# Kapitel 11: Optimierungen

## Aufgabe

**Verbesserung der Qualität des erzeugten Maschinenprogramms (Laufzeit, Codegröße, Speicherverbrauch, Stromverbrauch)**

## Themen

- **Analysen**
- **Skalare Optimierungen**
- **Schleifenoptimierungen**

# Analysen

Kontrollflußanalyse bestimmt Grundblöcke und baut den Kontrollflußgraphen auf.

Schleifenanalyse erkennt natürliche Schleifen.

Datenflußanalyse bestimmt Datenabhängigkeiten.

Aliasanalyse stellt fest ob Zeiger möglicherweise auf die selbe Adresse zeigen.

Der klassische Algorithmus ist die iterative Analyse.

Ein weiteres Framework ist abstrakte Interpretation.

# Static Single Assignment Form (SSA)

**Die static single assignment Form vereinfacht viele Analysen.**

```
a =                 a0 =
if ()               if ()
   a =                 a1 =
else                else
   a =                 a2 =
                    a3 = ϕ(a1,a2)
= a                 = a3
```

**Die $\phi$ Knoten können durch Kopierbefehle wieder entfernt werden.**

# Skalare Optimierungen

Eine lokale Optimierung ist die Auswertung konstanter Ausdrücke : (4 – 3) * a

Constant propagation leitet konstante Ausdrücke über das ganze Programm weiter.

Copy propagation merkt sich über das gesamte Programm wenn Variablen Kopien anderer sind.

Die Elimination gemeinsamer Teilausdrücke (common subexpression elimination) vermeidet Doppelberechnungen. Eine Verallgemeinerung ist die partial redundancy elimination.

```
if ()
   a = b;
else
   x = a + c;
x = a + c;
```

```
if () {
   a = b;
   x = a + c;
} else
   x = a + c
```

# Programmoptimierungen

**Dead code elimination** enfernt nicht erreichbare Programmteile.

**Function (method) inlining** ersetzt Unterprogrammaufrufe durch Kopien des Unterprogramms. **Procedural abstraction** ist das Gegenteil davon.

**Rekursionseliminierung** ersetzt Rekursionen durch Schleifen.

```
int fak(int n) {
   if (n == 1)
      return n;
   return n * fak(n - 1);
   }
```

```
int fak(int n) {
   int f;
   for (f = 1; n > 1; n--)
      f = f * n;
   return f;
   }
```

**Peephole Optimierung** optimiert kurze Befehlssequenzen.

# Elimination der Induktionsvariablen

**Induction variable elimination erkennt vom Schleifenzähler abhängige Variablen, ersetzt diese und vereinfacht die Adressberechnungen (strength reduction). Diese Optimierung wird oft noch mit invariant code motion kombiniert.**

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      a[i][j] = 0;
```

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      *(a + i * N + j) = 0;
```

```
aptr = a; eptr = a + N * N;
   while (aptr < eptr)
      *aptr++ = 0;
```

# Loop Unrolling

**Der Schleifenrumpf wird vervielfacht um die Schleifenkosten zu reduzieren**

```
for (i = 0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i = 0; i < N - 1; i += 2) {
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
   }
if (N % 2)
   a[N-1] = b[N-1] + c[N-1];
```

# Vektorisierung

**Schleifen mit Zugriffen auf einzelne Felder werden auf Vektoroperationen abgebildet**

```
for (i = 0; i < N; i++)
   a[i]= b[i] + c[i];
```

```
a[0:N-1] = b[0:N-1] + c[0:N-1];
```

# Parallelisierung

**Schleifen werden auf mehrere Prozessoren abgebildet.**

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      a[i][j] = b[i][j];
```

```
for (j = 0; j < M; j++) // CPU 0
   a[0][j] = b[0][j];
...
for (j = 0; j < M; j++) // CPU N-1
   a[N-1][j] = b[N-1][j];
```

# Loop Distribution

**Eine Schleife wird in mehrere Schleifen gespalten.**

```
for (i = 0; i < N; i++) {
   a[i] = 0;
   for (j = 0; j < M; j++)
      b[i][j] = a[i];
   }
```

```
for (i = 0; i < N; i++)
   a[i] = 0;
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      b[i][j] = a[i];
```

# Loop Interchange

**Bei geschachtelten Schleifen werden die innere und äußere Schleife vertauscht.**

```
for (j = 0; j < N; j++)
   for (i = 0; i < N; i++)
      a[i][j] = a[i-1][j]+b[i];
```

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      a[i][j] = a[i-1][j]+b[i];
```

# Loop Fusion

**Mehrere Schleifen werden zu einer kombiniert.**

```
for (i = 0; i < N; i++)
   a[i] = b[i];
for (i = 0; i < N; i++)
   c[i] = a[i] + d[i];
```

```
for (i = 0; i < N; i++) {
   a[i] = b[i];
   c[i] = a[i] + d[i];
   }
```

# Strip Mining

**Strip mining (Abarbeitung in Streifen) wird angewendet, um kurze Vektorregiste auszunützen oder den Cache geblockt anzusprechen**

```
for (i = 0; i < N*32; i++) {
   a[i]= b[i] + 1;
   a[i]= b[i] - 1;
   }
```

```
for (j = 0; i < N*32; j+=32)
   for (i = j; i < j+32; i++) {
      a[i]= b[i] + 1;
      a[i]= b[i] - 1;
      }
```

# Loop Collapsing

**Geschachtelte Schleifen werden in eine einfache transformiert.**

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; i++)
      a[i][j] = b[i][j];
```

```
for (i = 0; i < N*N; i++)
   a[i] = b[i];
```

# Loop Peeling

**Iterationen am Anfang oder Ende werden herausgelöst.**

```
j = N - 1,
for (i = 0; i < N; i++) {
   a[i] = b[j];
   j = i;
   }
```

```
a[0] = b[N-1];
for (i = 1; i < N; i++)
   a[i] = b[i-1];
```

# Software Pipelining

**Mehrere Iterationen einer Schleife werden überlappend ausgeführt.**

```
for (i = 0; i < N; i++) {
   t = a[i];
   b[i] = t;
   }
```

```
t = a[0];
for (i = 1; i < N; i++) {
   b[i-1] = t; t = a[i];
   }
b[N-1] = t;
```