Program Optimization

俞子杰

Performance/Complexity

- 我们在本章（对于编译器的）所谓的优化：Performance
- 一般不会影响<u>时间复杂度</u>
- 但是会影响<u>常数（constant）</u>，而常数同样对程序的运行效率影响很大（ O(logn)不一定执行时间短于O(n) ）

Optimizing Compilers

- 编译器可以优化
- 编译器优化限制

Generally Useful Optimizations

一般编译器可以执行的优化

Eg.删去无用代码

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

1、Code Motion

不影响程序正常运行结果的情况下代码移动，减少运算

Eg.如上图，对二维数组的访问过程的优化

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L1                     # If 0, goto done
        imulq   %rcx, %rdx              # ni = n*i
        leaq    (%rdi,%rdx,8), %rdx     # rowp = A + ni*8
        movl    $0, %eax                # j = 0
.L3:                                    # loop:
        movsd   (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd   %xmm0, (%rdx,%rax,8)    # M[A+ni*8 + j*8] = t
        addq    $1, %rax                # j++
        cmpq    %rcx, %rax              # j:n
        jne     .L3                     # if !=, goto loop
.L1:                                    # done:
        rep ; ret
```

2、Reduction in Strength

将运行代价更高的操作替换成更小的

Eg.　　16*x → x<<4

依赖于运行的机器，如乘除的运行时间

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

↑循环中，乘法优化成加法

3、Share Common Subexpressions

如果计算中出现重复的部分，反复利用（-O1）

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
...
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
...
```

Optimizing Bubblesort

此处A为数组起始位置，元素4字节——A[j] is located in &A+4*(j-1)
所使用的优化均独立于机器

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]    // A[j]
     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]    // A[j+1]
     if t3<=t7 goto L3
```

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

```
     t8 := j-1
     t9 := 4*t8
     temp := A[t9]   // temp:=A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]   // A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13   // A[j]:=A[j+1
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp    // A[j+1]:=ten
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instructions
29 in outer loop
25 in inner loop

Final Pseudo Code

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := i << 2
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instruction Count
Before Optimizations
29 in outer loop
25 in inner loop

Instruction Count
After Optimizations
15 in outer loop
9 in inner loop

- These were **Machine-Independent Optimizations**.
- Will be followed by **Machine-Dependent Optimizations**, including allocating temporaries to registers, converting to assembly code

Limitations of Optimizing Compilers

在编译器对某个优化存疑时，编译器选择保守
编译器的备选方案：不优化

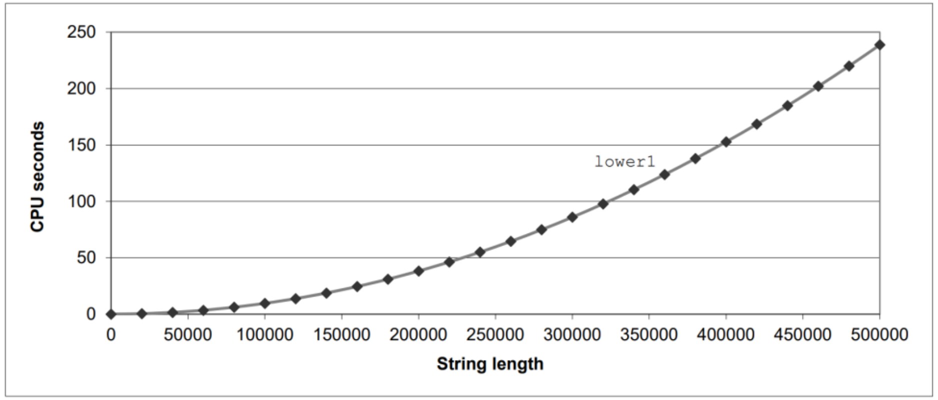
Optimization Blocker #1: Procedure Calls

事实上每次循环都会执行一次strlen，而strlen本身的复杂度为O(n)

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

对于编译器而言，不敢保证循环中字符串不会改变长度；

以及如果优化后，与其他文件链接后，strlen可能改变意义，也会出错。



编译器将过程视为黑箱

Optimization Blocker #2: Memory Aliasing

每次都需要重新从内存读取，但是编译器不优化

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

原因是存在“别名”情况导致如果优化，程序运行结果不同

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0      # FP load
        addsd   (%rdi), %xmm0             # FP add
        movsd   %xmm0, (%rsi,%rax,8)      # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

消除别名可以使用临时变量

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

Value of B:

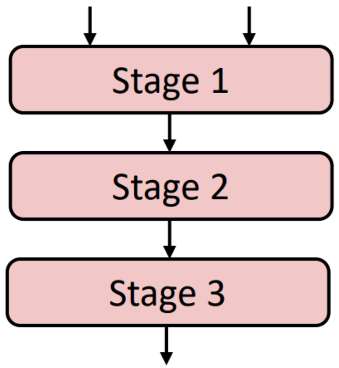
```
init:   [4, 8, 16]

i = 0:  [3, 8, 16]

i = 1:  [3, 22, 16]

i = 2:  [3, 22, 224]
```

Instruction-Level Parallelism

- Pipeline
- Out-of-order execution

Pipelined Functional Units

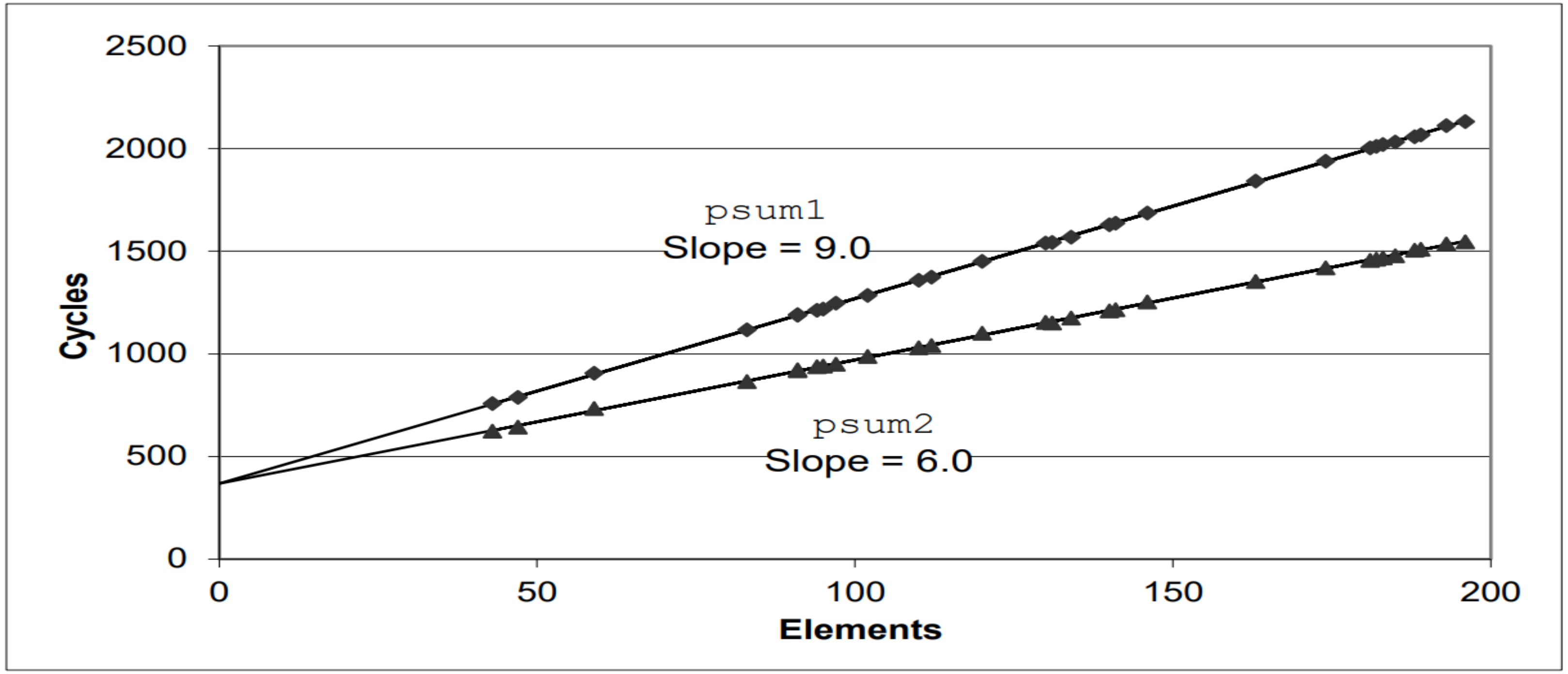
```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



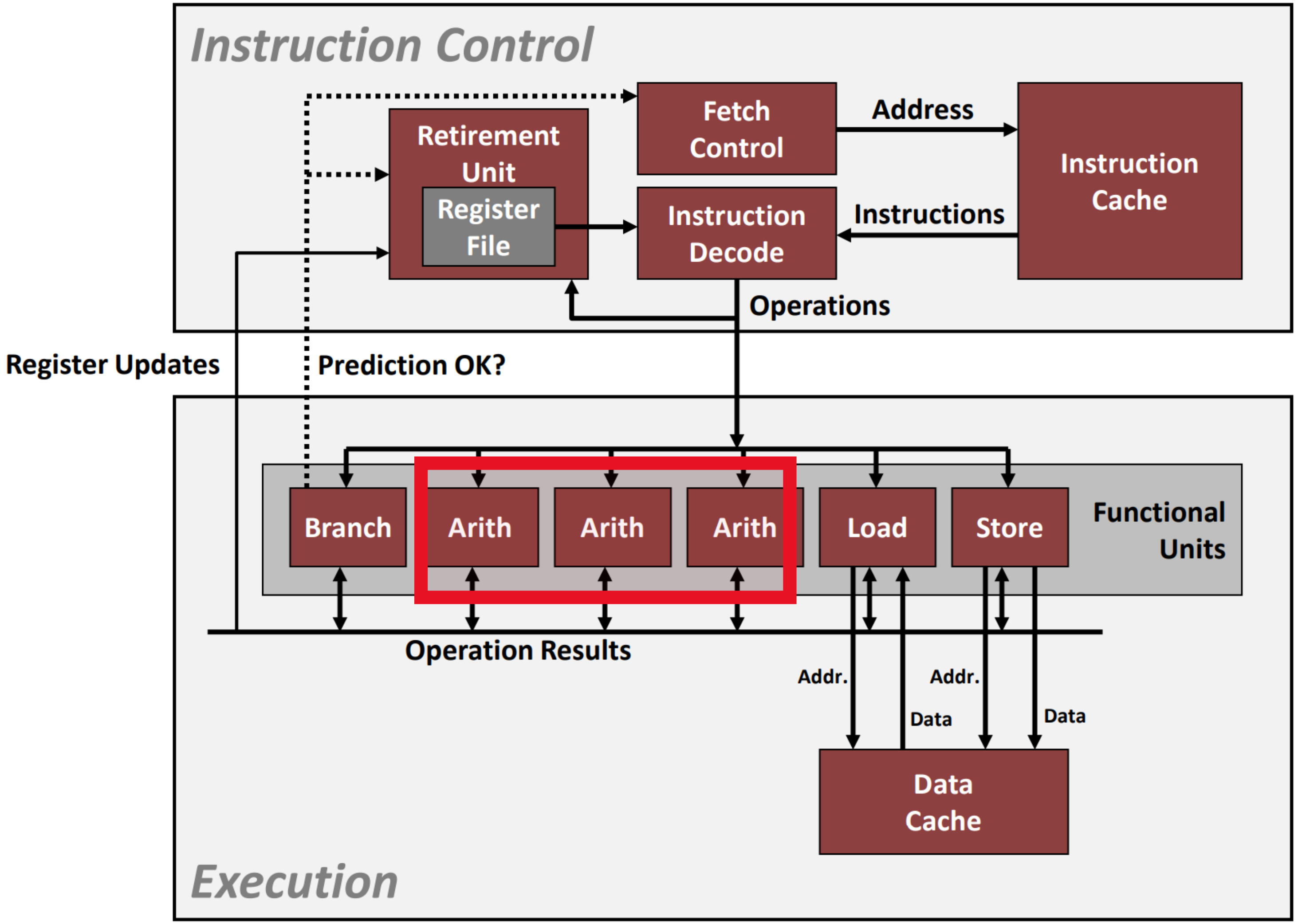
Time							
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

Cycles Per Element (CPE)

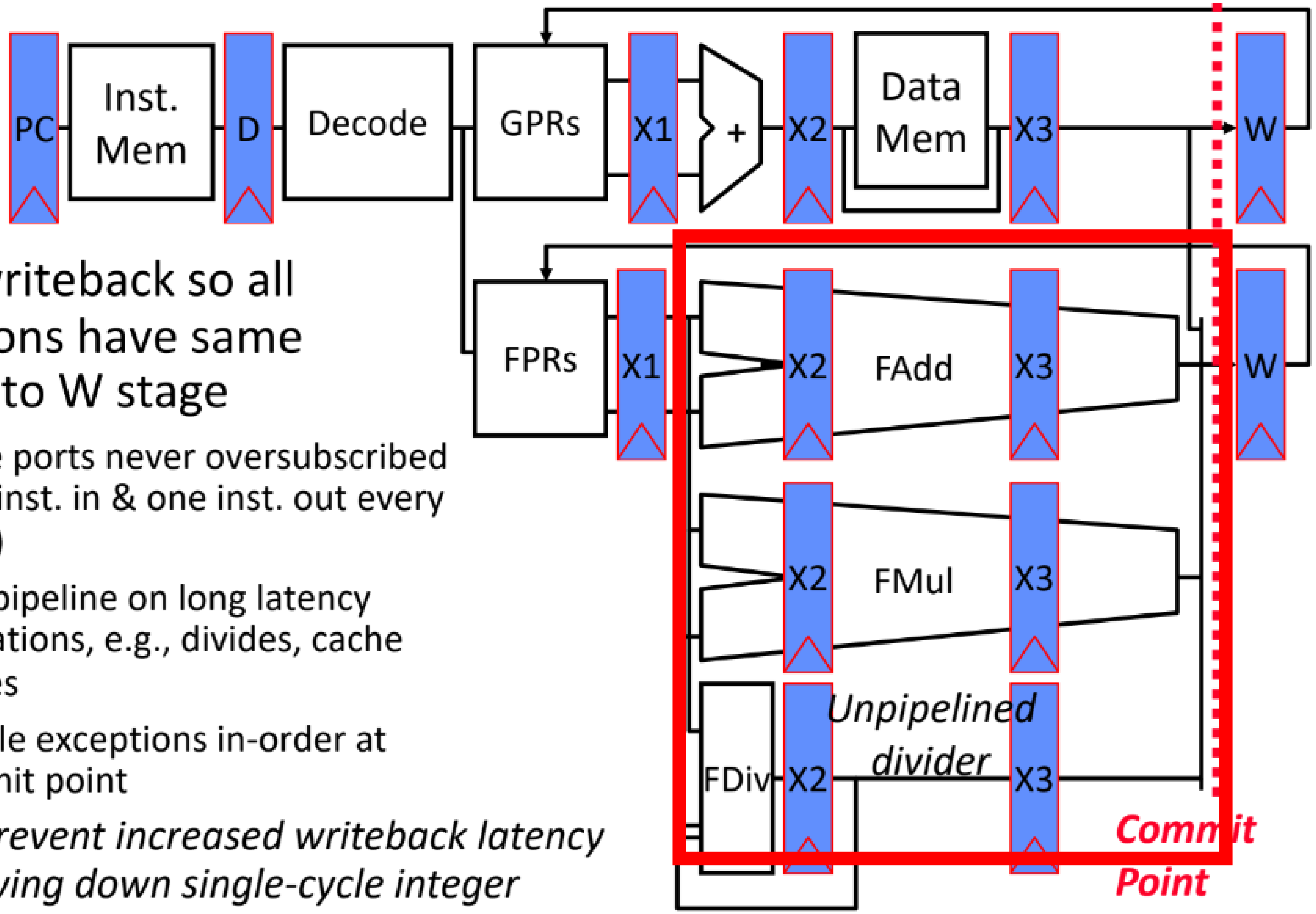
- **Convenient way to express performance of program that operates on vectors or lists**
- **Length = n**
- **In our case: CPE = cycles per OP**
- **T = CPE*n + Overhead**
  - CPE is slope of line



Modern CPU Design



多个算术单元



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single-cycle integer operations? Bypassing

存在单独计算乘除法的单元

所以有时不一定需要将乘除优化成加减和左右移

实现并行

Modern CPU Design

超标量乱序执行：CPU读取尽可能多的读取指令序列，相互独立的指令在一个时钟周期内同时进行

Critical Path

- 依赖关系限制执行顺序
- 关键路径是所有路径中运行用时最长的，是运行所需时钟周期数的下界

```c
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

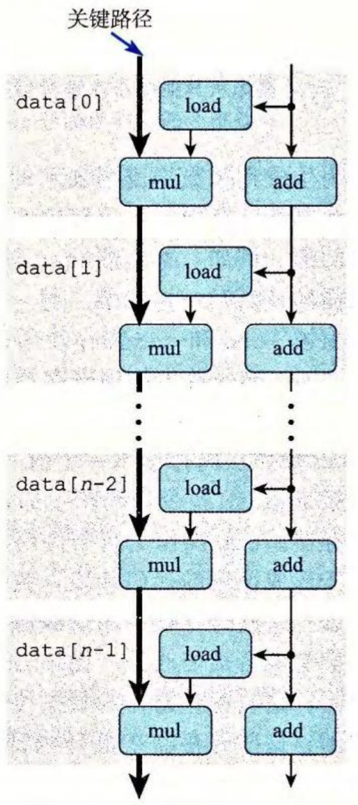


图 5-15 combine4 的内循环的 n 次迭代计算的数据流表示。乘法操作的序列形成了限制程序性能的关键路径

Parallelism : Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

延迟界限（Latency Bound）：有依赖关系顺序执行时对程序性能的限制
吞吐量界限（Throughput Bound）：程序性能的终极限制

此处编译器对FP+、FP*不会自动优化2×1a，浮点运算不具备结合律

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

2×1

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

2×2

2×1a： reassociassion

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

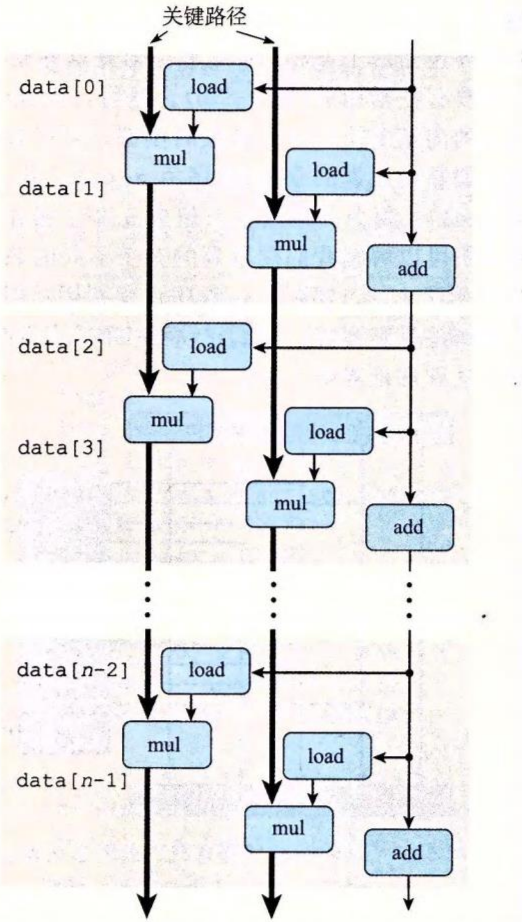
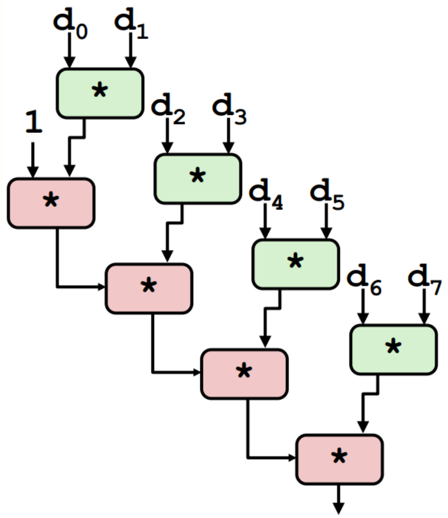
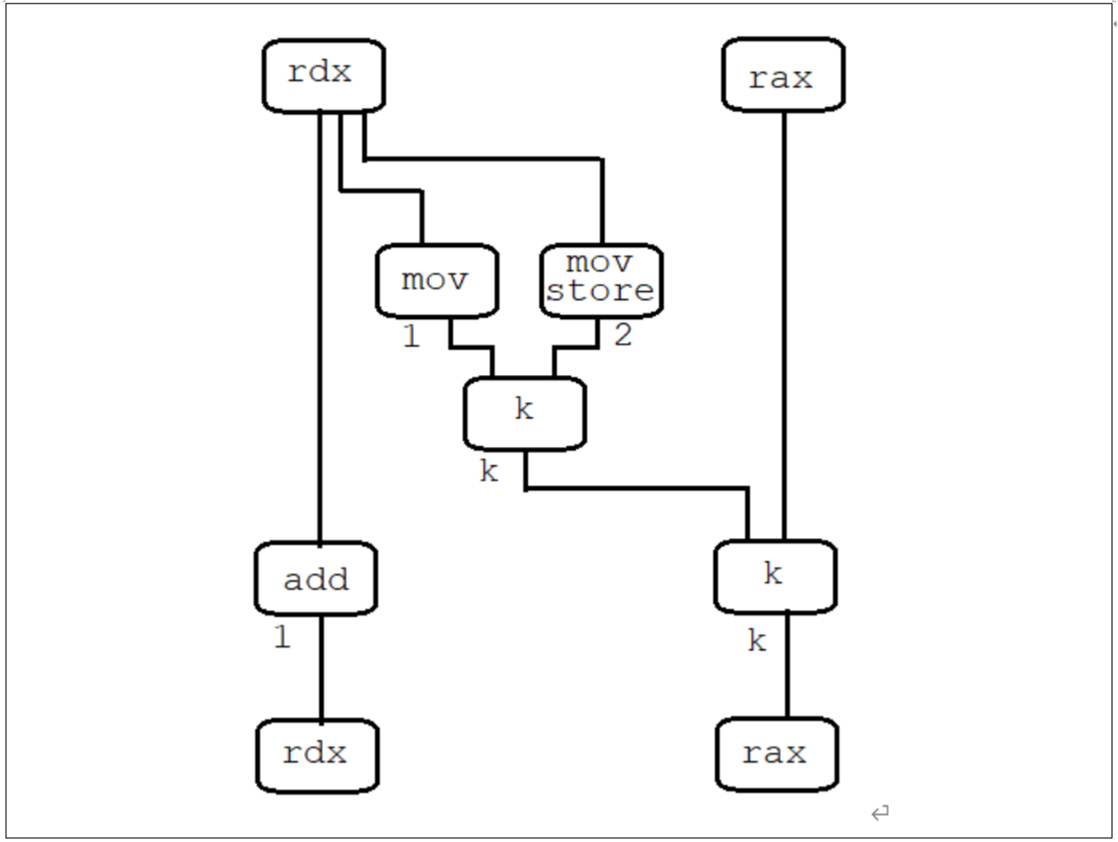


图 5-24 combine6 对一个长度为 n 的向量进行操作的数据流表示。现在有两条关键路径，每条关键路径包含 $n/2$ 个操作



- 该程序每轮循环处理两个元素。在理想的机器上（执行单元足够多），每条指令消耗的时间周期如右边所示。
- (1) 当问号处为乘法时，k = 8。此时这段程序的CPE为____
- (2) 当问号处为加法时，k = 1。此时这段程序的CPE为____

数据相关图：



k×1、k×1a、k×k

- 可突破延迟界限，逼近吞吐量界限
- 但是仍有限制，如k=20，超出寄存器数量，在运行时堆栈上分配空间，导致效率降低

11. 假设已有声明 int i, int sum, int *p, int *q, int *r, const int n = 100, float a[n], float b[n], float c[n], int foo(int), void bar(), 以下哪项程序优化编译器总是可以进行？

<table>
<tr><td>A</td><td><pre>for(i = 0; i < n; ++i) {
    a[i] += b[i];
    a[i] += c[i];
}</pre></td><td><pre>float tmp;
for(i = 0; i < n; ++i) {
    tmp = b[i] + c[i];
    a[i] += tmp;
}</pre></td></tr>
<tr><td>B</td><td><pre>*p += *q;
*p += *r;</pre></td><td><pre>int tmp;
tmp = *q + *r;
*p += tmp;</pre></td></tr>
<tr><td>C</td><td><pre>for(i = 0; i < n; ++i)
    sum += i * 4;</pre></td><td><pre>int N = n * 4;
for(i = 0; i < N; i += 4)
    sum += i;</pre></td></tr>
<tr><td>D</td><td><pre>for(i = 0; i < foo(n); ++i)
    bar();</pre></td><td><pre>int tmp = foo(n);
for(i = 0; i < tmp; ++i)
    bar();</pre></td></tr>
</table>

答案：C

A 浮点数不满足结合律

B p, q, r 可能指向同一个地址

D foo 函数可能有副作用

12. 针对程序优化，请挑出下面唯一正确的陈述：

A. 用 add/sub 和 shift 替代 multiply/divide 永远能提高程序的运行速度。

B. 最有效的提高程序运行效率的方法是提高 compiler 的优化级别。

C. 跨 procedure 优化的障碍之一是因为使用了全局变量。

D. 程序中，*a += *b; *a += *b;永远可以用*a +=2*(*b);代替。

答案：C

解析： A 错因为如果 cpu 支持硬件乘除，则用 add/sub 来模拟乘除通常并不划算。

B 错因为优化算法更能提高运行效率。

C 对

D 错因为 a 和 b 可能指向同一数据。

20. 以下哪些程序优化编译器总是可以自动进行？（假设 int i, int j, int A[N], int B[N], float m 都是局部变量，N 是一个整数型常量，int foo(int) 是一个函数）

	优化前	优化后

A.	for (j = 0 ; j < N ; j ++)
	  m + = i*N*j;
		int temp = i*N;
		for (j= 0 ; j < N ; j ++)
		  m + = temp * j;

B.	for (j = 0 ; j < N ; j ++)
	  B[i] *= A[j];
		int temp = B[i];
		for (j= 0 ; j < N ; j ++)
		  temp *= A[j];
		B[i] = temp;

C.	for (j = 0 ; j < N ; j ++)
	  m = (m + A[j]) + B[j];
		for (j = 0 ; j < N ; j ++)
		  m = m + (A[j] + B[j]);

D.	for (j = 0 ; j < foo(N) ; j ++)
	  m++;
		int temp = foo(N);
		for (j= 0 ; j < temp ; j ++)
		  m++;

答案：A

说明：考察 procedure, memory aliasing, 和 floating 的精度问题

13、下面关于程序性能的说法中，哪种是正确的？

A. 处理器内部只要有多个功能部件空闲，就能实现指令并行，从而提高程序性能。

B. 同一个任务采用时间复杂度为 O(logN)算法一定比采用复杂度为 O(N)算法的执行时间短

C. 转移预测总是能带来好处，不会产生额外代价，对提高程序性能有帮助。

D. 增大循环展开（loop unrolling）的级数，有可能降低程序的性能（即增加执行时间）

答案： D

15、以下哪些程序优化编译器总是可以自动进行？（假设 int i, int j, int A[N], int B[N], int m 都是局部变量，N 是一个整数型常量，int foo(int) 是一个函数）答：(　　　　)

	优化前	优化后

A.	for (j = 0 ; j < N ; j ++)
	　m + = i*N*j;
		int temp = i*N;
		for (j= 0 ; j < N ; j ++)
		　m + = temp * j;

B.	for (j = 0 ; j < N ; j ++)
	　B[i] *= A[j];
		int temp = B[i];
		for (j= 0 ; j < N ; j ++)
		　temp *= A[j];
		B[i] = temp;

C.	for (j = 0 ; j < N ; j ++) 　m = (m + A[j]) + B[j];	for (j = 0 ; j < N ; j ++) 　m = m + (A[j] + B[j]);

D.	for (j = 0 ; j < foo(N) ; j ++)
	　m ++;
		int temp = foo(N);
		for (j= 0 ; j < temp ; j ++)
		　m ++;

答案：AC