LLVM

Seminar Effiziente Programmierung

Benjamin Hosseini

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

4. Februar 2021

Gliederung

Was ist LLVM?

- Anfangs Akronym für Low Level Virtual Machine
- LLVM ist im Kern eine Compiler-Struktur
- Idee und erste Umsetzung von Chris Lattner
- In der Industrie weit verbreitet: Apple inc., NVIDIA, Intel etc. [1]
- Ziel: Schwächen klassischer Compiler-Strukturen zu beseitigen

Klassische Compiler-Struktur

- Front-End
  - Scanner
  - Lexer
  - Parser

Klassische Compiler-Struktur

- Front-End
  - Scanner
  - Lexer
  - Parser
- Back-End
  - Abgestimmt auf ein bestimmtes Zielsystem
  - Generiert passenden Native-Code
  - Kann auch Byte-Code erzeugen

Klassische Compiler-Struktur

- Front-End
  - Scanner
  - Lexer
  - Parser
- Back-End
  - Abgestimmt auf ein bestimmtes Zielsystem
  - Generiert passenden Native-Code
  - Kann auch Byte-Code erzeugen
- Optimierer, der unterschiedlich implementiert werden kann
  - Link-Time Optimierer
  - Run-Time Optimierer
  - Profile-Driven Optimierer

Link-Time Optimierung

- Link-Time ist die Zeit in der verschiedene Module verbunden werden
- Niedrige Ebene (Maschinencode)
  - Erlaubt freie Wahl von Front-End

- [9]

Link-Time Optimierung

- Link-Time ist die Zeit in der verschiedene Module verbunden werden
- Niedrige Ebene (Maschinencode)
  - Erlaubt freie Wahl von Front-End
  - Probleme
    - Verlust von Informationen auf Source-Code Ebene
    - Nur Optimierungen auf niedriger Ebene (Bspw. Inlining)

- [9]

Link-Time Optimierung

- Link-Time ist die Zeit in der verschiedene Module verbunden werden
- Niedrige Ebene (Maschinencode)
  - Erlaubt freie Wahl von Front-End
  - Probleme
    - Verlust von Informationen auf Source-Code Ebene
    - Nur Optimierungen auf niedriger Ebene (Bspw. Inlining)
- Höhere Ebene (IR/AST)
  - Kein Verlust von Source-Code-Level Informationen

- [9]

Link-Time Optimierung

- Link-Time ist die Zeit in der verschiedene Module verbunden werden
- Niedrige Ebene (Maschinencode)
  - Erlaubt freie Wahl von Front-End
  - Probleme
    - Verlust von Informationen auf Source-Code Ebene
    - Nur Optimierungen auf niedriger Ebene (Bspw. Inlining)
- Höhere Ebene (IR/AST)
  - Kein Verlust von Source-Code-Level Informationen
  - Probleme
    - Änderungen an einzelnen Dateien –> Komplette Re-Kompilation
    - Nicht-Standardisierte Darstellung –> fehlende Interoperabilität mit anderen Compiler
- [9]

Schwächen klassischer Compiler-Strukturen

- Eingeschränkt in Optimierungsmöglichkeiten
- Wenig bis keine Modularität
- Retargeting ist in den meisten Fällen schwierig/aufwendig

LLVM Compiler-Struktur

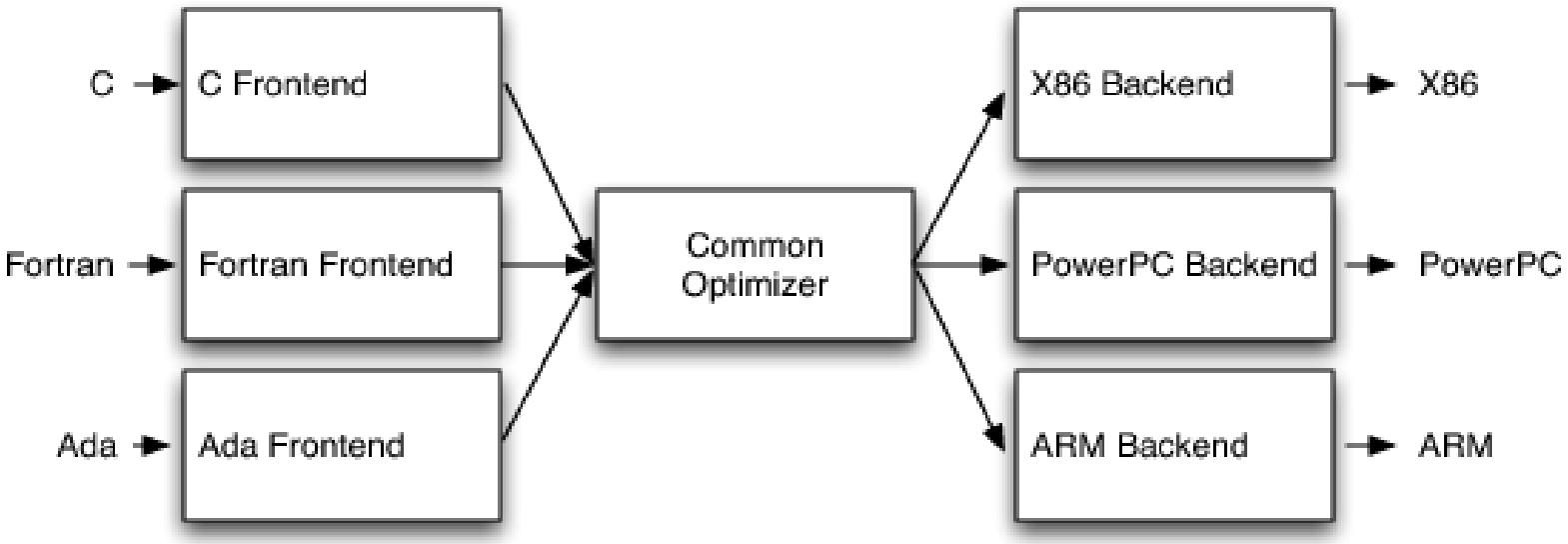

Abbildung: Aufteilung in Front-End, Optimierer und Back-end [4]

LLVM Compiler-Struktur

Abbildung: Aufteilung in Front-End, Middle-End und Back-end [6]

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)
- Module

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)
- Module
- Globale Werte

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)
- Module
- Globale Werte
- Globale Variablen
- Structures
- Funktionen

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)
- Module
- Globale Werte
- Globale Variablen
- Structures
- Funktionen
- Basic Blocks

LLVM Intermediate Representation

- Ist in C++ geschrieben
- Hat verschiedene Formate
  - vom Menschen lesbares Format (.ll)
  - Bit-Code-Format (.bc)
- Module
- Globale Werte
- Globale Variablen
- Structures
- Funktionen
- Basic Blocks
- Instruktionen

Virtuelle Befehlssatz

- Assemblernahe
- Bildet Informationen der Source-Code-Ebene ab (Bspw. Struct für eine Klasse)
- Strenge Typisierung
- Unendliche Anzahl an virtuellen Registern
- In SSA-Form

Type	Name
arithmetic	add, sub, mul, div, rem
bitwise	and, or, xor, shl, shr
comparison	seteq, setne, setlt, setgt, setle, setge
control-flow	ret, br, mbr, invoke, unwind
memory	load, store, getelementptr, alloca
other	cast, call, phi

Abbildung: Tabelle aller Instruktionen [7]

Additionsbeispiel (C++)

```
1  int multiAddition(int x, int y) {
2      int ret = 0;
3      for (int i = 0; i <= 10; i++) {
4          if (i % 2 == 0) {
5              x += y;
6          }
7      }
8      ret = x;
9      return ret;
10 }
```

Additionsbeispiel (LLVM IR)

```
1  define  i32 @multiAddition(i32 %x, i32 %y) #5 {
2  entry:
3    %y.addr = alloca i32, align 4
4    %x.addr = alloca i32, align 4
5    %ret = alloca i32, align 4
6    %i = alloca i32, align 4
7    store i32 %y, i32* %y.addr, align 4
8    [...]
9    br label %for.cond
10 for.cond:
11   %0 = load i32, i32* %i, align 4
12   %cmp = icmp sle i32 %0, 10
13   br i1 %cmp, label %for.body, label %for.end
14 for.body:
15   %1 = load i32, i32* %i, align 4
16   %rem = srem i32 %1, 2
17   %cmp1 = icmp eq i32 %rem, 0
18   br i1 %cmp1, label %if.then, label %if.end
```

Additionsbeispiel Fortsetzung (LLVM IR)

```
19 if.then:
20   [...]
21   %add = add nsw i32 %3, %2
22   store i32 %add, i32* %x.addr, align 4
23   br label %if.end
24   if.end:
25     br label %for.inc
26 for.inc:
27   %4 = load i32, i32* %i, align 4
28   %inc = add nsw i32 %4, 1
29   store i32 %inc, i32* %i, align 4
30   br label %for.cond, !llvm.loop !11
31 for.end:
32   %5 = load i32, i32* %x.addr, align 4
33   store i32 %5, i32* %ret, align 4
34   %6 = load i32, i32* %ret, align 4
35   ret i32 %6
36   }
```

Optimierung in LLVM

- Hauptsächlich zwischen Front-End und Back-End
- Passes als Werkzeuge
  - Analysis Passes
  - Transformation Passes
  - Utility Passes
- [1]

Analysis Passes

- Sammelt Informationen aus der IR
- Kann von anderen Passes benutzt werden

Analysis Passes

- Sammelt Informationen aus der IR
- Kann von anderen Passes benutzt werden
- Beispiel dot-cfg
  - Baut den Kontrollflussgraphen
  - hat .dot-Datei als Output (GraphViz)

Transformation Passes

- Passes die Änderungen an den LLVM IR-Dateien ausführen
- Kann Analysis Passes benutzen
- Transformierung muss validiert sein

Transformation Passes

- Passes die Änderungen an den LLVM IR-Dateien ausführen
- Kann Analysis Passes benutzen
- Transformierung muss validiert sein
- Beispiel instcombine
  - Simplifiziert algebraisch
  - Kombiniert Instruktionen

Utility Passes

- Passes die in keine der vorherigen Kategorien passen

Utility Passes

- Passes die in keine der vorherigen Kategorien passen
- Beispiele
  - view-cfg (Veranschaulichung)
  - instnamer (Unterstützung für den Entwickler)

Additionsbeispiel (C++) Wdh.

```
1  int multiAddition(int x, int y) {
2      int ret = 0;
3      for (int i = 0; i <= 10; i++) {
4          if (i % 2 == 0) {
5              x += y;
6          }
7      }
8      ret = x;
9      return ret;
10 }
```

Additionsbeispiel Wdh.(LLVM IR)

```
1  define  i32 @multiAddition(i32 %x, i32 %y) #5 {
2  entry:
3    %y.addr = alloca i32, align 4
4    %x.addr = alloca i32, align 4
5    %ret = alloca i32, align 4
6    %i = alloca i32, align 4
7    store i32 %y, i32* %y.addr, align 4
8    [...]
9    br label %for.cond
10 for.cond:
11   %0 = load i32, i32* %i, align 4
12   %cmp = icmp sle i32 %0, 10
13   br i1 %cmp, label %for.body, label %for.end
14 for.body:
15   %1 = load i32, i32* %i, align 4
16   %rem = srem i32 %1, 2
17   %cmp1 = icmp eq i32 %rem, 0
18   br i1 %cmp1, label %if.then, label %if.end
```

Additionsbeispiel Fortsetzung Wdh.(LLVM IR)

```
19 if.then:
20   [...]
21   %add = add nsw i32 %3, %2
22   store i32 %add, i32* %x.addr, align 4
23   br label %if.end
24   if.end:
25     br label %for.inc
26 for.inc:
27   %4 = load i32, i32* %i, align 4
28   %inc = add nsw i32 %4, 1
29   store i32 %inc, i32* %i, align 4
30   br label %for.cond, !llvm.loop !11
31 for.end:
32   %5 = load i32, i32* %x.addr, align 4
33   store i32 %5, i32* %ret, align 4
34   %6 = load i32, i32* %ret, align 4
35   ret i32 %6
36   }
```

Additionsbeispiel optimiert (LLVM IR)

```
1  define  i32 @multiAddition(i32 %x, i32 %y) #6 {
2  entry:
3    %reass.add = shl i32 %y, 1
4    %spec.select.2 = add i32 %reass.add, %x
5    %reass.add9 = shl i32 %y, 1
6    %spec.select.6 = add i32 %spec.select.2, %reass.add9
7    %reass.add10 = shl i32 %y, 1
8    %spec.select.10 = add i32 %spec.select.6,
         ↪ %reass.add10
9    ret i32 %spec.select.10
10 }
```

Zusammenfassung

- LLVM ist eine Compiler-Struktur
- Modularer und erweiterbarer als klassische Compiler-Strukturen

Zusammenfassung

- LLVM ist eine Compiler-Struktur
- Modularer und erweiterbarer als klassische Compiler-Strukturen
- LLVM IR als Schnittstelle einzelner Komponenten
- Assemblernaher Befehlssatz mit high-level Informationen

Zusammenfassung

- LLVM ist eine Compiler-Struktur
- Modularer und erweiterbarer als klassische Compiler-Strukturen
- LLVM IR als Schnittstelle einzelner Komponenten
- Assemblernaher Befehlssatz mit high-level Informationen
- Optimierung hauptsächlich mittels Passes (Analysis, Transformation und Utility)

Literatur I

[1] *LLVM Website*. `https://llvm.org/`

[2] *Static single assignment Form*. `https://en.wikipedia.org/wiki/Static_single_assignment_form`

[3] *Parsingintro Sourceforge*. `http://parsingintro.sourceforge.net/`. Version: Oct 2007

[4] *Kapitel* LLVM. In: *The Architect of Open Source Applications*. 2011

[5] *Introducing LLVM Intermediate Representation*. `https://hub.packtpub.com/introducing-llvm-intermediate-representation/`. Version: Aug 2014

Literatur II

[6] *LLVM IR and GO*. `https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/`. Version: Dec 2018

[7] Adve, Vikram ; Lattner, Chris ; Brukman, Michael ; Shukla, Anand ; Gaeke, Brian: LLVA: A Low-level Virtual Instruction Set Architecture. In: *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*. San Diego, California, Dec 2003

[8] Chris Lattner and Vikram Adve: The LLVM Instruction Set and Compilation Strategy / CS Dept., Univ. of Illinois at Urbana-Champaign. 2002 (UIUCDCS-R-2002-2292). – Tech. Report

Literatur III

[9] Lattner, Chris: *LLVM: An Infrastructure for Multi-Stage Optimization*. Urbana, IL, Computer Science Dept., University of Illinois at Urbana-Champaign, Diplomarbeit, Dec 2002

[10] Lattner, Chris ; Adve, Vikram: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004, S. 1–12