

第五章 优化程序性能

教 师： 史先俊

计算机科学与技术学院

哈尔滨工业大学

要点 --本章PPT与书上内容互为补充

- 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（不同名字指向相同内存）
- 运用指令级并行
- 处理条件

怎么优化源程序？

1. 更快（本课程重点！本章重点！）
2. 更省（存储空间、运行空间）
3. 更美（UI 交互）
4. 更正确（本课程重点！各种条件下）
5. 更可靠（各种条件下的正确性、安全性）
6. 可移植
7. 更强大（功能）
8. 更方便（安装、使用、帮助/导航、可维护）
9. 更规范（格式符合编程规范、接口规范）
10. 更易懂（能读明白、有注释、模块化—清晰简洁）

性能的现实

性能比时间复杂度更重要

- 常数因子也很重要!
 - 代码编写不同，性能会差10倍!
 - 要在多个层次进行优化:
 - 算法、 数据表示/结构、过程、循环（重点优化内循环）
- 优化性能一定要理解“系统”
 - 程序是怎样被编译和执行的---编译器友好的代码
 - 理解编译器的能力与局限性很重要!!!
 - 现代处理器/存储系统是怎么运作的-CPU/RAM友好的代码
 - 怎样测量程序性能、确定“瓶颈” ---**valgrind** gprof
profiler代码剖析程序/**Test Studio/Load Runner**
 - 如何在不破坏代码模块性和通用性的前提下提高性能

优化编译器---编写编译器友好的代码！

- 提供从程序到机器的有效映射
 - 寄存器分配
 - 代码选择与排序（调度）
 - 消除死代码
 - 消除轻微的低效率问题
- 源程序稍变一下，编译器优化方式与性能变化很大
- 不要（通常）提高渐进效率asymptotic efficiency
 - 由程序员来选择最佳的总体算法
 - 大O常常比常数因子更重要，但常数因子也很重要
- 难以克服“优化障碍”
 - 潜在的内存别名使用memory aliasing
 - 潜在的函数副作用
- 利用多线程多核并行，提高程序并行性-降低数据相关

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
 - 通常阻止它进行优化那些只会影响到病态条件下的行为
- 对程序员来说很明显的行为，可能会因语言和编码风格而变得模糊/混乱
 - 如 ,数据的范围可能比变量类型对应的范围更小
- 低级别优化往往降低程序可读性和模块性
 - 程序易出错，难以修改和扩展

优化编译器的局限性

- 大多数分析只在过程范围内进行
 - 在大多数情况下，全程序分析过于昂贵
 - 新版本的GCC在单个文件中进行过程间分析
 - 但是, 不做文件间的代码分析
- 大多数分析都是基于静态信息的
 - 编译器很难预测运行时的输入
- 编译后执行顺序可能不同于他们在程序中出现顺序。
- 当有疑问时，编译器必须是保守的
- 反汇编并分析代码是理解编译器运作的最有效手段
- 修改代码试着控制编译器产生更高效的实现
- 假定编译器很Low，重写程序到编译器的有效代码-试图欺骗编译器产生高效代码。-保证可读性、模块化、可移植性等。性能虽不是最好，但比用汇编好

一般有用的优化

- 不考虑处理器/编译器，您或编译器应该做的优化
- 消除不必要的工作：函数调用、条件测试、内存引用等——不依赖于机器。如全局变量比局部变量快
- 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:
    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:
    rep ; ret                    # done:
```

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、加，替代乘法/除法

$$16 * x \quad \rightarrow \quad x \ll 4$$

- 实际效果依赖于机器
 - 取决于乘法或除法指令的成本
 - Intel Nehalem CPU整数乘需要3个CPU周期
- 识别乘积的顺序 (Recognize sequence of products)
 - 识别产品（编译生成对的机器程序）的顺序

```
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` 选项实现这个优化

```
/* 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: $i*n$, $(i-1)*n$, $(i+1)*n$

```
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
```

```
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: $i*n$

```
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
```

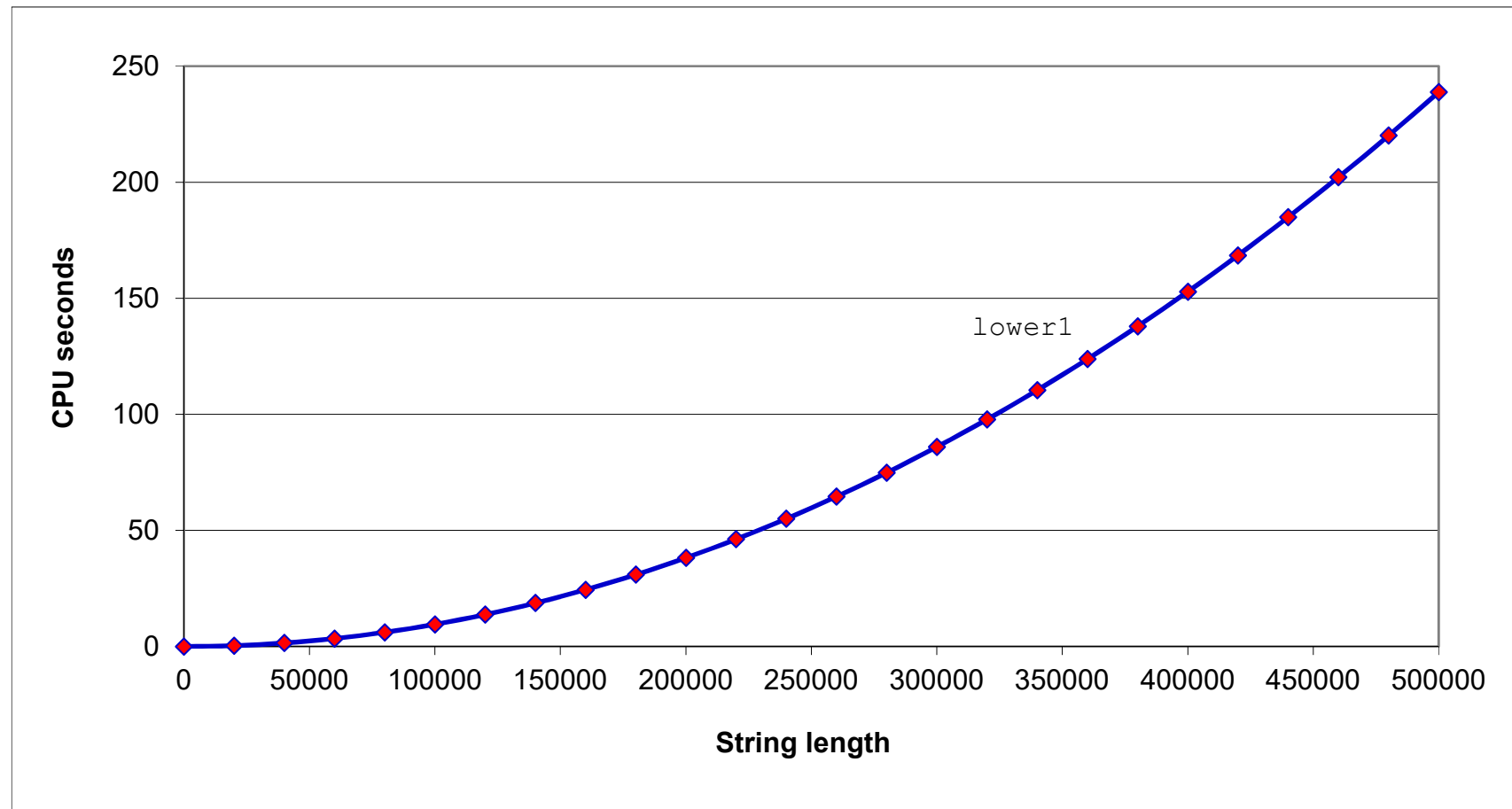
妨碍优化的因素/优化障碍#1: 函数调用

- 程序行为中严重依赖执行环境的方面，程序员要编写容易优化的代码，以帮助编译器。
- 将字符串转换为小写的函数

```
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');
}
```

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



把循环变成 Goto形式--- 类汇编实现

```
void lower(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:
}
```

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

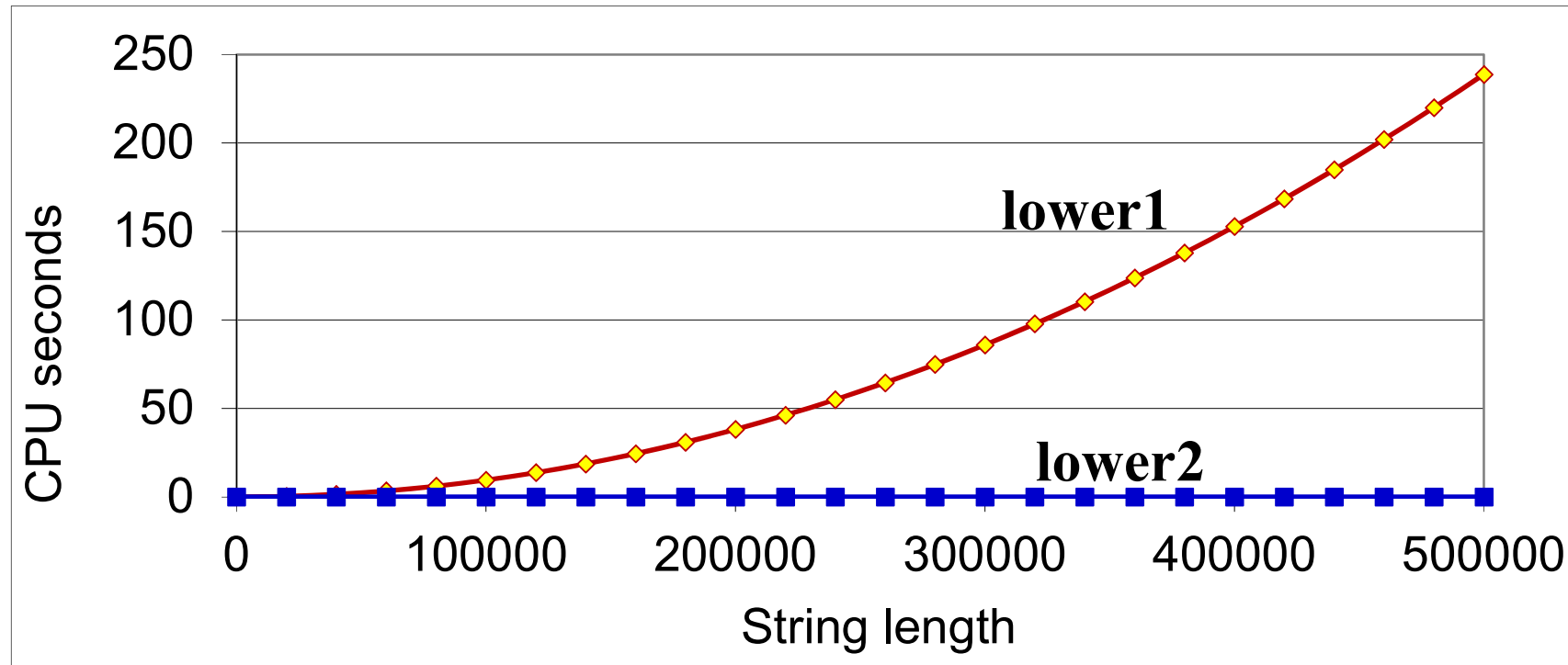
- **strlen**每次循环都要重复执行
- **strlen** 性能
 - 确定字符串长度的唯一方法是扫描它的整个长度，查找null字符
- **整体性能，长度为N的字符串**
 - N 次调用 strlen
 - 整体 $O(N^2)$ 性能

提高性能

```
void lower(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` 移到循环外
- 根据：从一次迭代到另一次迭代时结果不会变化
- ——代码移动的形式

Lower 小写转换的效率



- 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`从内层循环中移出呢?

- 函数可能有副作用
 - 例如: 每次被调用都改变全局变量/状态
- 对于给定的参数, 函数可能返回不同的值
 - 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:

- 使用 `inline` 内联函数
 - 用 `-O1` 时GCC这样做, 但局限于单一文件之内
- 程序员自己做代码移动

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

内存的事

```

/* 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 $b[i]$ —每次都要读出来再加写回
- 为什么编译器不能优化这个?

妨碍优化的因素#2: 内存别名使用

■ 别名使用

- 两个不同的内存引用指向相同的位置
- C很容易发生
 - 因为允许做地址运算
 - 直接访问存储结构
- 编译器不知道函数什么时候被调用，会不会在别处修改了内存，特别是并行化后，改变顺序的优化等。
- 编译器保守的方法是不断的读和写内存，即使这样效率不高
- 养成引入局部变量的习惯
 - 在循环中累积—用寄存器别名替换
 - 告诉编译器不要检查内存别名使用的方法

内存别名使用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];
sum_rows1(A, A+3, 3);

```

- 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

Value of **b**:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

移除 内存别名

```

/* 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;
    }
}

```

- 不需要存储中间结果

sum_rows2 inner loop

```

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

```

利用指令级并行

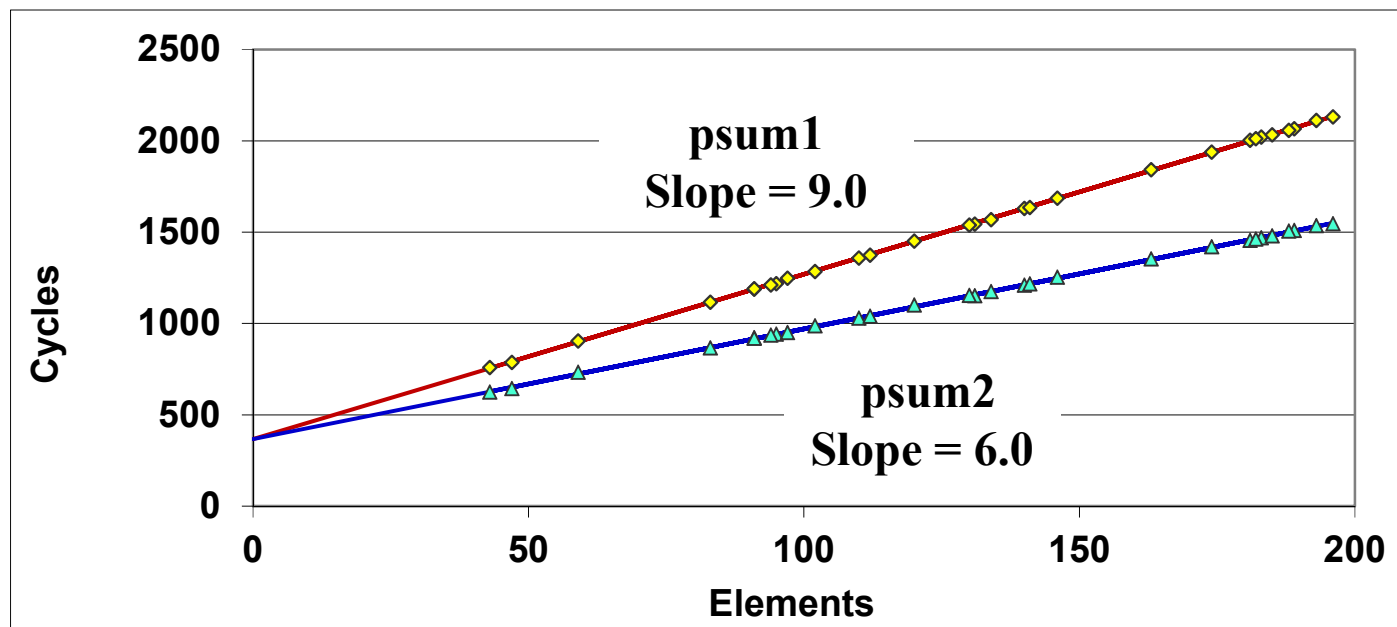
- 需要理解现代处理器的设计
 - 硬件可以并行执行多个指令
- 性能受数据依赖的限制
- 简单的转换可以带来显著的性能改进
 - 编译器通常无法进行这些转换
 - 浮点运算缺乏结合性和可分配性

看2个例子----循环展开

```
1  /* Compute prefix sum of vector a */
2  void psum1(float a[], float p[], long n)
3  {
4      long i;
5      p[0] = a[0];
6      for (i = 1; i < n; i++)
7          p[i] = p[i-1] + a[i];
8  }
9
10 void psum2(float a[], float p[], long n)
11 {
12     long i;
13     p[0] = a[0];
14     for (i = 1; i < n-1; i+=2) {
15         float mid_val = p[i-1] + a[i];
16         p[i] = mid_val;
17         p[i+1] = mid_val + a[i+1];
18     }
19     /* For even n, finish remaining element */
20     if (i < n)
21         p[i] = p[i-1] + a[i];
22 }
```

每个元素的周期数(Cycles Per Element,CPE)

- 表示向量或列表操作的程序性能的方便方式
- $\text{Length} = n$
- In our case: **CPE = cycles per OP**
- $T = \text{CPE} * n + \text{经常开销/费用Overhead}$
 - CPE 是线的斜率slope $T = 368 + \text{Slope} * n$ 怎么减

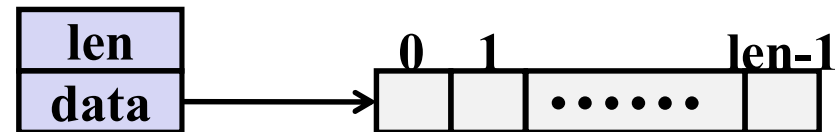


Benchmark例子: 向量的数据类型

```
/* 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;
}
```

Benchmark 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;  
    }  
}
```

计算向量元素的
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 和 IDENT 的不同定义
- + / 0
- * / 1

Benchmark 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

基础/简单优化

```
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
- 用临时/局部变量累积结果

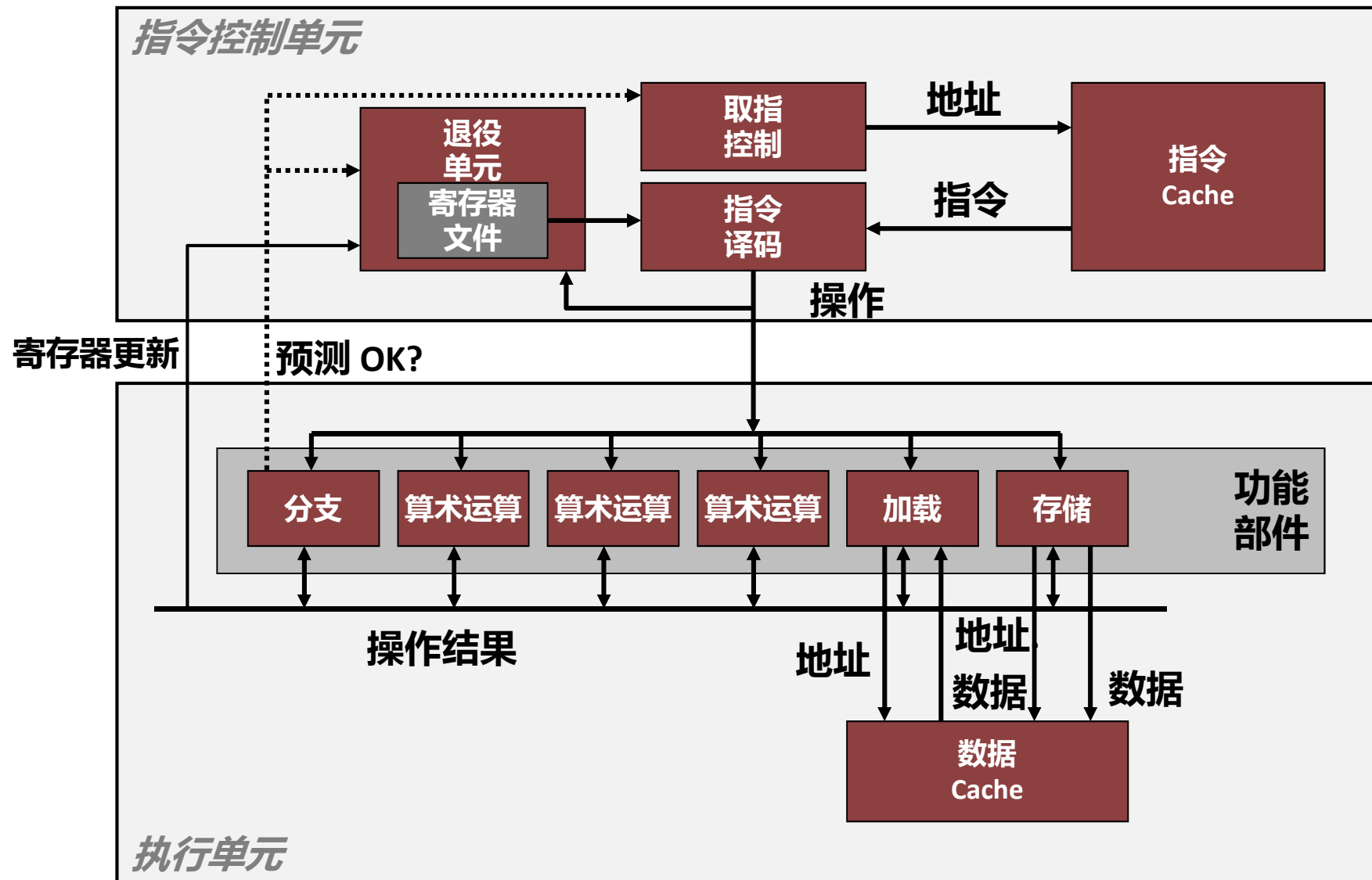
基础/简单优化的效果

```
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 经常开销的来源sources of overhead

现代CPU设计

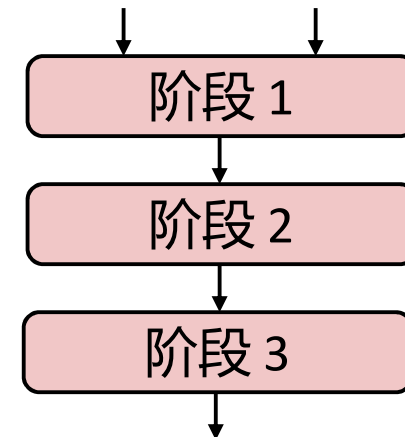


超标量Superscalar处理器

- **定义:** 一个周期执行多条指令. 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.
- 多级流水线: 2、5、7、15.....
- 多流水线/多线程、多核心
- 每类功能部件 ≥ 1 : 如整数加、浮点加等
- 乱序执行: 执行顺序与代码顺序可能不一致
- 分支预测、投机执行: 不确定条件是否正确就执行, 但是结果不放到寄存器或内存 ----性能开销-惩罚
- 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)

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分计算
- 一旦值传送给 $i+1$ ，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 P361

指令

延迟Latency

周期/发射

Load / Store

4

1

Integer 除法

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

0: 整数运算、浮点乘、整数和浮点数除法、分支

1: 整数运算、浮点加、整数乘、浮点乘

2: 加载、地址计算

3: 加载、地址计算

4: 存储

5: 整数运算

6: 整数运算、分支

7: 存储、地址计算

容量: 能够执行该运算的功能单元数

延迟: 完成运算所需要的总clk

发射时间: 连续同类型运算间最小clk

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3 ~ 30	3 ~ 30	1	3 ~ 15	3 ~ 15	1

图 5-12 参考机的操作的延迟、发射时间和容量特性。延迟表明执行实际运算所需要的时钟周期总数，而发射时间表明两次运算之间间隔的最小周期数。容量表明同时能发射多少个这样的操作。除法需要的时间依赖于数据值

```

1  /* Accumulate result in local variable */
2  void combine4(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      data_t *data = get_vec_start(v);
7      data_t acc = IDENT;
8
9      for (i = 0; i < length; i++) {
10         acc = acc OP data[i];
11     }
12     *dest = acc;
13 }

```

Combine4-P355的x86-64 编译

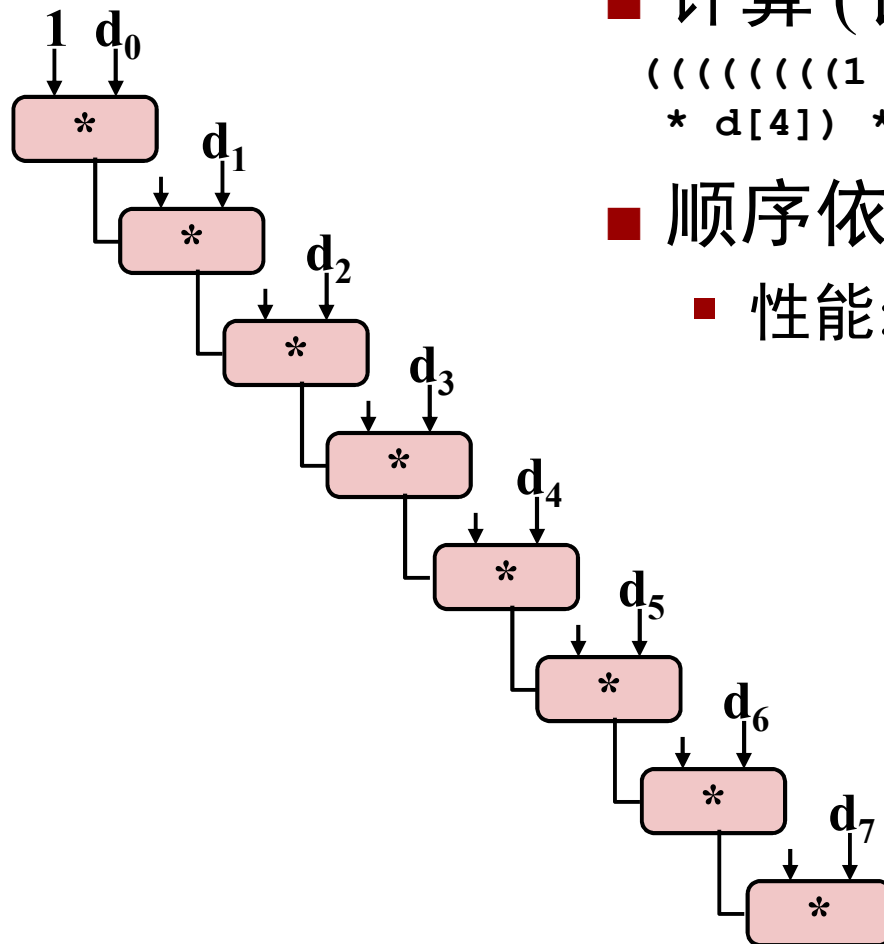
■ 内循环(做整数乘法) $acc = acc \text{ OP } data[i]$

.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

延迟界限：任何必须按照严格顺序完成合并运算的函数所需的最小CPE值

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

$$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$$

- 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

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

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

- 这能改变运算结果吗?
- 是的, 对 FP浮点数. *Why?*

重组的效果/影响

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

吞吐量界限：CPE的最小界限

■ 接近 2 倍的速度提升：Int *, FP +, FP *

- 原因：打破了顺序依赖

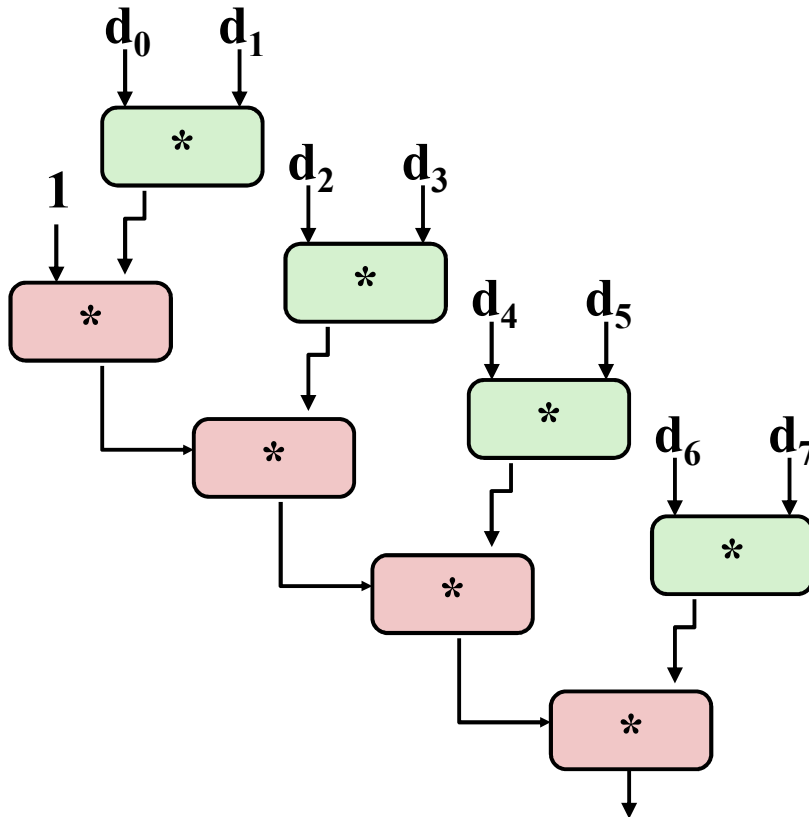
```
x = x OP (d[i] OP d[i+1]);
```

- 为何是这样? (下一页)

2个浮点乘法功能单元
2个浮点加载功能单元

4个整数加法功能单元
2个加载功能单元

重组的计算

$$x = x \text{ OP } (d[i] \text{ OP } d[i+1]);$$


■ 什么改变了:

- 下一个循环的操作可以早一些开始 (没有依赖)

■ 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：使用分离的累加器 (2x2)

■ 重组的不同形式

```
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;
}
```

分离的累加器的效果

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

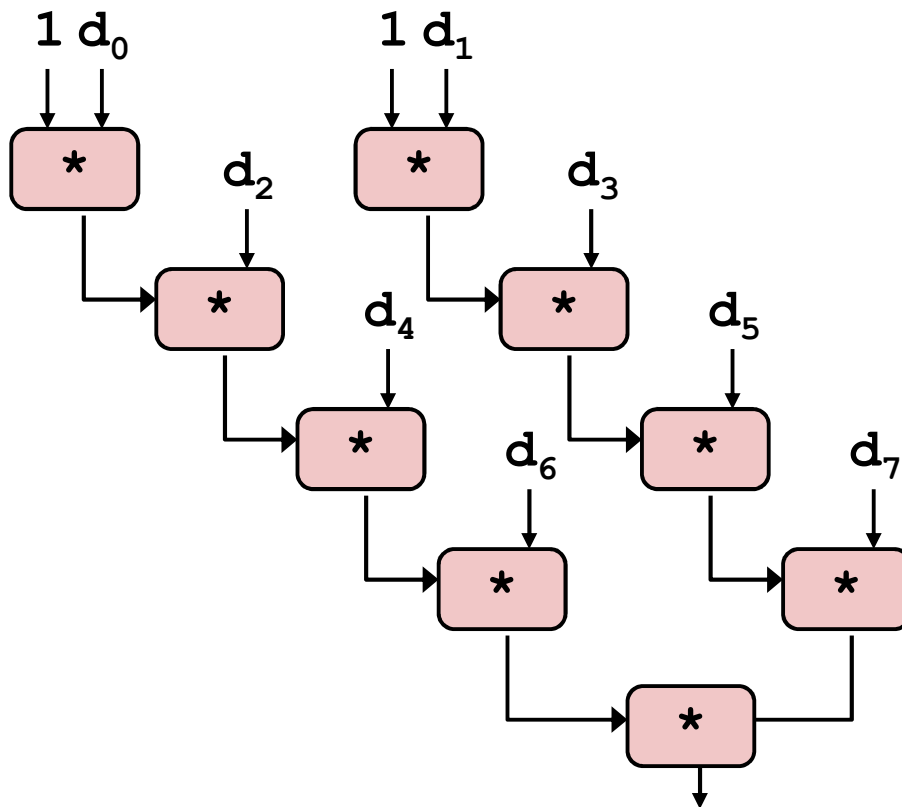
- 整数 加 + 使用 了两个加载单元

```
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:

- 两个独立的操作的“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?

```

1  /* 2 x 1 loop unrolling */
2  void combine5(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      long limit = length-1;
7      data_t *data = get_vec_start(v);
8      data_t acc = IDENT;
9
10     /* Combine 2 elements at a time */
11     for (i = 0; i < limit; i+=2) {
12         acc = (acc OP data[i]) OP data[i+1];
13     }
14
15     /* Finish any remaining elements */
16     for (; i < length; i++) {
17         acc = acc OP data[i];
18     }
19     *dest = acc;
20 }

```

图 5-16 使用 2×1 循环展开。这种变换能减小循环开销的影响

```

1  /* 2 x 2 loop unrolling */
2  void combine6(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      long limit = length-1;
7      data_t *data = get_vec_start(v);
8      data_t acc0 = IDENT;
9      data_t acc1 = IDENT;
10
11     /* Combine 2 elements at a time */
12     for (i = 0; i < limit; i+=2) {
13         acc0 = acc0 OP data[i];
14         acc1 = acc1 OP data[i+1];
15     }
16
17     /* Finish any remaining elements */
18     for (; i < length; i++) {
19         acc0 = acc0 OP data[i];
20     }
21     *dest = acc0 OP acc1;
22 }

```

图 5-21 运用 2×2 循环展开。通过维护多个累积变量，这种方法利用了多个功能单元以及它们的流水线能力

循环展开 & 累加

- 设想Idea 对元素 i 到 $i+k-1$ 合并运算 补 $0 \sim k-1$
 - 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
- 只有保持能够执行该操作的所有功能单元的流水线都是满的, 程序才能达到这个操作的吞吐量界限。 $K \geq C_{\text{容量}} * L_{\text{延迟}}$
- 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

Accumulators	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

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

Accumulators	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

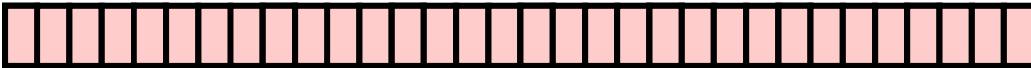






可得到的 性能

- 只受功能单位的吞吐量限制
- 比原始的、未优化的代码提高了42倍

方法	Integer		Double FP	
操作	+	*	+	*
最好Best	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

用 AVX2 编程

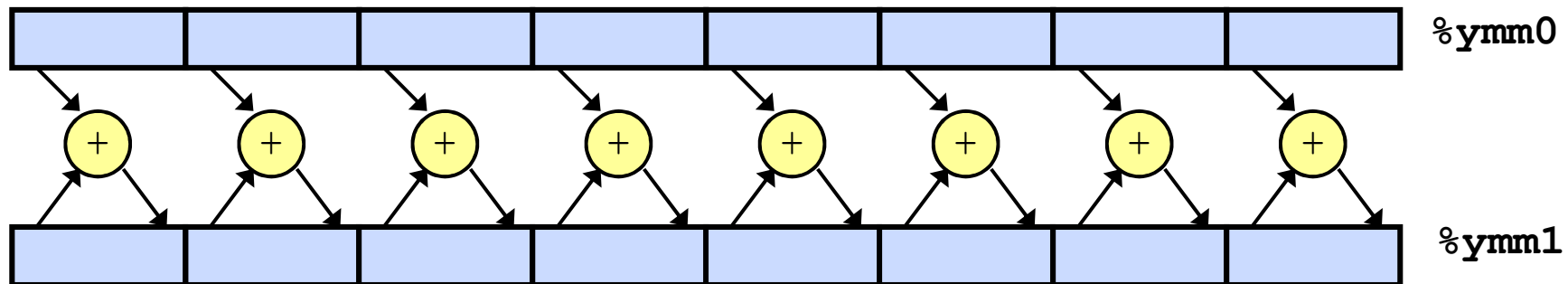
YMM 寄存器：16 个，每个32字节

- 32个单字节整数 
- 16个 16位整数 
- 8 个 32位整数 
- 8 个单精度浮点数 
- 4 个双精度浮点数 
- 1个单精度浮点数 
- 1 个双精度浮点数 

SIMD 操作

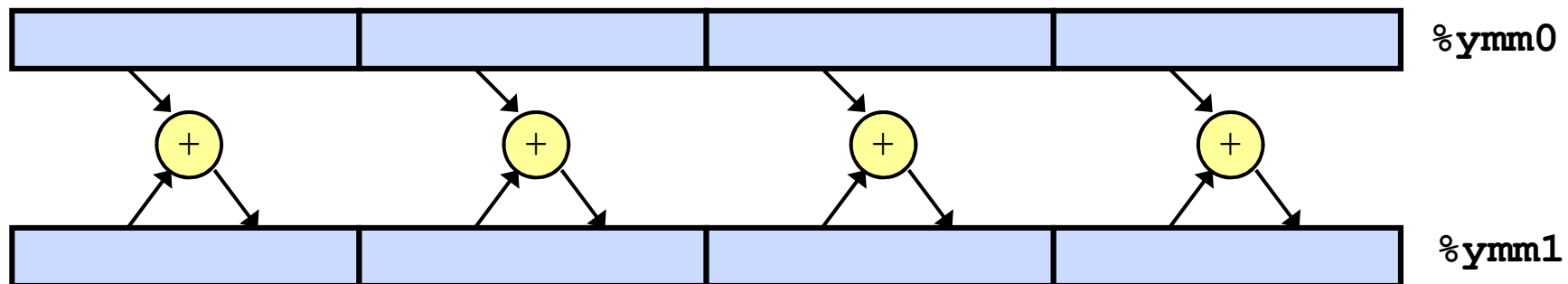
■ SIMD 操作: 单精度

`vaddsd %ymm0, %ymm1, %ymm1`



■ SIMD 操作: 双精度

`vaddpd %ymm0, %ymm1, %ymm1`



使用向量指令

方法	Integer		Double FP	
操作	+	*	+	*
标量 Best	0.54	1.01	1.01	0.52
向量 Best	0.06	0.24	0.25	0.16
延迟界限	0.50	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50
向量 吞吐量界限	0.06	0.12	0.25	0.12

■ 使用AVX 指令

- 多数据元素的并行操作
- 看网络旁注 OPT:SIMD on CS:APP web 页面

分支怎么处理?

■ 挑战

- 在**执行单元**前，**指令控制单元**必须工作好，以生成足够的操作来使EU保持繁忙

```
404663: mov    $0x0,%eax
404668: cmp    (%rdi),%rsi
40466b: jge    404685 ←
40466d: mov    0x8(%rdi),%rax

...

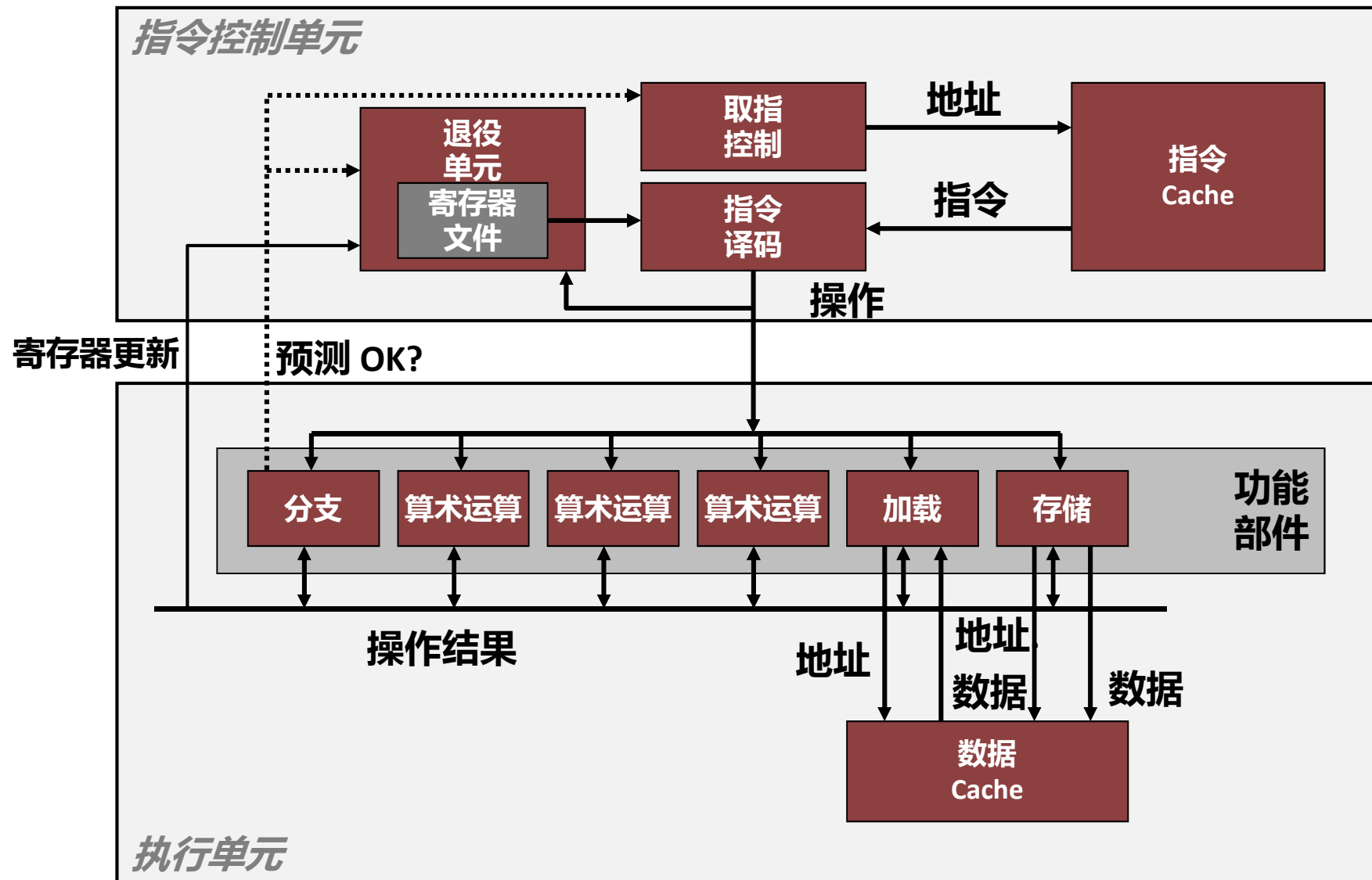
404685: repz retq
```

} Executing

How to continue?

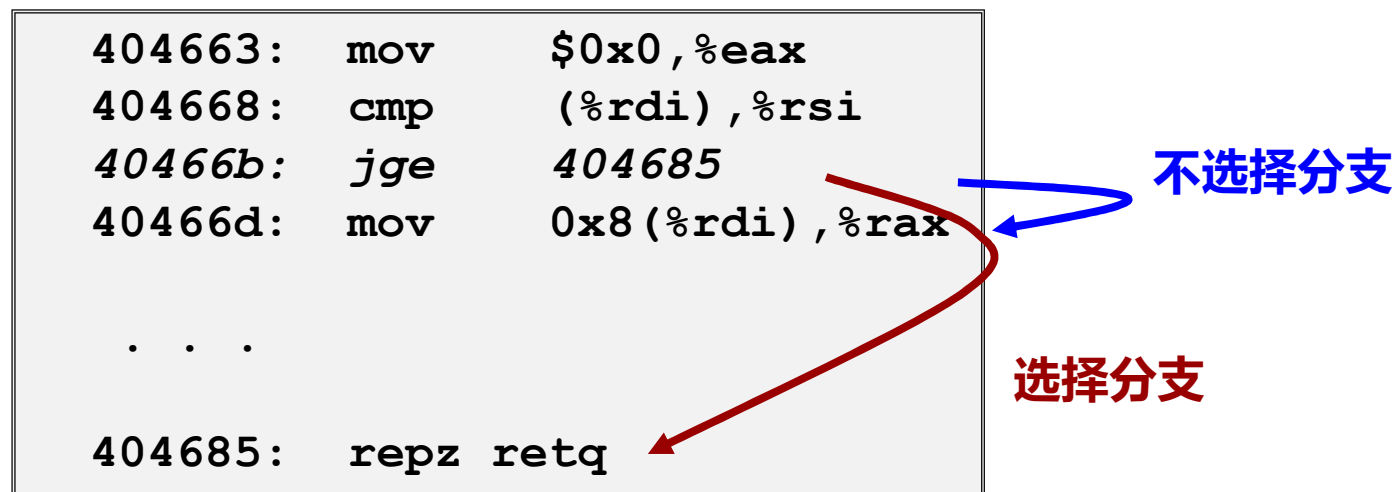
- 遇到条件分支时，无法可靠地确定继续取指的位置

现代CPU设计



分支的结果

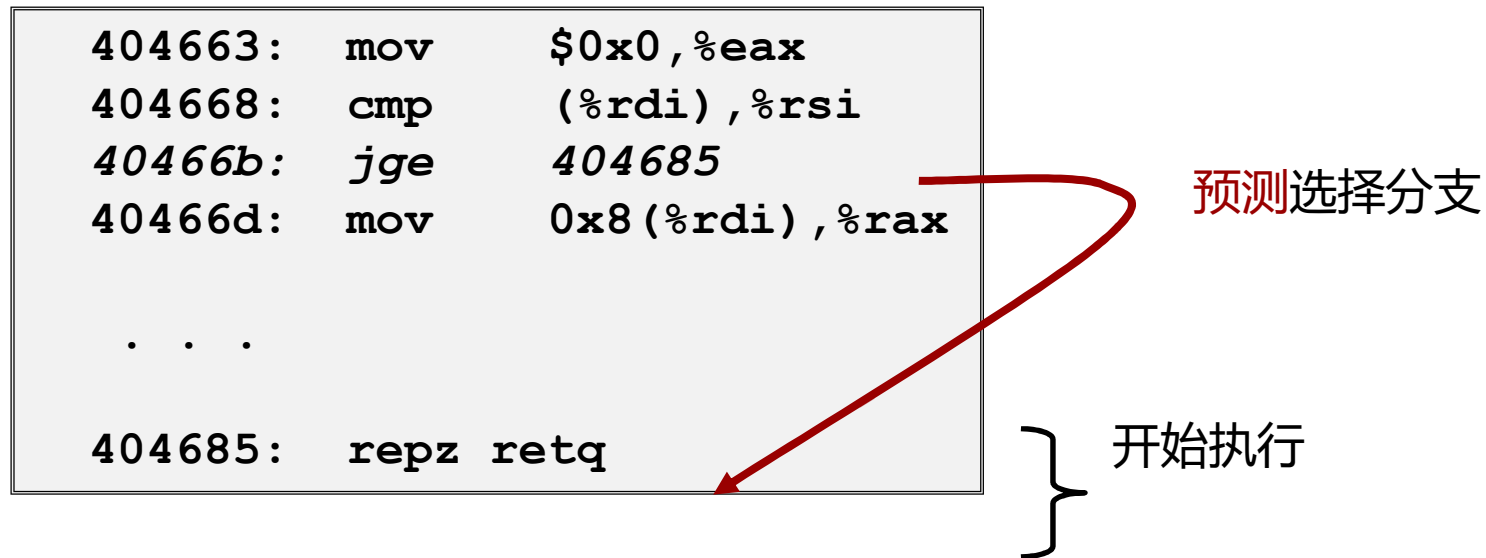
- 当遇到条件分支时，无法确定继续取指的位置
 - 选择分支:将控制转移到分支目标
 - 不选择分支:继续下一个指令
- 直到分支/整数单元的结果确定后 才能解决



分支预测

■ 设想

- 猜测会走哪个分支
- 在预测的位置开始执行指令
 - 但不要真修改寄存器或内存数据



穿过循环的分支预测

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 98

假定
向量长度 = 100

预测选择分支(OK)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 99

预测选择分支(Oops)

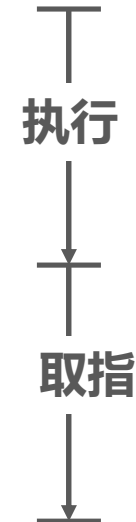
```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 100

读无效位置

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 101



分支错误预测的失效

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029      i = 98
```

假定
向量长度 = 100

预测选择分支(OK)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029      i = 99
```

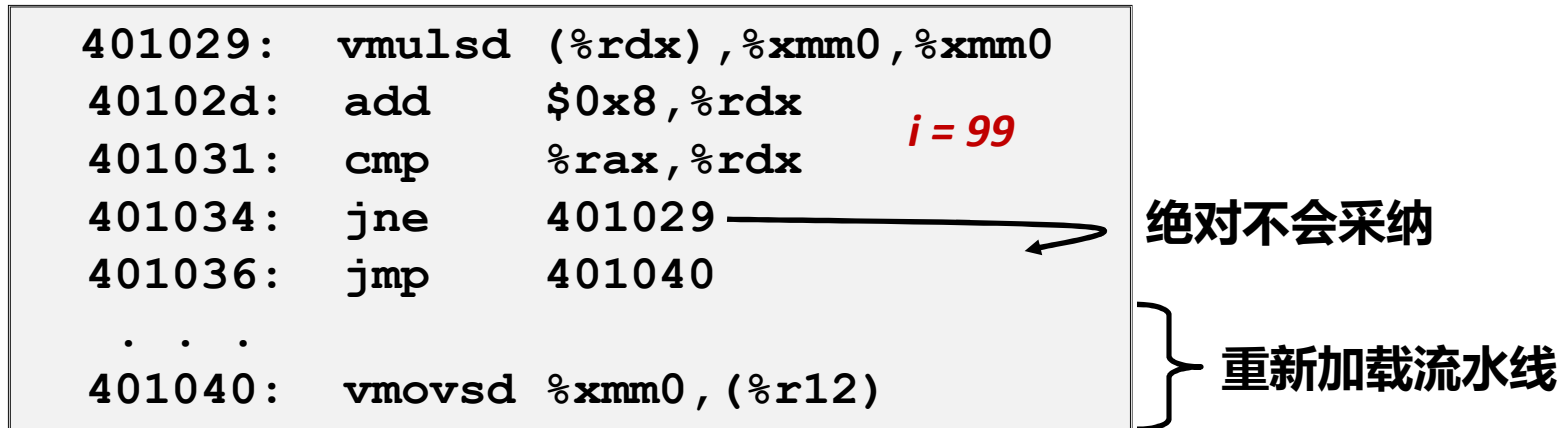
预测选择分支(Oops)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029      i = 100
```

无效

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029      i = 101
```


分支错误预测的恢复



■ 性能开销

- 现代处理器上的多个时钟周期
- 可能是一个主要的性能限制器

怎么办？

- Intel的分支预测：太多了.....哭
- 条件true--分支正确正确率60%
- 距离为负---分支正确正确率80%
- 尽量少用分支！！！！！！能替换吗？
- 编程时提高跳转到预测正确分支的概率
- 用条件传送与条件运算指令
- Arm等嵌入式CPU

获得高性能

- 好的编译器和标志
- 别做傻事
 - 当心隐藏算法效率低下
 - 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
 - 仔细观察最内层的循环 (多数工作在那儿完成)
- 为机器优化代码
 - 利用指令级并行
 - 避免不可预测的分支
 - 使代码能较好地缓存 (在后续的课程介绍)