Codeoptimierung – Übungsblatt 3
chain – Einzelarbeit

Abgabe: 23.5.00 vor 9:30h

(10 Punkte) *chain* ist ein Optimierer, der Sprungketten reduziert. Er liest Code von *stdin* und schreibt optimierten Code nach *stdout*. Wenn eine Sprunganweisung S der Form

```
        bxx     Lyy
```

für beliebige Zeichenketten `xx, yy` erkannt wird, dann wird die Anweisungen am Springziel überprüft. Befindet sich dort eine unbedingte Sprunganweisung T, d.h. für beliebige `zz`

```
Lyy:                    oder    Lyy:
        ba      Lzz                     ba,a    Lzz
```

dann soll Label `Lyy` in S durch `Lzz` ersetzt werden. Man beachte, daß die Suche für Label `Lzz` wiederholt werden muß, damit transitiv Sprungketten durch das letztmögliche Label ersetzt werden. Es sollen dabei weder Labels noch Anweisungen gelöscht werden.

Beispiel:

```
Eingabe:                Ausgabe:
...                     ...
        bge     L10             bge     L12
...
L10:                    L10:
        ba      L11             ba      L12
...
L11:                    L11:
        ba      L12             ba      L12
...
L12:                    L12:
        retl                    retl
```

Zur Bearbeitung der Aufgabe loggen Sie sich bitte auf einer Sun SPARC unter Solaris 2.x ein und geben (in Ihrem Home-Verzeichnis) die folgenden Befehle ein:

```
mkdir ~/codeopt
mkdir ~/codeopt/ub1
cd ~/codeopt/ub1
cp ~mueller/classes/codeopt/ub1/* .
```

Erweitern Sie dann die Dateien *chain.c* (das C Programm). Das Programm enthält bereits Datenstrukturen und Routinen zum Lesen/Schreiben der Befehle innerhalb einer Funktion, wobei Sequenzen von whitespace-Zeichen zu genau einem Leerzeichen reduziert werden, damit Befehle einfacher erkannt werden können. Benutzen Sie das *Makefile*, um die ausführbare Datei *chain* zu generieren. Testen Sie das Programm z.B. wie folgt:

```
ub1.cc ~mueller/test/src/cal.c   # erzeugt Assembler-Code "cal.s"
chain < cal.s > cal.S            # erzeugt optimierten Assembler-
Code "cal.S"
ub1.ld cal.S                     # erzeugt ausf{\"u}hrbares Pro-
gramm "cal.out"
cal.out 1997                     # Programm ausf{\"u}hren
```

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

Optimierende Compiler im Vergleich

Optimierung	Sparc	Mips	HP	IBM	VPO
Tail Rekursion eliminieren	g	g	-	-	-
Codeverschiebung	g	g	g	-	g
Elim. von Induktionsvariablen	g	g	g	g	g
Reduktion von Operationskosten	g	g	g	g	g
Elim. von gemeins. Teilausdr.	g	g	g	g	g
Propagierung von Kopien/Konst.	g/m	g	g/m	g	g
Registervergabe	g	g	g	g	g
Registerzuweisung	g	g	g	g	g
Entrollen von Schleifen	g	g	-	-	[g]
Software Pipelining	-	-	g	g	-
Elim. von totem Code	g	g	g/m	-	g
Fusion von Grundblöcken	-	g	-	g	g
Inlining	g/m	-	g	-	[g]
Cloning (part. Auswertung)	-	m	g	-	-
Schleifenparallelisierung	-	-	g	-	-
Cacheopt.	g	m	g	-	[g]
Peephole-Opt.	m	m	g/m	m	g/m
Elim. von Speicherzugriffen	m	g	g	g	[g/m]
Schleifenumkehrung	m	-	-	-	-
spez. Anpassung an Architektur	m	m	m	-	g/m
Registerzusammenlegung	m	-	m	-	-
Instruktionsscheduling	m	m	m	m	g/m
Opt. von Blattfunktionen	m	m	-	-	g/m
Elim. von Sprungketten	m	-	g/m	-	g

g: globale Optimierungsphase
m: maschinenabhängige Optimierungsphase
-: keine Information

Very Portable Optimizer (VPO) – Beispiel ex_cgen

C Programm

```
extern int b;

main()
{
  int a;

  b = a + 1;
}
```

C-Code

```
NAME a local
DEREF INT
CON 1
PLUS INT
NAME b global
ASSIGN INT
```

Expander RTLs

```
Mbwrfd
-       .seg    "data"
-       .seg    "text"
-       .align  8
-       .global _main
-_main:
fmain
d.1_a   LOC[0]  2       2       4
+r[32]=r[14]+LOC[0]
+r[33]=R[r[32]]         r[32]
+r[34]=1
+r[33]=r[33]+r[34]      r[34]
d_b     GLO[0]  0       2
+r[35]=HI[GLO[0]]
+r[35]=r[35]+LO[GLO[0]]
+R[r[35]]=r[33]         r[35]r[33]
L14
+PC=RT
*
-       .seg    "data"
```

Optimierte RTLs

```
-       .seg    "data"
-       .seg    "text"
-       .align  8
-       .global _main
-_main:
-.1_a = 64
+1          r[14]=SV[r[14]+-72];
+2          r[8]=HI[_b];
+3          r[9]=R[r[14]+.1_a];     r[14].1_a
+4   3      r[9]=r[9]+1;
+5   4 2    R[r[8]+LO[_b]]=r[9];    r[8]r[9]
+6          PC=RT;
+7          NL=RS[];
*
-       .seg    "data"
```

Assembler Code

```
        .seg    "data"
        .seg    "text"
        .align  8
        .global _main
_main:
.1_a = 64
        save    %sp,(-72),%sp
        sethi   %hi(_b),%o0
        ld      [%sp + .1_a],%o1
        add     %o1,1,%o1
        st      %o1,[%o0 + %lo(_b)]
        ret
        restore
```

Anordnung der Optimierungen in VPO

```
Elim. von Sprungketten
Elim. leerer Grundblöcke
Elim. unnötiger Sprünge
Elim. von totem Code
Umkehrung von Sprüngen
Umordnen der Grundblöcke zur Reduzierung von Sprüngen
Fusion von Grundblöcken
lokale Instruktionsauswahl (Peephole-Opt.)
globale Instruktionsauswahl (Peephole-Opt.)
lokale Registervergabe
IF change
  Instruktionsauswahl (Peephole-Opt.)
Reduzieren von Sprüngen in Schleifen
Fusion von Grundblöcken
REPEAT
  globale Registervergabe (Registerfärben)
  Instruktionsauswahl (Peephole-Opt.) wegen Spills
UNTIL !change
REPEAT
  Elim. gemeinsamer Teilausdrücke (CSE)
  Elim. toter Variablen
  Codeverschiebung
  Recurrences
  Elim. von Induktionsvariablen
  Elim. unnötiger Sprünge
  Reduktion von Operationskosten
  Elim. toter Variablen
  Instruktionsauswahl (Peephole-Opt.)
UNTIL !change
Elim. von Sprungketten
Elim. von totem Code
Elim. leerer Grundblöcke
Umkehrung von Sprüngen
Umordnen der Grundblöcke zur Reduzierung von Sprüngen
Generieren des Entry- und Exit-Codes
Instruktionsscheduling
Füllen von Delay-Slots
Elim. unnötiger Sprünge
Elim. von Sprungketten
Assemblercode-Generierung
```