Methoden zur Code-Optimierung in GCC

GCC-Versionen

GCC entstanden 1984, ungefähr in seiner heutigen Form

PGCC entstanden 1995 aus den Intel-Patches

EGCS entstanden 1997; Brennpunkt der Entwicklung

Preprocessing

↓

Lexing/Parsing

↓ Frontend-Optimierungen (sprachenspezifisch)

Baumdarstellung ("tree")

↓ Backend-Optimierungen auf Trees

Register Transfer Language (RTL)

↓ Backend-Optimierungen auf RTL oder anderen Formen

Assembler

↓ Optimierungen auf Assembler-Ebene

Objektdatei

↓ Optimierungen auf Datei-Ebene (Linker)

Ausführbares Programm

Interne Darstellungen

HIR high level intermediate representation, entspricht den GCC-Trees

MIR medium level intermediate representation, entspricht RTL *vor* der „reload“-Phase

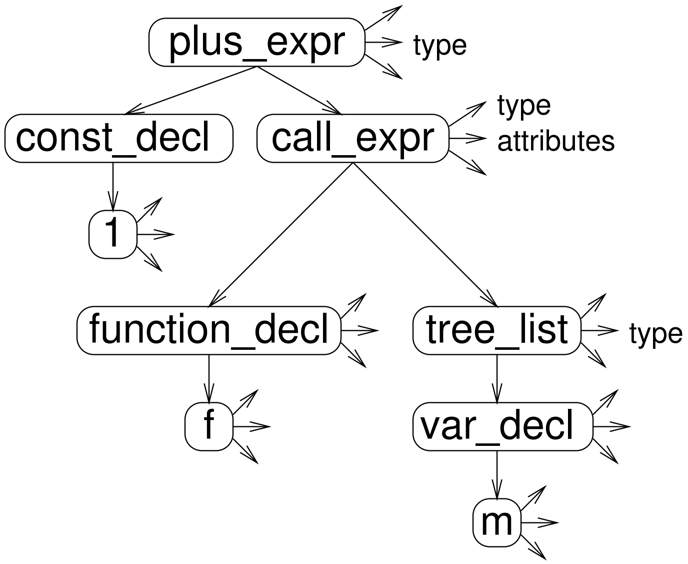
LIR low level intermediate representation, entspricht RTL *nach* der „reload“-Phase

Interne Darstellungen / Trees

- eine Funktion (bzw. die aktuelle Anweisung) wird als Baum dargestellt
- architekturunabhängig
- weitestgehend sprach**un**abhängig
- dient in GCC als Schnittstelle zwischen Front- und Backend
- sehr detailiert (Quellcode läßt sich weitestgehend restaurieren)
- sehr speicheraufwending

- Beispiel

  `1 + f(m)`



Interne Darstellungen / RTL

- RTL = Register Transfer Language
- GCC-Spezifisch, entspricht dem 3-Address code/MIR/LIR anderer Compiler:
- stark architekturabhängig (die „machine description“ bestimmt, welche RTL-Kombinationen möglich sind)
- stark pass-abhängig
- LISP-ähnlich
- 1 + f(m)

```
(insn 11 8 13 (set (mem:SI (pre_dec:SI (reg:SI %esp)) 0)
                   (reg:SI 23)))
(call_insn 13 11 15 (set (reg:SI %eax)
                         (call (mem:QI (symbol_ref:SI ("f")) 0)
                               (const_int 4))))
(insn 15 13 17 (set (reg:SI 25)
                    (reg:SI %eax)))
(insn 17 15 19 (set (reg:SI 24)
                    (plus:SI (reg:SI 25)
                             (const_int 1))))
(insn 19 17 20 (set (reg/i:SI %eax)
                    (reg:SI 24)))
(insn 20 19 22 (use (reg/i:SI %eax)))
```

```
if (c > 3) c = 7
```

```
;; Start of basic block 0, registers live: 6 [bp] 7 [sp]
(note 6 4 8 0 NOTE_INSN_BLOCK_BEG)
(insn:TI 8 6 10 (set (reg/v:SI 0 %eax) (mem:SI (symbol_ref:SI ("x")) 0))
                58 {movsi+2} (nil) (nil))
(insn 10 8 11 (set (cc0) (compare (reg/v:SI 0 %eax) (const_int 3)))
               16 {cmpsi_1} (insn_list 8 (insn_list 8 (nil))) (nil))
(jump_insn/s 11 10 13 (set (pc) (if_then_else
                                   (le (cc0) (const_int 0))
                                       (label_ref 14) (pc))))
;; Start of basic block 1, registers live: 6 [bp] 7 [sp]
(insn 13 11 14 (set (reg/v:SI 0 %eax) (const_int 7))
                58 {movsi+2} (expr_list:REG_EQUAL (const_int 7) (nil)))
;; Start of basic block 2, registers live: 0 [ax] 6 [bp] 7 [sp]
(code_label 14 13 17 2 "")
```

Strukturanalyse

- Grundblöcke (basic blocks) werden gesucht und erkannt
  - Sequenz aus Befehlen, die von einem Sprunglabel oder Sprung begrenzt wird
- erweiterte Grundblöcke (extended basic blocks)
  - Sprünge aus dem Block heraus beenden den erweiterten Block nicht
- Schleifen werden gesucht und klassifiziert

„lokale“ Optimierungen Optimierungen, die nur innerhalb eines Basic-Blocks operieren („linear“)

„globale“ Optimierungen Optimierungen, die über Blockgrenzen hinweg optimieren

Frontend (C, C++, Java, ...)
Aliasanalyse

Backend (x86, alpha, ...)
Tree constant folding
RTL function inlining
tail-recursion elimination
Sprungoptimierung
common subexpression elimination (CSE)
global CSE (GCSE)
- partial redundancy elimination (PRE)
- global constant propagation
- copy propagation

invariant code motion
strength reduction
loop unrolling
(CSE + Sprungoptimierung)

Datenflußanalyse
instruction combining
instruction scheduling
register class preferencing
local register allocation
global register allocation

register reloading

instruction scheduling 2
Sprungoptimierung
delayed branch scheduling
reg-stack (nur x86)

final

Aliasanalyse

- keine Optimierung
- stellt wichtige Informationen zur Verfügung
- sprachabhängig (Information kommt aus dem Frontend)
  - Objekte mit mehreren Namen

```
void
memcpy (char *dst, char *src, int len)
{
  while (len--)
    *dst++ = *src++;
}
```

- Alias von globalen/lokalen Variablen

```
int i, *p = &i;
i = 0;
f();
*p = 0;
```

Datenflußanalyse

- Auch: *life analysis*, *dataflow analysis*: wann werden welche Ergebnisse noch benötigt, wann sind sie *aktiv*

```
i = 5;
j = i+4;
k = i+2;
m = k*2;
return i+m;
```

- In GCC repräsentiert durch death notes:

```
(insn 15 13 16
      (set (reg/i:SI 0 %eax)
           (reg:SI 26)) 58 {movsi+2}
      (insn_list 13 (nil))
      (expr_list:REG_DEAD (reg:SI 26)
      (nil)))
```

constant folding & algebraic simplification

- Beispiele für constant folding:

```
i = j*3*3; => i = j*9;
i = 1+j-1; => i = j;
```

- Beispiele für algebraic simplification:

```
i = j+0;    => i = j;
i = j*1;    => i = j;
i = j*2;    => i = j+j;
i = j**2;   => i = j*j;
```

- elementare Optimierung, wird (in vereinfachter Form) in verschiedenen Teilen des Compilers benutzt

- viele Umformungen sind abhängig von

  – der Sprache (exceptions in Ada, Klammerregelung in Fortran)
  – den Datentypen (es gibt kaum gültige Umformungen für IEEE-Fließkomma)
  – der Zielmaschine (ist $v * v$ schneller als v^2?)

function inlining

- Ausgewählte Funktionen (frontend- und backend-abhängig) werden direkt in aufrufende Funktionen eingebaut

- GCC kann nur Funktionen inlinen die er schon einmal „gesehen“ hat (d.h. die vor dem Aufruf definiert wurden). Denkbar sind folgende Verbesserungen:
  - „Globales“ (d.h. dateibezogenes) inlining, für Funktionen die erst nach dem Aufruf definiert werden
  - Dateiübergreifendes Inlining
  - Partielles Inlining (besonders günstig für C++-Methoden mit „guards“):

```
void someclass::somemethod () {
  if (initialized)
    {
      do_something...
    }
}
```

- Aus dem GCC-Handbuch: „inline functions are as fast as macros" — Pustekuchen:

  ```
  inline int mult (int a, int b) { return a*b; }
  #define MULT(a,b) ((a)*((b))
  mult(a,3) => imull $3,%eax,%eax
  MULT(a,3) => leal (%eax,%eax,2),%eax
  ```

  – Massiver Eingriff in GCC nötig

tail-recursion elimination

- relativ spezielle Optimierung
- wenn anwendbar jedoch sehr effektiv (spart Zeit und Speicher) (Beispiel: sparc-port für Linux)
- endrekursive Aufrufe werden iterativ umgeschrieben

```
f1(...) { .... return f2(...); }
  call f2  =>  jump f2
  return   =>
```

jump optimization / Sprungoptimierung

- elementare Optimierung, wird an mehreren Stellen aufgerufen
- dead code elimination, entfernt code, der nicht mehr aufgerufen werden kann
- jump threading, Sprünge auf Sprünge werden entfernt
- bedingte Sprünge auf bedingte Sprünge

- Beispiel

```
  goto a
a goto b
  t1 = 5
b if t2 = 0 goto c
  t2 = 1
c goto d
d t3 = t2 + t1
  if t3 < 10 goto e
  goto e
```

```
b if t2 = 0 goto e
  t2 = 1
  goto e
```

common subexpression elimination (CSE)

- gemeinsame Unterausdrücke werden nur einmal berechnet:

```
t2 = t2 + 2 * t1
t3 = t3 + 2 * t1
```

```
t4 = 2 * t1
t2 = t2 + t4
t3 = t3 + t4
```

- findet bei weitem nicht alle Redundanzen:

```
i = a + 1
r = i - 1
s = r + 1
u = s - 1
```

- kann zu einer starken Beeinträchtigung führen, da der Registerdruck zunimmt (mehr Zwischenergebnisse)

- GCC unterstützt lokale und globale CSE

strength reduction

- Schleifenvariablen = Induktionsvariablen

biv basic induction variable; wird bei jedem Schleifendurchlauf um einen Konstanten Wert erhöht/erniedrigt

giv general induction variable; linear abhängig von einer **biv**

- Ausdrücke, die in einem linearen Zusammenhang mit Schleifenvariablen stehe, werden auf einfachere Ausdrücke zurückgeführt

```
for (i = 0; i < 10; i++)
   sum += i*10+5;
```

wird überführt in

```
t = 0*10+5;
for (i = 0; i < 10; i++)
   sum += t,
   t = t + 10;
```

und später:

```
for (t = 5; t < 100; t += 10)
   sum += t;
```

invariant code motion

- schleifeninvarianter Code wird vor (oder hinter) die Schleife gezogen

```
for (i=0; i<10; i++)
  sum += array [i+10] + array[0];
```

$\Downarrow$

```
sum += array[0] * 10;
t1 = array + 10;
for (i=0; i<10; i++)
  sum += t1[i];
```

partial redundancy elimination (PRE)

- eine „Teilredundanz“ (*partial redundancy*) ist eine Berechnung, die mehr als einmal in einem bestimmten Ausführungspfad auftaucht
- erfordert ursprünglich sehr komplizierte bidirektionale Datenflußanalyse
- entspricht in der Mächtigkeit in etwa CSE + invariant code motion
- Umformulierung von PRE kann neue, mächtige Optimierungen ergeben („you only have to know how to formulate a problem for PRE“)
- die meisten Compiler verwenden **entweder** CSE+ICM **oder** PRE

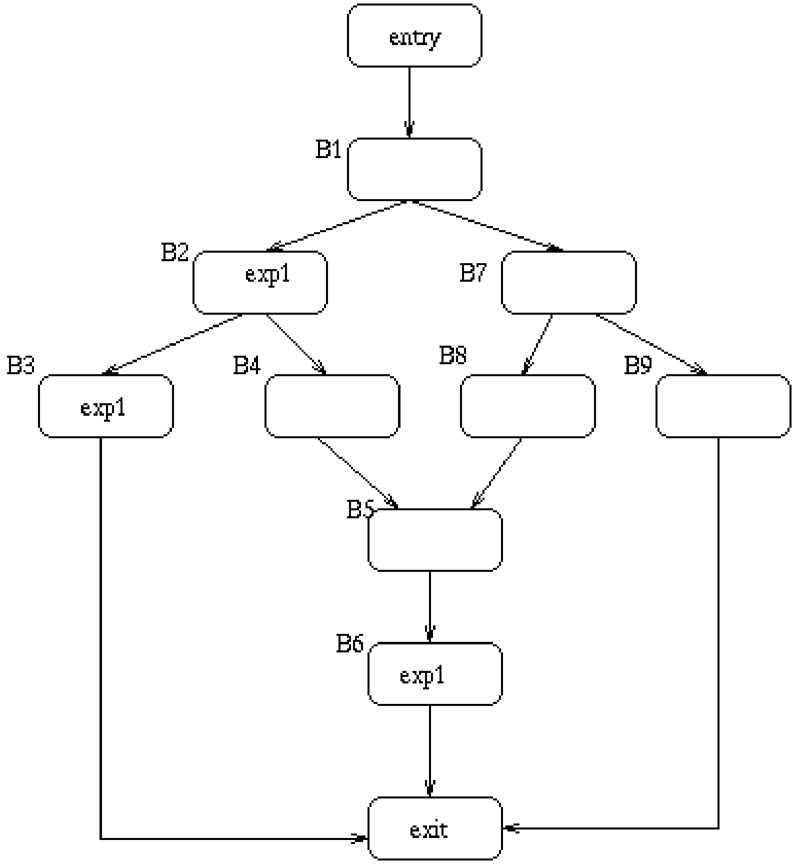
- aber: keine Übermenge zu CSE
- GCC: ICM nicht Teil von PRE

- global constant propagation (gehört nicht zu PRE)

```
i = 2
j = i*2
k = i+2
```

- global copy propagation (gehört nicht zu PRE)

```
i = ...;
for (...) {
   k = i;
   use (k);
}
```


entry
B1
B2
exp1
B7
B3
exp1
B4
B8
B9
B5
B6
exp1
exit

loop unrolling

- Der Schleifenkörper wird vervielfacht („aufgerollt“), der Test auf Ende der Schleife und Sprungbefehle werden eingespart
  - vgl. „Duff's device“

```
sum += 5;
for (t = 15;t < 100; t+=10) {
  sum += t;
  t += 10;
  sum += t;
}
```

instruction combining

- eine Spezialität von GCC
- gut vor allem für CISC-Maschinen und RISC's mit komplizierten Addressierungsmodi
- GCC versucht, jeweils zwei oder drei aufeinanderfolgende Befehle zusammenzufassen (durch Einsetzen und anschließendes Vereinfachen)

```
(insn 14 12 16 (set (reg:SI 26)
                    (symbol_ref:SI ("arr"))))
(insn 16 14 18 (set (reg:SI 27)
                    (mem/s:SI (plus:SI (reg:SI 26)
                                       (reg:SI 25))
               )))

(note 14 12 16 "" NOTE_INSN_DELETED)
(insn 16 14 18 (set (reg:SI 27)
                    (mem/s:SI (plus:SI (reg:SI 25)
                              (symbol_ref:SI ("arr")))
               )))
```

instruction scheduling

- Sehr wichtige Optimierung, wird in GCC zweimal angewendet (vor und nach der Registervergabe, auf x86 nur danach)
- CPU-abhängig (EPIC verspricht Abhilfe)
- Neusortierung der Befehle, um die Funktionsgruppen der CPU auszulasten:
  - multiplier
  - divider
  - shifter
  - memory
  - fpu
  - fpu-division

- Kontroll- und Datenabhängigkeiten zwischen zwei Befehlen

  Echte Abhängigkeit Befehl A errechnet einen Wert der von Befehl B benötigt wird

  Anti-Abhängigkeit Befehl B zerstört ein Registerinhalt der von Befehl A benötigt wird

  Ausgabe-Abhängigkeit Beide Befehle beschreiben dasselbe Register/Speicherwort

- delay slots werden gefüllt (wenn möglich)

- Just in time compilierung (JIT) = Scheiße

```
movl $15,%eax       ;0
addl v1,%eax        ;1
movl %eax,v2        ;2
xorl %ebx,%ebx      ;2
movw v3,%bx         ;3
                    ;4 Zyklen
```

```
movl $15,%eax       ;0
xorl %ebx,%ebx      ;0
addl v1,%eax        ;1
movw v3,%bx         ;1
movl %eax,v2        ;2
                    ;2.5-3 Zyklen
```

local & global register allocation

- einer der wichtigsten Phasen bei der Übersetzung
- GCC bildet alle Zwischenergebnisse auf ein Register ab, und nimmt sich so viele wie nötig („Pseudo-Register", „Pseudos")
- bei der Registervergabe werden die „Pseudos" den physikalisch vorhandenen Register zugewiesen

- Beispiel:

```
t1 = a  + 1
t2 = b  * a
t3 = t1 + t2
```

auf einer Maschine mit zwei Registern (r1 und r2)

```
r1 = a
r1 = r1 + 1
r2 = b
r2 = r2 * a
r1 = r1 + r2
```

- leider einer der eher schlechter entwickelten Teile von GCC, wirkt sich besonders fatal auf Maschinen mit wenig Registern (x86) aus.

reload

- schiebt Zwischenergebnisse zeitweise in den Speicher (in „spill-slots“), wenn nicht genügend Register vorhanden sind

- dient dazu, auch unter ungünstigen Umständen korrekten (aber nicht schnellen) Code zu erzeugen.

- „wenn reload zu tun hat kann man den Code vergessen“

- manche Ports (x86!) überlassen reload die Hauptarbeit. . .

reg-stack

- wandelt gcc-register-code für stackbasierte Maschinen um, wird nur für x86-Fließkomma verwendet

- keine Optimierung („eher das Gegenteil“)

final

- wandelt RTL in Assembler um
- peephole-Optimierungen

Weitergehende Optimierungen

. . . wie sie (noch?) nicht in GCC/EGCS verwendet werden

- Software Pipelining
- SSA based GVN
- I-cache optimization
- D-cache optimization
- Parallelisierung (z.B. SUIF)

- Prädikatisierung

References

[1] Using and Porting GCC, Teil der GCC bzw. EGCS Distributionen (`info -f gcc`)

[2] Advanced Compiler Design and Implementation, Stephen S. Muchnik, 1997 Morgan Kaufmann Publishers, ISBN 1-55860-320-4