Compiler Optimierungen

Seminar “Effiziente Programmierung in C”

Timo Bemmer

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

6. Juni 2021

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik die zukunft

Gliederung

1 Einleitung
2 Compiletime
3 Linktime
4 Runtime
5 Limitierungen
6 Zusammenfassung
7 Literatur

Was heißt "Optimierung"? [Muc98]

> Only very rarely does applying optimizations to a program result in object code whose performance is optimal, by any measure.
>
> — Steven Muchnick, *Advanced Compiler Design and Implementation*

Was heißt “Optimierung”?

Der *Versuch*, die Nutzung einer Ressource zu *verbessern*

- Laufzeit
- Speicherplatz
- Energie

Heuristiken [ALSU06, Muc98]

Was führt zu einer Optimierung?

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen

Heuristiken [ALSU06, Muc98]

Was führt zu einer Optimierung?

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen
- Nutze die Speicherhierarchie aus
  - Register > Cache > RAM

Heuristiken [ALSU06, Muc98]

Was führt zu einer Optimierung?

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen
- Nutze die Speicherhierarchie aus
  - Register > Cache > RAM
- Vermeide Sprünge
  - Funktionsaufrufe
  - Kontrollfluss (`if`, `for`, `while`)

Überblick [LA04, Lat02]

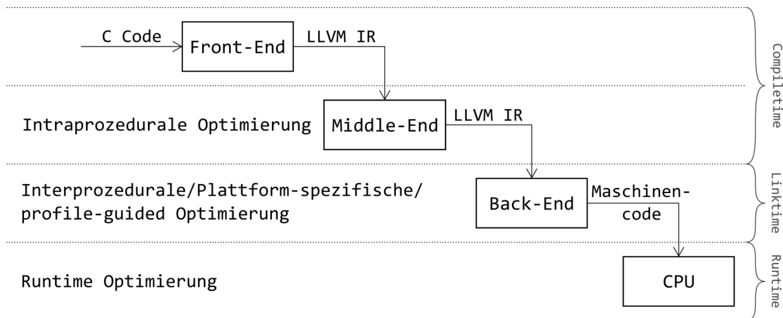


Abbildung: Ebenen der Optimierung

Intraprozedurale Analyse [ALSU06, Muc98]

Wir betrachten Funktionen als alleinstehende Einheiten

- Der lokale Daten-/Kontrollfluss kann analysiert werden
- *Strength reduction* kann angewendet werden
- Redundanter/toter Code kann entfernt werden

Konzept: *Strength reduction* [ALSU06, Muc98]

Idee: ersetze komplexe/schwere Berechnungen durch äquivalente, aber simplere. Z.B. durch

- algebraische Identitäten
  - `1 * x = x * 1 = x / 1 = x`
  - `0 * x = x * 0 = 0 / x = 0`
  - `0 + x = x + 0 = x - 0 = x`
- Induktionsvariablen

Induktionsvariablen [ALSU06, Muc98]

```
1 for (int i = 0; i < 9; i++) {
2     for (int j = 0; j < 9; j++) {
3       int idx = i * 9 + j;
4       a[idx] = (j + 1) * (i + 1);
5     }
6 }
```

Induktionsvariablen [ALSU06, Muc98]

```
1  for (int i = 0; i < 9; i++) {
2      int x = i * 9;
3      for (int j = 0; j < 9; j++) {
4        int idx = x + j;
5        a[idx] = (j + 1) * (i + 1);
6      }
7  }
```

Induktionsvariablen [ALSU06, Muc98]

```
1  int x = 0;
2  for (int i = 0; i < 9; i++) {
3      for (int j = 0; j < 9; j++) {
4         int idx = x + j;
5         a[idx] = (j + 1) * (i + 1);
6      }
7      x += 9;
8  }
```

Induktionsvariablen [ALSU06, Muc98]

```
1  int x = 0;
2  for (int i = 0; i < 9; i++) {
3      for (int j = 0; j < 9; j++) {
4        int idx = x + j;
5        a[idx] = (j + 1) * (i + 1);
6      }
7      x += 9;
8  }
```

Induktionsvariablen [ALSU06, Muc98]

```
1  int x = 0;
2  int y = 1;
3  for (int i = 0; i < 9; i++) {
4      for (int j = 0; j < 9; j++) {
5        int idx = x + j;
6        a[idx] = (j + 1) * y;
7      }
8      x += 9;
9      y += 1;
10 }
```

Induktionsvariablen [ALSU06, Muc98]

```
1  int x = 0;
2  int y = 1;
3  for (int i = 0; i < 9; i++) {
4      int z = y;
5      for (int j = 0; j < 9; j++) {
6        int idx = x + j;
7        a[idx] = z;
8        z += y;
9      }
10     x += 9;
11     y += 1;
12 }
```

Daten-/Kontrollfluss [ALSU06, Muc98]

Für den Compiler ist ein Programm ein Graph, dessen

- Knoten *Grundblöcke* (basic blocks) sind
- Kanten den Kontrollfluss zwischen Grundblöcken modellieren

Grundblöcke [ALSU06]

- Grundblockanfang
  - Erste Zeile
  - Ziel eines Sprungbefehls
  - Zeile nach einem Sprungbefehl

Grundblöcke [ALSU06]

- Grundblockanfang
  - Erste Zeile
  - Ziel eines Sprungbefehls
  - Zeile nach einem Sprungbefehl
- Grundblock
  - Grundblockanfang
  - Alle darauffolgenden Zeilen
  - Bis zum (ausschließlich) nächsten Grundblockanfang...
  - ...oder dem Programmende

Grundblöcke [ALSU06]

```
1  for (int i = 0; i < 9; i++) {
2      for (int j = 0; j < 9; j++) {
3        int idx = i * 9 + j;
4        a[idx] = (j + 1) * (i + 1);
5      }
6  }
```

Grundblöcke [ALSU06]

```
 1      int i = 0;
 2  L1:;
 3      int j = 0;
 4  L2:;
 5      int t1 = i * 9;
 6      int idx = t1 + j;
 7      int t2 = j + 1;
 8      int t3 = i + 1;
 9      int t4 = t2 * t3;
10      a[idx] = t4;
11      j = j + 1;
12      if (j < 9) goto L2;
13      i = i + 1;
14      if (i < 9) goto L1;
```

Grundblöcke [ALSU06]

```
1      int i = 0;
2  L1:;
3      int j = 0;
4  L2:;
5      int t1 = i * 9;
6      int idx = t1 + j;
7      int t2 = j + 1;
8      int t3 = i + 1;
9      int t4 = t2 * t3;
10     a[idx] = t4;
11     j = j + 1;
12     if (j < 9) goto L2;
13     i = i + 1;
14     if (i < 9) goto L1;
```

Grundblöcke [ALSU06]

```
1      int i = 0;
2  L1:;
3      int j = 0;
4  L2:;
5      int t1 = i * 9;
6      int idx = t1 + j;
7      int t2 = j + 1;
8      int t3 = i + 1;
9      int t4 = t2 * t3;
10     a[idx] = t4;
11     j = j + 1;
12     if (j < 9) goto L2;
13     i = i + 1;
14     if (i < 9) goto L1;
```

Grundblöcke [ALSU06]

```
1      int i = 0;
2  L1:;
3      int j = 0;
4  L2:;
5      int t1 = i * 9;
6      int idx = t1 + j;
7      int t2 = j + 1;
8      int t3 = i + 1;
9      int t4 = t2 * t3;
10     a[idx] = t4;
11     j = j + 1;
12     if (j < 9) goto L2;
13     i = i + 1;
14     if (i < 9) goto L1;
```

Grundblöcke [ALSU06]

B1:
```
int i = 0;
```

B2:
```
int j = 0;
```

B3:
```
int t1 = i * 9;
int idx = t1 + j;
int t2 = j + 1;
int t3 = i + 1;
int t4 = t2 * t3;
a[idx] = t4;
j = j + 1;
if (j < 9) goto B3;
```

B4:
```
i = i + 1;
if (i < 9) goto B2;
```

Abbildung: Grundblöcke

Kontrollflussgraph [ALSU06]

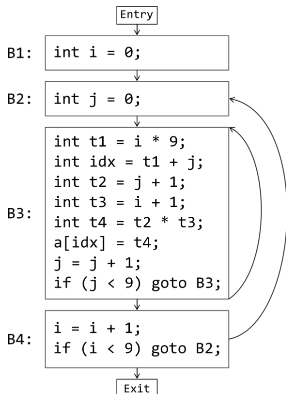
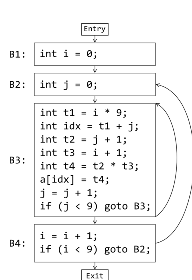
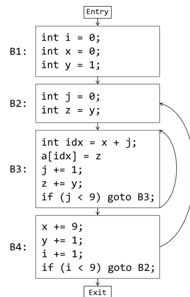


Abbildung: Kontrollflussgraph

Kontrollflussgraph [ALSU06]



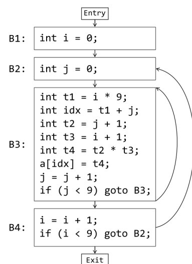
(a)



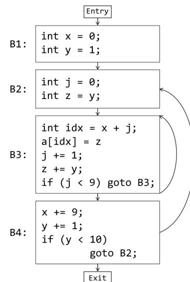
(b)

Abbildung: KFG vor (4a) und nach (4b) *strength reduction*

Kontrollflussgraph [ALSU06]



(a)



(b)

Abbildung: KFG vor (5a) und nach (5b) *strength reduction*

Constant propagation [ALSU06, WZ91]

```
1  int foo() {
2      int a = 7;
3      int b = (a - 1);
4      int c = (a + 1);
5      int d = a * b * c;
6      return d;
7  }
```

Constant propagation [ALSU06, WZ91]

```
1  int foo() {
2      int b = (7 - 1);
3      int c = (7 + 1);
4      int d = 7 * b * c;
5      return d;
6  }
```

Constant propagation [ALSU06, WZ91]

```
1  int foo() {
2      int d = 7 * 6 * 8;
3      return d;
4  }
```

Constant propagation [ALSU06, WZ91]

```
1  int foo() {
2      int d = 336;
3      return d;
4  }
```

Dead code elimination [ALSU06]

```
1  int bar() {
2      int a = 10;
3      int b = 7;
4      return a;
5  }
```

Dead code elimination [ALSU06]

```
1  int bar() {
2      int a = 10;
3      return a;
4  }
```

Common subexpression elimination [ALSU06]

```
1  int baz(int a) {
2      int b = (a * a) * 6;
3      int c = (a * a) / 2 * 3;
4      return b + c;
5  }
```

Common subexpression elimination [ALSU06]

```
1  int baz(int a) {
2      int t1 = (a * a);
3      int b = t1 * 6;
4      int c = t1 / 2 * 3;
5      return b + c;
6  }
```

Interprozedurale Analyse [ALSU06, Lat02]

Zur Link-Zeit steht uns ein Großteil des Programms zur Verfügung

- Der *globale* Daten-/Kontrollfluss kann analysiert werden
- Zeigeranalyse kann durchgeführt werden
- Das Laufzeitverhalten kann untersucht werden

Zeigeranalyse [ALSU06]

```
int noalias(int a, int b) {
    b = 0;
    return a;
}

int main() {
    int x = 4;
    int y = 2;
    printf("%d", noalias(x, y)); // 4
}
```

Zeigeranalyse [ALSU06]

```
1  int alias(int* a, int* b) {
2      *b = 0;
3      return *a;
4  }
5
6  int main() {
7      int x = 4;
8      int y = 2;
9      printf("%d", noalias(&x, &y)); // 4
10 }
```

Zeigeranalyse [ALSU06]

```
1  int alias(int* a, int* b) {
2      *b = 0;
3      return *a;
4  }
5
6  int main() {
7      int x = 4;
8      int* y = &x;
9      printf("%d", alias(&x, y)); // 0
10 }
```

Profile-guided Optimierung [Lat02, LA04]

> Premature optimization is the root of all evil.
>
> Donald Knuth
> *Structured Programming with go to Statements*

Profile-guided Optimierung

- Instrumentiere das Programm
- Führe es mit Beispieldaten aus
- Identifiziere anhand der Laufzeitanalyse oft aufgerufenen Code/Flaschenhälse
- Optimiere diese Codestellen

Der LLVM runtime reoptimizer [Lat02]

- Sammle Laufzeitdaten während der Programmausführung
- Führe darauf basierend Optimierungen aus
  - Direkt auf Maschinencode
  - Auf LLVM IR, die erneut compiliert werden muss

Grenzen von Compiler Optimierungen [ALSU06, Muc98]

- Optimierungsprobleme sind i.A.
  - NP-vollständig oder sogar NP-schwer
  - unentscheidbar

Grenzen von Compiler Optimierungen [ALSU06, Muc98]

- Optimierungsprobleme sind i.A.
  - NP-vollständig oder sogar NP-schwer
  - unentscheidbar
- Compiler müssen
  - Semantik bewahren
  - sich Heuristiken bedienen
  - konservativ sein

Zusammenfassung

- Compiler Optimierungen sind selten optimal
- Sie können Performanz merklich verbessern
- Optimierungsprobleme sind schwer
- Compiler brauchen gute Daten und Heuristiken, um sie zu lösen

Zusammenfassung

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen

Zusammenfassung

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen
- Nutze die Speicherhierarchie aus
  - Register > Cache > RAM

Zusammenfassung

- Vermeide unnötige Arbeit
  - Redundanz
  - Toter code
  - Komplexe Berechnungen
- Nutze die Speicherhierarchie aus
  - Register $>$ Cache $>$ RAM
- Vermeide Sprünge
  - Funktionsaufrufe
  - Kontrollfluss (`if`, `for`, `while`)

Zusammenfassung

- Im “Middle-End” des Compilers finden intraprozedurale Optimierungen statt
- Im “Back-End” finden
  - interprozedurale
  - plattform-spezifische
  - profile-guided Optimierungen statt

Literatur I

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

[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.

Literatur II

[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* https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf.

[Muc98] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.