Humboldt-Universität zu Berlin

Institut für Informatik

Codeoptimierung

SS 00

Übungsblatt 6
loop – Einzelarbeit

Abgabe: 27.6.00 vor 9:30h

Die Lösung dieses Übungsblattes muß in Einzelarbeit erfolgen, keine Gruppenarbeit! Sollte eine Abgabe eine Kopie (mit oder ohne Modifikation) einer anderen Abgabe sein, findet keine Bewertung statt (weder des Orginals noch der Kopie). Studenten können sich über Lösungskonzepte austauschen (z. B. Pseudo-Code), nicht jedoch über die tatsächliche Implementierung! Wer sich nicht sicher ist, ob eine "Zusammenarbeit" diesen Grundsätzen entspricht, sollte sich *vor der Abgabe* per e-mail an mich wenden und nachfragen.

loop ermittelt die Schleifen eines Kontrollflußgraphen. Dazu sollen zunächst mit dem Algorithmus auf S. 822 des Drachenbuches die Dominatoren bestimmt werden. Für jeden Block B soll eine Liste "doms" erstellt werden, die genau die Blöcke enthält, die den Block B dominieren. Darauf aufbauend werden mit Algorithmus 10.1 (S. 739) die natürlichen Schleifen berechnet. Eine Liste der Blöcke einer Schleife wird dabei mit dem Kopf der Schleife assoziiert, d.h. die Komponente "loops" der Blockstruktur ist genau dann nicht leer, wenn es sich um einen Kopf handelt. Die Dominatorliste und die Schleifenblockliste werden bereits von dem Programm ausgegeben (zum Testen und Bewerten der Aufgabe). Weiterhin wird der Flußgraph bereits für jede Funktion erstellt. Zur Bearbeitung der Aufgabe loggen Sie sich bitte auf einer Sun SPARC ein und geben (in Ihrem Home-Verzeichnis) die folgenden Befehle ein:

```
mkdir ~/codeopt
mkdir ~/codeopt/ub6
cd ~/codeopt/ub6
cp ~mueller/classes/codeopt/ub6/* .
```

(10 Punkte) Erweitern Sie dann die Datei *loop.c* (das C Programm). Verändern Sie dazu nur die bisher nicht ausgefüllten Fuktionen. Testen Sie das Programm wie üblich, stellen Sie z.B. den Kontrollfluß mittels der Nachfolger/Vorgängerinfos graphisch dar und vergleichen Sie Ihre Ergebnisse mit dem Graphen.

```
ub1.cc ~mueller/test/src/cal.c   # erzeugt Assembler-Code "cal.s"
loop < cal.s > cal.S             # erzeugt optimierten Assembler-
Code "cal.S"
ub1.ld cal.S                     # erzeugt ausf{\"u}hrbares Pro-
gramm "cal.out"
cal.out 1995                     # Programm ausf{\"u}hren
```

Die Abgabe umfaßt die Datei *loop.c* und erfolgt per Upload durch http://eiche.informatik.hu-berlin.de:8081/codeopt bis zum oben genannten Datum.

- Entdecken von Induktionsvariablen, Algorithmus (Drachenbuch S. 788), erweitert für VPO:

```
Eingabe: Schleife, defs, invariante Berechnungen
Ausgabe: Induktionsvariablen X der Form (i, c, d) mit
           i = Grund-Induktionsvariable, c und d konstant, so da{\ss}
           X = c * i + d (am Definitionspunkt von X)
Algorithmus:
1. Pass: Finde Grund-Induktionsvariablen mit Trippel (i, 1, d),
         d = Int-Konst.
2. Pass:
  while change do
    Finde restliche Induktionsvariablen mit X = c * i + d,
      c = Int-Konst, d = invarianter Ausdruck
    if i ist Grund-Induktionsvariable then
      neues Trippel (i, c, d) geh{\"o}rt zur Familie i
    else
      i hat Trippel (g, old_c, old_d) der Familie g,
      neues Trippel (g, old_c * c, old_d * c + d) der Familie g, wobei
        keine Def. von g zwischen Def. von i und X liegt (in Schleife),
        und keine Def. von i au{\ss}erhalb der Schleife Def. X erreicht
3. Pass: Finde Speicherzugriff, dessen Operationskosten reduziert
         werden k{\"o}nnen

Beispiel:
for (i = 1; i < n; i++) {
  y = i*3+4; /* y: (i, 3, 4) */
  ...
  z = y*5+6; /* z = (i*3+4)*5+6 = i*(3*5)+(4*5)+6  z: (i, 15, 26) */
}
```

- Algorithmus zur Reduktion von Operationskosten (Drachenbuch S. 791):

```
for each Induktionsvariable X: (i, c, d) mit X <> i) do
  if kein Register frei then
    continue;
  W{\"a}hle Register R aus;
  Generiere "R = i * c + d;" im Vorkopf;
  Ersetze letztes Use von X durch R in Schleife;
    /* andere Sets/Uses sind tot */
  Generiere "R = R + c * d_i;" am Schleifenende,
    wobei i: (i, 1, d_i);
```

- Eliminierung von Induktionsvariablen, Algorithmus (Drachenbuch S. 792):

```
for each Grund-Induktionsvariable i : (i, c, d) do
  if i in compare-and-branch instruction(s) then
    Finde g{\"u}nstige Induktionsvariable x : (i, new_c, new_d)
      der Familie i
    Ersetze "NZ=i?k;" mit "NZ=x?r;" wobei k konstant;
    Generiere "r=new_c*k+new_d;" im Vorkopf;
Eliminiere Grund-Induktionsvariablen, die nur inkrementiert/
  dekrementiert, jedoch nicht benutzt werden
  (auch nicht au{\ss}erhalb der Schleife);
```

- Algorithmus Konstruktion des Konfliktgraphen, siehe build_interference_graph() in dataflow.c:

```
for each Block X do
  if Variable V lebendig am Anfang von X then
    erzeuge ignode N;
    f{\"u}ge N in Conin ein (Liste lebendiger Var. am Anfang);
    f{\"u}ge N in Conout ein (Liste momentan lebendiger Var.);
    markiere Konflikt zwischen V und anderen Knoten in Conin;
  if Register R lebendig am Anfang von X then
    f{\"u}ge R zur Regstate-Liste der Variablen in Conin hinzu;
    f{\"u}ge R zur Liste der momentan lebendigen Register hinzu;
  for each RTL do
    if V referenced (used) then
      erh{\"o}he Frequenz von N;
    if V dead then
      l{\"o}sche V von Conout;
    if R dead then
      l{\"o}sche R von Liste der momentan lebendigen Register;
    if V set (und noch nicht lebendig) then
      erzeuge ignode N;
      markiere Konflikt zw. V und anderen Knoten in Conout;
      speichere die momentan lebendigen Register in N;
      f{\"u}ge N zu Conout hinzu;
    if R set then
      f{\"u}ge R zu den momentan lebendigen Registern hinzu;
      l{\"o}sche R aus Regstate-Listen aller ignode in Conout;
```

- Algorithmus Färben des Graphen, siehe sort_interference_graph() und color_global_allocation() in locals.c:

```
sortiere ignodes des Konfliktgraphen
  (gr{\"o}{\ss}te Frequenz zuerst);
for each ignode (Lebensraum der Variablen V)
  vereinige die Regstates aller Knoten im Lebensraum;
  if Lebensraum f{\"a}rbbar (d.h. noch ein Register R frei) then
    ersetzte Auftreten von V durch R im Lebensraum;
    f{\"u}ge R zu den Regstate-Listen der ignodes,
      die in einem Konflikt stehen;
```

- Beispiel:

```
int f, g, h;
foo()
{
  int a, b, c, d, e;

  a = f;
  b = g;
  if (a == b) {
    c = a + 5;
    a = b + c + c;
    d = 6;
    g = d;
  }
  else {
    c = 7;
    g = c;
  }
  e = a + 8 * a;
  h = e;
}
```

ignode	item	freq	block	conflicts	regstate	#
0	c	3	2	8	r(8)r(9)r(10)r(11)	5.
1	e	2	4			11.
2	a	2	4		r(8) r(10)r(11)	10.
3	a	2	1	5	r(8)r(9)r(10)r(11)	1.
4	c	2	3	10	r(10)	9.
5	b	2	1	3	r(8) r(10)r(11)	2.
6	d	2	2	9	r(8)r(9)r(10)r(11)	7.
7	a	1	2	8	r(8) r(10)r(11)	3.
8	b	1	2	0,7	r(8)r(9)r(10)	4.
9	a	1	2	6	r(8)r(9)r(10)	6.
10	a	0	3	4	r(8)r(9)r(10)	8.

V	Lebensraum (Blöcke)	Lebensraum(ignodes)	Block	Conin	Conout
a	(4,1,2,2,3)	(2,3,7,9,10)	1		3,5
b	(1,2)	(5,8)	2	7,8	9
c	(2),(3)	(0),(4)	3	10	10
d	(2)	(6)	4	2	
e	(4)	(1)			