MO615B - Implementação de Linguagens II
e
MC900A - Tópicos Especiais em Linguagem de Programação

Prof. Sandro Rigo

www.ic.unicamp.br/~sandro

Transformações em Laços

Loop Unrolling

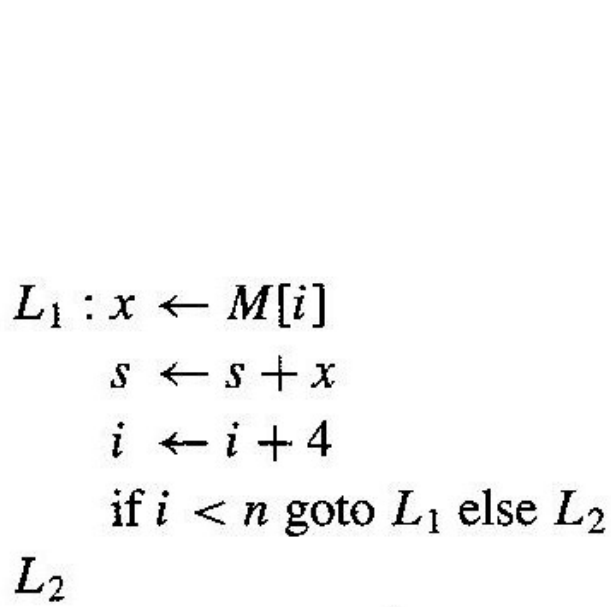
- Laços podem ter corpos muito pequenos
- Maior parte do tempo gasto em:
  - Incremento de variável de indução
  - Teste da condição de saída
- Como podemos aumentar a eficiência?
  - Desenrolando
    - Colocar duas ou mais cópias do corpo em seqüência

Loop Unrolling

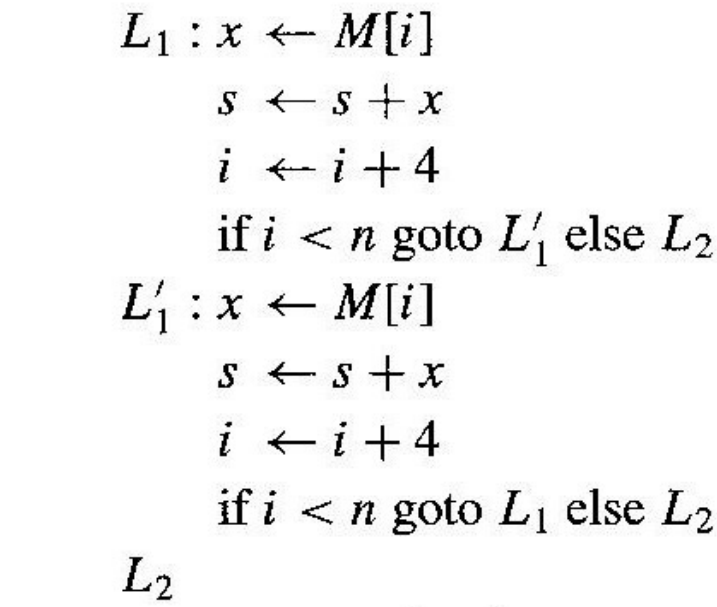
- Dados:
  - Laço L
  - *Back-edges* si → h
- Desenrolar:
  1. Copie os nós fazendo
     1. Laço L’ com *header* h’ com arestas si’ → h’
  2. Mude as *back-edges* em L para:
     1. si → h’
  3. Mude as *back-edges* em L’ para:
     1. si’ → h

Exemplo

- Melhorou??



(a) Before



(b) After

Fonte: Appel fig 18.10

Loop Unrolling

- Melhorou???
  - Não!!!
    - Cada iteração ainda faz incremento e teste da condição
- O que podemos fazer?
  - Queremos juntar os incrementos e os testes
- Como isso pode ser feito?
  - Usando informações de variáveis de indução

Loop Unrolling

- Devemos ter uma variável de indução i tal que:
  - Todo incremento i = i + c (c constante) domina toda back-edge do laço
- O que concluimos agora?
  - Toda iteração incrementa i exatamente pela soma dos c's
- Agora podemos juntar os incrementos e testes:

Loop Unrolling

- Melhorou?? Funciona ???

$$
\begin{aligned}
L_1 : x &\leftarrow M[i] \\
s &\leftarrow s + x \\
x &\leftarrow M[i+4] \\
s &\leftarrow s + x \\
i &\leftarrow i + 8 \\
&\text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2 &
\end{aligned}
$$

(a) Fragile

Fonte: Appel fig 18.11

Loop Unrolling

- Ajustado para qualquer # de iterações

```
        if i < n − 8 goto L1 else L2
L1 :    x ← M[i]
        s ← s + x
        x ← M[i + 4]
        s ← s + x
        i ← i + 8
        if i < n − 8 goto L1 else L2
L2      x ← M[i]
        s ← s + x
        i ← i + 4
        if i < n goto L2 else L3
L3
```

(b) Robust

Fonte: Appel fig 18.11

Loop Unrolling

- Caso Geral:
  - Laço desenrolado por um fator de K
- Epílogo
  - É um laço como o original
  - Itera por até K-1 vezes

Re-ordenação de Sentenças

- Pode ser feita em várias granularidades
  - Operação
  - Sentença
  - Seqüência de sentenças
  - Etc
- Consideramos re-ordenação em um CFG acíclico
  - Sentenças
  - Laços
    - Um nó é uma sentença ou um laço todo

Re-ordenação de Sentenças

- CFG acíclico => Grafo dependência acíclico
- Ordenação legal das sentenças/laços
  - Qualquer ordenação topológica do grafo de dependência
- Utilidade
  - Escalonamento de instruções
  - Amortizar latências: *pipeline* e memória
  - Melhorar localidade de dados
  - Colocar laços separados próximos no programa
    - Gera oportunidade para *loop fusion*

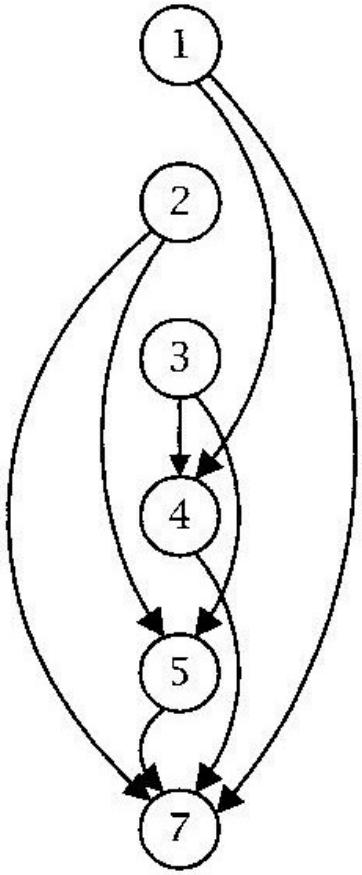
Re-ordenação de Sentenças

- Grafo de Dependência
  - Deve incluir tanto dependências de controle como de dados
- Exemplo: Colocar os acessos ao mesmo vetor próximos.

```
1. A(1) = 0
2. B(1) = 0
3. If C > 0 then
4.  A(2) = 1
5.  B(2)= 9
6. endif
7. For I = 3 to 9 do
8.  A(I) = A(I-2) + A(I-1) * 2
9.  B(I) = B(I-2)*2 + B(I-1)
10. endfor
```

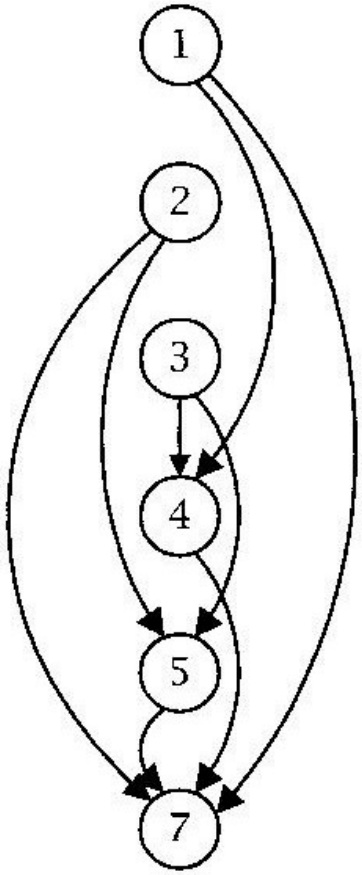
Exemplo

```
1. A(1) = 0
2. B(1) = 0
3. If C > 0 then
4. A(2) = 1
5. B(2)= 9
6. endif
7. For I = 3 to 9 do
8. A(I) = A(I-2) + A(I-1) * 2
9. B(I) = B(I-2)*2 + B(I-1)
10.endfor
```



Código Re-ordenado

```
2. B(1) = 0
3. Test = C > 0
5. if Test then B(2)= 9
1. A(1) = 0
4. if Test then A(2) = 1
7. For I = 3 to 9 do
8. A(I) = A(I-2) + A(I-1) * 2
9. B(I) = B(I-2)*2 + B(I-1)
10.endfor
```



Unswitching

- Trabalha em laços contendo condições
- Remover uma condição independente do laço para fora dele
  - Transforma o laço em um ou dois laços envoltos por uma condição
- É sempre válido
- Diminui a freqüência com que a condição é executada
- Torna a estrutura do laço mais complexa

Unswitching

- Um laço externo contendo um laço interno pode se transformar em um laço contendo mais de um laço interno
- Isto pode coibir a aplicação de outras transformações no laço

cond é independente do laço

```
1. loop
2.     statements
3.     if cond then
4.         then part
5.     else
6.         else part
7.     endif
8.     more statements
9. endloop
```

cond é independente do laço

```
3.if cond then
1.    loop
2.       statements
4.        then part
8.        more statements
9.    endloop
5.else
1.   loop
2.      statements
6.       else part
8.      more statements
9.   endloop
7.endif
```

Exemplo

```
(1)   for I = 1 to N do
(2)       for J = 2 to N do
(3)           if T[I] > 0 then
(4)               A[I,J] = A[I,J-1]*T[I] + B[J]
(5)           else
(6)               A[I,J] = 0.0
(7)           endif
(8)       endfor
(9)   endfor
```

Solução

```
(1)   for I = 1 to N do
(3)      if T[I] > 0 then
(2)         for J = 2 to N do
(4)            A[I,J] = A[I,J-1]*T[I] + B[J]
(8)         endfor
(5)      else
(2)         for J = 2 to N do
(6)            A[I,J] = 0.0
(8)         endfor
(7)      endif
(9)   endfor
```

Loop Peeling

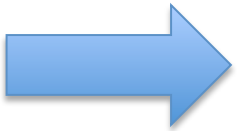
- Remove a última ou a primeira iteração do laço.
- Pode ser aplicada múltiplas vezes para remover diversas iterações.
- Pode ser usada para ajustar o espaço de iteração (*trip count*) do laço.

Loop Peeling

zero trip

count test

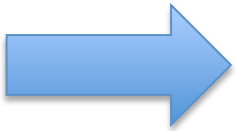
```
compute tc
for i = 0 to tc-1 do
  body
endfor
```



```
compute tc
if tc > 0 then
  i=0
  body
  for i = 1 to tc-1 do
    body
  endfor
endif
```

Loop Peeling

```
for i = 1 to N do
  A[i]= (X+Y) *B[i]
endfor
```



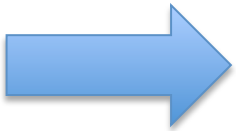
```
if N >= 1 then
  Z = X+Y
  A[1]= Z * B[1]
  for i = 2 to N do
     A[i]= Z*B[i]
  endfor
endif
```

Loop Splitting

- Divide o conjunto de índices de um laço
- Cria dois laços
- Replica o corpo
- Utilidade
  - Ajuste do *trip count*
  - Eliminação de condições sobre a variável de indução
- Também é chamado de *Index Set Splitting*

Loop Splitting

```
compute tc
for i = 0 to tc-1 do
  body
endfor
```



```
compute tc
for i = 0 to s-1 do
  body
endfor
for i = s to s-1 do
  body
endfor
```

Loop Splitting - Exemplo

```
for I = 1 to 100 do
  A[I] = B[I] + C[I]
  if I > 10 then
    D[I] = A[I] + A[I-10]
  endif
endfor
```

Loop Splitting - Exemplo

```
for I = 1 to 100 do
  A[I] = B[I] + C[I]
  if I > 10 then
    D[I] = A[I] + A[I-10]
  endif
endfor
```

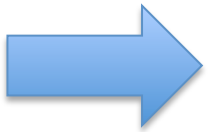
```
for I = 1 to 10 do
  A[I] = B[I] + C[I]
endfor
for I = 11 to 100 do
  A[I] = B[I] + C[I]
  D[I] = A[I] + A[I-10]
endfor
```

Loop Fusion

- Unir dois laços adjacentes
  - Devem ter o mesmo limite de iterações

```
for I = ... do
  body1
endfor

for I = ... do
  body2
endfor
```



```
for I = ... do
  body1
  body2
endfor
```

Loop Fusion

- Unir dois laços adjacentes
  - Devem ter o mesmo limite de iterações
- Originalmente
  - Reduzir o custo com os testes e desvios do laço
  - Era restrita a laços completamente independentes em termos de dados
  - Recente avanço na teoria de dependência melhorou isso

Loop Fusion

- Utilidade:
  - Localidade de memória
    - Fusão de laços que acessam os mesmos dados
      - Melhora a localidade temporal
      - Impacto positivo na *cache*
  - Aumenta o corpo do laço
    - Pode trazer novas oportunidades para outras otimizações
      - Escalonamento de instruções
      - CSE

Loop Fusion

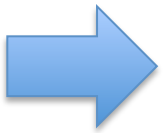
- Desvantagem:
  - O aumento no tamanho do laço pode piorar o desempenho na *cache* de instruções
    - Quando a *cache* é muito pequena; ou
    - Quando o corpo do laço é muito grande

Exemplo - *Loop Fusion*

```
for I = 1 to N do
  A[I] = B[I] + 1
endfor
for I = 1 to N do
  C[I] = A[I] / 2
endfor
for I = 1 to N do
  D[I] = 1 / C[I+1]
endfor
```

Exemplo - *Loop Fusion*

```
for I = 1 to N do
  A[I] = B[I] + 1
endfor
for I = 1 to N do
  C[I] = A[I] / 2
endfor
for I = 1 to N do
  D[I] = 1 / C[I+1]
endfor
```

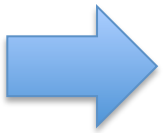


```
for I = 1 to N do
  A[I] = B[I] + 1
  C[I] = A[I] / 2
  D[I] = 1 / C[I+1]
endfor
```

Funciona?

Exemplo - *Loop Fusion*

```
for I = 1 to N do
  A[I] = B[I] + 1
endfor
for I = 1 to N do
  C[I] = A[I] / 2
endfor
for I = 1 to N do
  D[I] = 1 / C[I+1]
endfor
```



```
for I = 1 to N do
  A[I] = B[I] + 1
  C[I] = A[I] / 2
endfor
for I = 1 to N do
  D[I] = 1 / C[I+1]
endfor
```

Possíveis Complicações

- *Loop fusion* só é aplicável para laços contáveis e com o mesmo *trip count*
  - As v.i. não precisam ter o mesmo nome
- E se dois laços adjacentes possuem *trip count* diferente?
  - Se *loop fusion* ainda assim for desejável
    - Posso dividir o laço maior para ajustar o *trip-count (loop splitting)*
    - Posso usar condicionais para evitar que sentenças sejam executadas

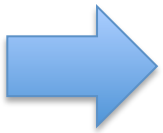
Exemplo - *Loop Fusion*

```
for I = 1 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 98 do
  C[I] = A[I+1]*2
endfor
```

- O primeiro laço tem uma iteração a mais.

Exemplo - *Loop Fusion*

```
for I = 1 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 98 do
  C[I] = A[I+1]*2
endfor
```

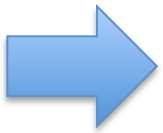


```
A[1] = B[1] + 1
for I = 2 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 98 do
  C[I] = A[I+1]*2
endfor
```

- Podemos aplicar *Loop Peeling* para ajustar o *trip count* do primeiro laço.

Exemplo - *Loop Fusion*

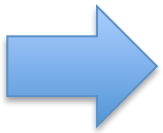
```
A[1] = B[1] + 1
for I = 2 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 98 do
  C[I] = A[I+1]*2
endfor
```



```
A[1] = B[1] + 1
for ib = 0 to 97 do
  I = ib+2
  A[I] = B[I] + 1
  I = ib + 1
  C[I] = A[I+1]*2
endfor
```

Exemplo - *Loop Fusion*

```
A[1] = B[1] + 1
for ib = 0 to 97 do
  I = ib + 2
  A[I] = B[I] + 1
  I = ib + 1
  C[I] = A[I+1]*2
endfor
```



```
A[1] = B[1] + 1
for I = 1 to 98 do
  s = I + 1
  A[s] = B[s] + 1
  C[I] = A[s] * 2
endfor
```

Outro Exemplo - *Loop Fusion*

```
for I = 1 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 99 do
  C[I] = A[I+1] * 2
endfor
```

- Mesmo trip count! Podemos aplicar *loop fusion*?

Outro Exemplo - *Loop Fusion*

```
for I = 1 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 99 do
  C[I] = A[I+1] * 2
endfor
```

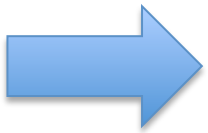
```
A[1] = B[1] + 1
for I = 2 to 99 do
  A[I] = B[I] + 1
endfor
for I = 1 to 98 do
  C[I] = A[I+1] * 2
endfor
I = 99
C[I] = A[I+1] * 2
```

- E agora?

Loop Fission

- Dividir um laço em dois ou mais laços pequenos
  - Também chamada de *Loop distribution*
  - Oposto de *Loop Fusion*

```
for I = ... do
 part1
 part2
endfor
```



```
for I = ... do
  part1
endfor

for I = ... do
  part2
endfor
```

Loop Fission

- Dividir um laço em dois ou mais laços pequenos
  - Também chamada de *Loop distribution*
  - Oposto de *Loop Fusion*
- Vantagens
  - Máquinas com *cache* de instruções pequenas
  - Melhorar localidade de dados
  - Habilita outras transformações: p.e. *loop interchanging*

Loop Fission

- Alguns casos exigem uma análise detalhada das dependências de dados.

```
for I = ... do
  A[I] = A[I] + B[I-1]
  B[I] = C[I-1]*X + C
  C[I] = 1/B[I]
  D[I] = sqrt(C[I])
endfor
```

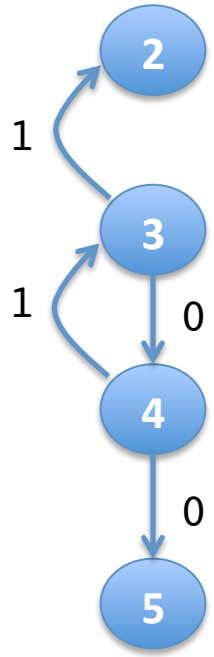
```
for I = ... do
  A[I] = A[I] + B[I-1]
endfor
for I = ... do
  B[I] = C[I-1]*X + C
endfor
for I = ... do
  C[I] = 1/B[I]
endfor
for I = ... do
  D[I] = sqrt(C[I])
endfor
```

Está correto?

Loop Fission

- Alguns casos exigem uma análise detalhada das dependências de dados.

```
(1) for I = ... do
(2)   A[I] = A[I] + B[I-1]
(3)   B[I] = C[I-1] * X
(4)   C[I] = 1/B[I]
(5)   D[I] = sqrt(C[I])
(7) endfor
```

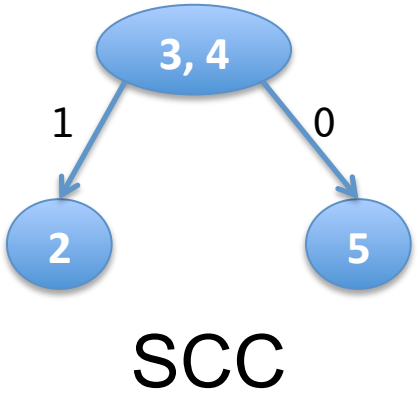


Está correto?

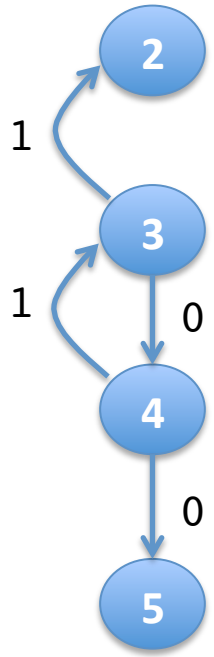
Loop Fission

- Alguns casos exigem uma análise detalhada das dependências de dados.

```
(1) for I = ... do
(2)   A[I] = A[I] + B[I-1]
(3)   B[I] = C[I-1] * X
(4)   C[I] = 1/B[I]
(5)   D[I] = sqrt(C[I])
(7) endfor
```



SCC



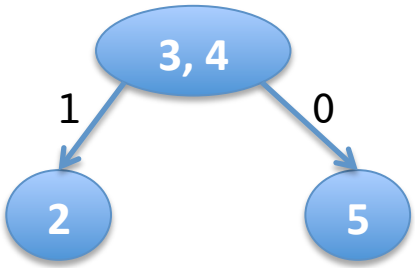
Grafo de dependências

Loop Fission

- Alguns casos exigem uma análise detalhada das dependências de dados.

```
(1) for I = ... do
(2)   A[I] = A[I] + B[I-1]
(3)   B[I] = C[I-1] * X
(4)   C[I] = 1/B[I]
(5)   D[I] = sqrt(C[I])
(7) endfor
```

SCC



```
for I = ... do
  B[I] = C[I-1]*X + C
  C[I] = 1/B[I]
endfor
for I = ... do
  A[I] = A[I] + B[I-1]
endfor
for I = ... do
  D[I] = sqrt(C[I])
endfor
```

Loop Fission

- Laços com variáveis escalares podem requerer *Scalar Expansion*

```
(1) for I = 1 to N do
(2)   T = A[I] + B[I]
(3)   C[I] = T + 1/T
(4) endfor
```

```
for I = 1 to N do
  T = A[I] + B[I]
endfor
for I = 1 to N do
  C [I] = T + 1/T
endfor
```

Está correto?

Loop Fission

- *Scalar Expansion*: Promove variáveis escalares a vetores

```
(1) for I = 1 to N do
(2)   T = A[I] + B[I]
(3)   C[I] = T + 1/T
(4) endfor
```

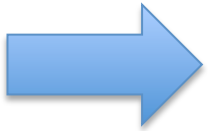
```
if N >=1 then
  allocate Tx(1:N)
  for I = 1 to N do
    Tx[I] = A[I] + B[I]
    C[I] = Tx[I] + 1/Tx[I]
  endfor
  T = Tx[N]
endif
```

E agora? Podemos aplicar *Loop Fission*?

Loop Interchanging

- Troca o laço mais externo pelo mais interno
  - Ajuda a expor paralelismo
  - Pode melhorar o desempenho da *cache*

```
for I = ... do
  for J = ... do
    body
  endfor
endfor
```

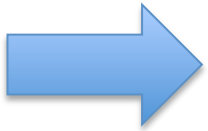


```
for J = ... do
  for I = ... do
    body
  endfor
endfor
```

Exemplo - *Loop Interchanging*

- *Arrays* em C são representados no formato *row-major*
  - Ex: ..., a[1,1], a[1,2], a[1,3], ..., a[2,1], a[2,2], ...
  - Após acessar a[1,1], possívelmente a[1,2], a[1,3] e a[1,4] estão na *cache* (Mesma linha de *cache*). Depende do modelo da *cache.*

```
for J=0 to 20 do
  for I=0 to 10 do
    A[I,J] = I+J
  endfor
endfor
```



```
for I=0 to 10 do
  for J=0 to 20 do
    A[I,J] = I+J
  endfor
endfor
```

Exemplo - *Loop Interchanging*

- Assumindo que o *Array* B está no formato *row-major*, é vantajoso aplicar *Loop Interchanging* no laço abaixo?
  - Row-major: B[0,0], B[0,1], B[0,2], ... B[1,0], B[1,1], ...

```
for I=0 to 10000 do
  for J=0 to 1000 do
    A[I] = A[I] + B[I,J] * C[I]
  endfor
endfor
```