

Tema 11. Optimización de Código

Francisco José Ribadas Pena

PROCESADORES DE LENGUAJES

4º Informática

`ribadas@uvigo.es`

18 de mayo de 2011

11.1 Introducción

Objetivo: Mejorar cód. objeto final, preservando significado del programa

Factores a optimizar {
 velocidad de ejecución
 tamaño del programa
 necesidades de memoria

Se sigue una aproximación conservadora

→ No se aplican todas las posibles optimizaciones, solo las “seguras”

Clasificación de las optimizaciones

1. En función de la dependencia de la arquitectura

- Dependientes de la máquina: Aprovechan características específicas de la máquina objetivo
 - asignación de registros, uso de modos de direccionamiento
 - uso instrucciones especiales (IDIOMS)
 - relleno de pipelines, predicción de saltos, aprovechamiento estrategias de mem. caché, etc..
- Independientes de la máquina: Aplicables en cualquier tipo de máquina objetivo
 - ejecución en tiempo de compilación
 - eliminación de redundancias
 - cambios de orden de ejecución, etc..

2. En función del ámbito de aplicación

- Optimizaciones locales: Aplicadas dentro de un Bloque Básico
 - Sólo estudian las instrucciones del B.B. actual
- Optimizaciones globales: Aplicadas a más de un B.B.
 - Consideran contenido y flujo de datos entre todos o parte de los B.B.
 - Necesidad de recoger info. sobre los B.B. y sus interrelaciones
 - Algoritmos de análisis global de flujo de datos

Posibilidades de optimización: {
 sobre código fuente (programador/preprocesador)
 durante generación cod. objeto
 sobre Código Intermedio ⇐

11.2 Optimizaciones Locales

1. Ejecución en tiempo de compilación

- Precalcular expresiones constantes (con constantes o variables cuyo valor no cambia)

$i = 2 + 3$	$\rightarrow$	$i = 5$
$j = 4$	$\rightarrow$	$j = 4$
$f = j + 2.5$	$\rightarrow$	$f = 6.5$

2. Reutilización de expresiones comunes

$a = b + c$	$\rightarrow$	$a = b + c$
$d = a - d$		$d = a - d$
$e = b + c$		$e = a$
$f = a - d$		$f = a - d$

3. Propagación de copias

- Ante instrucciones $f = a$, sustituir todos los usos de f por a

$a = 3 + i$	$\rightarrow$	$a = 3 + i$
$f = a$		$b = a + c$
$b = f + c$		$d = a + m$
$d = a + m$		$m = a + d$
$m = f + d$		

4. Eliminación redundancias en acceso matrices

- Localizar expresiones comunes en cálculo direcciones de matrices

Cod. fuente:

```
A: array [1..4, 1..6, 1..8] of integer
A[i,j,5] := A[i,j,6] + A[i,j,4]
```

Sin optimización:

```
direc(A[i,j,5]) = direc(A[1,1,1]) + (i-1)*6*8 + (j-1)*8 + (5-1)
direc(A[i,j,6]) = direc(A[1,1,1]) + (i-1)*6*8 + (j-1)*8 + (6-1)
direc(A[i,j,4]) = direc(A[1,1,1]) + (i-1)*6*8 + (j-1)*8 + (4-1)
```

Con optimización:

```
k := direc(A[1,1,1]) + (i-1)*6*8 + (j-1)*8 + (5-1)
direc(A[i,j,5]) = k + 4
direc(A[i,j,6]) = k + 5
direc(A[i,j,4]) = k + 3
```

5. Transformaciones algebraicas:

Aplicar propiedades matemáticas para simplificar expresiones

a) Eliminación secuencias nulas

$$x + 0 \rightarrow x$$

$$1 * x \rightarrow x$$

$$x / 1 \rightarrow x$$

...

b) Reducción de potencia

- Reemplazar una operación por otra equivalente menos costosa

$$x^2 \rightarrow x * x$$

$$2*x \rightarrow x + x \text{ (suma); } x \ll 1 \text{ (despl. izq.)}$$

$$4*x, 8*x, \dots \rightarrow x \ll 2, x \ll 3, \dots$$

$$x / 2 \rightarrow x \gg 2$$

...

c) Reacondicionamiento de operadores

- Cambiar orden de evaluación aplicando propiedades conmut., asociativa y distributiva

$$A := B * C * (D + E) \equiv A := (D + E) * B * C$$

MOV B,R0

MUL C,R0

MOV R0, t1

MOV D,R0

ADD E, R0

MUL t1,R0

MOV R0,A

MOV D,R0

ADD E,R0

MUL B,R0

MUL C,R0

MOV R0,A

5 instrucciones

0 temporales

11.2.1 Optimización de tipo Mirilla (*Peephole optimization*)

Aplicable en código intermedio o código objeto.

Constituye una nueva fase aislada.

Idea Básica

- Se recorre el código buscando combinaciones de instrucciones que puedan ser reemplazadas por otras equivalentes más eficientes.
- Se utiliza una ventana de n instrucciones y un conjunto de patrones de transformación (**patrón, secuencias reemplazam.**)
- Si las instrucciones de la ventana encajan con algún patrón se reemplazan por la secuencia de reemplazamiento asociada.
- Las nuevas instrucciones son reconsideradas para las futuras optimizaciones

Ejemplos:

- Eliminación de cargas innecesarias

```
MOV Ri, X      →  MOV Ri, Rj
MOV X, Rj
```

- Reducción de potencia
- Eliminación de cadenas de saltos

```
      if C goto L1
L1:    goto L2      →      if C goto L2
```

```
      if C goto L1
      goto FIN
L1:    ...           →      L1:    if not C goto FIN
      ...           ...
FIN:   ...           FIN:    ...
```

11.3 Optimizaciones Globales

Optimizaciones entre Bloques Básicos

Optimizaciones típicas:

- Identificación de expresiones comunes entre bloques
→ afecta a la asignación de registros entre B.B.
- Optimización de llamadas a procedimientos
- Optimización de bucles

11.3.1 Optimización Llamadas a Procedimientos

Llamadas a procedimientos son muy costosas

- cambios del ámbito de referencias
- gestión de Registros de Activación
- paso de parámetros + asignación de datos locales

Mejoras

1. Optimizar manejo de Reg. Activación
 - minimizar copia de parámetros y n° registros a salvar
 - uso almacenamiento estático si no hay llamadas recursivas
2. Expansión en línea.

Expansión en línea

Idea: Eliminar llamadas a proc., “copiando” el cuerpo del proc. en el lugar donde es llamado

- Evita sobrecarga asociada con llamadas a proc.
- Permite optimizaciones adicionales
 - el código llamado pasa a ser parte del código que lo llama

Limitaciones

- Aumenta uso de memoria $\Rightarrow$ útil en proc. pequeños y llamados desde pocos lugares
 - si se llama en un único lugar (es frecuente), no supone coste de espacio adicional
- No aplicable en proc. recursivos

Ejemplos

- Directiva `inline` en C++ (en Java el compilador hace la expansión de forma automática)
- En C puede simularse con macros `#define`

11.3.2 Optimización de Bucles

Idea: Centrar optimización en partes más usadas, no en todo el programa

→ Optimizar bucles internos

Mejoras $\left\{ \begin{array}{l} \text{factorización de expresiones invariantes} \\ \text{reducción de intensidad y eliminación de variables de inducción} \end{array} \right.$

11.3.2.1 Factorización de expresiones invariantes

Expr. invariantes de bucle: expr. cuyo valor es constante durante toda la ejecución del bucle

→ incluyen constantes y/o variables no modificadas en el cuerpo del bucle

Idea: Mover expr. invariantes desde el cuerpo hasta la cabeza del bucle

→ al sacarlas del bucle, pueden quedar dentro de otro bucle externo $\Rightarrow$ repetir proceso

Ejemplos:

<pre>while (i <= (limite*2 - 1)) do { ... }</pre>	→	<pre>tmp1 = limite*2 - 1; while (i <= tmp1) { ... }</pre>
--	---	--

A: array [1..900, 1..900, 1..900] of integer

<pre>for i:=1 to 900 do for j:=1 to 900 do for k:=1 to 900 do A[i][j][k] := i*j*k; end for end for end for</pre>	<pre>for i:=1 to 900 do for j:=1 to 900 do tmp1:= i*j; tmp2:= dir(A[i][j]); for k:= 1 to 900 do tmp2[k] := tmp1*k; end for end for end for</pre>	<pre>for i:= 1 to 900 do tmp3:= dir(A[i]); for j:= 1 to 900 do tmp1:= i*j; tmp2:= dir(tmp3[j]); for k:= 1 to 900 do tmp2[k] := tmp1*k; end for end for end for</pre>
--	--	--

i, j y $A[i][j]$ son constantes en el bucle de k	$A[i]$ es constante en el bucle de j
---	---

$\Rightarrow$ En C.I. aparecerán mas invariantes al incluir el cálculo de desplazamientos en arrays basados en el tamaño de los datos (int 4, float 8,...)

11.3.2.2 Reducción potencia y elimin. variables de inducción

Variable de inducción (en un bucle): variable cuyo valor sistemáticamente se incrementa/decrementa en un valor constante en cada iteración

→ Ejemplo: índices en bucles **for**, contadores en bucles **while**

Var. de inducción básica: su única definición en el cuerpo del bucle es de la forma $i = i + s$

Caracterizada por: $\begin{cases} i_0: \text{valor inicial (antes del bucle)} \\ s: \text{salto del bucle, constante o expr. invariante} \end{cases}$

Idea: Si hay varias variables de inducción $\Rightarrow$ todas están ligadas:

- se puede aplicar reducción de potencia sobre expresiones con esas vars.

→ sustituyendo multiplicaciones/divisiones por sumas/restas

- se pueden sustituir todas por una única var. de inducción

Expresión de Inducción: define una variable de inducción ligada, k .

$$k = i * c + d \text{ con } \begin{cases} i: \text{var. de inducción} \\ c, d: \text{constantes o expr. invariantes} \end{cases}$$

- i toma los valores: $i_0, i_0 + s, i_0 + 2s, \dots$

- la expr. de inducción tomará:

$$i_0 * c + d, (i_0 * c + d) + s * c, (i_0 * c + d) + 2(s * c), \dots$$

→ se incrementa $s * c$ en cada iteración

NOTA: accesos a array basados en vars. de inducción, provocan aparición de vars. de inducción ligadas al generar C.I.

1. REDUCCIÓN INTENSIDAD EN VARS. DE INDUCCIÓN

- Cada expr. de inducción $k = i * c + d$ que defina **k** se reemplazará por un temporal t_i
 - inicializado a $\underline{i_0 * c + d}$, antes de bucle
 - incrementado en un valor $\underline{s * c}$, al final del cuerpo del bucle
- Asignar a **k** el valor de t_i
- NOTA: Se podrá aplicar propagación de copias sobre **k** y t_i
Si **k** no se usa fuera del bucle se podrá eliminar, dejando las referencias equivalentes a t_i

- Ejemplo:

<pre> i := 1 while i <= 1000 do { ... a := 4*i + 3; b := 2*i; ... i := i + 1; } </pre>	<pre> i := 1; t1 := 7; t2 := 2; while i <= 1000 do { ... a := t1; b := t2; ... i := i + 1; t1 := t1 + 4; t2 := t2 + 2; } </pre>
---	--

i : var. induc. básica ($i_0 = 1, s = 1$)

a : expr. induc. ($c = 4, d = 3$)

b : expr. induc. ($c = 2, d = 0$)

$t1$ sustitute a a	$\left\{ \begin{array}{l} \text{inicial} = 1*4 + 3 \\ \text{salto} = 1*4 \end{array} \right.$
$t2$ sustitute a b	$\left\{ \begin{array}{l} \text{inicial} = 1*2 + 0 \\ \text{salto} = 1*2 \end{array} \right.$

→ Se sustituyen multiplicaciones por sumas

2. ELIMINACIÓN DE VARS. DE INDUCCIÓN

- Se pueden eliminar todas las vars. de inducción ligadas del bucle, dejando sólo una de ellas.
- Se aplicará sobre toda var. de inducción **i** usada únicamente en el cálculo de otra var. de inducción **j** (calculada como $j := i * c + d$) o en saltos condicionales.
- Nos quedaremos con **j** y eliminaremos **i**
 - a) modificar comparaciones sobre **i** para usar equivalentes sobre **j**
 $\text{if } i \text{ OP_REL } X \text{ goto } \dots \rightarrow \text{if } j \text{ OP_REL } X*c + d \text{ goto } \dots$
 $\text{OP_REL} = \{<, >, =, \dots\}$
 - b) Eliminar toda instrucción que incluya a **i**
- Ejemplo:

COD. FUENTE

```
A: array 0..100 of integer
...
suma := 0;
for i:=50 to 100 do
    suma := suma + A[i];
end for;
```

COD. INTERMEDIO

```
(101)  i = 50
(102)  t1 = i*4
(103)  t2 = A[t1]
(104)  suma = suma + t2
(105)  i = i + 1
(106)  if i <= 100 goto (102)
```

t1 var. ind. derivada de i

RED. POTENCIA

```
(101)  i = 50 (b)
(102)  t1 = 200
(103)  t2 = A[t1]
(104)  suma = suma + t2
(105)  i = i + 1 (b)
(106)  t1 = t1 + 4
(107)  if i <= 100 goto (103) (a)
```

(a) comparaciones a modificar

(b) expr. con *i* a eliminar

ELIMIN. VAR IND. i

```
(101)  t1 = 200
(102)  t2 = A[t1]
(103)  suma = suma + t2
(104)  t1 = t1 + 4
(105)  if t1 <= 400 goto (102)
```

t1 sustituye a i