

Optimierungstechniken

(Compiler)

Prof. Dr. Oliver Braun

Letzte Änderung: 27.06.2017 11:13

- ▶ nach der Übersetzung in die Zwischenrepräsentation kann der Compiler Optimierungen durchführen
- ▶ übliche Optimierungsziele:
 - ▶ schnellere Ausführung des kompilierten Codes
 - ▶ geringerer Energieverbrauch
 - ▶ weniger Speicherverbrauch

Sicherheit (safety)

- ▶ der optimierte Code muss die gleichen Ergebnisse wie der ursprüngliche Code liefern

Rentabilität (profitability)

- ▶ die Optimierung muss sich *lohn*en

Optimierungsansätze

- ▶ Abstraktionsoverhead reduzieren
 - ▶ z.B. Datenstrukturen und Typen “wegoptimieren”
- ▶ Vorteile von Spezialfällen nutzen
 - ▶ z.B. Funktionsaufrufe analysieren und statt dynamisch, statisch binden
- ▶ Code an Systemressourcen anpassen
 - ▶ wenn die Voraussetzungen des Programms nicht vom Prozessor geleistet werden können

Bereiche von Optimierungsmethoden

- ▶ lokale Methoden
 - ▶ optimieren innerhalb eines *single basic blocks* (keine Verzweigungen)
- ▶ regionale Methoden
 - ▶ mehr als ein einzelner Block, aber noch weniger als eine Prozedur
 - ▶ z.B. eine Schleife die eine if-Anweisung enthält
- ▶ globale Methoden (intraprozedurale Methoden)
 - ▶ gesamte Prozedur als Kontext
- ▶ Interprozedurale Methoden (*whole-program methods*)
 - ▶ betrachten mehr als eine Prozedur

Lokale Optimierungen

▶ Beispiel:

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - d$$

▶ ist da etwas redundant?

Lokale Optimierungen

- ▶ der Block kann ersetzt (*rewritten*) werden durch

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = b$$

- ▶ denn weder a noch d werden zwischen der 2. und 4. Zeile verändert
- ▶ nachdem b in Zeile 2 verändert wird, kann die Zeile 3 **nicht** durch

$$c = a$$

ersetzt werden

- ▶ ein Algorithmus der solche Redundanzen findet und beseitigt, ist der *local value numbering* Algorithmus

Höhenbalancierte Bäume

- ▶ ein Parsebaum für den Ausdruck $a+b+c+d+e+f+g+h$ ist mit den vorgestellten Parsing-Algorithmen entartet
- ▶ wird dieser Parsebaum in einen höhenbalancierten transformiert, können Teile der Berechnung in verschiedenen Addierwerken durchgeführt werden

Regionale Optimierungen

- ▶ *superlocal value numbering*
 - ▶ *local value numbering* mit erweitertem Bereich
- ▶ *loop unrolling*
 - ▶ eine Schleife wird durch Kopien des Schleifenrumpfs ersetzt
 - ▶ bei verschachtelten Schleifen werden die entstehenden Rümpfe der inneren Schleife zusammengefasst (*loop fusion*)
 - ▶ solche Kombinationen nennt man auch *unroll-and-jam*

Globale Optimierungen

- ▶ nicht initialisierte Variablen finden und an den Benutzer melden
- ▶ Code so anordnen, dass wahrscheinlich weniger Sprünge bei der Ausführung gemacht werden müssen (*global code placement*)
 - ▶ dazu können bei Verzweigungen untersucht werden welcher Pfad der wahrscheinlichere ist
 - ▶ dieser wird dann so angeordnet, dass kein Sprung notwendig ist und der Code linear (im *fall-through branch*) ausgeführt wird
 - ▶ der unwahrscheinlichere Block wird so in den Speicher platziert, dass hin- und auch wieder zurück gesprungen werden muss

Interprozedurale Optimierungen

- ▶ *inline substitution*

- ▶ ein Prozeduraufruf wird durch den Rumpf der Prozedur ersetzt
- ▶ wie kann das in C++ vom Programmierer forciert werden?

- ▶ *procedure placement*

- ▶ in welcher Reihenfolge Prozeduren in einer ausführbaren Datei angeordnet werden, kann Einfluß auf den virtuellen Speicher und den Cache haben

Compiler-Organisation für Interprozedurale Optimierungen

- ▶ traditionelle Compiler haben als *compilation unit* eine einzelne Prozedur, Klasse oder Datei
- ▶ Ansätze um dennoch in größerem Scope optimieren zu können
 - ▶ größere Einheiten die zusammen verarbeitet werden
 - ▶ Zusammenarbeit Compiler - IDE (z.B. per Reflection)
 - ▶ Interprozedurale Optimierungen erst im Linker

Skalare Optimierungen

- ▶ nutzlosen und nicht erreichbaren Code eliminieren
- ▶ *code motion*
 - ▶ eine Berechnung an einen Punkt verschieben, wo Sie weniger oft ausgeführt wird
 - ▶ Beispiel: Code aus einer Schleife entfernen
- ▶ *code hoisting* = spezielle *code motion*
 - ▶ wenn ein Berechnung am Ende eines Blockes vor jedem Pfades ausgeführt wird, kann diese in den Block verschoben werden
 - ▶ dadurch wird der compilierte Code auch kleiner

Spezialisierungen

- ▶ *tail-call optimization*
 - ▶ wenn die letzte Aktion einer Prozedur p der Aufruf einer anderen Prozedur q ist, kostet der Kontextwechsel von q nach p unnötig
- ▶ *leaf-call optimization*
 - ▶ wenn klar ist, dass eine Prozedur keine andere aufruft, muss kein Code eingefügt werden, der nur dazu da ist eine Prozedur nach dem Aufruf einer anderen weiter auszuführen