程序优化

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

内容提纲

- **概览**
- **常见优化技术**
  - 代码外提/频度削弱
  - 强度削弱
  - 公共子表达式删除
  - 减少不必要的函数调用
- **优化的困难/Optimization Blockers**
  - 过程间调用/Procedure calls
  - 内存别名/Memory aliasing
- **指令级并行挖掘/Exploiting Instruction-Level Parallelism**
- **条件处理/Dealing with Conditionals**

性能现实/Performance Realities

- ***性能不仅仅是渐进复杂性/There's more to performance than asymptotic complexity***
- **常量倍的加速也很重要/Constant factors matter too!**
  - 代码编写方式不同很容易导致10倍左右的性能差异/Easily see 10:1 performance range depending on how code is written
  - 必须在多个层级进行优化/Must optimize at multiple levels:
    - 算法、数据表示、过程和循环/algorithm, data representations, procedures, and loops
- **性能优化必须了解系统/Must understand system to optimize performance**
  - 程序是怎样编译和运行的/How programs are compiled and executed
  - 现在的处理器+内存系统是如何运行的/How modern processors + memory systems operate
  - 如何测量性能并识别瓶颈/How to measure program performance and identify bottlenecks
  - 如何在不改变代码模块化和通用性的前提下提升性能How to improve performance without destroying code modularity and generality

带有优化功能的编译器/Optimizing Compilers

- **能够高效完成从高级代码到机器代码的映射/Provide efficient mapping of program to machine**
  - 寄存器分配/register allocation
  - 指令选择和指令调度/code selection and ordering (scheduling)
  - 死代码删除/dead code elimination
  - 消除小规模性能问题/eliminating minor inefficiencies
- **通常不会提升渐进有效性（编译器级别的优化替代不了算法级别的优化）/Don't (usually) improve asymptotic efficiency**
  - 依赖于程序员选择最佳的算法/up to programmer to select best overall algorithm
  - Big-O级别的效果通常会比常量级别更显著/big-O savings are (often) more important than constant factors
    - 但是常量加速也很重要/but constant factors also matter
- **优化难点/Have difficulty overcoming "optimization blockers"**
  - 内存别名/potential memory aliasing
  - 过程副作用/potential procedure side-effects

优化编译器的局限/Limitations of Optimizing Compilers

- **存在众多的前提和限制/Operate under fundamental constraint**
  - 不能改变程序行为/Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - 为了避免极端情况下行为不一致而采取了非常保守的优化/Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- **程序员非常清楚的行为编码后将变得模糊/Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - 例如程序员了解的数据取值区间可能远小于数据类型的对应的取值区间
  - e.g., Data ranges may be more limited than variable types suggest
- **大部分是过程内优化/Most analysis is performed only within procedures**
  - 全局优化是非常耗时/Whole-program analysis is too expensive in most cases
  - 新版GCC支持单个文件内的过程优化
    - 但是不支持多个文件之间的优化
- **大部分分析仅有静态信息/Most analysis is based only on *static* information**
  - 编译器无法预判运行时输入/Compiler has difficulty anticipating run-time inputs
- **编译器采用保守的策略/When in doubt, the compiler must be conservative**

常见优化措施/Generally Useful Optimizations

- **机器无关优化/Optimizations that you or the compiler should do regardless of processor / compiler**

- **代码移动/Code Motion**
  - 减少代码执行频度/Reduce frequency with which computation performed
    - 要求产生相同的结果/If it will always produce same result
    - 通常关注循环内不变代码外提/Especially moving code out of loop

```
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];
}
```

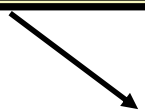


```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

代码外提示例/Compiler-Generated Code Motion (-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
```

计算强度削弱/Reduction in Strength

- 使用简单操作替代复杂操作/Replace costly operation with simpler one
- 移位和加法代替乘法和除法/Shift, add instead of multiply or divide

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

  - 与机器相关/Utility machine dependent
  - 依赖于乘除指令的开销/Depends on cost of multiply or divide instruction
    - 在Intel Nehalem机器上，整数乘法需要3个时钟周期/On Intel Nehalem, integer multiply requires 3 CPU cycles
- 识别连乘操作/Recognize sequence of products

```
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;
}
```

公共子表达式删除/Share Common Subexpressions

- 重复利用表达式的一部分/Reuse portions of expressions
- GCC在-O1时支持/GCC will do this with –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
```

优化障碍#1:过程调用/Optimization Blocker #1: Procedure Calls

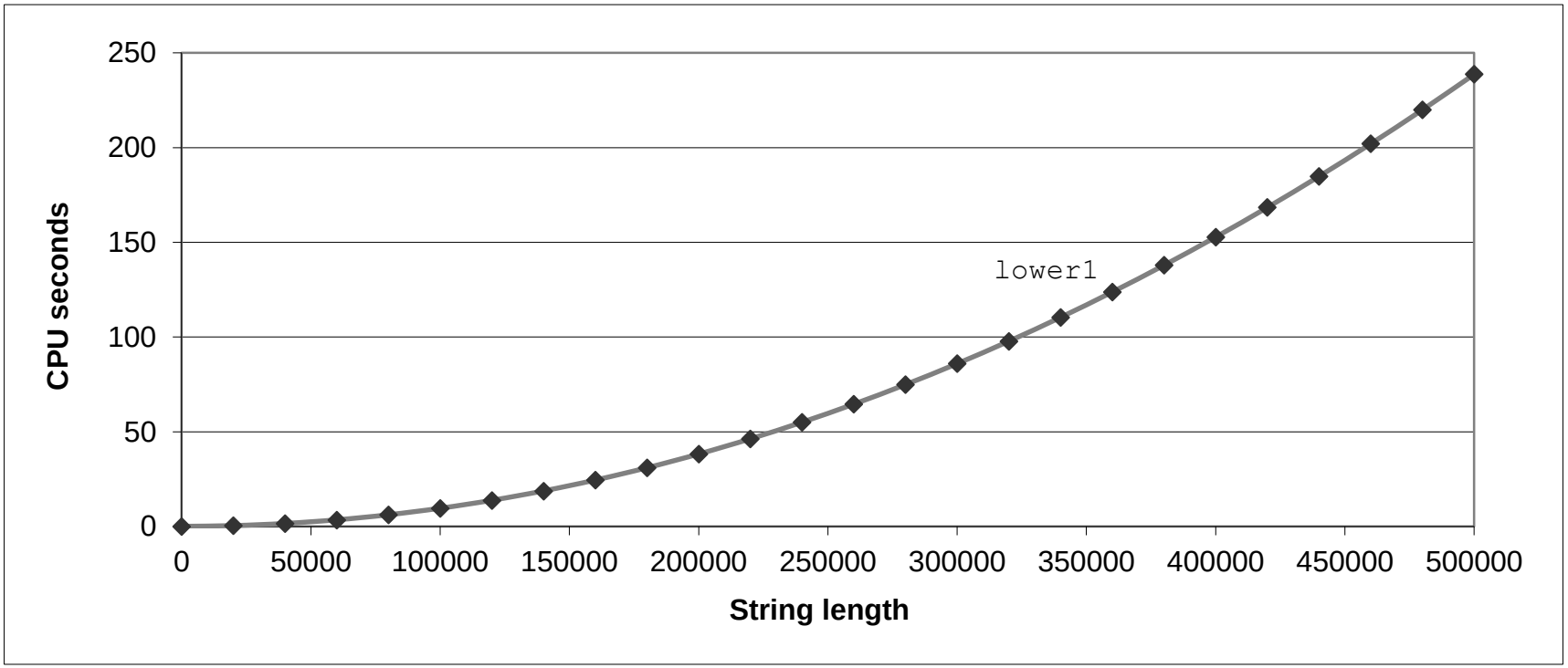
- **字符串转为小写字母/Procedure to Convert String to Lower Case**

```
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');
}
```

  - 从1998年秋季学期作业中提取的例子/Extracted from 213 lab submissions, Fall, 1998

小写转换性能/Lower Case Conversion Performance

- 字符串长度加倍时，时间变为4倍/Time quadruples when double string length
- 平方增长/Quadratic performance



循环转为Goto/Convert Loop To Goto Form

```
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:
}
```

- `strlen`在每个循环迭代执行/ `strlen` executed every iteration

strlen的实现/Calling 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/Strlen performance**
  - 需要扫描整个字符串查找null字符/Only way to determine length of string is to scan its entire length, looking for null character.
- **长度为N的字符串的整体性能/Overall performance, string of length N**
  - N次strlen调用/N calls to strlen
  - 每次长度为/Require times N, N-1, N-2, …, 1
  - 总体时间复杂度为$O(N^2)$ /Overall $O(N^2)$ performance

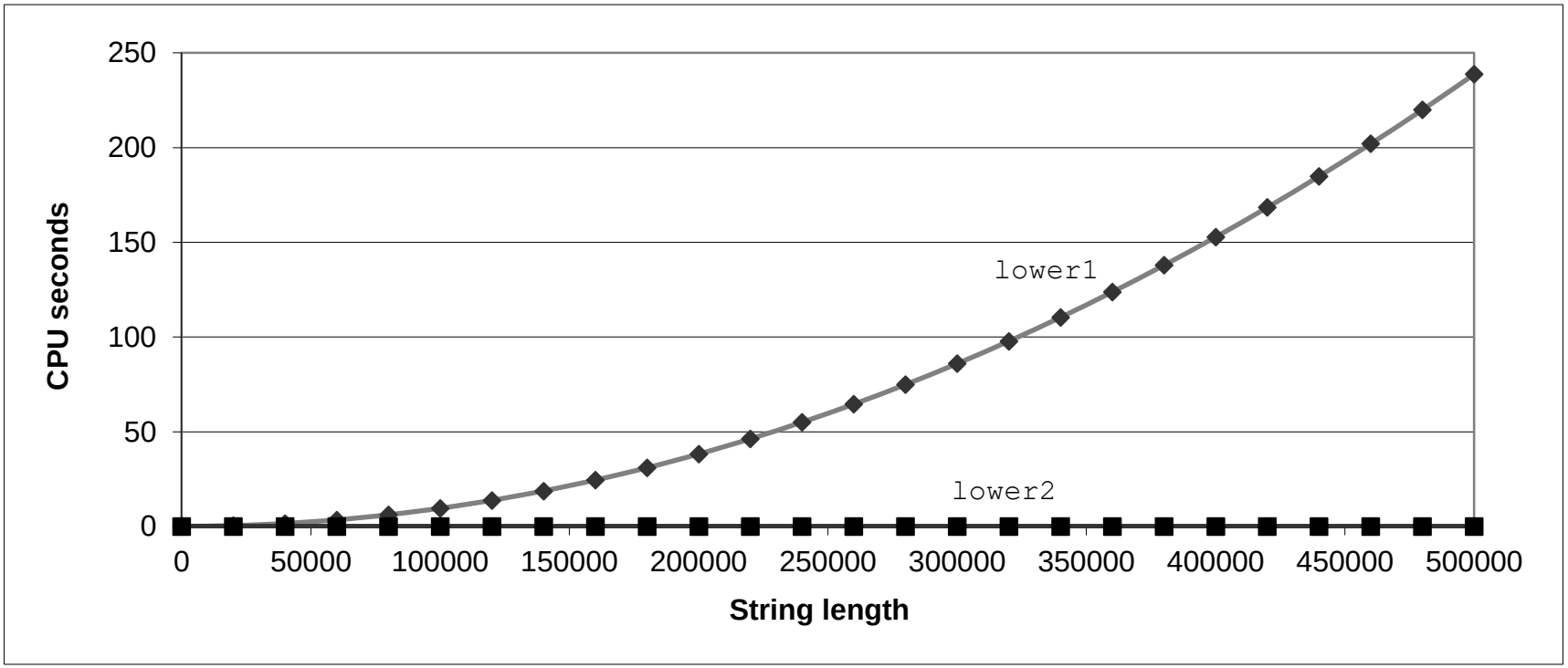
性能改进

```
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');
}
```

- 将strlen移除循环/Move call to `strlen` outside of loop
- 迭代之间结果不会改变/Since result does not change from one iteration to another
- 代码移动的一种形式/Form of code motion

小写转换性能/Lower Case Conversion Performance

- 随着字符串长度线性增长/Time doubles when double string length
- Linear performance of lower2



优化障碍:过程调用/Optimization Blocker: Procedure Calls

- ***编译器为什么不会自动将strlen外提? /Why couldn't compiler move `strlen` out of inner loop?***
  - 过程可能会有副作用/Procedure may have side effects
    - 每次调用时改变全局状态/Alters global state each time called
  - 对于同样的参数每次返回结果可能不同/Function may not return same value for given arguments
    - 依赖于全局状态/Depends on other parts of global state
    - 过程lower可能与strlen互相作用/Procedure `lower` could interact with `strlen`
- **Warning:**
  - 编译器将过程调用看做黑盒/Compiler treats procedure call as a black box
  - 很少做优化/Weak optimizations near them
- **Remedies:**
  - GCC –O1单个文件内优化/Use of inline functions
    - GCC does this with –O1
      - Within single file
  - 需要手动移动代码/Do your own code motion

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

内存的问题/Memory Matters

```
/* 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
```

- 每个迭代都会更新b[i]/ Code updates `b[i]` on every iteration
- 为什么编译器不会优化？ Why couldn't compiler optimize this away?

内存别名/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]
```

- 每个迭代更新b[i]/Code updates `b[i]` on every iteration
- 要考虑这些更新是不是会改变程序的行为/Must consider possibility that these updates will affect program behavior

消除别名/Removing Aliasing

```
/* 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
```

- 不用写回每个中间值/No need to store intermediate results

优化障碍:内存别名/Optimization Blocker: Memory Aliasing

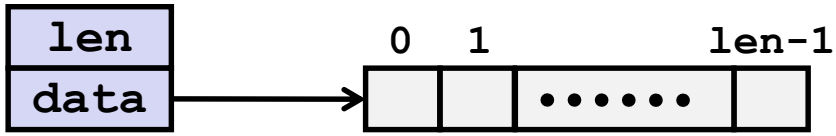
- **别名/Aliasing**
  - 两个不同的指针指向同一个位置/Two different memory references specify single location
  - C语言里面很常见/Easy to have happen in C
    - 允许地址计算/Since allowed to do address arithmetic
    - 直接访问存储结构/Direct access to storage structures
  - 尽可能使用局部变量/Get in habit of introducing local variables
    - 随着循环累计/Accumulating within loops
    - 以这种方式告知编译器不需要做别名/Your way of telling compiler not to check for aliasing

指令级并行挖掘/Exploiting Instruction-Level Parallelism

- **需要深入理解现在的处理器架构/Need general understanding of modern processor design**
  - 硬件可以并行执行多条指令/Hardware can execute multiple instructions in parallel
- **性能受限于数据依赖/Performance limited by data dependencies**
- **简单的变换可以获得较大的性能收益/Simple transformations can yield dramatic performance improvement**
  - 编译器通常无法完成类似的变换/Compilers often cannot make these transformations
  - 浮点不满足结合律和交换律，无法受益/Lack of associativity and distributivity in floating-point arithmetic

测试样例：向量的数据类型/Benchmark Example: Data Type for Vectors

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



- 数据类型/Data Types
  - 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;
}
```

测试计算/Benchmark Computation

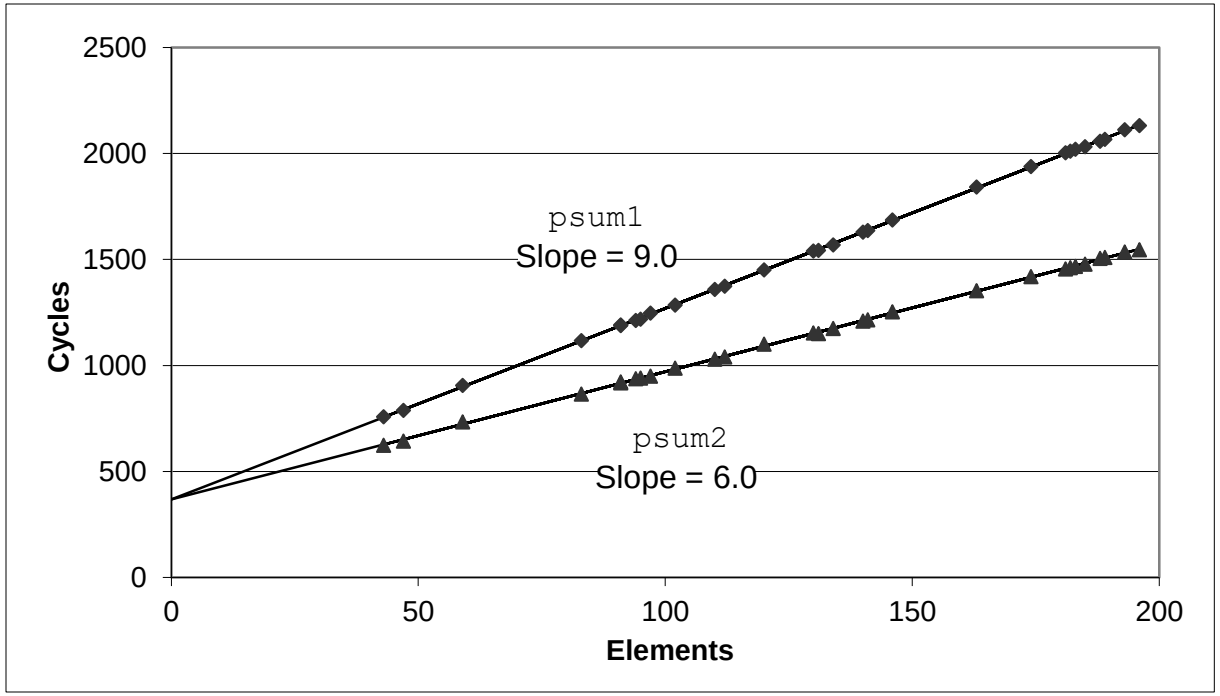
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

计算向量元素的和或者乘积/Compute sum or product of vector elements

- 数据类型/Data Types
  - int
  - long
  - float
  - double
- 操作/Operations
  - 定义不同的 OP 和 IDENT
  - + / 0
  - * / 1

每个元素需要的时钟周期/Cycles Per Element (CPE)

- 便于表述向量或者list的性能/Convenient way to express performance of program that operates on vectors or lists
- Length = n
- CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE 是线的斜率



测试性能/Benchmark Performance

```
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者乘积/Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

基本优化/Basic Optimizations

```
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;
}
```

- 将vec_length移出循环/Move vec_length out of loop
- 每个迭代避免边界检查/Avoid bounds check on each cycle
- 使用临时变量进行累积/Accumulate in temporary

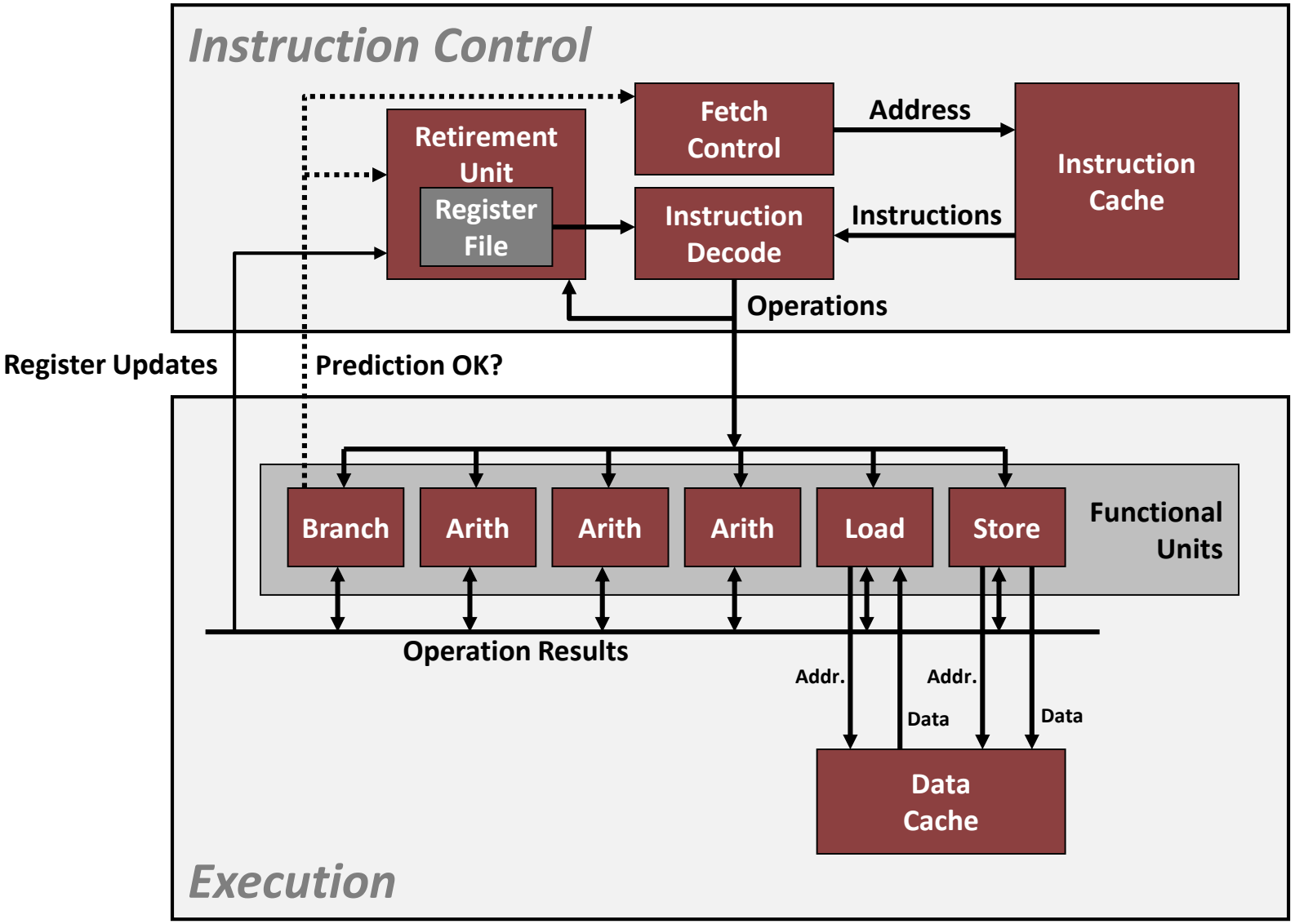
基本优化效果/Effect of Basic Optimizations

```
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;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
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/Eliminates sources of overhead in loop

现代CPU设计/Modern CPU Design



超标量处理器/Superscalar Processor

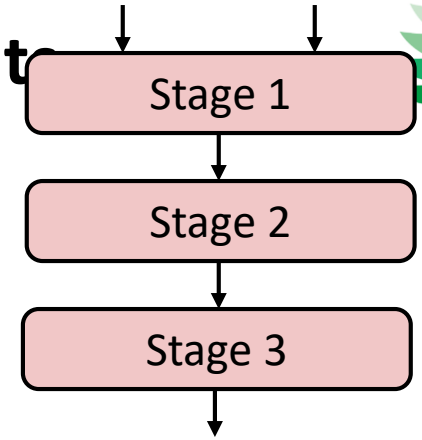
- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- 一个超标量处理器每个周期可以发射和执行多条指令。这些指令乱序执行但是顺序提交完成

- Benefit: without programming effort, superscalar processor can take advantage of the ***instruction level parallelism*** that most programs have
- 好处：不需要程序员特别设计，超标量处理器可以挖掘大多数程序中的指令级并行

- 大多数现在的处理器都是超标量/Most modern CPUs are superscalar.
- Intel从1993年的奔腾处理器开始/Intel: since Pentium (1993)

流水功能单元/Pipelined Functional Unit

```
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

- 将计算划分为不同的阶段/Divide computation into stages
- 在不同的阶段之间流动/Pass partial computations from stage to stage
- 每段将一个值传给后续的阶段后就可以开始新的计算/Stage i can start on new computation once values passed to i+1
- 例如在7个周期内完成3个乘法，尽管每个乘法需要3个周期/E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Haswell CPU

- 总共8个功能部件/8 Total Functional Units
- **多条指令可以并行执行/Multiple instructions can execute in parallel**

  2 load, with address computation

  1 store, with address computation

  4 integer

  2 FP multiply

  1 FP add

  1 FP divide
- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

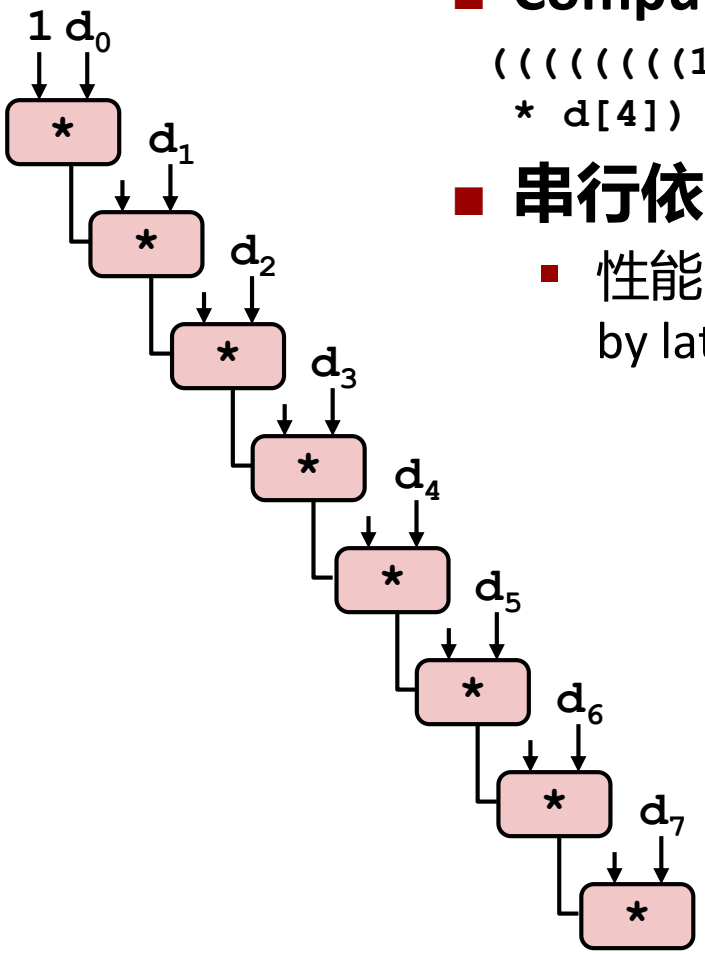
Combine4 对应的x86-64汇编/x86-64 Compilation of Combine4

- 内层循环(整数乘法)/Inner Loop (Case: Integer Multiply)

```
.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
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Combine4 = Serial Computation (OP = *)



- **Computation (length=8)**

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

- **串行依赖/Sequential dependence**
  - 性能：由OP的延迟决定/Performance: determined by latency of OP

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;
}
```

- 每个迭代做2倍更有用的工作/Perform 2x more useful work per iteration

循环展开效果/Effect of 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
Latency Bound	1.00	3.00	3.00	5.00

- **整数加法有用/Helps integer add**
  - Achieves latency bound

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

- **为什么其他无效？ /Others don't improve. *Why?***
  - 仍然存在线性依赖/Still sequential dependency

循环展开与重结合/Loop Unrolling with Reassociation (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];
```

- 会改变计算结果吗? Can this change the result of the computation?
- Yes, for FP. *Why?*

重结合效果/Effect of Reassociation

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
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- **Nearly 2x speedup for Int *, FP +, FP ***
  - 打破串行依赖/Reason: Breaks sequential dependency

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

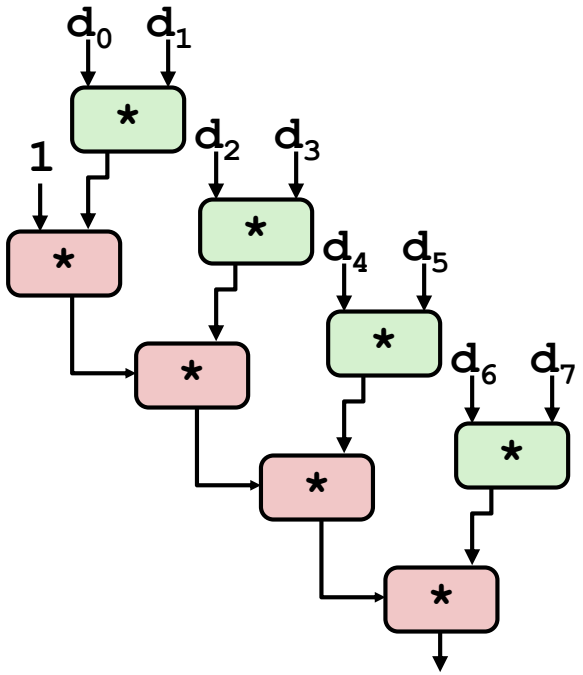
  - Why is that? (next slide)

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

重结合计算/Reassociated Computation

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



- **不同之处/What changed:**
  - 下一个迭代的内容可以早点开始/Ops in the next iteration can be started early (no dependency)
- **整体性能/Overall Performance**
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2

循环展开与单独累加器/Loop Unrolling with Separate Accumulators (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;
}
```

- 与association不同/Different form of reassociation

单独累加器效果/Effect of Separate Accumulators

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

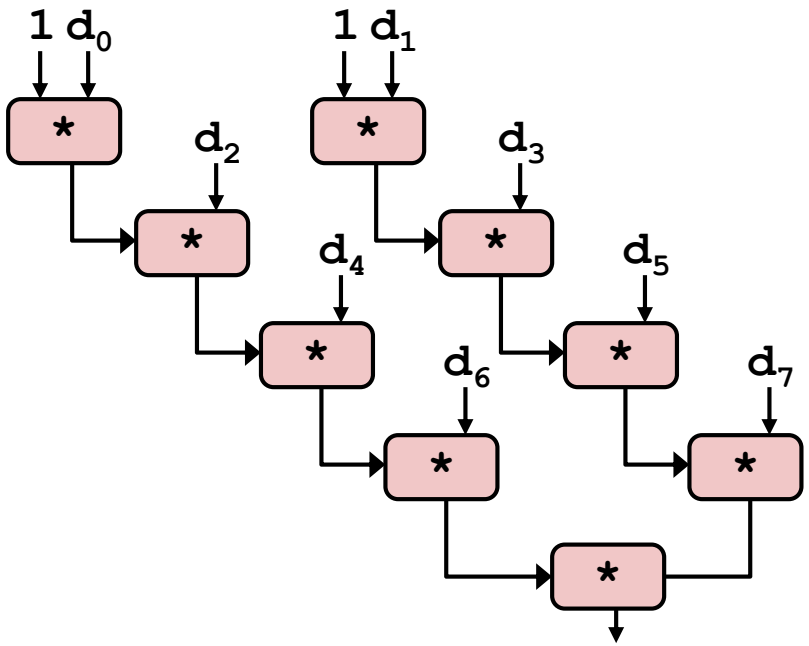
- Int + 使用两个load单元/Int + makes use of two load units

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

- 2x加速比/2x speedup (over unroll2) for Int *, FP +, FP *

单独累加器/Separate Accumulators

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



- **不同之处/What changed:**
  - 两个独立的操作流/Two independent “streams” of operations
- **总体性能/Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

循环展开&累加/Unrolling & Accumulating

- **主要思路/Idea**
  - 可以做L次展开/Can unroll to any degree L
  - 可以并行做K个结果累加/Can accumulate K results in parallel
  - L必须是K的倍数/L must be multiple of K

- **局限性/Limitations**
  - Diminishing returns
    - 受限与执行部件的数量/Cannot go beyond throughput limitations of execution units
  - 输入规模较小时开销会比较大/Large overhead for short lengths
    - 结束的部分通常串行完成/Finish off iterations sequentially

展开与累加/Unrolling & Accumulating: Double *

- **Case**
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor 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

展开与累加/Unrolling & Accumulating: Int +

- **Case**
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

FP *	Unrolling Factor 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

性能收益/Achievable Performance

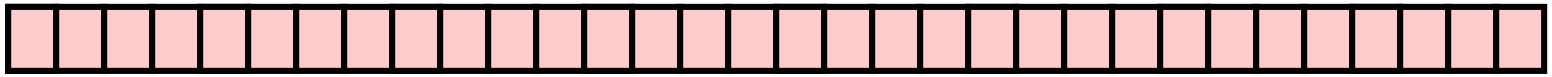
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- 受限于功能单元的吞吐率/Limited only by throughput of functional units
- 与未优化代码相比有42x的性能提升/Up to 42X improvement over original, unoptimized code

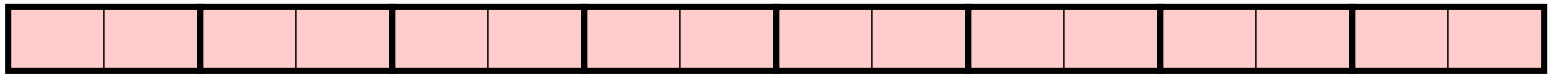
基于AVX2的性能/Programming with AVX2

YMM Registers

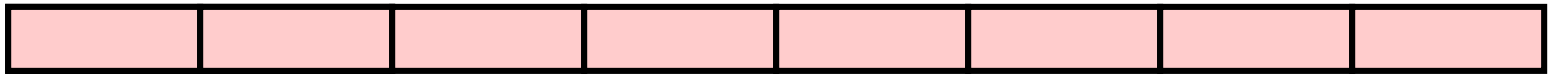
- 16 total, each 32 bytes
- 32 single-byte integers



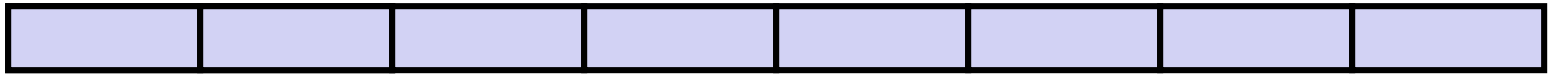
- 16 16-bit integers



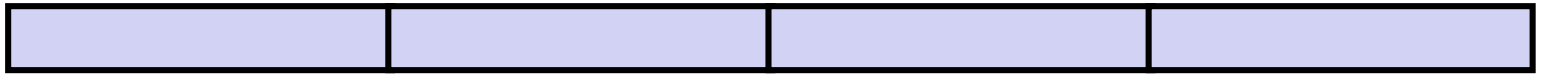
- 8 32-bit integers



- 8 single-precision floats



- 4 double-precision floats



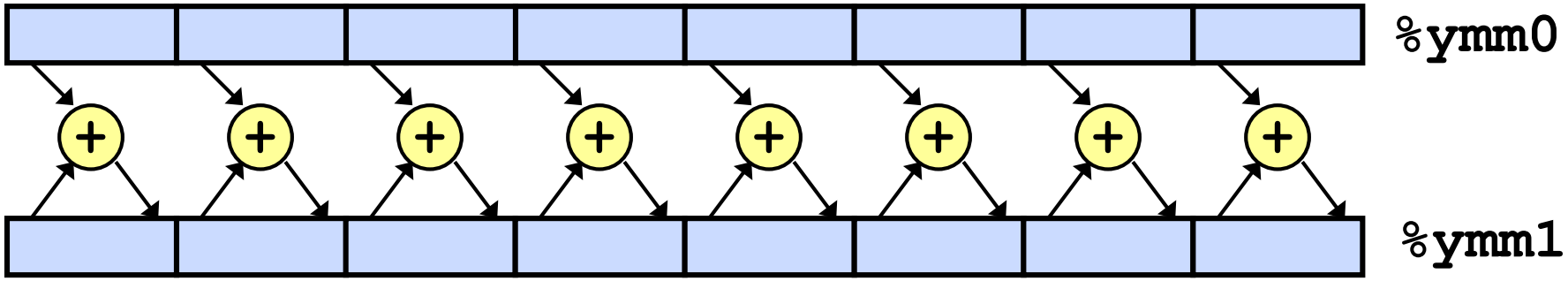
- 1 single-precision float

- 1 double-precision float

SIMD操作/SIMD Operations

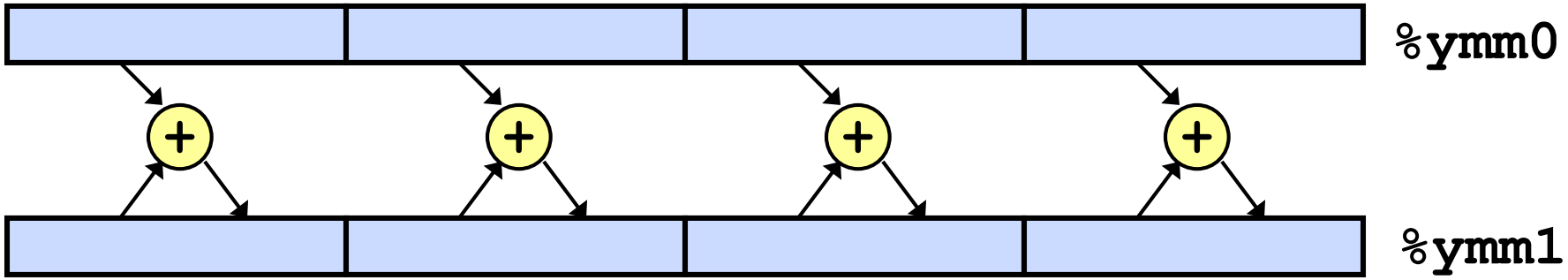
- SIMD操作：单精度/SIMD Operations: Single Precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



- SIMD操作：双精度/SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



使用向量指令/Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- **利用AVX指令/Make use of AVX Instructions**
  - 多个数据单元上的并行操作/Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

如何处理分支？/What About Branches?

- **挑战/Challenge**
  - 指令控制单元必须早于执行单元获知地址以充满流水线/Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

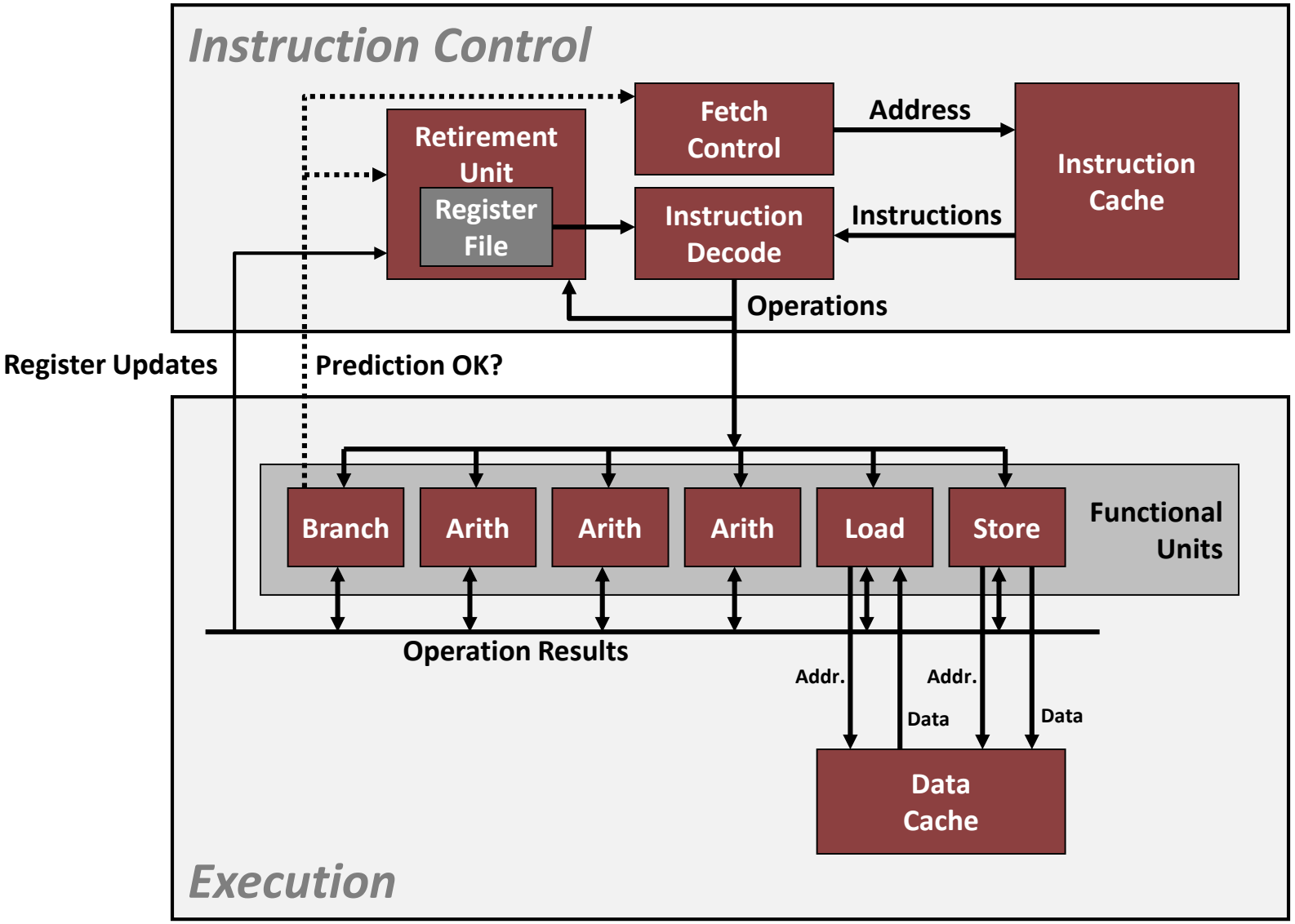
```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```

Executing

How to continue?

  - 遇到分支时不确定下一个取指地址/When encounters conditional branch, cannot reliably determine where to continue fetching

当代CPU设计/Modern CPU Design



分支预测输出/Branch Outcomes

- **遇到分支时不知道走哪个/When encounter conditional branch, cannot determine where to continue fetching**
  - 实际上会走的分支/Branch Taken: Transfer control to branch target
  - 实际上不会走的分支/Branch Not-Taken: Continue with next instruction in sequence
- **只有分支指令执行完之后才知道/Cannot resolve until outcome determined by branch/integer unit**

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

    . . .

  404685:   repz retq
```

Branch Not-Taken

Branch Taken

分支预测/Branch Prediction

- 思路/Idea
  - 猜想走哪个分支/Guess which way branch will go
  - 从预测的位置开始执行指令/Begin executing instructions at predicted position
    - 实际上并不修改寄存器和内存数据/But don't actually modify register or memory data

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

  . . .

404685:   repz retq
```

Predict Taken

Begin Execution

循环分支预测/Branch Prediction Through Loop

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

i = 98

Assume
vector length = 100

Predict Taken (OK)

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

i = 99

Predict Taken (Oops)

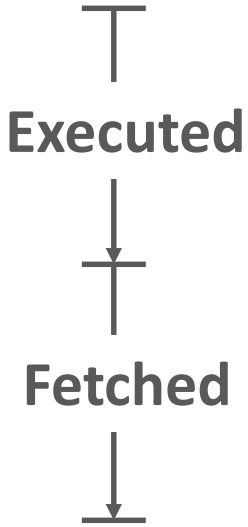
```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```

i = 100

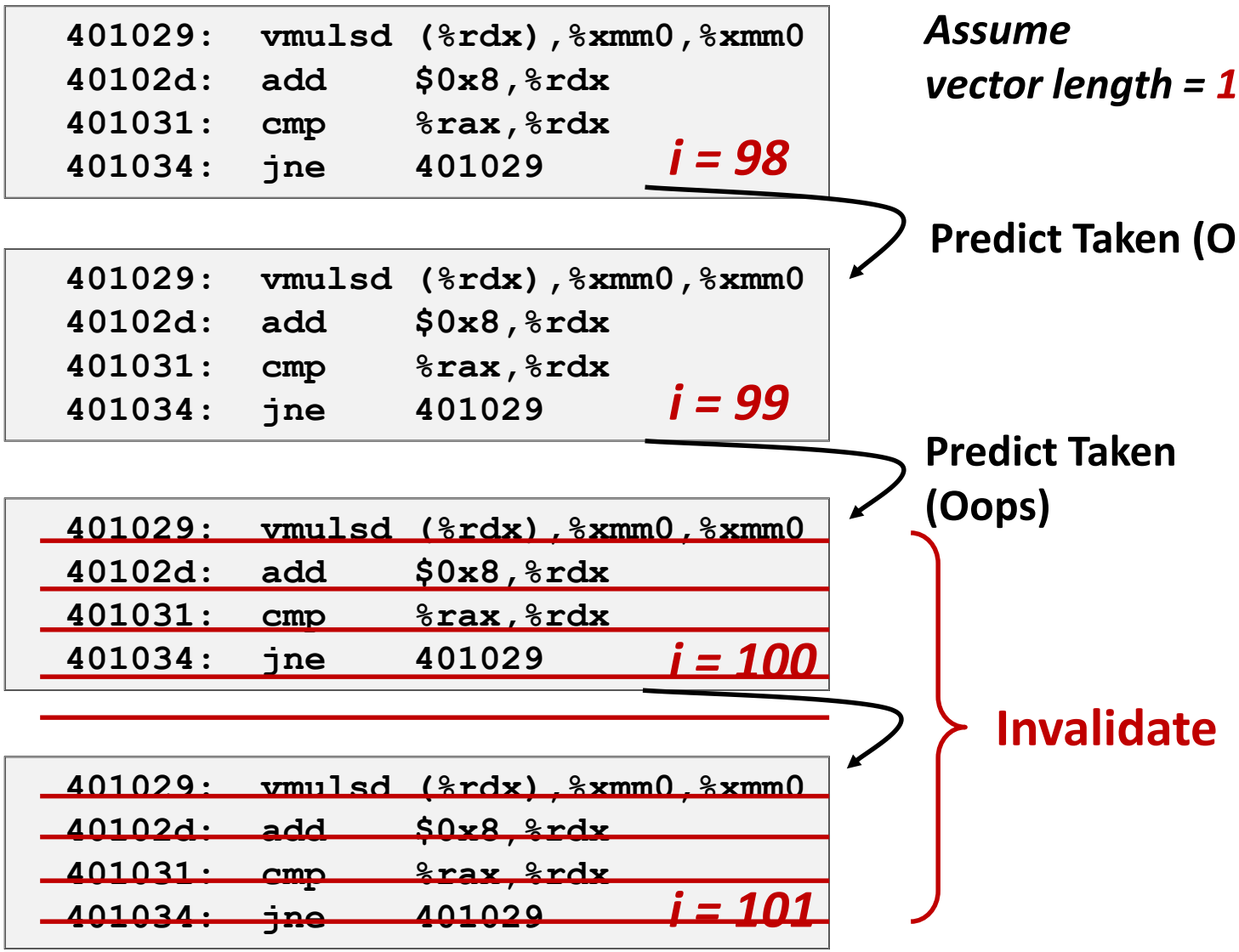
Read invalid location

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

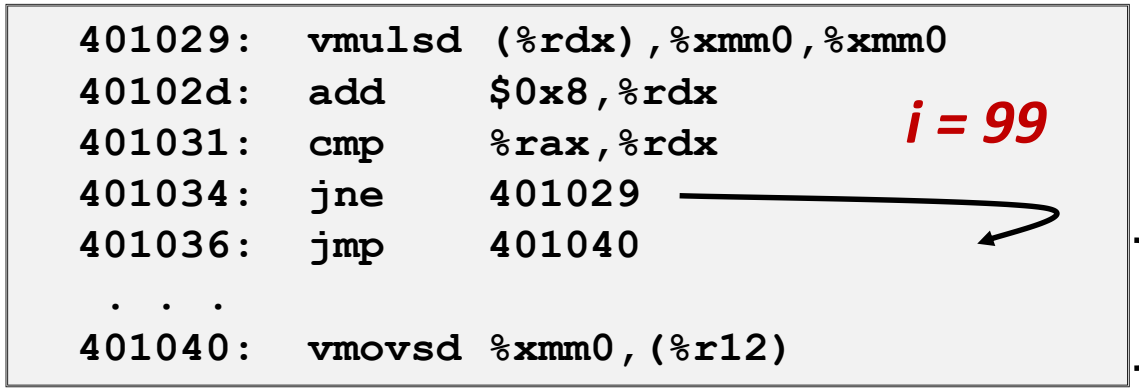
i = 101



预测失败后的无效操作/Branch Misprediction Invalidation



分支预测失败的恢复/Branch Misprediction Recovery



- **性能开销/Performance Cost**
  - 现代处理器上需要多个时钟/Multiple clock cycles on modern processor
  - 对性能影响较大/Can be a major performance limiter

获得更高的性能/Getting High Performance

- **代码编写**
  - 注意隐藏的性能瓶颈/Watch out for hidden algorithmic inefficiencies
  - 编写编译器友好的代码/Write compiler-friendly code
    - 注意优化障碍：过程调用内内存引用/Watch out for optimization blockers: procedure calls & memory references
  - 注意最内层循环/Look carefully at innermost loops (where most work is done)

- **面向机器的性能调优/Tune code for machine**
  - 挖掘指令级并行/Exploit instruction-level parallelism
  - 避免分支预测失败/Avoid unpredictable branches
  - 编写Cache友好的代码/Make code cache friendly (Covered later in course)

致谢

- 本文档基于CMU的15-213: Introduction to Computer Systems课程材料加工获得
- 感谢原作者Randal E. Bryant 和David R. O'Hallaron的辛苦付出