第5章 程序优化

教 师： 吴锐
计算机科学与技术学院
哈尔滨工业大学

要点

- **综述**
- **普遍有用的优化方法**
  - 代码移动/预先计算
  - 复杂运算简化Strength reduction
  - 公用子表达式的共享
  - 去掉不必要的过程调用
- **妨碍优化的因素Optimization Blockers（优化障碍）**
  - 过程调用
  - 存储器别名使用Memory aliasing
- **运用指令级并行**
- **处理条件**

关于性能的现实

- ***性能比渐进复杂度（asymptotic complexity）更重要***
- **常数因子也很重要!**
  - 代码编写不同，性能差10倍
  - 一定要在多个层次进行优化:
    - 算法, 数据表示，过程，循环
- **优化性能一定要理解“系统”**
  - 程序是怎样被编译和执行的
  - 现代处理器+存储系统是怎么运作的
  - 怎样测量程序性能与定位“瓶颈”
  - 在不破坏代码模块化与整体性generality下，怎样改进程序性能

具备优化功能的编译器

- **提供程序到机器间的有效映射**
  - 寄存器分配
  - 代码选择与排序（调度）ordering (scheduling)
  - 消除死代码
  - 消除较小的低效代码eliminating minor inefficiencies
- **(通常)不提高渐进效率asymptotic efficiency**
  - 由程序员来选择最佳的总体算法
  - 大-O 比常数因子（常常）更重要
    - 但常数因子也很重要
- **难以克服“妨碍优化的因素/优化障碍”**
  - 潜在的内存别名使用memory aliasing
  - 潜在的函数副作用

优化编译器的局限性

- **在基本约束条件下运行**
  - 不能引起程序行为的任何改变
    - 例外：可能是程序在使用非标准语言特性
  - 会阻止可能导致病态行为的优化
- **对程序员来说很明显的行为可能会被语言和编码风格混淆**
  - 如 ,数据范围可能比变量类型建议的范围更小
- **大多数分析只在过程中执行**
  - 在大多数情况下，全程序分析过于昂贵
  - 新版本的GCC在单独的文件中进行了过程间分析
    - 但是,不是在不同文件之间的代码
- **大多数分析都是基于静态信息的**
  - 编译器很难预测运行时输入
- **当有疑问时，编译器必须是保守的**

常用的优化手段

- ***不考虑具体处理器与编译器***
- **代码移动**
  - 减少计算执行的频率
    - 如果它总是产生相同的结果
    - 将代码从循环中移出

```
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];
```

编译器生成的代码移动 (-O1)

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

```
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
```

复杂运算简化Reduction in Strength

- 用更简单的方法替换昂贵的操作
- 移位，而不是乘法或除法

  `16*x -->  x << 4`

  - 基于机器的实际运用
  - 取决于乘法或除法指令的成本
    - 在Intel Nehalem,整数乘需要3个CPU周期

```
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;
}
```

共享公用子表达式

- 重用表达式的一部分
- GCC 使用 –O1 做到这个

```
/* 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;
```

3 乘法: i*n, (i–1)*n, (i+1)*n

```
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;
```

1 乘法: 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
```

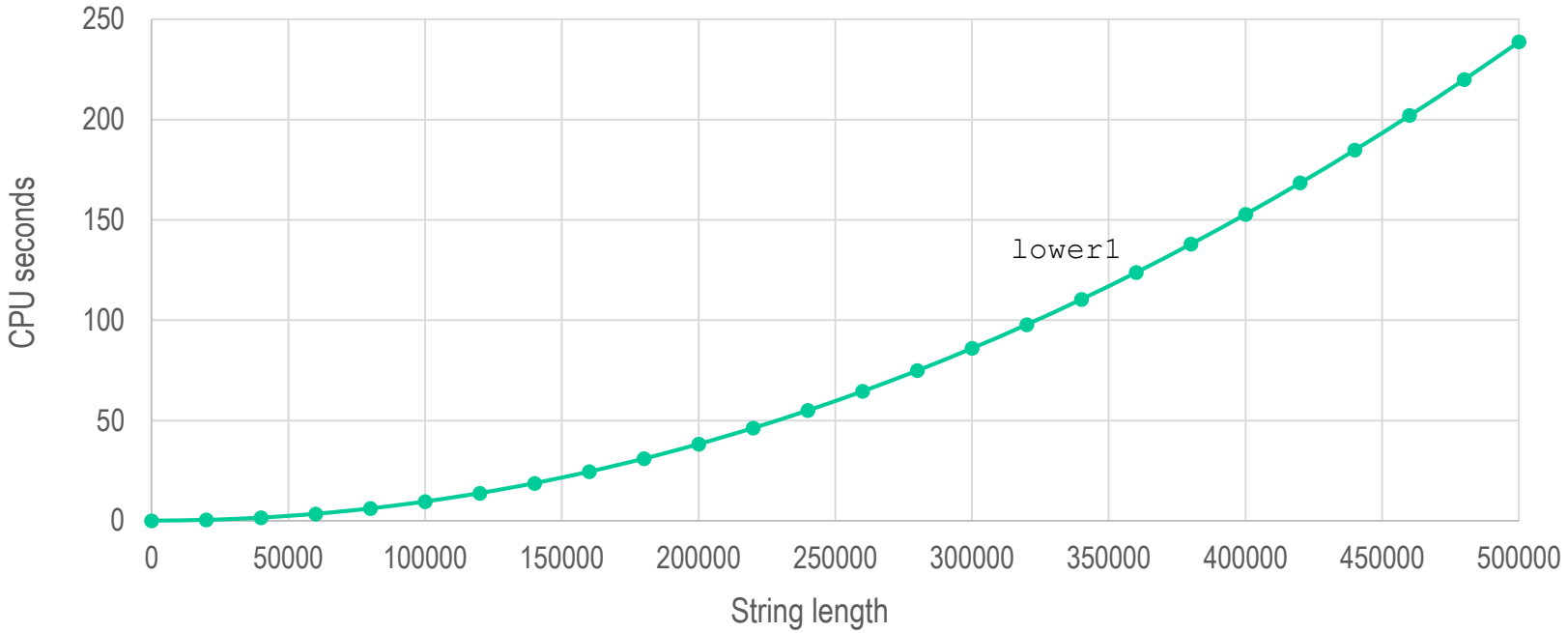
妨碍优化的障碍#1: 函数调用

- 将字符串转换为小写的函数

```
void lower1(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

小写转换性能

- 当字符串长度双倍时，时间增加了四倍
- 二次方（平方）的性能Quadratic performance



把循环loop变成 Goto形式

```
void lower1(char *s)
{
   size_t i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- strlen每次循环都要重复执行

调用Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- **Strlen 性能**
  - 确定字符串长度的惟一方法是扫描它的整个长度，查找null字符
- **整体性能，长度为N的字符串**
  - N 次调用 strlen
  - 需要时间 N, N-1, N-2, …, 1
  - 整体 $O(N^2)$ 性能

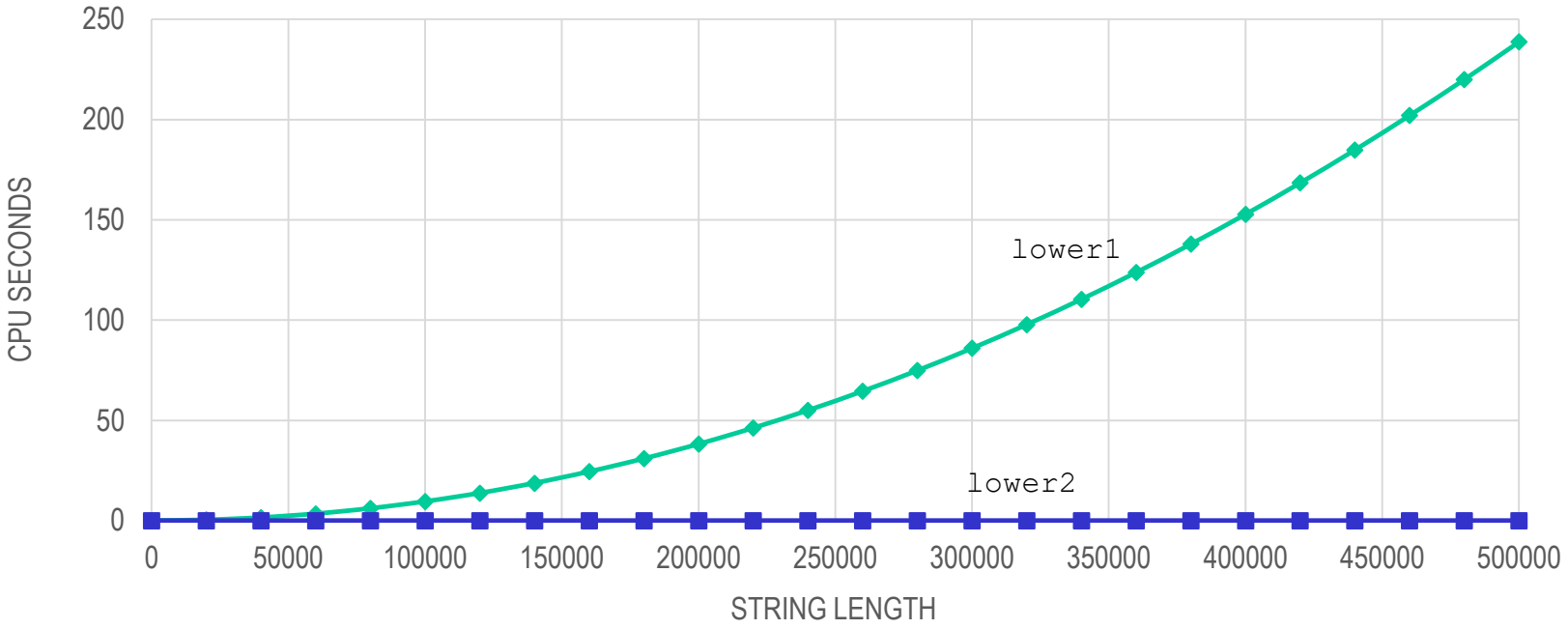
提高性能

```
void lower2(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- 把调用 `strlen` 移动到循环外
- 因为结果（strlen的长度）不会在迭代中变化
- 代码移动的形式

Lower 小写 转换效率

- 字符串长度2倍时，时间也2倍
- lower2 的线性效率



妨碍优化的因素: 函数调用

- ***为什么编译器不能将strlen从内部循环中移出呢?***
  - 函数可能有副作用
    - 每次都改变了全局变量/状态
  - 对于给定的参数，函数可能不会返回相同的值
    - 依赖于全局状态/变量的其他部分
    - 函数 `lower` 可能与 `strlen` 相互作用
- **Warning:**
  - 编译器将函数调用视为黑盒
  - 在函数附近进行弱优化
- **补救措施Remedies:**
  - 使用 inline 内联函数
    - GCC 用 –O1 做到这个
      - 在单一的文件
  - 自己做代码移动

```
size_t lencnt = 0;
size_t strlen(const char
*s)
{
    size_t length = 0;
    while (*s != '\0') {
      s++; length++;
    }
    lencnt += length;
    return length;
}
```

内存相关妨碍优化

```
/* 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];
    }
}
```

```
# 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
```

- 代码每次循环都更新 `b[i]`
- 为什么编译器不能优化这个?

存储器别名使用（Memory Aliasing）

```
/* 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];
    }
}
```

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

double B[3] = A+3;

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

Value of B:

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

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

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

- 代码每次循环都更新 `b[i]`
- 必须考虑这些更新会影响程序行为的可能性

消除别名使用

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
            double val = 0;
            for (j = 0; j < n; j++)
               val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
    addsd   (%rdi), %xmm0 # FP load + add
    addq    $8, %rdi
    cmpq    %rax, %rdi
    jne     .L10
```

- 不需要存储中间结果

妨碍优化的因素: 存储器别名使用

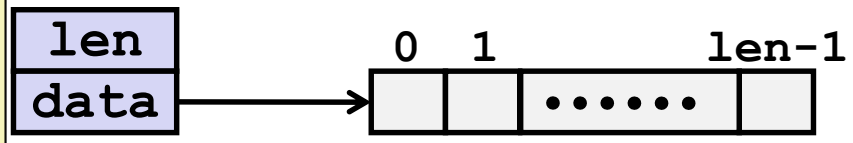
- **别名使用**
  - 两个不同的内存引用指向同一个位置
  - C中很容易发生
    - 允许做地址运算
    - 直接访问存储结构
  - 通常引入局部变量
    - 在循环中累积
    - 编译器不检查存储器别名使用情况

利用指令级并行

- **需要理解现代处理器的设计**
  - 硬件可以并行执行多个指令
- **性能受限于数据相关性**
- **简单的转换即可以带来显著的性能改进**
  - 编译器通常无法进行这些转换
  - 浮点运算缺乏结合性和可分配性

程序示例: 向量数据结构

```
/* data structure for vectors */
typedef struct{
   size_t len;
   data_t *data;
} vec;
```



- **数据类型**
  - 使用 data_t的不同声明
  - int
  - long
  - float
  - double

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

向量元素的计算

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

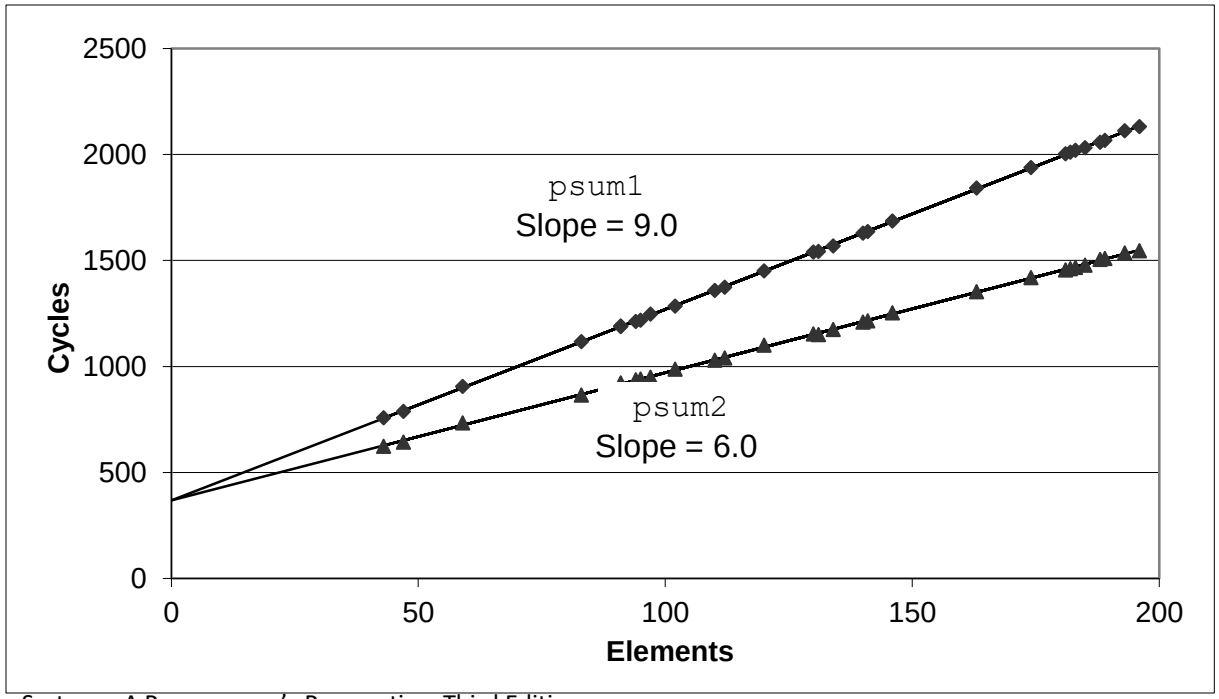
合并运算：将向量所有元素合并成一个值。

合并运算OP：和或积

- **数据类型**
  - 使用 `data_t`的不同声明
  - `int`
  - `long`
  - `float`
  - `double`
- **操作**
  - 使用 `OP` and `IDENT` 的不同定义
  - `+` / `0`
  - `*` / `1`

程序性能度量标准——每个元素的周期数（Cycles Per Element, CPE）

- 表示向量或列表操作性能的一种方便的度量方法（重复计算）
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + 额外开销
  - CPE 是斜率



测试样例的性能

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

计算向量元素的和或积

方法	Integer		Double FP	
操作 OP	+	*	+	*
Combine1 未优化	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

基本优化

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- **把函数vec_length移到循环外**
- 避免每个循环的边界检查
- 结果累积在临时变量

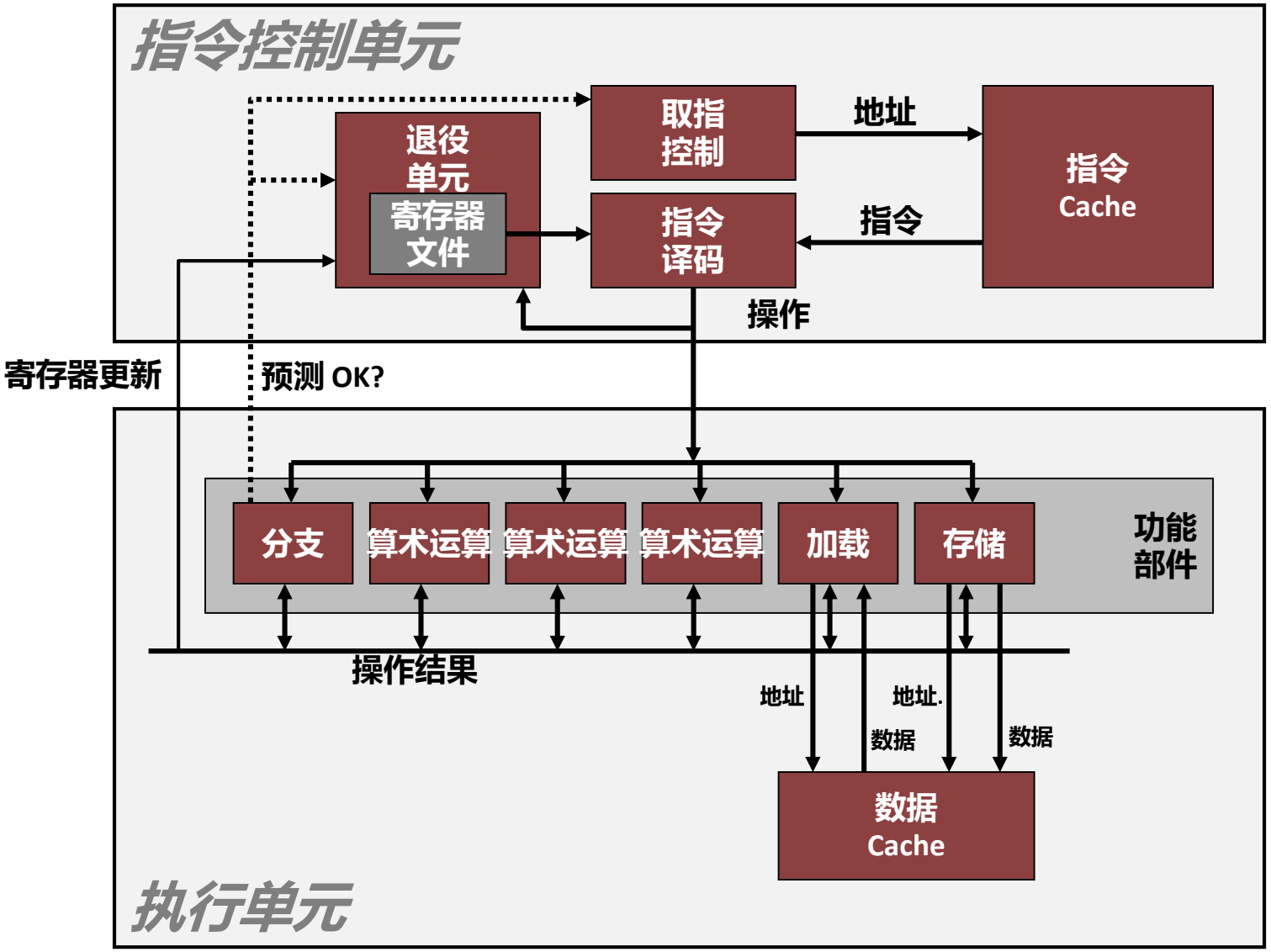
基本优化的效果

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

方法	Integer		Double FP	
操作OP	+	*	+	*
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- 消除循环中的额外开销

现代CPU设计

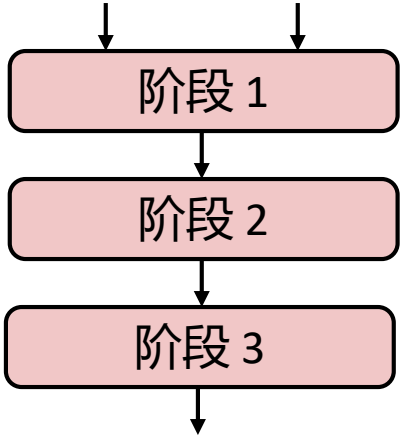

指令控制单元
退役
单元
寄存器
文件
取指
控制
地址
指令
Cache
指令
译码
指令
操作
寄存器更新
预测 OK?
分支
算术运算
算术运算
算术运算
加载
存储
功能
部件
操作结果
地址
地址.
数据
数据
数据
Cache
执行单元

超标量处理器

- 定义: 一个时钟周期执行多条指令. 这些指令是从一个连续的指令流获取的，通常被动态调度的.
- 好处: 不需要编程的努力,超标量处理器可以利用大多数程序所具有的指令级并行性
- 大多数现代的cpu都是超标量
- Intel: 从Pentium (1993)起

流水线功能单元

```
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
阶段 1	a*b	a*c			p1*p2		
阶段 2		a*b	a*c			p1*p2	
阶段 3			a*b	a*c			p1*p2

- 把计算分解为多个阶段
- 一个阶段又一个阶段地通过各部件计算
- 一旦值传送给阶段 i+1，阶段 i 就能开始新的计算，
- 例如, 即使每个乘法需要3个周期，在7个周期里完成3个乘法

Haswell 架构的Intel CPU

- 8 个功能单元—P359
- **可并行执行多条指令**

  2 个加载，带地址计算
  1个存储，带地址计算
  4 个整数运算
  2 个浮点乘法运算
  1 个浮点加法
  1 个浮点除法
- **某些指令 > 1 周期,但能够被流水**

指令	*延迟Latency*	*周期/发射*
Load / Store	4	1
Integer 乘法	3	1
Integer/Long 除法	**3-30**	**3-30**
Single/Double FP 乘法	5	1
Single/Double FP 加法	3	1
Single/Double FP 除法	**3-15**	**3-15**

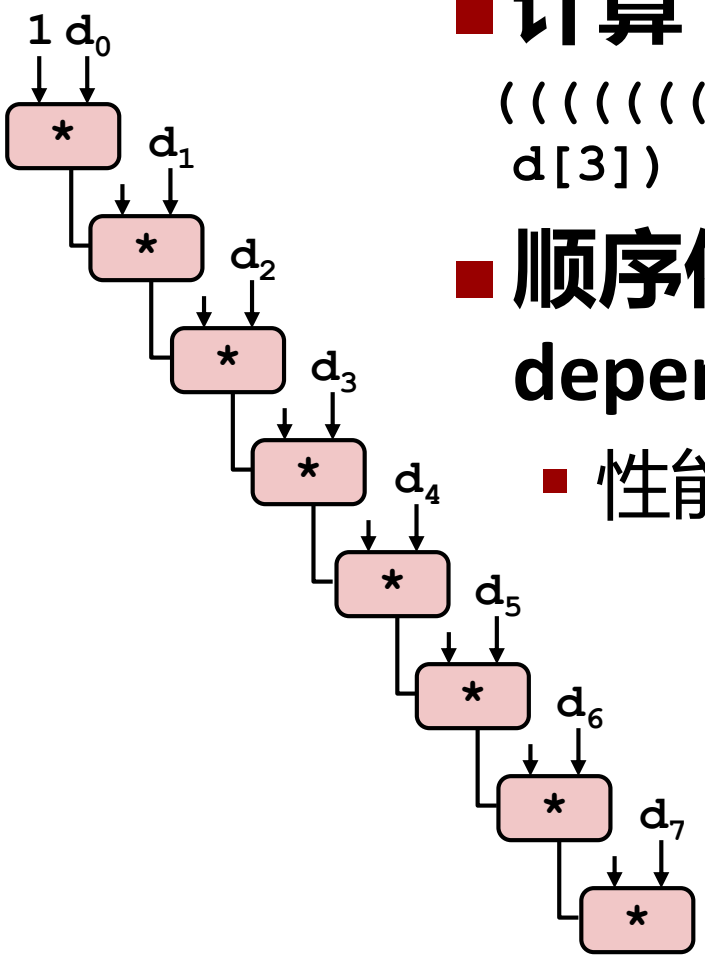
Combine4的x86-64 编译

- 内循环(当做: 整数乘法)

```
.L519:                                  # Loop:
  imull  (%rax,%rdx,4), %ecx            # t = t * d[i]
  addq   $1, %rdx                       # i++
  cmpq   %rdx, %rbp                     # Compare length:i
  jg     .L519                          # If >, goto Loop
```

方法	Integer		Double FP	
操作	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

Combine4 = 串行计算(操作OP = *)



- 计算 (长度=8)

```
((((((((1 * d[0]) * d[1]) * d[2]) *
 d[3]) * d[4]) * d[5]) * d[6]) * d[7])
```

- 顺序依赖性Sequential dependence
  - 性能: 由OP的延迟决定

循环展开Loop Unrolling 2x1

```
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;
}
```

- 每个循环 运行 2倍的更有用的工作

循环展开的效果

方法	Integer		Double FP	
操作	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
2x1循环展开	1.01	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

- **对整数 + 有帮助**
  - 达到延迟界限
- **其他没有改进. *Why?***
  - 仍然是顺序依赖

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

带重组Reassociation的循环展开 (2x1a)

```
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];
```

- 能改变运算结果吗?
- 是的，对 FP浮点数. *Why?*

重组的效果

方法	Integer		Double FP	
操作OP	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
循环展开 2x1	1.01	3.01	3.01	5.01
循环展开 2x1a	1.01	1.51	1.51	2.51
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

- **接近 2倍的速度提升：** Int *, FP +, FP *
  - 原因: 打破了顺序依赖

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

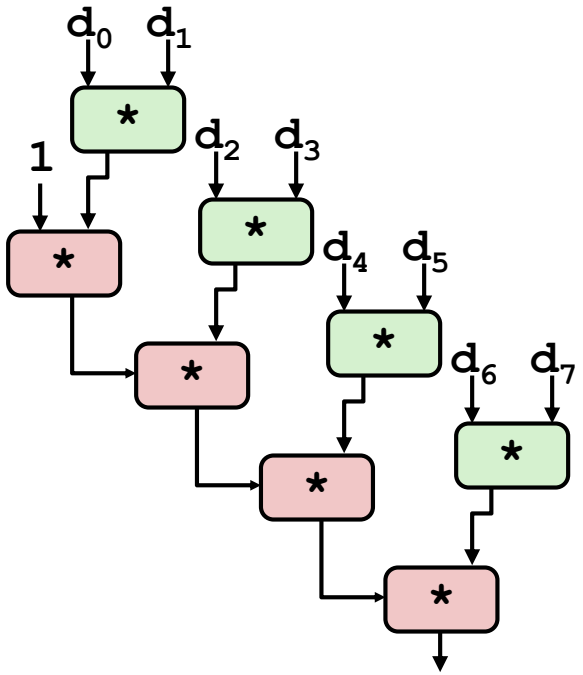
  - 为什么? (下一页)

4 个整数加法功能单元
2 个加载功能单元

2个浮点乘法功能单元
2个浮点加载功能单元

重组的计算

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



- **什么改变了:**
  - 下一个循环的操作可以早一些开始 (没有依赖性)
- **整体性能**
  - N 个元素, 每个操作D 个周期延迟
  - (N/2+1)*D cycles:
    CPE = D/2

循环展开：使用分别的累加器 (2x2)

```
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;
}
```

- 重组的不同形式

分别累加的效果

方法	Integer		Double FP	
操作	+	*	+	*
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
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

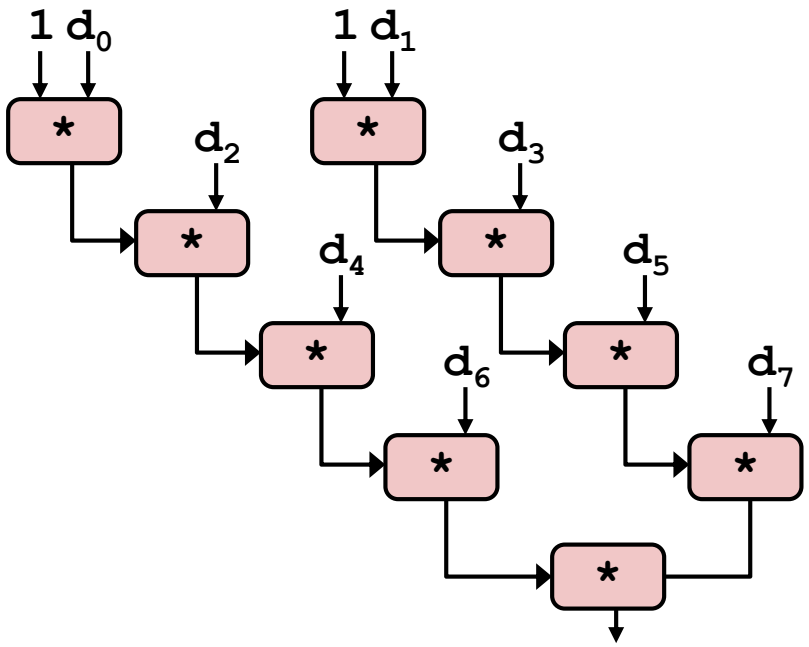
- 整数 加 + 使用 了两个加载单元

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2倍速度提升 ： Int *, FP +, FP *

分离的累加器

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **什么改变了:**
  - 两个独立操作的“流水”
- **整体性能**
  - N 个元素, 每个操作 D个周期延迟
  - 应为 (N/2+1)*D cycles:
    CPE = D/2
  - CPE与预测匹配!

What Now?

循环展开 & 累加

- **设想Idea**
  - 能循环展开到任一程度 L吗?
  - 能够并行累加K个结果吗? Can accumulate K results in parallel
  - L 一定 要 乘以 K

- **限制**
  - 效果/收益递减 Diminishing returns
    - 不能超出执行单元的吞吐量限制
  - 片段的 大的经常开销 Large overhead for short lengths
    - 顺序地完成循环

循环展开 & 累加: Double *

- 案例
  - Intel Haswell
  - Double FP 乘法
  - 延迟界限: 5.00. 吞吐量界限: 0.50

FP *	循环展开因子 L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

循环展开 & 累加: Int +

- 案例
  - Intel Haswell
  - Integer 加法
  - 延迟界限: 1.00. 吞吐量界限: 1.00

FP *	循环展开因子 L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

可达性能

方法	Integer		Double FP	
操作	+	*	+	*
最好	0.54	1.01	1.01	0.52
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

- 只受功能单元吞吐量的限制
- 比原始的、未优化的代码提高了42倍（22.68/0.54）

获得高性能

- **好的编译器**
- **别做傻事**
  - 当心隐藏的算法效率低下
  - 编写编译器友好的代码
    - 小心妨碍优化的因素:
      函数调用 & 存储器引用
  - 仔细观察最里层的循环 (多数工作在那儿完成)

- **机器级优化**
  - 应用指令级并行
  - 避免不可预测的分支
  - 使代码缓存 (后续课程覆盖)