Helmut Seidl

# Programmoptimierung

*TU München*

Wintersemester 2003/04

# Organisatorisches

**Termine:**

| | |
|---|---|
| Vorlesung: | Montag, 13-15 |
| | Donnerstag, 10-12 |
| Übung: | Freitag, 10-12 |
| | Alex Berlea: `berlea@in.tum.de` |
| Materialien: | Folien, Aufzeichnung :-) |
| | Literatur :-)) |
| | Vorlesungs-Mitschrift (in Überarbeitung) |

**Schein:**

- 50% der Aufgaben
- zweimal vorrechnen :-)

# Geplanter Inhalt:

1. Vermeidung überflüssiger Berechnungen

   $\longrightarrow$ verfügbare Ausdrücke

   $\longrightarrow$ Konstantenpropagation/Array-Bound-Checks

   $\longrightarrow$ Code Motion

2. Ersetzen teurer Berechnungen durch billige

   $\longrightarrow$ Peep Hole Optimierung

   $\longrightarrow$ Inlining

   $\longrightarrow$ Reduction of Strength

   ...

3. Anpassung an Hardware

- $\longrightarrow$ Instruktions-Selektion
- $\longrightarrow$ Registerverteilung
- $\longrightarrow$ Scheduling
- $\longrightarrow$ Speicherverwaltung

# 0 Einführung

Beobachtung 1: Intuitive Programme sind oft ineffizient.

Beispiel:

```
void swap (int i, int j) {
     int t;
     if (a[i] > a[j]) {
          t = a[j];
          a[j] = a[i];
          a[i] = t;
         }
     }
```

## Ineffizienzen:

- Adressen `a[i]`, `a[j]` werden je dreimal berechnet :-(
- Werte `a[i]`, `a[j]` werden zweimal geladen :-(

## Verbesserung:

- Gehe mit Pointer durch das Feld `a`;
- speichere die Werte von `a[i]`, `a[j]` zwischen!

```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
         t = aj;
         *q = ai;
         *p = t;    // t kann auch noch
        }           // eingespart werden!
     }
```

## Beobachtung 2:

Höhere Programmiersprachen (sogar C :-) abstrahieren von Hardware und Effizienz.

Aufgabe des Compilers ist es, den natürlich erzeugten Code an die Hardware anzupassen.

## Beispiele:

- ... Füllen von Delay-Slots;
- ... Einsatz von Spezialinstruktionen;
- ... Umorganisation der Speicherzugriffe für besseres Cache-Verhalten;
- ... Beseitigung (unnötiger) Tests auf Overflow/Range.

## Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

## Beispiel:

```
y = f() + f();
```

$\Longrightarrow$

```
y = 2 * f();
```

**Idee:** Spare zweite Auswertung von `f()` ...

## Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

## Beispiel:

```
y = f() + f();
```

$\Longrightarrow$

```
y = 2 * f();
```

**Idee:** Spare zweite Auswertung von `f()` ???

**Problem:** Die zweite Auswertung könnte ein anderes Ergebnis liefern als die erste (z.B. wenn `f()` aus der Eingabe liest :-)

## Folgerungen:

$\Longrightarrow$ Optimierungen haben Voraussetzungen.

$\Longrightarrow$ Die Voraussetzungen muss man:

- formalisieren,
- überprüfen :-)

$\Longrightarrow$ Man muss beweisen, dass die Optimierung korrekt ist, d.h. die Semantik erhält !!!

## Beobachtung 4:

Optimierungs-Techniken hängen von der Programmiersprache ab:

→ welche Ineffizienzen auftreten;

→ wie gut sich Programme analysieren lassen;

→ wie schwierig / unmöglich es ist, Korrektheit zu beweisen ...

Beispiel: Java

## Unvermeidbare Ineffizienzen:

* Array-Bound Checks;
* dynamische Methoden-Auswahl;
* bombastische Objekt-Organisation ...

## Analysierbarkeit:

+ keine Pointer-Arithmetik;
+ keine Pointer in den Stack;
− dynamisches Klassenladen;
− Reflection, Exceptions, Threads, ...

## Korrektheitsbeweise:

+ mehr oder weniger definierte Semantik;
− Features, Features, Features;
− Bibliotheken mit wechselndem Verhalten ...

# ... in der Vorlesung:

eine einfache imperative Sprache mit:

- Variablen // Register
- $R = e;$ // Zuweisungen
- $R_1 = M[R_2];$ // Laden
- $M[R_1] = R_2;$ // Speichern
- if $(e)$ $s_1$ else $s_2$ // bedingte Verzweigung
- goto $L$ // keine Schleifen :-)

## Beachte:

- Vorerst verzichten wir auf Prozeduren :-)
- Externe Funktionen berücksichtigen wir, indem wir als Ausdruck $e$ auch $f(R_1, \ldots, R_k)$ gestatten für eine unbekannte Funktion $f$.

  $\Longrightarrow$ intra-prozedural

  $\Longrightarrow$ eine Art Zwischensprache, in die man (fast) alles übersetzen kann.

## Beispiel: `swap()`

$$
\begin{array}{llll}
0: & A_1 = A_0 + 1 * i; & // & A_0 == \&a \\
1: & R_1 = M[A_1]; & // & R_1 == a[i] \\
2: & A_2 = A_0 + 1 * j; & & \\
3: & R_2 = M[A_2]; & // & R_2 == a[j] \\
4: & \text{if } (R_1 > R_2) \ \{ & & \\
5: & \quad A_3 = A_0 + 1 * j; & & \\
6: & \quad t = M[A_3]; & & \\
7: & \quad A_4 = A_0 + 1 * j; & & \\
8: & \quad A_5 = A_0 + 1 * i; & & \\
9: & \quad R_3 = M[A_5]; & & \\
10: & \quad M[A_4] = R_3; & & \\
11: & \quad A_6 = A_0 + 1 * i; & & \\
12: & \quad M[A_6] = t; & & \\
 & \quad \} & &
\end{array}
$$

**Optimierung 1:** $1 * R \Longrightarrow R$

**Optimierung 2:** Wiederbenutzung von Teilausdrücken

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Damit erhalten wir:

$$
\begin{array}{lcl}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\text{if } (R_1 > R_2) \ \{ & & \\
\qquad t & = & R_2; \\
\qquad M[A_2] & = & R_1; \\
\qquad M[A_1] & = & t; \\
\qquad \} & &
\end{array}
$$

## Optimierung 3: Verkürzung von Zuweisungsketten :-)

## Ersparnis:

| | vorher | nachher |
|---|---|---|
| $+$ | 6 | 2 |
| $*$ | 6 | 0 |
| load | 4 | 2 |
| store | 2 | 2 |
| $>$ | 1 | 1 |
| $=$ | 6 | 2 |

# 1 Vermeidung überflüssiger Berechnungen

## 1.1 Mehrfach-Berechnungen

Idee:

Wird der gleiche Wert mehrfach berechnet, dann

$\rightarrow$ speichere ihn nach der ersten Berechnung;

$\rightarrow$ ersetze jede weitere Berechnung durch Nachschlagen!

$\Longrightarrow$ Verfügbarkeit von Ausdrücken

$\Longrightarrow$ Memoisierung

**Problem:** Erkenne Mehrfach-Berechnungen!

**Beispiel:**

$$
\begin{array}{lrcl}
 & z & = & 1; \\
 & y & = & \mathsf{read}(); \\
A: & x_1 & = & \boxed{y+z}; \\
 & & \dots & \\
B: & x_2 & = & \boxed{y+z};
\end{array}
$$

## Achtung:

$B$ ist eine Mehrfach-Berechnung des Werts von $\boxed{y + z}$, falls:

(1) $A$ stets vor $B$ ausgeführt wird; und

(2) $y$ und $z$ an $B$ die gleichen Werte haben wie an $A$ :-)

$\Longrightarrow$ Wir benötigen

$\rightarrow$ eine operationelle Semantik :-)

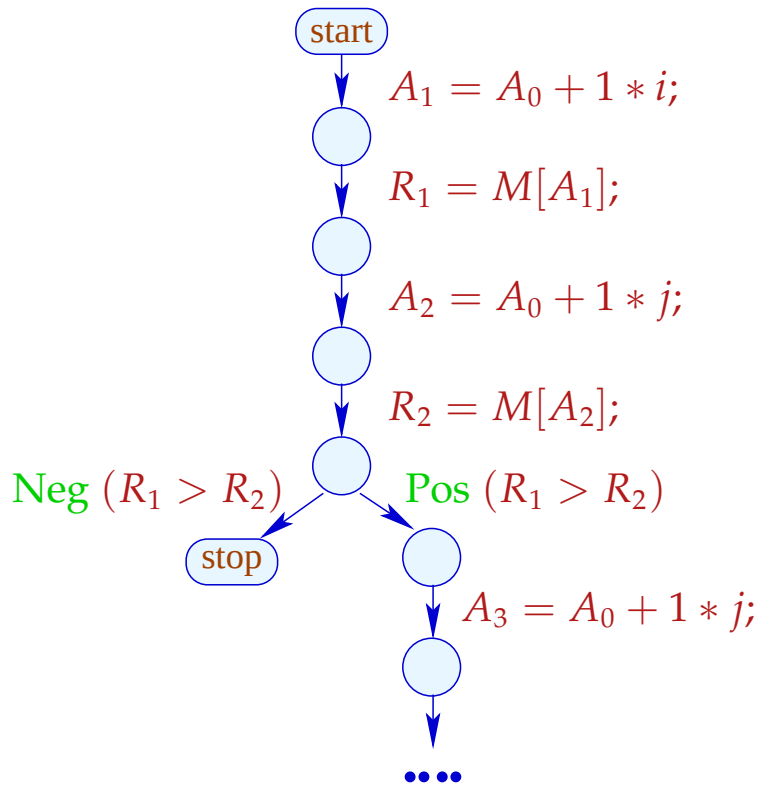
$\rightarrow$ ein Verfahren, das einige Mehrfach-Berechnungen erkennt ...

# Exkurs 1: Eine operationelle Semantik

Wir wählen einen small-step operationellen Ansatz.

Programme repräsentieren wir als Kontrollfluss-Graphen.

Im Beispiel:



Dabei repräsentieren:

| | |
|---|---|
| Knoten | Programm-Punkt |
| start | Programm-Anfang |
| stop | Programm-Ende |
| Kante | Berechnungs-Schritt |

Dabei repräsentieren:

| Knoten | Programm-Punkt |
|---|---|
| start | Programm-Anfang |
| stop | Programm-Ende |
| Kante | Berechnungs-Schritt |

## Kanten-Beschriftungen:

| | |
|---|---|
| **Test** : | Pos $(e)$ oder Neg $(e)$ |
| **Zuweisung** : | $R = e;$ |
| **Load** : | $R_1 = M[R_2];$ |
| **Store** : | $M[R_1] = R_2;$ |
| **Nop** : | $;$ |

Berechnungen folgen Pfaden.

Berechnungen transformieren den aktuellen Zustand

$$s = (\rho, \mu)$$

wobei:

| | |
|---|---|
| $\rho : Vars \to \mathbf{int}$ | Inhalt der Register |
| $\mu : \mathbb{N} \to \mathbf{int}$ | Inhalt des Speichers |

Jede Kante $k = (u, lab, v)$ definiert eine partielle Transformation

$$[\![k]\!] = [\![lab]\!]$$

des Zustands:

$$
\begin{aligned}
[\![;]\!]\,(\rho,\mu) &= (\rho,\mu) \\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) &= (\rho,\mu) && \text{falls } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) &= (\rho,\mu) && \text{falls } [\![e]\!]\,\rho = 0
\end{aligned}
$$

$$
\begin{array}{lcll}
[\![;]\!]\,(\rho,\mu) & = & (\rho,\mu) & \\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho = 0
\end{array}
$$

// $[\![e]\!]$ : Auswertung des Ausdrucks $e$, z.B.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$
\begin{array}{lcll}
[\![;]\!]\,(\rho,\mu) & = & (\rho,\mu) & \\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho = 0
\end{array}
$$

// $[\![e]\!]$ : Auswertung des Ausdrucks $e$, z.B.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 0$

$$
[\![R = e;]\!]\,(\rho,\mu) \quad = \quad (\boxed{\rho \oplus \{R \mapsto [\![e]\!]\,\rho\}}, \mu)
$$

// wobei "$\oplus$" eine Abbildung an einer Stelle ändert