TÉCNICO LISBOA

Compiladores[1]

Optimização de Código

Capítulos 8.4, 8.5 e 9 - "Compilers: Principles, Techniques and Tools"

Prof. Alberto Abad

IST - Universidade de Lisboa

2021/2022

[1] Slides adaptados de Prof. Pedro T. Monteiro (2017/2018)

Outline

Optimização de código

Geração de código intermédio é executada:

- Por visita da AST
- Cada nó gera código independentemente

Optimização de código

Geração de código intermédio é executada:

- Por visita da AST
- Cada nó gera código independentemente

Problemas potenciais:

- Instruções desnecessárias
- Código que consome demasiada memória
- Código que consome demasiado CPU
- Conjunto de instruções não optimizadas

Optimização de código

Objectivo:

- Melhorar o desempenho do código
- Preservando a semântica do programa original

Outline

Grafos de fluxo

Uma melhor optimização de código pode ser alcançada se forem conhecidas as dependências entre blocos de código

Grafos de fluxo

Uma melhor optimização de código pode ser alcançada se forem conhecidas as dependências entre blocos de código

Bloco básico é uma sequência de instruções:

- sem saltos, excepto na última instrução
- sem ser alvo de salto excepto na primeira instrução

Grafos de fluxo

Exemplo – Código C

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Grafos de fluxo

Exemplo – Código C3E

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```

Grafos de fluxo

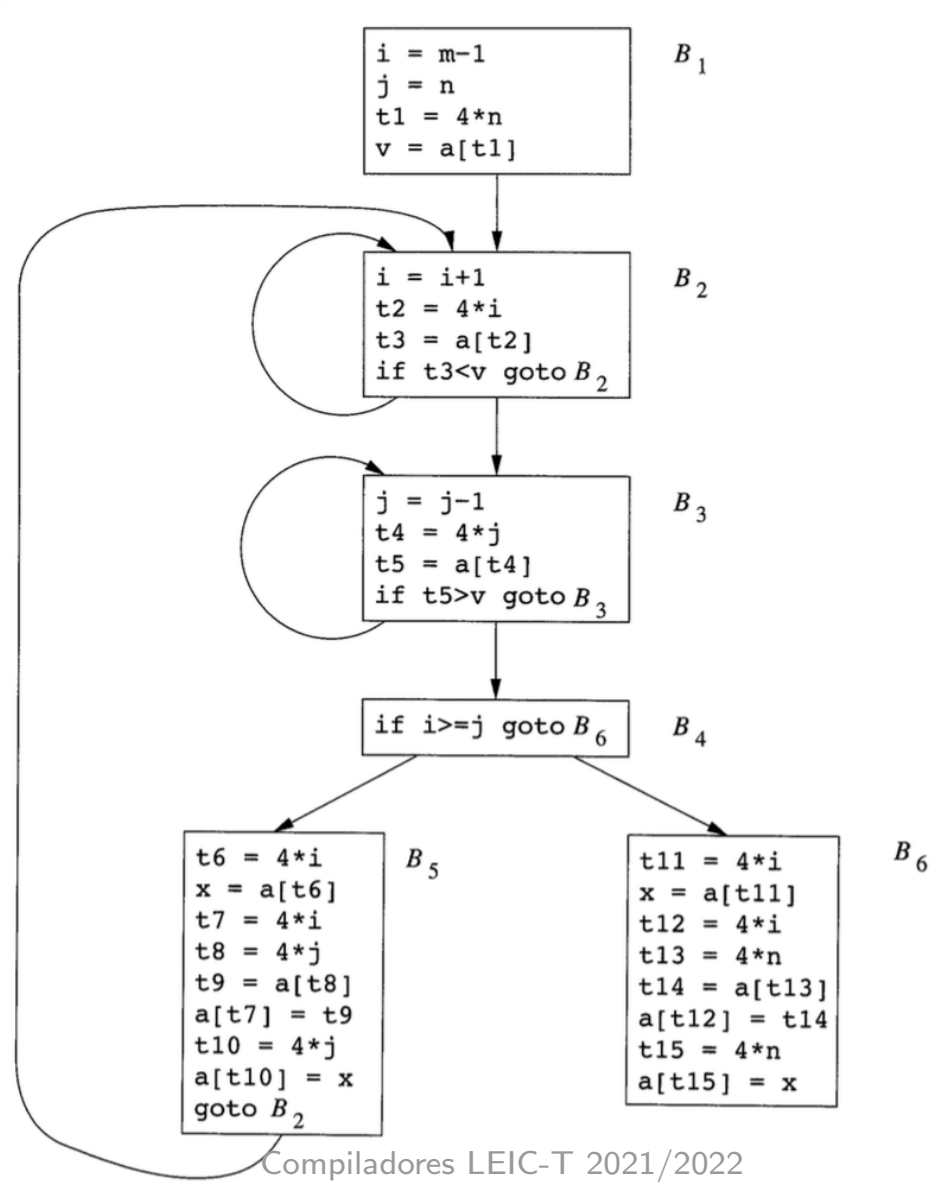
Exemplo – Código C3E

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```

- **Q**: Existe redundância **??**
- **Q**: Que simplificações fazer **??**

Grafos de fluxo

Exemplo – Blocos básicos



Tipos de optimizações

As optimizações podem ser:

- **Dependentes** da máquina
- **Independentes** da máquina

Tipos de optimizações

As optimizações podem ser:

- **Dependentes** da máquina
- **Independentes** da máquina

... e a vários níveis:

- **User** - nível do algoritmo
- **Generic** - nível do código intermédio
- **Peephole** - janela sobre conjunto de instruções
- **Local** - dentro de um bloco básico
- **Inter-block** - fluxo de informação entre blocos
- **Global** - saltos para saltos

Outline

Optimizações dependentes da máquina

Optimizações dependentes da máquina, têm em conta:

- Número de registos
- Endereços reais de memória

Optimizações dependentes da máquina

Spilling de registos:

- Registos são a unidade de memória mais rápida na máquina
- Mas... têm número limitado

Optimizações dependentes da máquina

Spilling de registos:

- Registos são a unidade de memória mais rápida na máquina
- Mas... têm número limitado

Sempre que um registo adicional é necessário, o conteúdo de um dos registos tem de ir para memória (spilled)

Optimizações dependentes da máquina

Spilling de registos:

- Registos são a unidade de memória mais rápida na máquina
- Mas... têm número limitado

Sempre que um registo adicional é necessário, o conteúdo de um dos registos tem de ir para memória (spilled)

- Instruções podem ser reordenadas
- Blocos básicos como indicação da validade das variáveis

Selecção de instruções

Operações podem ser realizadas com diferentes combinações de instruções

- Processadores com muitas instruções (*e.g.* *CISC*) é útil
- Processadores com poucas instruções (*e.g.* *RISC*) não tanto

Selecção de instruções

Operações podem ser realizadas com diferentes combinações de instruções

- Processadores com muitas instruções (*e.g. CISC*) é útil
- Processadores com poucas instruções (*e.g. RISC*) não tanto

Exemplos:

- Simplificação algebraica:
  `mov 0,eax` → `xor eax,eax`
- Redução de força:
  `i*8` → `i<<3`

Outline

Optimizações independentes da máquina

Transformações que preservam a semântica:

- Loop unrolling
- Simplificação de expressões
- Simplificação de constantes
- Eliminação de subexpressões comuns
- Propagação de cópias
- Eliminação de código morto
- Deslocação de código
- Variáveis de indução
- Redução de força

Loop unrolling:
Redução do número de iterações replicando o corpo do ciclo

Código:

```
for (i = 0; i < 100; i++) {
        f();
}
```

Loop unrolling:
Redução do número de iterações replicando o corpo do ciclo

Código:

```
for (i = 0; i < 100; i++) {
        f();
}
```

Substituído por:

```
for (i = 0; i < 100; i+=2) {
        f();
        f();
}
```

Loop unrolling:
Redução do número de iterações replicando o corpo do ciclo

Código:

```
for (i = 0; i < 100; i++) {
        f();
}
```

Substituído por:

```
for (i = 0; i < 100; i+=2) {
        f();
        f();
}
```

Código:

```
for (i = 0; i < n; i++)
    a[i] = b[i] + 10;
```

Substituído por:

```
max = n - n%4
for (i = 0; i < max; i+=4) {
  a[i]   = b[i]   + 10;
  a[i+1] = b[i+1] + 10;
  a[i+2] = b[i+2] + 10;
  a[i+3] = b[i+3] + 10;
}
for (; i < n; i++)
  a[i] = b[i] + 10;
```

Simplificação de expressões:
Substituição de expressões por equivalentes mais eficientes

Simplificação de expressões:
Substituição de expressões por equivalentes mais eficientes

Código:

```
a[0] = i + 0;
a[1] = i * 0;
a[2] = i - i;
```

Simplificação de expressões:
Substituição de expressões por equivalentes mais eficientes

Código:

```
a[0] = i + 0;
a[1] = i * 0;
a[2] = i - i;
```

Substituído por:

```
a[0] = i;
a[1] = 0;
a[2] = 0;
```

Optimizações independentes da máquina

Simplificação de constantes:
Avaliação de expressões contendo apenas literais e valores conhecidos

Simplificação de constantes:
Avaliação de expressões contendo apenas literais e valores conhecidos

Código:

```
int *x = (int *)malloc(1024 * 4);
int  y = 3;
int  z = 20 * y + 10;
```

Simplificação de constantes:
Avaliação de expressões contendo apenas literais e valores conhecidos

Código:

```
int *x = (int *)malloc(1024 * 4);
int  y = 3;
int  z = 20 * y + 10;
```

Substituído por:

```
int *x = (int *)malloc(4096);
int  y = 3;
int  z = 70;
```

Eliminação de subexpressões comuns

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

`t7` e `t10` podem ser eliminados e substituídas por `t6` e `t8`, respectivamente

Optimizações independentes da máquina

Propagação de cópias:
Substituição do valor de atribuições directas

Optimizações independentes da máquina

Propagação de cópias:
Substituição do valor de atribuições directas

Código:

```
x = z;
y = x + 1;
```

Propagação de cópias:
Substituição do valor de atribuições directas

Código:

```
x = z;
y = x + 1;
```

Substituído por:

```
y = z + 1;
```

Optimizações independentes da máquina

Eliminação de código morto:
Eliminação de código que nunca é usado (*e.g.* código após um return)

Eliminação de código morto:
Eliminação de código que nunca é usado (*e.g.* código após um return)

Código:

```
debug = false;
if (debug) {
        print ...
}
```

Optimizações independentes da máquina

Deslocação de código (código invariante num ciclo):
Deslocar para fora do ciclo código cujo resultado é invariante num ciclo

Deslocação de código (código invariante num ciclo):
Deslocar para fora do ciclo código cujo resultado é invariante num ciclo

Código:

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

Deslocação de código (código invariante num ciclo):
Deslocar para fora do ciclo código cujo resultado é invariante num ciclo

Código:

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

Substituído por:

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
  a[i] = 6 * i + t1;
}
```

Optimizações independentes da máquina

Variáveis de indução:
Variáveis cujo valor aumenta/diminui de uma constante c de cada vez que é actualizada

Variáveis de indução:
Variáveis cujo valor aumenta/diminui de uma constante c de cada vez que é actualizada

Código:

```
for (i = 1; i < 5; i++) {
  j = 5 * i + 4;
  // more code
}
```

Variáveis de indução:
Variáveis cujo valor aumenta/diminui de uma constante c de cada vez que é actualizada

Código:

```
for (i = 1; i < 5; i++) {
  j = 5 * i + 4;
  // more code
}
```

Substituído por:

```
j = 4;
for (i = 1; i < 5; i++) {
  j = j + 5;
  // more code
}
```

Optimizações independentes da máquina

Redução de força:
Instruções pesadas são substituídas por instruções menos pesadas

Redução de força:
Instruções pesadas são substituídas por instruções menos pesadas

Código:

```
x = y * 8;
x = y * 15;
x = y / 4;
```

Redução de força:
Instruções pesadas são substituídas por instruções menos pesadas

Código:

```
x = y * 8;
x = y * 15;
x = y / 4;
```

Substituído por:

```
x = y << 3;
x = y << 4 - y;
x = y >> 2;
```

Exercício

```
int *fun(int *a, int *b, int len) {
  int *c = (int *)malloc(2 * 4);
  int i;
  for (i = 0; i < (len < 2 ? len : 2); i++) {
    c[i] = a[i] + b[i];
  }
  return c;
}
```

Exercício

```
int *fun(int *a, int *b, int len) {
  int *c = (int *)malloc(2 * 4);
  int i;
  for (i = 0; i < (len < 2 ? len : 2); i++) {
    c[i] = a[i] + b[i];
  }
  return c;
}
```

Q: Que optimizações independentes da máquina é que são possíveis?

Exercício

```
int *fun(int *a, int *b, int len) {
  int *c = (int *)malloc(2 * 4);
  int i;
  for (i = 0; i < (len < 2 ? len : 2); i++) {
    c[i] = a[i] + b[i];
  }
  return c;
}
```

Q: Que optimizações independentes da máquina é que são possíveis?

- Simplificação de constantes `2 * 4`
- Loop unrolling
- Deslocamento de código `len < 2 ? len : 2`

Exercício

```
int *click(int *x, int dim) {
  int *res, i;
  for (i = dim-2, res = x+dim-1; i >= 0; i--)
    if (x[i] > *res) res = &x[i];
  return res;
}
```

Exercício

```
int *click(int *x, int dim) {
  int *res, i;
  for (i = dim-2, res = x+dim-1; i >= 0; i--)
    if (x[i] > *res) res = &x[i];
  return res;
}
```

```
TEXT
ALIGN
LABEL click
ENTER 8
; x@8 dim@12
; fp@0
; res@-4 i@-8

LOCAL 12
LDINT
INT 2
SUB
DUP32
LOCAL -8
STINT
TRASH 4
```

```
LOCAL 8
LDINT
LOCAL 12
LDINT
INT 4
MUL
ADD
INT 1
INT 4
MUL
SUB
DUP32
LOCAL -4
STINT
TRASH 4
```

```
ALIGN
LABEL forcond
LOCAL -8
LDINT
INT 0
GE
JZ forend

LOCAL +8
LDINT
LOCAL -8
LDINT
INT 4
MUL
ADD
LDINT ; x[i] = *(x+i)
LOCAL -4
LDINT
LDINT  ; *res
GT
JZ ifend
```

```
LOCAL 8
LDINT
LOCAL -8
LDINT
INT 4
MUL
ADD ; x+i=&x[i]
DUP32
LOCAL -4
STINT
TRASH 4

ALIGN
LABEL ifend
```

```
ALIGN
LABEL forincr
LOCAL -8
LDINT
DUP32
INT 1
SUB
LOCAL -8
STINT
TRASH 4
JMP forcond

ALIGN
LABEL forend

LOCAL -4
LDINT
STFVAL32
LEAVE
RET
```

Exercício

```
int *click(int *x, int dim) {
  int *res, i;
  for (i = dim-2, res = x+dim-1; i >= 0; i--)
    if (x[i] > *res) res = &x[i];
  return res;
}
```

Exercício

```
int *click(int *x, int dim) {
  int *res, i;
  for (i = dim-2, res = x+dim-1; i >= 0; i--)
    if (x[i] > *res) res = &x[i];
  return res;
}
```

```
;; BB1 start
TEXT
ALIGN
LABEL click
ENTER 8
; x@8 dim@12
; fp@0
; res@-4 i@-8

LOCAL 12
LDINT
INT 2
SUB
DUP32
LOCAL -8
STINT
TRASH 4
```

```
LOCAL 8
LDINT
LOCAL 12
LDINT
INT 4
MUL
ADD
INT 1
INT 4
MUL
SUB
DUP32
LOCAL -4
STINT
TRASH 4
;; BB1 end
```

```
;; BB2 start
ALIGN
LABEL forcond
LOCAL -8
LDINT
INT 0
GE
JZ forend
;; BB2 end
;; BB3 start
LOCAL +8
LDINT
LOCAL -8
LDINT
INT 4
MUL
ADD
LDINT ; x[i] = *(x+i)
LOCAL -4
LDINT
LDINT  ; *res
GT
JZ ifend
;; BB3 end
```

```
;; BB4 start
LOCAL 8
LDINT
LOCAL -8
LDINT
INT 4
MUL
ADD ; x+i=&x[i]
DUP32
LOCAL -4
STINT
TRASH 4
;; BB4 end
;; BB5 start
ALIGN
LABEL ifend
```

```
;; BB5 end
;; BB6 start
ALIGN
LABEL forincr
LOCAL -8
LDINT
DUP32
INT 1
SUB
LOCAL -8
STINT
TRASH 4
JMP forcond
;; BB6 end
;; BB7 start
ALIGN
LABEL forend

LOCAL -4
LDINT
STFVAL32
LEAVE
RET
;; BB7 end
```

Questões?

Dúvidas?