Hilfe von weiteren externen Programmen direkt auf Hardwareebene ausgeführt werden kann. Es werden hier zwar keine weiteren Programme benötigt, um das Binary auszuführen, es werden jedoch zusätzliche Programme benötigt, um es zu erzeugen. In Kapitel 4.1 wird näher auf dieses Problem eingegangen.

Diese beiden Möglichkeiten stellen jedoch ausschließlich Offlinemethoden dar, d.h. der Code wird durch einen Compiler einmalig in eine andere Form umgewandelt, welche zu späteren Zeitpunkten beliebig oft ausgeführt werden kann. Die *OCaml-Toplevel* `ocaml` stellt eine Onlinemethode dar, d.h. der

Code wird einmalig compiliert und direkt ausgeführt. Zusätzlich bietet `ocaml` eine interaktive Shell, in der sukzessiv *OCaml-Code* eingegeben werden kann, welcher augenblicklich durch die *OCaml-Toplevel* compiliert und ausgeführt wird.

So wie es zu der `ocamlc-ocamlrun`-Kombination eine native Alternative durch `ocamlopt` gibt, so gibt es zu der *OCaml-Toplevel* `ocaml` eine native Alternative durch `ocamlnat`. Die Ausführungszeit des Codes lässt sich durch Nutzung von `ocamlnat` gegenüber `ocaml` stark verbessern, jedoch kommen mit der nativen Alternative zwei Probleme:

1. Wie bei `ocamlopt` werden bei `ocamlnat` zusätzliche externe Programme zum compilieren benötigt, welche gegebenenfalls auf dem ausführenden System nicht vorhanden sind. Dies macht die Nutzung von `ocamlnat` nicht auf allen Systemen einfach möglich. In Kapitel 4.1 wird auf dieses Problem näher eingegangen.

2. Der native Compile-Workflow nutzt temporäre Dateien, welche von den externen Programmen weiterverarbeitet werden, welche ihrerseits wieder Dateien schreiben, welche erneut weiterverarbeitet werden. Dieser Schreib-Lese-Schreib-Lese-Prozess beansprucht derart viel Zeit, dass der Geschwindigkeitsvorteil, den die native Programmausführung bietet, für Programme mit kurzer Laufzeit zunichte gemacht wird.

4.1 Der Workflow

In diesem Abschnitt soll zunächst auf den Compile-Workflow eingegangen werden, wie dieser durch die bisherigen *OCaml-Tools* - `ocamlc`, `ocaml`, `ocamlopt` und `ocamlnat` - implementiert ist. In den Abschnitten 4.3 und 4.4 wird die Implementierung des `ocamlnatjit` erläutert. Diese Implementierung ist Bestandteil dieser Arbeit. In Abbildung 4.1 ist der Workflow dieser Tools (inkl. `ocamlnatjit`) grob skizziert.

4.1.1 Der Bytecode-Workflow

Grundsätzlich kann bei den *OCaml-Tools* zwischen Tools unterschieden werden, welche auf interpretiertem Bytecode und auf nativem Binärcode basieren. Der Workflow dieser Varianten beginnt, wie in Kapitel 3.2 beschrieben, mit dem Parsen des *ML-Codes* in einen abstrakten Syntaxbaum. Dieser Baum wird im Folgenden zunächst in einen getypten Baum und dann

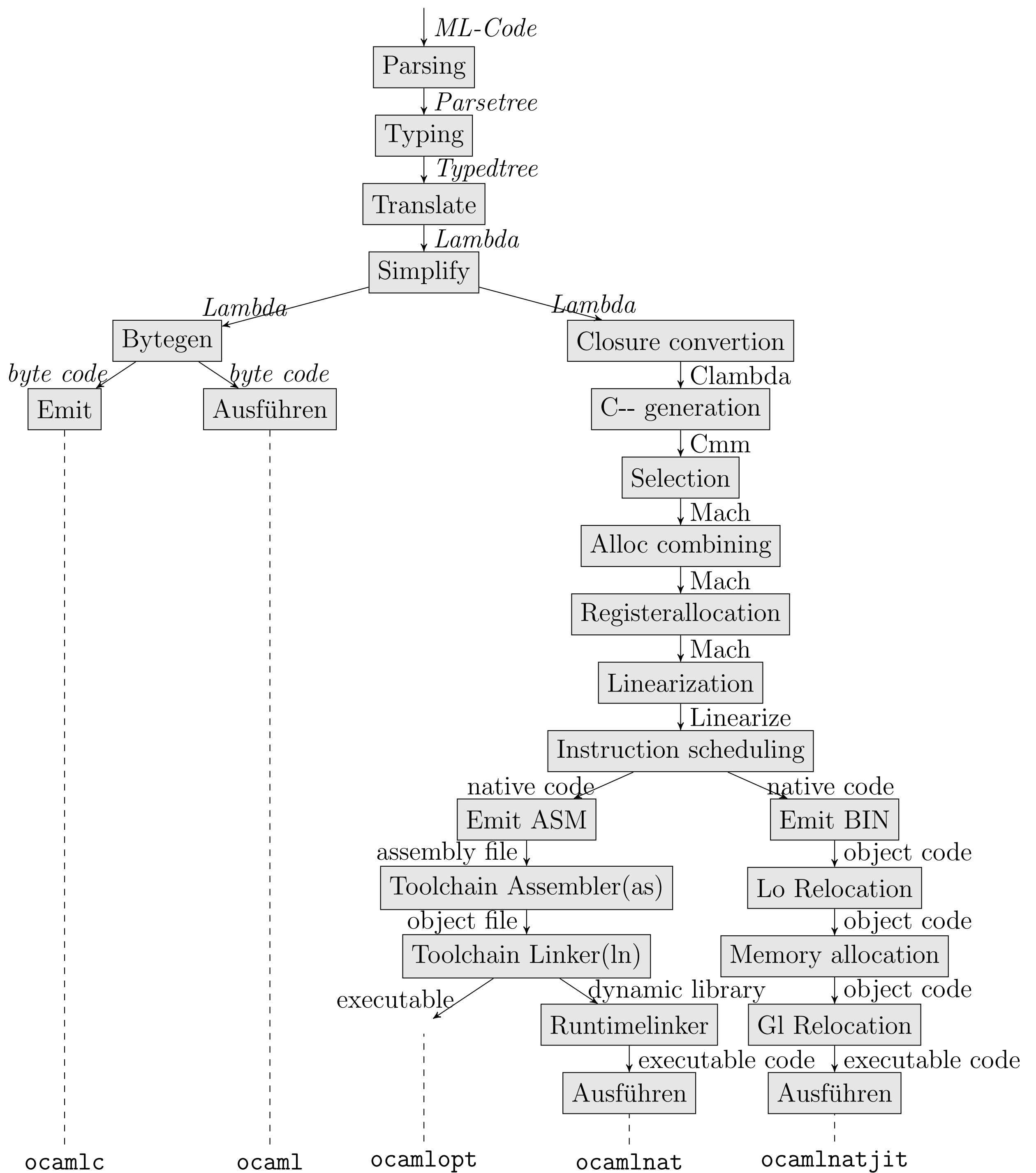

Abbildung 4.1: Workflow von `ocamlc`, `ocaml`, `ocamlopt`, `ocamlnat` und `ocamlnatjit`

in die bereits erwähnte Lambda-Baumrepräsentation transformiert. Die folgenden Schritte unterscheiden sich zwischen den Bytecode- und den nativen Binärcodevarianten. Sowohl für `ocamlc` und `ocaml` wird diese Lambda-Baumrepräsentation in eine Folge Bytecode-Instruktionen transformiert, welche wiederum optimiert werden. Aus dem in Kapitel 3.2 bereits erwähnten Beispielausdruck `let x=1 in x+3` würde eine Bytecodeinstruktionsfolge wie in Abbildung 4.2 erzeugt.

```
const 1
push
acc 0
offsetint 3
```

Abbildung 4.2: Bytecodeinstructionsfolge des Ausdrucks `let x=1 in x+3`

Im Falle des *OCaml-Compilers* `ocamlc` wird diese Bytecode-Instruktions–folge in eine Datei gespeichert, welche nun mit dem Tool `ocamlrun` ausgeführt werden kann. Diese Folge von einzelnen Prozessschritten ist in Abbildung 4.1 durch den ersten Pfad dargestellt, welcher mit `ocamlc` markiert ist.

Die *OCaml-Toplevel* `ocaml` behält diese Folge von Bytecode-Instruktionen im Speicher, um sie direkt interpretiert auszuführen. Diese Folge von einzelnen Prozessschritten ist in Abbildung 4.1 durch den zweiten Pfad dargestellt, welcher mit `ocaml` markiert ist.

4.1.2 Der native Workflow

Die ersten Prozessschritte des nativen Workflows für die Tools `ocamlopt` und `ocamlnat` sind bis zu der Erzeugung der Lambdabaumrepräsentation identisch mit der Bytecode-Variante. Die folgenden Schritte, von der Selektion bis hin zum Emitting der Mnemonics der Assemblersprache, wurden bereits in Kapitel 3.2 beschrieben. Diese im Emit-Schritt erzeugten Mnemonics werden in den aktuellen nativen *OCaml-Tools* in eine Datei gespeichert, welche im Anschluss weiterverarbeitet wird.

Für diese Verarbeitung in den folgenden Schritten werden externe Programme benötigt, welche nicht durch *OCaml* bereitgestellt werden. Diese benötigten Entwicklungsprogramme, die Toolchain, gehören bei den meisten *Unix/Linux* basierten Betriebssystemen durch die *GNU Binutils* [bin11] mit zum Distributionsumfang, bzw. sind leicht nachinstallierbar. Unter *Windows-Systemen* gibt es die Möglichkeiten, entweder die Entwicklungstools von *MinGW* [min11] oder *Cygwin* [cyg11] zu installieren, oder das *Microsoft Windows SDK for Visual Studio*, welches jedoch durch Lizenzen beschränkt ist.

All diese möglichen Optionen gehören unter *Windows* weder zum Installationsumfang dazu noch lassen sie sich einfach einrichten. Die Beschreibungen im Folgenden beziehen sich auf das Vorhandensein der *GNU Binutils*.

Mit dem Assembler-Compiler `as` können die erzeugten Dateien im nächsten Schritt zu Object-Files compiliert werden. Der `ocamlopt` verwendet im Anschluss den Linker `ld`, um aus diesen Object-Files ausführbare Programme zu erzeugen. Diese Programme können nun ohne die Zuhilfenahme weiterer Programme ausgeführt werden. Diese Folge von Prozessschritten ist in Abbildung 4.1 durch den dritten Pfad dargestellt, welcher mit `ocamlopt` markiert ist.

Der `ocamlnat` verwendet den Linker `ld`, um aus den compilierten Object-Files eine Shared-Library zu erzeugen. Um das Programm, welches in dieser Bibliothek enthalten ist, auszuführen, verwendet der `ocamlnat`-Prozess den Runtime-Linker, um diese Bibliothek in den Prozess zu laden. Jede Bibliothek, die über diesen nativen Workflow erzeugt wurde, enthält mehrere definierte Symbole, welche von der Bibliothek exportiert werden, um in den `ocamlnat`-Prozess eingebunden zu werden. Eins dieser Symbole ist das `entry`-Symbol, welches den Haupteinsprungspunkt für das enthaltene Programm darstellt und durch den `ocamlnat` geladen und ausgeführt wird. Wird durch diese Bibliothek lediglich eine Funktion bereitgestellt, ohne dass tatsächlich Programmcode ausgeführt werden muss, so wird dieses Symbol dennoch exportiert, es verweist jedoch auf eine Programmcodestelle, welche ohne Funktionalität hinterlegt ist. Diese Folge von Prozessschritten ist in Abbildung 4.1 durch den vierten Pfad dargestellt, welcher mit `ocamlnat` markiert ist.

4.2 Motivation

Es ist leicht ersichtlich, dass der Workflow, wie in Kapitel 4.1.2 für den `ocamlnat` beschrieben, nicht effizient arbeiten kann, denn `ocamlnat` verbringt bei Programmen mit kurzer Laufzeit erheblich mehr Zeit damit, auf die Lese- und Schreiboperationen der Festplatte zu warten, als durch die Ausführung von nativem Code Zeit erspart wird.

Des Weiteren existiert ebenfalls das Problem, dass die benötigte Toolchain gegebenenfalls auf dem System nicht installiert ist, oder es gar nicht möglich ist, diese zu benutzen. Dies ist zum Beispiel in dem *Mirage*-Projekt [ope11] der Fall, bei dem das System auf einem Mikrokernel aufsetzt, auf dem keine Toolchain verfügbar ist.

Um auf die Nutzung der Toolchain verzichten zu können, müssen Teile dieser Funktionalität reimplementiert werden. Diese Reimplementierung bringt zusätzlich den Vorteil mit sich, dass die Kontrolle über den Work-

flow nun komplett in der Hand des `ocamlnat` liegt und somit der Weg über temporäre Dateien umgangen werden kann und sowohl das Erzeugen der Opcodes sowie das Linken direkt im Speicher des `ocamlnat`-Prozesses durchgeführt werden kann.

Die Implementierung dieses geänderten Workflows soll in den kommenden Kapiteln beschrieben werden. In Kapitel 4.3 wird beschrieben, wie aus der linearisierten Darstellung der Instruktionen entsprechende Prozessoropcodes erzeugt werden, und in Kapitel 4.4 wird erläutert, wie dieser erzeugte native Code in den Prozess geladen und ausgeführt wird. Diese im Folgenden beschriebene Prozessfolge wird in Abbildung 4.1 durch den rechten Pfad dargestellt, welcher mit `ocamlnatjit` markiert ist. Die kommenden Erläuterungen setzten in dieser Abbildung an dem Punkt Emit-BIN an.

4.3 Opcode und ASM-Emitter

Um eine Implementierung zu bieten, die ohne größere strukturelle Änderung–en auskommt, welche direkt Opcodes im Arbeitsspeicher erzeugt, lag die Idee nahe am Ende der Compile-Phase anzusetzen. In der Emit-Phase werden sequenziell alle Instruktionen des linearisierten Codes abgearbeitet. Für jede dieser Instruktionen wird ein entsprechendes Äquivalent in Assembler in eine Datei geschrieben. Um die bisherige Implementierung nicht zu verlieren, wurde eine abstrakte Basisklasse `base_emitter` entworfen, von welcher es zwei verschiedene Implementierungen gibt:

1. Die Klasse `asm_emitter` beinhaltet die bestehende Funktionalität und schreibt alle Mnemonics der Instruktionen in eine Datei.

2. Die Klasse `bin_emitter` beinhaltet die neue Implementierung, die alle Opcodes in einen *OCaml-String* schreibt. Diese können später an jede beliebige Stelle im Speicher kopiert werden.

Die Idee besteht darin, für jede Assembler-Instruktion eine entsprechende Methode in den Emitterklassen bereitzustellen.

Jede Funktion des Emitters bekommt nun für die tatsächliche Ausgabe eine Instanz einer solchen Implementierung übergeben. So werden nun, an Stelle direkter Ausgaben in eine Datei, die entsprechenden Methoden der Emitterklasse ausgeführt. Jede Methode der Emitterklassen muss so gestaltet sein, dass diverse Formen von Operanden übergeben werden können. So müssen z.B. folgende Instruktionen mit einer Methode ausgeführt werden können: `mov %eax %ebx`, `mov $10 %eax`, `mov label (%eax)`. Es ist einfach

```
class base_emitter
  method mov arg0 arg1
  method add arg0 arg1
  method sub arg0 arg1
  method push arg
  method pop arg
  ...
```

zu sehen, dass die Möglichkeiten der verschiedenen Operanden recht vielseitig sind. Die Struktur, welche für die Operanden verwendet wird, wird in Kapitel 4.3.1 näher beschrieben.

Jedes native Programm, welches auf Hardwareebene ausgeführt wird, besteht aus mehreren Sektionen. Jede Sektion enthält unterschiedliche Informationen, auf welche in unterschiedlicher Weise zugegriffen wird. Der durch den *OCaml-Compiler* entstehende Code besteht aus zwei Sektionen. Diese sind:

1. `text` - In der Sektion `text` befinden sich alle Instruktionen. Hier befindet sich also der ausführbare Code.

2. `data` - In der Sektion `data` befinden sich die konstanten Werte des *OCaml-Programms*. Dies können Strings, Floats oder Ähnliches sein.

Für jeder dieser Sektionen hält die Emitterklasse eine Struktur, wie in Abbildung 4.3, in welcher die Daten der entsprechenden Sektion gespeichert werden.

```
struct section
  data        : string
  position    : int
  labels      : <name, pos> table
  globals     : <name, pos> table
  relocations : <symbol, offset, type> list
```

Abbildung 4.3: Datenstruktur für die Datenhaltung einer Sektion

Da der String, welcher für die Daten zur Verfügung steht, mit einer ausreichenden Größe reserviert ist, muss ein Zeiger `position` gespeichert werden, welcher die aktuelle Position der Einfügeoperationen darstellt. Wird die Größe des reservierten Strings überschritten, so wird der reservierte Bereich verdoppelt. Alle Daten, egal in welcher Bitbreite sie vorliegen, werden Byteweise mit der Funktion `emit_int8` in den String geschrieben. Zusätzlich zu der `emit_int8` existieren noch die Hilfsfunktionen `emit_int16` und

`emit_int32`, welche ihrerseits jedoch die `emit_int8` verwenden, um die 16bit- und 32bit-Worte byteweise zu schreiben.

```
emit_int8 (byte)
  if position >= data.length then
    double capacity of data
  data[position] = byte
  position = position + 1
```

Abbildung 4.4: Funktion zum Schreiben eines Bytes in den Datenstrom

Die Verwendung der `labels` und `globals` sollen in Kapitel 4.3.2 beschrieben werden, die `relocations` in Kapitel 4.3.3.

4.3.1 Operandendarstellung

Die Darstellung der Operanden muss so gewählt werden, dass es einen Zentralen Typ gibt, welcher alle Möglichkeiten der verschiedenen Operandentypen abdeckt. Da jede Instruktion des Instructionsets nur durch eine Methode der Emitterklasse repräsentiert werden soll, um das Set an Methoden übersichtlich zu halten, muss die Darstellung der Operanden die folgenden Typen abbilden:

- **Register** - Einfache Registeroperanden
- **Speicheradresse** - Speicherzugriffe in der Form `disp(base, index, scale)`
- **Symbol** mit optionalem **Displacement** - Referenzen auf symbolisch benannte Adressen innerhalb einer Sektion und zusätzliches Displacement zu diesem Symbol
- **Konstante** - Integerbasierter, konstanter Wert
- **Nativeint** - Wie **Konstante**, jedoch vom *OCaml-Datentyp* `nativeint`
- **Float** - Index des Floatregisters im Floatstack der i386 Architektur
- **None** - Kein echter Datentyp, wird lediglich gebraucht, um ein gemeinsames Interface für Instruktionen zu bilden, welche es in Varianten mit unterschiedlicher Operandenzahl gibt

In den meisten Instruktionen lassen sich Registeroperanden und Speicheroperanden gleichermaßen verwenden. Somit liegt die Entscheidung nahe, einen gemeinsamen Obertypen `reg_memory` für diese Typen zu verwenden. Ähnliches gilt auch für die konstanten Werte und die Symbole, welche ebenfalls nur eine andere Repräsentation eines konstanten Wertes darstellen. Der Floatstack bildet einen eigenen, für sich selbst stehenden Datentyp.

Eine Speicheradresse besteht wiederum aus vier Bestandteilen: einem Displacement, einer Basis, einem Index und einer Skalierung. Da sowohl Basis, als auch Index keine zwingenden Werte darstellen, wird für diese Registeroperanden ein eigener Typ eingeführt, welcher eine Option aus einem Register und „keine Angabe“ darstellt.

Abbildung 4.5 zeigt das endgültige Format, welches für die Darstellung der Operanden genutzt wird.

```
offset
| Const: int
| Symbol (name: string,
          disp: int)
| NativeInt: nativeint

optional_reg
| Reg: int
| None

reg_memory
| Reg: int
| Mem (disp:  offset,
       base:  optional_reg,
       index: optional_reg,
       scale: int)

operand
| RegMem: reg_memory
| Loc: offset
| Float: int
| None
```

Abbildung 4.5: Format für die Darstellung der Operanden

4.3.2 Label und Symbole

An diversen Stellen des Codes werden Symbole bzw. Label[1] verwendet, um Positionen in den Sektionen zu markieren. Symbole werden verwendet, um in der `data` Sektion bestimme Werte zu markieren, und in der `text` Sektion markieren sie die erste Instruktion einer Instruktionsfolge, welche eine Funktion darstellt. Label werden hingegen ausschließlich in der `text` Sektion verwendet, um lokale Sprungziele zu definieren. Solche lokalen Sprungziele werden benutzt, um den Flow-Control bei Bedingungen, Schleifen, Garbage-Collector-Aufrufen usw. zu steuern.

In jeder Sektion wird, wie in Abbildung 4.3 dargestellt, eine Tabelle mit Einträgen für Label gehalten. Eine solche Tabelle bildet ein Mapping von Symbolnamen auf die Positionen an denen diese Symbole stehen. Da dem gesamten Code zu diesem Zeitpunkt noch kein fixer Speicherbereich zugewiesen wurde, sind diese Positionen relative zu dem Beginn der Sektion. Die Ermittlung der absoluten Werte wird in Kapitel 4.4.3 behandelt.

In der Emitterklasse gibt es für das Anlegen eines solchen Verweises eine Methode `store_symbol`. Um die Position eines solchen Symbols zu speichern, wird in der Tabelle `labels` zu dem entsprechenden Symbolnamen die aktuelle Position `position` abgelegt.

4.3.3 Relocations

Wie in Kapitel 4.3.2 beschrieben, beschreiben Symbole eine Position innerhalb einer Sektion. Diese Symbole können an diversen Stellen im Programm referenziert werden, indem sie als Parameter von Instruktionen dienen. Unter Relocation versteht man das Auflösen der Symbolnamen zu Speicheradressen, welche dann an den referenzierenden Stellen im Code eingetragen werden. Generell kann zwischen zwei verschiedenen Arten der Relocation unterschieden werden:

- Der relativen Relocation und
- der direkten Relocation

In allen innerhalb dieser Arbeit[2] verwendeten Sprunginstruktionen werden, unabhängig ob dies bedingte oder unbedingte Sprünge sind, relati-

[1] Im Kommenden wird, sofern nicht ausdrücklich anders erwähnt, Symbol stellvertretend für Symbol oder Label verwendet.

[2] Das *Intel-Instruction-Set* sieht auch Sprünge mit absoluten Adressen vor, jedoch werden diese innerhalb des *OCaml-Compilers* nicht verwendet.

ve Sprungadressen verwendet. Für alle relativen Sprünge, deren Sprungziele in der gleichen Sektion liegen, werden somit die absoluten Adressen nicht benötigt und können bereits ausgewertet werden bevor die endgültigen Speicherbereiche alloziert wurden.

Bei allen anderen Instruktionen, in denen symbolische Bezeichner verwendet werden können, werden absolute Adressen verwendet. Alle direkten Relocations benötigen eine absolute Speicheradresse, an der die Daten oder Instruktionen stehen. Dies ist jedoch erst möglich, wenn für die Sektionen der Speicher alloziert wurde und somit deren endgültige Position feststeht. Dies wird in Kapitel 4.4.3 beschrieben.

Wenn während der Erzeugung des Codes symbolische Bezeichner auftreten, so ist deren Position evtl. an dieser Stelle noch nicht bekannt, denn bei absoluten Adressen wird der absolute Speicheroffset benötigt und bei relativen Adressen wird das Symbol evtl. erst noch definiert. Da die Adresse nicht bekannt ist, kann auch noch keine Entscheidung darüber getroffen werden, ob eine 8bit Adresse ausreicht oder ob eine 32bit Adresse benötigt wird. Somit wird immer die Instruktion mit der 32bit Adresse gewählt. Ähnlich wie bei den Symbolen in Kapitel 4.3.2 muss die aktuelle Position innerhalb dieser Sektion gesichert werden, denn an dieser Stelle muss, wenn die Adresse des Symbols bekannt ist, der korrekte Wert eingetragen werden. Wie die Definition einer Sektion in Kapitel 4.3 zeigt, wird eine Liste `relocations` gehalten, welche die folgenden Werte enthält:

- `offset` - Bezeichnet die aktuelle Position innerhalb der Sektion.
- `symbol` - Bezeichnet das Symbol, dessen Speicheradresse an dieser Stelle einzutragen ist.
- `typ` - Besagt, ob dies ein relativer oder ein direkter Verweis ist.

Wenn die Emit-Phase abgeschlossen ist, d.h. alle Instruktionen und alle Daten geschrieben wurden, so können innerhalb der `text` Sektion alle relativen Verweise aufgelöst werden, denn zu diesem Zeitpunkt sind alle Symbole, die diese Sektion definieren kann, definiert. Es werden alle Einträge in der `relocations` Liste abgearbeitet, welche einen relativen Typ haben. Die Direkten werden erst in Kapitel 4.4.3 behandelt. Die Differenz zwischen dem Offset der Relocation, also die Position an der das Symbol referenziert wird, und der nun bekannten Adresse des Symbols ergibt die Zieladresse, welche an der Stelle `offset` eingetragen werden muss. Abbildung 4.6 zeigt schematisch wie ein solcher Sprungbefehl aussieht. Die `jmp`-Instruktion selbst ist ein 8bit Opcode und die Adresse besteht aus einem 32bit breiten Wort. Der Relocation-Offset zeigt auf das erste Byte der Adresse. Wenn dieser Befehl

nun von einem Prozessor ausgeführt wird, so wird erst der komplette Befehl gelesen bevor dieser Ausgeführt wird. Zu diesem Zeitpunkt zeigt der Instruktionszeiger bereits auf die nächste Instruktion. Dadurch entsteht eine Differenz von vier Byte (die Länge der Adresse), welche bei Durchführung der Relocation korrigiert werden muss.

Instr.	Adresse	
`jmp`	`XX XX XX XX`	*Nächste Instruktion*
	↑ Offset	↑ Instruktionszeiger

Abbildung 4.6: Jump-Befehle mit Sprungziel und Position des Instruktionszeigers, nachdem die Instruktion gelesen wurde, jedoch bevor sie ausgeführt wurde.

Der somit entstehende Term für die Berechnung einer relativen Adresse ergibt sich zu `symbol_address - relocation_offset - 4`.

4.4 Linken und Ausführen

In Kapitel 4.3 wurde beschrieben, wie die Emit-Phase des Compileprozesses angepasst werden muss, um ausführbaren Code direkt im Kontext des *ocamlnat* zu erzeugen. Doch dies reicht noch nicht aus, um diesen Code auch tatsächlich auszuführen, denn die folgenden, noch offenen, Punkte müssen noch durchgeführt werden:

- Der endgültige Speicherbereich für die Sektionen muss bereitgestellt werden.
- Symbole, welche durch das zu compilierende Programm exportiert werden, müssen registriert werden.
- Die global Relocation für Symbole aus anderen Sektionen und anderen kompilierten Programmen muss durchgeführt werden.
- Die Speicherbereiche müssen für die Ausführung vorbereitet werden.
- Das Programm muss ausgeführt werden.

Für einige dieser Punkte reichen die Mittel, die *OCaml* bietet, jedoch nicht aus. Es werden Implementierungen in einer Sprache benötigt, die mehr Hardwarenähe bieten kann. Da in der bestehenden Implementierung des *ocamlnat* ebenfalls bereits für das Laden und Ausführen von nativen Bibliotheken Bereiche in *C* geschrieben wurden, bietet es sich an, die hier benötigten Implementierungen ebenfalls in *C* umzusetzen.

4.4.1 Allozieren des Speichers

Die Opcodes und die dazugehörigen Daten stehen bisher in einem *OCaml-String*. Der Speicher, der durch einen solchen String belegt wird, und die Struktur, welche *OCaml* für Strings bereitstellt, eignet sich jedoch nicht dazu, den enthaltenen Code auf Hardwareebene ausführen zu können. In einer *C-Funktion* wird mit dem Befehl `mmap` der Speicherbereich alloziert, welcher für die beiden Sektionen `text` und `data` benötigt wird. Da zu diesem Zeitpunkt noch ein schreibender Zugriff auf beide Sektionen benötigt wird, werden beide Speicherbereiche sowohl lesbar also auch schreibbar erzeugt.

Der allozierte Speicher muss nun in den Kontext des ausführenden *OCaml-Prozesses* übergeben werden. Da *OCaml* jedoch keinen Datentyp für Speicher hat, wie dies in *C* ein `void*` bietet, muss die Adresse des Speicherbereichs als Integer übergeben werden. Auf 32bit Systemen deckt der *OCaml* `int` jedoch, aufgrund des Boxings, nur einen Bereich von 31bit ab. Somit muss an dieser Stelle die Wahl auf den *OCaml* `nativeint` fallen, welcher den kompletten Bereich eines 32bit Integers abdeckt. Mit der, durch die *OCaml-C-Runtime* bereitgestellten Methode `caml_copy_nativeint`, lässt sich der *C* `void`-Pointer in einen *OCaml* `nativeint` konvertieren. Mit dieser Speicheradresse lassen sich die folgenden Relocations in Kapitel 4.4.3 durchführen.

An dieser Stelle wird der entstandene Code noch nicht in die entsprechenden Speicherbereiche kopiert, da die weiteren Verarbeitungen in den bestehenden Strukturen erheblich einfacher erfolgen. Erst wenn alle weiteren Schritte abgeschlossen sind, kann in Kapitel 4.4.4 der Code an seine finale Stelle kopiert und ausgeführt werden.

4.4.2 Globale Symbole

Alle Symbole, die eine Sektion bietet, welche auch von anderer Stelle (andere Sektionen oder auch andere kompilierte Teilprogramme) referenziert werden, werden in einer Liste `globals` innerhalb der Sektion gesammelt. Es soll für den aufrufenden Prozess keinen Unterschied machen, ob diese Symbole durch das aktuell auszuführende Programm exportiert werden, oder sie durch andere Ausführungen oder bereits geladene *OCaml-Bibliotheken* geladen wurden. Zu diesem Zweck wurde eine weitere *C-Funktion* `register_symbol` eingeführt, welche in einer Liste ein Mapping von Symbolnamen zu absoluten Speicheradressen, in der diese Symbole zu finden sind, hält.

Durch eine weitere *C-Funktion* `get_symbol` kann die entsprechende Speicheradresse zu einem Symbol wieder aufgelöst werden. Die Symbole, die hier abgefragt werden können, können auf zwei unterschiedlichen Wegen registriert worden sein:

- Durch die Funktion `register_symbol` innerhalb des Compileprozesses oder

- sie wurden durch den Runtimelinker geladen und registriert.

Mit der *Standard-C-Funktion* `dlsym` können Symbole aufgelöst werden, welche durch den Runtimelinker in den aktuellen Prozess geladen und dort registriert wurden. Wenn diese Namensauflösung zu keinem Ergebnis führt, wird die Liste mit eigens gehaltenen Symbolen durchsucht. Im Fehlerfall, dass das gesuchte Symbol nicht gefunden wird, wird eine Exception geworfen, welche von dem aufrufenden `ocamlnat` gefangen und behandelt wird.

4.4.3 Globale Relocation

Nachdem nun die endgültigen Speicherpositionen der einzelnen Sektionen bekannt sind, kann im letzten Schritt vor der Ausführung die globale Relocation durchgeführt werden. Sie funktioniert analog zu der lokalen Relocation, welche in Kapitel 4.3.3 vorgestellt wurde.

Die relative Relocation funktioniert in der Globalen exakt genauso wie diese in der lokalen Relocation, bis auf den Umstand, dass nicht mehr mit den Offsets relativ zum Beginn der Sektion, sondern mit den absoluten Speicheradressen gearbeitet wird.

In der direkten Relocation können nun an den entsprechenden Stellen die absoluten Speicheradressen direkt in den Code eingetragen werden. Eine Korrektur von vier Bytes ist hier nicht nötig, da diese Adressen absolute Werte sind und nicht relativ zum Instruktionszeiger zu betrachten sind.

4.4.4 Ausführen

Im letzten Schritt können nun, nachdem alle symbolischen Referenzen aufgelöst und im Code eingetragen wurden, die Strings der Sektionen in den finalen Speicher kopiert werden. Hierzu existiert erneut eine *C-Funktion* `write_memory`. Diese Funktion bekommt den String, also die Daten der Sektion, und die Größe durch den `ocamlnat` übergeben. Zusätzlich wird noch angegeben, ob der entsprechende Speicherbereich ausführbaren Code oder die statischen Daten der Sektion enthält. Der Inhalt des Strings kann nun letztlich in den allozierten Speicher kopiert werden und entsprechend dem Executable-Flag wird der Speicherbereich

- ausführbar für die `text` Sektion und

- lesbar und schreibbar für die `data` Sektion gesetzt.

Um den erzeugten Code nun erfolgreich auszuführen, müssen zuvor ein paar definierte Symbole an die *OCaml-Runtime* übergeben werden, damit Funktionalitäten wie Exception-Handling oder der korrekte Ablauf des Garbage-Collectors gewährleistet werden können.

Mit der durch die *OCaml-C-Runtime* bereitgestellten Methode `caml_page_table_add` werden die Speicherbereiche der Code-Sektion und der Data-Sektion registriert. Für diesen Zweck exportieren die `text` und die `data` Sektionen global Symbole, welche die entsprechenden Bereiche umschließen. Für die Data-Sektion sind dies die beiden Symbole `caml__data_begin` und `caml__data_end`. Da die `data` Sektion nichts außer dieser Information enthält, umschließen diese beiden Symbole den vollständigen Bereich der `data` Sektion. Für die Code-Sektion sind dies die beiden Symbole `caml__code_begin` und `caml__code_end`. Da die `text` Sektion zusätzlich noch die *Frametable* enthält, umschließen diese beiden Symbole nicht den kompletten Bereich der `text` Sektion.

Ein weiteres definiertes exportiertes Symbol ist die `caml__frametable`, welche ebenfalls in der *OCaml-Runtime* registriert werden muss.

Das letzte dieser Symbole ist der `caml__entry`. Er definiert den Haupteinsprungspunkt des auszuführenden Programms. Auch hier existiert eine *C-Funktion* `caml_callback`, welche die Speicheradresse des Entry-Points als Eingabe bekommt. Diese Funktion führt letztlich das Programm aus und liefert das Ergebnis zurück, welches an den aufrufenden `ocamlnat` Prozess weitergereicht wird.

Kapitel 5

Performancevergleich

In den vorangegangen Kapiteln wurde vorgestellt, wie im Rahmen dieser Diplomarbeit zur Weiterentwicklung der nativen *OCaml-Toplevel* `ocamlnat` beigetragen wurde. In den folgenden Abschnitten sollen diese entstandenen Implementierungen mit denen Verglichen werden, welche durch *OCaml* bisher angeboten werden. Dazu wurden die folgenden Testgebiete untersucht.

- **Compiletime** - Die Stärke des *Linear-Scan-Algorithmus* liegt in der Ausführungsgeschwindigkeit des Allokators, womit eine Steigerung verglichen mit dem bisherigen *Graph-Coloring-Algorithmus* zu erwarten ist.

- **Runtime** - Es wird ein Vergleich der Laufzeit zwischen Programmen, welche mit dem *Linear-Scan-Registerallokator* compiliert wurden, und denen, welche mit dem *Graph-Coloring-Algorithmus* compiliert wurden, angestellt.

- ***Native-OCaml-Toplevel*** - Die Änderungen des Workflows der nativen *OCaml-Toplevel* `ocamlnatjit` werden mit den bisherigen zwei Varianten, der interpretierten Bytecode-Toplevel `ocaml` und der nativen Toplevel `ocamlnat`, verglichen.

Für die Tests in den folgenden Kapiteln wurden diverse Benchmarkprogramme herangezogen, die zum Teil der *OCaml-Testsuite* und zum Teil aus *The Computer Language Benchmarks Game* [sho11] entnommen sind. Im Folgenden sollen kurz die für diese Testfälle herangezogenen Testprogramme vorgestellt werden:

- `binarytree` baut Binärbäume in großer Anzahl auf und wieder ab.

- `fasta` erzeugt und schreibt zufällig gebildete DNS-Sequenzen.

- `mandelbrot` Implementierung des wohl bekannten Mandelbrot-Algorithmus.
- `meteor` Lösung für ein Packproblem auf Basis von Hexagonen.
- `nbody` führt eine n-nody Simulation der jovianischen Planeten durch.
- `spectral` führt eine Eigenwertberechnung durch.
- `almabench` ist ein Testprogramm für Fließkommazahlenverarbeitung designed für den Vergleich auf unterschiedlichen Plattformen.
- `bdd` ist eine Implementierung eines binären Entscheidungsbaums.
- `boyer` führt diverse Manipulationen über Termen durch.
- `fft` ist eine Implementierung einer schnellen Fourier-Transformation.
- `nucleic` ist ein weiterer Floating-Point Benchmark.
- `quicksort` ist eine Implementierung des wohl bekannten Quicksort-Algorithmus für Arrays und dient als guter Test für Schleifen.
- `soli` ist ein einfaches Programm, für die Lösung des Solitär-Problems, welches gut geeignet ist, um die Performance von nicht-trivialen kurzläufigen Programmen zu testen.

Alle Messungen dieses Kapitels wurden auf den folgenden Systemen durchgeführt:

- Ein Fujitsu Siemens Primergy server mit zwei Intel(R) Xeon(TM) E5520 2.26GHz CPUs (8 MiB L2 Cache, 4 Cores), und 12 GiB RAM, CentOS release 5.5 (Final) mit Linux/x86_64 2.6.18-274.3.1.el5. C Compiler ist `gcc-4.1.2` (Red Hat 4.1.2-50).
- Ein Intel(R) Core(TM) i5-2300 @ 2.8 GHz CPU (4 Cores) 4 GiB RAM, Ubuntu Linux i686. C Compiler ist gcc-4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4)
- Ein Fujitsu Siemens Primergy server mit einem Intel Pentium 4 “Northwood” 2.4 GHz CPU (512 KiB L2 Cache), und 768 MiB RAM, Debian testing von 2011/09 mit Linux/i686 3.0.0-1-686-pae. C Compiler ist `gcc-4.6.1` (Debian 4.6.1-4).
- Ein Macintosh PowerBook G3 mit einem PPC 740/750 und 154 MiB RAM, Gentoo-Linux. C Compiler ist `gcc-3.2.3-r4` (Gentoo Linux 1.4 3.2.3-r4)

5.1 Compiletime

In diesem Abschnitt wird die Performance des *Linear-Scan-Algorithmus* mit der des *Graph-Coloring-Algorithmus* verglichen. Gegenstand dieses Vergleichs ist die Ausführungsgeschwindigkeit des Compilers. Die gemessenen Zeiten beinhalten alle Compile-Phasen, welche in den Kapiteln 3.2 und 4.1 vorgestellt wurden. Die aufgeführten Programme wurden auf allen Systemen jeweils vierzig Mal mit dem *Graph-Coloring-Algorithmus* und dem *Linear-Scan-Algorithmus* compiliert. Der jeweils beste Messwert ist als finaler Wert in die Messung eingegangen.

Der Performance-Gewinn des Registerallokators hängt stark von dem zu compilierenden Programm ab. Der wichtigste Faktor der Geschwindigkeitsverbesserung hängt mit der Anzahl der parallelen Intervalle zusammen. Dies wird in Abschnitt 5.1.1 näher dargestellt. Um diese Geschwindigkeitssteigerung deutlich zu machen, zeigt die Abbildung 5.1 die prozentualen Zeiten der Compile-Vorgänge des *Linear-Scan-Algorithmus* im Vergleich zum *Graph-Coloring-Algorithmus*. Die angetragenen Werte entsprechen dem Wert $\tau = 100 * \frac{t_{ls}}{t_{gc}}$.

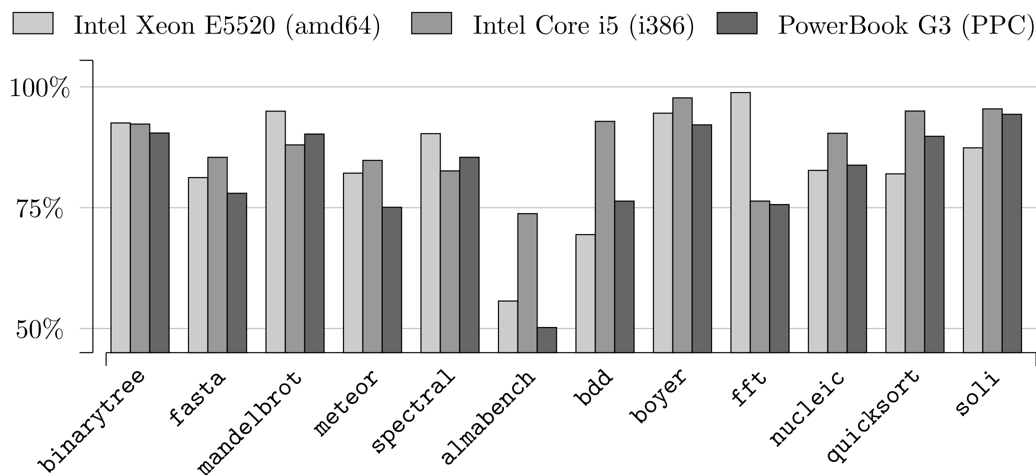

Abbildung 5.1: Prozentualer Compiletime-Vergleich

Wie man sehen kann, variiert der Performance-Gewinn von Programm zu Programm. Während der Compile-Vorgang bei `almabench` mit dem *Linear-Scan-Algorithmus* annähernd doppelt so schnell ist wie dem *Graph-Coloring-Algorithmus*, so ist bei `fft` kaum eine Verbesserung zu verzeichnen. Es ist leicht zu Verstehen, dass der *Linear-Scan-Algorithmus* nur dann eine merkliche Verbesserung bringen kann, wenn der *Graph-Coloring-Algorithmus* viel Zeit verglichen mit den anderen Phasen des Compilers benötigt.

5.1.1 Pathologischer Fall

In diesem Abschnitt soll das Verhalten der beiden Registerallokatoren untersucht werden, wenn die Anzahl der zeitgleichen Livetime-Intervalle stark zunimmt. Zu diesem Zweck wurden Programme geschrieben, welche lediglich dem Zweck dienen, viele zeitgleiche Überschneidungen von Intervallen zu erzeugen. Diese Programme wurden jeweils von dem *Linear-Scan-* und dem *Graph-Coloring- Registerallokator* verarbeitet. Des Weiteren wurde der *OCaml-Compiler* so angepasst, dass er die Zeit der Registerallokation exportiert. So sind die in diesem Test gezeigten Zeiten die reinen Zeiten der Registerallokationen. Das Zeitverhalten der beiden Allokatoren ist in Abbildung 5.2 dargestellt. Die vertikale Achse zeigt die benötigte Zeit der Register–allokation in Millisekunden, während die horizontale Achse die Anzahl der zeitgleichen Intervalle angibt. Beide Achsen sind logarithmisch dargestellt.

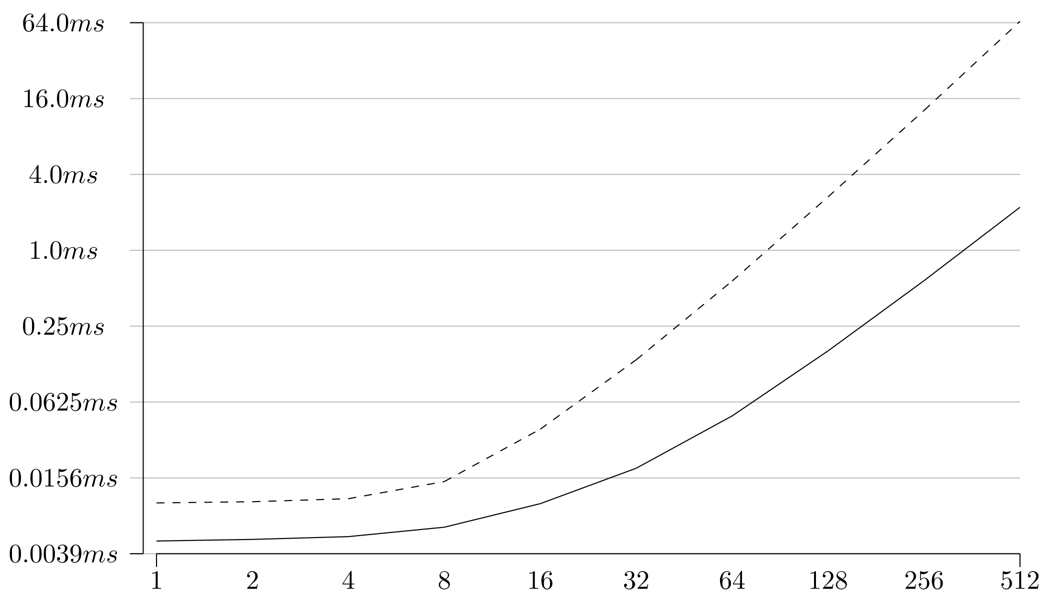

Abbildung 5.2: Compiletimeverhalten bei steigender Anzahl zeitgleicher Livetimeintervalle

Die gestrichelte Kurve zeigt das Verhalten des *Graph-Coloring-Algorithmus*, während die durchgezogene Kurve das Verhalten des *Linear-Scan-Algorithmus* darstellt. Es ist deutlich zu sehen, wie der Zeitbedarf des *Graph-Coloring-Algorithmus* mit steigender Anzahl zeitgleicher Livetime-Intervalle steiler ansteigt als der des *Linear-Scan-Algorithmus*. Bei 512 überschneidenden Intervallen ist der *Linear-Scan-Algorithmus* bereits ca. dreißig Mal schneller als der *Graph-Coloring-Algorithmus* und es ist abzusehen, dass der Unterschied weiter ansteigt.

5.2 Runtime

Die große Stärke des *Linear-Scan-Algorithmus* liegt in seiner schnellen Aus–führung, jedoch sind Compiletime-Performance und Runtime-Performance zwei gegensätzliche Ziele und im Allgemeinen nicht gleichzeitig zu realisieren. Dennoch ist die Runtime-Performance ein wichtiger Faktor in der Bewertung des Registerallokationsverfahrens, denn es ist nichts durch ein schnelles Allokationsverfahren gewonnen, dessen Code-Ausführung erheblich langsamer ist.

Für die Ermittlung dieser Werte wurden die Programme mit *Graph-Coloring-Registerallokation* und *Linear-Scan-Registerallokation* compiliert und jeweils vier Mal ausgeführt. Der jeweils beste gemessene Wert ist als finaler Wert in die Messung eingegangen. Die folgende Abbildung 5.3 zeigt die prozentuale Laufzeit des mit *Linear-Scan* compilierten Programms im Vergleich zu dem mit *Graph-Coloring* compilierten Programms. Der angetragene Wert entspricht also $\tau = 100 * \frac{t_{ls}}{t_{gc}}$.

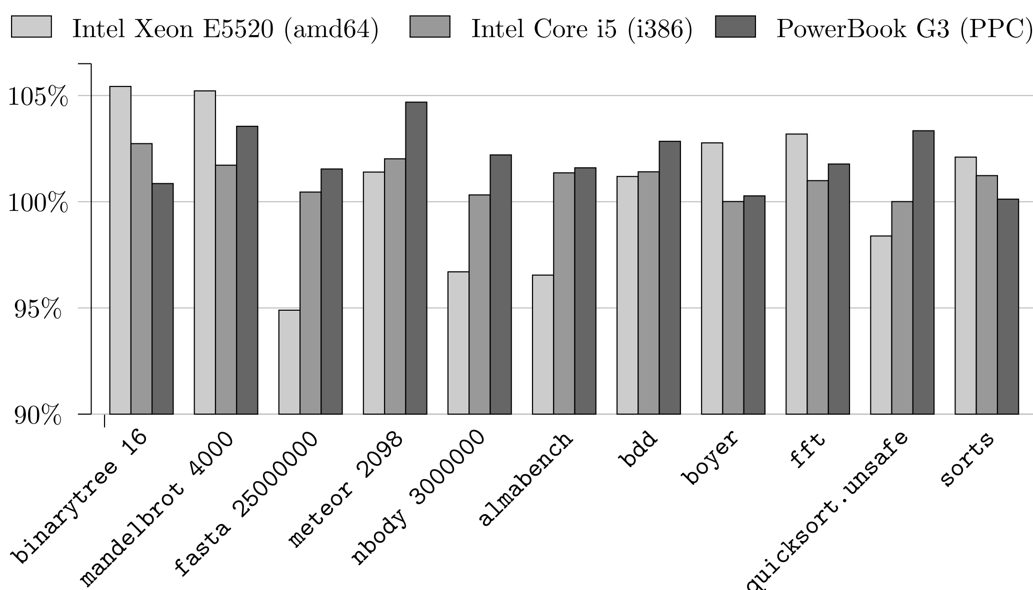

Abbildung 5.3: Prozentualer Runtime-Vergleich

Man kann gut sehen, dass sich die Ausführungszeit bei *Linear-Scan* nicht signifikant von der mit *Graph-Coloring* unterscheidet. Die schlechtesten Werte dieser Messreihe treten bei `binayrtree` und `mandlbrot` auf, wo die Ausfüh–rungszeit mit *Linear-Scan* bei etwa 105% der Zeit der *Graph-Coloring-Variante* liegt. Bei den besten Ergebnissen dieser Messreihe `fasta` mit Parameter `25000000` erfolgt die Ausführung mit *Linear-Scan* sogar deutlich schneller

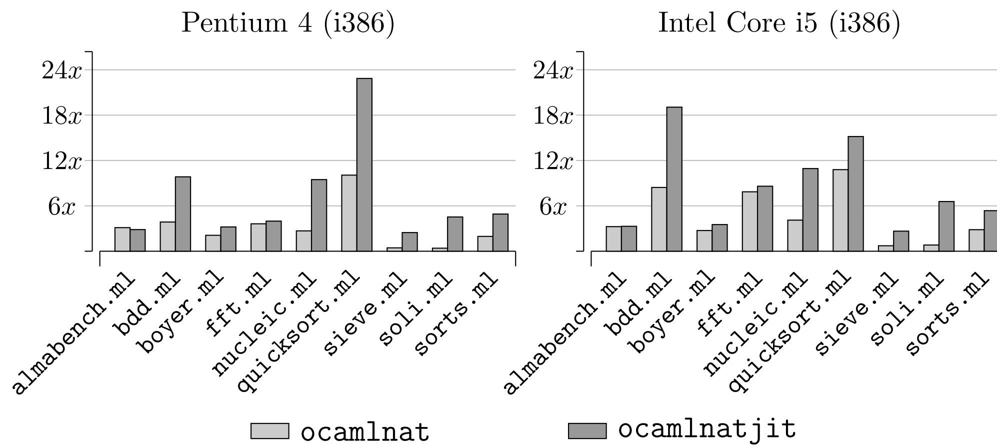

Abbildung 5.4: Geschwindigkeit, im Vergleich zu der Bytecode-Toplevel `ocaml`

als mit *Graph-Coloring*. Im Durchschnitt lässt sich jedoch sehen, dass die Ausführung der mit *Linear-Scan-Registerallokator* compilierten Programme annähernd gleich schnell ist wie die mit *Graph-Coloring-Registerallokator*, wobei jedoch die Compilezeit in allen Testfällen unter der mit *Graph-Coloring* liegt. Diese Kombination macht das *Linear-Scan-Verfahren* zu einem guten Registerallokator für eine Umgebung wie die der nativen *OCaml-Toplevel*, wo ein Geschwindigkeitsvorteil im Zusammenspiel von Compiletime und Run–time zu sehen ist.

5.3 Native OCaml Toplevel

In diesem Abschnitt wird ein Vergleich zwischen den *OCaml-Toplevels* angestellt. Dies sind die drei Varianten:

- `ocaml` - die Bytecode-Toplevel
- `ocamlnat` - die Native-Toplevel basierend auf der nativen Toolchain
- `ocamlnatjit` - die Native-Toplevel, die innerhalb dieser Arbeit entstanden ist

Da die Entwicklung des `ocamlnatjit` lediglich als Proof-Of-Concept für die *i386-Architektur* erfolgte, sind auch alle angestellten Messungen und Vergleiche lediglich *i386-Systemen* entnommen. Abbildung 5.4 zeigt die Geschwindigkeitssteigerung im Vergleich zu der Bytecode-Toplevel. Die Messun-

gen, welche durch die hellen Balken angezeigt werden, stellen die Geschwindigkeitssteigerung durch die Nutzung des `ocamlnat` dar und die dunklen Balken die Steigerung durch die Nutzung des `ocamlnatjit`. Diese Messungen zeigen deutlich eine enorme Performancesteigerung von bis zu annähernd 25facher Geschwindigkeit bezogen auf die Bytecode-Toplevel. Selbst die schlechtesten gemessenen Werte zeigen immer noch eine Verdoppelung der Geschwindigkeit.

Kapitel 6

Fazit

Wie die Ergebnisse der Messungen aus dem vorangegangenen Kapitel 5 zeigen, ist durch die Nutzung des *Linear-Scan-Algorithmus* in jedem Fall eine Steigerung der Compiletime-Performance zu verzeichnen, während der entstehende Code sich in jedem Fall mit dem durch *Graph-Coloring* entstandenen messen kann. In einigen Fällen ist sogar eine Steigerung der Runtime-Performance zu beobachten. Wenn man diese Aspekte der Performance zusammen betrachtet, lässt sich durchaus erkennen, dass der *Linear-Scan-Regis–terallokator* in Onlinesituationen wie bei der nativen *OCaml-Toplevel*, eine adäquate Alternative zu dem bestehenden *Graph-Coloring-Algorithmus* darstellt.

In Kapitel 3 wurde ja bereits angesprochen, dass einige Kompromisse geschlossen wurden, um möglichst viel bestehenden Code wieder zu verwenden. Es ist durchaus denkbar, dass, wenn einige der den Registerallokator umgebenden Codestellen auf die Bedürfnisse des *Linear-Scan-Algorithmus* angepasst werden, die Compiletime-Performance noch weiter gesteigert werden kann. So wäre es durchaus denkbar, innerhalb der Livetime-Analyse direkt die Intervalle zu erzeugen. Auf diese Weise könnte man sich einen kompletten Schritt der Registerallokation sparen. Auch in der Durchführung des *Linear-Scan-Algorithmus* sind noch Änderungen denkbar. So ist das Second-Chance-Bin-Packing [THS98] eine denkbare Möglichkeit, wobei dies jedoch wieder zu Lasten der Compiletime-Performance geht.

Die Ergebnisse der Messungen bezüglich der nativen *OCaml-Toplevel* haben jedoch alle Erwartungen mehr als erfüllt. Diese Entwicklung hat gezeigt, dass die native *OCaml-Toplevel* `ocamlnatjit` in allen Belangen ihren Konkurrenten weit überlegen ist. Während der Entwicklung wurden bereits Bemühungen angestellt [MF11], diese Implementierung auf die *amd64-Architektur* zu portieren, wo auf einem *iMac* bis zu 100fache Geschwindigkeiten gemessen wurden.

Literaturverzeichnis

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.

[bin11] GNU Binutils `http://www.gnu.org/software/binutils/`, Oct 2011.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17:98–101, June 1982.

[cyg11] Cygwin `http://cygwin.com/`, Oct 2011.

[Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A Instruction Set Reference, A-M*, May 2011. Order Number: 325383.

[MF11] Benedikt Meurer and Marcell Fischbach. Towards a native toplevel for the OCaml language. Sep 2011.

[min11] MinGW - Minimalist GNU for Windows `http://www.mingw.org/`, Oct 2011.

[ope11] Mirage a cloud operating system `http://www.openmirage.org/`, Oct 2011.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, September 1999.

[Rem02] Didier Remy. Using, Understanding, and Unraveling the OCaml Language From Theory to Practice and Vice Versa. In Gilles Barthe et al., editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 413–537, Berlin, 2002. Springer-Verlag.

[San97] The Santa Cruz Operation, Santa Cruz, California, USA. *SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement*, 4th edition, March 1997. http://www.sco.com/developers/devspecs/.

[sho11] The Computer Language Benchmarks Game http://shootout.alioth.debian.org/, Oct 2011.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *SIGPLAN Not.*, 33:142–151, May 1998.

[Wim04] Christian Wimmer. Linear scan register allocation for the java hotspot client compiler, 2004.

Erklärung

Hiermit versichere ich, dass ich vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Siegen, den 3. November 2011 ______________________________

(Marcell Fischbach)