Compiler und Compiler-Optimierungen

Seminar - Effiziente Programmierung

Michael Blesel

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

2019-01-24

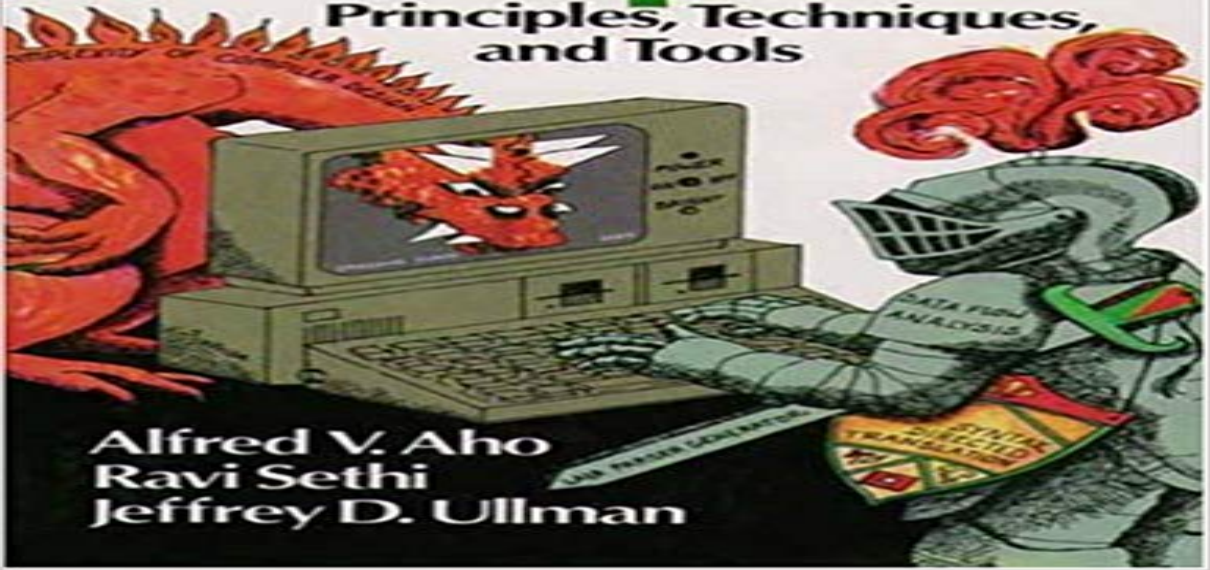
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik
die zukunft

Gliederung (Agenda)

Compilers

Principles, Techniques, and Tools



Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

Aufbau

- Der Compile Vorgang hat 3 Phasen
- Frontend ist Programmiersprachen-abhängig
- Backend ist Hardware-abhängig

Abbildung: Aufbau eines Compilers

LLVM

Abbildung: The LLVM Project

- LLVM ist ein C++ Open-Source Compiler Infrastruktur Projekt
- Bietet eine große Sammlung an Compiler Libraries
- Verwendet von z.B. Apple (Xcode) und Google

Design von LLVM

- Modulares Design (Pluspunkt von LLVM im Vergleich zu anderen Compilern)
- Sprachen unabhängiger Optimizer
- Low-level Intermediate Representation (IR) im gesamten Optimizer

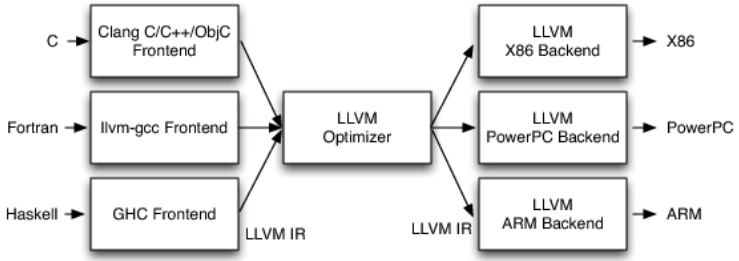


Abbildung: LLVM Infrastruktur

Schritte eines Kompilierungsvorganges

- Frontend: Source -> AST -> IR
- Optimizer: Optimierungspässe auf IR ausführen
- Backend: IR in target spezifischen machine-code

Ein simples C Programm

```
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 }
```

Abbildung: Das zu kompilierende Programm

AST

```
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
```

Abbildung: AST für das Programm

IR Code

```
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 }
```

Abbildung: IR Code für das Programm

Assembler

```
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
```

Aufgaben des Optimizers

- Optimizer führt ausgewählte Pässe aus um IR Code zu modifizieren
- Beispielpässe:
  - Dead-code elimination
  - Function inlining
  - Loop unrolling
  - Combine redundant instructions

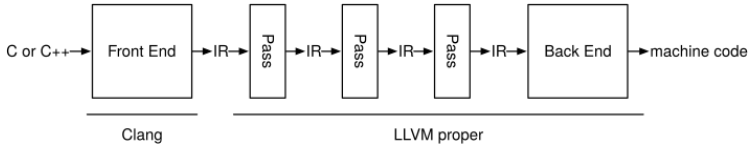


Abbildung: Optimizer workflow

Optimierungsphasen (Phase Ordering)

Schritte im Optimizer:

- Canonicalization
- IR Code in SSA (static single assignment) Form bringen
- Allgemeine Optimierungen (dead-code elimination, CFG vereinfachen,...)
- Aufwändigere Optimierungen (Schleifen-Optimierungen, inlining,...)

Canonicalization

- Frontend erstellt verbosen IR Code
- Optimizer formt IR in feste Muster um Optimierungen zu erleichtern

```
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 }              | }             |
```

Abbildung: Gleicher Effekt, verschiedener Code

```
1  ; Function Attrs: noinline nounwind uwtable
2  define dso_local i32 @multiply(i32 %a, i32 %b) #0 {
3  entry:
4    %a.addr = alloca i32, align 4
5    %b.addr = alloca i32, align 4
6    store i32 %a, i32* %a.addr, align 4
7    store i32 %b, i32* %b.addr, align 4
8    %0 = load i32, i32* %a.addr, align 4
9    %1 = load i32, i32* %b.addr, align 4
10   %mul = mul nsw i32 %0, %1
11   ret i32 %mul
12 }
13 ; Function Attrs: noinline nounwind uwtable
14 define dso_local i32 @main() #0 {
15 entry:
16   %retval = alloca i32, align 4
17   %a = alloca i32, align 4
18   %b = alloca i32, align 4
19   %c = alloca i32, align 4
20   store i32 0, i32* %retval, align 4
21   store i32 10, i32* %a, align 4
22   store i32 5, i32* %b, align 4
23   %0 = load i32, i32* %a, align 4
24   %1 = load i32, i32* %b, align 4
25   %call = call i32 @multiply(i32 %0, i32 %1)
26   store i32 %call, i32* %c, align 4
27   ret i32 0
28 }
```

Canonicalization

```
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 }
```

Abbildung: IR nach Canonicalization / SSA Form

Schleifen-Optimierungen

- In vielen komplexen Programmen wird der Großteil der Laufzeit in Schleifen verbracht.
- Viel Optimierungspotential
  - Unrolling
  - Herausziehen von Konstanten
  - Vektorisierung
- Keine einfache Aufgabe (Umstrukturierung darf Semantik nicht ändern)

Schleifen Canonicalization

```
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  }
```

Abbildung: Eine einfache Schleife

```
1  define dso_local i32 @main() #0 {
2  entry:
3    br label %for.body
4
5  for.body:                         ; preds = %entry, %for.body
6    %i.02 = phi i32 [ 0, %entry ], [ %inc, %for.body ]
7    %result.01 = phi i32 [ 0, %entry ], [ %add, %for.body ]
8    %add = add nsw i32 %result.01, %i.02
9    %inc = add nsw i32 %i.02, 1
10   %cmp = icmp slt i32 %inc, 10
11   br i1 %cmp, label %for.body, label %for.end
12
13 for.end:                          ; preds = %for.body
14   %result.0.lcssa = phi i32 [ %add, %for.body ]
15   ret i32 %result.0.lcssa
16 }
```

Abbildung: 'Normalisierter' Schleifen IR Code

Loop unrolling

- Oft kann die Iterationszahl zur Compile-Zeit fesgestellt werden.
- Jumps und das Überprüfen von Bedingungen kostet Zeit.
- 'Erkauft' Performance für Codegröße.
- Macht weitere Optimierungen möglich.

```
1 define dso_local i32 @main() #0 {
2 entry:
3   ret i32 45
4 }
```

Abbildung: Optimierungen nach unrolling

Loop interleaving

- 'Unrolled' Schleifen und sortiert Code um.
- Sammelt gleiche Arten von Anweisungen und gruppiert sie.
- Beispiel: Array Addition
- Kann hardware-abhängig Performance gewinnen (Pipelining).
- Mögliche Vorstufe für Vektorisierung.

Schleifencode minimieren

- Herausziehen von Invarianten aus dem Schleifenkörper.
- Kann oft die Anzahl an loads drastisch verringern.

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

Abbildung: vec[0] und vec.size() sind jede Iteration gleich.

Optimierungsstufen Benchmark

- **Programm:** Partdiff, ein pde-solver aus den HR Übungen
- **CPU:** Intel Xeon X5650

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

Abbildung: Benchmark für verschiedene Optimierungsstufen

Zusammenfassung

- Compiler sind keine Magie
- Compiler Optimierungen können großen Einfluss auf die Performance haben.
- Es ist hilfreich eine grobe Idee zu haben wie ein Compiler optimiert.

Fragen?

Literatur

[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.

LLVM Live Demo