哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第四章 CPU调度与优化：优化

夏文 副教授

哈尔滨工业大学（深圳）

2021年秋季

Email：xiawen@hit.edu.cn

本章主要内容

- CPU调度基本概念与调度算法
  - 基本概念
  - 调度准则
  - 调度算法
- CPU调度与程序优化（csapp第五章）
  - 为何要优化代码
  - 深入理解现代处理器
  - 常用程序优化方法

本节重点

- 熟练掌握普遍有用优化的方法
  - 代码移动/预先计算
  - 复杂运算简化
  - 公用子表达式的共享
  - 去掉不必要的过程调用
- 优化障碍（两个）
  - 理解函数调用为什么会阻碍优化
  - 理解内存别名的使用为什么会阻碍优化
- 循环展开以及利用指令级并行

如何优化源程序

1. **更快（本节重点！）**
2. **更省（存储空间、运行空间）**
3. **更美（UI 交互）**
4. **更正确（本节重点！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）**

关于性能的现实---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“计算机内部系统”软硬件
  - 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/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
  - 代码的选择与顺序
  - 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
- （通常）不要提高渐进效率(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”（共享全局变量问题）
  - 潜在的函数副作用
  - 潜在的内存别名使用

编译器常用优化（O1）

- 有限的、保守的优化
- 将常量表达式移出循环
- 共享一些公共的表达式
- 用移位代替乘法
- 将条件跳转转成无分支
- 可能内联部分函数

常用优化方法

- 您或编译器应该做的优化
- 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];
```

复杂运算简化

- 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 → x << 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

```
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;//用加来替代乘
}
```

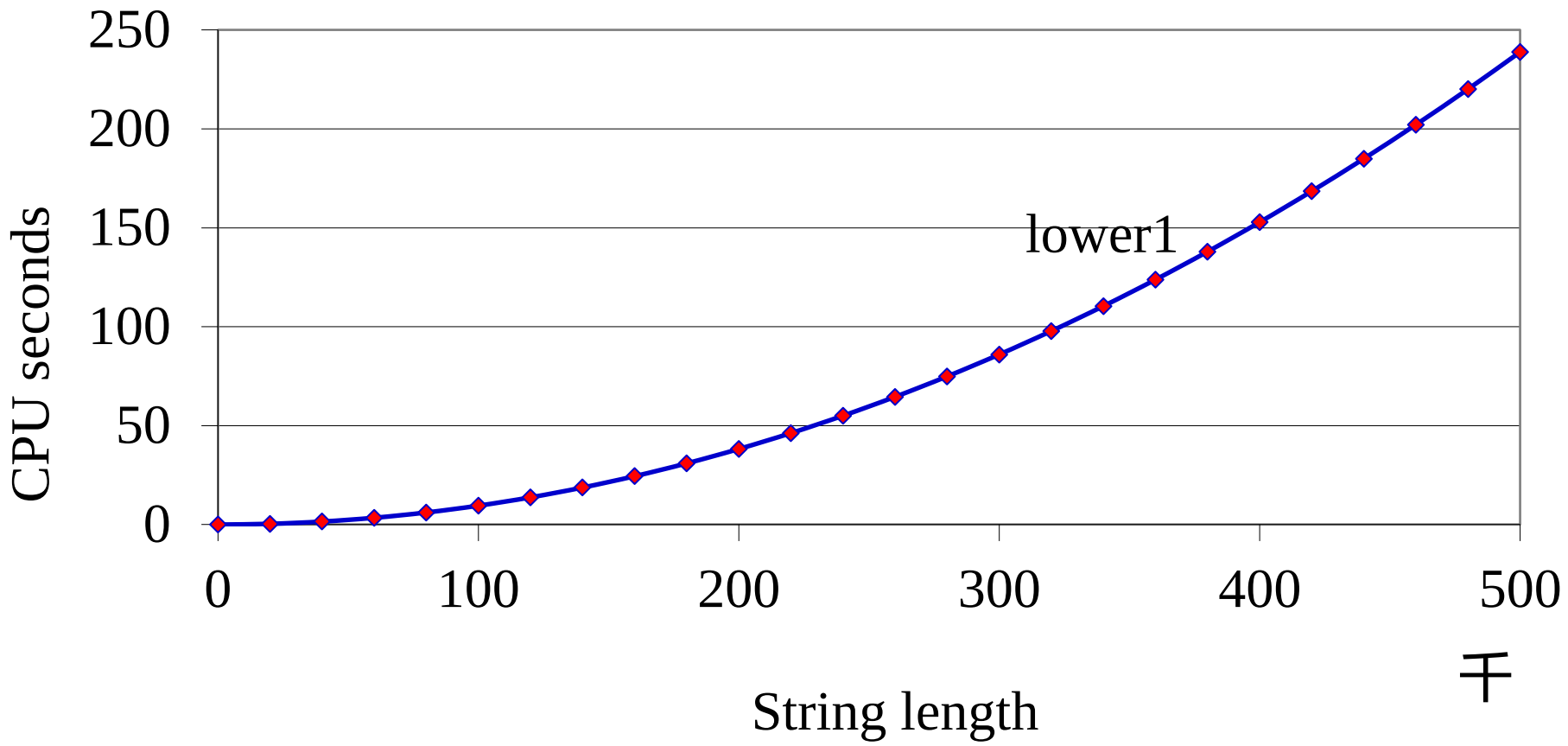
妨碍优化的因素/优化障碍#1: 函数调用

- 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

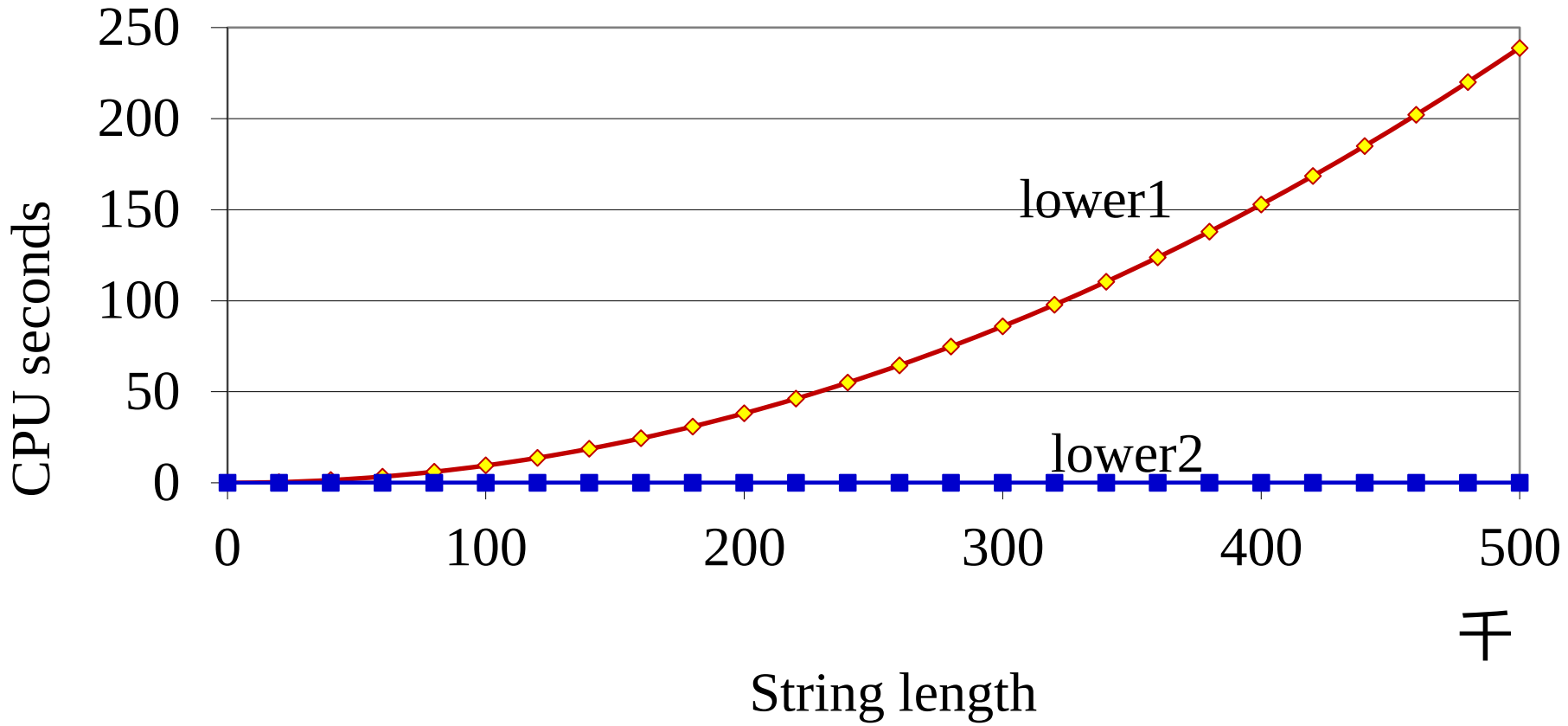


提高性能

- 代码移动：把调用 strlen 移到循环外
- 根据：从一次迭代到另一次迭代，strlen返回结果不会变化

```
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');
}
```

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#1: 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;
}
```

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
  - 养成引入局部变量的习惯
    - ✓在循环中累积

```
/* 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 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;
    }
}
```

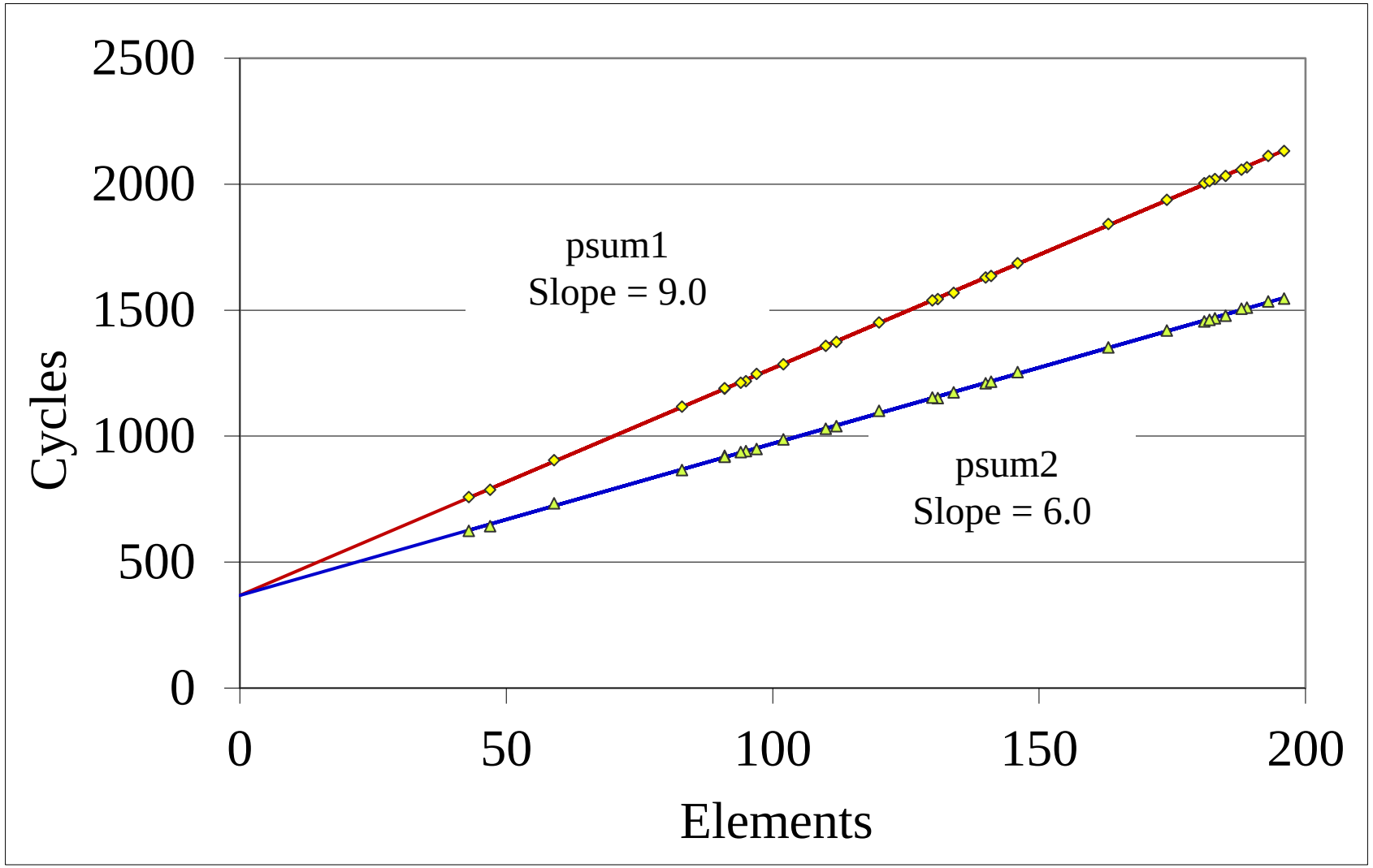
引入程序性能指标CPE

- CPE：每个元素的周期数(Cycles Per Element)
- 表示向量或列表操作的程序性能的方便方式
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + 经常开销(Overhead)
  - CPE 是线的斜率slope

CPE：例子

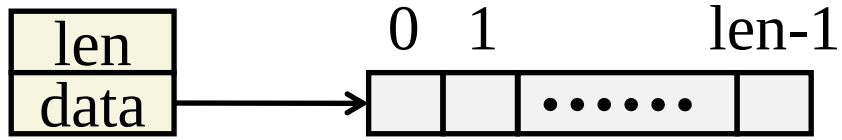
```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long int n) {
   long int i;
   p[0] = a[0];
   for (i = 1; i < n; i++)
      p[i] = p[i-1] + a[i];
}
void psum2(float a[], float p[], long int n) {
   long int i;
   p[0] = a[0];
   for (i = 1; i < n-1; i+=2) {
      float mid_val = p[i-1] + a[i];
      p[i] = mid_val;
      p[i+1] = mid_val + a[i+1]; }
   /* For odd n, finish remaining element */
   if (i < n)
      p[i] = p[i-1] + a[i];
}
```

CPE


2500
2000
1500
1000
500
0
Cycles
psum1
Slope = 9.0
psum2
Slope = 6.0
0
50
100
150
200
Elements

程序示例: 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;
}
```

程序示例的计算

- 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

```
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;
    }
}
```

- 操作
  - 使用 OP 和IDENT 的不同定义
  - + / 0（op为+时IDENT为0）
  - * / 1（op为*时IDENT为1）

程序示例的性能

- 计算向量元素的和或积

```
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;
    }
}
```

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 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;
  }
}
```

```
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;//局部变量累计结果
  for (i = 0; i < length; i++)
    t = t OP d[i];//消除不必要的内存引用
  *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;
}
```

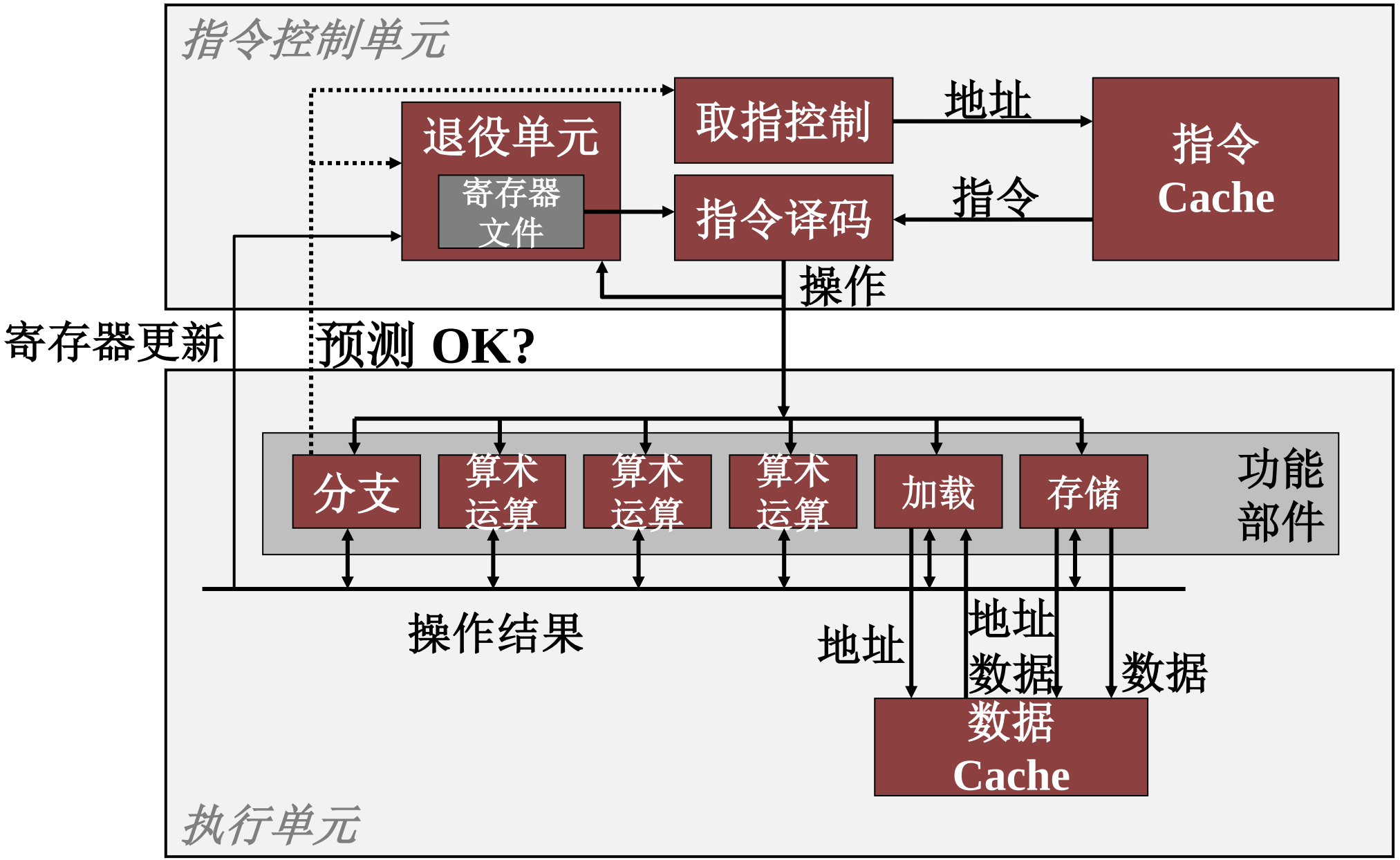
方法	Integer		Double FP	
操作OP	+	*	+	*
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- 消除循环中大量开销的来源（sources of overhead）

利用指令级并行进行优化

- 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

现代CPU设计-超标量


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

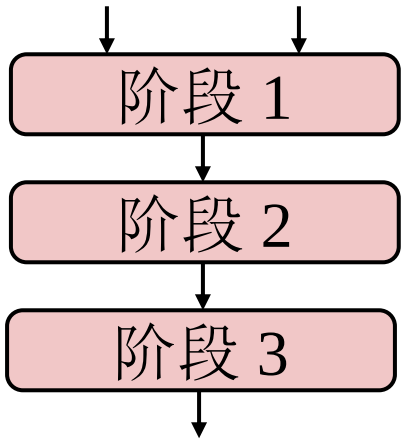
超标量处理器（Superscalar）

- 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。
- 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

- 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，

```
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**

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

- 该CPU包含了8 个功能单元
- 意味着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

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（时钟周期）

发射时间：连续同类型运算间最小clk

Haswell 架构的CPU

- 参考机的操作的延迟、发射时间和容量特性。
  - 延迟表明执行实验运算所需要的时钟周期总数
  - 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

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

```
/* Accumulate result in local variable
*/
void combine4(vec_ptr v, data_t *dest)
{
  long int i;
  long int 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;
}
```

Combine4的x86-64 编译

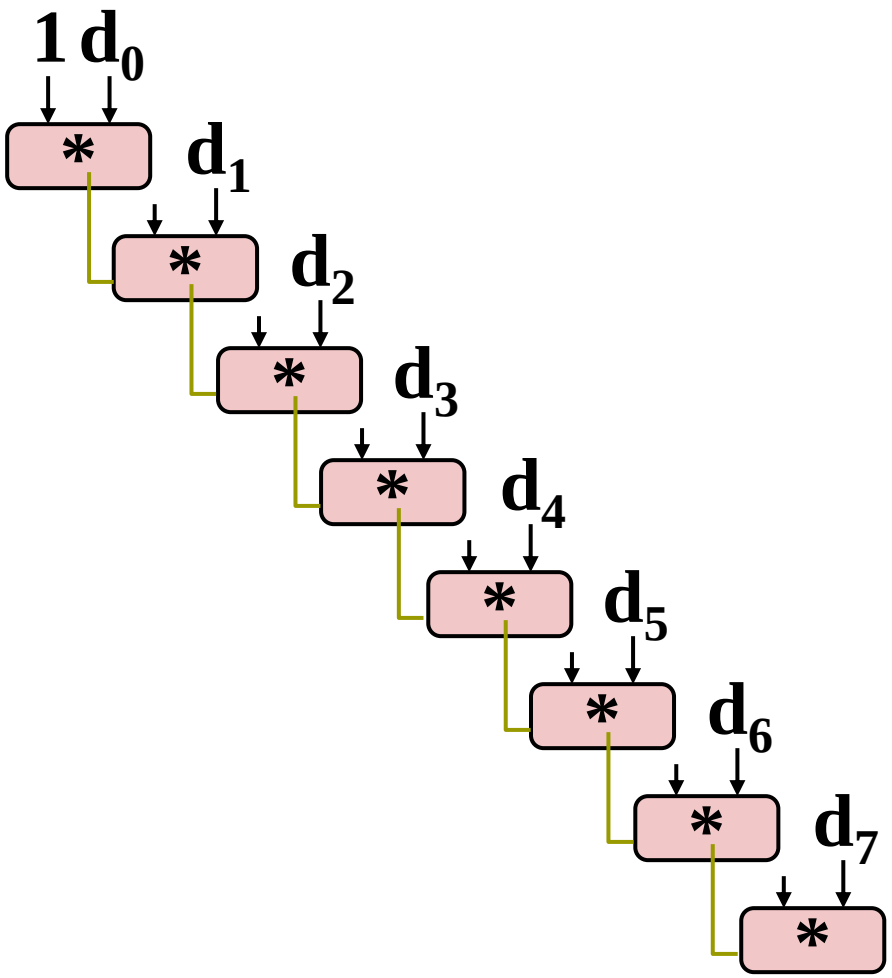
- 内循环(做整数乘法)　acc=acc OP data[i]

```
.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要的最小CPE值（等于单独完成每次操作需要的时钟周期）

Combine4 = 串行计算(操作OP = *)



- 计算 (向量长度=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
2×1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

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

  - 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;
}
```

- 这能改变运算结果吗?
- 是的，对 FP浮点数。

  Why?

Compare to before

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

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

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

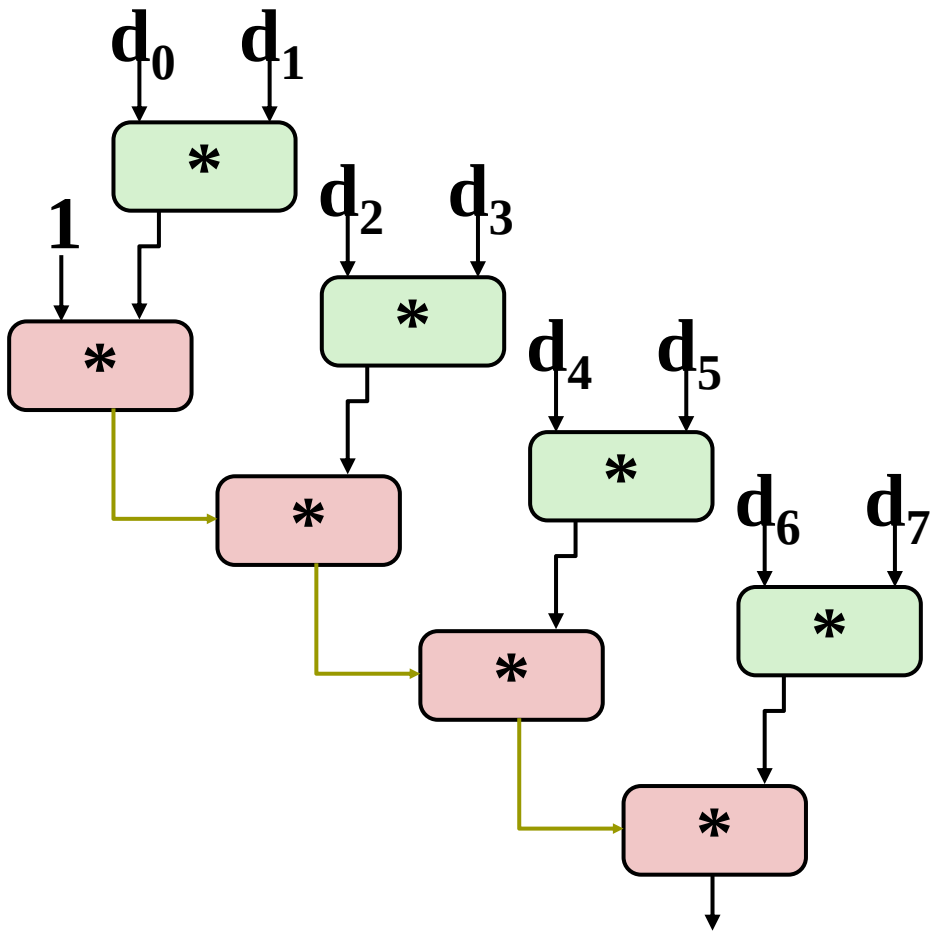
- 接近 2倍的速度提升： Int *, FP +, FP *
  - 原因: 打破了顺序依赖

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

  - 为何是这样? (下一页)

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)
- 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

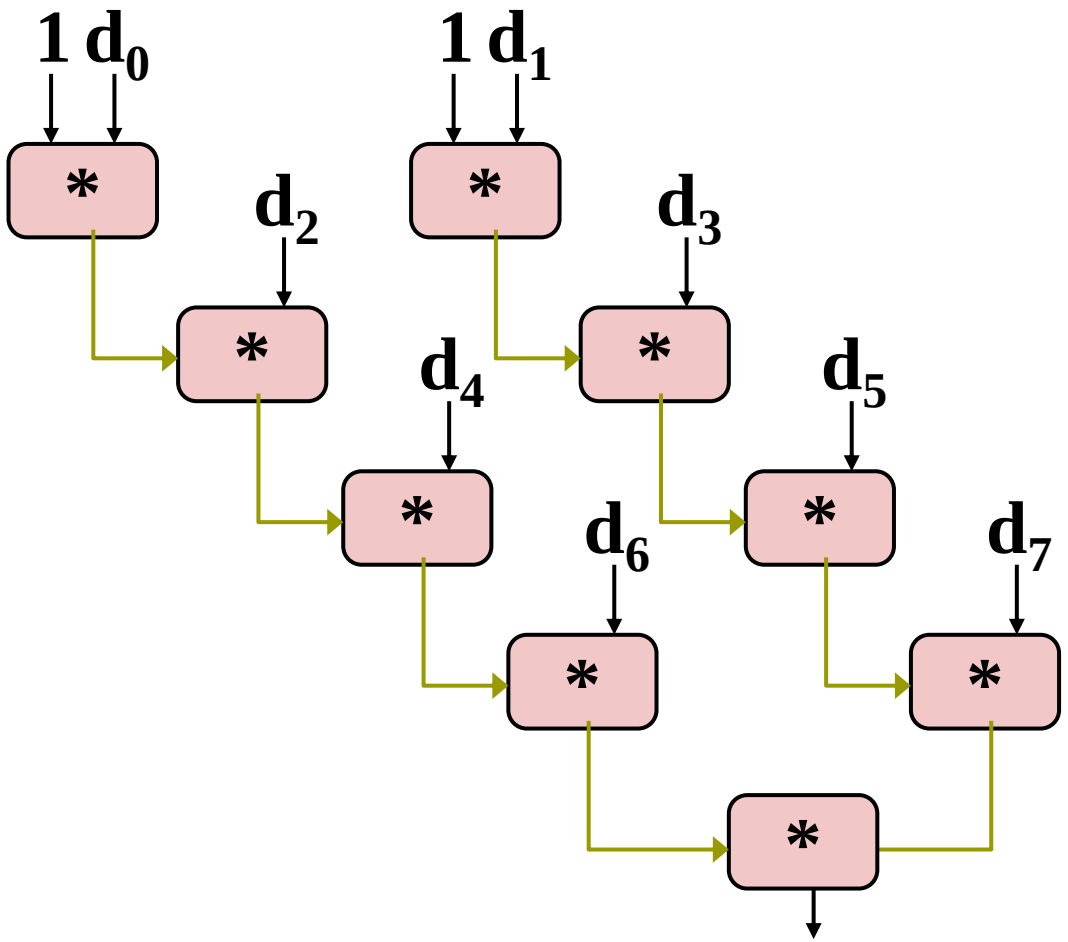
- 整数加 + 同时使用了两个加载单元

```
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!

循环展开 & 累加

- 设想对元素i到i+k-1合并运算
- 能循环展开到任一程度L吗？
  - 能够并行累加K个结果吗？
  - L是K的倍数
- 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>=C容量*L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
  - 长度小开销大（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							
Accumulators 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: 0.50

INT+	循环展开因子 L							
Accumulators 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

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

本节总结

- 田螺姑娘：编译器
- 做一名优秀的程序员
  - 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)

经典例题

- 1. 优化如下程序，给出优化结果并说明理由。

```
int sum_array(int a[M][N][N])   //M、N足够大
{
      int i, j, k, sum = 0;
      for (i = 0; i < N; i++)
          for (j = 0; j < N; j++)
              for (k = 0; k < M; k++)
                  sum += a[k][i][j];
      return sum;
}
```

参考答案


```
int sum_array(int a[M][N][N])   //M、N足够大
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++){
        int  *i_ptr = &a[i][0][0];
        for (j = 0; j < N; j++){
            int *j_ptr = i_ptr+j*N;
            for (k = 0; k < N-1; k+=2)
                sum =sum + (*(j_ptr+k)+*(j_ptr+k+1));
            if(k < N )
                sum += *(j_ptr+N-1);
         }
    }
    return sum;
}
```

解答

- (1) 一般优化：通过计算i_ptr,j_ptr减少第三层循环中的计算量。
- (2) 面向编译器的优化：使用循环展开2x1，也可使用其他循环展开。
- (3) 面向cache的优化：修改i,j,k的遍历顺序，使的cache的命中率尽可能地高。
- 运行时间比较：性能提升了2.53倍
- 机器：Intel（R）Core(TM) i7-8550U CPU @1.8GHz 1.99GHz

```
M = 100,N = 70
Runtime of v1 is : 0.001620 s
Runtime of v2 is : 0.000620 s
Process returned 0 (0x0)   execution time : 0.136 s
Press any key to continue.
```

```
M = 70,N = 70
Runtime of v1 is : 0.001000 s
Runtime of v2 is : 0.000420 s
Process returned 0 (0x0)   execution time : 0.091 s
Press any key to continue.
```

```
M = 100,N = 60
Runtime of v1 is : 0.001020 s
Runtime of v2 is : 0.000400 s
Process returned 0 (0x0)   execution time : 0.090 s
Press any key to continue.
```

拓展阅读：FastCDC

- 数据去重
- 基于内容分块

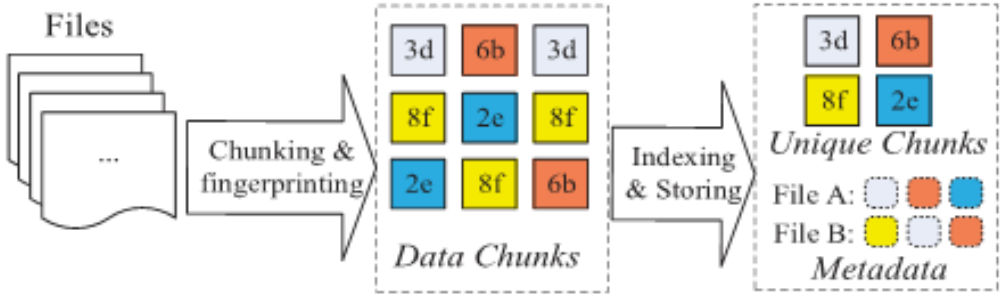


Fig. 1. General workflow of chunk-level data deduplication.

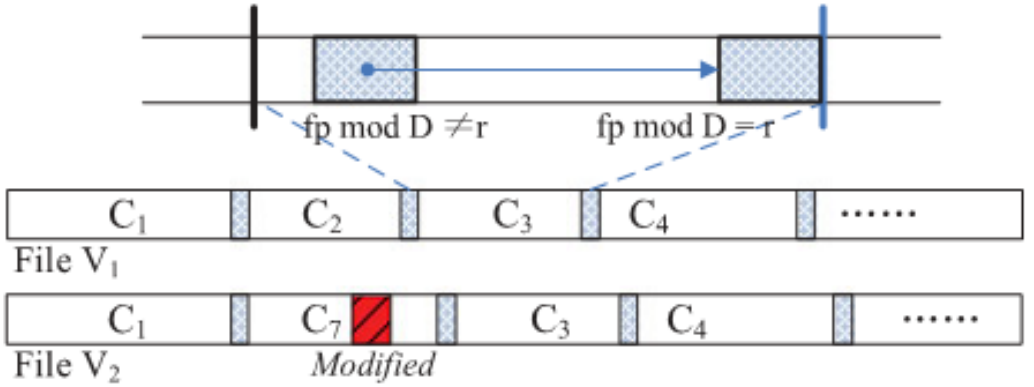
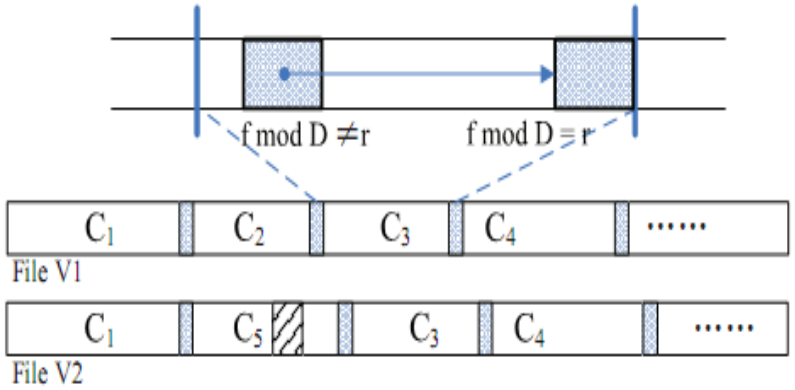


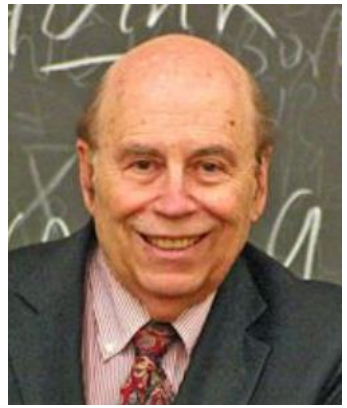
Fig. 2. The sliding window technique for the CDC algorithm. The hash value of the sliding window, *fp*, is computed via the Rabin algorithm (this is the *hashing stage* of CDC). If the lowest $log_2 D$ bits of the hash value matches a threshold value r, i.e., *fp* mod $D = r$, this offset (i.e., the current position) is marked as a chunk cut-point (this is the *hash-judging stage* of CDC).

论文链接：https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia

数据分块算法的探索

- 基于内容分块算法





Michael Rabin，图灵奖得主

- 基于内容分块算法主要解决数据修改带来的位置偏移问题：主流方案采纳基于Rabin的分块方案

$$Rabin(B_1, B_2, ..., B_\alpha) = \{\sum_{x=1}^{\alpha} B_x p^{\alpha - x}\} mod D$$

$$Rabin(B_i, B_{i+1}, ..., B_{i+\alpha-1}) =$$
$$\{[Rabin(B_{i-1}, ..., B_{i+\alpha-2}) - B_{i-1}P^{\alpha-1}]p + B_{i+\alpha-1}\} mod S$$

算法名称	伪代码实现和计算开销分析
Rabin	$hash = ((hash \hat{} U[a]) << 8) \mid b \hat{} T[hash >> N]$
	计算开销：一次或运算、两次异或运算、两次位移运算、两次数组查找。
Gear	$hash = (hash << 1) + GearTable[b]$
	计算开销：一次加运算、一次位移运算、一次数组查找。

拓展阅读：实例FastCDC

- 基于内容的分块算法

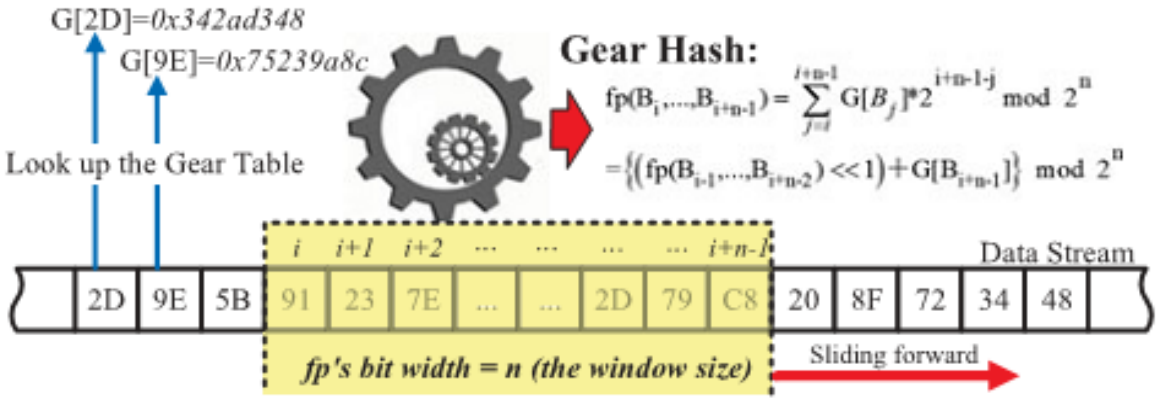


Fig. 4. A schematic diagram of the Gear hash.

Algorithm 1. GearCDC8KB

Input: data buffer, src; buffer length, n
Output: chunking breakpoint i
$MinSize \leftarrow 2KB$; $MaxSize \leftarrow 64KB$;
$i \leftarrow MinSize$; $fp \leftarrow 0$;
if $n \leq MinSize$ **then**
  return n
while $i < n$ **do**
  $fp = (fp << 1) + Gear[\ src[i]\ ]$;
  if $(fp\ \&\ 0x1fff == 0x78)\ ||\ i >= MaxSize$ **then**
    return i;
return i;

Algorithm 4. Rolling Two Bytes Each Time on FastCDC8KB (Without NC for Simplicity)

Input: data buffer, src; buffer length, n
Output: chunking breakpoint i
$MaskA \leftarrow 0x0000d93003530000LL$; // 13 '1' bits ;
$MaskA\_ls \leftarrow (MaskA << 1)$; $fp \leftarrow 0$; $i \leftarrow MinSize$;
$MinSize \leftarrow 2KB$; $MaxSize \leftarrow 64KB$;
if $n \leq MinSize$ **then**
  return n;
if $n \geq MaxSize$ **then**
  $n \leftarrow MaxSize$;
while $i < (n/2)$ **do**
  $fp = (fp << 2) + Gear\_ls[\ src[2*i]\ ]$;
  if $!\ (\ fp\ \&\ MaskA\_ls\ )$ **then**
    return $2*i$;
  $fp+ = Gear[\ src[2*i+1]\ ]$;
  if $!\ (\ fp\ \&\ MaskA\ )$ **then**
    return $2*i+1$;
return n;

拓展阅读：实例FastCDC

- 测试结果（用的perf工具）

TABLE 6
Number of Instructions, Instructions Per Cycle (IPC), and CPU Cycles Required to Chunk Data *Per Byte* by the 11 CDC Approaches on the Intel i7-8770 Processor

Approaches	Instructions	IPC	CPU cycles	branches
RC-v1	19.54	2.49	7.85	2.44
RC-v2	11.22	2.30	4.88	1.02
RC-v3	9.72	2.27	4.28	0.88
AE-v1	11.75	3.77	3.12	3.84
AE-v2	7.00	3.08	2.27	2.00
FC-v1	7.32	3.89	1.88	1.63
FC-v2	4.89	3.83	1.28	1.02
FC-v3	4.23	3.72	1.14	0.88
FC′-v1	5.28	3.87	1.36	1.13
FC′-v2	3.57	3.59	0.99	0.76
FC′-v3	3.09	3.47	0.89	0.66

代码链接：https://github.com/wxiacode/FastCDC-c

Hope you enjoyed the OS course!