

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 ...

## Beispiel:

## Zwischendarstellung von `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       |

## 5 Perspektiven

Herausforderungen:

- neue Hardware;
- neue Programmiersprachen;
- neue Anwendungen für Compiler-Technologie :-)

## 5.1 Hardware

Die Code-Erzeugung soll die Möglichkeiten der Hardware **optimal** ausnutzen ...

### Herausforderungen:

#### Neue Hardware:

- Speicher-Hierarchie mit unterschiedlich schnellen Caches für verschiedene Zwecke;
- On-Board Nebenläufigkeit mit Pipelines, mehreren ALUs, spekulativer Parallelität, ...
- Interaktion mit mächtigen Zusatzkomponenten wie Graphik-Karten ...

## Eingeschränkte Hardware:

z.B. auf Chip-Karten, in Kühlschränken, Bremsanlagen, Steuerungen ...

⇒ ubiquitous Computing

- minimaler Energie-Verbrauch :-)
- minimaler Platz :-)
- Echtzeit-Anforderungen;
- Korrektheit;
- Fehler-Toleranz.



## 5.2 Programmiersprachen

Spezielle Features:

- mobiler Code;
- Nebenläufigkeit;
- graphische Benutzeroberflächen;
- Sicherheits-Komponenten;
- neue / bessere Typsysteme;
- Unterstützung für Unicode und XML.

## Neue Programmiersprachen:

- XSLT;
- XQuery;
- Web-Services;
- anwendungs-spezifische Sprachen ...

## 5.3 Programmierumgebungen

Diverse **Programmierhilfsmittel** benutzen Compiler-Technologie ...

- syntax-gesteuerte Editoren;
- Programm-Visualisierung;
- automatische Programm-Dokumentation;
- **partielle** Codeerzeugung aus **UML**-Modellen;
- **UML**-Modell-Extraktion  $\implies$  **reverse engineering**
- Konsistenz-Überprüfungen, Fehlersuche;
- Portierung.

## 5.4 Neue Anforderungen

- Zuverlässigkeit
- Sicherheit

... im Rest der Vorlesung behandeln wir ausgewählte Themen, die bei der Code-Erzeugung für **reale** Maschinen relevant sind. Auch hier spielt die Idee der **Generierung** einzelner Komponenten eine wichtige Rolle: