程序的性能优化

王晶

jwang@ruc.edu.cn，信息楼124

2024年12月

提纲

- 概述
- 常用的优化技术
  - 代码移动/预计算
  - 减小强度
  - 共享相同子表达式
  - 示例：冒泡排序
- 优化的阻碍
  - 函数调用
  - 内存别名
- 指令级并行
- 处理条件判断

关于性能

- *There's more to performance than asymptotic complexity*
  （复杂度）
- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - 算法、数据类型、函数、循环
- Must understand system to optimize performance
  - How programs are compiled and executed，编译
  - How modern processors + memory systems operate，架构+存储
  - How to measure program performance and identify bottlenecks，瓶颈分析
  - How to improve performance without destroying code modularity and generality，模块化和通用性

优化编译器

- Provide efficient mapping of program to machine
  - register allocation寄存器分配
  - code selection and ordering (scheduling)代码选择和调度
  - dead code elimination消除无效代码
  - eliminating minor inefficiencies消除低效代码
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

提纲

- 概述
- 常用的优化技术
  - 代码移动/预计算
  - 减小强度
  - 共享相同子表达式
  - 示例：冒泡排序
- 优化的阻碍
  - 函数调用
  - 内存别名
- 指令级并行
- 处理条件判断

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- 代码移动Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - 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];
```

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
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

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

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- 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;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- 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;
```

3 multiplications: `i*n`, `(i–1)*n`, `(i+1)*n`

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 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
...
```

Bubblesort

比较遍数 →

数据	1	2	3	4
587	-632	-632	-632	-632
-632	587	234	-34	-34
777	234	-34	234	234
234	-34	587	587	587
-34	777	777	777	777

↓ 从小到大排序

Optimization Example: Bubblesort

- **Bubblesort program that sorts an array `A` that is allocated in static storage:**
  - an element of **`A`** requires four bytes of a byte-addressed machine
  - elements of **`A`** are numbered 1 through **`n`** (**`n`** is a variable)
  - **`A[j]`** is in location **`&A+4*(j-1)`**

```
for (i = n-1; i >= 1; i--) {
   for (j = 1; j <= i; j++)
      if (A[j] > A[j+1]) {
         temp = A[j];
         A[j] = A[j+1];
         A[j+1] = temp;
      }
}
```

Translated (Pseudo) Code

```
    i := n-1
L5:  if i<1 goto L1
    j := 1
L4:  if j>i goto L2
    t1 := j-1
     t2 := 4*t1
     t3 := A[t2]    // A[j]
     t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    // A[j+1]
     if t3<=t7 goto L3
```

```
    t8 := j-1
    t9 := 4*t8
    temp := A[t9]   // temp:=A[j]
    t10 := j+1
    t11:= t10-1
    t12 := 4*t11
    t13 := A[t12]   // A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13   // A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp    // A[j+1]:=temp
L3: j := j+1
    goto L4
L2: i := i-1
   goto L5
L1:
```

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

Instructions

29 in outer loop

25 in inner loop

Redundancy in Address Calculation

```
     i := n-1
L5:   if i<1 goto L1
     j := 1
L4:   if j>i goto L2
     t1 := j-1
      t2 := 4*t1
      t3 := A[t2]     // A[j]
      t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]      // A[j+1]
      if t3<=t7 goto L3
```

```
      t8 :=j-1
      t9 := 4*t8
      temp := A[t9]   // temp:=A[j]
      t10 := j+1
      t11:= t10-1
      t12 := 4*t11
      t13 := A[t12]   // A[j+1]
      t14 := j-1
      t15 := 4*t14
      A[t15] := t13   // A[j]:=A[j+1]
      t16 := j+1
      t17 := t16-1
      t18 := 4*t17
      A[t18]:=temp    // A[j+1]:=temp
L3:   j := j+1
      goto L4
L2:  i := i-1
    goto L5
L1:
```

Redundancy Removed

```
    i := n-1
L5: if i<1 goto L1
    j := 1
L4: if j>i goto L2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      // A[j]
    t6 := 4*j
    t7 := A[t6]      // A[j+1]
    if t3<=t7 goto L3
```

```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]    // temp:=A[j]
    t12 := 4*j
    t13 := A[t12]    // A[j+1]
    A[t9]:= t13      // A[j]:=A[j+1]
    A[t12]:=temp     // A[j+1]:=temp
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:
```

Instructions

20 in outer loop

16 in inner loop

More Redundancy

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t6 := 4*j
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
```

```
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]    // temp:=A[j]
     t12 := 4*j
     t13 := A[t12]    // A[j+1]
     A[t9]:= t13      // A[j]:=A[j+1]
     A[t12]:=temp     // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Removed

```
    i := n-1                                  A[t2] := t7    // A[j]:=A[j+1]
L5: if i<1 goto L1                            A[t6] := t3    // A[j+1]:=old_A[j]
    j := 1
L4: if j>i goto L2                        L3: j := j+1
    t1 := j-1                                 goto L4
    t2 := 4*t1                            L2: i := i-1
    t3 := A[t2]     // old_A[j]               goto L5
    t6 := 4*j                             L1:
    t7 := A[t6]     // A[j+1]
    if t3<=t7 goto L3
```

Instructions

15 in outer loop

11 in inner loop

Redundancy in Loops

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     // A[j]
     t6 := 4*j
     t7 := A[t6]     // A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Eliminated

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     // A[j]
     t6 := 4*j
     t7 := A[t6]     // A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := 4*i
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Final Pseudo Code

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := i << 2
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instruction Count
Before Optimizations
29 in outer loop
25 in inner loop

Instruction Count
After Optimizations
15 in outer loop
9 in inner loop

- These were **Machine-Independent Optimizations**.
- Will be followed by **Machine-Dependent Optimizations**, including allocating temporaries to registers, converting to assembly code

提纲

- 概述
- 常用的优化技术
  - 代码移动/预计算
  - 减小强度
  - 共享相同子表达式
  - 示例：冒泡排序
- 优化的阻碍
  - 函数调用
  - 内存别名
- 指令级并行
- 处理条件判断

优化编译器的能力和局限性

- GCC胜任基本的优化，不适合执行激进的变换
- GCC程序员需要强化代码优化，以简化编译器生成高效代码任务的方式编写程序

GCC-conscious programming

编译器优化的能力和局限性

- 优化行为受到基本的约束
  - 不能改变程序行为
    - 除非程序使用非标准语言功能时
  - 通常会阻止它进行仅影响特殊条件下行为的优化
- 对程序员来说可能很明显的行为可能会被语言和编码风格混淆
  - e.g., Data ranges may be more limited than variable types suggest例如，数据范围可能比变量类型的建议范围更有限
- 大多数分析仅在程序内执行
  - 在大多数情况下，整个程序分析的成本太高
  - 较新版本的 GCC 在单个文件中执行过程间分析
    - 但是，不是在不同文件中的代码之间
- 大多数分析仅基于静态信息
  - 编译器难以预测运行时输入
- 如有疑问，编译器必须是保守的

Optimization Blocker #1: Procedure Calls

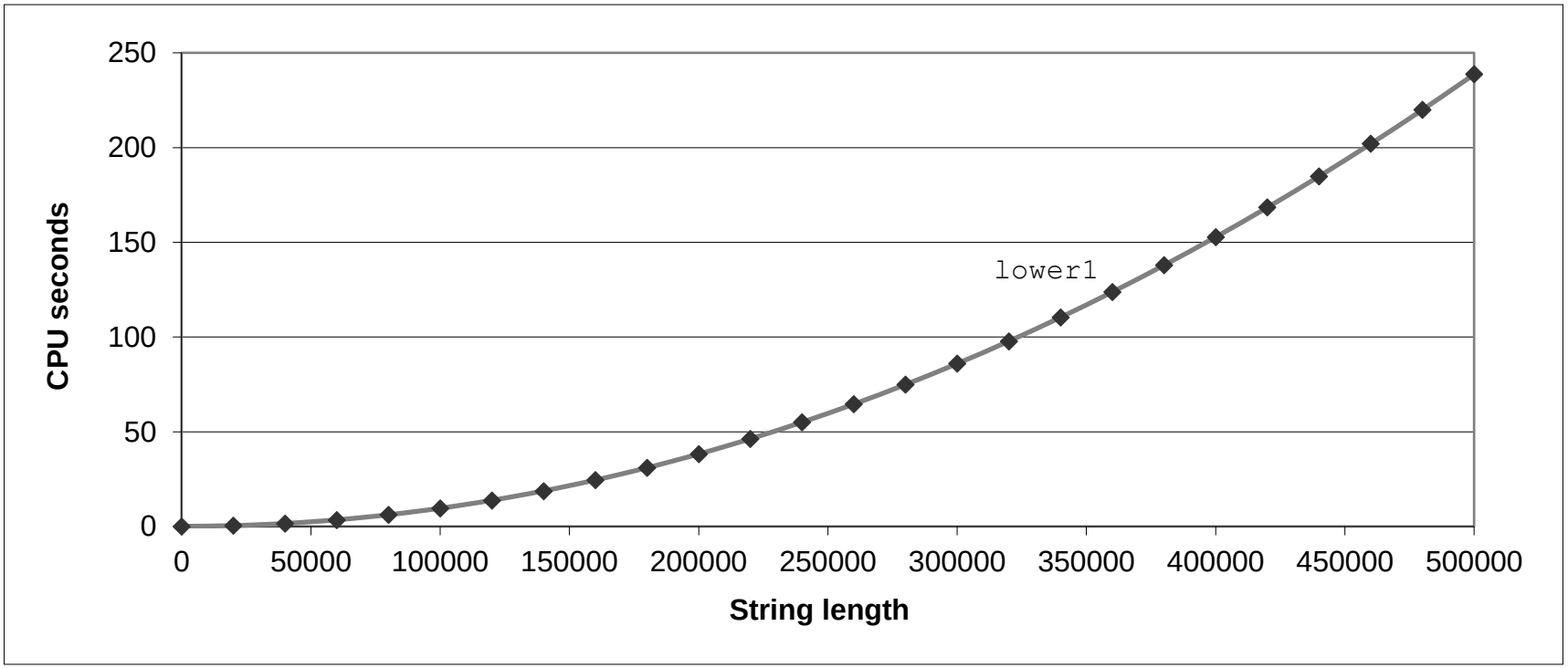
- Procedure to Convert String to Lower Case

```
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');
}
```

- Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



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` executed every iteration

Calling Strlen

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

- Strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null character.
- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

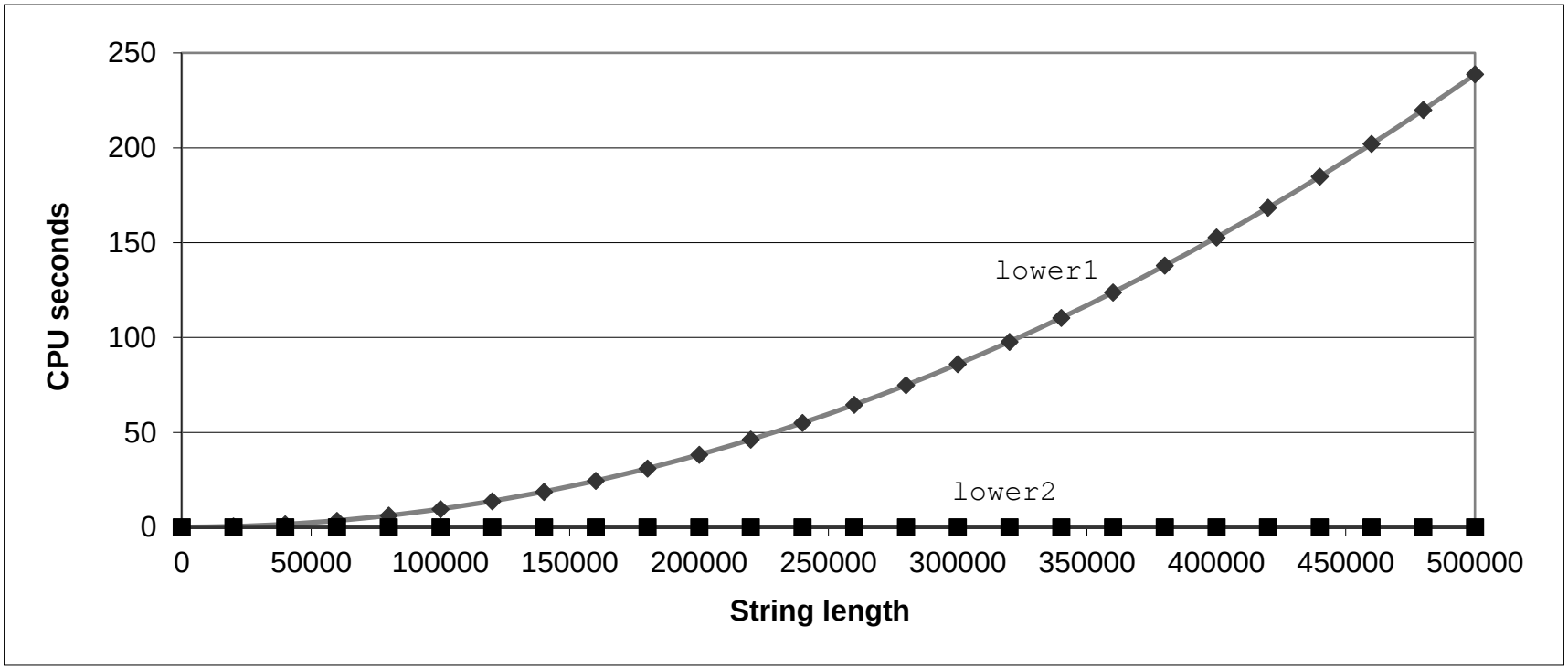
Improving Performance

```
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');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- Remedies:
  - Use of inline functions
    - GCC does this with –O1
      - Within single file
  - 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;
}
```

单选题 1分

设置

Which one is better?

```
long f()
long func1(){
        return f()+f()+f()+f();
}

long func2(){
        return 4*f();
}
```

A func1

B func2

提交

Optimization Blocker: Procedure Calls

- *Which one is better?*

```
long f()
long func1(){
        return f()+f()+f()+f();
}

long func2(){
        return 4*f();
}
```

- 函数f()中改变counter值，调用次数不同函数返回值不同。
- func1()中f()被调用4次，函数返回值分别为0，1，2，3，因此func1()返回值为0+1+2+3=6；
- func2()中只调用f()一次，因此返回值为4*0=0

- But if …

```
long counter = 0;
long f (){
        return counter++;
}
```

Optimization Blocker: Procedure Calls

- 内联函数替换函数调用
  - 使用inline关键字定义内联函数，编译器优化时将函数调用替换为内联函数体

```
inline long f(){
        return counter++;
}
```

```
long func1(){
        return f()+f()+f()+f();
}
```

- **通过内联函数替换掉对函数f()的四次调用**
- **编译器统一funclin中对全局变量counter的更新，产生优化版本funclopt();**

```
[nsrc@localhost ~]$ inlinefcall
inline f call:6
f call:6
```

```
Long func1in(){
        long t=counter++;   //0
        t+=counter++;       //1
        t+=counter++;       //2
        t+=counter++;       //3
        return t;
}
```

```
Long func1opt(){
        long t=4*counter+6; //6
        counter+=4;         //4
        return t;
}
```

练习题5.3

```
long min(long x,long y){
        cmin++;
        return x<y?x:y;
}
long max(long x,long y){
        cmax++;
        return x<y?y:x;
}
void incr(long *xp,long v){
        cincr++;
        *xp+=v;
}
long square(long x){
        csqu++;
        return x*x;
}
A.
 for (i=min(x,y);i<max(x,y);incr(&i,1))
                t+=square(i);
B.
 for (i=max(x,y);i>=min(x,y);incr(&i,-1))
                t+=square(i);
C.
 long low=min(x,y);
 long high=max(x,y);
 for (i=low;i<high;incr(&i,1))
                t+=square(i);
```

- 改进1：使用内联函数

```
inline
void inline_incr(long *xp,long v){
        cincr++;
        *xp+=v;
}
inline long inline_square(long x){
        csqu++;
        return x*x;
}
```

- 改进2：使用宏定义

```
#define SQUA(X)(X*X)
```

单选题 1分

设置

Which one is better?

```
void twiddle1()
{
        *xp += *yp;
        *xp += *yp;
}

void twiddle2()
{
        *xp += 2**yp;
}
```

A twiddle1

B twiddle2

提交

Optimization Blocker:

- *Which one is better?*

```
void twiddle1()
{
        *xp += *yp;
        *xp += *yp;
}

void twiddle2()
{
        *xp += 2**yp;
}
```

- But if xp = yp?

```
twiddle1 result:6
twiddle2 result:6
case xp=yp
twiddle1 result:8
twiddle2 result:6
```

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
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

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);
```

```
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]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

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
```

- No need to store intermediate results

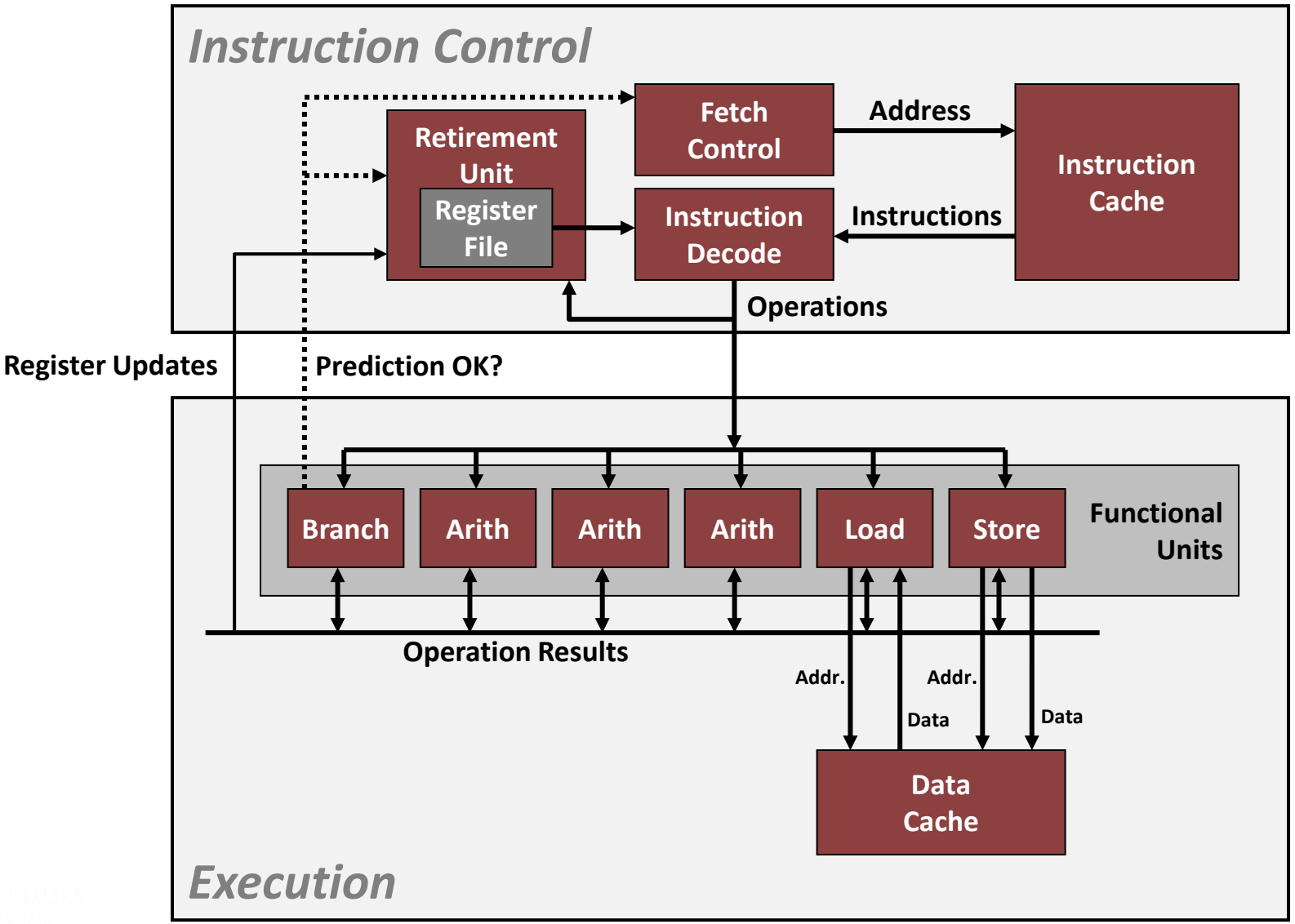
Optimization Blocker: Memory Aliasing

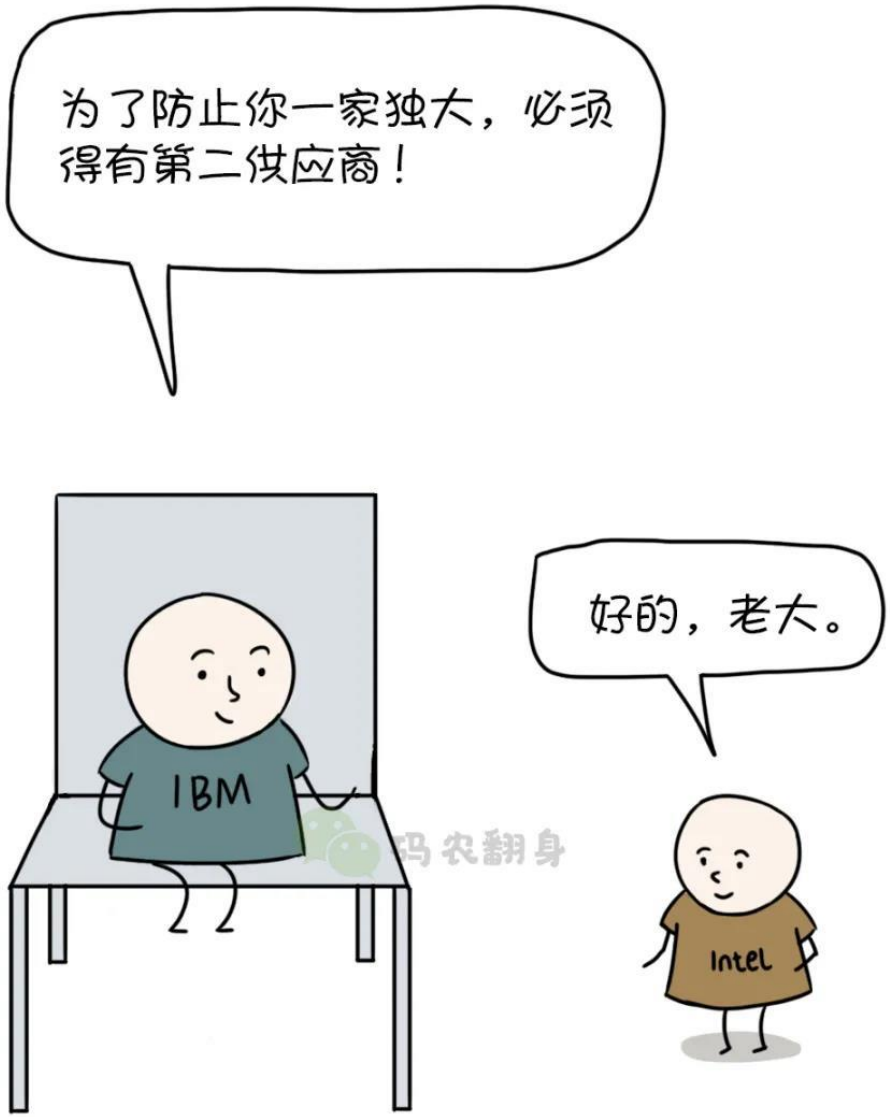
- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

提纲

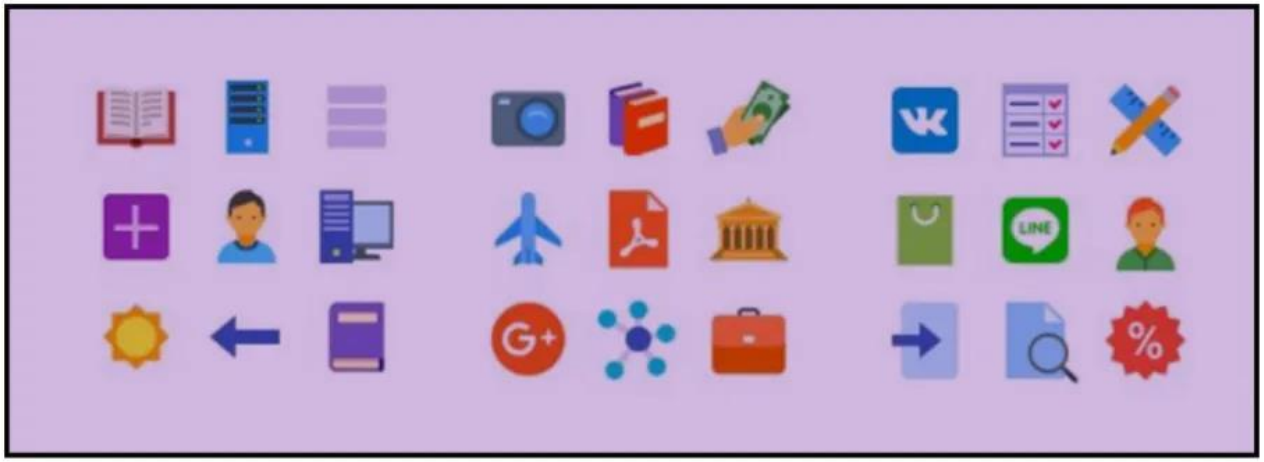
- 概述
- 常用的优化技术
  - 代码移动/预计算
  - 减小强度
  - 共享相同子表达式
  - 示例：冒泡排序
- 优化的阻碍
  - 函数调用
  - 内存别名
- 指令级并行
- 处理条件判断

Modern CPU Design




为了防止你一家独大，必须得有第二供应商！
IBM
好的，老大。
Intel


兄弟，咱们俩合伙把IBM这个生意吃下来啊！
两位大哥，也拉兄弟一把！
AMD
Intel
Cyrix



Windows	Linux	Mac

x86指令集

中国芯的发源地

中国第一款正向设计的自主CPU(1999)

指令系统体系结构

基准程序

编译、汇编

模拟环境

性能评测

结构系统指令修改

发现瓶颈

软硬件协同设计环境

可重定目标的编译器

汇编器生成器

指令系统结构评测环境

Cache性能评测工具

程序开发和调试环境

应用需求

应用程序

FPGA验证

芯片

FINISH

FPGA综合

FPGA布局布线

系统设计描述

系统划分

子模块设计描述

逻辑综合

布局布线

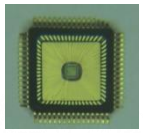
GDSII版图

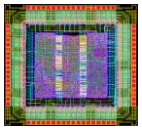
系统验证

功能验证性能考量

后验证

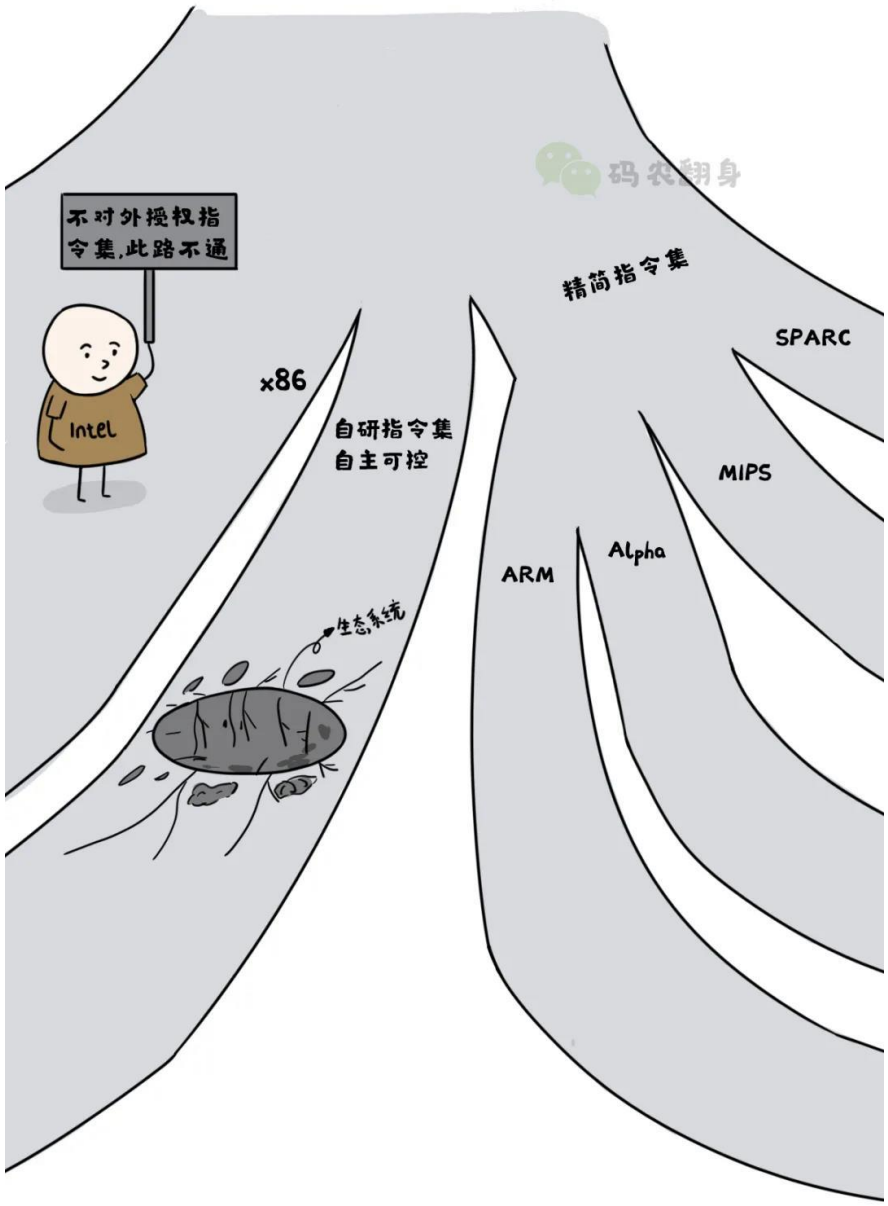
版图物理验证

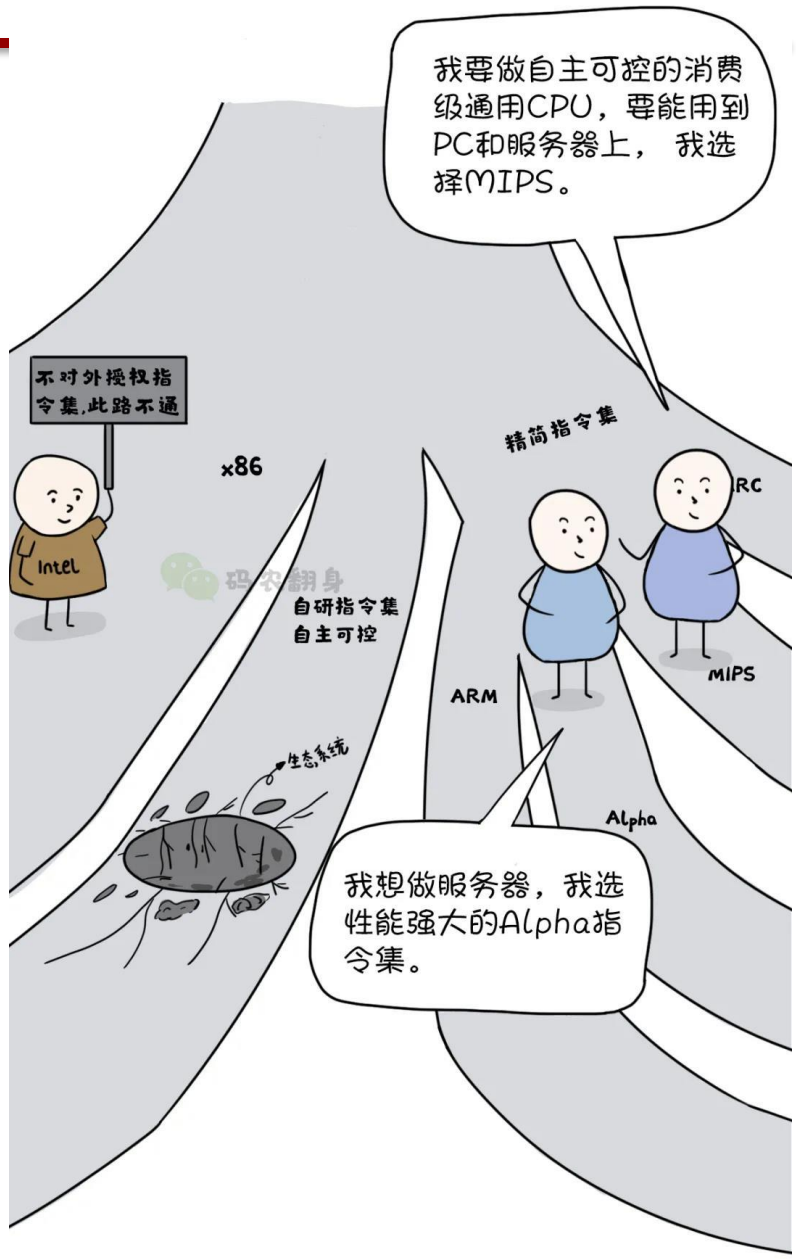


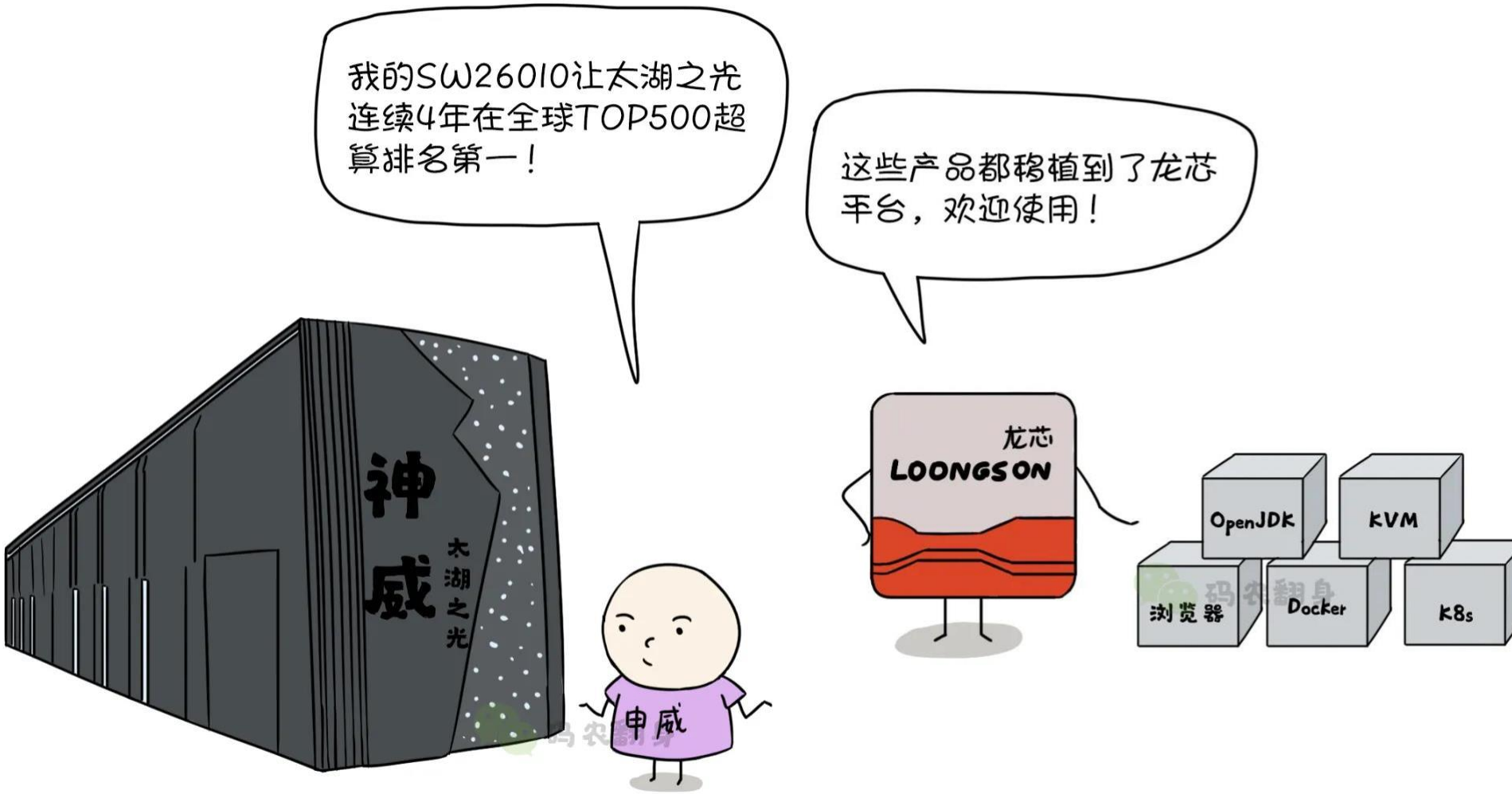


1999年12月 16位嵌入式微处理器核 50 MHz@0.35 μm

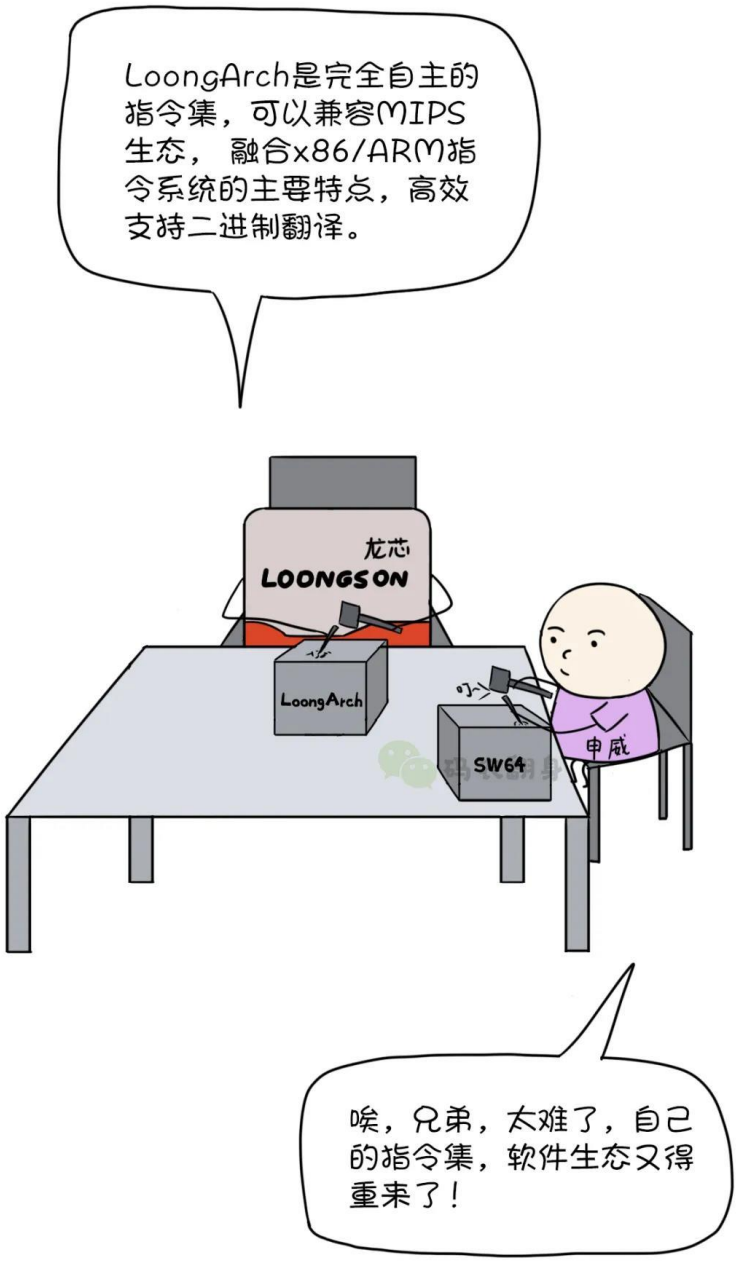
2001年1月　32位/16位微处理器核 200MHz@0.25 μm


不对外授权指令集,此路不通
Intel
x86
自研指令集
自主可控
精简指令集
SPARC
MIPS
ARM
Alpha
生态系统


我要做自主可控的消费级通用CPU，要能用到PC和服务器上， 我选择MIPS。
不对外授权指令集,此路不通
Intel
x86
自研指令集
自主可控
精简指令集
RC
MIPS
ARM
Alpha
生态系统
我想做服务器，我选性能强大的Alpha指令集。

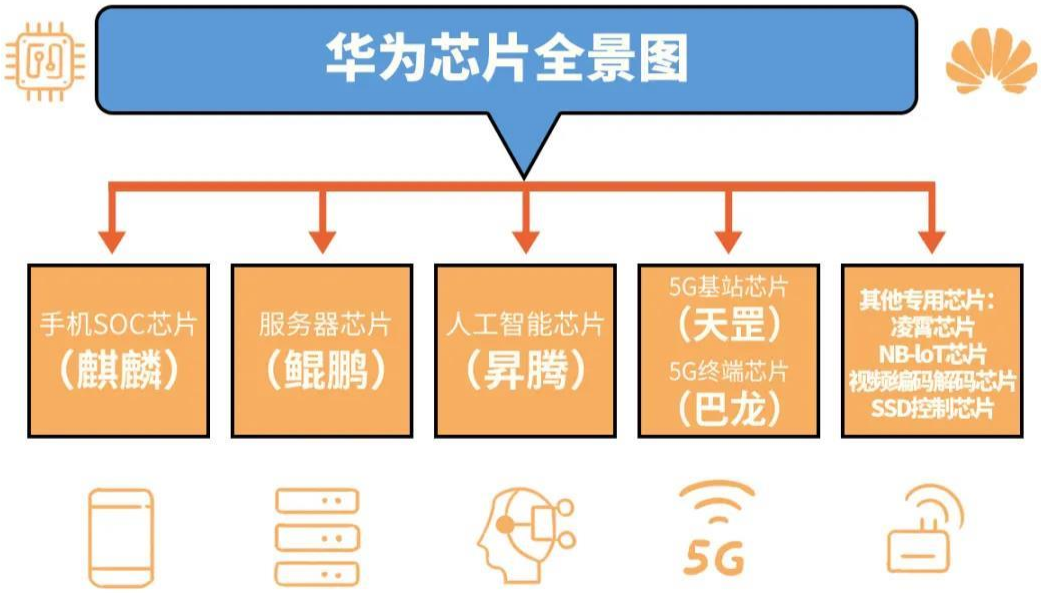

我的SW26010让太湖之光连续4年在全球TOP500超算排名第一！
这些产品都移植到了龙芯平台，欢迎使用！
神威
太湖之光
申威
龙芯
LOONGSON
OpenJDK
KVM
浏览器
Docker
k8s

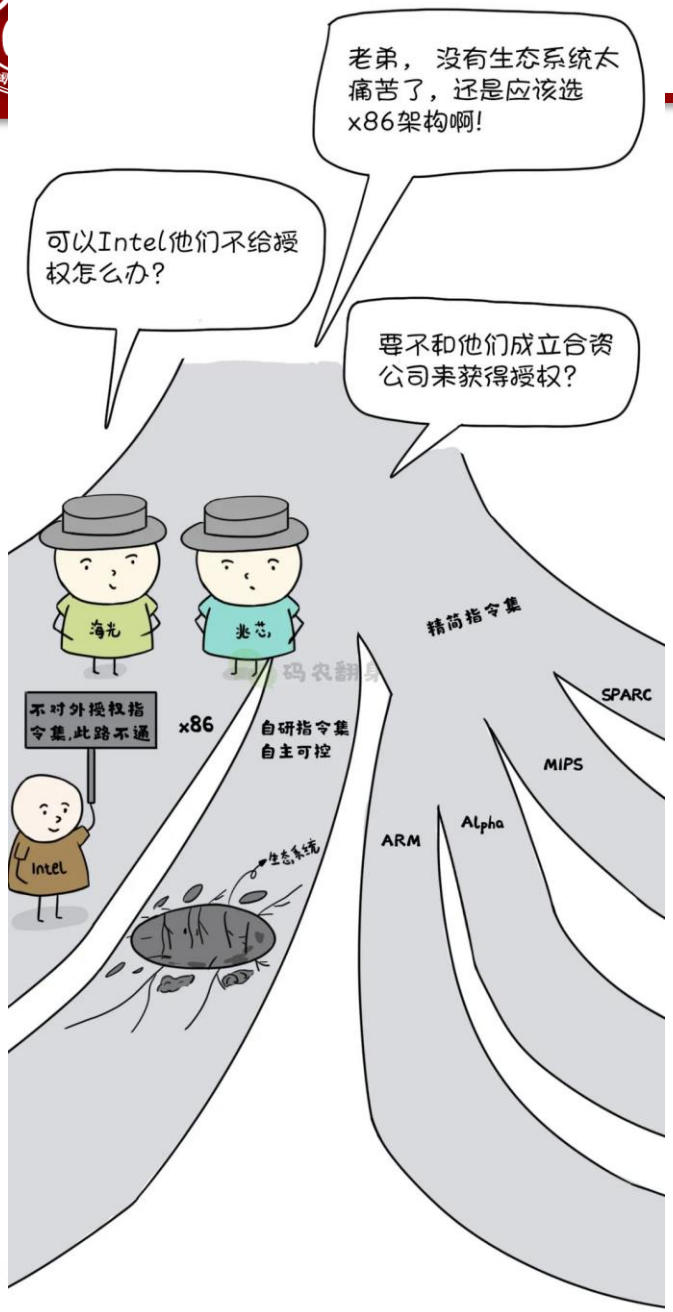

哈哈，最终还是落到了我的手中， 你的寿命也到此为止了。
Alpha
Intel
码农翻身

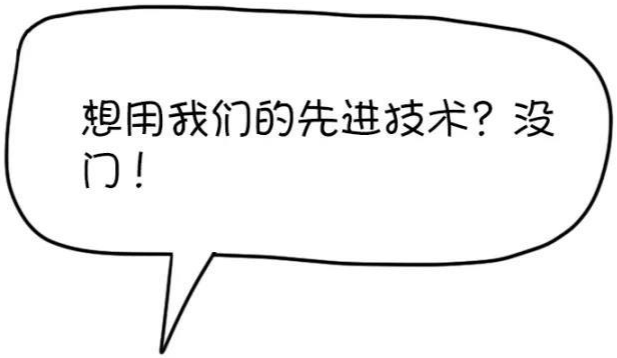

LoongArch是完全自主的指令集，可以兼容MIPS生态， 融合x86/ARM指令系统的主要特点，高效支持二进制翻译。
龙芯
LOONGSON
LoongArch
SW64
申威
唉，兄弟，太难了，自己的指令集，软件生态又得重来了！


手机没有自己的CPU是没有灵魂的，我要做手机CPU，ARM是绕不过去的。
不对外授权指

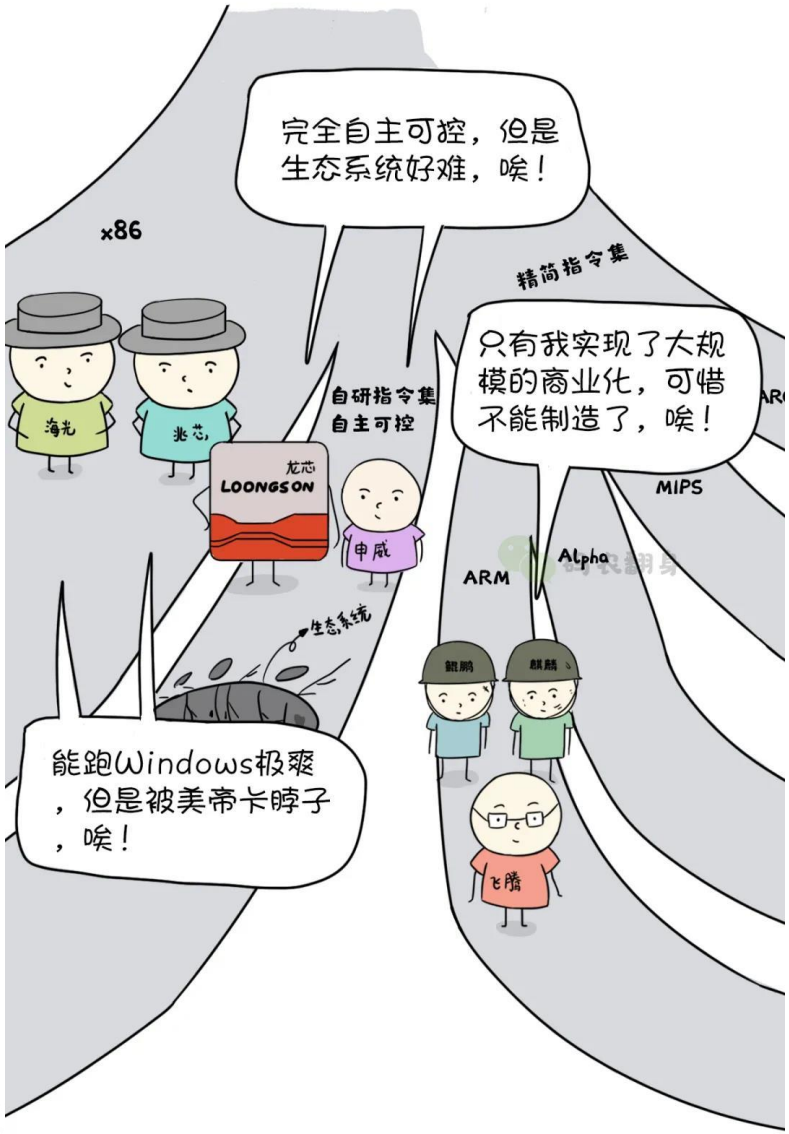

我们什么什么时候才能打出去！
等到中国有了自主的光刻机！
鲲鹏
巴龙
麒麟
只是光刻机还不行，得有去美国化的芯片制造全产业链！

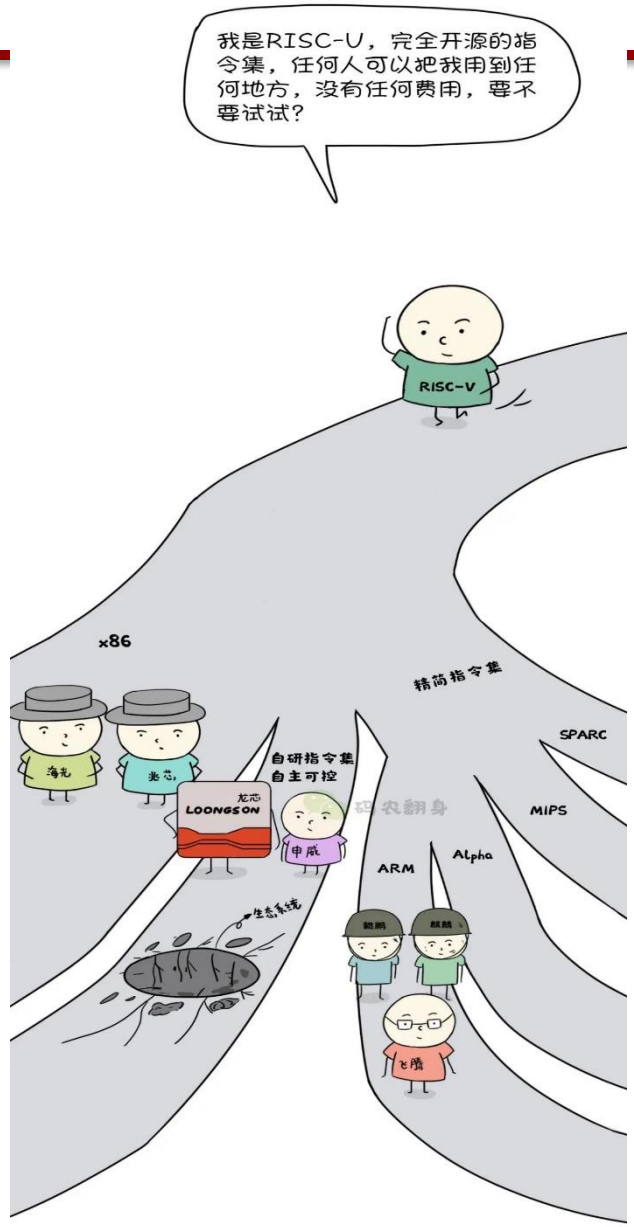

华为芯片全景图
手机SOC芯片
（麒麟）
服务器芯片
（鲲鹏）
人工智能芯片
（昇腾）
5G基站芯片
（天罡）
5G终端芯片
（巴龙）
其他专用芯片：
凌霄芯片
NB-IoT芯片
视频编解码芯片
SSD控制芯片
5G


老弟，没有生态系统太痛苦了，还是应该选x86架构啊!
可以Intel他们不给授权怎么办?
要不和他们成立合资公司来获得授权?
海光
兆芯
不对外授权指令集,此路不通
x86
自研指令集
自主可控
Intel
生态系统
精简指令集
SPARC
MIPS
ARM
Alpha


想用我们的先进技术？没门！


美国商务部
海光
实体清单


完全自主可控，但是生态系统好难，唉！
x86
精简指令集
自研指令集
自主可控
只有我实现了大规模的商业化，可惜不能制造了，唉！
海光
兆芯
龙芯
LOONGSON
申威
MIPS
ARM
Alpha
生态系统
麒麟
麒麟
飞腾
能跑Windows极爽，但是被美帝卡脖子，唉！

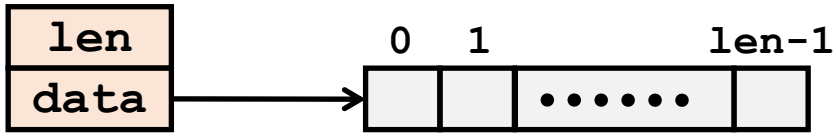

我是RISC-V，完全开源的指令集，任何人可以把我用到任何地方，没有任何费用，要不要试试？
RISC-V
x86
精简指令集
SPARC
自研指令集
自主可控
海光
兆芯
龙芯
LOONGSON
申威
MIPS
ARM
Alpha
生态系统
麒麟
麒麟
飞腾

Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Data Type for Vectors

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



- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

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

Benchmark Computation

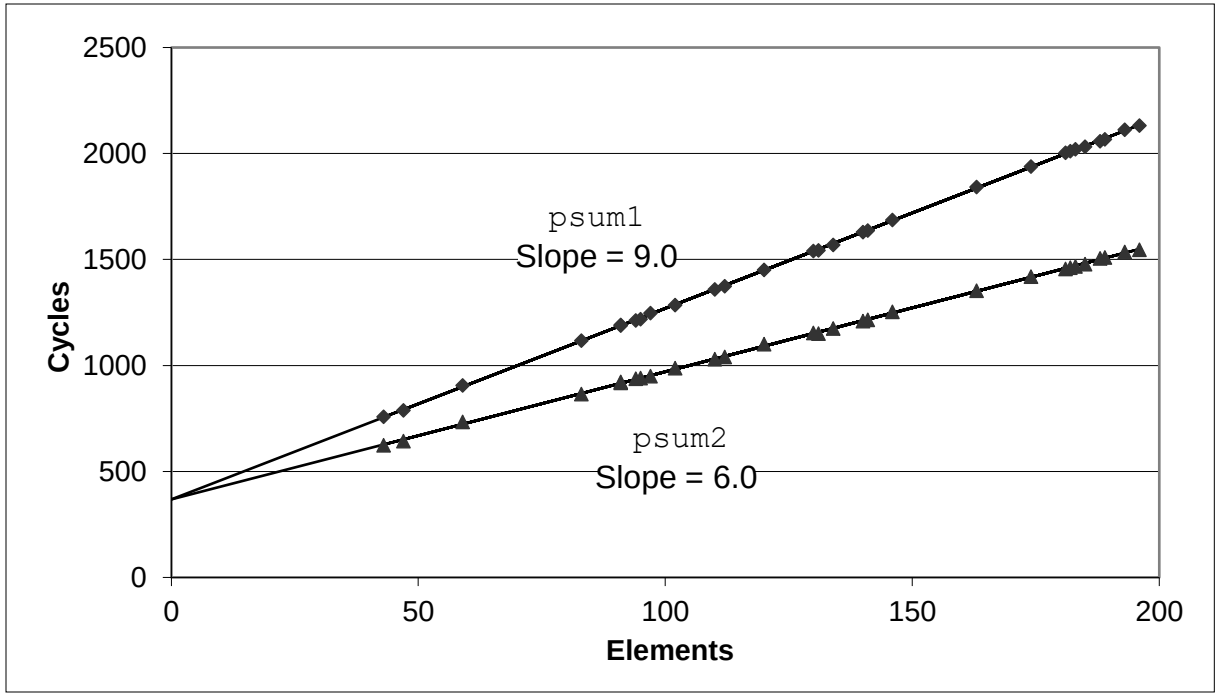
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line



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;
    }
}
```

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
Combine1 –O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

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;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

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

- Eliminates sources of overhead in loop

Superscalar Processor

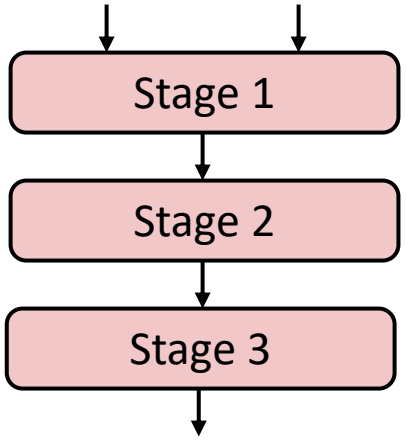
- Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

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

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

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

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Haswell CPU

- 8 Total Functional Units
- 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**

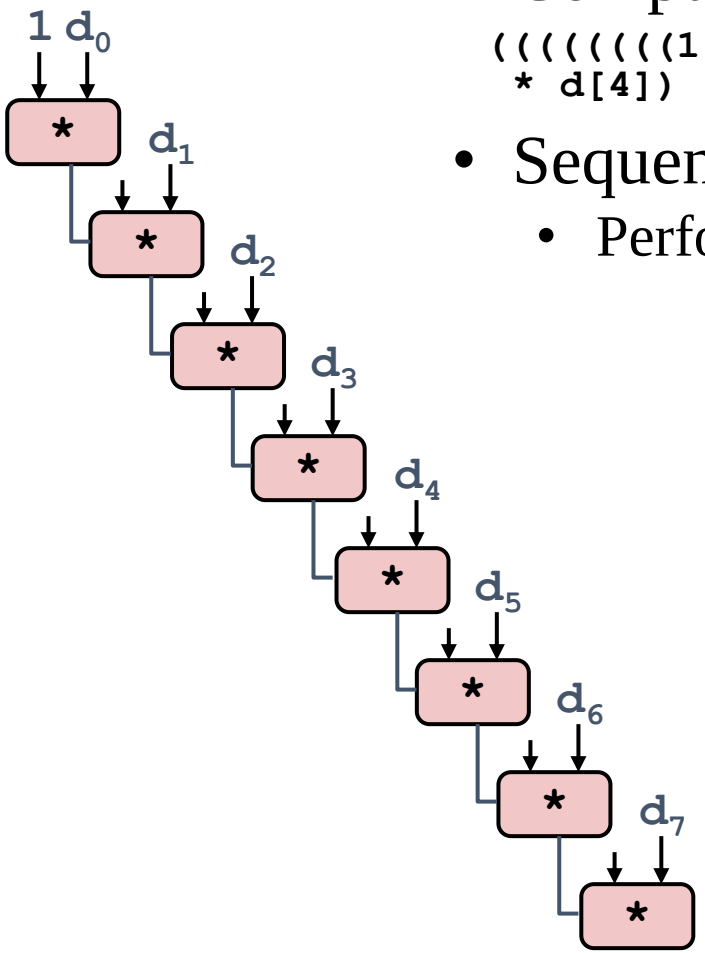
x86-64 Compilation of Combine4

- 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 = *)

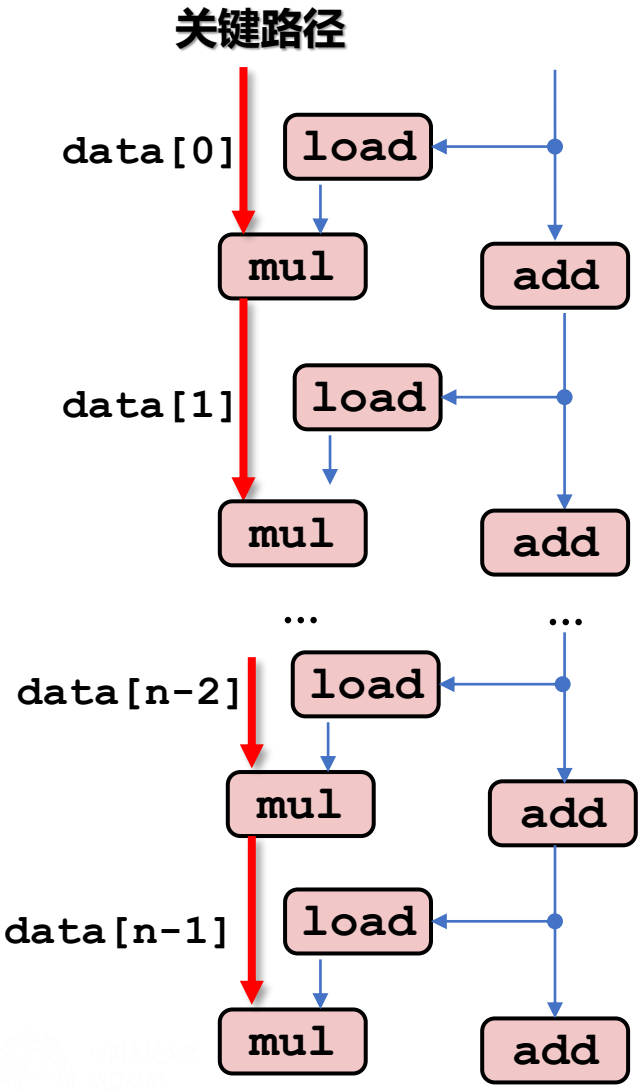


- Computation (length=8)

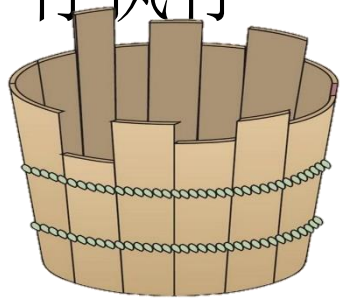
```
(((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP

处理器操作的抽象模型



- 函数combine4内循环的n次迭代数据流图
  - 两条数据相关链：mul和add对程序值acc和data+i的修改
  - 假设乘法延迟为5个周期，加法延迟为1个周期
    - 左边的链成为关键路径，需要5n个周期执行
    - 右边的链需要n个周期执行，不会制约程序性能
- 浮点乘法器成为程序制约资源，其他操作与乘法器并行执行



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;
}
```

- Perform 2x more useful work per iteration

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*

- Helps integer add
  - Achieves latency bound
- Others don't improve. *Why?*
  - Still sequential dependency

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

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;
}
```

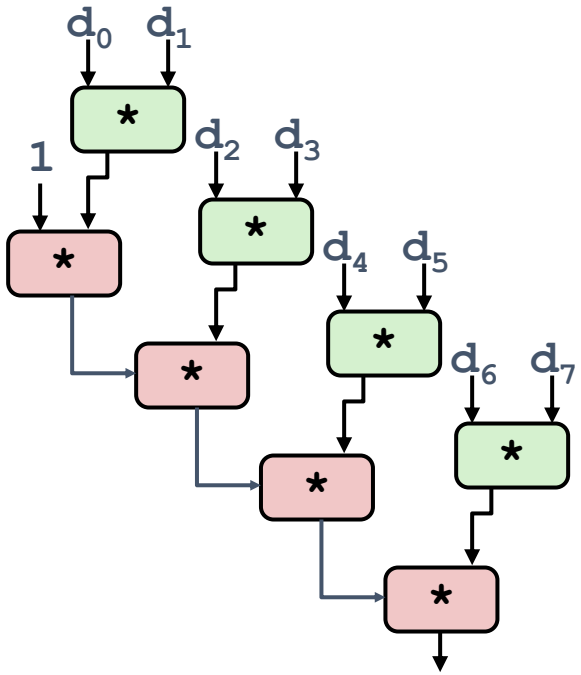
Compare to before

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

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Reassociated Computation

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



- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2

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*

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

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

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

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

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;
}
```

- Different form of reassociation

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*

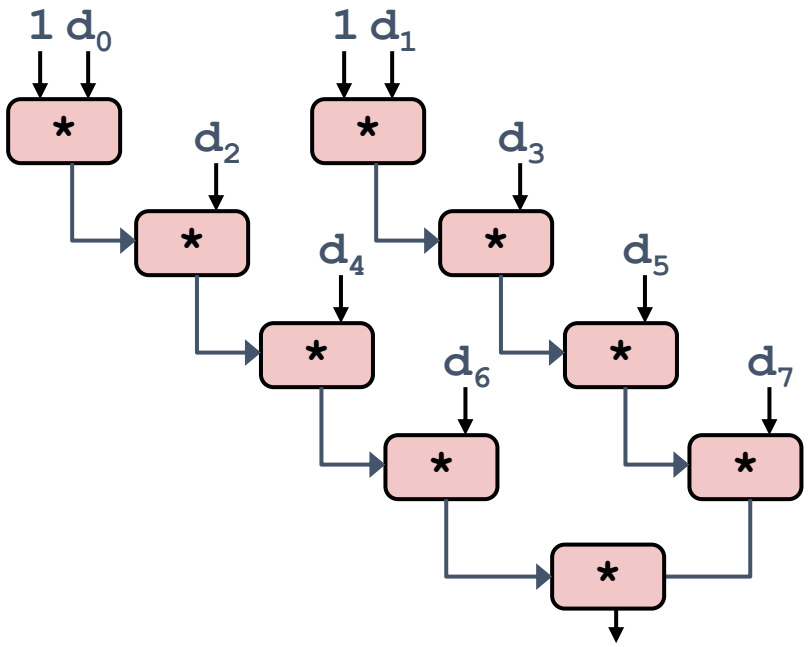
- Int + makes use of two load units

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

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

Separate Accumulators

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



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

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

Unrolling & Accumulating: Int +

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

Int +	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

Achievable Performance

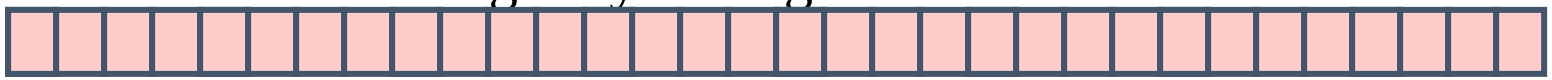
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

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

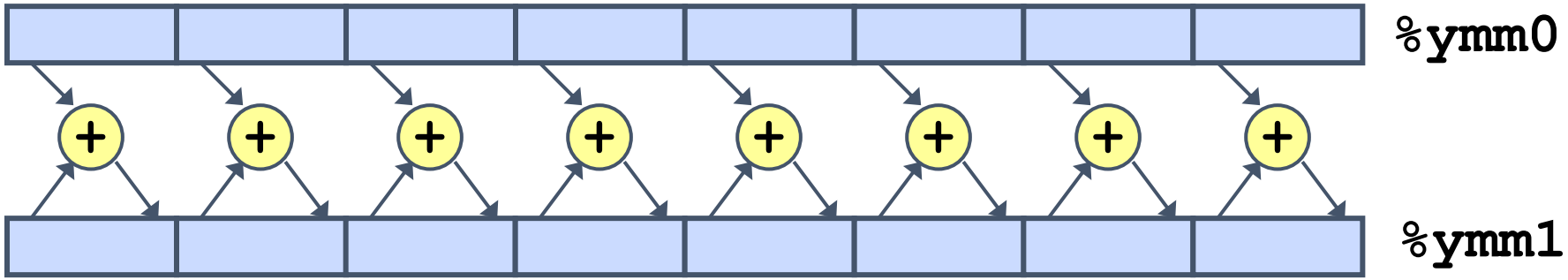
- 1 single-precision float

- 1 double-precision float

单指令多数据

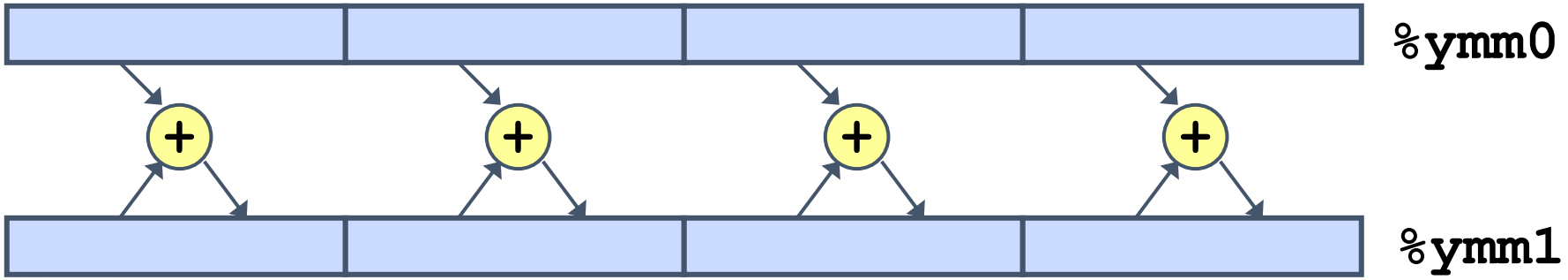
■SIMD Operations: Single Precision

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



■SIMD Operations: Double Precision

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



向量指令

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*

- Make use of AVX Instructions
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

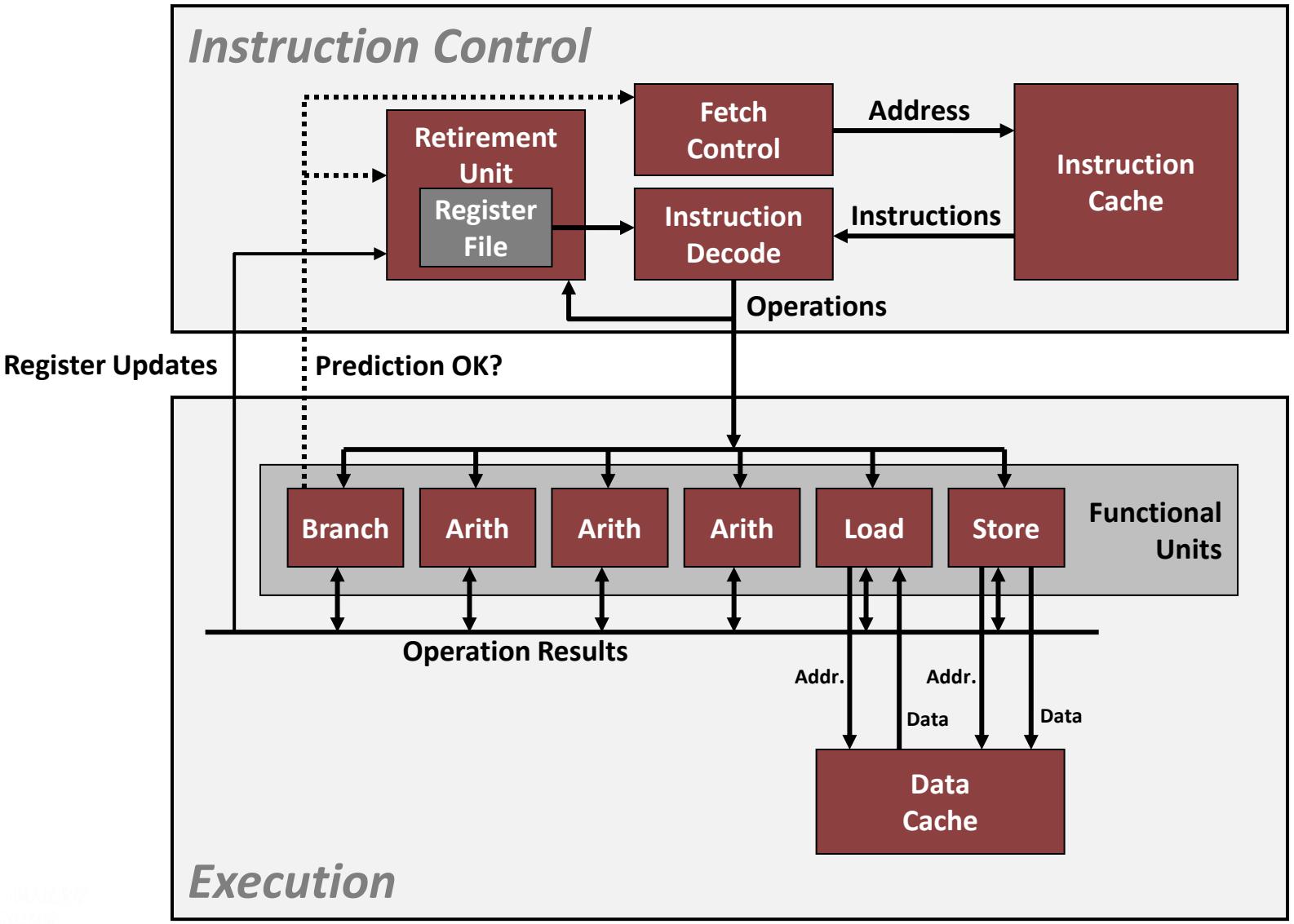
```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **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 Prediction

- Idea
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - 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

Branch Prediction Through Loop

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

i = 98

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
```

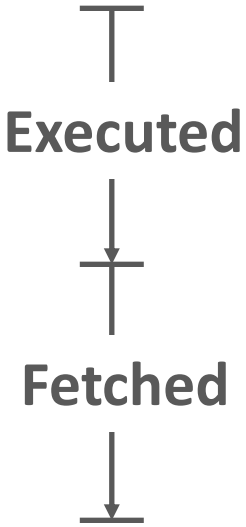
i = 100

Read invalid location

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

i = 101

Assume
vector length = 100



Branch Misprediction Invalidation

```
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

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

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
401036:   jmp    401040
 . . .
401040:   vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- Performance Cost
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

Branch Prediction Numbers

- Default behavior:
  - Backwards branches are often loops so predict taken
  - Forwards branches are often if so predict not taken
- Predictors average better than 95% accuracy
  - Most branches are already predictable.
- Bonus material: http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array

确信和消除性能瓶径

程序剖析

- 程序剖析(profiling)：是程序执行时收集性能数据的分析工具，方法是运行程序的一个版本，其中插入了工具代码，以确定程序的各个部分需要多少时间
- GRPOF：给出函数执行时间和调用次数
  1. 为剖析编译和链接程序：使用-pg选项
     - 例：gcc -O3 -pg prefixsum.c -o prefixsum
  2. 执行程序
     - 例：prefixsum
  3. 调用GPROF分析
     - 例：gprof prefixsum，剖析报告显示函数执行信息

```
granularity: each sample hit covers 2 byte(s) for 0.15% of 6.86 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    1.38    5.48                 main [1]
                2.87    0.00       1/1           psum1 [2]
                1.70    0.00       1/1           psum2 [3]
                0.91    0.00       1/1           psuma [4]
-----------------------------------------------
                2.87    0.00       1/1           main [1]
[2]     41.9    2.87    0.00       1         psum1 [2]
-----------------------------------------------
                1.70    0.00       1/1           main [1]
[3]     24.8    1.70    0.00       1         psum2 [3]
-----------------------------------------------
                0.91    0.00       1/1           main [1]
[4]     13.2    0.91    0.00       1         psuma [4]
-----------------------------------------------
```

perf监测性能指标

- Perf 是用来进行软件性能分析的工具，可以同时分析应用代码和内核，从而全面理解应用程序中的性能瓶颈。
- Perf程序剖析示例：监测rearrange程序branch与cache misses
  - 命令：perf stat -e branches -e branch-misses -e cache-misses ./rearrange

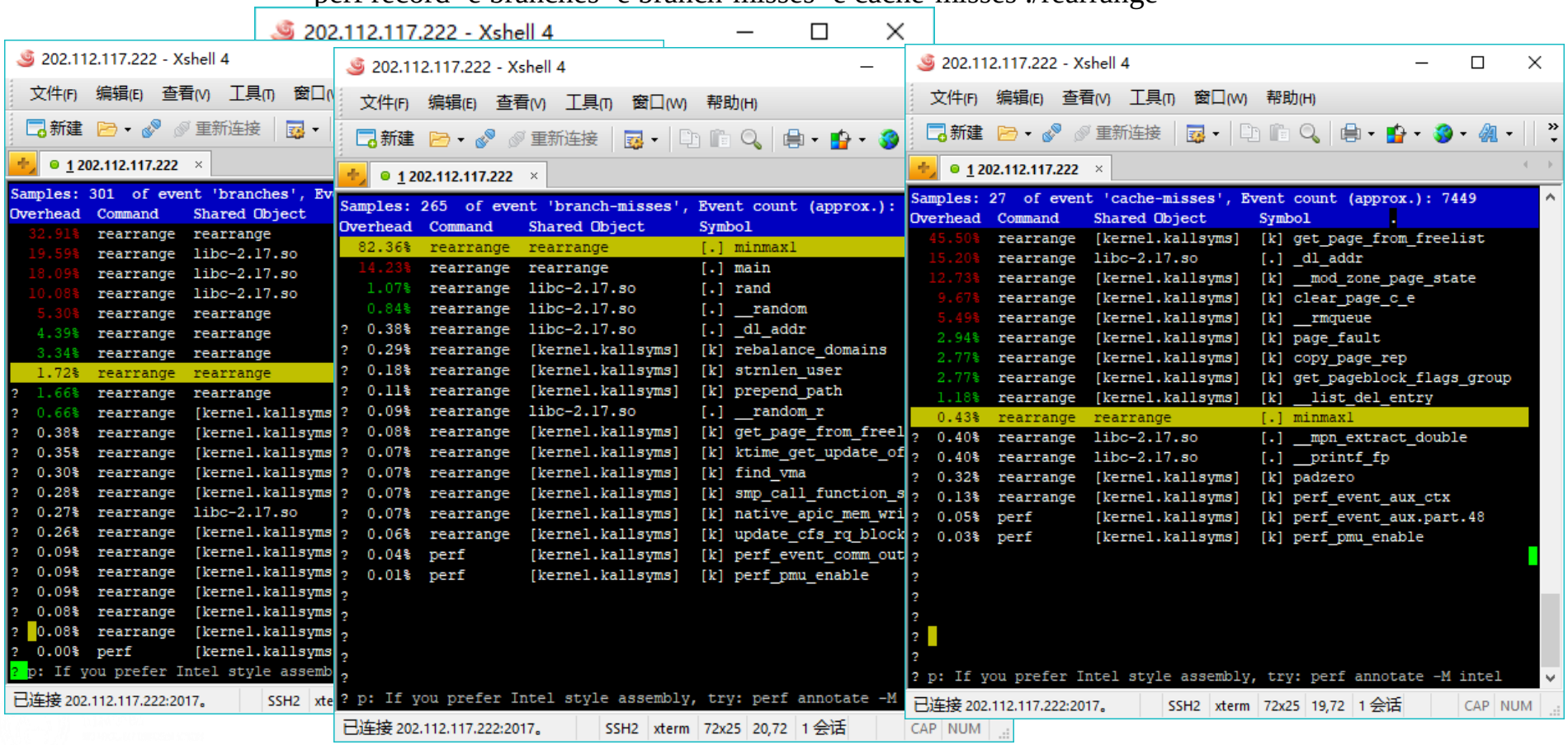
```
[nsrc@localhost ~]$ perf stat -e branches -e branch-misses -e cache-misses ./rearrange
Input length of random array a and b:1000000
Input filter ratio of array:1
numbers of 1:1000000
minmax1 cycles_per_elem:17.6
minmax1 cycles_per_elem: 2.7
bredicate branching: Items:1000000, cycles_per_elem: 5.9
bredicate branching: Items:1000000, cycles_per_elem: 3.9

 Performance counter stats for './rearrange':

       56,337,543      branches
          613,754      branch-misses             #    1.09% of all branches
            3,327      cache-misses

      3.949698873 seconds time elapsed
```

- Perf主要监测指标： perf list

```
[nsrc@localhost ~]$ perf list

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  alignment-faults                                   [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
  node-load-misses                                   [Hardware cache event]
  node-loads                                         [Hardware cache event]
  node-store-misses                                  [Hardware cache event]
  node-stores                                        [Hardware cache event]

  branch-instructions OR cpu/branch-instructions/    [Kernel PMU event]
  branch-misses OR cpu/branch-misses/                [Kernel PMU event]
  bus-cycles OR cpu/bus-cycles/                      [Kernel PMU event]
  cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
  cache-references OR cpu/cache-references/          [Kernel PMU event]
  cpu-cycles OR cpu/cpu-cycles/                      [Kernel PMU event]
  cycles-ct OR cpu/cycles-ct/                        [Kernel PMU event]
  cycles-t OR cpu/cycles-t/                          [Kernel PMU event]
```

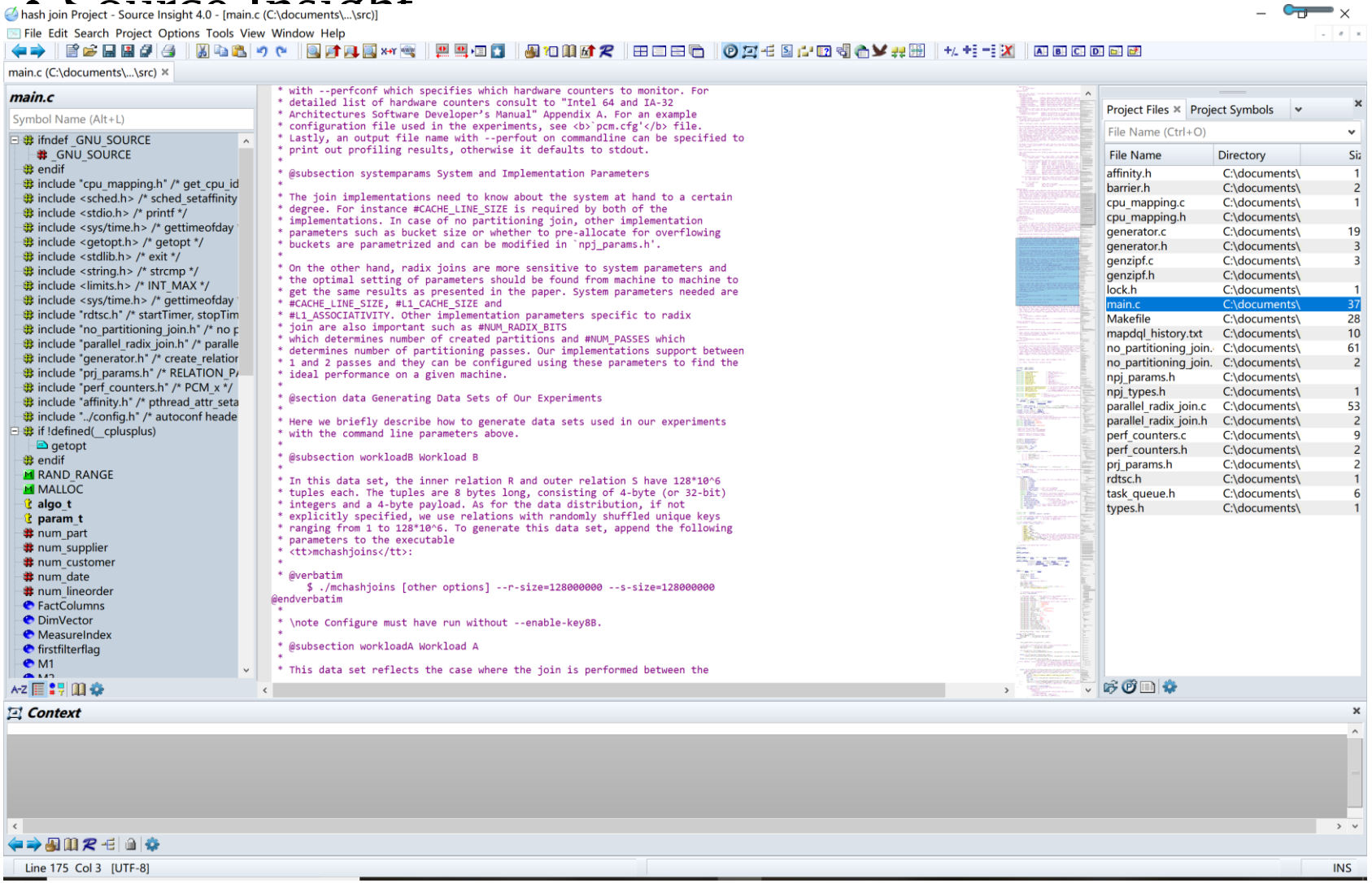
perf监测性能指标

- Perf性能分析：监测程序性能并将其数据保存到perf.data中，使用perf report进行分析。
  - perf record和perf report可以更精确的分析一个应用，perf record可以精确到函数级别。并且在函数里面混合显示汇编语言和代码。
  - 命令：
    - perf record -e branches -e branch-misses -e cache-misses ./rearrange



代码阅读工具

Source Insight



Getting High Performance

- Good compiler and flags
- Don’t do anything stupid
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)

预祝同学们：
考得都会
蒙的都对
新年快乐

相约2026《人工智能计算系统》