程序优化

100076202： 计算机系统导论

任课教师：

宿红毅　　张艳　　黎有琦　　李秀星

原作者：

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

Carnegie Mellon University

议题 Today

- **编译器优化的原理和目标 Principles and goals of compiler optimization**
- **局部优化 Local optimization**
  - 常量折叠、强度削弱、死代码删除、公共子表达式删除CSE Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **全局优化 Global optimization**
  - 内联、代码外提、循环转换 Inlining, code motion, loop transformations
- **优化的障碍 Obstacles to optimization**
  - 内存别名、过程调用、非可结合运算 Memory aliasing, procedure calls, non-associative arithmetic
- **机器相关的优化 Machine-dependent optimization**
  - 分支预测、循环展开、调度和向量化 Branch predictability, loop unrolling, scheduling, vectorization

Back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums
and vacuum tubes,
Real Programmers wrote in machine code.

Not FORTRAN. Not RATFOR. Not, even,
assembly language.

Machine Code.

Raw, unadorned, inscrutable hexadecimal numbers.
Directly.

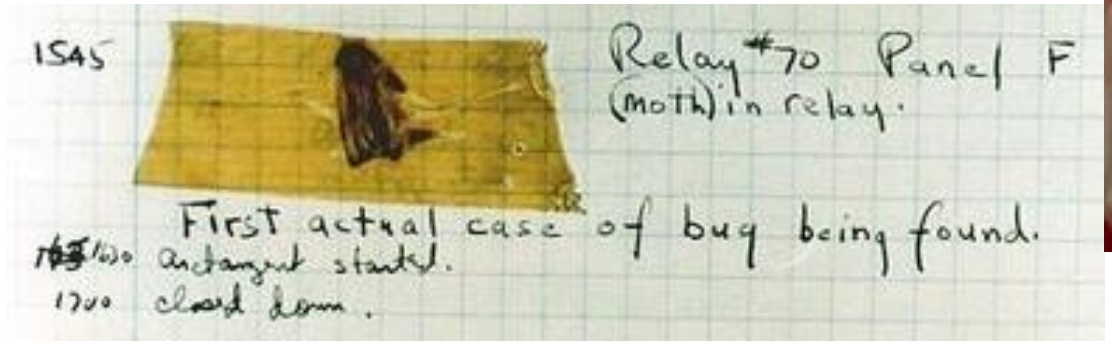
回想过去，那时“软件”这个词听起来很有趣，而真正的计算机是由磁鼓和电子管制成的，真正的程序员编写机器代码。不是FORTRAN。不是RATFOR。甚至不是汇编语言。而是机器代码。原始、朴素、难以理解的十六进制数。

— “Mel的故事，一个真正的程序员”

“The Story of Mel, a Real

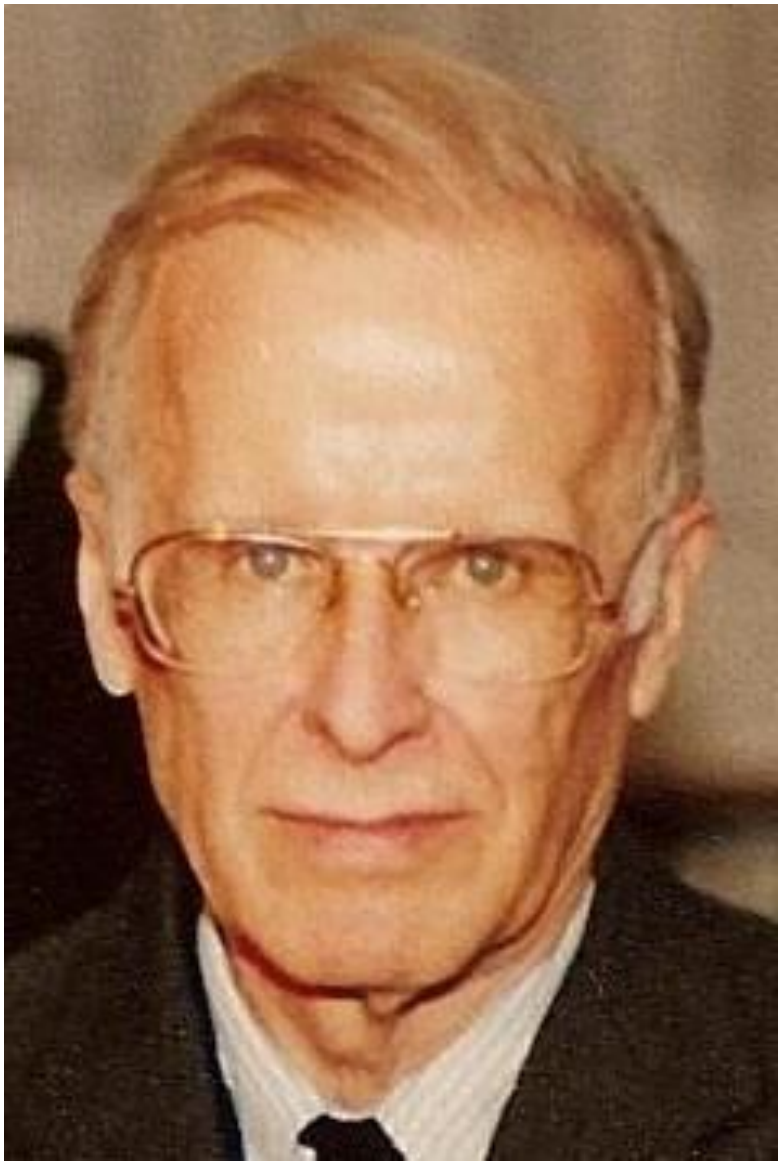
海军少将格蕾丝·霍珀 Rear Admiral Grace Hopper

- 第一个发现实际bug（飞蛾）的人 First person to find an actual bug (a moth)
- 1951年发明了第一个编译器（COBOL的前身） Invented first compiler in 1951 (precursor to COBOL)
- “我决定数据处理器应该能够用英语编写程序，计算机会把它们翻译成机器代码” “I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code”



John Backus

- 1957年为IBM 704开发了FORTRAN/Developed FORTRAN in 1957 for the IBM 704
- 至今仍在使用的最古老的独立于机器的编程语言 Oldest machine-independent programming language still in use today
- “我的大部分工作都源于懒惰。我不喜欢编写程序，因此，当我在IBM 701上工作时，我开始研究编程系统，以使编写程序更容易”“Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs”



Fran Allen

- 许多优化编译技术的先驱 Pioneer of many optimizing compilation techniques
- 1966年写了一篇论文，介绍了控制流图的概念，这一概念至今仍是编译器理论的核心 Wrote a paper in 1966 that introduced the concept of the control flow graph, which is still central to compiler theory today
- 第一位获得ACM图灵奖的女性 First woman to win the ACM Turing Award

编译器优化的目标
Goals of compiler optimization

- **指令数最少 Minimize number of instructions**
  - 不要多次计算 Don’t do calculations more than once
  - 根本不要做不必要的计算 Don’t do unnecessary calculations at all
  - 避免慢指令（乘法、除法） Avoid slow instructions (multiplication, division)
- **避免等待内存 Avoid waiting for memory**
  - 尽可能将所有内容保存在寄存器中 Keep everything in registers whenever possible
  - 以缓存友好模式访问内存 Access memory in cache-friendly patterns
  - 尽早从内存加载数据，并且只加载一次 Load data from memory early, and only once
- **避免分支 Avoid branching**
  - 不要做出根本不必要的决定 Don’t make unnecessary decisions at all
  - 使CPU更容易预测分支目标 Make it easier for the CPU to predict branch destinations
  - “展开”循环以将分支成本分摊到更多指令 “Unroll” loops to spread cost of branches over more instructions

编译器优化的限制
Limits to compiler optimization

- **一般不能改进算法复杂度 Generally cannot improve algorithmic complexity**
  - 只有不变因素，但这些因素的价值可能是10倍或更多… Only constant factors, but those can be worth 10x or more…
- **一定不要引起程序行为的任何变化 Must not cause *any* change in program behavior**
  - 程序员可能不关心“边缘情况”行为，但编译器不知道 Programmer may not care about “edge case” behavior, but compiler does not know that
  - 例外：语言可能声明某些更改是可接受的 Exception: language may declare some changes acceptable
- **通常一次只分析一个函数 Often only analyze one function at a time**
  - 全程序分析（“LTO”）成本高昂，但越来越受欢迎 Whole-program analysis (“LTO”) expensive but gaining popularity
  - 例外：内联将多个函数合并为一个函数Exception: *inlining* merges many functions into one
- **预测运行时输入的技巧 Tricky to anticipate run-time inputs**
  - 性能剖析引导优化可以帮助处理常见情况，但… Profile-guided optimization can help with common case, but…
  - “最坏情况”的性能可能与“正常情况”一样重要 “Worst case” performance can be just as important as “normal”
  - 特别是对于暴露于恶意输入的代码（例如网络服务器） Especially for code exposed to *malicious* input (e.g. network servers)

性能事实 Performance Realities

- ***性能不仅仅是渐进复杂性 There's more to performance than asymptotic complexity***
- **不变的因素也很重要 Constant factors matter too!**
  - 根据代码编写方式，很容易看到10:1的性能范围 Easily see 10:1 performance range depending on how code is written
  - 必须在多种级别进行优化 Must optimize at multiple levels:
    - 算法、数据表示、过程和循环 algorithm, data representations, procedures, and loops
- **必须理解系统以优化性能 Must understand system to optimize performance**
  - 如何编译和执行程序 How programs are compiled and executed
  - 现代处理器和内存系统如何运行 How modern processors + memory systems operate
  - 如何测量程序性能和识别瓶颈 How to measure program performance and identify bottlenecks
  - 如何改进性能同时不损坏代码模块性和通用性 How to improve performance without destroying code modularity and generality

带有优化功能的编译器 Optimizing Compilers

- **提供高效的程序和机器代码之间的映射 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
  - 删除轻微的低效率 eliminating minor inefficiencies
- **（通常）不提高渐进效率 Don't (usually) improve asymptotic efficiency**
  - 由程序员选择最佳总体算法 up to programmer to select best overall algorithm
  - 降低时间复杂度（经常）比不变因素更重要 big-O savings are (often) more important than constant factors
    - 但是不变因素也很重要 but constant factors also matter
- **优化难点 Have difficulty overcoming "optimization blockers"**
  - 内存别名 potential memory aliasing
  - 潜在过程副作用 potential procedure side-effects

优化编译器的局限 Limitations of Optimizing Compilers

- **在基本约束下运行 Operate under fundamental constraint**
  - 不能改变程序行为 Must not cause any change in program behavior
    - 可能在程序使用非标准语言功能时除外 Except, possibly when program making use of nonstandard language features
  - 通常会阻止它进行只会影响不理智条件下行为的优化。 Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- **对于程序员来说显而易见的行为可能会被语言和编码风格所混淆 Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - 例如，数据范围可能比变量类型建议的更有限 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 Newer versions of GCC do interprocedural analysis within individual files
    - 但是不支持多个文件之间的优化 But, not between code in different files
- **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**

- **代码外提 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)

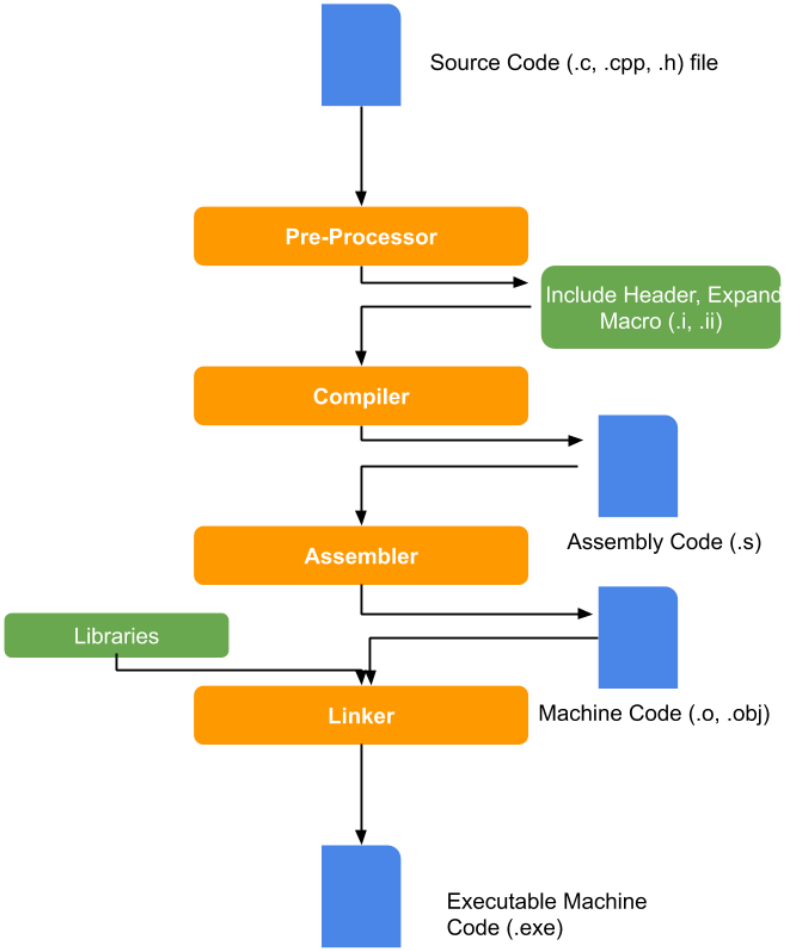
```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    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
```

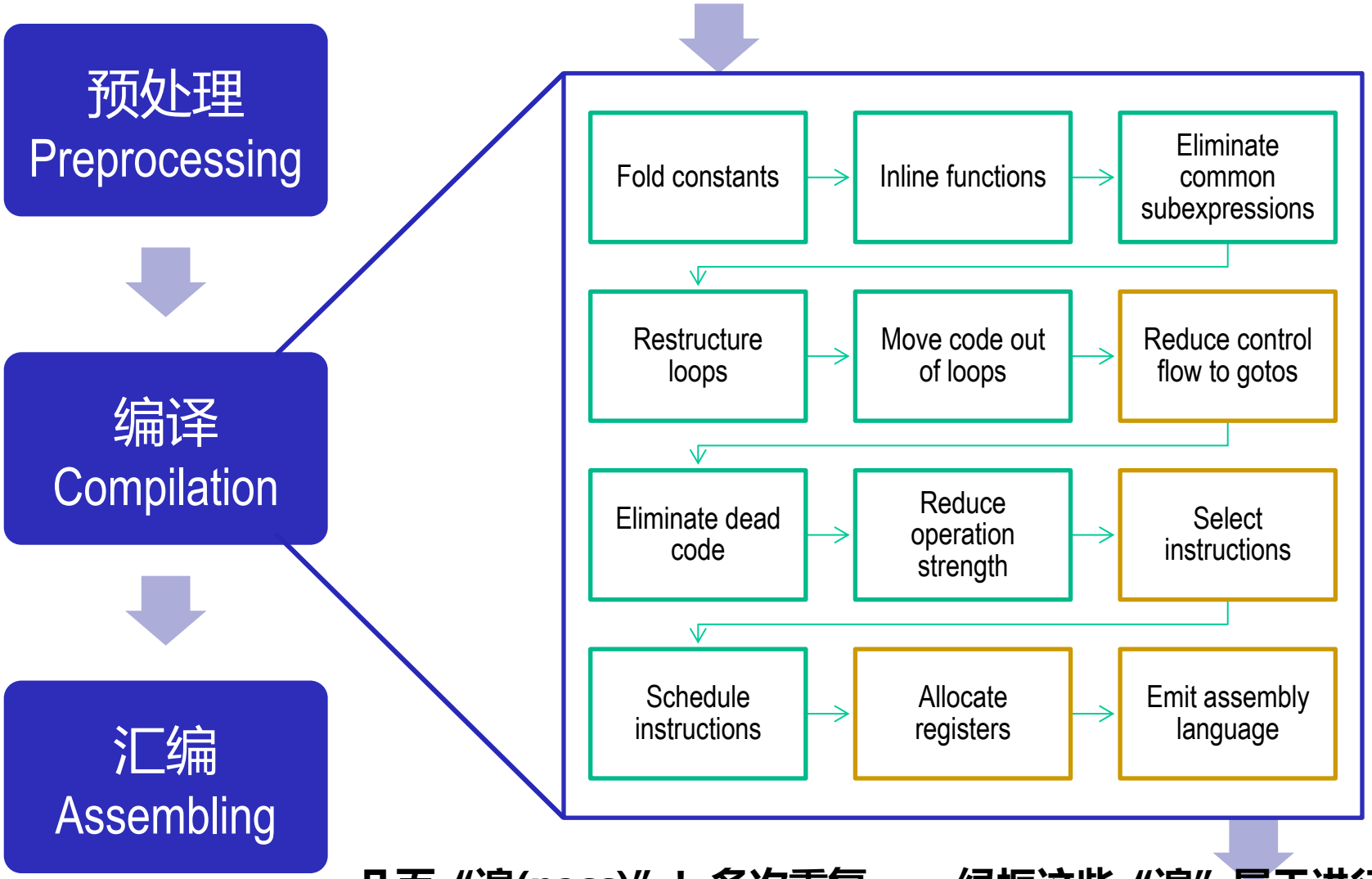
编译代码意味着什么?
What does it mean to compile code?

- **CPU仅理解机器代码 The CPU only understands *machine code* directly**
- **所有其他语言必为以下二者之一 All other languages must be either**
  - *解释型:* 由软件执行 *interpreted:* executed by software
  - *编译型:* 由软件翻译成机器代码 *compiled:* translated to machine code by software



编译系统是一个流水线

Compilation is a pipeline

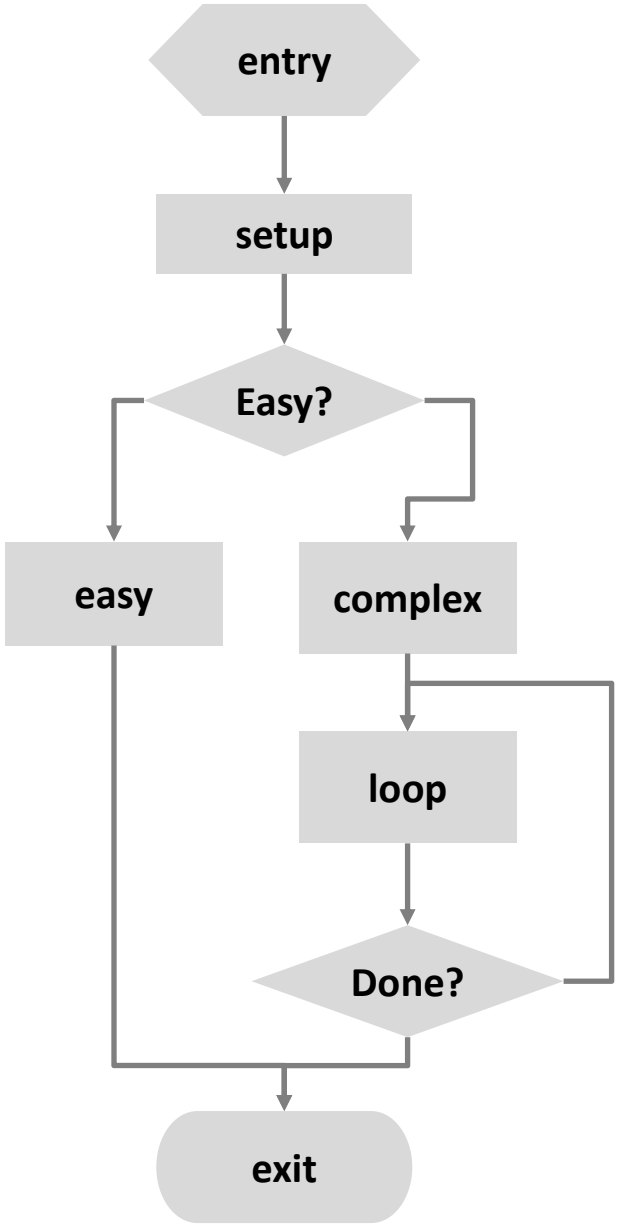


几百“遍(pass)”！多次重复。---绿框这些“遍”属于进行优化

Hundreds! Many repeated.

两类优化 Two kinds of optimizations

- **局部优化在单个基本块中工作**
  Local optimizations work inside a single *basic block*
  - 常量折叠、强度削弱、死代码删除、（局部）CSE… Constant folding, strength reduction, dead code elimination, (local) CSE, …
- **全局优化处理函数的整个*控制流图* Global optimizations process the entire *control flow graph* of a function**
  - 循环转换、代码外提、（全局）CSE… Loop transformations, code motion, (global) CSE, …



优化分类

- 编译器优化的原理和目标 Principles and goals of compiler optimization
- **局部优化 Local optimization**
  - 常量折叠、强度削弱、死代码删除和公共子表达式删除cse Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **全局优化 Global optimization**
  - 内联、代码外提和循环转换 Inlining, code motion, loop transformations
- **优化的障碍 Obstacles to optimization**
  - 内存别名、过程调用和非可结合运算 Memory aliasing, procedure calls, non-associative arithmetic
- **机器相关的优化 Machine-dependent optimization**
  - 分支预测、循环展开、调度和向量化 Branch predictability, loop unrolling, scheduling, vectorization

常量折叠 Constant Folding

- **编译器中完成运算 Do arithmetic in the compiler**

```
long mask = 0xFF << 8;      →
long mask = 0xFF00;
```

- **任何具有常量输入的表达式都可以折叠 Any expression with constant inputs can be folded**
- **甚至可能删除库函数调用。。。 Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");    →
size_t namelen = 11;
```

强度削弱 Strength reduction

- **用省时的操作替代费时的操作 Replace expensive operations with cheaper ones**

```
long a = b * 5;      →
long a = (b << 2) + b;
```

- **乘法和除法通常是被替代的目标 Multiplication and division are the usual targets**
- **乘法经常用内存访问表达式替代 Multiplication is often hiding in memory access expressions**

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
  - 依赖于乘除指令开销 Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
    - Intel Nehalem上，整型乘法需要3个CPU周期
- 识别乘积序列 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;
}
```

死代码删除 Dead code elimination

- **不要编写从不会执行的代码 Don't emit code that will never be executed**

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- **不要编写结果会被覆盖的代码 Don't emit code whose result is overwritten**

```
x = 23;
x = 42;
```

- **这样可能看起来很愚蠢，但是 These may look silly, but...**
  - 可能通过其他优化产生 Can be produced by other optimizations
  - 对x的赋值可能相隔很远 Assignments to x might be far apart

公共子表达式删除
Common Subexpression Elimination

- **排除重复计算，只进行一次 Factor out repeated calculations, only do them once**

```
norm[i] = v[i].x*v[i].x + v[i].y*v[i].y;
   →
elt = &v[i];
x = elt->x;
y = elt->y;
norm[i] = x*x + y*y;
```

公共子表达式共享 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;
```

三次乘法 3 multiplications: i*n, (i–1)*n, (i+1)*n

一次乘法 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
```

议题

- 编译器优化原理和目标 Principles and goals of compiler optimization
- 局部优化 Local optimization
  - 常量折叠、强度消弱、死代码删除和公共子表达式删除 Constant folding, strength reduction, dead code elimination, common subexpression elimination
- 全局优化 Global optimization
  - 内联、代码外提和循环转换 Inlining, code motion, loop transformations
- 优化的障碍 Obstacles to optimization
  - 内存别名、过程调用和非可结合计算 Memory aliasing, procedure calls, non-associative arithmetic
- 机器相关的优化 Machine-dependent optimization
  - 分支预测、循环展开、调度和向量化 Branch predictability, loop unrolling, scheduling, vectorization

内联函数 Inlining

- **将函数体复制到其调用者中 Copy body of a function into its caller(s)**
  - 可以为许多其他优化创造机会 Can create opportunities for many other optimizations
  - 可以使代码更大，因此更慢（大小；指令-高速缓冲区） Can make code much bigger and therefore slower (size; i-cache)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

内联函数 Inlining

- **将函数体复制到其调用者中 Copy body of a function into its caller(s)**
  - 可以为许多其他优化创造机会 Can create opportunities for many other optimizations
  - 可以使代码更大，因此更慢（大小；指令-高速缓冲区） Can make code much bigger and therefore slower (size; i-cache)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

始终为真
Always true

什么也不做
Does nothing

可以常量折叠
Can constant fold

内联函数 Inlining

- **将函数体复制到其调用者中 Copy body of a function into its caller(s)**
  - 可以为许多其他优化创造机会 Can create opportunities for many other optimizations
  - 可以使代码更大，因此更慢（大小；指令-高速缓冲区） Can make code much bigger and therefore slower (size; i-cache)

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

代码外提 Code Motion

- 将计算移出循环 Move calculations out of a loop
- 仅当每次迭代会产生同样的结果时有效 Only valid if every iteration would produce same result

```
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];
```

循环转换 Loop Transformations

重新安排整个循环嵌套，以取得最大效率 Rearrange entire loop nests for maximum efficiency

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      a[j*n + i] = atan2(i, j);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j]
                 + (i >= 1 && j >= 1)
                   ? a[(i-1)*n + (j-1)]
                   : 0;
}
```

循环转换 Loop Transformations

*循环交换：*以高速缓存友好顺序做迭代 *Loop interchange*: do iterations in cache-friendly order

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      a[i*n + j] = atan2(j, i);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j]
                 + (i >= 1 && j >= 1)
                   ? a[(i-1)*n + (j-1)]
                   : 0;
}
```

循环转换 Loop Transformations

*循环融合：*合并具有相同限制的相邻循环 *Loop fusion*: combine adjacent loops with the same limits

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++) {
    for (long j = 0, j < n; j++) {
      a[i*n + j] = atan2(j, i);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j]
                 + (i >= 1 && j >= 1)
                   ? a[(i-1)*n + (j-1)]
                   : 0;
    }
  }
}
```

循环转换 Loop Transformations

归纳变量消除：用代数替换循环索引 *Induction variable elimination*: replace loop indices with algebra

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n*n; i++) {
    for (long j = 0, j < n; j++) {
      a[i] = atan2(i%n, i/n);



      b[i]       = a[i]
                 + (i >= n && i%n >= 1)
                    ? a[i - n - 1]
                    : 0;
    }
  }
}
```

议题

- **编译器优化原理和目标 Principles and goals of compiler optimization**
- **局部优化 Local optimization**
  - 常量折叠、强度消弱、死代码删除和公共子表达式删除 Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **全局优化 Global optimization**
  - 内联、代码外提、循环转换 Inlining, code motion, loop transformations
- **优化的障碍 Obstacles to optimization**
  - 内存别名、过程调用和非可结合运算 Memory aliasing, procedure calls, non-associative arithmetic
- **机器相关的优化 Machine-dependent optimization**
  - 分支预测、循环展开、调度和向量化 Branch predictability, loop unrolling, scheduling, vectorization

内存很重要 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**
  - 两个不同的内存引用指向同一个位置 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

内存别名 Memory Aliasing

```
/* Sum rows 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
```

- 每次迭代代码都会更新b[i] Code updates b[i] on every iteration
- 为何编译器不能优化掉？ Why couldn't compiler optimize this away?

内存别名 Memory Aliasing

```
/* Sum rows 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);
```

```
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]
```

- 每次迭代代码都会更新b[i] Code updates b[i] on every iteration
- 必须考虑这些更新影响程序行为的可能性 Must consider possibility that these updates will affect program behavior

避免别名惩罚 Avoiding Aliasing Penalties

```
/* Sum rows 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
.Loop:
        addsd   (%rdi), %xmm0        # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

- 使用局部变量保存中间结果 Use a local variable for intermediate results

避免别名惩罚 Avoiding Aliasing Penalties

```
/* Sum rows 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;
    }
}
```

```
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,  27, 224},
   32,  64, 128};
```

Value of B:

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

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

```
i = 1: [3, 27, 16]
```

```
i = 2: [3, 27, 224]
```

- 仍然在操作中间改变了A数组 Still changes A in the middle of the operation
- 不同的结果 Different results

避免别名惩罚 Avoiding Aliasing Penalties

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows3(double *restrict a, double *restrict 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_rows3 inner loop
.Loop:
        addsd   (%rdi), %xmm0         # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

- 使用restrict修饰符告诉编译器a和b不是别名 Use `restrict` qualifier to tell compiler that a and b cannot alias
- 比局部变量可靠性低 Less reliable than using local variables

避免别名惩罚 Avoiding Aliasing Penalties

```
subroutine sum_rows4(a, b, n)
    implicit none
    integer, parameter :: dp = kind(1.d0)
    real(kind=dp), dimension(:), intent(in) :: a
    real(kind=dp), dimension(:), intent(out) :: b
    integer, intent(in) :: n
    integer :: i, j
    do i = 1,n
        b(i) = 0
        do j = 1,n
            b(i) = b(i) + a(i*n + j)
        end
    end
end
```

```
# sum_rows4 inner loop
.Loop:
        addsd   (%rdi), %xmm0          # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

- 使用Fortran语言 Use Fortran
- 在Fortran中数组参数假设不是别名 Array parameters in Fortran are assumed not to alias

函数调用是不透明的Function calls are opaque

- **编译器一次检查一个函数 Compiler examines one function at a time**
  - 有些代码异常在单个文件中 Some exceptions for code in a single file
- **必须假设函数调用可以做任何事情 Must assume a function call could do anything**
- **通常不能 Cannot usually**
  - 移动函数调用 move function calls
  - 改变函数的调用次数 change number of times a function is called
  - 跨函数调用从内存缓存数据到寄存器 cache data from memory in registers across function calls

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

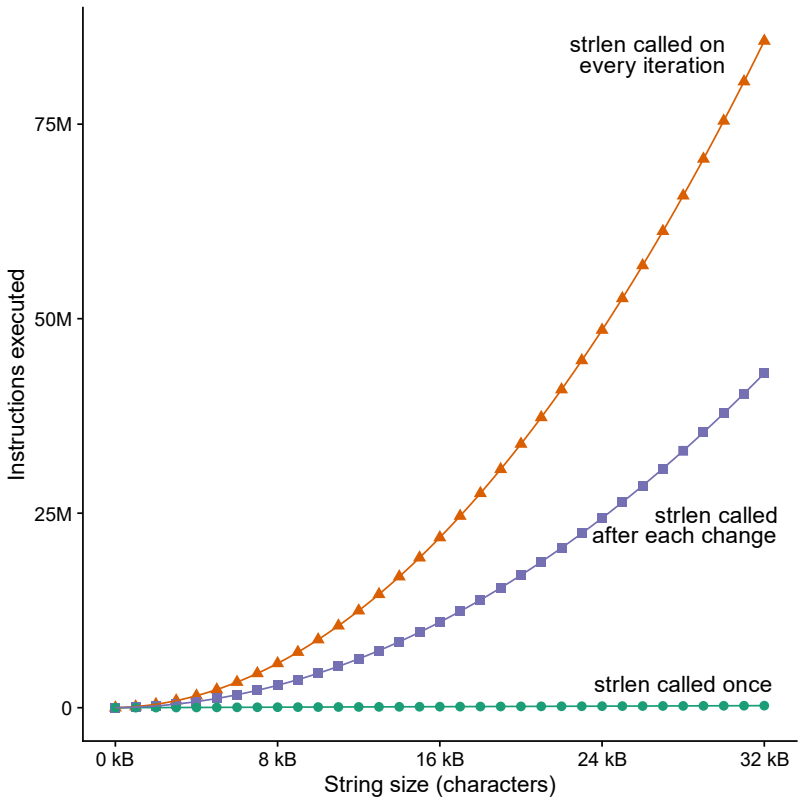
- **时间复杂度为 O(n) execution time**
- **返回值取决于 Return value depends on:**
  - s的值 value of *s*
  - 地址s处内存的内容 contents of memory at address *s*
    - 仅关心是否单独字节为零 Only cares about whether individual bytes are zero
    - 不修改内存 Does not modify memory
- **编译器可能知道或不知道部分情况 Compiler might know *some* of that (but probably not)**

未能将函数调用移出循环
Can't move function calls out of loops

```
void lower_quadratic(char *s) {
  size_t i;
  for (i = 0; i < strlen(s); i++) //每次迭代调用
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}

void lower_still_quadratic(char *s) {
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] += 'a' - 'A';
      n = strlen(s); //每次改变后调用
    }
}

void lower_linear(char *s) {
  size_t i, n = strlen(s); //绿线，调用一次
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}
```



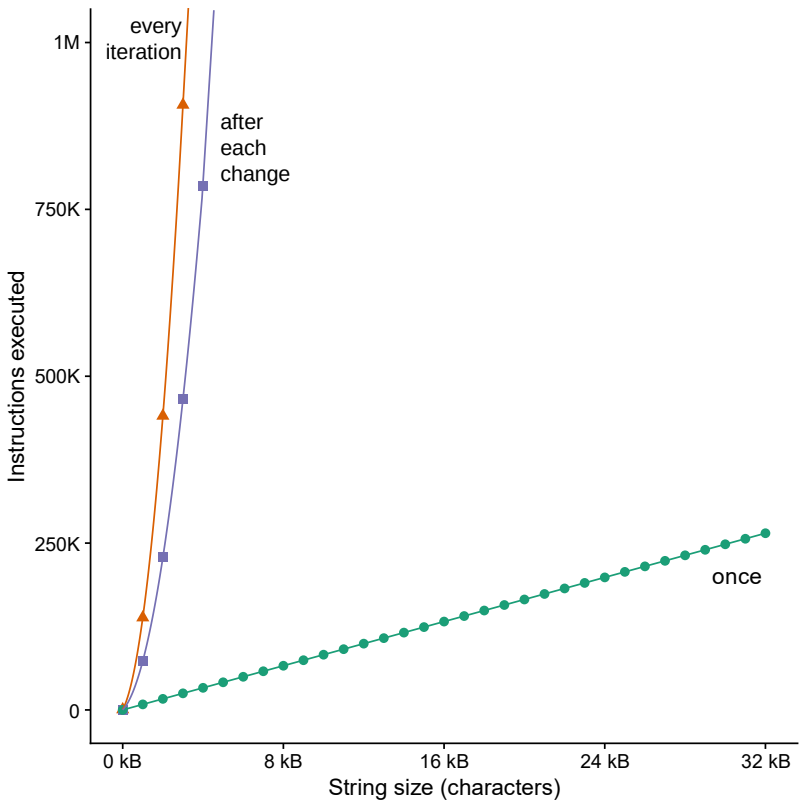
有更多这类bug示例 Lots more examples of this kind of bug: accidentallyquadratic.tumblr.com

未能将函数调用移出循环
Can’t move function calls out of loops

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

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

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



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`
- **警告 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
- **补救措施 Remedies:**
  - 使用内联函数 Use of inline functions
    - GCC –O1优化 GCC does this with –O1
      - 在单个文件内部 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;
}
```

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

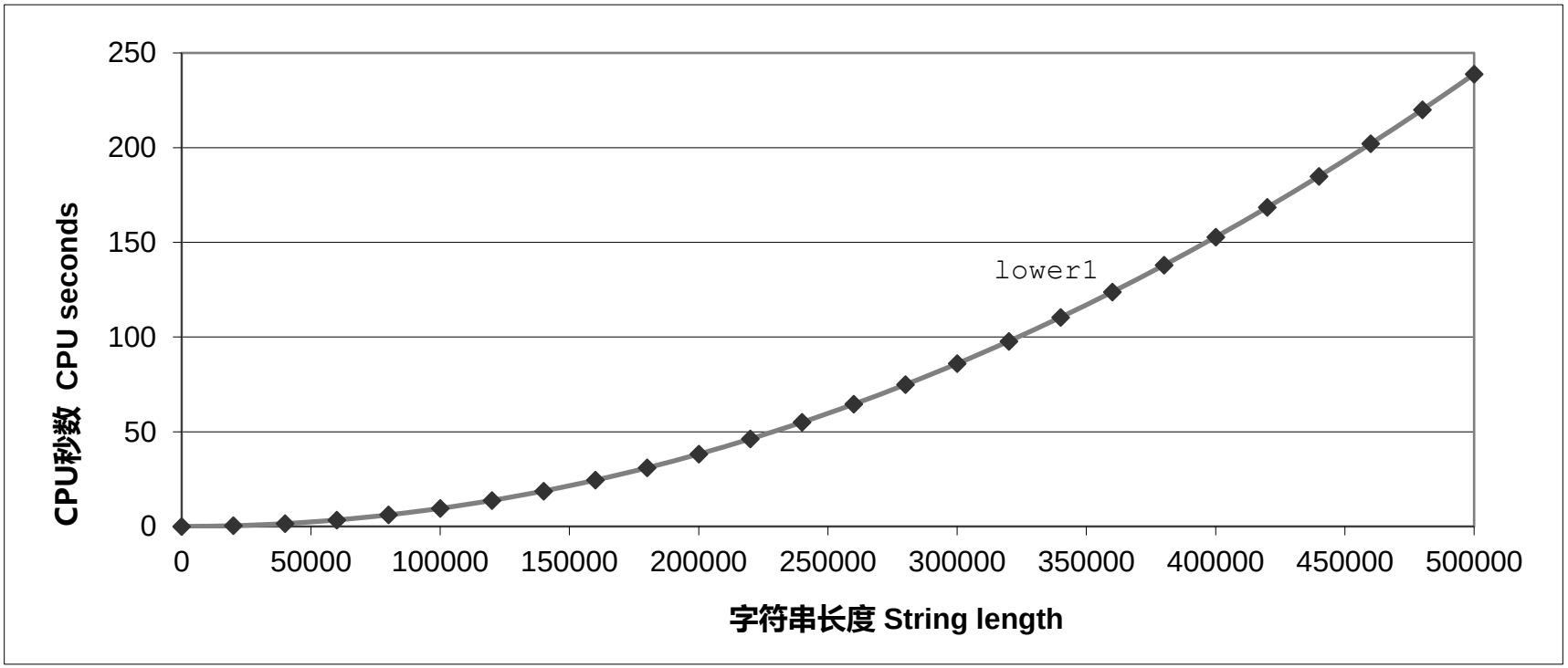
- **该过程将字符串转为小写字母 Procedure to Convert String to Lower Case**

```
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');
}
```

  - 摘自实验提交代码 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形式 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**
  - 确定字符串长度的惟一方法是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.
- **整体性能，串长为N Overall performance, string of length N**
  - N次strlen调用 N calls to strlen
  - 需要次数为 Require times N, N-1, N-2, …, 1
  - 总体时间复杂度为$O(N^2)$ Overall $O(N^2)$ performance

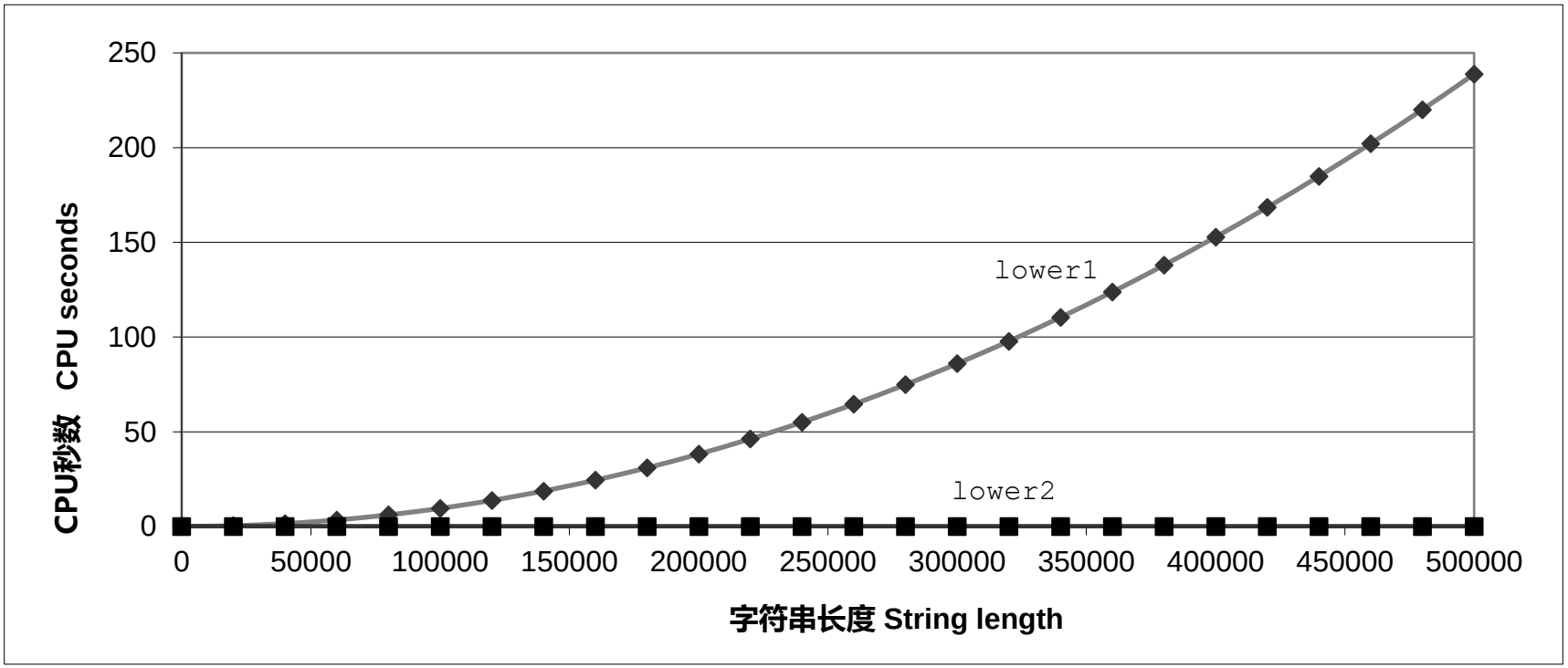
性能改进 Improving Performance

```
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

- 运行时间随着字符串长度线性增长 Time doubles when double string length
- Lower2为线性性能 Linear performance of lower2



非可结合运算 Non-associative arithmetic

- **当$(a \odot b) \odot c$不等于$a \odot (b \odot c)$时 When is $(a \odot b) \odot c$ not equal to $a \odot (b \odot c)$?**
  - 八元数 Octonions
  - 向量叉乘 Vector cross product
  - *浮点数 Floating-point numbers*

- **例如： Example: $a = 1.0, b = 1.5 \times 10^{38}, c = -1.5 \times 10^{38}$**
  (单精度IEEE浮点数 single precision IEEE fp)

$$a + b = 1.5 \times 10^{38} \quad (a + b) + c = 0$$
$$b + c = 0 \quad a + (b + c) = 1$$

- **阻止任何改变操作顺序的优化 Blocks any optimization that changes order of operations**

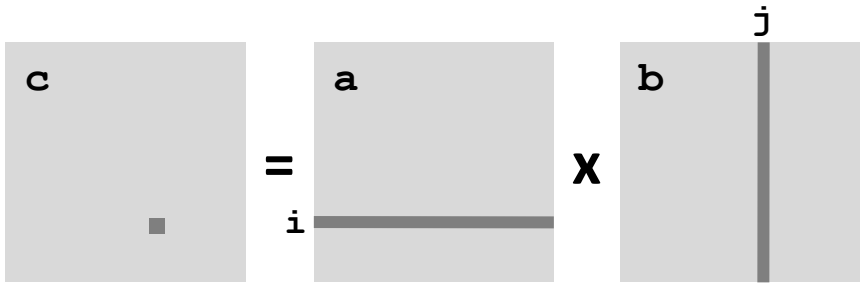
非可结合运算 Non-associative arithmetic

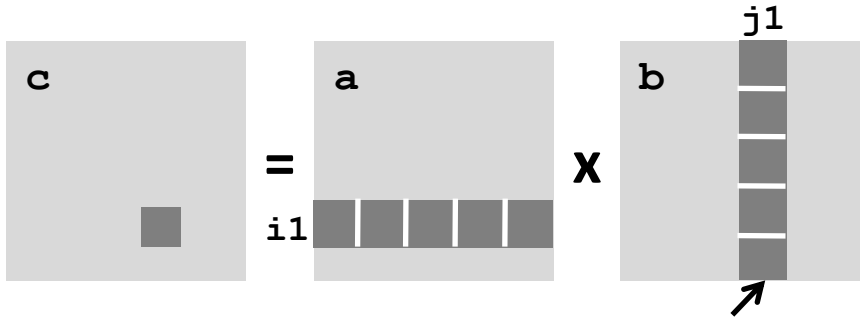
```
void mmm(double *a, double *b,
         double *c, int n) {
  memset(c, 0, n*n*sizeof(double));

  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i*n + j] += a[i*n + k]
                    * b[k*n + j];
}
```

```
void mmm(double *a, double *b,
         double *c, int n) {
  memset(c, 0, n*n*sizeof(double));

  int i, j, k, i1, j1, k1;
  for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
      for (k = 0; k < n; k+=B)
        for (i1 = i; i1 < i+B; i1++)
          for (j1 = j; j1 < j+B; j1++)
            for (k1 = k; k1 < k+B; k1++)
              c[i1*n+j1] += a[i1*n + k1]
                          * b[k1*n + j1];
}
```





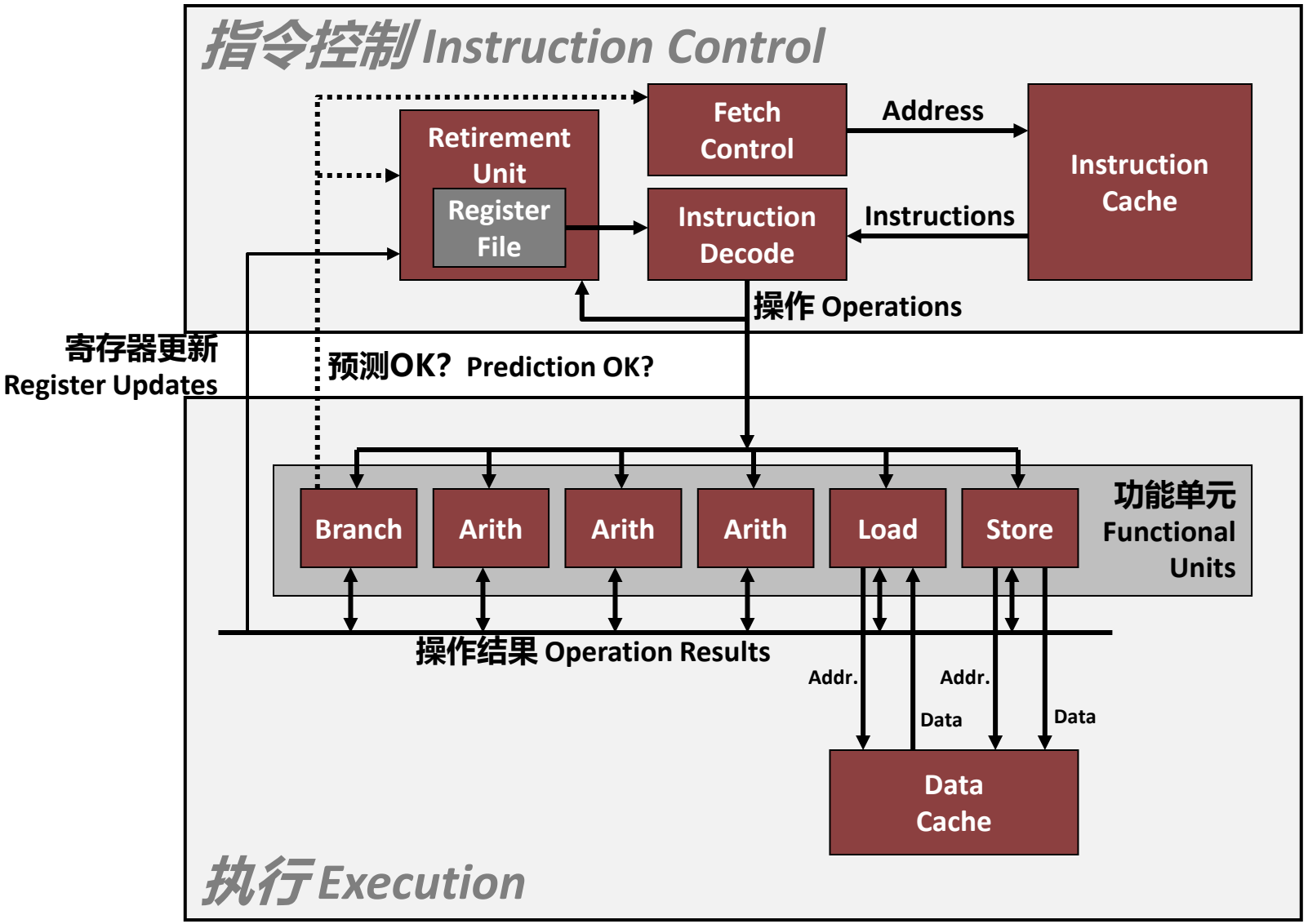
编译器不能自动做这类转换

Compiler cannot do this transformation automatically

议题 Today

- **编译器优化的原理和目标 Principles and goals of compiler optimization**
- **局部优化 Local optimization**
  - 常量折叠、强度削弱、死代码删除、公共子表达式删除cse Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **全局优化 Global optimization**
  - 内联、代码外提、循环转换 Inlining, code motion, loop transformations
- **优化的障碍 Obstacles to optimization**
  - 内存别名、过程调用、非可结合运算 Memory aliasing, procedure calls, non-associative arithmetic
- **机器相关优化 Machine-dependent optimization**
  - 分支预测、循环展开、调度、向量化 Branch predictability, loop unrolling, scheduling, vectorization

现代CPU设计 Modern CPU Design



超标量处理器 Superscalar Processor

- **定义：超标量处理器可以在*一个周期内*发出和执行*多条指令*。指令是从顺序指令流中检索的，通常是动态调度的。**
  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.

- **优点：无需编程，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* Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have**

- **大多数现代CPU是超标量的 Most modern CPUs are superscalar.**
- **从奔腾开始 Intel: since Pentium (1993)**

分支是一个挑战 Branches Are A Challenge

- **指令控制单元必须在执行单元之前工作，以产生足够的操作，使执行单元EU保持繁忙** 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
```

正在执行 Executing

需要知道向哪分支 Need to know which way to branch …

如果CPU在继续取指令之前必须等待cmp指令的结果，这样会浪费几十个周期无事可做 If the CPU has to wait for the result of the cmp before continuing to fetch instructions, may waste tens of cycles doing nothing!

分支结果 Branch Outcomes

- **遇到条件分支时不能确定从哪继续取指 When encounter conditional branch, cannot determine where to continue fetching**
  - 选择分支：转换控制到分支目标处 Branch Taken: Transfer control to branch target
  - 不选择分支：继续按顺序执行下一条指令 Branch Not-Taken: Continue with next instruction in sequence
- **只有分支/整数单元确定结果之后才知道 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
```

不选择分支
Branch Not-Taken

选择分支 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
    - 并不实际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
```

预测选择 Predict Taken

开始执行 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

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

i = 99

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

i = 100

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

i = 101

假设 Assume
向量长度 vector length = 100

预测选择（OK）
Predict Taken (OK)

预测选择（Oops）
Predict Taken(Oops)

读非法位置 Read invalid location

执行 Executed

取指 Fetched

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

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

i = 98

假设 Assume
向量长度 vector length = 100

预测选择（OK）
Predict Taken (OK)

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

i = 99

预测选择（Oops）
Predict Taken(Oops)

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

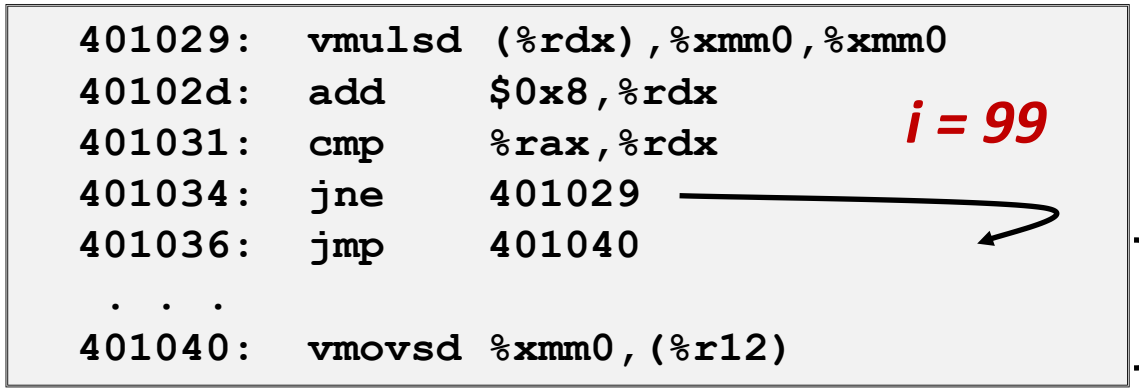
i = 100

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

i = 101

无效 Invalidate

分支预测失败的恢复 Branch Misprediction Recovery



确定不选择
Definitely not taken

重新装载流水线
Reload
Pipeline

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

分支预测现状 Branch Prediction Numbers

- **简单启发式 A simple heuristic:**
  - 向后分支通常是循环，所以预测选择分支 Backwards branches are often loops, so predict taken
  - 向前分支通常是if语句，所以预测不选择分支 Forwards branches are often ifs, so predict not taken
  - 仅这种方案就可以取得95%以上的预测准确率 >95% prediction accuracy just with this!
- **Fancier算法跟踪每个分支的行为 Fancier algorithms track behavior of each branch**
  - 正在进行的研究主题 Subject of ongoing research
  - 2011年记录：每1000条指令34.1个预测失误 2011 record (https://www.jilp.org/jwac-2/program/JWAC-2-program.htm): 34.1 mispredictions per 1000 instructions
  - 目前的研究重点是剩余的少数“不可能预测”分支（强烈依赖数据，与历史无关）Current research focuses on the remaining handful of “impossible to predict” branches (strongly data-dependent, no correlation with history)
    - 例如 e.g. https://hps.ece.utexas.edu/pub/PruettPatt_BranchRunahead.pdf

分支预测优化Optimizing for Branch Prediction

- **减少分支数 Reduce # of branches**
  - 转换循环 Transform loops
  - 循环展开 Unroll loops
  - 使用条件传送 Use conditional moves
    - 并不总是好主意 Not always a good idea
- **做分支预测 Make branches predictable**
  - 排序情况 Sort data
    https://stackoverflow.com/questions/11227809
  - 避免间接分支 Avoid indirect branches
    - 函数指针 function pointers
    - 虚方法 virtual methods

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    leal   -65(%rdx), %ecx
    cmpb   $25, %cl
    ja     .Lskip
    addl   $32, %edx
    movb   %dl, 0(%rbp,%rbx)
.Lskip:
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    movl   %edx, %esi
    leal   -65(%rdx), %ecx
    addl   $32, %edx
    cmpb   $25, %cl
    cmova  %esi, %edx
    movb   %dl, 0(%rbp,%rbx)
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

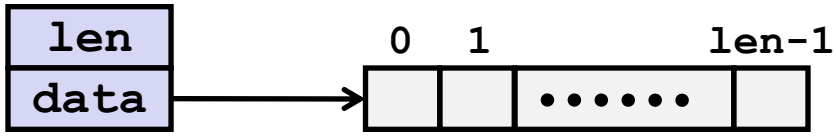
现在是无条件的内存写
Memory write now unconditional!

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分配律，无法受益 Lack of associativity and distributivity in floating-point arithmetic

基准测试样例：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;
}
```

基准测试计算 Benchmark Computation

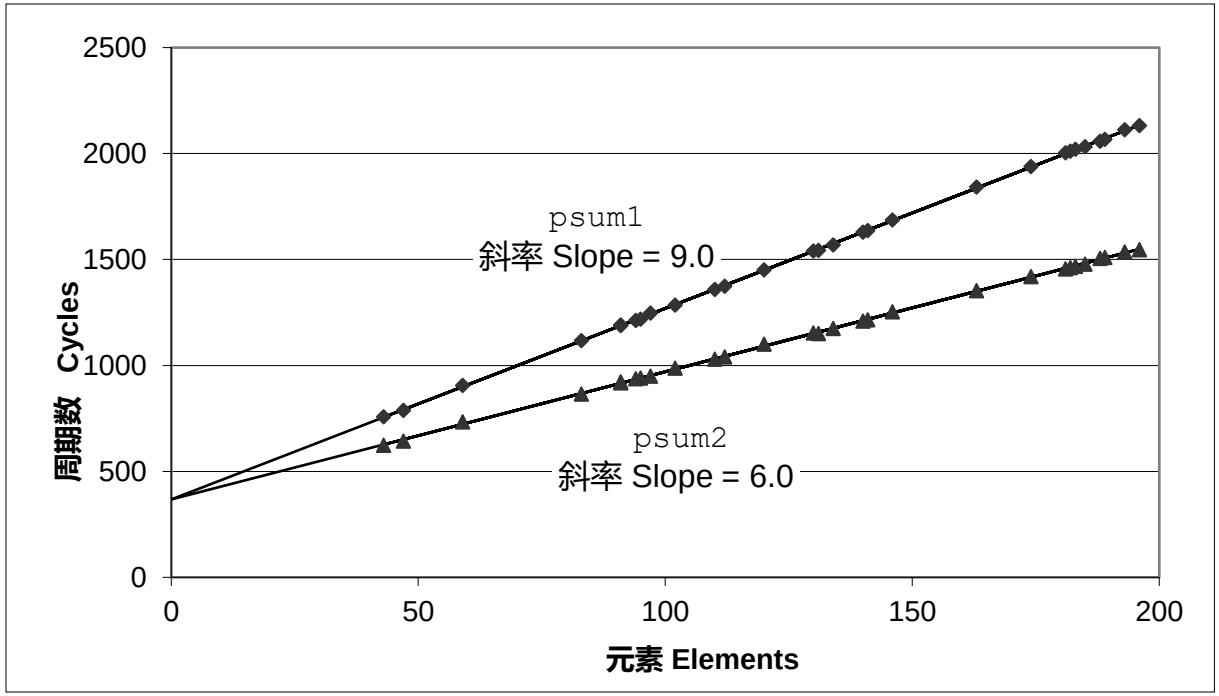
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

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

每个元素的周期数（CPE）
Cycles Per Element (CPE)

- 便于表述向量或者列表操作程序的性能 Convenient way to express performance of program that operates on vectors or lists
- Length = n
- CPE = cycles per OP 每个操作占用的周期数
- T = CPE*n + Overhead CPE*n+开销
  - CPE 是线的斜率 CPE is slope of line



基准测试性能 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

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 Mult
没优化 Combine1 unoptimized	22.68	20.02	19.98	20.18
优化 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

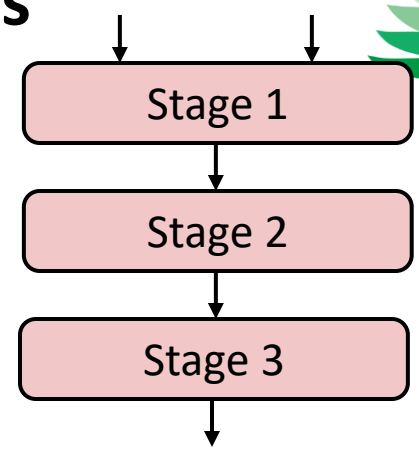
```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 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

流水功能单元 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

- 把计算分成若干阶段 Divide computation into stages
- 在不同阶段之间传递部分计算 Pass partial computations from stage to stage
- 一旦将值传递给i+1阶段，第i阶段可以开始新的计算 Stage i can start on new computation once values passed to i+1
- 例如7个周期完成三个乘法，即使每个乘法需要3个周期 E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Haswell CPU

- 总计8个功能单元 8 Total Functional Units

- **多条指令可以并行执行 Multiple instructions can execute in parallel**

  2条load指令，带地址计算 2 load, with address computation
  1条store指令，带地址计算 1 store, with address computation
  4条整数指令 4 integer
  2条浮点乘法指令 2 FP multiply
  1条浮点加法指令 1 FP add
  1条浮点除法指令 1 FP divide

- **有些指令占用1个以上周期，但是可以流水线化 Some instructions take > 1 cycle, but can be pipelined**

指令 Instruction	*时延 Latency*	*周期/发射 Cycles/Issue*
装载/存储 Load / Store	4	1
整数乘法 Integer Multiply	3	1
整数/长整数除法 Integer/Long Divide	**3-30**	**3-30**
单/双精度浮点乘法 Single/Double FP Multiply	5	1
单/双精度浮点加法 Single/Double FP Add	3	1
单/双精度浮点除法 Single/Double FP Divide	**3-15**	**3-15**

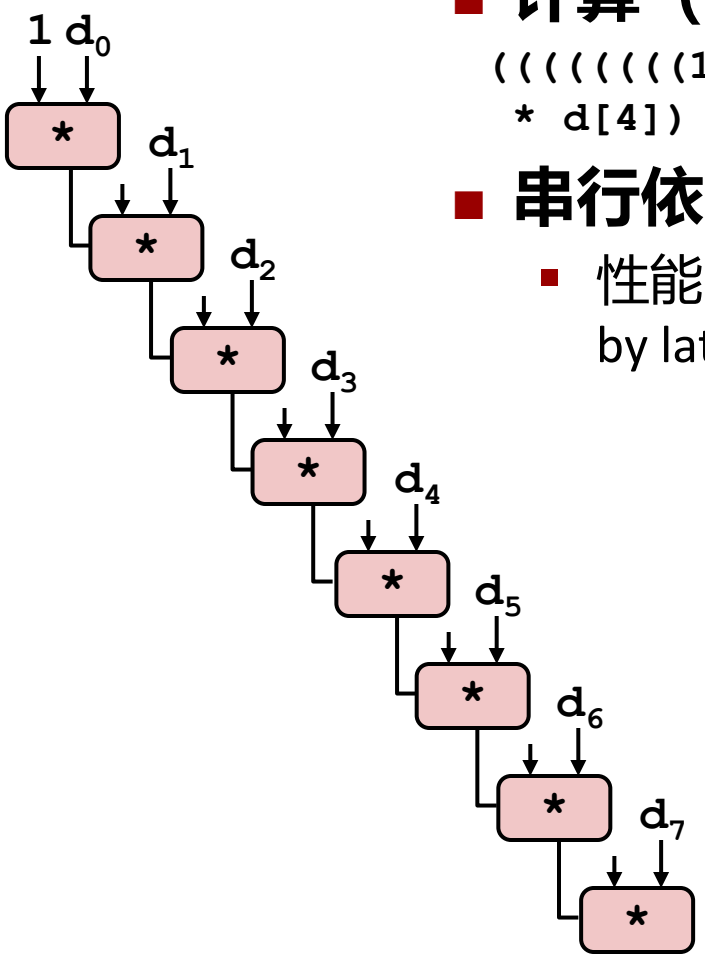
Combine4 对应的x86-64汇编
x86-64 Compilation of Combine4

- 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
```

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 Mult
Combine4	1.27	3.01	3.01	5.01
延迟界限 Latency Bound	1.00	3.00	3.00	5.00

Combine4 =串行计算 Serial Computation (OP = *)



- 计算（长度=8） 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

- 通过复制循环体分摊循环条件成本 Amortize cost of loop condition by duplicating body
- 为CSE、代码移动和调度创造机会 Creates opportunities for CSE, code motion, scheduling
- 为向量化准备代码 Prepares code for vectorization
- 增加代码大小可能会影响性能 Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

何时这种改变不正确?

When would this change be incorrect?

当元素个数不是4的倍数时

when the number of elements is not a multiple of 4.

调度 Scheduling

- **重新安排指令顺序，以便于CPU让所有功能单元保持忙碌**
  Rearrange instructions to make it easier for the CPU to keep all functional units busy
- **例如，把所有的装载操作移到未展开循环的开始 For instance, move all the loads to the top of an unrolled loop**
  - 现在这可能是为何需要很多寄存器的原因 Now maybe it's more obvious why we need lots of registers

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

何时这种变化不正确?
When would *this* change be incorrect?
当A和B或C有重叠时
when A overlaps B or C

向量化 Vectorization

- **使用特殊的指令，这些指令可以一次对几个数组元素进行操作 Use special instructions that operate on several array elements at once**
  - 经常称为“SIMD”即单指令流多数据流 Often called “SIMD” for “Single Instruction Multiple Data”
  - 1966年ILLIAC IV超级计算机发明 Invented in 1966 for ILLIAC IV supercomputer
  - 对语音和视频处理非常有价值；已经得到广泛使用 Valuable for audio and video processing; has become ubiquitous

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

```
kkkk = _mm_set_ps1(k);
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0123 = _mm_load_ps(&B[i]);
    C0123 = _mm_load_ps(&C[i]);
    A0123 = _mm_fmadd_ps(B0123, kkkk, C0123);
    _mm_store_ps(&A[i], A0123);
}
```

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;
}
```

- 每次迭代做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

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
延迟界限Latency Bound	1.00	3.00	3.00	5.00

- **整数加法有用 Helps integer add**
  - 达到延迟界限 Achieves latency bound

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

- **为什么其他无效？ Others don't improve. *Why?***
  - 仍然存在串行依赖 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;
}
```

和之前比较 Compare to before

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

- 这样会改变计算结果吗？Can this change the result of the computation?
- 会，对于浮点数。为何？ Yes, for FP. *Why?*

重结合效果 Effect of Reassociation

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 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
延迟界限 Latency Bound	1.00	3.00	3.00	5.00
吞吐量界限 Throughput Bound	0.50	1.00	1.00	0.50

- **接近2倍的加速比，对于整数*、浮点+和浮点* Nearly 2x speedup for Int *, FP +, FP ***
  - 原因：打破串行依赖 Reason: Breaks sequential dependency

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

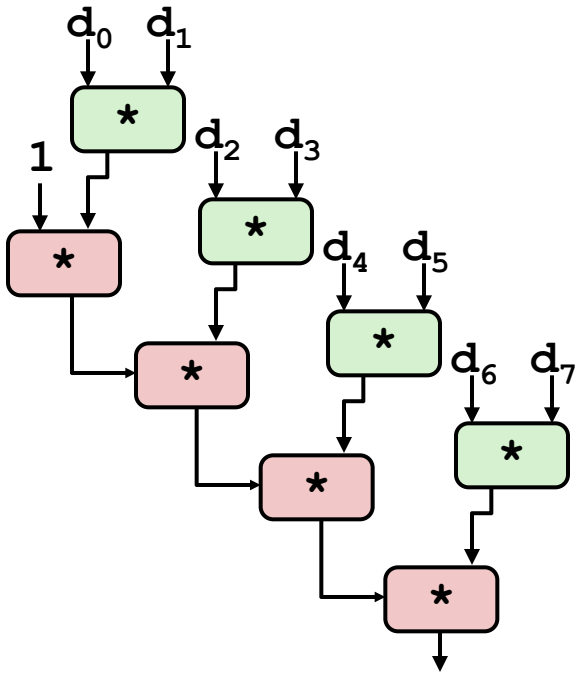
4 func. units for int +
2 func. units for load

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

  - 为何会这样？（见下一个幻灯片）Why is that? (next slide)

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（无依赖）/Ops in the next iteration can be started early (no dependency)
- **整体性能 Overall Performance**
  - N个元素，每个操作占用D个周期延迟 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;
}
```

- 不同的重结合形式 Different form of reassociation

单独累加器效果 Effect of Separate Accumulators

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 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
延迟界限 Latency Bound	1.00	3.00	3.00	5.00
吞吐量界限 Throughput Bound	0.50	1.00	1.00	0.50

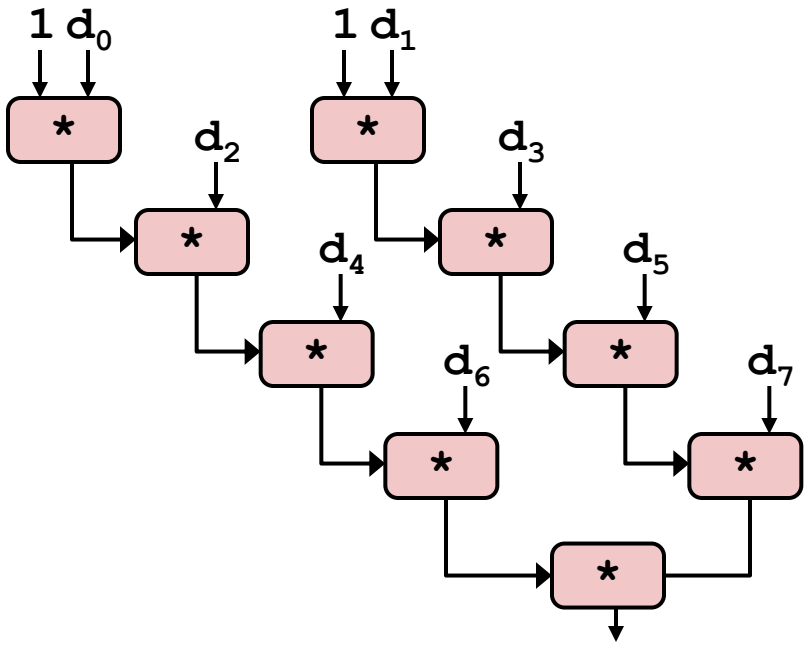
- 整数加法使用两个load单元 Int + makes use of two load units

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

- 2x加速比（相比unroll2）/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个元素，每次操作D个周期延迟 N elements, D cycles latency/op
  - 应该为 Should be (N/2+1)*D cycles:
    CPE = D/2
  - CPE符合预期 CPE matches prediction!

现在是多少？ What Now?

循环展开&累加 Unrolling & Accumulating

- **主要思路 Idea**
  - 可以展开到任何度L Can unroll to any degree L
  - 可以并行累积K个结果 Can accumulate K results in parallel
  - L必须是K的倍数 L must be multiple of K

- **局限性 Limitations**
  - 收益递减 Diminishing returns
    - 不能超过执行单元的吞吐量限制 Cannot go beyond throughput limitations of execution units
  - 长度较短时开销很大 Large overhead for short lengths
    - 按顺序完成迭代 Finish off iterations sequentially

展开与累加 Unrolling & Accumulating: Double *

- **案例 Case**
  - Intel Haswell
  - 双精度浮点乘法 Double FP Multiplication
  - 延迟界限 Latency bound: 5.00. 吞吐量界限 Throughput bound: 0.50

累加器 Accumulators

浮点乘法 FP *	展开因子为L 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

展开与累加 Unrolling & Accumulating: Int +

- **案例 Case**
  - Intel Haswell
  - 整数加法 Integer addition
  - 延迟界限 Latency bound: 1.00. 吞吐量界限 Throughput bound: 0.5

累加器 Accumulators

整数加Int +	展开因子为L 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

性能收益 Achievable Performance

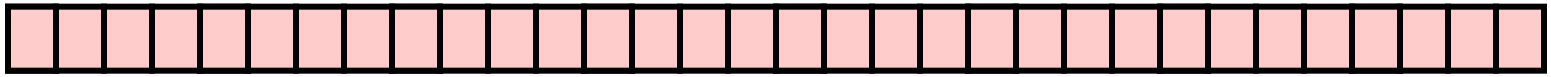
方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 Mult
最好 Best	0.54	1.01	1.01	0.52
延迟界限 Latency Bound	1.00	3.00	3.00	5.00
吞吐量界限 Throughput Bound	0.50	1.00	1.00	0.50

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

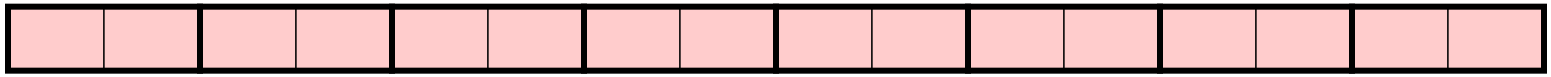
基于AVX2的编程 Programming with AVX2

YMM寄存器 YMM Registers

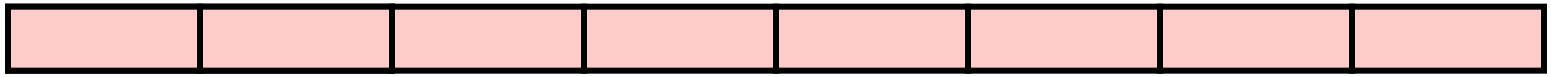
- 共计16个寄存器，每个32字节 16 total, each 32 bytes
- 32个单字节整数 32 single-byte integers



- 16个16位整数 16 16-bit integers

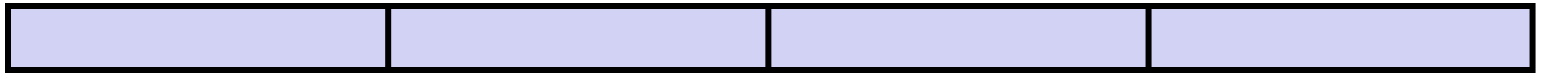


- 8个32位整数 8 32-bit integers



- 8个单精度浮点数 8 single-precision floats

- 4个双精度浮点数 4 double-precision floats



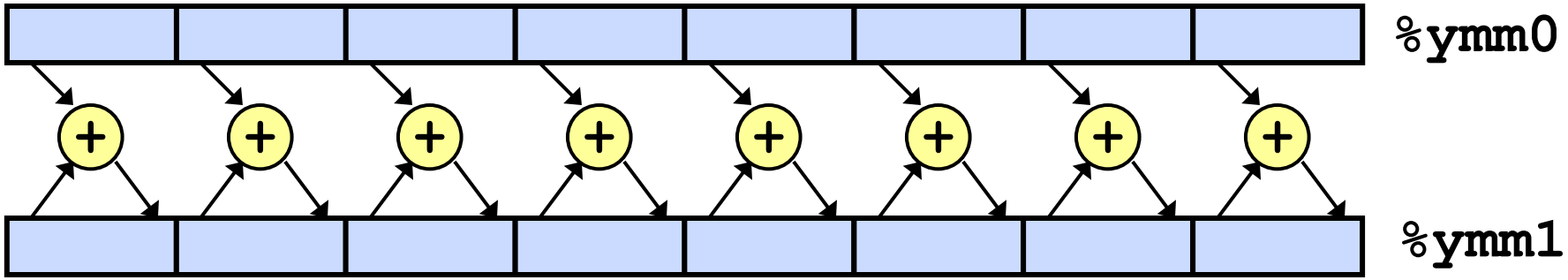
- 1个单精度浮点数 1 single-precision float

- 1个双精度浮点数 1 double-precision float

SIMD操作 SIMD Operations

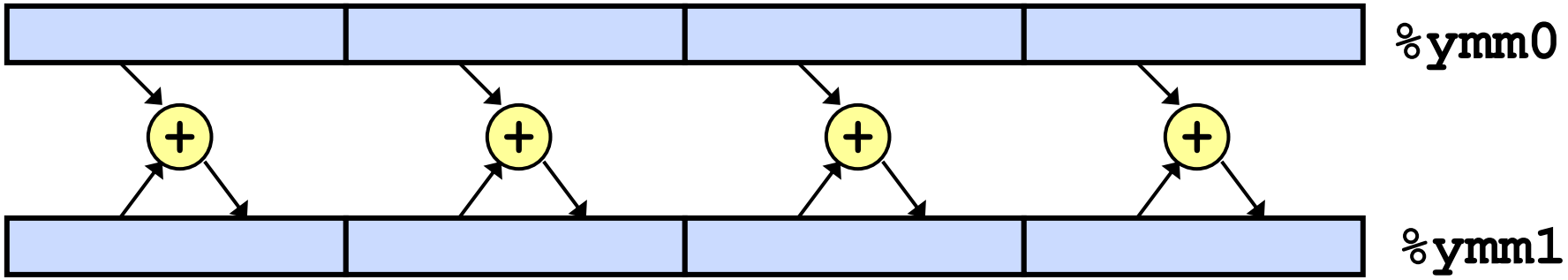
- SIMD操作：单精度 SIMD Operations: Single Precision

```
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

方法 Method	整数 Integer		双精度浮点数 Double FP	
操作 Operation	加法 Add	乘法 Mult	加法 Add	乘法 Mult
标量最好 Scalar Best	0.54	1.01	1.01	0.52
向量最好 Vector Best	0.06	0.24	0.25	0.16
延迟界限 Latency Bound	0.50	3.00	3.00	5.00
吞吐量界限 Throughput Bound	0.50	1.00	1.00	0.50
向量吞吐量界限 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
  - 参见网站旁注OPT： See Web Aside OPT:SIMD on CS:APP web page

获得更高的性能 Getting High Performance

- **好的编译器和标志 Good compiler and flags**
- **不做任何愚蠢的事 Don't do anything stupid**
  - 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：过程调用和内存引用 Watch out for optimization blockers: procedure calls & memory references
  - 注意最内层循环（最内循环做大多数工作） 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（课程后面讲述） 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