FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

LLVM Compiler-Pass zur Optimierung von Switch-Instruktionen beim Dispatch in Interpretern

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software and Information Engineering

eingereicht von

Andreas Rohner
Matrikelnummer 0502196

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl
Mitwirkung:

Wien, 14.07.2011 ________________________ ________________________

(Unterschrift Verfasser) (Unterschrift Betreuer)

Zusammenfassung

Interpreter verbringen einen Großteil ihrer Laufzeit mit der Ausführung von Indirect-Branches. Eine gute Indirect-Branch-Prediction ist also Schlüssel für eine gute Performanz. Interpreter, die eine Switch-Instruktion verwenden, sind um einen Faktor von bis zu $2,02$ langsamer als Interpreter mit Direct-Threaded-Code, weil letzere für jede VM-Instruktion einen eigenen Indirect-Branch haben, was die Indirect-Branch-Prediction enorm verbessert. Diese Arbeit stellt einen Compiler-Pass vor, der Switch-Statements so optimiert, dass ein ähnlicher Effekt wie bei Direct-Threaded-Code entsteht. Das führt für Ocaml zu einer Verbesserung der Performanz um einen Faktor von $1,45$-$2,4$.

Inhaltsverzeichnis

Inhaltsverzeichnis ii

1 Einführung 1

2 Implementierung 4

2.1 Wahl des Compiler-Frameworks 4
2.2 Begriffsklärung 4
2.3 Algorithmus 5
2.4 Implementierungsdetails 8

3 Benchmarks 12

3.1 Testbedingungen 12
3.2 Dateigröße des Interpreters 13
3.3 Compile-Zeit des Interpreters 14
3.4 Mandelbrot-Menge 15
3.5 Fibonacci-Zahlen 15

4 Verwandte Arbeiten 19

5 Zusammenfassung und Ausblick 21

Literaturverzeichnis 23

KAPITEL 1

Einführung

Effiziente Bytecode-Interpreter, wie etwa Python, Ocaml oder Java, verwenden für den Bytecode-Dispatch, sprich die Zuordnung von Bytecode-Instruktionen zur auszuführenden Funktion, meist einen der beiden folgenden Ansätze. Zum einen Direct-Threaded-Code, was die effizienteste Methode mit dem geringsten Overhead ist [Bel73], aber nicht dem ANSI-C-Standard entspricht, und zum andern ein großes Switch-Statement. Bei Direct-Threaded-Code besteht jede VM-Instruktion aus einer Speicheradresse zu dem Ort, an dem die entsprechende Funktionalität implementiert wurde, und der Dispatch besteht aus einem direkten Sprung an diese Stelle [EG01]. Beim Ansatz mit Switch besteht eine VM-Instruktion hingegen aus einem beliebigen Integerwert und der Programmcode, der die zugehörige Funktionalität impelementiert, befindet sich in den Case-Statements. Beide Ansätze werden jedoch von den meisten Compilern (z.B.: GCC,LLVM,..) mit Hilfe von Indirect-Branches in Maschinencode übersetzt, wobei die Switch-Methode zum einen wegen dem notwendigen Range-Check und Table-Lookup und zum andern auch durch eine höhere Indirect-Branch-Misprediction-Rate unterlegen ist [EG03b]. Da für jede VM-Instruktion mindestens ein Dispatch durchgeführt werden muss und die Implementierung der Funktionalität der meisten Instruktionen nur einige wenige Maschinencodebefehle umfasst, verbringen Interpreter viel Zeit beim Dispatch und mit der Ausführung von Indirect-Branches [EG03b]. Interpreter profitieren also ganz besonders von einer verbesserten Indirect-Branch-Prediction, denn wie bei jedem Sprungbefehl muss der Prozessor, im Falle einer falschen Vorhersage, die gesamte Pipeline verwerfen [EG03b]. Das kann zum einen durch verbesserte Hardware-Branch-Prediction und zum andern durch Software-Maßnahmen, die die Arbeit der Hardware unterstützen, realisiert werden.

Der Grund für die bereits erwähnten Unterschiede in der Indirect-Branch-Misprediction-Rate zwischen Direct-Threaded-Code und Switch ist, dass Direct-Threaded-Code bei jeder VM-Instruktion ein separates Dispatch mit einem eigenen Indirect-Branch anhängt. Bei Switch gibt es allerdings nur einen Indirect-Branch, der von allen VM-Instruktionen geteilt wird [EG03a]. Die Anzahl der ausgeführten Indirect-Branches ist bei beiden Ansätzen gleich, jedoch kann die Branch-Prediction-Hardware, wenn sie beispielsweise einen BTB (Branch-Target-Buffer) verwendet, im Fall von Direct-Threaded-Code und unter der Voraussetzung, dass der BTB groß ge-

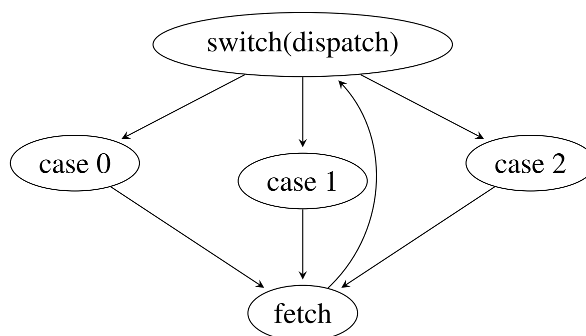

Abbildung 1.1: Vereinfachter CFG einer Switch-Instruktion

nung ist, für jede VM-Instruktion ein eigenes Sprungziel speichern. Der Branch-Predictor hat dadurch sozusagen für seine Vorhersagen mehr Kontext. Wenn beispielsweise eine VM-Instruktion in einem Programm nur einmal vorkommt, wird dessen Indirect-Branch immer korrekt vorhergesagt [EG01].

Eine von ERTL UND GREGG [EG03b] vorgeschlagene Methode, um diesen Unterschied auszugleichen, besteht darin, auch für Switch an das Ende eines jeden Case-Statements, statt der üblichen Break-Instruktion, eine seperate Kopie der Dispatch-Routine einzufügen. Der zusätzliche Overhead durch Range-Checks und Table-Lookups bleibt dabei allerdings bestehen. Da Threaded-Code um bis zu einem Faktor von $2,02$ schneller ist als derselbe Interpreter mit Switch [EG03b], könnte eine solche Optimierung ähnliche Verbesserungen erzeugen.

Der Gegenstand dieser Arbeit ist ein Compiler-Pass, der versucht eine solche Optimierung von Switch-Statements innerhalb von Schleifen umzusetzen. Abbildung 1.1 zeigt einen stark vereinfachten CFG (Control-Flow-Graph) eines Interpreters mit drei verschiedenen Instruktionen, die durch die Integerwerte 0-2 repräsentiert werden. Der hier vorgestellte Compiler-Pass sucht innerhalb von Schleifen im CFG nach Knoten, an denen sich der Kontrollfluss von mehreren Vorgängern vereinigt. Von diesen Knoten wird eine Tiefensuche im CFG nach passenden Switch-Instruktionen durchgeführt, welche in Abbildung 1.1 nach einem Schritt erfolgreich beendet wird. Dann werden alle Knoten, die auf dem Pfad vom Vereinigungspunkt bis zur Switch-Instruktion liegen, an die Vorgänger des Vereinigungspunktes kopiert. Das Ergebnis dieser Operation ist in Abbildung 1.2 zu sehen, wobei die ausgehenden Kanten bei den Kopien von Switch wieder auf die Case-Knoten zeigen. Der dadurch erzeugte Code ist trotzdem korrekt und da jede Kopie eines Switch-Knotens einen eigenen Indirect-Branch enthält, entsteht ein ähnlicher positiver Effekt für die Branch-Prediction wie bei Threaded-Code. Diese Optimierung ist nicht nur für Interpreter, sondern beispielsweise auch für Parser oder Regex-Engines interessant.

In Kapitel 2 wird der Implementierungsansatz genauer diskutiert und auf die Besonderheiten der Implementierung mit LLVM eingegangen. Kapitel 3 beschreibt die Durchführung der Benchmarks und diskutiert deren Ergebnisse. In Kapitel 5 werden die Resultate nochmals zusammengefasst und Fragen, die während der Implementierung aufgekommen sind, diskutiert.

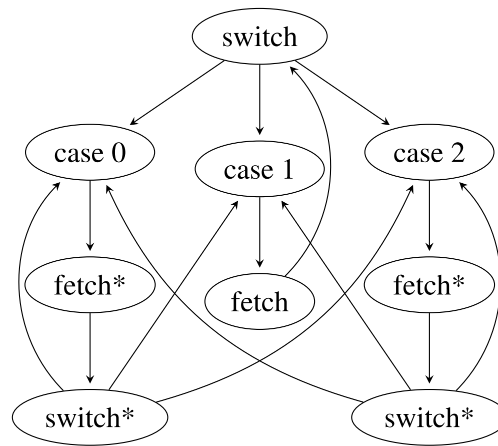

Abbildung 1.2: CFG von Abbildung 1.1 nach der Optimierung (*...Kopie)

KAPITEL 2

Implementierung

2.1 Wahl des Compiler-Frameworks

Für die Implementierung, der in dieser Arbeit vorgestellten Optimierung, wurde das LLVM-Compiler-Framework (Low Level Virtual Machine) ausgewählt, weil es ein mit GCC kompatibles Frontend und ein übersichtliches und schnell zu erlernendes Interface zur Erstellung neuer Compiler-Passes anbietet. Ersteres hat den Vorteil, dass bereits bestehende Makefiles von Testprogrammen nicht verändert werden müssen.
Alle Optimierungen in LLVM können auf zwei Arten durchgeführt werden [Kor07]:

1. Als LLVM-zu-LLVM-Pass mit LLVM Zwischencode als Eingabe und Ausgabe
2. Während der Codeerzeugung für die Zielarchitektur

Der LLVM Zwischencode beschreibt dabei ein Programm mit Hilfe eines abstrakten, RISC-ähnlichen Befehlssatzes, jedoch mit High-Level-Informationen wie etwa Typinformationen, explizitem Kontrollflussgraphen und einer expliziten Datenflussrepräsentation (durch einen unbegrenzten, typisierten Registersatz in Static-Single-Assignment-Form) [LA04].

In dieser Arbeit wurde der erste Ansatz gewählt, da es für die Duplizierung von Zwischencode bereits vorgefertigte Funktionen in LLVM gibt, und die daraus entstehenden Nachteile, wie beispielsweise die überflüssige Erzeugung mehrerer identischer Jump-Tables, durch eine kleine Änderung am Backend ausgeglichen werden können. Es wäre allerdings durchaus möglich und vielleicht sogar vorteilhaft gewesen die zweite Option zu wählen.

2.2 Begriffsklärung

Dieser Abschnitt dient der Einführung einiger Begriffe und Abkürzungen, die im folgenden Kapitel verwendet werden.

2.2.1 BasicBlock

Eine Funktion in LLVM Zwischencode besteht aus einer Reihe von BasicBlocks und jeder BasicBlock besteht wiederum aus einer Liste von LLVM-Instruktionen, die mit exakt einer Terminator-Instruktion endet [LA04]. Eine Terminator-Instruktion ist ein Sprungbefehl der alle Nachfolger explizit spezifiziert. Der Einfachheit halber wird im weiteren Verlauf dieser Arbeit statt BasicBlock nur Block verwendet.

2.2.2 CFG

Der Control-Flow-Graph bezeichnet einen Graphen, der den Kontrollfluss in einem Programm modelliert. Die Knoten stellen dabei BasicBlocks und die Kanten die möglichen eingehenden bzw. ausgehenden Wege, die der Kontrollfluss nehmen kann, dar. In LLVM ist der Kontrollflussgraph explizit, durch die Definition des BasicBlocks, gegeben. Da ein BasicBlock immer mit nur einem einzigen Sprungbefehl enden muss, können alle Nachfolgeknoten leicht bestimmt werden.

2.2.3 Pass

Ein Compiler führt auf dem Zwischencode eines Programms mehrere verschiedene Passes bzw. Durchgänge aus, die darauf abzielen den Zwischencode zu verändern, Analysen durchzuführen oder den Zwischencode in einen andern Zwischencode bzw. Maschinencode umzuwandeln.

2.2.4 Switch-Instruktion

Mit einer Switch-Instruktion ist in diesem Kontext die Terminator-Instruktion im Zwischencode von LLVM gemeint, die einen BasicBlock beendet.

2.3 Algorithmus

2.3.1 Switch-Instruktion

Dieser Abschnitt befasst sich mit den Eigenheiten von LLVM bezüglich Switch-Instruktionen und den Implikationen für die Umsetzung des, in dieser Arbeit vorgestellten, Compiler-Passes.

Switches werden in LLVM durch Switch-Instruktionen realisiert, die folgende Syntax in der Textversion des LLVM-Zwischencodes haben:

```
switch < intty > < value >, label < defaultdest > [
< intty > < val >, label < dest > ...
]
```

Diese Switch-Instruktion verwendet drei Parameter: Ein Integervergleichswert, ein Default-Sprungziel und ein Array bestehend aus Paaren von konstanten Vergleichswerten und zugehörigen Sprungzielen [Kor07].

Im Zwischencode von LLVM ist eine Switch-Instruktion eigentlich nur eine Verzweigung mit beliebig vielen Sprungzielen. Es gibt allerdings weder Informationen über mögliche Vereinigungsknoten im CFG, noch irgend einen Hinweis, ob es im Maschinencode durch einen Indirect-Branch auf eine Jump-Table oder durch normale bedingte Sprünge umgesetzt wird.

Die Entscheidung über die Umsetzung in Maschinencode geschieht erst später, da der Zwischencode plattformunabhängig ist. Diese Entscheidung basiert im Grunde auf der Anzahl der Sprungziele (> 3) und der Dichte ($>= 0.4$) der jeweiligen Integer-Vergleichswerte [Kor07]. Die Funktion, die die Dichte im Backend berechnet, kann allerdings, mit geringen Veränderungen, auch auf die Switch-Instruktion in der Zwischencodeebene angewandt werden.

2.3.2 Vereinigungsknoten

In diesem Abschnitt wird der Vereinigungsknoten, an dem sich der Kontrollfluss aus mehreren Vorgängern vereinigt, beschrieben. Dieser wird als Startpunkt für die Tiefensuche benutzt und in Abbildung 1.1 mit dem Titel "fetch" bezeichnet.

Wie schon in Abschnitt 2.3.1 erwähnt, enthält die Switch-Instruktion keinerlei Hinweise auf den Block, an dem die Switch-Anweisung im ursprünglichen Quellcode endet bzw. auf den Vereinigungspunkt der Verzweigung. Es ist außerdem überhaupt nicht erforderlich, dass ein solcher Punkt überhaupt existiert, denn jede Verzweigung könnte mit einer Goto-, Continue- oder Return-Anweisung enden. Es können auch mehrere solcher Punkte existieren die sich jeweils die Hälfte der Vorgängerknoten teilen. Ein Vereinigungspunkt mit mehreren Vorgängerknoten ist allerdings der Startpunkt einer jeden Optimierung der zugehörigen Switch-Instruktion.

Die in dieser Arbeit vorgestellte Implementierung führt daher ausgehend von jedem Block in einer Schleife, der eine, durch einen veränderbaren Parameter bestimmte, Anzahl an Vorgängerknoten hat, eine im Abschnitt 2.3.3 beschriebene Tiefensuche auf dem CFG durch, um eine passende Switch-Instruktion zu finden. Dabei ist es wohlgemerkt unerheblich ob es tatsächlich die "zugehörige" Switch-Instruktion ist, solange sie den nötigen Voraussetzungen (siehe Abschnitt 2.3.1) entspricht.

2.3.3 Limitierte Tiefensuche

Hierbei handelt es sich um eine gewöhnliche rekursive Tiefensuche auf dem CFG. Dieser ist in LLVM durch die Struktur des Zwischencodes explizit gegeben, da jeder Block mit einem expliziten Sprungbefehl enden muss. Die Tiefensuche ist auf mehrfache Weise limitiert. Zum einen werden Kanten die aus der aktuellen Schleife herausführen nicht verfolgt. Des weiteren werden alle bereits besuchten Knoten gespeichert, um Endlosschleifen bzw. das doppelte Durchlaufen eines Pfades, der sich zuerst abzweigt und später wieder vereinigt, zu verhindern. Die letzten beiden Fälle müssen spätestens auf der obersten Ebene mit einem erfolgreichen Unterbaum kombiniert werden, ansonsten gilt die Suche als nicht erfolgreich. Die Suche endet erfolgreich wenn eine passende Switch-Instruktion gefunden wird und nicht erfolgreich wenn das Block-Limit, das bestimmt wie viele Blöcke maximal kopiert werden dürfen, erreicht wird. Das Ergebnis der Tiefensuche ist im Erfolgsfall eine Liste mit allen Blöcken, die es zu kopieren gilt.

Algorithm 2.1 Rekursion der Tiefensuche als Pseudocode - Teil 1

```
1:  function searchSwitch(block, blocklist, blockcount,blocklimit)
2:      if blockcount > blocklimit then return 0
3:      end if
4:      if block is outside of loop then return 1
5:      end if
6:      if block was visited before then return 1
7:      end if
8:      blocklist(blocklist.size()) ← block
9:      blockcount + +
10:     if block contains switch then return 2
11:     end if
12:     result ← 0
13:     newblocks ← 0
14:     for all succ ∈ Successors of block do
15:         t ← blocklist.size()
16:         result ← result∨ searchSwitch(block, blocklist, blockcount,blocklimit)
17:         newblocks ← newblocks + (blocklist.size() − t)
18:     end for
```

2.3.4 Optimierung der Tiefensuche

In diesem Abschnitt werden zusätzliche Optimierungen der Tiefensuche beschrieben, die darauf abzielen, schwierige Sonderfälle zu umgehen und trotz einem restriktiven Block-Limit gute Ergebnisse zu liefern. Diese Optimierungen sind allerdings nicht zwingend für die Funktionsweise des Algorithmus notwendig.

Limit später erzwingen

Algorithm 2.2 Rekursion der Tiefensuche als Pseudocode - Teil 2

```
19:     while blockcount + newblocks > blocklimit do
20:         newblocks ← newblocks− deleteBiggestSubtree(blocklist)
21:     end while
```

Bei einer Verzweigung wird das Block-Limit erst erzwungen, wenn alle Unterbäume durchlaufen worden sind. Das heißt, jeder rekursive Aufruf für einen Unterbaum bekommt die momentane Anzahl der Blöcke in der Liste als Parameter mitgegeben, wobei dieser Wert auf der Ebene der Verzweigung für alle Aufrufe der gleiche ist, selbst wenn ein vorhergehender Aufruf die Liste bereits erweitert hat. Diese Maßnahme stellt sicher, dass keine Abzweigungsrichtung privilegiert wird, denn die Reihenfolge spielt durchaus eine Rolle. Wenn beispielsweise immer zuerst der linke Unterbaum untersucht wird und dieser zwar erfolgreich ist, aber das Block-Limit fast vollkommen ausschöpft, wäre der darauf folgende eventuell optimalere rechte Unterbaum zum Scheitern verurteilt.

Sollte es mehrere erfolgreiche Unterbäume geben, die für sich genommen innerhalb des Block-Limits sind, es aber gemeinsam überschreiten, wird so lange der größte Unterbaum gelöscht bis das Block-Limit wieder erreicht wird.

Nicht erfolgreiche Unterbäume löschen

Algorithm 2.3 Rekursion der Tiefensuche als Pseudocode - Teil 3

```
22:     if result == 0 then
23:         blocklist(blocklist.size()).remove()
24:     end if
          return result
25: end function
```

Sollte für einen Unterbaum das Block-Limit erreicht worden sein, wird die Suche nicht abgebrochen, sondern es werden alle Blöcke bis zu einer Verzweigung mit mindestens einem erfolgreichen Unterbaum aus der Liste gestrichen. So enthält die resultierende Kopie nur Blöcke die zu einem erfolgreichen Pfad gehören, während alle Sprünge auf nicht erfolgreiche Pfade auf die nicht duplizierten Originalblöcken zeigen.

Neuordnung der Blöcke

Das Ergebnis der Tiefensuche ist eine Liste mit allen zu kopierenden Blöcken. Diese sind allerdings in der Reihenfolge, in der sie von der Tiefensuche gefunden wurden, gespeichert. Diese Reihenfolge kann, je nachdem wie der Graph bei der Suche durchwandert wurde, sehr unterschiedlich von der Reihenfolge der Originalblöcke sein. Es ist also notwendig die Liste nach der Position der jeweiligen Originalblöcke in der Funktion umzuordnen. Es wird dabei allerdings trotzdem alles, was vor dem Vereinigungspunkt gelegen ist, an das Ende der Liste kopiert, damit der Vereinigungspunkt der erste Block in der Liste ist.

2.4 Implementierungsdetails

2.4.1 LLVM Passmanager

Der Passmanager von LLVM bietet eine Reihe unterschiedliche Klassen von Passes an. Auf der obersten Ebene befindet sich der sogenannte ModulePass. Ein solcher hat Zugriff auf ein ganzes Modul mit allen Funktionen und Definitionen und darf diese auch verändern. Auf der nächsten darunterliegenden Ebene befindet sich der FunctionPass, welcher für jede Funktion einzeln aufgerufen wird und auch nur diese eine Funktion verändern darf. Darauf folgt in der Rangordnung der LoopPass, welcher für jede Schleife in einer Funktion aufgerufen wird, und schlussendlich der BlockPass. Da eine Duplizierung von Switch-Instruktionen nur dann Sinn macht, wenn sich diese in einer Schleife befinden, scheint der LoopPass auf den ersten Blick die richtige Wahl zu sein. Ein LoopPass verlangt allerdings, dass die Schleifenanalyse, die für die Arbeit des LoopPassMangers notwendig ist, nach dem Durchlauf aktualisiert wird. Letzteres ist allerdings

nicht trivial, denn für jedes Ziel einer Switch Instruktion muss möglicherweise der Schleifenkopf selbst dupliziert werden. Dabei werden zahlreiche neue Kanten und Unterschleifen eingeführt.

Der LoopPassManager ist selbst wiederum als FunctionPass implementiert. Es ist also naheliegend einen FunctionPass zu verwenden, der einen stark vereinfachten LoopPassManager implementiert. Dieser ignoriert allerdings im Gegensatz zum original LoopPass von LLVM die fehlerhaften Schleifenanalyseinformationen bis zum Ende der Funktion und berechnet sie erst dann neu. Diese Mischung aus FunctionPass und LoopPass hat weitere Vorteile, da es dadurch möglich wird die neu erzeugten Duplikate erst ganz zum Schluss eines Funktionsdurchlaufes einzufügen, was mögliche Endlosschleifen verhindert.

Die Positionierung des Passes in der Abarbeitungsliste des PassManagers muss aus mehreren Gründen sorgfältig gewählt werden. Zum einen erhöht sich für nachfolgende Passes die Datenmenge durch den redundanten Code enorm, was schlecht für die Laufzeit des Compilers ist. Zum andern gibt es bestimmte Passes, die entweder versuchen redundanten Code zu entfernen, oder den CFG und Schleifen zu vereinfachen. Wie sich leicht nachvollziehen lässt würden diese Passes die Ergebnisse der Optimierung im besten Fall wieder rückgängig machen. Im Normalfall führt es jedoch zu deutlich schlechterem Code. Gleichzeitig ist es besser bereits optimierten Code zu duplizieren. Aus den eben genannten Gründen muss die Optimierung für Switch-Instruktionen möglichst spät in die Liste eingefügt werden.

2.4.2 SSA-Form

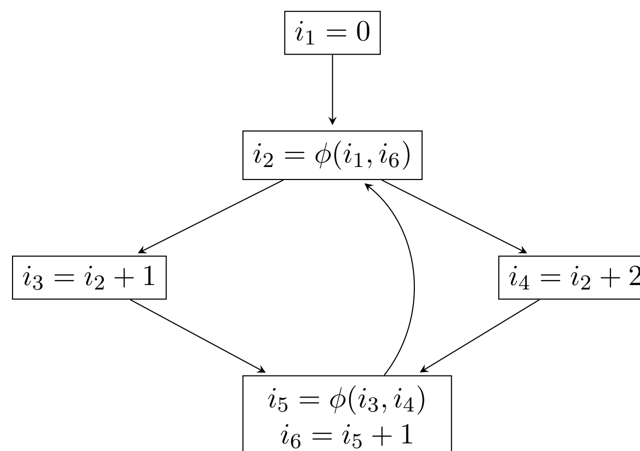

Abbildung 2.1: SSA-Form

Der Zwischencode von LLVM ist in SSA-From. Das bedeutet, dass jedes virtuelle Register nur genau ein mal beschrieben wird und alle Verwendungen des Registers von seiner Definition dominiert werden [Lat02]. Wenn ein Block mehrere Vorgängerblöcke hat, die verschiedene Versionen derselben Variable enthalten, wird ein sogenannter ϕ-Knoten erzeugt, der die beiden Versionen vereinigt und wieder einem neuen virtuellen Register zuweist. Die SSA-Form stellt einen kompakten Def-Use-Graphen zur Verfügung, der viele Datenflussoptimierungen verein-

facht und es schnellen, flow-insensitiven Algorithmen ermöglicht viele der Verbesserungen von flow-sensitiven Algorithmen ohne teure Datenflussanalysen zu erreichen [LA04].

Möchte man aber beliebig ausgewählte Blöcke in einer neuen Reihenfolge kopieren, sodass die kopierten Blöcke selbst neue Vorgängerknoten für bereits existierende Blöcke bilden und unterschiedliche Vorgänger haben als die Originalblöcke, dann steht man vor der Herausforderung alle betroffenen ϕ-Knoten und alle Variablen bzw. virtuellen Register zu aktualisieren.

Es gibt allerdings in LLVM eine Möglichkeit das Problem zu umgehen, indem man alle virtuellen Register, die außerhalb ihres eigenen Blocks verwendet werden, auf den Stack speichert und vor jeder Verwendung den Wert vom Stack liest. Damit verschwinden alle Abhängigkeiten bezüglich der Variablen zwischen den einzelnen Blöcken, weil es in jedem Block nur noch lokale Register gibt. Diese Blöcke können dann frei kopiert werden. Nachdem alle Veränderungen am CFG abgeschlossen sind, können die Werte wieder vom Stack zurück in Register gespeichert werden. Letzteres ist allerdings nicht trivial. LLVM stellt jedoch bereits eine fertige Funktion dafür zur Verfügung, welche auf dem Algorithmus von SREEDHAR UND GAO aus der Arbeit "A Linear Time Algorithm for Placing phi-nodes" basiert (siehe [SG95]).

Diese Vorgehensweise wird auch von Frontends verwendet, die nach LLVM-Zwischencode übersetzen. Dadurch, dass sie alle Variablen auf dem Stack ablegen, müssen sie keine keine SSA-Konstruktion durchführen. Die LLVM-Stack-Promotion- und Scalar-Expansion-Passes können dann verwendet werden um daraus effektiv Code in SSA-Form zu erzeugen [LA04]. Die in dieser Arbeit vorgestellte Optimierung muss allerdings möglichst spät (siehe Abschnitt 2.4.1) im PassManager eingefügt werden und deswegen muss die Stack-Promotion nochmals durchgeführt werden.

2.4.3 Jump-Tables

Wenn man eine Switch-Instruktion, die im Maschinencode in einen Indirect-Branch auf eine Jump-Table übersetzt wird, dupliziert, dann wird für jedes Duplikat eine eigene Jump-Table erzeugt. Alle diese Jump-Tables zeigen auf die gleichen Sprungziele und sind somit identisch. Um dies zu verhindern wurde die Funktion im Backend, die die Jump-Tables erzeugt, so verändert, dass zuerst eine Überprüfung auf bereits existierende Jump-Tables mit den gleichen Sprungzielen durchgeführt wird, bevor eine neue erzeugt wird. Existiert bereits eine Jump-Table wird eine Referenz darauf zurückgegeben.

2.4.4 Verbindung der Kopien

Die von der Tiefensuche ausgewählten und von der SSA-Form unabhängigen Blöcke werden für jeden Vorgängerknoten des Vereinigungspunktes kopiert, jedoch noch nicht mit diesem Verbunden. Das heißt, die Verzweigung im Vorgängerknoten wird noch nicht abgeändert, sodass sie auf die kopierten Blöcke zeigt. Dieser letzte Schritt geschieht erst nachdem die gesamte Funktion abgearbeitet wurde, da es sonst zu Kopien von Kopien und Endlosschleifen kommen kann. Dadurch dass die neuen Blöcke über den normalen Kontrollfluss unerreichbar sind bis ganz zum Schluss, kann die Optimierung bei jedem Durchlauf auf der Originalfunktion arbeiten, als ob noch keine Veränderungen vorgenommen worden wären.

2.4.5 Explosion von Kanten im CFG

Wie sich anhand von Abbildung 1.2 leicht nachvollziehen lässt, führt die Duplizierung der Switch-Instruktion nicht nur zu erheblich mehr Code, da alle Blöcke zwischen dem Vereinigungspunkt und der Switch-Instruktion mitkopiert werden müssen, sondern auch zu einer n^2 Explosion der eingehenden Kanten bei den Zielblöcken der Switch-Instruktion. Beides erhöht unabhängig von der Laufzeit des Passes selbst die Compile-Zeit, da nachfolgende Passes zum einen mehr Code und zum anderen einen erheblich komplizierteren CFG abarbeiten müssen.

KAPITEL 3

Benchmarks

In diesem Kapitel werden die durchgeführten Benchmarks und deren Ergebnisse sowie die Testbedingungen und Testprogramme beschrieben.

3.1 Testbedingungen

3.1.1 Verwendete Systeme

In Tabelle 3.1 sind alle Testsysteme mit einigen Daten aufgelistet. Zur Interpretation der in diesem Kapitel vorkommenden Grafiken sei auf eben genannte Tabelle verwiesen. Alle haben ein Linux-System mit einem aktuellen 2.6 Kernel installiert.

Typ	Prozessor	Leistung
Desktop	AMD Phenom(tm) II X6 1090T	3,4 Ghz
Netbook	AMD E-350	1,6 Ghz
Notebook	Intel Core Duo T2250	1,73 Ghz
Server	Intel Atom 330	1,6 Ghz
Server	AMD Athlon(tm) 64 X2 Dual Core	2,2 Ghz
Server	Dual Core AMD Opteron(tm)	2 Ghz

Tabelle 3.1: Testsysteme

3.1.2 Umgebung

Die Tests werden automatisiert durch ein Testscript ausgeführt. Hierbei handelt es sich um ein relativ simples Shell-Script, das zuallererst den Quellcode von LLVM sowie dessen C/C++-Frontend LLVM-GCC neu konfiguriert, compiliert und in ein lokales Verzeichnis installiert. Letzteres ist somit ohne privilegierte Nutzerrechte möglich. War dies erfolgreich, beginnt es

damit die zu testenden Programme mit verschiedensten Compiler-Optionen zu konfigurieren, zu compilieren und zu testen. Jeder Test wird 25 Mal hintereinander ausgeführt und von den Ergebnissen wird der Durchschnittswert ermittelt. Währenddessen wird die verstrichene Zeit für die jeweiligen Schritte gemessen und gespeichert. Sind die Tests abgeschlossen werden die gesammelten Daten automatisch eingelesen und für die spätere Auswertung in CSV-Dateien umgewandelt.

3.1.3 Interpreter

Da sich die in dieser Arbeit beschriebene Optimierung besonders für Interpreter, die für den Bytecode-Dispatch eine große Switch-Instruktion verwenden, eignet, wurden für die Performance Tests exemplarisch CPython 2.7.1 und Ocaml 3.12.1 ausgewählt. Im weiteren Verlauf dieser Arbeit werden diese jedoch der Einfachheit halber ohne Versionsnummer und vorangestelltem C als Python bzw. Ocaml bezeichnet.

Python verwendet ausschließlich eine große Switch-Instruktion und Ocaml unterstützt sowohl Threaded-Code für Compiler, die die Lables-As-Values-Erweiterung von GCC unterstützen, als auch eine Lösung mit Switch, welche über eine Präprozessorvariable an- bzw abgeschaltet werden kann. Leider ist es momentan nicht möglich die Effizienz einer Threaded-Code-Version von Ocaml mit den restlichen Ergebnissen zu vergleichen, da LLVM Lables-As-Values nur sehr bedingt unterstützt.

Python hat im Vergleich zu Ocaml relativ viel zusätzlichen Code innerhalb der Dispatch-Schleife, der bei der Optimierung immer teilweise oder vollständig mit dupliziert werden muss. Außerdem wurde die Dispatch-Schleife bereits mit zahlreichen goto-Anweisungen auf Labels innerhalb der Schleife optimiert, was eine weitere Optimierung der Switch-Instruktionen erschwert. Ocaml verwendet im Vergleich dazu eine sehr schlanke Dispatch-Schleife. Python und Ocaml sind sozusagen, was die Komplexität der Dispatch-Schleife angeht, relativ nah an den jeweils entgegengesetzten Enden des Spektrums anzusiedeln und somit gute Testkandidaten.

3.1.4 Programme

Als Testprogramme für die Interpreter wurden zum einen eine einfache Implementierung der Berechnung einer Mandelbrot-Menge und zum andern eine ebenso einfache rekursive Implementierung der Berechnung von Fibonacci-Zahlen in den jeweiligen Programmiersprachen ausgewählt. Dabei wurde darauf geachtet, dass die Implementierungen, soweit das bei so unterschiedlichen Programmiersprachen möglich ist, ähnlich aufgebaut sind.

Durch diese Auswahl lassen sich sicherlich nur bedingte Rückschlüsse über den Leistungsgewinn bei echten Anwendungen machen, jedoch eignet sie sich gut um die Rechenleistung des Interpreters unabhängig von anderen Einflussfaktoren wie beispielsweise IO zu testen.

3.2 Dateigröße des Interpreters

Tabelle 3.2 zeigt die Dateigröße der Ausführungsdatei relativ zur normalen Dateigröße bei unterschiedlichen Block-Limits. Da die Daten für alle Prozessormodelle ähnlich waren, wurde der

Mittelwert verwendet. Gemessen wurde die Dateigröße der Ausführungsdatei und der Sprachlibrary.

	Opt(20)	Opt(40)	Opt(80)	Opt(Unlimited)
Python	109,32%	111,72%	117,89%	124,57%
Ocaml	106,47%	106,54%	106,50%	106,50%

Tabelle 3.2: Dateigrößen in Prozent

Im Fall von Python werden mit jeder Erhöhung des Block-Limits mehr Code-Blöcke kopiert und die Dateigröße steigt entsprechend schrittweise immer weiter an. Bei Ocaml hingegen werden bei einem Block-Limit von 20 scheinbar schon alle vorhandenen Blöcke kopiert, denn die Dateigröße bleibt mehr oder weniger konstant.

Dieses Ergebnis spiegelt deutlich die in Abschnitt 3.1.3 genannten Unterschiede in der Komplexität der jeweiligen Dispatch-Schleifen wieder.

3.3 Compile-Zeit des Interpreters

Tabelle 3.3 zeigt die Compile-Zeit relativ zur normalen Compile-Zeit ohne Optimierung bei unterschiedlichen Block-Limits. Es wird der Mittelwert über alle Prozessormodelle verwendet. Gemessen wurde die Zeit die nach der abgeschlossenen Konfiguration ein Aufruf von make bei Python bzw. make world bei Ocaml benötigt.

	Opt(20)	Opt(40)	Opt(80)	Opt(Unlimited)
Python	124,08%	132,49%	141,46%	149,28%
Ocaml	80,90%	81,00%	81,14%	80,84%

Tabelle 3.3: Compile-Zeit in Prozent

Für Python steigert sich mit jeder Stufe die Zahl der kopierten Blöcke und somit die Komplexität und Compile-Zeit.

Erstaunlich ist jedoch das Ergebnis bei Ocaml. Die Compile-Zeit reduziert sich um fast 20%. Das liegt vermutlich daran, dass Ocaml zum größten Teil in Ocaml geschrieben ist. Nur der Bytecode-Interpreter ist in c geschrieben. Das heißt der Interpreter interpretiert den Ocaml-Compiler, welcher den Rest der Compilierung durchführt. Dadurch, dass der Interpreter schneller wird, reduziert sich folglich die Compile-Zeit. Dieses unerwartete Ergebnis zeigt, dass die Optimierung auch abseits der recht primitiven Testskripte (Fibonacci, Mandelbrot) gute Ergebnisse liefert.

Tabelle 3.3 enthält nur den Mittelwert über alle Prozessoren. Die Ergebnisse von Python weichen nicht weit von diesem ab, jedoch im Fall von Ocaml gibt es einige interessante Unterschiede. Abbildung 3.1 stellt die gleichen Daten für jeden Prozessortyp einzeln dar. Wie sich auch bei den anderen Benchmarks herausgestellt hat, gibt es bei den älteren Prozessoren die besten Ergebnisse.

3.4 Mandelbrot-Menge

3.4.1 Python

Wie in Abbildung 3.2 zu sehen ist, sind die Ergebnisse bei Python nicht ganz eindeutig. Auf dem Athlon 64, Opteron und Phenom werden die Testskripte durch die Optimierung etwa $1,1$ Mal so schnell ausgeführt. Beim Intel CoreDuo scheint es weder eine Verbesserung noch eine Verschlechterung zu geben. Bei den leichtgewichtigen Prozessoren dem Intel Atom 330 und dem Netbook Prozessor AMD E-350 scheint es nur bei einem Block-Limit von 40 eine leichte Verbesserung zu geben. Man muss natürlich bedenken, dass diese Verbesserungen durch eine bis zu 50% höhere Compile-Zeit und eine bis zu 25% größere Ausführungsdatei erkauft werden müssen. Die stark erhöhte Dateigröße könnte auch einer der Gründe sein, weshalb gerade die leichtgewichtigen Prozessoren schlechter abschneiden.

3.4.2 Ocaml

Abbildung 3.3 zeigt bei Ocaml ganz im Gegensatz zu Python einen Leistungsgewinn bei allen Prozessortypen von bis zu $1,45$. Wie bei Python scheinen die leichtgewichtigen Prozessoren trotzdem nicht so gut abzuschneiden.

3.5 Fibonacci-Zahlen

3.5.1 Python

Aus Abbildung 3.4 ergibt sich ein ähnliches Bild wie in Abbildung 3.2 mit der Ausnahme, dass der Intel Atom wesentlich besser und der AMD Phenom wesentlich schlechter abschneidet. Der Netbook-Prozessor E-350 scheint hier besonders schlechte Ergebnisse zu liefern.

3.5.2 Ocaml

Ocaml verbessert seine Leistung, wie in Abbildung 3.5 zu sehen ist, bei allem Prozessormodellen. Im Fall vom Athlon 64 und Opteron sogar um fast das $2,4$-fache. Auch der Intel Atom scheint bei den Fibonacci-Zahlen im Gegensatz zu Abbildung 3.3 deutlich besser abzuschneiden.

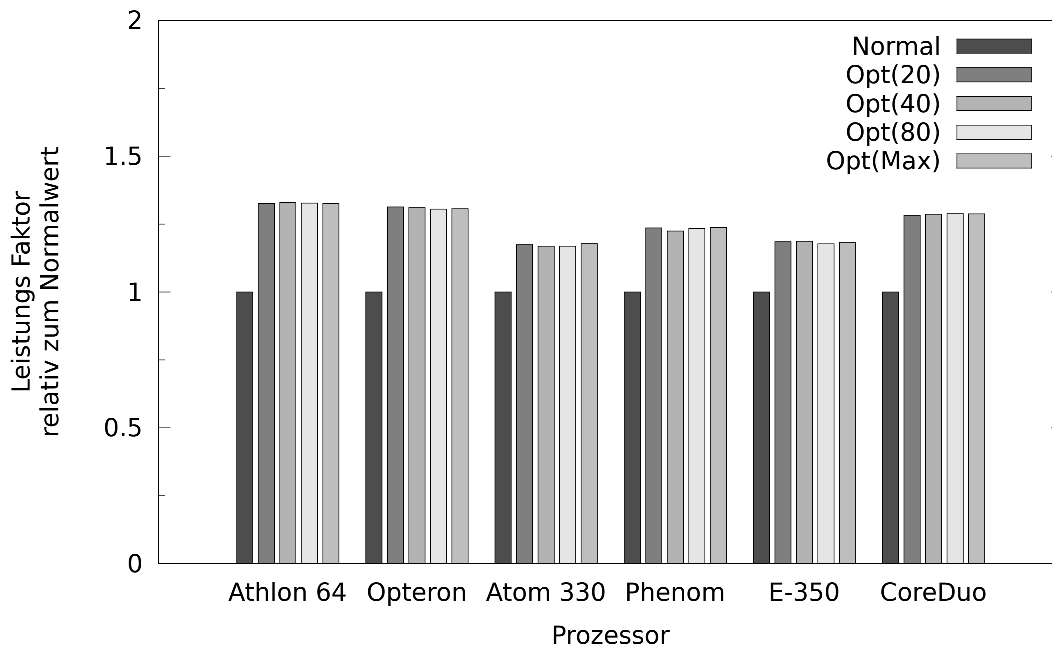

Abbildung 3.1: Compile-Zeit von Ocaml 3.12.1

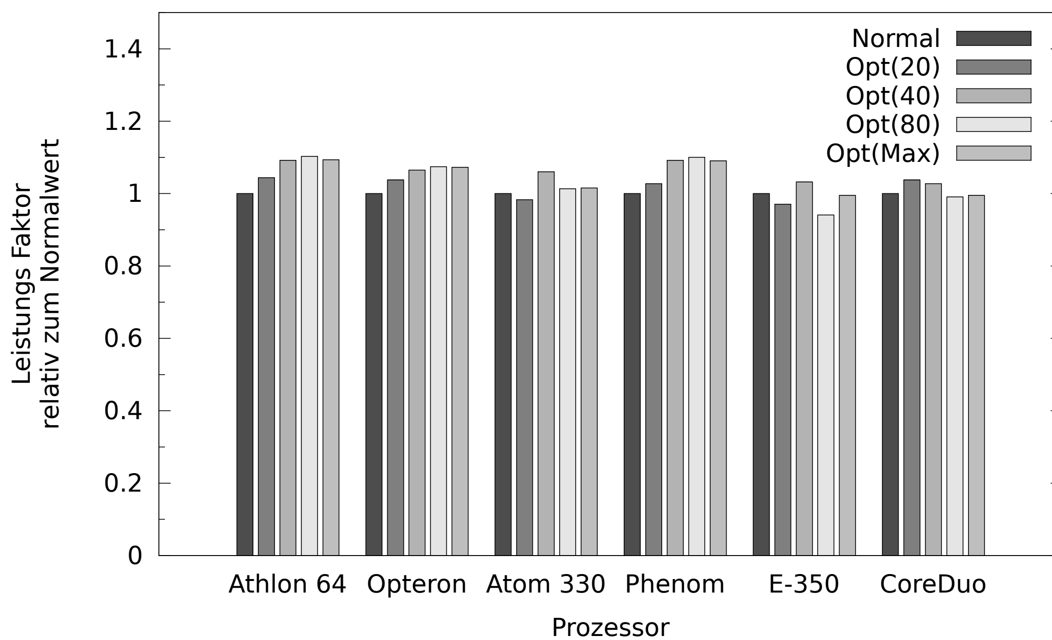

Abbildung 3.2: Mandelbrot-Menge mit Python 2.7.1

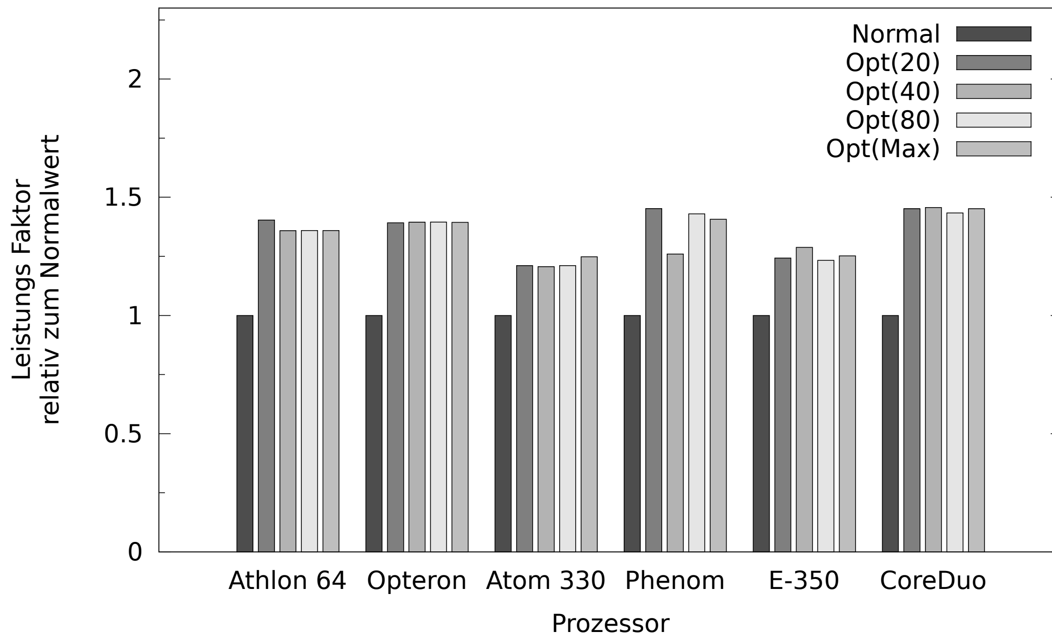

Abbildung 3.3: Mandelbrot-Menge mit Ocaml 3.12.1

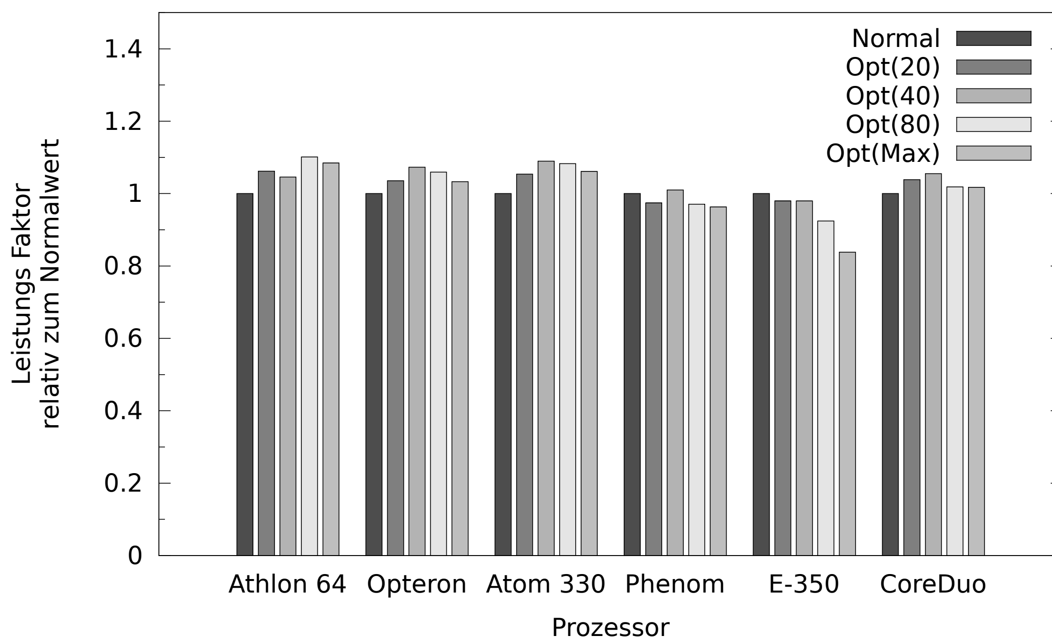

Abbildung 3.4: Fibonacci-Zahlen mit Python 2.7.1

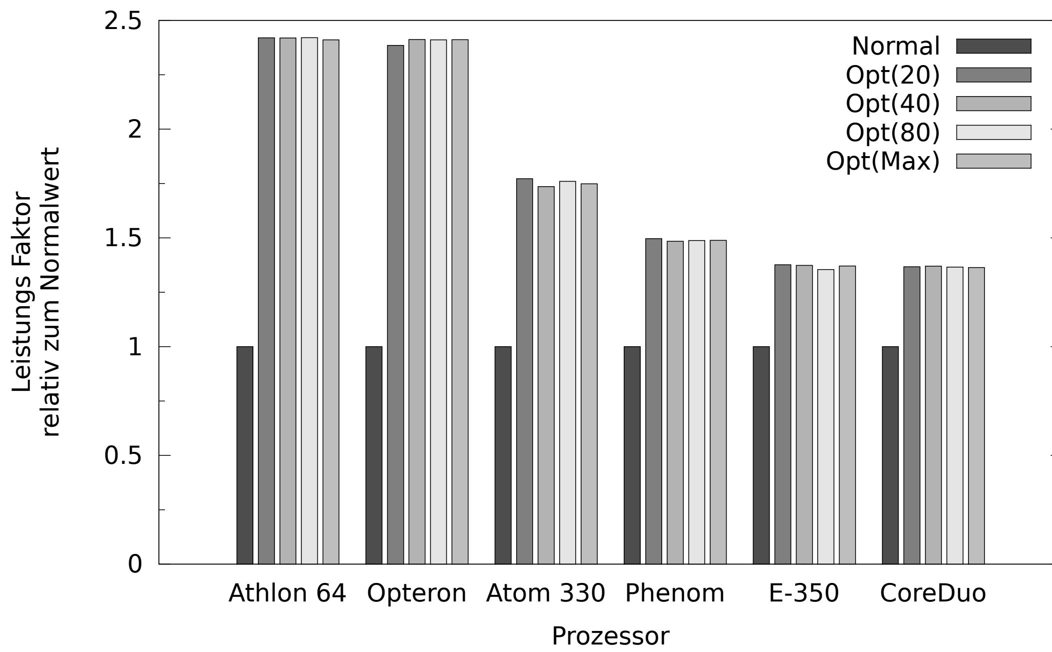

Abbildung 3.5: Fibonacci-Zahlen mit Ocaml 3.12.1

KAPITEL 4

Verwandte Arbeiten

Das einfachste Interpreter-Design besteht aus einem großen Switch-Statement in einer Schleife. Der Bytecode-Dispatch erfolgt dabei über das Switch-Statement und die Funktionalität befindet sich in den Case-Statements. Dieser Ansatz ist zwar sehr einfach und kompatibel mit dem ANSI-C-Standard, jedoch ist es oft besser einen Direct-Threaded Ansatz zu verwenden. Dabei besteht der Bytecode aus direkten Speicheradressen auf Code der die Funktionalität enthält. Diese beiden Ansätze wurden sehr ausführlich in der Arbeit von ERTL UND GREGG [EG01], besonders im Bezug auf die verschiedensten existierenden und theoretischen Branch-Prediction-Mechanismen, analysiert.

Nach ERTL UND GREGG [EG03b] können allerdings beim Ansatz mit Switch-Statement signifikante Performance-Gewinne erreicht werden, wenn der Sprungbefehl am Ende des Case-Blockes durch ein weiteres Switch-Statement ersetzt wird, das ein Dispatch durchführt aber dann wieder auf die Case-Blöcke des ursprünglichen Switch-Statements springt. Dafür müssen existierende Switch-Statements manuell umgebaut werden und der Compiler erzeugt für jedes zusätzliche Switch eine eigene Jump-Table. Der Gegenstand dieser Arbeit ist ein Compiler-Pass, der nach passenden Switch-Statements sucht und diese Optimierung automatisch durchführt, wobei nur eine Jump-Table erzeugt wird. Außerdem müssen keine Veränderungen am Sourcecode durchgeführt werden und der Compiler-Pass kann nicht nur auf Interpreter, sondern auf beliebigen Code angewandt werden.

Ein weiterer Ansatz, um die Indirect-Branch-Prediction in Interpretern zu optimieren, ist eine Kombination von dynamischer bzw. statischer Replication und Superinstructions [EG03a]. Kommt eine Instruktion nur einmal vor wird ihr Indirect-Branch durch einen Branch-Target-Buffer korrekt vorhergesagt. Replication erzeugt mehrere Replikate der gleichen Instruktion und ersetzt sie im Programm, wodurch jedes Replikat korrekt vorhergesagt werden kann. Superinstruktions kombinieren mehrere kleinere Instruktionen zu einer großen, wodruch der Dispatch nur einmal durchgeführt werden muss [ETK06]. Wie sich zeigte verbessern Superinstruktionen außerdem die Indirect-Branch-Prediction und die dadurch erreichte Performance-Verbesserung überwiegt in der Regel den verursachten Overhead.

Interpreter mit dynamischen, zur Laufzeit erzeugten, Superinstructions werden auch als

Code-copying Interpreter bezeichnet. Ein grundlegendes Problem dieses Ansatzes ist, dass Maschinencode, der von einem C-Compiler erzeugt wurde, Abhängigkeiten außerhalb der Umgebung im C-Code haben kann. Wegen dieser möglichen Abhängigkeiten ist es schwierig die Ausführbarkeit des Codes an einer anderen Stelle im Speicher zu garantieren. Normalerweise werden nicht kopierbare Instruktionen von der Integration in Superinstruktions ausgeschlossen [ETK06]. GREGORY B. PROKOPSKI AND CLARK VERBRUGGE verwenden in ihrer Arbeit [PV08] eine neue gcc-Erweiterung, die speziell für Code-Copying gemacht wurde und die Optimierungen des Compilers und damit die möglichen Abhängigkeiten des erzeugten Codes auf einen bestimmten Bereich einschränkt. Dadurch wird es möglich die Sicherheit von Code-Copying zu garantieren.

Die Arbeit [ETK06] beschäftigt sich mit der Implementierung von Replication und Superinstructions für die Cacau JVM. Es wird eine Lösung vorgeschlagen, um die Quickening-Optimierung und Bytecode-Instructions, die Exceptions werfen können, effizient in Superinstructions integrieren zu können.

Alle diese Arbeiten beschäftigen sich damit die Indirect-Branch-Prediction bei Interpretern mit Direct-Threaded-Code zu optimieren. Der in dieser Arbeit vorgestellte Compiler-Pass versucht allerdings das Konzept von Direct-Threaded-Code für die Optimierung von Switch-Instruktionen zu verwenden.

ANTON KOROBEYNIKOV beschreibt in seiner Arbeit [Kor07] die Möglichkeiten ein Switch-Statement in Maschinencode umzusetzen und den konkreten Algorithmus der in LLVM verwendet wird. Dabei hängt es maßgeblich von der Anzahl der Case-Zweige und der Dichte der Integerwerte ab, welche Umsetzung in Maschinencode vom Compiler gewählt wird.

KAPITEL 5

Zusammenfassung und Ausblick

Die Benchmarks haben deutlich gezeigt, dass die Optimierung für Python in seinem momentanen Zustand nicht wirklich sinnvoll ist. In einigen Fällen wurde zwar ein geringer Performance-Gewinn erzielt, der allerdings durch die Nachteile einer deutlich höheren Compile-Zeit und Dateigröße wieder ausgeglichen wurde. Das heißt allerdings nicht dass eine Optimierung unmöglich ist. Sie ist nur ohne Veränderung am Quellcode der Dispatch-Schleife schwer durchzuführen. Die Ergebnisse von Ocaml sind im Gegensatz dazu erstaunlich gut, was zeigt, dass das Prinzip des in dieser Arbeit vorgestellten Passes funktioniert und tatsächlich ordentliche Leistungsgewinne möglich sind. Ein überraschendes Ergebnis war die bedeutende Verbesserung der Compile-Zeit von Ocaml bedingt durch die Tatsache, dass Ocaml größtenteils in Ocaml programmiert wurde. Vielleicht könnte der Bytecode-Compiler von Ocaml in nachfolgenden Arbeiten als realistischere Testeingabe für den Interpreter verwendet werden.

Die vorliegenden Ergebnisse lassen darauf schließen, dass das Block-Limit für den Optimierungs-Pass eher niedrig angesetzt werden sollte, denn bei einem höheren Limit wird zu viel redundanter Code erzeugt, was scheinbar die positiven Effekte einer geringeren Indirect-Branch-Misprediction-Rate bei den meisten Prozessoren wieder ausgleicht. Da Python so viel zusätzlichen Code innerhalb der Dispatch-Schleife hat, ist ein sehr hohes Block-Limit für eine effektive Optimierung notwendig. Das Problem hierbei liegt allerdings nicht darin, dass die Dispatch-Schleife von Python zu kompliziert ist um sie zu optimieren, denn das konnte durch Tests auf einem simulierten Prozessor und der Inspektion des erzeugten Assemblercodes ausgeschlossen werden. Es wäre interessant in einer weiteren Arbeit die genaue Ursache der schlechten Leistung von Python mittels Performance-Counter zu untersuchen.

Das LLVM-Framework sammelt Profiling-Informationen zur Laufzeit und kann diese für Profil geleitete Transformationen sowohl zur Laufzeit als auch im Leerlauf verwenden [LA04]. Eine Möglichkeit die Anzahl der zu kopierenden Blöcke zu reduzieren wäre es diese Profiling-Informationen bei der Entscheidung, welche Unterbäume verworfen werden sollen, einzubeziehen (siehe Abschnitt 2.3.4). Dann könnte im Fall von Python gezielt der selten ausgeführte Exception-Handling-Code, der aber gleichzeitig einen Großteil des überschüssigen Codes in der Dispatch-Schleife darstellt, verworfen werden.

Eine Möglichkeit die Compile-Zeit zu verbessern wäre die Optimierung nicht im Zwischencode durchzuführen, sondern erst sehr spät in der Pipeline den fertigen Maschinencode zu duplizieren. Das hat den Vorteil, dass alle Transformationen, die den Zwischencode in Maschinencode umwandeln, eine geringere Datenmenge verarbeiten müssen. Hierfür wurde im Laufe dieser Arbeit schon ein Prototyp entwickelt, der, obwohl noch unausgereift, sehr vielversprechend erscheint. Dieser Prototyp könnte im Laufe einer Nachfolgearbeit weiterentwickelt werden.

Es wäre außerdem interessant, inwieweit man die Dispatch-Schleife von Python verändern müsste, sodass sie sich besser für eine Optimierung mit dem in dieser Arbeit vorgestellten Verfahren eignet. Eine einfache Möglichkeit wäre beispielsweise den selten ausgeführten Code aus der Dispatch-Schleife zu entfernen und in eine eigene Funktion zu geben, sodass statt des gesamten Codes nur mehr ein Funktionsaufruf dupliziert werden muss.

Literaturverzeichnis

[Bel73] James R. Bell. Threaded code. *Commun. ACM*, 16:370–372, June 1973.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *In Euro-Par 2001*, pages 403–412. Springer LNCS, 2001.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/.

[ETK06] Martin Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the cacao jvm interpreter. *Journal of .NET Technologies*, Vol. 4(ISBN 80-86943-13-5):25–32, 2006.

[Kor07] Anton Korobeynikov. Improving Switch Lowering for The LLVM Compiler System. In *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007)*, Moscow, Russia, May 2007.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See http://llvm.cs.uiuc.edu.*

[PV08] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing phi-nodes. In *POPL'95*, pages 62–73, 1995.