Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht

Compiler und Compiler-Optimierungen

vorgelegt von

Michael Blesel

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Abstract

Compiler sind ein äußerst wichtiger Bestandteil der Softwareentwicklung aber für viele Entwickler eher eine Art Blackbox. Um optimal performende Programme zu erhalten ist ein generelles Verständnis über Compiler eine große Hilfe. Dieser Bericht beschreibt den Aufbau von Compilern und den genauen Ablauf einer Kompilierung. Hierbei wird der Fokus auf den Optimizer Teil des Compilers gesetzt und es werden verschiedene mögliche Optimierungen exemplarisch vorgestellt.

Contents

1 Compiler allgemein

1.1 Einleitung

Compiler sind eines der wichtigsten Werkzeuge für Softwareentwickler. Sie erlauben das Programmieren in Hardware unabhängigen Programmiersprachen mit einem hohen Abstraktionslevel. Sie bieten Syntax- und (teilweise) Semantik-Überprüfungen zur Kompilierzeit und ermöglichen das Ausführen von Software auf allen möglichen Architekturen. Des weiteren beinhalten moderne Compiler äußerst mächtige Optimizer, welche Code Optimierungen vornehmen und einen großen Einfluss auf die Programm-Performance haben können. Um möglichst effiziente Programme zu entwickeln kann das korrekte Benutzen und ein allgemeines Verständnis über den Compiler und seine Optimierungsschritte ein wichtiger Bestandteil des Entwicklungsprozesses sein.
Trotz ihrer hohen Relevanz für die Softwareentwicklung ist ein Compiler für viele Programmierer eine Art Blackbox die Code entgegen nimmt und Binaries ausgibt. Dieser Bericht hat das Ziel den allgemeinen Ablauf eines Kompilierungsvorganges genauer zu erklären und geht besonders auf die Schritte des Optimizer ein um ein grobes Verständnis über die möglichen Arten von Optimierungen die ein Compiler vornehmen kann zu schaffen. Zuerst werden die verschiedenen Phasen einer Kompilierung erläutert. Danach wird genauer auf den Optimizer eingegangen und ausgewählte Optimierungen werden anhand von Beispielen im Detail betrachtet.

1.2 Aufbau eines Compilers

1.2.1 Aufbau

Figure 1.1: Struktur eines Compilers

In 1.1 wird die Struktur eines allgemeinen Compilers dargestellt. Ein Compiler besteht aus drei Haupt-Komponenten.

1. **Frontend:** Das Frontend eines Compilers ist Programmiersprachen spezifisch. Es ist dafür da den Programmcode des Entwicklers zu verarbeiten und für den

weiteren Gebrauch im Compiler vorzubereiten. Das Frontend ist die Schnittstelle zwischen Compiler und Benutzer. Es parsed den Programmcode und prüft als erstes auf eine korrekte Syntax um festzustellen ob der gegebene Code überhaupt übersetzbar ist. Als Schnittstelle zum Benutzer ist für das Frontend die Ausgabe von aussagekräftigen Fehlermeldungen über vorhandene Syntax-Fehler besonders wichtig. Die meisten modernen Compiler liefern dem Benutzer allerdings nicht nur Syntax-Fehler, welche das Programm unübersetzbar machen, sondern auch Warnungen über mögliche Semantik Fehler im Programm. Um dies zu ermöglichen laufen im Frontend neben dem Lexer und dem Parser auch verschiedene statische Analyse Tools, welche bestimmte Code Muster erkennen können die wahrscheinlich einen Semantik Fehler im Programm darstellen. Beispiele hierfür wären z.B. das Zugreifen auf uninitialisierte Variablen, das Erkennen von Tautologien in if- oder Schleifen-Bedingungen oder die Ausgabe von Variablen die zwar deklariert wurden aber nirgendwo anders im Code verwendet werden. Je nach Compiler und den statischen Analyse Methoden die verwendet werden können allerdings auch Warnungen zu weit aus komplexeren möglichen Fehlern ausgegeben werden.
Die Hauptaufgabe des Frontend ist es aber den Code in eine Form umzuwandeln mit der der Rest des Compilers weit aus besser arbeiten kann als mit Sourcecode. Hierfür wird der Sourcecode zuerst in einen 'Abstract Syntax Tree' (AST) umgeformt. Der AST eines Programms ist, wie der Name vermuten lässt, eine Baumstruktur aller aller Anweisungen des Programms. Hierbei werden schon die ersten Abstraktionen der verwendeten Programmiersprache abgebaut und in ihre elementaren Bestandteile Zerlegt (genauere Erklärung in 1.3.1).
Im nächsten Schritt wird der AST in eine so genannte 'Intermediate Representation' (IR) umgeformt. Die Intermediate Representation ist die Form des Codes welche im restlichen Teil des Compiler benutzt wird um Optimierungen vorzunehmen und um den finalen Maschinencode zu erstellen. Die exakte Form der 'Intermediate Representation' unterscheidet sich von Compiler zu Compiler und wird in späteren Teilen des Berichts am Beispiel des LLVM Compilers genauer erklärt.
Nach diesem Schritt ist die Arbeit des Frontend beendet und der nächste Teil des Compilers beginnt.

2. **Optimizer:** Der Optimizer ist ein großer Teil des Compilers, welcher allerdings optional ist (falls der Compiler Aufruf Optimierungen abschaltet). Er ist weder direkt Programmiersprachen- noch direkt Hardware-abhängig, aber Informationen über z.B. die Zielarchitektur können bei manchen Optimierungen verwendet werden um möglichst effizienten Code zu erstellen. Wie aus dem Namen hervorgeht ist der Optimizer hauptsächlich dafür zuständig den Code zu transformieren um eine bessere Performance des Programms zu erreichen. Er arbeitet mit der Intermediate Representation des Codes und wendet eine Vielzahl an Optimierungsmethoden an. Beispiele für Optimierungen sind 'dead code elimination', 'Function inlining' und verschiedenste Schleifenoptimierungen.
Welche Optimierungen vorgenommen werden kann beim Aufruf des Compilers vom Benutzer entweder explizit angegeben werden oder es wird zur Compile-Zeit

dynamisch entschieden. Die am häufigsten benutzen Einstellungen die ein Benutzer dem Compiler übergeben kann sind die verschiedenen Optimierungslevel (-O0,...,-O3), welche jeweils eine Sammlung von verschiedenen Optimierungsmethoden beinhalten die ausgeführt werden sollen. Wie genau die verschiedenen Optimierungen ausgeführt werden ist abhängig vom benutzen Compiler und wird im Optimizer Kapitel (2) am Beispiel des LLVM Compilers gezeigt.
Nachdem alle gewollten Optimierungen durchgeführt wurden übergibt der Optimizer den transformierten Code an das Backend.

3. **Backend:** Das Backend ist dafür zuständig aus der Intermediate Representation spezifischen Maschinencode zu formen welcher auf der Zielarchitektur ausgeführt werden kann. Das Backend ist daher auf Informationen über die Zielarchitektur angewiesen und muss für jede Architektur die der Compiler unterstützten soll speziell angepasst sein. Im Backend wird der Code in eine passende Assembler Form (beziehungsweise in das genaue Format das eine Executable auf der Zielarchitektur haben muss) gebracht. Dies ist der letzte Schritt des Compilers und die erstellten object Dateien müssen nun gegebenenfalls noch gelinkt und zu einer Binary gemacht werden. Auf die exakten Details des Backend und die des Linker wird in diesem Bericht nicht weiter eingegangen, da das Hauptthema der Optimizer ist.

1.2.2 LLVM

Die Bisher genannten Konzepte gelten allgemein für alle Compiler aber ihre genaue Implementierung unterscheidet sich von Compiler zu Compiler. Um spezifische Beispiele zeigen zu können habe ich den LLVM Compiler gewählt, welcher in den nächsten Kapiteln für alle konkreten Beispiele benutzt wird.
Das LLVM Projekt [Pro19a] ist ein Open-Source Compiler Projekt entwickelt in C++. LLVM bietet verschiedene Libraries und Tools für die Compilerentwicklung und beinhaltet verschiedene Unterprojekte aus dem Compiler Bereich. Der wichtigste Teil ist der LLVM-Core, ein vollständiger Compiler mit austauschbaren Front- und Back-Ends. Bei der Entwicklung von LLVM wird ein großer Wert auf Modularität und Wiederverwendbarkeit gesetzt was dazu führt, dass LLVM viel in der Forschung eingesetzt wird da es sich gut für Modifikationen und eigene Compiler-Tools eignet.

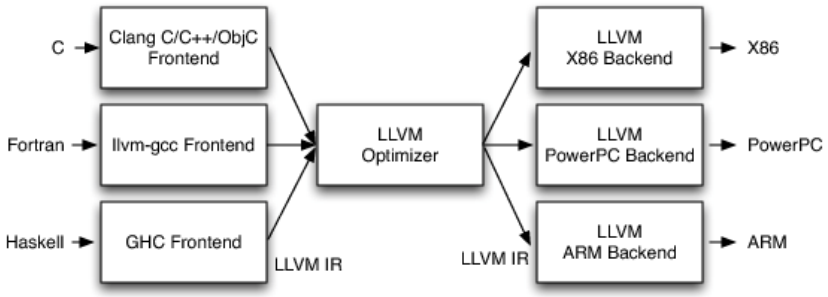

Figure 1.2: Aufbau des LLVM Compilers

In 1.2 sieht man des modularen Aufbau des LLVM Compilers. Man sieht die zuvor besprochenen drei Hauptbestandteile eines Compilers. Die Modularität von LLVM erlaubt es verschiedene Front- und Back-Ends mit dem Optimizer zu verbinden. An der Schnittstelle zwischen den Komponenten wird der Code in Form der LLVM Intermediate Representation (IR) übergeben. Durch diesen Aufbau kann z.B. ein LLVM Compiler für eine neue Sprache entwickelt werden indem nur ein neues Frontend für die Sprache programmiert wird welches korrekten IR Code ausgibt und der Rest der bereits bestehenden LLVM Infrastruktur kann von da an weiterverwendet werden und man hat Zugang zu dem kompletten Optimizer und allen unterstützten Backends von LLVM. Gleichermaßen könnte man eine Backend für eine neue Hardware entwickeln welches nur dazu in der Lage sein muss IR Code zu übersetzen und kann danach alle Programmiersprachen für die es LLVM Frontends gibt mit dem neuen Backend benutzen. In dieser Hinsicht unterscheidet sich LLVM von vielen anderen Compilern, welche eher monolithischen Code haben und daher Frontend, Optimizer und Backend nicht so einfach voneinander trennbar sind.

1.3 Der Kompiliervorgang

Im Folgenden werden die einzelnen Schritte einer Kompilierung vorgestellt. Wir starten mit dem Sourcecode eines simplen C-Programms und betrachten die einzelnen Transformationsschritte die der Code während der Ausführung einer Kompilieranweisung durchläuft.

```
1  int multiply(int a, int b)
2  {
3      return a*b;
4  }
5
6  int main()
7  {
8      int a = 10;
9      int b = 5;
10     int c = multiply(a,b);
11     return 0;
12 }
```

Figure 1.3: Ein simples C Programm

1.3 ist der betrachtete C Code. Es handelt sich um ein einfaches Programm welches aus einer `main` und einer `multiply` Funktion besteht. Es werden Variablen deklariert und ein Funktionsaufruf durchgeführt.

1.3.1 Frontend

Der erste Schritt des Frontends ist das umwandeln des C Codes in einen Abstract Syntax Tree (AST). Im AST ist das übersetzte Programm in seine atomaren Operationen zerlegt und als Baumstruktur gespeichert welche die Reihenfolge und die Art der jeweiligen Operationen beinhaltet. In 1.4 wird der AST für die `main` Funktion unseres Beispiels gezeigt. Man sieht, dass die Wurzel des Baumes die Deklaration einer Funktion mit dem Namen 'main' und vom Typ 'int' ist. folgt man den Ästen im Baum sieht man die folgenden Variablendeklarationen und die Zuweisung ihrer Initialwerte.
Da dieser Bericht sich im Kern mit dem Optimizer beschäftigt wird die exakte Funktionsweise des ASTs hier nicht weiter im Detail erklärt. Nachdem der AST erstellt wurde muss das Frontend nun noch IR Code daraus formen und diesen dann an den Optimizer übergeben.

```
FunctionDecl 0x55e9686def20 <line:8:1, line:16:1> line:8:5 main 'int ()'
`-CompoundStmt 0x55e9686df488 <line:9:1, line:16:1>
  |-DeclStmt 0x55e9686df050 <line:10:5, col:15>
  | `-VarDecl 0x55e9686defd0 <col:5, col:13> col:9 used a 'int' cinit
  |   `-IntegerLiteral 0x55e9686df030 <col:13> 'int' 10
  |-DeclStmt 0x55e9686df100 <line:11:5, col:14>
  | `-VarDecl 0x55e9686df080 <col:5, col:13> col:9 used b 'int' cinit
  |   `-IntegerLiteral 0x55e9686df0e0 <col:13> 'int' 5
  |-DeclStmt 0x55e9686df440 <line:13:5, col:26>
  | `-VarDecl 0x55e9686df130 <col:5, col:25> col:9 c 'int' cinit
  |   `-CallExpr 0x55e9686df230 <col:13, col:25> 'int'
  |     |-ImplicitCastExpr 0x55e9686df218 <col:13> 'int (*)(int, int)' <FunctionToPointerDecay>
  |     | `-DeclRefExpr 0x55e9686df190 <col:13> 'int (int, int)' Function 0x55e9686ded70 'multiply' 'int (int, int)'
  |     |-ImplicitCastExpr 0x55e9686df260 <col:22> 'int' <LValueToRValue>
  |     | `-DeclRefExpr 0x55e9686df1b0 <col:22> 'int' lvalue Var 0x55e9686defd0 'a' 'int'
  |     `-ImplicitCastExpr 0x55e9686df278 <col:24> 'int' <LValueToRValue>
  |       `-DeclRefExpr 0x55e9686df1d0 <col:24> 'int' lvalue Var 0x55e9686df080 'b' 'int'
  `-ReturnStmt 0x55e9686df478 <line:15:5, col:12>
    `-IntegerLiteral 0x55e9686df458 <col:12> 'int' 0
```

Figure 1.4: Ausschnitt aus dem AST des Programms

1.3.2 Optimizer

Der Optimizer erhält vom Frontend IR Code und wendet nun seine Optimierungen auf diesen an. In 1.5 sehen wir den optimierten IR Code den der Optimizer erstellt.

Eine kurze Einweisung in LLVM IR

Für die folgenden Beispiele im Optimizer Kapitel wird ein grobes Wissen über die Funktionsweise der LLVM IR Sprache benötigt. Die wichtigsten Konzepte können gut anhand des Code Beispiels 1.5 erklärt werden.
LLVM IR ist eine low-level Sprache mit Assembler ähnlichen Operationen, welche jedoch im Gegensatz zu echtem Assembler Code Hardware unabhängig sind. In Zeilen 2 und 9 des Beispiels sieht man die Definition von Funktionen. Das Keyword `define` wird gefolgt von dem Rückgabetyp der Funktion (in unserem Beispiel `i32`, ein 32-Bit Integer) und dem Funktionsnamen. Oberhalb der Funktionsdefinitionen befinden sich Listen von Attributen die der Compiler der Funktion zugeordnet hat und welche während dem

```
1  ; Function Attrs: noinline nounwind uwtable
2  define dso_local i32 @multiply(i32 %a, i32 %b) #0 {
3  entry:
4    %mul = mul nsw i32 %a, %b
5    ret i32 %mul
6  }
7
8  ; Function Attrs: noinline nounwind uwtable
9  define dso_local i32 @main() #0 {
10 entry:
11   %call = call i32 @multiply(i32 10, i32 5)
12   ret i32 0
13 }
```

Figure 1.5: IR Code des Programms

Optimieren Informationen über die Art und das Verhalten der Funktion geben. Die Körper beider Funktionen beginnen mit einem `entry:` Label. Diese Label eröffnen einen so genannten Block innerhalb der Funktion und Jumps innerhalb des IR Codes werden durch Branch-Instruktionen zu den jeweiligen Labels der Blöcke umgesetzt.
In den Zeilen 4 und 11 sehen wir eines der wichtigsten Konzepte von LLVM IR. Hier werden Registern Werte zugewiesen. Da LLVM IR hardwareunabhängig ist wird in der Sprache davon ausgegangen, dass unendlich viele Register zur Verfügung stehen und jeder Wert bekommt daher sein eigenes Register. LLVM IR folgt der so genannten 'Static Single Assignment' Form (SSA). Dies bedeutet, dass ein einmal zugewiesener Wert sich niemals ändern kann und stattdessen bei Modifikationen eines Wertes ein neues Register für den geänderten Wert erstellt wird. Der Code wird in SSA Form gebracht um viele Optimierungen zu vereinfachen und um bessere Aussagen über den Status einer Variable zu bestimmten Stellen im Code machen zu können. Es ist in SSA Form zum Beispiel sehr einfach zu erkennen ob eine bestimmte Variable im Code niemals verwendet wird oder welchen Wert sie nach einem konditionalen Code-Block hat.
Dies sind die wichtigsten Grundkonzepte von LLVM IR. Die verschiedenen Instruktionen im LLVM IR Befehlssatz, wie z.B. `add, mul, call, ret` sind relativ selbsterklärend und werden Folgenden einzeln erklärt wenn es nötig für das Verständnis eines Beispiels ist.
Das gezeigte Code Beispiel ähnelt noch stark dem original C Code und die einzige wirkliche Optimierung die hier stattgefunden hat ist das Auslassen der Deklarationen der beiden Variablen a und b, deren Werte direkt in den Funktionsaufruf der `multiply` Funktion eingefügt wurden. Dies ist legitim, da beide Variablen im Code feste Werte zugewiesen bekommen haben, welche zu Kompilierzeit eindeutig sind und die Variablen nirgendwo sonst weiter verwendet werden. Dies ist ein Beispiel für eine simple Optimierung die Compiler oft vornehmen. Berechnungen deren Ergebnisse erst dynamisch während dem Lauf des Programms bestimmt werden können werden direkt zur Kompilierzeit

ausgeführt.
Nachdem der Optimizer alle geforderten Optimierungen durchgeführt hat wird der resultierende IR Code an das passende Backend übergeben.

1.3.3 Backend

Das Backend hat nun die Aufgabe Maschinencode zu erstellen, welcher auf der gewollten Hardware ausgeführt werden kann. In 1.6 sehen wir den Assembler Code für unser Beispielprogramm der auf meiner Maschine erstellt wurde. Da das Backend kein direktes Thema dieses Berichts ist wird auf den Assembler Code nicht weiter im Detail eingegangen und das Beispiel steht hier der Vollständigkeit halber.

```
1  main:                                   # @main
2      .cfi_startproc
3  # %bb.0:                                # %entry
4      pushq   %rbp
5      .cfi_def_cfa_offset 16
6      .cfi_offset %rbp, -16
7      movq    %rsp, %rbp
8      .cfi_def_cfa_register %rbp
9      movl    $10, %edi
10     movl    $5, %esi
11     callq   multiply
12     xorl    %eax, %eax
13     popq    %rbp
14     .cfi_def_cfa %rsp, 8
15     retq
```

Figure 1.6: Assembler des Programms

2 Der Optimizer

In diesem Kapitel wird sich genauer mit dem Optimizer beschäftigt und wir betrachten die verschiedenen Optimierungsschritte und ein paar ausgewählte Optimierungen anhand von Beispielen.

2.1 Aufgaben des Optimizers

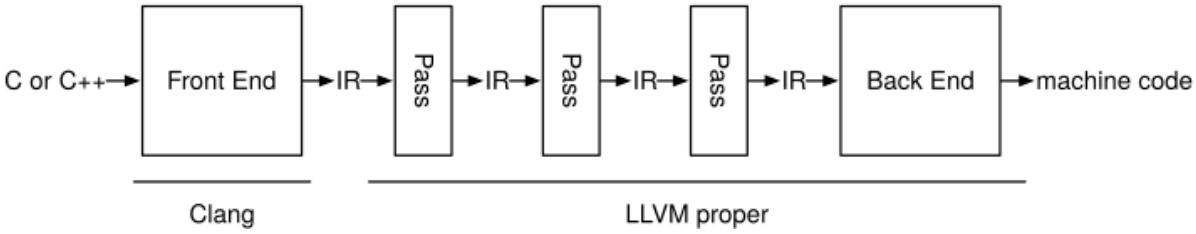

Figure 2.1: Aufbau des LLVM Optimizers

In 2.1 sehen wir den Aufbau des LLVM Optimizers. Der Optimizer enthält eine Vielzahl an möglichen Optimierungen. Diese sind in der Form von Optimierungs-Pässen implementiert. Ein Pass hat eine genaue Aufgabe und nimmt als Input IR Code entgegen, welcher dann transformiert und wieder ausgegeben wird. Man könnte Optimierungs-Pässe als eine Art von Funktionen interpretieren welche eine spezifische Optimierung als Aufgabe haben. Innerhalb des Optimizers durchläuft der Code die verschiedenen Pässe in einer Pipeline. Die Ausführungsreihenfolge der Optimierungen ist hierbei wichtig, da manche Pässe den Code in einer Art transformieren die es für folgende Pässse einfacher oder schwerer machen kann ihre Optimierungen durchzuführen. Leider ist es kaum möglich eine optimale Reihenfolge der Pässe für ein Programm zu bestimmen und das so genannte Phase-Ordering wird teilweise dynamisch während der Laufzeit des Compilers bestimmt und ansonsten nach bestem Gewissen von den Entwicklern des Compilers vorher festgelegt. Trotz dessen kann man den Optimierungsvorgang in grobe Phasen unterteilen die immer in einer bestimmten Reihenfolge durchlaufen werden.
Zu Beginn des Optimizers wird als erstes ein 'Canonicalization' Schritt durchgeführt. Hierbei wird der vom Frontend geformte IR Code in eine gewünschte Form gebracht welche sich am besten für die folgenden Optimierungen eignet. Dieser Schritt ist notwendig, da das Frontend den original Sourcecode in relativ verbosen und sehr unoptimierten IR Code umwandelt. Dies stammt daher, dass das Frontend nur die Bedingung hat korrekten IR Code zu erstellen und daher den Sourcecode fast eins-zu-eins in seine IR Repräsentation übersetzt ohne dabei auf Performance zu achten. Dies beinhaltet das

```
1 if( !cond )    | if( cond )    | x = 0;
2 {              | {             |
3     x = 0;     |     x = 42;   | if ( cond )
4 }              | }             | {
5 else           | else          |     x = 42;
6 {              | {             | }
7     x = 42;    |     x = 0;    |
8 }              | }             |
```

Figure 2.2: Gleicher Effekt, verschiedener Code

Transformieren des Codes in SSA Form und eliminieren von nicht benutztem oder nicht erreichbarem Code. Des weiteren werden im Canonicalization Schritt Code Stücke in feste Muster umgeformt mit denen der Optimizer gut umgehen kann. In 2.2 sehen wir ein Beispiel hierfür. Alle drei Codebeispiele haben den gleichen Effekt, sind aber in verschiedenen Formen aufgeschrieben. Damit Optimierungen nicht alle Möglichkeiten für ein Stück Code mit der gleichen Semantik kennen müssen wird bei der Canonicalization versucht so viele bekannte Muster wie möglich in eine feste Form zu bringen welche die Pässe später erwarten können. Beispiele hierfür sind z.B. das Umformen von Negationen indem der Variablenwert invertiert wird oder das Umformen von Schleifen und ihren Bedingungen in eine konkrete Form.

Nach der Canonicalization werden die nicht so komplexen Optimierungen vorgenommen wie z.B. 'dead-code-elimination' oder das vereinfachen des Kontrollflusses. Am Ende der Optimierungsphase geschehen die komplexeren Optimierungen wie 'inlining', Schleifenoptimierungen oder 'Vectorization'.

2.2 Schleifenoptimierungen

Schleifenoptimierungen sind ein wichtiger Bestandteil des Optimizers für das Verbessern der Programm-Performance. In den meisten lang laufenden Programmen wird wahrscheinlich ein Großteil der Laufzeit innerhalb von Schleifen verbracht und der Code innerhalb eines Schleifenkörpers wird oft ausgeführt, was ihn natürlich stark für mögliche Optimierungen eignet. Ausgenommen sind hier Zeitintensive Operationen wie Ein-/Ausgabe, da diese eher Hardware beschränkt sind und der Compiler daher wenig Möglichkeiten hat diese zu Optimieren. Das Optimieren von Schleifen ist allerdings auch ein komplexes Problem, da es für Schleifencode auch komplizierter ist zu gewährleisten, dass sich die Semantik des Programms nicht ändert wenn der Code eines Schleifenkörpers modifiziert wird.
Im folgenden werden ausgewählte Schleifenoptimierungen etwas genauer betrachtet und aufgezeigt wie der LLVM Optimizer versucht Schleifen zu optimieren.

2.2.1 Loop-Canonicalization

Der erste Schritt in der Optimierung von Schleifen ist ähnlich zum Beginn des gesamten Optimizers und transformiert den Schleifen-Code in eine einheitliche Form die von den folgenden Optimierungen erwartet wird. In 2.3 sehen wir eine Schleife die Werte auf eine Zählervariable addiert. Der IR Code in 2.4 ist der normalisierte Schleifencode nach dem ersten Schritt der Schleifenoptimierung. Die Schleife wird in eine Form gebracht in der es einen Block für den gesamten Schleifenkörper mit einer Abfrage der Schleifenbedingung am ende gibt (Zeile 11). Je nach Zustand der Bedingung wird entweder zurück an den Anfang des Schleifenkörper Blocks oder oder zu einem Endblock der Schleife gesprungen. Das erste Betreten der Schleife wird auch durch einen direkten Jump zum Schleifenkörper Block umgesetzt. Hier kann bereits die erste Überprüfung der Schleifenbedingung ausgelassen werden, da in unserem Beispiel zu Kompilierzeit klar ist, dass die Schleife mindestens einmal ausgeführt wird. Alle Arten von Schleifen werden bei der Canonicalization in diese Form gebracht.
In unserem Beispiel sehen wir, dass der Code im Schleifenkörper noch sehr nah an dem original Sourcecode ist. In Zeilen 6 und 7 werden der Zähler- und der Ergebnis-Variable mit Hilfe so genannter 'Phi-Nodes' ihre Werte in der aktuellen Iteration zugewiesen. Die `phi` Anweisung funktioniert hier so, dass abhängig davon von welchem Block zu dem Schleifenkörper Bock gesprungen wurde die Variablenwerte gesetzt werden. Die Zählervariable `i` zum Beispiel erhält einen Wert von 0 wenn zuvor der `entry` Block ausgeführt wurde und den Wert des Registers `inc` wenn zuvor schon mindestens eine Iteration der Schleife ausgeführt wurde. In Zeilen 8 und 9 werden die Zähler- und Ergebnis-Variablen durch Additionen erhöht. In Zeile 10 wird die Schleifenbedingung überprüft um dann in Zeile 11 abhängig vom Ergebnis des Vergleichs zu entscheiden ob eine weitere Iteration ausgeführt werden soll oder ob die Schleife über den `end` Block verlassen werden soll.

```
1 int main()
2 {
3     int result = 0;
4     for(int i = 0; i < 10; i++)
5     {
6         result += i;
7     }
8     return result;
9 }
```

Figure 2.3: Eine Beispiel Schleife

```
define dso_local i32 @main() #0 {
entry:
  br label %for.body

for.body:                                         ; preds = %entry, %for.body
  %i.02 = phi i32 [ 0, %entry ], [ %inc, %for.body ]
  %result.01 = phi i32 [ 0, %entry ], [ %add, %for.body ]
  %add = add nsw i32 %result.01, %i.02
  %inc = add nsw i32 %i.02, 1
  %cmp = icmp slt i32 %inc, 10
  br i1 %cmp, label %for.body, label %for.end

for.end:                                          ; preds = %for.body
  %result.0.lcssa = phi i32 [ %add, %for.body ]
  ret i32 %result.0.lcssa
}
```

Figure 2.4: IR Code der Schleife nach Canonicalization

2.2.2 Loop-Unrolling

Eine häufig benutzte Schleifenoptimierung ist das so genannte 'Loop-Unrolling'. Hierbei wird der Code innerhalb des Schleifenkörpers mehrfach dupliziert, was weniger Schleifeniterationen zur Folge hat. Diese Optimierung spart zumindest einige Vergleichsoperationen und Jumps, da weniger Operationen ausgeführt werden müssen und kann ein Vorbereitungsschritt für weiterführende Optimierungen sein, welche der Optimizer im original Schleifencode noch nicht erkennen konnte. In Fällen in denen die Anzahl der Iterationen zur Kompilierzeit festgestellt werden kann ist es möglich die komplette Schleife mit dieser Optimierung aufzulösen, indem der Code des Schleifenkörpers so oft dupliziert wird wie es Iterationen gibt. Diese Art der Optimierung hat allerdings ein Wachsen der Codegröße zur Folge und wägt somit Performance gegen Codegröße ab. Wie viele Iterationen in einem solchen Fall 'unrolled' werden kann entweder vom Benutzer mit den richtigen Compiler Flags eingestellt werden oder wird dynamisch im Optimizer mit Hilfe einer Heuristik bestimmt.
Im Fall unseres Beispiels erlaubt das 'unrollen' der kompletten Schleife weitere Optimierungen und führt Schlussendlich zu dem Code in 2.5, wo alle Berechnungsschritte weg optimiert wurden. Nach dem kompletten auflösen der Schleife bleibt bei unserem Beispiel nur eine lange liste an Additionen mit den Werten von 0 bis 9 auf die Ergebnis Variable über. Da alle Werte dieser Additionen zur Kompilierzeit bekannt sind kann der Optimizer die Berechnung bereits ausführen und den Endgültigen Wert bestimmen. Dies ist natürlich ein best-case Szenario und kommt in realem Code nicht unbedingt in dieser Form vor, aber es ist ein gutes Beispiel dafür wie 'Loop-Unrolling' die Möglichkeit für weitere Optimierungen bieten kann.

```
1 ; Function Attrs: noinline norecurse nounwind uwtable
2 define dso_local i32 @main() #0 {
3 entry:
4   ret i32 45
5 }
```

Figure 2.5: Die vollständig optimierte Schleife

2.2.3 Loop-Interleaving

Eine ähnliche Optimierung zum 'Loop-Unrolling' ist das so genannte 'Loop-Interleaving'. Das Grundkonzept des Duplizieren von Code um mehrere Iterationen gleichzeitig in einem Schleifendurchlauf auszuführen kommt hier auch vor. Der Unterschied ist aber, dass der duplizierte Code zusätzlich noch umsortiert wird um eine schnellere Ausführung möglich zu machen. Stellen wir uns zum Beispiel eine Schleife vor die zwei lange Arrays Elementweise miteinander addiert und das Ergebnis in eine neues Array schreibt. In diesem Fall müssen bei jeder Addition drei Speicherbereiche geladen werden (Wert i in allen drei Arrays). Dies kann im schlimmsten Fall dazu führen, dass in jeder Iteration drei Cache misses entstehen weil das Array aus dem der momentan geladene Wert ist nicht mehr im Cache geladen ist. Führen wir nun 'Loop-Unrolling' auf diese Schleife aus ändert sich diese Problematik nicht, da die Operationen immernoch in genau der gleichen Reihenfolge ausgeführt werden. Beim 'Loop-Interleaving' werden die Operationen nach dem 'unrolling' so umsortiert, dass in dem Array Additionsbeispiel alle Zugriffe auf die jeweiligen Array gruppiert werden. Hierdurch gibt es eine große Chance, dass nach dem laden des ersten Elements aus einem Array die folgenden Werte auch bereits im Cache sind weil zwischendurch auf keinen anderen Speicherbereich zugegriffen wurde. Dies ist einer der Haupt-Vorteile von 'Loop-Interleaving', aber auch hier kann der optimierte Code sich nach der Transformation besser für neue Optimierungen eigenen. Das oben beschriebene Speicherzugriffsmuster für das addieren von n Arrayelementen in einer Iteration eignet sich zum Beispiel gut für eine Vektorisierung der Addition falls die benutze Hardware solche Operationen anbietet.

2.2.4 Minimieren von Schleifen-Code

Eine weitere Schleifenoptimierung die oft verwendet wird ist das Minimieren des Codes im Schleifenkörper. Oft kommt es vor, dass innerhalb des Körpers der Schleife Werte geladen oder berechnet werden die über alle Iterationen invariant bleiben. Für diese Code-Teile ist es äußerst unratsam sie jede Iteration neu auszuführen da sich dies unnötigerweise negativ auf die Performance auswirkt. In 2.6 sehen wir ein solches Beispiel. In jeder Iteration werden hier drei Werte geladen (`x, vec[0], vec.size()`). Die letzten beiden Werte bleiben über alle Iterationen gleich wenn wir davon ausgehen, dass sie nicht im weiteren Code der Schleife modifiziert werden. Wenn zur Kompilierzeit bestimmt werden kann, dass diese Werte Invarianten für die Schleife sind können sie aus dem Körper der Schleife

herausgezogen werden und durch Konstanten ersetzt werden. Dies führt in unserem Beispiel zu dem notwendigen Laden von einem anstatt von drei Werten pro Iteration und verbessert damit eindeutig die Performance der Schleife, da Ladeoperationen deutlich länger dauern als das Lesen eines konstanten Wertes vom Stack.

```
1 for(auto x : vec)
2 {
3     double a = (x - vec[0]) / vec.size();
4     ...
5 }
```

Figure 2.6: Schleife mit Invarianten im Schleifenkörper

2.3 Abschließend zu Optimierungen

In diesem Kapitel wurde der allgemeine Ablauf des Optimizers und einige ausgewählte Optimierungen besprochen. In einem modernen Optimizer werden hunderte von Optimierungen ausgeführt und ein kompletter Überblick ist im Rahmen dieses Berichts nicht möglich. Die ausgewählten Beispiele sollen dem Leser einen groben Überblick darüber verschaffen was innerhalb eines Compilers passiert und warum es sinnvoll ist sich mit seinem Compiler etwas genauer auseinander zusetzen wenn die Programm-Performance wichtig ist. Compiler bieten eine große Menge an möglichen Einstellungen um die Optimierungen zu beeinflussen und es kann einen starken Einfluss auf die Programm-Performance haben diese für sein Programm richtig zu setzen. Des weiteren kann es beim Optimieren von kritischen Stellen im Code sehr hilfreich sein schlechtere Performance haben als erwartet. Dies ist natürlich einer der letzten Schritte wenn es notwendig ist den Code so gut wie irgendwie möglich zu optimieren. Bei einer solchen Inspektion des vom Compiler erstellten Codes kann sich in manchen Fällen herausstellen, dass der Compiler den Code entweder garnicht optimieren konnte oder die Laufzeit vielleicht sogar negativ beeinflusst hat. Dies kann durchaus in manchen Fällen vorkommen, da der Compiler trotz seiner Komplexität und den äußerst mächtigen Optimizern nicht den gleichen Kontext für Code hat wie ein menschlicher Entwickler. Sollte so ein Fall eintreten ist es hilfreich wenn der Entwickler ein ungefähres Verständnis von Compilern und Optimizern hat um die Möglichkeit zu haben solche Probleme aufzudecken und zu beheben.

3 Ergebnisse

3.1 Ein Benchmark für Optimierungen

Nachdem wir in den letzten Kapiteln die Funktionsweise von Compilern besprochen haben gibt es nun in 3.1 einen Benchmark der den Performance Einfluss von Optimierungen auf ein Beispielprogramm aufzeigt. Bei dem Beispielprogramm handelt es sich um ein Programm zum lösen von partiellen Differentialgleichungen welches in C geschrieben ist und das Jacobi und das Gauß-Seidel Verfahren benutzen. Kompiliert wurde es sowohl mit gcc als auch mit clang und es wurden die Optimierungs-Level (-O0 bis -O3) getestet. Die Messungen 1-3 haben eine steigende Problemgröße und jede Messung wurde drei mal ausgeführt und ein Mittelwert gebildet.

Bei den Messungen mit dem `-O0` flag werden von dem Compilern fast keine Optimierungen durchgeführt. Der Laufzeit Unterschied zwischen Clang und Gcc ist hier sehr auffällig, sollte aber nicht in der Hinsicht interpretiert werden, dass ein Compiler besser als der andere ist. Ein Programm ohne Optimierungen zu kompilieren ist im generellen Fall nicht ratsam und da der Compiler explizit keine Optimierungen vornimmt kommt der in den Messungen entstandene Unterschied wahrscheinlich hauptsächlich aus der unterschiedlichen Struktur der Frontends von Gcc und Clang und was für eine Art von Intermediate Representation diese für den Optimizer erstellen. Wir sehen, dass sobald mit mindestens einem Optimierungslevel von 1 kompiliert wird sich die Laufzeit der Messungen nurnoch wenig unterscheiden. Der konkrete Vergleich beider Compiler mit den jeweiligen Optimierungsleveln ist auch nicht komplett aussagekräftig, da beide Compiler nicht unbedingt die selben Optimierungen pro Level hinzufügen. Die Hauptaussage die aus den Messungen gezogen werden kann ist der allgemeine Einfluss von Compiler Optimierungen auf die Performance von Programmen. In unserem Beispiel haben wir

Compiler/Flags	Messung 1	Messung 2	Messung 3
Clang -O0	27s	108s	239s
Gcc -O0	30s	121s	269s
Clang -O1	22s	87s	192s
Gcc -O1	23s	91s	200s
Clang -O2	22s	86s	190s
Gcc -O2	21s	83s	184s
Clang -O3	22s	87s	192s
Gcc -O3	21s	83s	184s

Figure 3.1: Benchmark für verschiedene Optimierungsstufen

Laufzeitverbesserungen von über 20 Prozent von unoptimiertem zu optimiertem Code. Compileroptimierungen können also einen starken Einfluss auf die Laufzeit eines Programms haben und sollten von Entwickler beachtet und korrekt eingesetzt werden wenn die Programm-Performance wichtig ist. Des weiteren ist natürlich zu beachten, dass die möglichen Compileroptimierungen stark von dem betrachteten Programm abhängen und die Laufzeitunterschiede hier stark schwanken können wenn andere Software betrachtet wird. Das Beispielprogramm das für die Messungen benutzt wurde ist nicht allzu komplex und verbringt den Großteil seiner Laufzeit damit eine Matrix zu durchlaufen und arithmetische Operationen auf dieser auszuführen. Die Ergebnisse für ein hoch komplexes Programm mit mehr kritischen Stellen und einer vielfach größeren Codebase können sich hier stark von unserem Beispiel unterscheiden.
Ein interessanter Fakt an den Messungen ist die kleine Verschlechterung der Laufzeit für Clang wenn man von `-O2` zu `-O3` hoch geht. Dies könnte daher stammen, dass mit `-O3` einige komplexere Optimierungen am Code ausprobiert wurden welche aber letztendlich für die getesteten Eingabewerte keinen positiven Effekt hatten. Dies zeigt nochmal, dass ein Entwickler sich Gedanken über die verwendeten Compileroptimierungen machen und für seine Programme verschiedene Kompilieroptionen testen sollte um den maximalen Performancegewinn herauszuholen.

3.2 Abschließend

In diesem Bericht wurden der Aufbau und die Arbeitsweise von Compilern und ihren Optimizern erläutert. Der Ablauf einer Kompilierung und deren verschiedene Phasen im Compiler wurden genauer erklärt um ein generelles Verständnis darüber zu verschaffen was konkret passiert wenn ein Entwickler seine Programme kompiliert.
Der Hauptpunkt des Berichts ist nicht das Wissen darüber wie man einen Compiler implementieren kann sondern soll dazu anregen sich bei der Softwareentwicklung etwas mehr mit seinem Compiler auseinander zusetzen um die best mögliche Performance für seine Programme zu bekommen. LLVM bietet sich hierfür besonders an, da man sich zu jedem Schritt des Compilers eine Menschen lesbare Form der Intermediate Representation ausgeben lassen kann um zu beobachten wie der Code eines Programms im Compiler transformiert wird.
Abschließend ist zu sagen, dass Compileroptimierungen immer benutzt werden sollten, da sie einen Performance Gewinn bieten für den der Entwickler kaum Zeit investieren muss. Moderne Compiler sind große Projekte von komplexer Software und werden ständig weiterentwickelt. Trotz dessen ist der Compiler nicht unfehlbar und man sollte im Hinterkopf behalten verschiedene Optimierungsflags für seine Programme zu testen wenn man das optimale Ergebnis haben möchte.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[Car15] Chandler Carruth. Understanding Compiler Optimization. 12 2015.

[Pro19a] LLVM Project. llvm.org, 1 2019.

[Pro19b] LLVM Project. LLVM's Analysis and Transform Passes, 1 2019.

[Reg18] John Regehr. How LLVM Optimizes a Function, 9 2018.

[SbJ+18] Jannek Squar, michael blesel, Tim Jammer, Michael Kuhn, and Thomas Ludwig. Cato - compiler assisted source-to-source transformation of openmp kernels to utilise distributed memory. 09 2018.