

Helmut Seidl

Programmoptimierung

TU München

Wintersemester 2007/08

Organisatorisches

Termine:

Vorlesung: Montag, 12-14

Dienstag, 12-14

Übung: Freitag, 10-12

Vesal Vojdani: vojdanig@in.tum.de

Materialien: Folien, **Aufzeichnung** :-)

Simulatorumgebung

Vorlesungs-Mitschrift (in Überarbeitung)

- Schein:**
- Bonus durch Aufgaben
 - Klausur

Geplanter Inhalt:

1. Vermeidung überflüssiger Berechnungen

- verfügbare Ausdrücke
- Konstantenpropagation/Array-Bound-Checks
- Code Motion

2. Ersetzen teurer Berechnungen durch billige

- Peep Hole Optimierung
- Inlining
- Reduction of Strength

...

3. Anpassung an Hardware

- Instruktions-Selektion
- Registerverteilung
- Scheduling
- 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:

$$y = f() + f(); \quad \Longrightarrow \quad y = 2 * f();$$

Idee: Spare zweite Auswertung von $f()$...

Beobachtung 3:

Programm-**Verbesserungen** sind nicht immer korrekt :-)

Beispiel:

$$y = f() + f(); \quad \Longrightarrow \quad y = 2 * f();$$

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:

- ⇒ Optimierungen haben **Voraussetzungen**.
- ⇒ Die **Voraussetzungen** muss man:
 - formalisieren,
 - überprüfen :-)
- ⇒ 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
- $R = M[e];$ // Laden
- $M[e_1] = e_2;$ // 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 Statement auch $f()$ gestatten für eine unbekannte Prozedur f .
 - ⇒ intra-prozedural
 - ⇒ eine Art Zwischensprache, in die man (fast) alles übersetzen kann.

Beispiel: `swap()`

```

0 :   A1 = A0 + 1 * i;           //   A0 == &a
1 :   R1 = M[A1];               //   R1 == a[i]
2 :   A2 = A0 + 1 * j;
3 :   R2 = M[A2];               //   R2 == a[j]
4 :   if (R1 > R2) {
5 :       A3 = A0 + 1 * j;
6 :       t = M[A3];
7 :       A4 = A0 + 1 * j;
8 :       A5 = A0 + 1 * i;
9 :       R3 = M[A5];
10 :      M[A4] = R3;
11 :      A6 = A0 + 1 * i;
12 :      M[A6] = t;
      }

```


Optimierung 1: $1 * R \implies R$

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:

$$A_1 = A_0 + i;$$

$$R_1 = M[A_1];$$

$$A_2 = A_0 + j;$$

$$R_2 = M[A_2];$$

if ($R_1 > R_2$) {

$$t = R_2;$$

$$M[A_2] = R_1;$$

$$M[A_1] = t;$$

}

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

→ **speichere** ihn nach der ersten Berechnung;

→ ersetze jede weitere Berechnung durch **Nachschlagen!**

⇒ Verfügbarkeit von Ausdrücken

⇒ Memoisierung

Problem: Erkenne Mehrfach-Berechnungen!

Beispiel:

$$\begin{array}{l} z = 1; \\ y = M[17]; \\ A : x_1 = \boxed{y + z}; \\ \quad \dots \\ B : x_2 = \boxed{y + z}; \end{array}$$

Achtung:

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

- (1) A stets vor B ausgeführt wird; und
- (2) y und z an B die gleichen Werte haben wie an A :-)

⇒ Wir benötigen

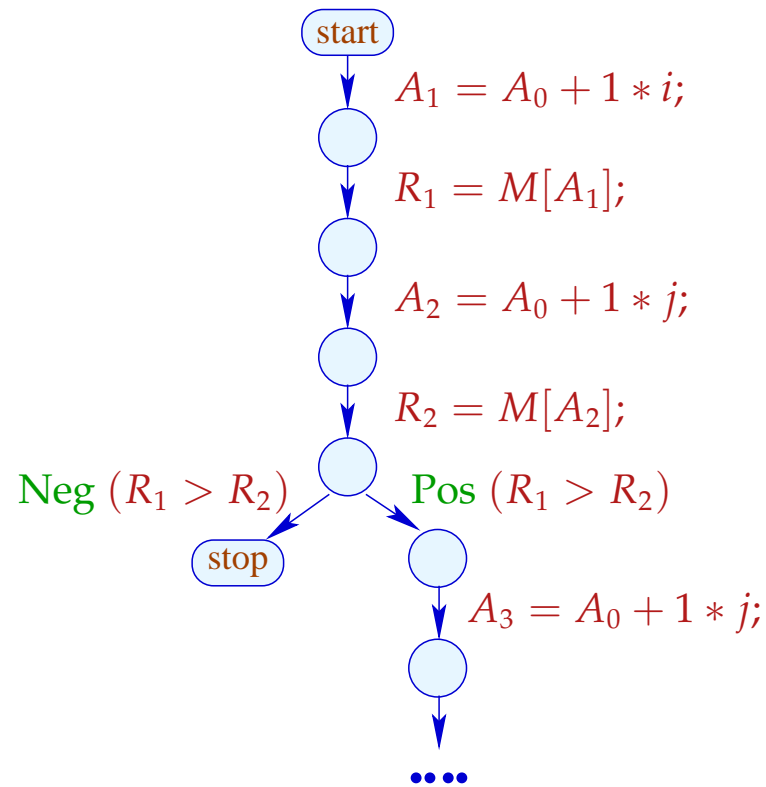
- eine operationelle Semantik :-)
- ein Verfahren, das einige Mehrfach-Berechnungen erkennt ...

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 : $R = M[e];$

Store : $M[e_1] = e_2;$

Nop : ;

Berechnungen folgen **Pfaden**.

Berechnungen transformieren den aktuellen **Zustand**

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

wobei:

$\rho : \text{Vars} \rightarrow \text{int}$	Inhalt der Register
$\mu : \mathbb{N} \rightarrow \text{int}$	Inhalt des Speichers

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

$$\llbracket k \rrbracket = \llbracket lab \rrbracket$$

des Zustands:

$$\llbracket ; \rrbracket (\rho, \mu) = (\rho, \mu)$$

$$\llbracket \text{Pos}(e) \rrbracket (\rho, \mu) = (\rho, \mu)$$

falls $\llbracket e \rrbracket \rho \neq 0$

$$\llbracket \text{Neg}(e) \rrbracket (\rho, \mu) = (\rho, \mu)$$

falls $\llbracket e \rrbracket \rho = 0$

$$\llbracket ; \rrbracket (\rho, \mu) = (\rho, \mu)$$

$$\llbracket \text{Pos}(e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{falls } \llbracket e \rrbracket \rho \neq 0$$

$$\llbracket \text{Neg}(e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{falls } \llbracket e \rrbracket \rho = 0$$

// $\llbracket e \rrbracket$: **Auswertung** des Ausdrucks e , z.B.

$$// \llbracket x + y \rrbracket \{x \mapsto 7, y \mapsto -1\} = 6$$

$$// \llbracket !(x == 4) \rrbracket \{x \mapsto 5\} = 1$$

$$\llbracket ; \rrbracket (\rho, \mu) = (\rho, \mu)$$

$$\llbracket \text{Pos}(e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{falls } \llbracket e \rrbracket \rho \neq 0$$

$$\llbracket \text{Neg}(e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{falls } \llbracket e \rrbracket \rho = 0$$

// $\llbracket e \rrbracket$: **Auswertung** des Ausdrucks e , z.B.

$$// \llbracket x + y \rrbracket \{x \mapsto 7, y \mapsto -1\} = 6$$

$$// \llbracket !(x == 4) \rrbracket \{x \mapsto 5\} = 1$$

$$\llbracket R = e; \rrbracket (\rho, \mu) = (\rho \oplus \{R \mapsto \llbracket e \rrbracket \rho\}, \mu)$$

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

$$\llbracket R = M[e]; \rrbracket (\rho, \mu) = (\rho \oplus \{R \mapsto \mu(\llbracket e \rrbracket \rho)\}, \mu)$$

$$\llbracket M[e_1] = e_2; \rrbracket (\rho, \mu) = (\rho, \mu \oplus \{\llbracket e_1 \rrbracket \rho \mapsto \llbracket e_2 \rrbracket \rho\})$$

Beispiel:

$$\llbracket x = x + 1; \rrbracket (\{x \mapsto 5\}, \mu) = (\rho, \mu) \quad \text{wobei:}$$

$$\begin{aligned} \rho &= \{x \mapsto 5\} \oplus \{x \mapsto \llbracket x + 1 \rrbracket \{x \mapsto 5\}\} \\ &= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ &= \{x \mapsto 6\} \end{aligned}$$