




# 并行编译与优化 Parallel Compiler and Optimization

计算机研究所编译系统室 方建滨

*Lecture Fourteen:*  
*Loop Transformation (I)*

**第十四课：循环变换（一）**

2024-05-14

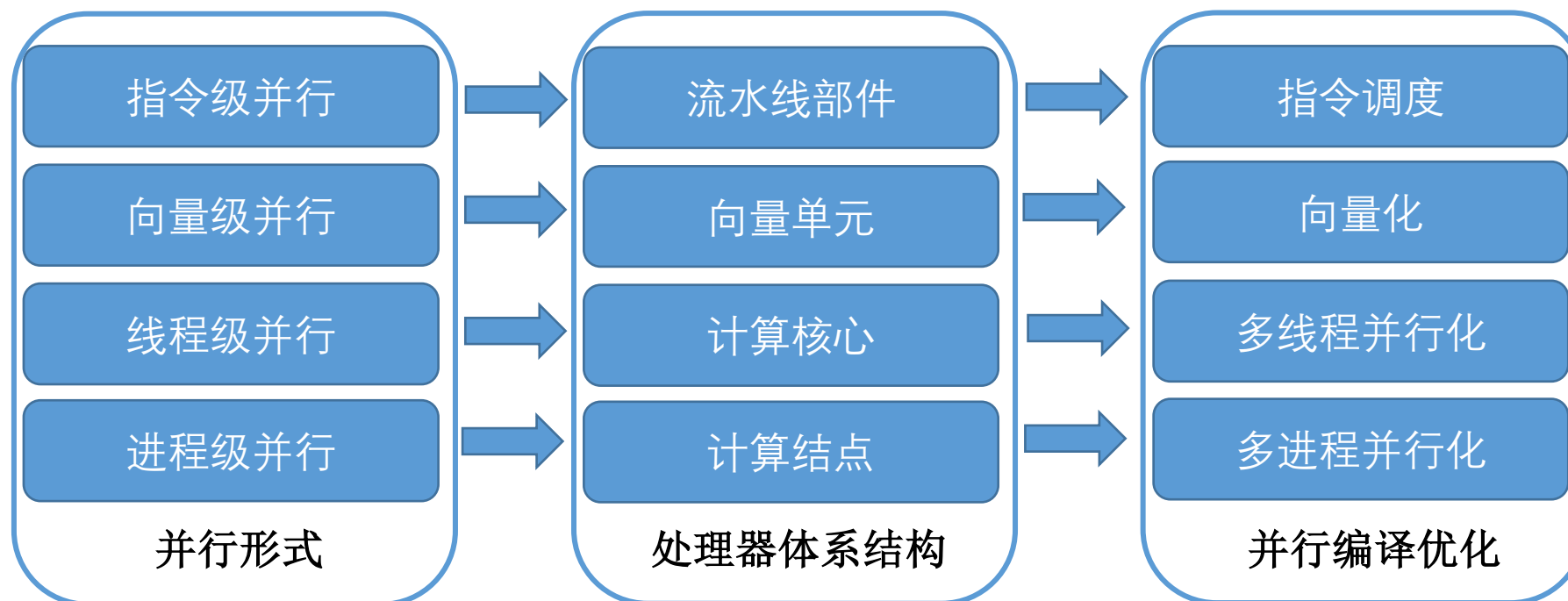
- **1. 循环变换简介**       理解循环重要性、循环变换概念及循环变换目的与分类
- **2. 简单循环变换**       掌握几种典型的简单循环变换
- **3. 高级循环变换**       掌握几种典型的高级循环变换
- **4. 课堂小结与作业**

## ■ SPEC CPU是国际公认的评测处理器的基准程序

⊕ 浮点计算性能和整数计算性能

⊕ 循环占据了SPEC CPU程序90%以上的执行时间

## ■ 循环代码是编译优化加速的主要对象(循环变换)



## ■ 循环变换 (Loop Transformation)

- ⊕ 改变循环的组织形式或样式

- ⊕ 有可能改变循环语句实例的**执行顺序**，但是不改变语句实例的**集合**

## ■ 对一个循环施加循环变换后得到的结果称为变换后的**程序** (Transformed Program)

```
L:  do I = 1,N  
S1:  A(I) = D(I) * 2  
S2:  C(I) = B(I) + A(I)  
      enddo
```

```
L1:  do I = 1,N  
S1:  A(I) = D(I) * 2  
      enddo  
L2:  do I = 1,N  
S2:  C(I) = B(I) + A(I)  
      enddo
```

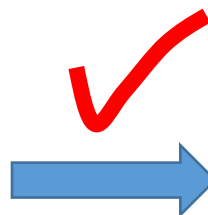
变换前:  $S_1(1), S_2(1), S_1(2), S_2(2), \dots, S_1(n), S_2(n)$



变换后:  $S_1(1), S_1(2), \dots, S_1(n), S_2(1), S_2(2), \dots, S_2(n)$

- **合法的变换**：只有变换后的程序与原循环保持**等价的**变换才是**合法的** (valid)
  - ⊕ 并不是所有循环变换都是合法的
- **等价的变换**：如果在原循环中语句S和语句T之间存在依赖关系，且变换后的程序仍**保持这种依赖关系**，那么变换后的程序就与原循环等价(**equivalent**)

```
L:  do I = 1,N  
S1:  A(I) = D(I) * 2  
S2:  C(I) = B(I) + A(I)  
      enddo
```



```
L1:  do I = 1,N  
S1:  A(I) = D(I) * 2  
      enddo  
L2:  do I = 1,N  
S2:  C(I) = B(I) + A(I)  
      enddo
```

- 降低循环控制开销
- 增加代码的指令级并行度 (instruction parallelism)
- 提高数据局部性 (data locality)
  - ⊕ 空间局部性和时间局部性
- 并行化 (parallelization)
- 向量化 (vectorization)

## ■ 简单循环变换

- ⊕ 只改变循环原有的组织形式，但**不改变循环语句实例的执行顺序**
- ⊕ 简单循环变换必定是合法的变换

## ■ 高级循环变换

- ⊕ **改变循环的语句实例执行顺序**，但不改变循环的语句实例集合
- ⊕ 为了保证变换的合法性，需要进行数据依赖关系分析

```
L:  do I = 1,N  
    S1:  A(I) = D(I) * 2  
    S2:  C(I) = B(I) + A(I)  
  enddo
```



```
L1:  do I = 1,N  
    S1:  A(I) = D(I) * 2  
  enddo  
L2:  do I = 1,N  
    S2:  C(I) = B(I) + A(I)  
  enddo
```



## ■ 简单循环变换与高级循环变换

	简单循环变换	高级循环变换
改变循环组织形式	是	是
改变语句实例执行顺序	否	是

## 2 简单循环变换

- 2.1 循环剥除 Loop peeling
- 2.2 循环分裂 Loop splitting
- 2.3 循环去开关化 Loop unswitching
- 2.4 循环展开 Loop unrolling
- 2.5 循环分段 Loop blocking/strip mine
- 2.6 标量扩张 Scalar expansion

# 什么是循环剥除?

- 将循环的前几次或者后几次迭代分离成单独的代码

```
do i =lo, hi  
  H(i)  
enddo
```



```
H(lo)  
...  
H(k)  
do i =k+1, m  
  H'(i)  
enddo  
H(hi-m+1)  
...  
H(hi)
```

## ■ 删除某些数据依赖

```
do i = 1, n
S:  y(i, n) = y(1, n) + y(n,n)
enddo
```

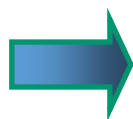
*i = 1 and i = n 有依赖*



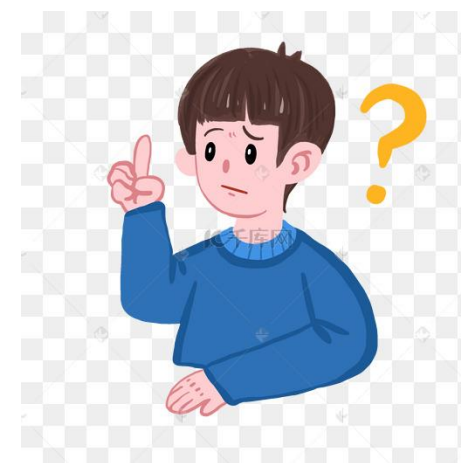
```
y(1, n) = y(1, n) + y(n,n)
do i = 2, n-1
S:  y(i, n) = y(1, n) + y(n,n)
enddo
y(n, n) = y(1, n) + y(n,n)
```

## ■ 消除循环中的分支判断

```
for ( i = 1; i < n; i++) {  
    if (i == 1)  
        a[i] = 0;  
    else  
        a[i] = b[i-1];  
}
```

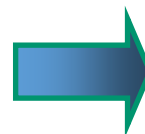


```
a[1] = 0;  
for ( i = 2; i < n; i++) {  
    a[i] = b[i-1];  
}
```



## ■ 消除小循环的循环控制开销

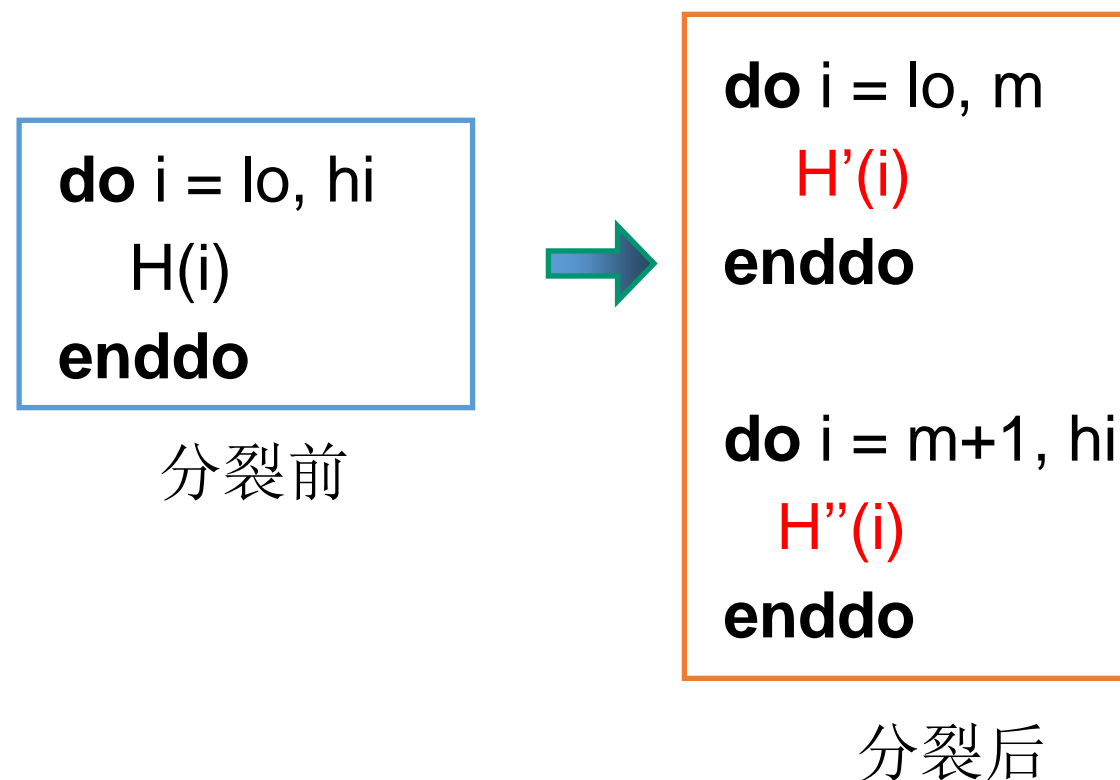
```
for (i = 0; i < 4; i++)  
    a[i] = c;
```



```
a[1] = c;  
a[2] = c;  
a[3] = c;  
a[4] = c;  
i = 4;
```

- 循环分裂又称索引集合 (Index Set) 分裂, 它将一个循环的索引集合分成两部分, 即变为两个循环, 并对两部分的循环体作必要的修改

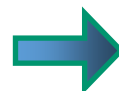
⊕ 副作用: 增大了代码体积



## ■ 删除与循环索引变量相关的条件判断

```
do i = 1, 100  
  A(i) = B(i) + C(i)  
  if i > 10 then  
    D(i) = A(i) + A(i-10)  
  endif  
enddo
```

分裂前



```
do i = 1, 10  
  A(i) = B(i) + C(i)  
enddo  
  
do i = 11, 100  
  A(i) = B(i) + C(i)  
  D(i) = A(i) + A(i-10)  
enddo
```

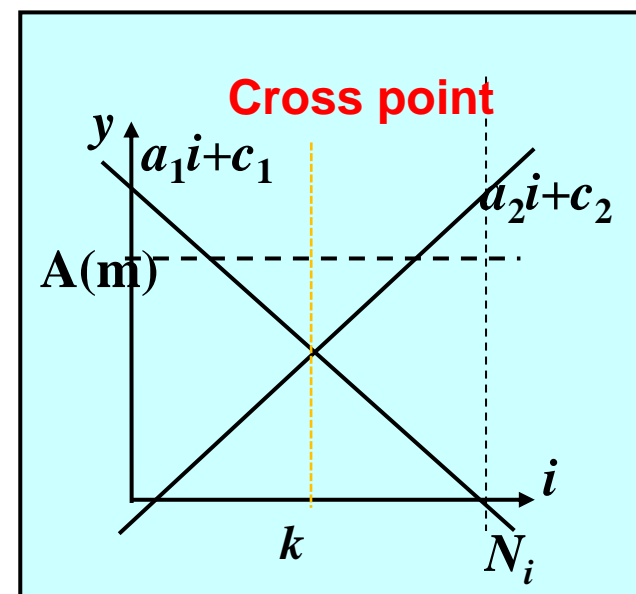
分裂后



## ■ 消除数据依赖

```
do i = 1, n
S:   y(i) = y(n-i+1)
enddo
```

```
do i = 1, (n+1)/2
  y(i) = y(n-i+1)
enddo
do i = (n+1)/2+1, n
  y(i) = y(n-i+1)
enddo
```

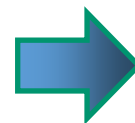


$$k = (n+1)/2$$

Weak Crossing SIV

■ 循环去开关化是指将**循环不变的条件分支**移至循环体之外

```
do i=1, n
  do j=2, n
    if (t(i) .gt. 0) then
      a(i,j) = a(i, j-1)*t(i)
    else
      a(i,j) = t(i)
    endif
  enddo
enddo
```



```
do i=1, n
  if (t(i) .gt. 0) then
    do j=2, n
      a(i,j) = a(i, j-1)*t(i)
    enddo
  else
    do j=2, n
      a(i,j) = t(i)
    endo
  endif
enddo
```

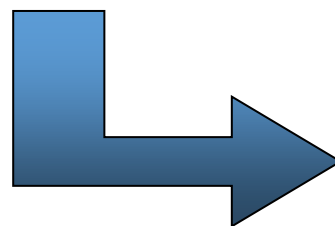
变换前

变换后

- 条件左右表达式必须都是循环不变量
- 条件判断分支必须紧接着将要被转换的循环
- 循环不变的条件不必是整个谓词
- 当没有else分支时，需提供相应的补偿代码

```
do i=1 to N
  if ((k > 2).and(a(i).gt.0) then
    A(i) = A(i)*T(i)
  endif
enddo
```

去开关化前



```
if (k > 2) then
  do i=1 to N
    if (a(i).gt.0) then
      A(i) = A(i)*T(i)
    endif
  enddo
else
  i=N+1
endif
```

## ■ 优点

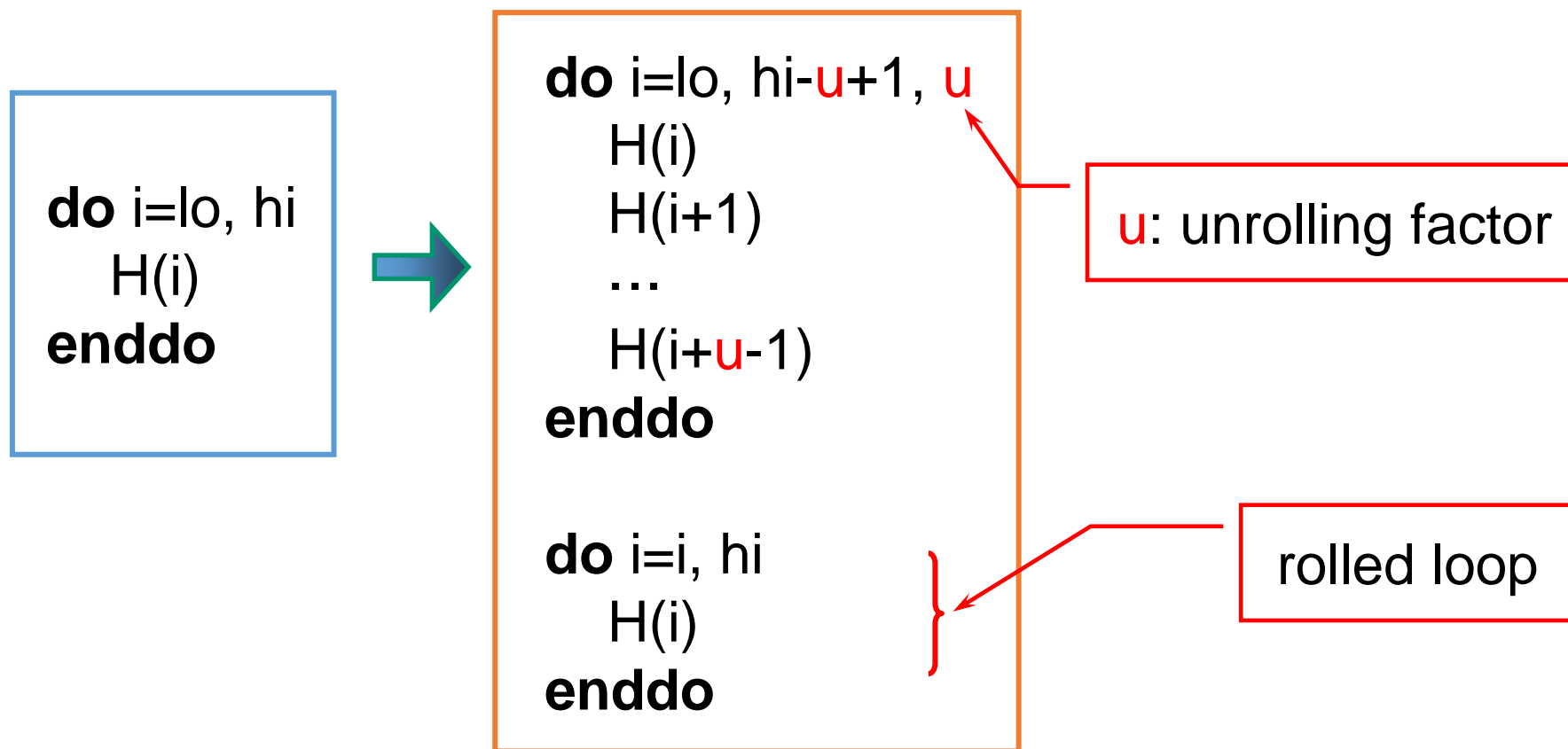
- ⊕ 减少了条件语句的执行次数
- ⊕ 减少了循环的执行次数
- ⊕ 将分支移出循环，可能有利于其他循环变换

## ■ 副作用：增加了代码体积

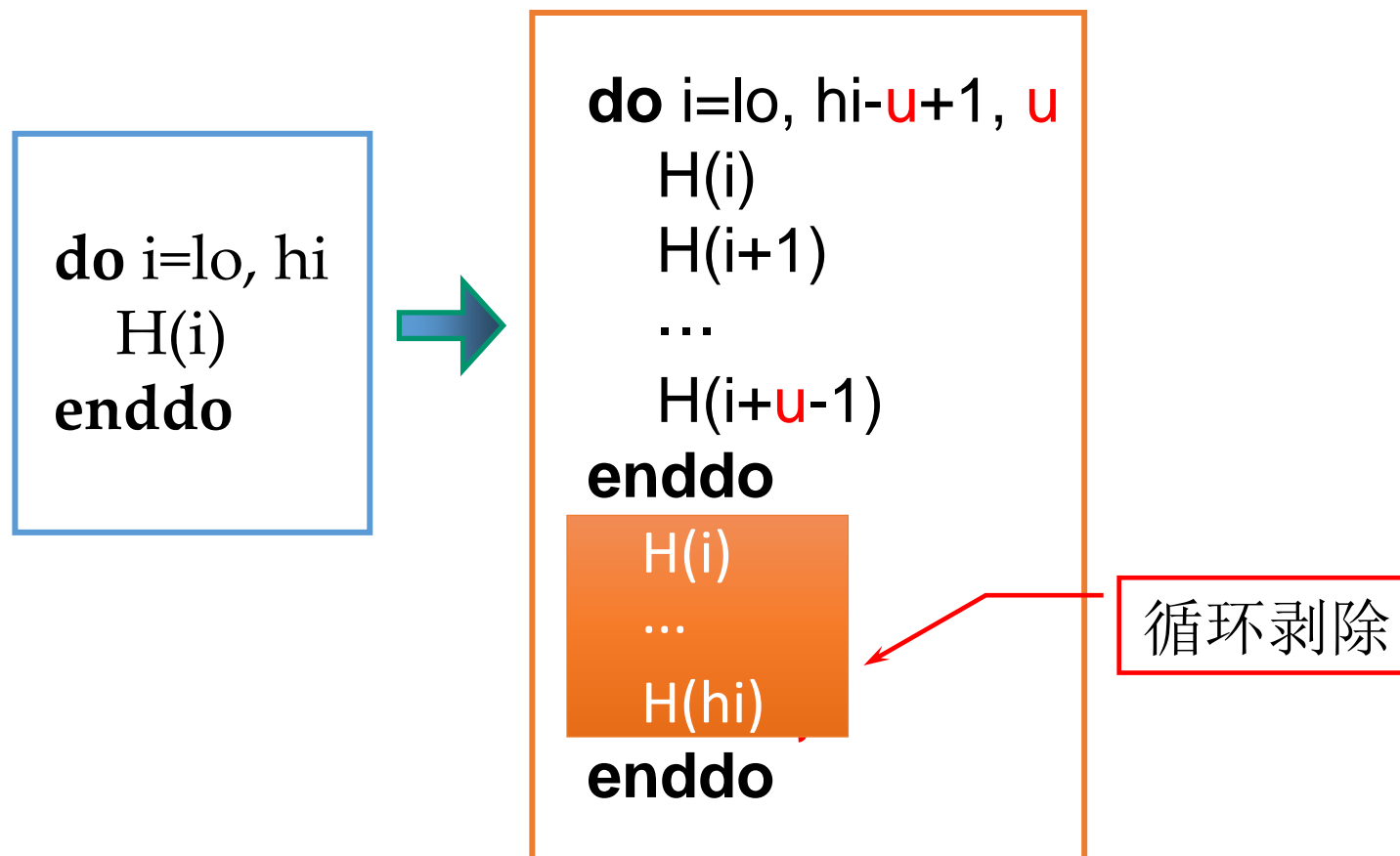
## ■ 优化建议

- ⊕ 避免过分去开关化
- ⊕ 避免对“冷代码区域”去开关化

- 循环展开是指将循环体替换为多个拷贝并调整对应的循环控制代码



- 当循环上下界为常数时，可通过循环剥除将尾循环(rolled loop)消除



## ■ 垂直展开

```
do i=1, n  
  a[i]=a(i+1)*b[i]  
enddo
```



```
do i=1, n-3, 4  
  a[i]=a(i+1)*b[i]  
  a[i+1]=a(i+2)*b[i+1]  
  a[i+2]=a(i+3)*b[i+2]  
  a[i+3]=a(i+4)*b[i+3]  
enddo  
  
do i=i, n  
  a[i]=a(i+1)*b[i]  
enddo
```

## ■ 水平展开

```
do i=1, 100  
  S = S + A(i)  
enddo
```



```
do i=1, 100, 4  
  S = S + A(i)  
    + A(i+1)  
    + A(i+2)  
    + A(i+3)  
enddo
```



## ■减少循环控制开销

## ■扩大循环体

- ⊕增加可用的指令级并行性

- ⊕为其它优化提供更多机会

  - ◆公用子表达式删除、归纳变量优化、指令调度、软流水等

## ■副作用：代码体积增大

- ⊕使用的操作数增多，可能增加寄存器压力

- ⊕可能使原本指令cache重用性很好的循环变得cache频繁失效

## ■两个基本问题

- ⊕选择展开哪一个循环？
- ⊕循环展开因子是多少？

## ■循环展开遵循的规则（确定展开哪个循环）

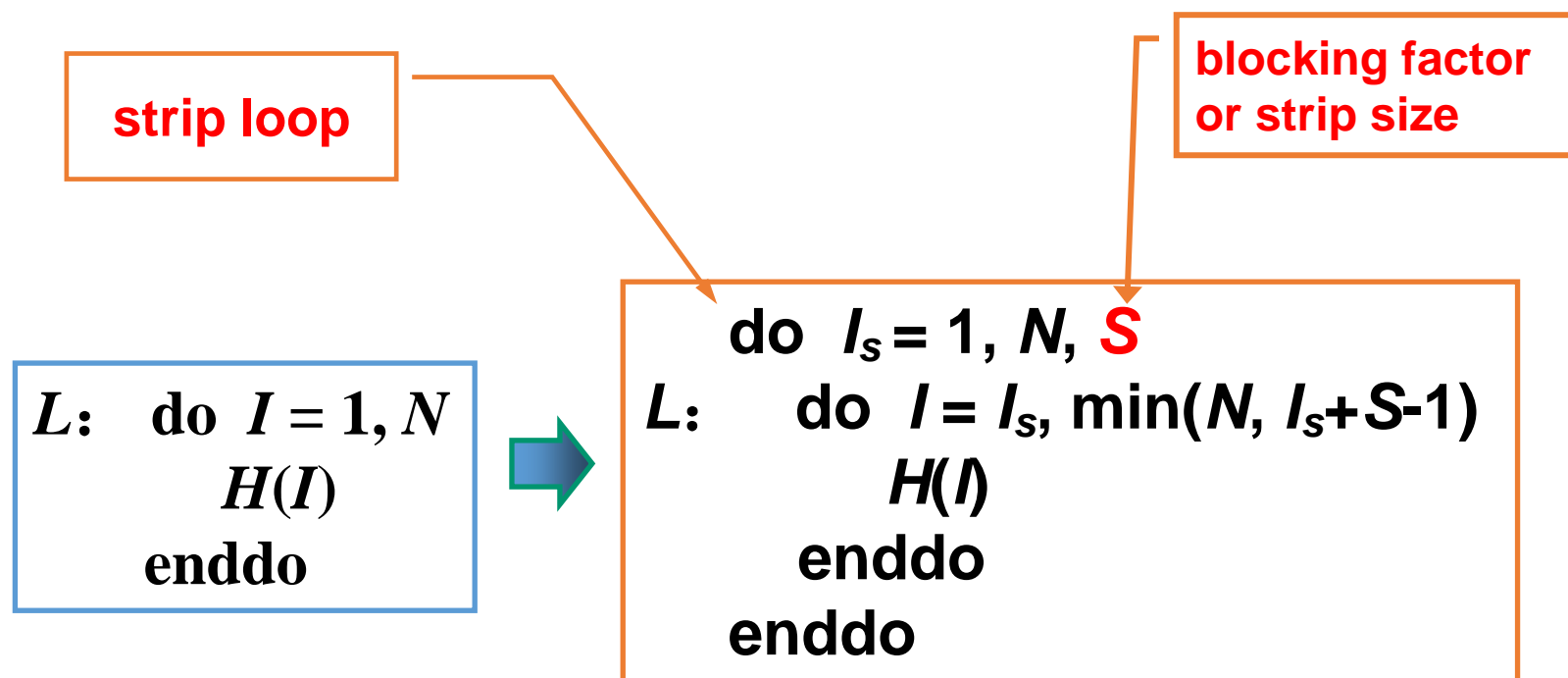
- ⊕循环中仅包含一个基本块
- ⊕循环中存访指令与浮点指令具有某种平衡性
- ⊕循环生成少量的中间指令
- ⊕具有简单的循环控制代码

扩展阅读：训练AI模型

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, Michael F. P. O'Boyle: **Milepost GCC: Machine Learning Enabled Self-tuning Compiler**. Int. J. Parallel Program. 39(3): 296-327 (2011)

# 什么是循环分段?

- 循环分段将一个单层循环分成**两层嵌套循环**，即外层循环把原循环迭代空间分成不同的段，一个段中含有多个原循环的迭代



- 有利于向量化
- 提升访存的数据局部性
- 降低访存请求数量

## ■ 确定分段大小

⊕ 向量化单元长度

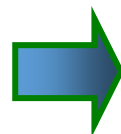
⊕ 缓存大小

⊕ 代价模型

```
L: do  $l_s = 1, N, S$   
    do  $l = l_s, \min(N, l_s + S - 1)$   
         $H(l)$   
    enddo  
enddo
```

■ 标量扩张是将循环中的一个标量扩展为一个向量或者一个数组

```
L:  do i=1, N  
S1:  T = A[i]  
S2:  A[i]=B[i]  
S3:  B[i] = T  
      enddo
```

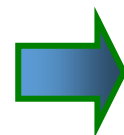


```
L':  do i=1, N  
S1:  $T[i] = A[i]  
S2:  A[i]=B[i]  
S3:  B[i] = $T[i]  
      enddo  
T=$T[N]
```

## ■由T引起的依赖关系分为两种情况

- ⊕ 因为值的重用而引起，必须保持
- ⊕ 因为存储单元的重用而引起，可以通过扩展来消除

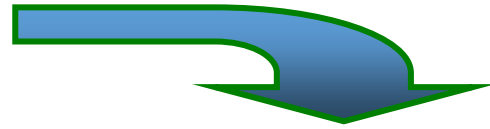
```
L:  do i=1, N
S1:  T = A[i]
S2:  A[i]=B[i]
S3:  B[i] = T
      enddo
```



```
L':  do i=1, N
S1:  $T[i] = A[i]
S2:  A[i]=B[i]
S3:  B[i] = $T[i]
      enddo
      T=$T[N]
```

*For T, S1 $\delta^f$  S3, S1 $\delta^o$  S1, S3 $\delta^a$  S1*

```
DO I=1,100  
S1    A(I)= T  
S2    T =B(I)+C(I)  
ENDDO
```



```
T$(0) = T  
DO I = 1, 100  
S1    A(I) = T$(I-1)  
S2    T$(I)= B(I) + C(I)  
ENDDO  
T = T$(100)
```

经过标量扩张



■副作用：增加了存储需求

■解决方案

⊕1. 在扩张前进行循环分段

⊕2. 向前替换

```
do I = 1, N
  A(I) = T
  T = B(I) + C(I)
enddo
```

```
do I = 1, N
  T = A(I) + A(I+1)
  A(I) = T + B(I)
enddo
```

```
do I = 1, N, 4
  do J=0, 3
    A(I) = T
    T = B(I) + C(I)
  enddo
enddo
```

```
do I = 1, N
  A(I) = A(I) + A(I+1) + B(I)
enddo
```

## ■ 针对给定程序代码进行**循环编译优化**的步骤

⊕ **识别**程序中的循环（回顾）

⊕ **分析**一个循环是否可被并行化/向量化/循环分块等

◆ 如果不能，是否有合适的、合法的**循环变换**改变循环形式

⊕ **循环优化目标**：并行化/向量化/循环分块等



- **龙书第11章**
- **Optimizing Compilers Modern Architecture, chap5~6, chap 8, chap9.1~9.4**
- **Utpal Banerjee, Loop Transformations for Restructuring Compilers, Kluwer A. Publishers, 1993**

# Backup Slides

- 循环分段增加了循环的嵌套层次，可能改变循环中依赖的依赖距离或方向
- 如果依赖距离  $d$ ，段长为  $s$ ，则该依赖的距离改变为  $(d/s, d \bmod s)$ ；如果  $d \bmod s$  不为零，还将产生一个依赖距离为  $(d/s+1, -(s-d) \bmod s)$  的依赖。
- 若依赖距离未知，或分段的段长不是编译时已知的常数，则只能知道依赖方向的变化情况

老依赖方向	新依赖方向
<	(<, *) (=, <)
=	(=, =)
>	(>, *) (=, >)

```
L:  do I = 1, 16  
      A(I+3) = A(I)+B(I)  
    enddo
```

$d = 3$   
 $S = 5$

*blocking*

```
L:  do I = 1, 16, 5  
      do J = I, min(16, I+4)  
          A(J+3) = A(J)+B(J)  
        enddo  
      enddo
```

$d = (0, 3)$   
 $d = (1, -2)$

$(d/s, d \bmod s)$

$(d/s+1, -(s-d) \bmod s)$

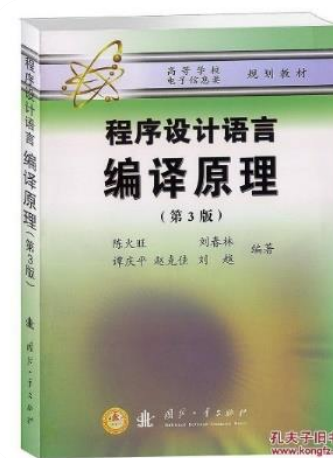
- 循环必须是可计数的，并且标量没有向上暴露的使用
- 如果该标量在循环出口活跃，则在循环后需要将数组的最后一次赋值的值赋给该标量
- 循环中来自标量的流依赖必须是循环无关的

- 循环变换的代价/好处
- 循环变换相互影响，如何选择？
- 建议手工改写程序，测试各种循环变换的优化结果



## ■媲美Cray I 的 YH-1巨型机

- ⊕ 1983 年，采用向量结构的银河 I 研制成功，运算速度达每秒 1 亿次 (100Mflops)，**我国首台亿次机**
- ⊕ 陈火旺主持研制了向量FORTRAN77语言编译系统，**我国第一个完整实现FORTRAN77全集的编译系统**
- ⊕ 1972-1975年，陈火旺主持研制了面向441B机的FORTRAN编译系统，**我国第一个FORTRAN编译系统（编译南方会战）**

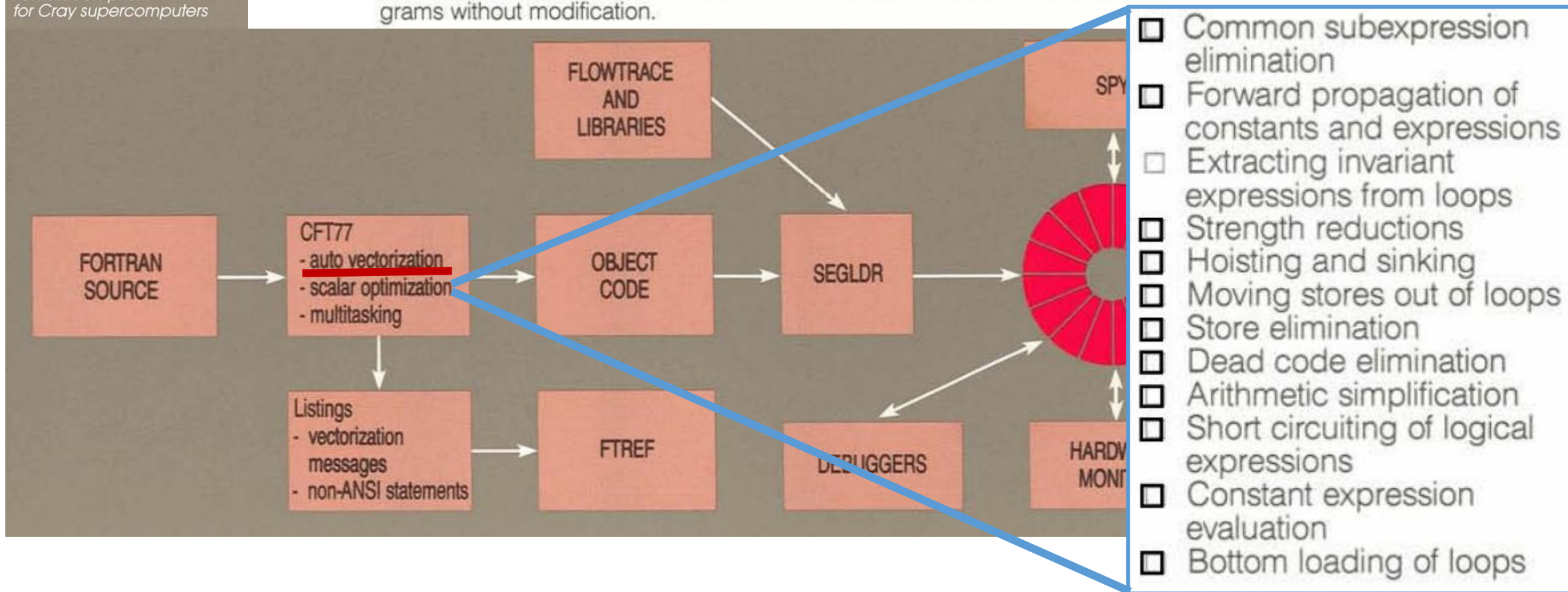


# 向量机时代：向量编译器

**CFT77**

Fortran compiler  
for Cray supercomputers

CFT77 is a multipass, optimizing, vectorizing, and multitasking compiler that adheres to the American National Standards Institute (ANSI) standard 3.9-1978 (often called Fortran 77). CFT77 processes existing standard Fortran programs without modification.



## ■ 循环变换的关键是找寻一个新的合适的、合法的迭代执行序列

⊕ 合适的：根据收益模型

⊕ 合法的：不违背依赖关系，保持方向向量为“正”