Optimierung und Datenflussanalyse

BC George (FH Bielefeld)

Motivation

Was geschieht hier?

```
01  {
02    var a;
03    var b = 2;
04    b = a;
05  }
```

Zwischencode ist eine gute Idee

Eine Sprache, viele Maschinen vs. viele Sprachen, eine Maschine

. . . und beides zusammen?

Zwischencode (intermediate code); hier: Drei-Adress-Code

- registerbasiert
- Formen: `x = y op z, x = op z, x = y`
- temporäre Variablen für Zwischenergebnisse
- bedingte und unbedingte Sprünge
- Pointerarithmetik für Indizierung

```
i = 0
while(f[i] > 100)
    i = i + 1;
```

```
    i = 0
L1: t1 = i * 8
    t2 = f + t1
    if t2 <= 100 goto L2
    t3 = i + 1
    i = t3
    goto L1
L2: ...
```

Optimierungen

Was ist Optimierung in Compilern?

Verändern von Quellcode, Zwischencode oder Maschinencode eines Programms mit dem Ziel,

- Laufzeit,
- Speicherplatz oder
- Energieverbrauch

zu verbessern.

Was ist machbar?

Manche Optimierungen machen den Code nur in bestimmten Fällen schneller, kleiner oder stromsparender.

Den optimalen Code zu finden, ist oft NP-vollständig oder sogar unentscheidbar.

- Heuristiken kommen zum Einsatz.
- Der Code wird verbessert, nicht in jedem Fall optimiert, manchmal auch verschlechtert.
- Der Einsatz eines Debuggers ist meist nicht mehr möglich.

Anforderungen an Optimierung

- sichere Transformationen durchführen
- möglichst keine nachteiligen Effekte erzeugen

Optimierung zur Übersetzungszeit vs. Optimierung zur Laufzeit

- Just-in-time-Compilierung (JIT), z. B. Java:

  Fast alle Optimierungsmaßnahmen finden in der virtuellen Maschine zur Laufzeit statt.

- Ahead-of-time-Compilierung (AOT), z. B. C:

  Der Compiler erzeugt Maschinencode, die Optimierung findet zur Übersetzungszeit statt.

Beide haben ihre eigenen Optimierungsmöglichkeiten, es gibt aber auch Methoden, die bei beiden einsetzbar sind.

Welcher Code wird optimiert?

- Algebraische Optimierung: Transformationen des Quellcodes
- Maschinenunabhängige Optimierung: Transformationen des Zwischencodes
- Maschinenabhängige Optimierung: Transformationen des Assemblercodes oder Maschinencodes

Viele Transformationen sind auf mehr als einer Ebene möglich. Wir wenden hier die meisten auf den Zwischencode an.

Welche Arten von Transformationen sind möglich?

- Eliminierung unnötiger Berechnungen
- Ersetzung von teuren Operationen durch kostengünstigere

Basisblöcke und Flussgraphen

Def.: Ein *Basisblock* ist eine Sequenz maximaler Länge von Anweisungen, die immer hintereinander ausgeführt werden.

Ein Sprungbefehl kann nur der letzte Befehl eines Basisblocks sein.

Def.: Ein *(Kontroll)Flussgraph* $G = (V, E)$ ist ein Graph mit

$V = \{B_i \mid B_i \text{ ist ein Basisblock des zu compilierenden Programms}\}$,

$E = \{(B_i, B_j) \mid \text{es gibt einen Programmlauf, in dem } B_j \text{ direkt hinter } B_i \text{ ausgeführt wird}\}$

Häufig benutzte Strategie: Peephole-Optimierung

Ein Fenster mit wenigen Zeilen Inhalt gleitet über den Quellcode, Zwischencode oder den Maschinencode. Der jeweils sichtbare Code wird mit Hilfe verschiedener Verfahren optimiert, wenn möglich.

Peephole-Optimierung ist zunächst ein lokales Verfahren, kann aber auch auf den gesamten Kontrollflussgraphen erweitert werden.

$\Rightarrow$

Anwendung von Graphalgorithmen!

Algebraische Optimierung

Ersetzen von Teilbäumen im AST durch andere Bäume

```
x = x*2          =>      x << 1

x = x + 0        // k.w.
x = x * 1        // k.w.

x = x*0          =>      x = 0
x = x*8          =>      x = x << 3
```

Sei $s = 2^a + 2^b$ die Summe zweier Zweierpotenzen:

```
x = n*s          =>      (n << a) + (n << b)
```

Diese Umformungen können zusätzlich mittels Peephole-Optimierung in späteren Optimierungsphasen durchgeführt werden.

Maschinenunabhängige Optimierung

Maschinenunabhängige Optimierung

- lokal (= innerhalb eines Basisblocks), z. B. Peephole-Optimierung

  Einige Strategien sind auch global einsetzbar (ohne die sog. Datenflussanalyse s. u.)

- global, braucht nicht-lokale Informationen
  - meist unter Zuhilfenahme der Datenflussanalyse
  - Schleifenoptimierung

Lokale Optimierung

Constant Folding und Common Subexpression elimination

- "*Constant Folding*": Auswerten von Konstanten zur Compile-Zeit

```
x = 6 * 7          =>      x = 42
if 2 > 0 jump L    =>      jump L
```

- "*Common Subexpression Elimination*"

```
x = y + z
...
a = y + z
```

ersetze mit (falls in `...` keine weiteren Zuweisungen an `x`, `y`, `z` erfolgen)

```
x = y + z
...
a = x
```

Elimination redundanter Berechnungen in einem Basisblock mitels DAGs

Hier werden sog. *DAG*s benötigt:

Ein DAG *directed acyclic graph* ist ein gerichteter, kreisfreier Graph.

DAGs werden für Berechnungen in Basisblöcken generiert, um gemeinsame Teilausdrücke zu erkennen.

Bsp.: a = (b + c) * (b + c) / 2

Copy propagation

- "*Copy Propagation*"

```
x = y + z
a = x
b = 2*a
```

ersetze mit

```
x = y + z
a = x
b = 2*x
```

Wenn auf *a* vor seiner nächsten Zuweisung nicht mehr lesend zugegriffen wird, kann *a* hier entfallen.

Globale Optimierung

Control Flow und Dead Code

- Kontrollfluss-Optimierungen

```
    if debug == 1 goto L1          if debug != 1 goto L2
    goto L2                        print debug info
L1: print debug info           L2: ...
L2: ...
```

- Elimination of unreachable code

```
     goto L1                       L1: a = b+c
     ...
L1: a = b+c
```

Schleifenoptimierung

Loop unrolling:

```
for i = 1 to 3
    print(i)
```

```
print("1")
print("2")
print("3")
```

Code Hoisting:

- Invarianten vor die Schleife schieben

```
   x = 0
L: a = n*7
   x = x + a
   if x<42 jump L
```

```
   x = 0
   a = n*7
L: x = x + a
   if x<42 jump L
```

Kombination zweier Verfahren

- Loop Unrolling (für eine Iteration), danach Common Subexpression Elimination

```
while (cond) {
    body
}
```

```
if (cond) {
    body
    while (cond) {
        body
    }
}
```

Datenflussanalyse

Die Datenflussanalyse (auf 3-Adress-Code) basiert auf dem Wissen der Verfügbarkeit von Variablen und Ausdrücken am Anfang oder Ende von Basisblöcken, und zwar für alle möglichen Programmläufe.

Man unterscheidet:

- Vorwärtsanalyse (in Richtung der Nachfolger eines Basisblocks)
- Rückwärtsanalyse (in Richtung der Vorgänger eines Basisblocks)

In beiden Fällen gibt es zwei Varianten:

- any analysis: Es wird die Vereinigung von Informationen benachbarter Block berücksichtigt.
- all analysis: Es wird die Schnittmenge von Informationen benachbarter Block berücksichtigt.

Forward-any-analysis

Diese Analyse wird zur Propagation von Konstanten und Variablen benutzt und bildet sukzessive Mengen von Zeilen mit Variablendefinitionen.

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i)$$

$out(B_i)$: alle Zeilennummern von Variablendefinitionen, die am Ende von B_i gültig sind

$in(B_i)$: alle Zeilennummern von Variablendefinitionen, die am Ende von Vorgängerblöcken von B_i gültig sind

$gen(B_i)$: alle Zeilennummern von letzten Variablendefinitionen in B_i

$kill(B_i)$: alle Zeilennummern von Variablendefinitionen außerhalb von B_i, die in B_i überschrieben werden

Zunächst ist $in(B_1) = \emptyset$, danach ist $in(B_i) = \bigcup out(B_j)$ mit B_j ist Vorgänger von B_i.

Forward-all-analysis

Diese Analyse wird zur Berechnung verfügbarer Ausdrücke der Form $x = y \ op \ z$ für die Eliminierung redundanter Berechnungen benutzt und bildet sukzessive Mengen von Ausdrücken.

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i)$$

$out(B_i)$: alle am Ende von B_i verfügbaren Ausdrücke

$in(B_i)$: alle Ausdrücke, die am Anfang von B_i verfügbar sind

$gen(B_i)$: alle in B_i berechneten Ausdrücke

$kill(B_i)$: alle Ausdrücke $x \ op \ y$ mit einer Definition von x oder y in B_i und $x \ op \ y$ ist nicht in B_i

Zunächst ist $gen(B_1) = \emptyset$, danach ist $in(B_i) = \bigcap out(B_j)$ mit B_j ist Vorgänger von B_i.

Backward-any-analysis

Diese Analyse dient der Ermittlung von lebenden und toten Variablen (für die Registerzuweisung) und bildet sukzessive Mengen von Variablen.

$$in(B_i) = gen(B_i) \cup (out(B_i) - kill(B_i)$$

$out(B_i)$: alle Variablen, die am Ende von B_i lebendig sind

$in(B_i)$: alle Variablen, die am Ende von Vorgängerblöcken von B_i lebendig sind

$gen(B_i)$: alle Variablen, deren erstes Vorkommen auf der echten Seite einer Zuweisung steht

$kill(B_i)$: alle Variablen, denen in B_i Werte zugewiesen werden.

Zunächst ist $out(B_n) = \emptyset$, danach ist $out(B_i) = \bigcup in(B_j)$ mit B_j ist Nachfolger von B_i.

Backward-all-analysis

Diese Analyse wird zur Berechnung von "very busy" Ausdrücken der Form $x = y\ op\ z$, die auf allen möglichen Wegen im Flussgraphen vom aktuellen Basisblock aus mindestens einmal benutzt werden. Ausdrücke sollten dort berechnet werden, wo sie very busy sind, um den Code kürzer zu machen.

$$in(B_i) = gen(B_i) \cup (out(B_i) - kill(B_i)$$

$out(B_i)$: alle Ausdrücke $x\ op\ y$, die am Ende von B_i very busy sind

$in(B_i)$: alle Ausdrücke, die am Anfang von B_i very busy sind

$gen(B_i)$: alle in B_i benutzen Ausdrücke

$kill(B_i)$: alle Ausdrücke $x\ op\ y$, deren Operanden in B_i nicht redefiniert werden.

Zunächst ist $out(B_n) = \emptyset$, danach ist $out(B_i) = \bigcap in(B_j)$ mit B_j ist Nachfolger von B_i.

Maschinenabhängige Optimierung

Elimination redundanter Lade-, Speicher- und Sprungoperationen

```
LD a, R0
ST R0, a        // k.w.
```

```
    goto L1          goto L2
    ...              ...
L1: goto L2      L1: goto L2
```

Register Allocation: Liveness Analysis

```
a = b + c
d = a + b
e = d - 1
```

a, d, e können auf **ein** Register abgebildet werden!

```
r1 = r2 + r3
r1 = r1 + r2
r1 = r1 - 1
```

⇒ a und e sind nach Gebrauch "*tot*"

Berechnung der minimal benötigten Anzahl von Registern

$\Rightarrow$ Liveness-Graph, Färbungsproblem für Graphen!

Es wird ein Graph $G = (V, E)$ erzeugt mit

$V = \{v \mid v \text{ ist eine benötigte Variable}\}$ und $E = \{(v_1, v_2) \mid v_1 \text{ und } v_2 \text{ sind zur selben Zeit "lebendig"}\}$

Heuristisch wird jetzt die minimale Anzahl von Farben für Knoten bestimmt, bei der benachbarte Knoten nicht dieselbe Farbe bekommen.

$\Rightarrow$ Das Ergebnis ist die Zahl der benötigten Register.

Und wenn man nicht so viele Register zur Verfügung hat?

Registerinhalte temporär in den Speicher auslagern ("*Spilling*").

Kandidaten dafür werden mit Heuristiken gefunden, z. B. Register mit vielen Konflikten (= Kanten) oder Register mit selten genutzten Variablen.

In Schleifen genutzte Variablen werden eher nicht ausgelagert.

Optimierung zur Reduzierung des Energieverbrauchs

Energieverbrauch verschiedener Maschinenbefehle

Maschinenoperationen, die nur auf Registern arbeiten, verbrauchen die wenigste Energie.

Operationen, die nur lesend auf Speicherzellen zugreifen, verbrauchen ca. ein Drittel mehr Energie.

Operationen, die Speicherzellen beschreiben, benötigen zwei Drittel mehr Energie als die Operationen ausschließlich auf Register.

Energieeinsparung durch laufzeitbezogene Optimierung

Kürzere Programmlaufzeiten führen in der Regel auch zu Energieeinsparungen.

gcc -O1 spart 2% bis 70% (durchschnittlich 20%) Energie

Umgekehrt: Energiebezogene Optimierung führt in der Regel zu kürzeren Laufzeiten.

Prozessorspannung variieren

Viele Prozessoren ermöglichen es, die Betriebsspannung per Maschinenbefehl zu verändern.

Eine höhere Spannung bewirkt eine proportionale Steigerung der Prozessorgeschwindigkeit und des fließenden Stroms, aber einen quadratischen Anstieg des Energieverbrauchs. ($P = U \times I, U = R \times I$)

Folgendes kann man ausnutzen:

Die Verringerung der Spannung um 20% führt zu einer um 20% geringeren Prozessorgeschwindigkeit, d. h. das Programm braucht 25% mehr Zeit, verbraucht aber 36% $(1 - (1 - 0,2)^2)$ weniger Energie.

$\Rightarrow$ Wenn das Programm durch Optimierung um 25% schneller wird und die Prozessorspannung um 20% verringert wird, verändert sich die Laufzeit des Programms nicht, man spart aber 36% Energie.

Wrap-Up

Wrap-Up

- Verschiedene Optimierungsverfahren auf verschiedenen Ebenen, Peephole
- Datenflussanalyse
- Senkung des Energieverbrauchs durch Optimierung

LICENSE