

1 Maschinenunabhängige Optimierungen



Optimierungen

Automatische Optimierungen sind nötig, weil

- unoptimierter Code meist besser lesbar ist.

$x = 5 * y;$ $\iff x = (y \gg 2) + y;$

- höhere Programmiersprachen oft Redundanz erzwingen (z.B. Arrayzugriffe).

$x = a[i] + a[i];$ \implies

```
offset1 = i * sizeof(int);
* a_i1 = &a + offset1;
offset2 = i * sizeof(int);
* a_i2 = &a + offset2;
x = *a_i1 + *a_i2;
```

Gesucht sind **semantikerhaltende Transformationen** die bezüglich eines **Kostenmodells** optimieren.



Prinzip von Kostenmodellen bei Optimierungen

Kostenmodell: Laufzeit eines Programms, eventuell kombiniert mit Energieverbrauch

- Speicherbedarf kann in Laufzeit umgerechnet werden
 - daher ist oft auch Verkürzung des Codes Laufzeitoptimierung
- Statisch nur konservativ abschätzbar wegen Unkenntnis der
 - Anzahl Wiederholungen von Schleifen
 - Sprungbedingungen
- Selbst bei linearem Code nicht statisch bekannt
 - Befehlsanordnung durch Prozessor
 - Pufferspeicher (Cache): Zugriffe auf Speicher sind datenabhängig
 - Fließbandverarbeitung im Prozessor

Laufzeit **gewöhnlich** \neq Summe der Laufzeiten der einzelnen Befehle!



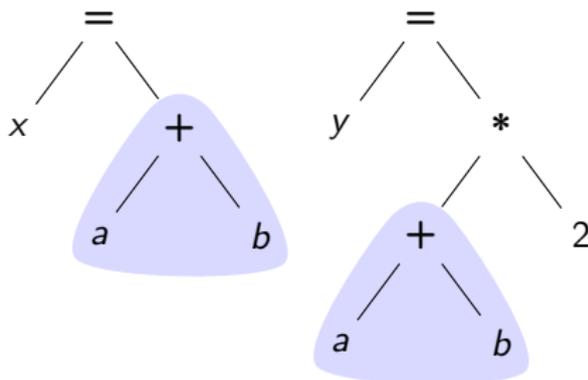
Optimierungen

- Gemeinsame Teilausdrücke
- Kopiepropagation
- Konstanten Falten
- Eliminieren von totem Code
- Codeverschiebung
- Induktionsvariablen und Kostenreduktion
- Load/Store Optimierung
- Arithmetische Vereinfachung
- Ausrollen von Schleifen
- Offener Einbau von Prozeduren (Inlining)
- If Konversion
- ...

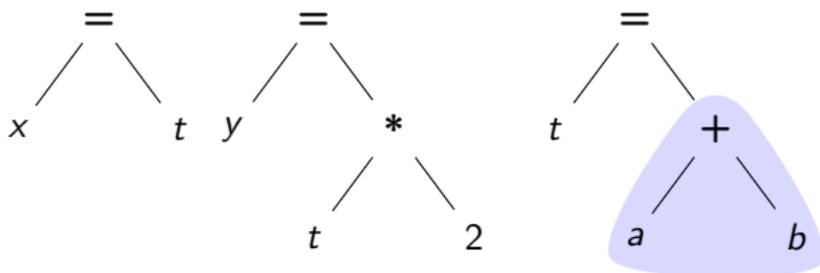


Gemeinsame Teilausdrücke

$x = a + b;$
 $y = (a + b) * 2;$



$t = a + b;$
 $x = t;$
 $y = t * 2;$



Kopiepropagation

- Entfernt Kopieranweisungen der Form $u = v$
- Kopien entstehen z.B. durch entfernen gemeinsamer Teilausdrücke

$$\begin{array}{l} u = v \\ w = u + 3 \end{array} \quad \Rightarrow \quad w = v + 3$$



Konstanten Falten

- Berechne (Teil-)Ausdrücke mit konstanten Operanden
- Wert von Variablen zur Übersetzungszeit bekannt \Rightarrow Kontrollkonstrukte statisch auswerten
- Zu faltende Konstanten werden zu 80 bis 90% bei der Adreßrechnung erzeugt.

```
debug = FALSE
if (debug) {
  print "foo"
} else {
  print "bar"
}
```

```
 $\Rightarrow$  if (FALSE) {
           print "foo"
         } else {
           print "bar"
         }
```



Eliminieren von totem Code

Toter Code : Anweisungen die niemals ausgeführt werden können

- kann als Ergebnis vorangegangener Transformationen auftreten

```
if (FALSE) {  
    print "foo"  
} else {  
    print "bar"  
}
```

\Rightarrow print "bar"



Codeverschiebung

- Optimierung für Schleifen
- Verringert die Anzahl der Befehle einer inneren Schleife
- Zieht **schleifeninvariante Berechnungen** vor die Schleife

```
while (i <= limit - 2) {  
    /* Anweisung aendert limit nicht */  
}
```



```
t = limit - 2  
while (i <= t) {  
    /* Anweisung aendert limit oder t nicht */  
}
```



Induktionsvariablen und Kostenreduktion

Induktionsvariable v : Der Wert der Variable v erhöht sich bei jeder Anweisung um eine positive oder negative Konstante c

Kostenreduzierung (strength reduction) : Transformation von aufwändigen durch preiswertere Operationen (z.B. Multiplikation durch Addition)

| | |
|------------------------|--------------------------------------|
| | $x = 0$ |
| for (i=0; i<10; ++i) { | \Rightarrow for (i=0; i<10; ++i) { |
| $x = 13 * i$ | $x = x + 13$ |
| } | } |



Load/Store Optimierung

- Entfernen unnötiger Zugriffe auf den Heap
- Vorsicht: Aliase, Threads, ...

```
// store a.x  
a.x = Q
```

```
/* a.x wird nicht */  
/* veraendert */
```

```
// load a.x  
x = a.x
```



```
t = Q  
// store a.x  
a.x = t
```

```
/* a.x wird nicht */  
/* veraendert */
```

```
x = t
```



Arithmetische Vereinfachung

- Vereinfachung arithmetischer Ausdrücke
- Bevorzugen preiswerter Operationen

$$x = -a + b$$

$$y = c * a + c * b$$

$$z = a + b + c - (a + b)$$

$$v = 0 + v$$

$$w = 1 \& w$$

$$x = a - b$$

$$y = c * (a + b)$$

$$\Rightarrow z = c$$

$$v = v$$

$$w = w$$



Ausrollen von Schleifen

- Das Auswerten der Abbruchbedingung ist ein Kostenfaktor
- Verbesserung der Geschwindigkeit auf Kosten der Größe

```
for (i=1; i<=3; i++) {  
    print i  
}  
⇒  
print 1  
print 2  
print 3
```

```
for (x=1; x<=1000; x++) {  
    foo(x)  
}  
⇒  
for (x=1; x<=1000; x+=5) {  
    foo(x)  
    foo(x + 1)  
    foo(x + 2)  
    foo(x + 3)  
    foo(x + 4)  
}
```



Offener Einbau von Prozeduren (Inlining)

- In Prozedur p , ersetze Prozeduraufruf an q durch den Rumpf von q unter Ersetzung der formalen Parameter durch die Argumente des Aufrufs.
- Verwaltungsaufwand für Prozeduraufruf:
 - Ablage der Argumente durch den Aufrufer
 - Sichern des Kontextes des Aufrufers
 - Erstellen des Stackframes der aufgerufenen Prozedur
 - Abbau des Stackframes der aufgerufenen Prozedur
 - Wiederherstellen des Kontextes des Aufrufers
 - Rückgabe des Ergebnisses an den Aufrufer
- Wird q an allen Aufrufstellen offen eingebaut, dann ist q redundant
- Optimierung des offen eingebauten Codes im Kontext der Aufrufer



Offener Einbau von Prozeduren (Inlining)

```
int add(int a, int b) {  
    return a + b;  
}
```

```
int main() {  
    return add(3, 4);  
}
```

\Rightarrow

```
int main() {  
    return 3 + 4;  
}
```



Offener Einbau von Prozeduren (Inlining)

`_add:`

```
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret
```

`_main:`

```
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl $4, 4(%esp)
    movl $3, (%esp)
    call _add
    leave
    ret
```



`_main:`

```
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $4, %ebx
    movl $3, %eax
    addl %ebx, %eax
    leave
    ret
```



If Konversion

- Vermeiden falscher Sprungvorhersagen (\rightarrow Verlust vieler Takte bei Fließbandverarbeitung)
- Parallele Berechnung + spätere Auswahl

```
if (<expr>) {  
    x = 3 + y;  
} else {  
    x = 2 + z;  
}
```

```
flag = <expr>  
 $\Rightarrow$  x1 = 3 + y  
      x2 = 2 + z  
      x = (flag ? x1 : x2)
```

