优化程序性能选讲

王慧妍
why@nju.edu.cn

南京大学 计算机科学与技术系 计算机软件研究所

概述

- 程序的性能优化
  - 算法的优化
  - 代码的优化
  - 指令的优化

- 一般practice
  - 选择适合的算法和数据结构
    - 算法课
  - 编写能够被编译器有效优化并转化为高效的可执行代码的源码
    - 理解编译器优化的能力和局限

理解编译器优化的能力和局限性

- 理解程序如何编译+执行
- 理解现代处理器和存储系统
- 如何诊断性能瓶颈和提高性能

- 编译器优化可以做什么?
  - 建立程序到机器的有效映射
    - 分配寄存器
    - 代码筛选和调度
    - 死代码消除
    - 简单低效消除
  - Optimization blockers

常见的有效优化：减少计算操作的次数频率

- 不考虑处理器或编译器之外的优化方式

- 减少计算操作的次数频率
  - 保持程序行为总是一致的前提
  - 尤其对循环中的操作

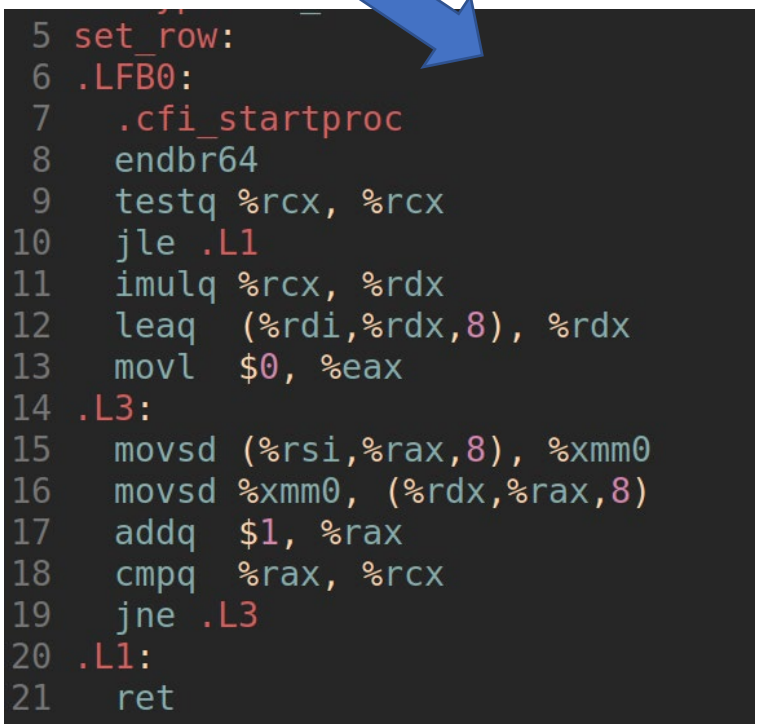
```
void set_row(double *a,
double *b, long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

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

GCC –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];
```



Test n
If 0, goto **done**
 # ni = n * I
rowp = A + ni * 8
j = 0
loop:
t = b[j]
M[A+ni*8+j*8] = t
j ++
if != , goto **loop**
done:

常见的有效优化：强度降低

- 用简单操作代替复杂操作
- Shift，add代替乘除
  - 16*x → x << 4

```
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;
}
```

常见的有效优化：子表达式的复用

```
/*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;
```

```
6  sum_func:
7  .LFB0:
8    .cfi_startproc
9    endbr64
10   subl   $1, %edi
11   movq   val(%rip), %rcx
12   movl   %esi, %eax
13   imull  %edx, %edi
14   addl   %edi, %esi
15   movslq  %esi, %rsi
16   movsd (%rcx,%rsi,8), %xmm0
17   leal   (%rdi,%rdx,2), %esi
18   leal   (%rsi,%rax), %edi
19   subl   %edx, %esi
20   movslq  %edi, %rdi
21   movsd %xmm0, up(%rip)
22   addl   %eax, %esi
23   movsd (%rcx,%rdi,8), %xmm3
24   movslq  %esi, %rsi
25   addsd %xmm3, %xmm0
26   movsd %xmm3, down(%rip)
27   movsd -8(%rcx,%rsi,8), %xmm2
28   movsd %xmm2, left(%rip)
29   movsd 8(%rcx,%rsi,8), %xmm1
30   addsd %xmm2, %xmm0
31   movsd %xmm1, right(%rip)
32   addsd %xmm1, %xmm0
33   movsd %xmm0, sum(%rip)
34   ret
```

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

```
5  sum_func:
6  .LFB0:
7    .cfi_startproc
8    endbr64
9    imull %edx, %edi
10   addl   %esi, %edi
11   movslq  %edi, %rsi
12   movq   val(%rip), %rax
13   movslq  %edx, %rdx
14   movq   %rsi, %rcx
15   subq   %rdx, %rcx
16   movsd (%rax,%rcx,8), %xmm1
17   movsd %xmm1, up(%rip)
18   addq   %rsi, %rdx
19   movsd (%rax,%rdx,8), %xmm3
20   movsd %xmm3, down(%rip)
21   movsd -8(%rax,%rsi,8), %xmm2
22   movsd %xmm2, left(%rip)
23   movsd 8(%rax,%rsi,8), %xmm0
24   movsd %xmm0, right(%rip)
25   addsd %xmm3, %xmm1
26   addsd %xmm2, %xmm1
27   addsd %xmm1, %xmm0
28   movsd %xmm0, sum(%rip)
29   ret
```

我们的目标

- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 编写适合编译优化的源码

  - GCC online documentation - GNU Project
    - -O0
    - -O1
    - -O2
    - -O3
    - -Os
    - -Ofast
    - -Og

GCC的优化选项

管理 控制 视图 热键 设备 帮助

```
$ gcc -Q -O1 --help=optimizers
```

GCC online documentation - GNU Project

`-O1` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With '`-O`', the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

'`-O`' turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

GCC online documentation - GNU Project

`-O2` Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to '`-O`', this option increases both compilation time and the performance of the generated code.

'`-O2`' turns on all optimization flags specified by '`-O`'. It also turns on the following optimization flags:

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-ffinite-loops
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
```

```
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-pre
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
```

理解编译器优化的能力和局限性

- 理解程序如何编译+执行
- 理解现代处理器和存储系统
- 如何诊断性能瓶颈和提高性能

- 编译器的一些局限，理解编译器的优化痛点
  - 编译器优化前提是*safe*
    - Within procedure analyses
    - Static information analyses
    - “Conservative” to be “safe”

理解编译器优化的痛点

- Two optimization blocker
  - Memory aliasing
  - Function calls

示例程序1.1

```
void sum_row1(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];
    }
}
```

```
21 .L3:
22    movsd (%rdx), %xmm0
23    addsd (%rax), %xmm0
24    movsd %xmm0, (%rdx)
25    addq  $8, %rax
26    cmpq  %rcx, %rax
27    jne .L3
```

B数组

```
double A[9] =
{   0,  1,  2,
    4,  8,  16,
    32, 64, 128};

double *B = A+3;

sum_row1(A, B, 3)
```

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

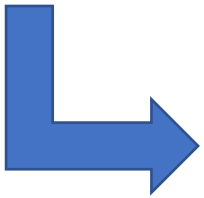
```
 i=0: [3,   8,   16]
```

```
 i=1: [3,   22, 16]
```

```
 i=2: [3,   22, 224]
```

示例程序1.1

```
void sum_row1(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];
    }
}
```



```
void sum_row1(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;
    }
}
```

示例程序1.2

```c
//a.c
#include <stdio.h>
void f1 (int *xp, int *yp){
        *xp += *yp;
        *xp += *yp;
}
```

gcc -O1 a.c

```
$ objdump -d a.o

a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <f1>:
   0:   f3 0f 1e fa             endbr64
   4:   8b 06                   mov    (%rsi),%eax
   6:   03 07                   add    (%rdi),%eax
   8:   89 07                   mov    %eax,(%rdi)
   a:   03 06                   add    (%rsi),%eax
   c:   89 07                   mov    %eax,(%rdi)
   e:   c3                      ret
```

```c
//b.c
#include <stdio.h>
void f1 (int *xp, int *yp){
        *xp += 2* *yp;
}
```

gcc -O1 b.c

```
$ objdump -d b.o

b.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <f1>:
   0:   f3 0f 1e fa             endbr64
   4:   8b 06                   mov    (%rsi),%eax
   6:   01 c0                   add    %eax,%eax
   8:   01 07                   add    %eax,(%rdi)
   a:   c3                      ret
```

如果xp和yp指向同一块内存地址?

示例程序1.2

```c
//a.c
#include <stdio.h>
void f1 (int *xp, int *yp){
        *xp += *yp;
        *xp += *yp;
}
```

```c
//b.c
#include <stdio.h>
void f1 (int *xp, int *yp){
        *xp += 2* *yp;
}
```

如果xp和yp指向同一块内存地址?

```c
        *xp += *xp;
        *xp += *xp;
```

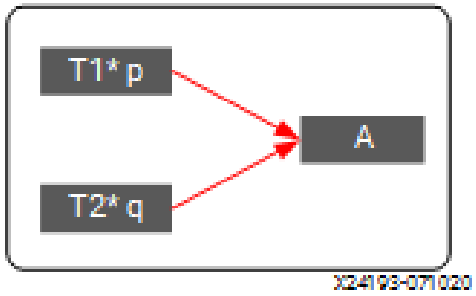
```c
        *xp += 2* *xp;
```

```c
        *xp = 4 * *xp
```

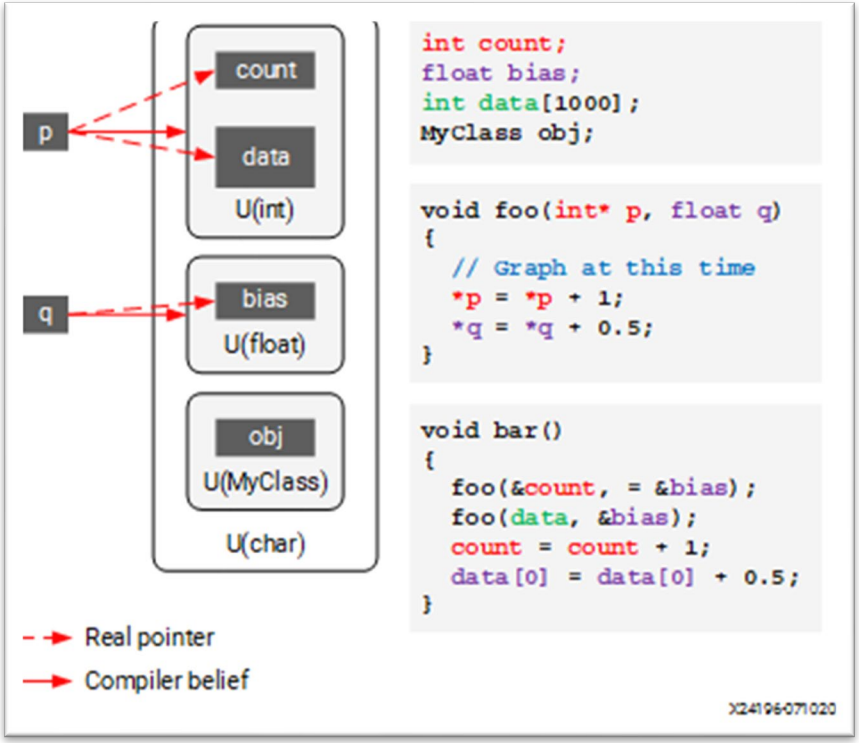
```c
        *xp = 3 * *xp
```

Memory aliasing

Memory aliasing & strict aliasing rules




```
//a.c
#include <stdio.h>
void f1 (int *xp, int *yp){
        *xp += *yp;
        *xp += *yp;
}
```




```
int count;
float bias;
int data[1000];
MyClass obj;
```

```
void foo(int* p, float q)
{
  // Graph at this time
  *p = *p + 1;
  *q = *q + 0.5;
}
```

```
void bar()
{
  foo(&count, = &bias);
  foo(data, &bias);
  count = count + 1;
  data[0] = data[0] + 0.5;
}
```

```c
// C program to illustrate aliasing
#include <stdio.h>

// Function to change the value of
// ptr1 and ptr2
int foo(int* ptr1, int* ptr2)
{
    *ptr1 = 10;
    *ptr2 = 11;
    return *ptr1;
}

// Driver Code
int main()
{
    int data1 = 10, data2 = 20;

    // Function Call
    int result = foo(&data1, &data2);

    // Print result
    printf("%d ", result);
    return 0;
}
```

```
Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   c7 07 0a 00 00 00       movl   $0xa,(%rdi)
   a:   c7 06 0b 00 00 00       movl   $0xb,(%rsi)
  10:   8b 07                   mov    (%rdi),%eax
  12:   c3                      ret
```

```
gcc -O1 a.c
```

```
gcc -O2 a.c
```

Memory aliasing引发不同的优化结果

- What to produce by GCC-O1, GCC-O2 optimizers?

```
 3 #include <stdio.h>
 4
 5 // Function to change the value of
 6 // ptr1 and ptr2
 7 int foo(int* ptr1, long* ptr2)
 8 {
 9     *ptr1 = 10;
10     *ptr2 = 11.0;
11     return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
24     return 0;
25 }
```

管理 控制 视图 热键 设备 帮助

```
 1
 2 // C program to illustrate aliasing
 3 #include <stdio.h>
 4
 5 // Function to change the value of
 6 // ptr1 and ptr2
 7 int foo(int* ptr1, long* ptr2)
 8 {
 9     *ptr1 = 10;
10     *ptr2 = 11.0;
11     return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
<ts/ICS2021/teach/Course17/demo.c[1] [c] unix utf-8 Ln 1, Col 0/25
```

```c
#include <stdio.h>

// Function to change the value of
// ptr1 and ptr2
int foo(int* ptr1, long* ptr2)
{
    *ptr1 = 10;
    *ptr2 = 11.0;
    return *ptr1;
}

// Driver Code
int main()
{
    long data = 100.0;

    // Function Call
    int result = foo((int *)&data, &data);

    // Print result
    printf("%d \n", result);
    return 0;
}
```

```
0000000000001149 <foo>:
    1149:       f3 0f 1e fa             endbr64
    114d:       c7 07 0a 00 00 00       movl   $0xa,(%rdi)
    1153:       48 c7 06 0b 00 00 00    movq   $0xb,(%rsi)
    115a:       8b 07                   mov    (%rdi),%eax
    115c:       c3                      ret
```

gcc -O1 a.c

```
0000000000001180 <foo>:
    1180:       f3 0f 1e fa             endbr64
    1184:       c7 07 0a 00 00 00       movl   $0xa,(%rdi)
    118a:       b8 0a 00 00 00          mov    $0xa,%eax
    118f:       48 c7 06 0b 00 00 00    movq   $0xb,(%rsi)
    1196:       c3                      ret
```

gcc -O2 a.c

```
  -fstack-reuse=[all|named_vars|none]   all
  -fstdarg-opt                          [enabled]
  -fstore-merging                       [enabled]
  -fstrict-aliasing                     [enabled]
  -fstrict-enums                        [available in C++, ObjC++]
  -fstrict-volatile-bitfields           [enabled]
  -fthread-jumps                        [enabled]
```

管理 控制 视图 热键 设备 帮助

```
    1200:       41 5e                   pop    %r14
    1202:       41 5f                   pop    %r15
    1204:       c3                      ret
    1205:       66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%ra
x,1)
    120c:       00 00 00 00

0000000000001210 <__libc_csu_fini>:
    1210:       f3 0f 1e fa             endbr64
    1214:       c3                      ret

Disassembly of section .fini:

0000000000001218 <_fini>:
    1218:       f3 0f 1e fa             endbr64
    121c:       48 83 ec 08             sub    $0x8,%rsp
    1220:       48 83 c4 08             add    $0x8,%rsp
    1224:       c3                      ret
$ gcc -O1 demo.c
$ ./a.out
11
$ gcc -O2 demo.c
$ ./a.out
10
$ gcc -O2 demo.c
```

Right Shift + Right Alt

```c
#include <stdio.h>

// Function to change the value of
// ptr1 and ptr2
int foo(int* ptr1, long* ptr2)
{
    *ptr1 = 10;
    *ptr2 = 11.0;
    return *ptr1;
}

// Driver Code
int main()
{
    long data = 100.0;

    // Function Call
    int result = foo((int *)&data, &data);

    // Print result
    printf("%d \n", result);
    return 0;
}
```

gcc -O2 a.c -fno-strict-aliasing

```
0000000000001149 <foo>:
    1149:       f3 0f 1e fa             endbr64
    114d:       c7 07 0a 00 00 00       movl   $0xa,(%rdi)
    1153:       48 c7 06 0b 00 00 00    movq   $0xb,(%rsi)
    115a:       8b 07                   mov    (%rdi),%eax
    115c:       c3                      ret
```

gcc -O1 a.c

```
0000000000001180 <foo>:
    1180:       f3 0f 1e fa             endbr64
    1184:       c7 07 0a 00 00 00       movl   $0xa,(%rdi)
    118a:       b8 0a 00 00 00          mov    $0xa,%eax
    118f:       48 c7 06 0b 00 00 00    movq   $0xb,(%rsi)
    1196:       c3                      ret
```

gcc -O2 a.c

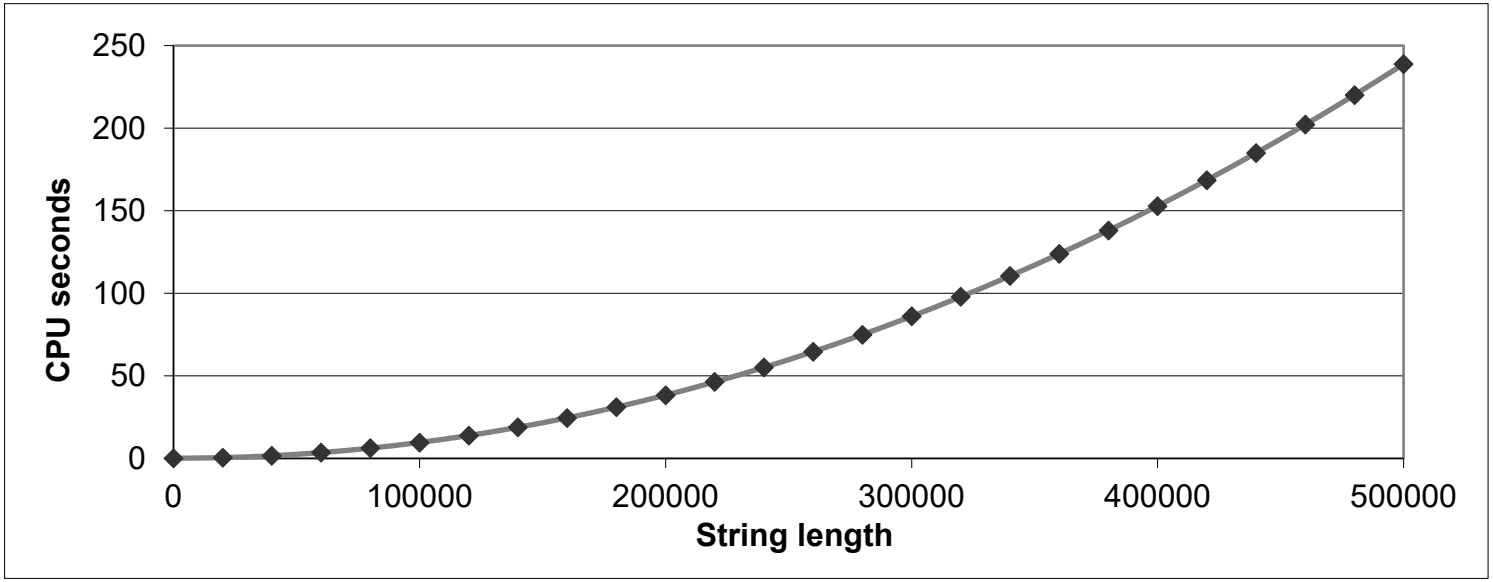
```
  -fstack-reuse=[all|named_vars|none]   all
  -fstdarg-opt                          [enabled]
  -fstore-merging                       [enabled]
  -fstrict-aliasing                     [enabled]
  -fstrict-enums                        [available in C++, ObjC++]
  -fstrict-volatile-bitfields           [enabled]
  -fthread-jumps                        [enabled]
```

Optimization blocker：memory aliasing

- Aliasing
  - 两个不同的内存reference可能指向同一块内存
  - C语言中常见
    - C允许地址操作
  - 合适地引入局部变量
    - 函数内部
    - 开发时消除aliasing

示例程序2.1

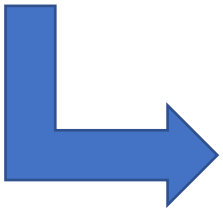
```
void lower(char *s){
    size_t i;
    for (i = 0; i<strlen(s); i++)
        if(s[i] >= ‘A’ && s[i] <= ‘Z’)
            s[i] -= (‘A’ – ‘a’);
}
```



示例程序2.1

```
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’);
}
```

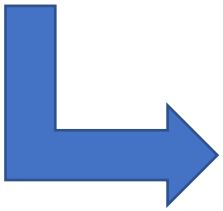
- strlen()：找结束符
  - 线性复杂度
  - N次进入循环，二次复杂度



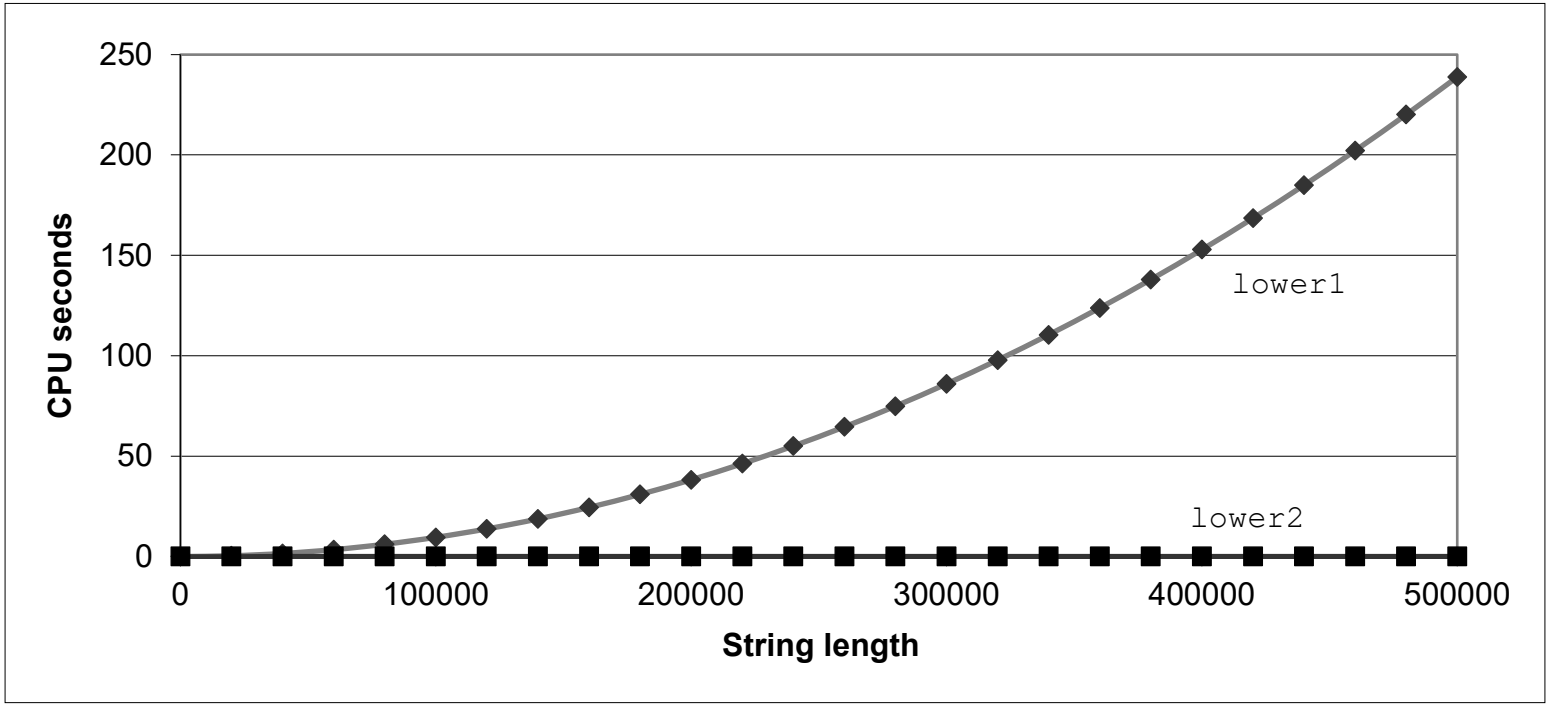
```
void lower(char *s){
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
        if(s[i] >= ‘A’ && s[i] <= ‘Z’)
            s[i] -= (‘A’ – ‘a’);
        i ++;
        if(i< strlen(s))
            goto loop;
done:
}
```

示例程序2.1

```
void lower1(char *s){
    size_t i;
    for (i = 0; i<strlen(s); i++)
        if(s[i] >= ‘A’ && s[i] <= ‘Z’)
            s[i] -= (‘A’ – ‘a’);
}
```



```
void lower2(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’);
}
```


250
200
150
100
50
0
CPU seconds
lower1
lower2
0
100000
200000
300000
400000
500000
String length

思考

- 为什么编译器不能将*strlen*自动优化出循环?
  - 过程调用的副作用
    - 全局变量
  - 每次调用返回值不一定一样

- 编译器一般会将其作为黑盒处理
  - Linking决定具体函数调用的实现

- 其他措施
  - Inline functions
  - 更好的coding

```
void lower1(char *s){
    size_t i;
    for (i = 0; i<strlen(s); i++)
        if(s[i] >= ‘A’ && s[i] <= ‘Z’)
            s[i] -= (‘A’ – ‘a’);
}
```

示例程序2.2

```
long f();

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

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

```
func1:
  endbr64
  pushq %rbx
  xorl  %eax, %eax
  call  f@PLT
  movq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rbx, %rax
  popq  %rbx
  ret
```

```
func2:
  endbr64
  subq  $8, %rsp
  xorl  %eax, %eax
  call  f@PLT
  addq  $8, %rsp
  salq  $2, %rax
  ret
```

为了safe地优化，编译器考虑函数调用可能存在副作用（side effect）

示例程序2.2

```
long f();

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

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

counter = 6

counter = 0

```
func1:
  endbr64
  pushq %rbx
  xorl  %eax, %eax
  call  f@PLT
  movq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rax, %rbx
  xorl  %eax, %eax
  call  f@PLT
  addq  %rbx, %rax
  popq  %rbx
  ret
```

```
func2:
  endbr64
  subq  $8, %rsp
  xorl  %eax, %eax
  call  f@PLT
  addq  $8, %rsp
  salq  $2, %rax
  ret
```

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

Inlir

`-findirect-inlining`

Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining. This option has any effect only when inlining itself is turned on by the '`-finline-functions`' or '`-finline-small-functions`' options.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

```
long

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

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

```
long func1in(){
    long t = counter ++;
    t += counter ++;
    t += counter ++;
    t += counter ++;
    return t;
}
```

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

管理 控制 视图 热键 设备 帮助

$

Inline substitution

```
long f();

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

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

gcc -O0 a.c

gcc -O1 a.c

```
0000000000000000 <func1>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   53                      push   %rbx
   9:   48 83 ec 08             sub    $0x8,%rsp
   d:   b8 00 00 00 00          mov    $0x0,%eax
  12:   e8 00 00 00 00          call   17 <func1+0x17>
  17:   48 89 c3                mov    %rax,%rbx
  1a:   b8 00 00 00 00          mov    $0x0,%eax
  1f:   e8 00 00 00 00          call   24 <func1+0x24>
  24:   48 01 c3                add    %rax,%rbx
  27:   b8 00 00 00 00          mov    $0x0,%eax
  2c:   e8 00 00 00 00          call   31 <func1+0x31>
  31:   48 01 c3                add    %rax,%rbx
  34:   b8 00 00 00 00          mov    $0x0,%eax
  39:   e8 00 00 00 00          call   3e <func1+0x3e>
  3e:   48 01 d8                add    %rbx,%rax
  41:   48 8b 5d f8             mov    -0x8(%rbp),%rbx
  45:   c9                      leave
  46:   c3                      ret
```

gcc -O2 a.c

```
0000000000000000 <func1>:
   0:   f3 0f 1e fa             endbr64
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <func1+0xb>
   b:   48 8d 50 04             lea    0x4(%rax),%rdx
   f:   48 8d 04 85 06 00 00    lea    0x6(,%rax,4),%rax
  16:   00
  17:   48 89 15 00 00 00 00    mov    %rdx,0x0(%rip)        # 1e <func1+0x1e>
  1e:   c3                      ret
  1f:   90                      nop
```

管理 控制 视图 热键 设备 帮助

```
$ gcc -O2 -c demo2.c
```

Inline substitution

```
long f();

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

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

gcc -O0 a.c

gcc -O1 a.c

gcc -O2 a.c -fno-inline

```
0000000000000000 <func1>:
   0:
   4:
   5:   48 89 e5                mov    %rsp,%rbp
   8:   53                      push   %rbx
   9:   48 83 ec 08             sub    $0x8,%rsp
   d:   b8 00 00 00 00          mov    $0x0,%eax
  12:   e8 00 00 00 00          call   17 <func1+0x17>
  17:   48 89 c3                mov    %rax,%rbx
  1a:   b8 00 00 00 00          mov    $0x0,%eax
  1f:   e8 00 00 00 00          call   24 <func1+0x24>
  24:   48 01 c3                add    %rax,%rbx
  27:   b8 00 00 00 00          mov    $0x0,%eax
  2c:   e8 00 00 00 00          call   31 <func1+0x31>
  31:   48 01 c3                add    %rax,%rbx
  34:   b8 00 00 00 00          mov    $0x0,%eax
  39:   e8 00 00 00 00          call   3e <func1+0x3e>
  3e:   48 01 d8                add    %rbx,%rax
  41:   48 8b 5d f8             mov    -0x8(%rbp),%rbx
  45:   c9                      leave
  46:   c3                      ret
```

gcc -O2 a.c

```
0000000000000000 <func1>:
   0:   f3 0f 1e fa             endbr64
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <func1+0xb>
   b:   48 8d 50 04             lea    0x4(%rax),%rdx
   f:   48 8d 04 85 06 00 00    lea    0x6(,%rax,4),%rax
  16:   00
  17:   48 89 15 00 00 00 00    mov    %rdx,0x0(%rip)        # 1e <func1+0x1e>
  1e:   c3                      ret
  1f:   90                      nop
```

管理 控制 视图 热键 设备 帮助

```c
#include <stdio.h>
long counter = 0;
static inline long f(){ return counter ++;
}

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

long func2(){
  return 4 * f();
}
int main(){
    printf("%ld \n", func1());

}
```

~/Documents/ICS2021/teach/Course17/demo2.c[1] [c] unix utf-8 Ln 1, Col 1/16

管理 控制 视图 热键 设备 帮助

```c
#include <stdio.h>
long counter = 0;
static inline long f(){ return counter ++;
}

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

long func2(){
  return 4 * f();
}
int main(){
    printf("%ld \n", func1());

}
```

~/Documents/ICS2021/teach/Course17/demo2.c[1] [c] unix utf-8 Ln 1, Col 1/16
1 change; before #1 3 seconds ago

Right Shift + Right Alt

Optimization blocker：function call

- Function call
  - 过程调用的副作用
    - 全局变量
  - 每次调用返回值不一定一样

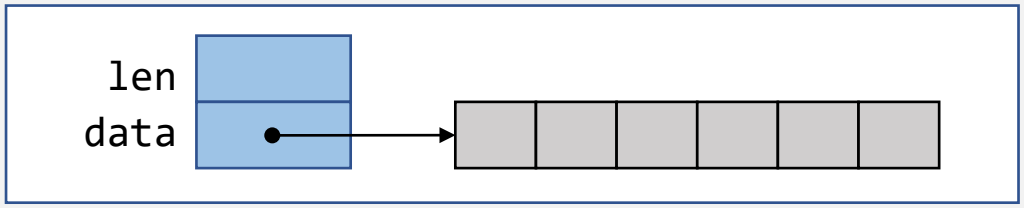
- 编译器一般会将其作为黑盒处理
  - Linking决定具体函数调用的实现

- 其他措施
  - Inline functions
  - 更好的coding

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

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

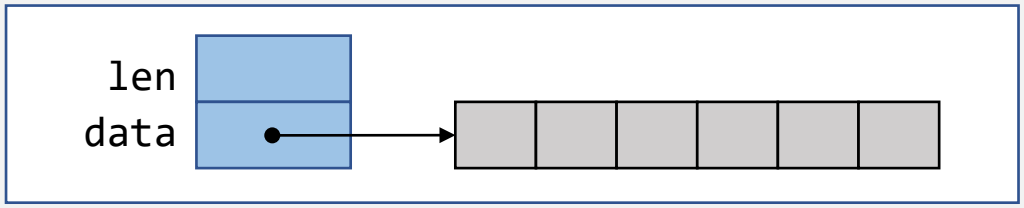
1



- 消除循环低效

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

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



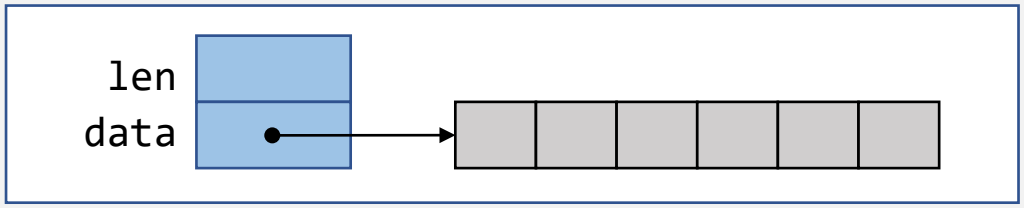
1

- 消除循环低效

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

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

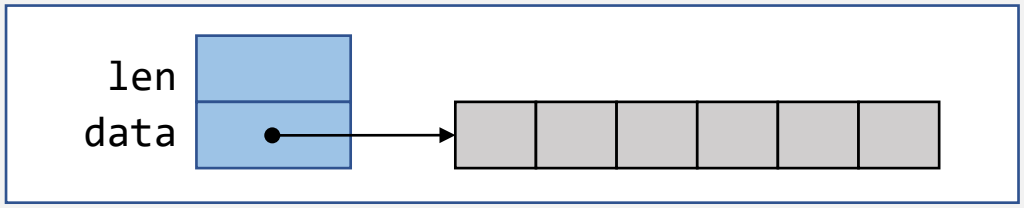
2



- 消除循环低效
- 减少函数调用

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

void combine1 (vec_ptr v, data_t *dest){
   long i;
   long length = vec_length(v);
   data_t *data = get_vec_start(v);
   *dest = 0;
   for (i =0; i < length; i++){
      *dest = *dest + data[i];
   }
}
```



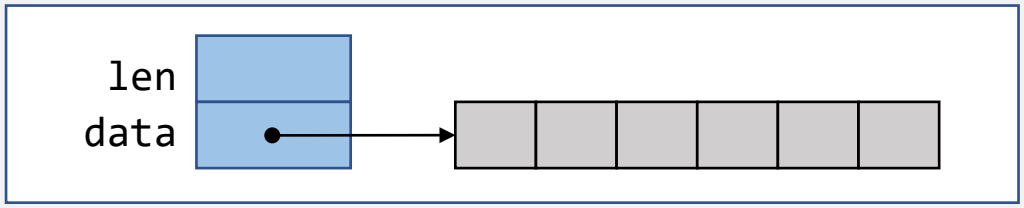
2

- 消除循环低效
- 减少函数调用

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

void combine1 (vec_ptr v, data_t *dest){
   long i;
   long length = vec_length(v);
   data_t *data = get_vec_start(v);
   *dest = 0;
   for (i =0; i < length; i++){
      *dest = *dest + data[i];
   }
}
```

3



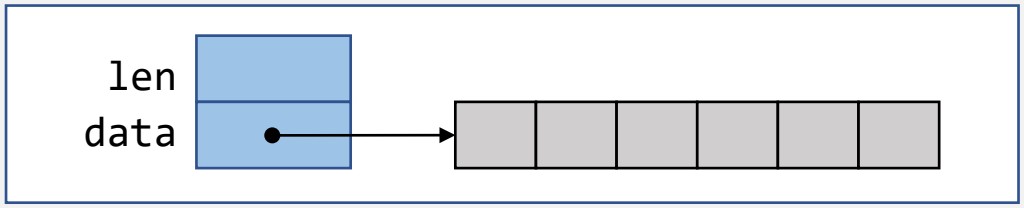

```
.L3:
  movq   (%rax), %rdx
  addq   %rdx, (%rbx)
  addq   $8, %rax
  cmpq   %rcx, %rax
  jne .L3
```

- 消除循环低效
- 减少函数调用
- 减少无需的内存访问

示例程序3

```
typedef struct{
   long len;
   data_t *data;
}  vec_rec, *vec_ptr;

typedef long data_t;

void combine1 (vec_ptr v, data_t *dest){
   long i;
   long length = vec_length(v);
   data_t *data = get_vec_start(v);
   data_t acc = 0;
   for (i =0; i < length; i++){
      acc = acc + data[i];
   }
   *dest = acc;
}
```

3




```
.L3:
  addq   (%rax), %rdx
  addq   $8, %rax
  cmpq   %rcx, %rax
  jne .L3
```

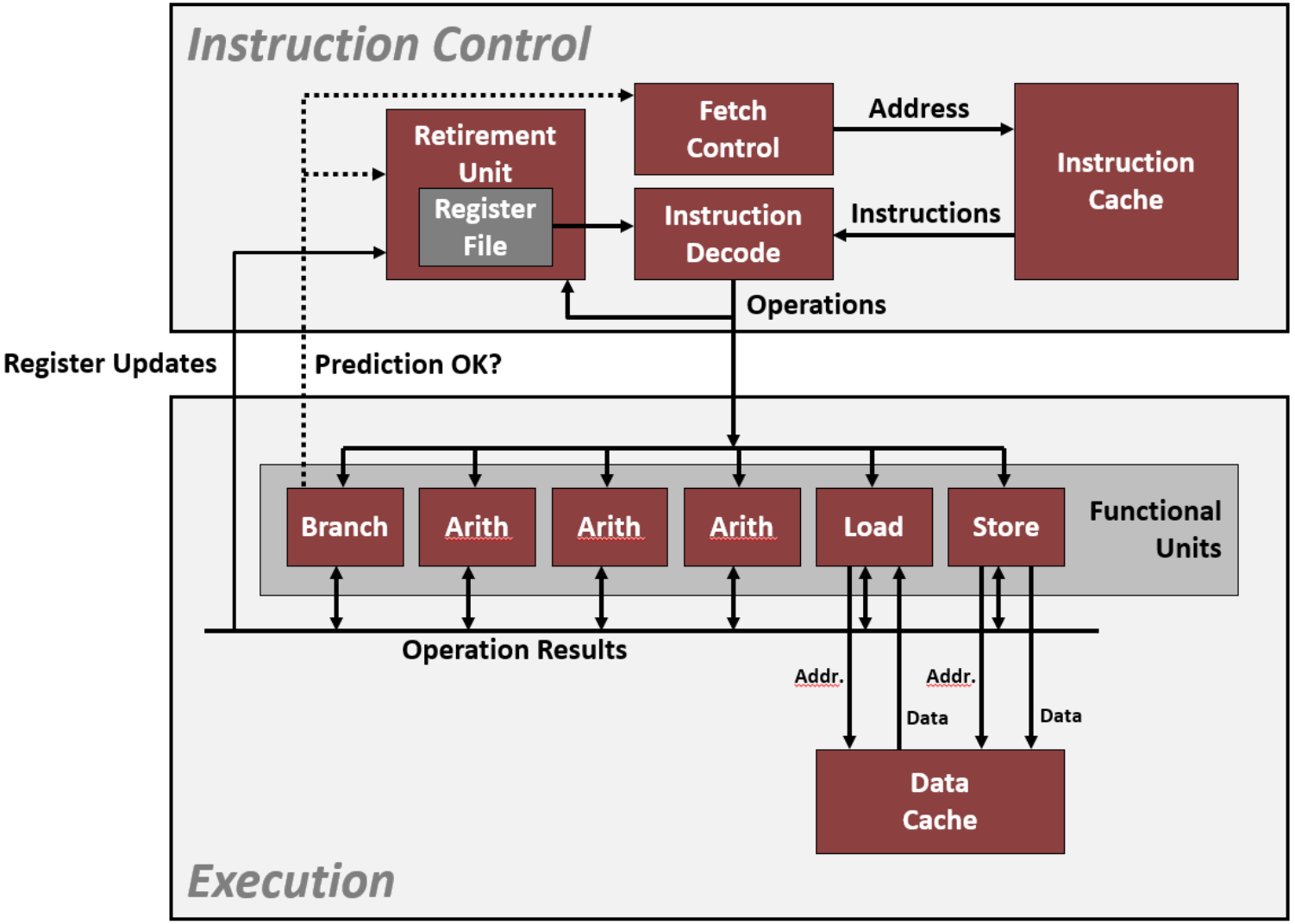
- 消除循环低效
- 减少函数调用
- 减少无需的内存访问

我们能做的事情

- 简单的优化treatment
  - 养成习惯是重要的

- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 理解编译器的优化痛点
    - 编译器优化前提是*safe*
  - 编写适合编译优化的源码
    - Memory aliasing
    - Function call

现代处理器

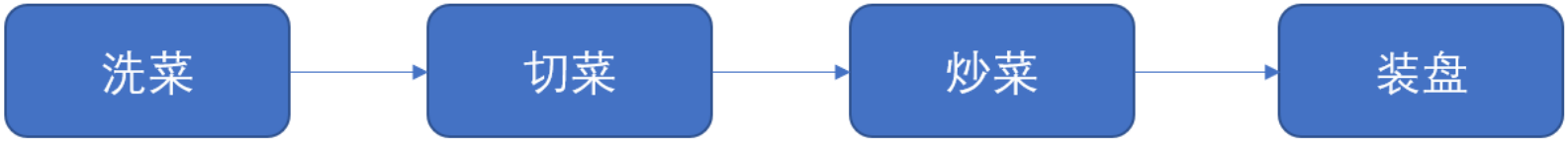


From CMU CSAPP Course

Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

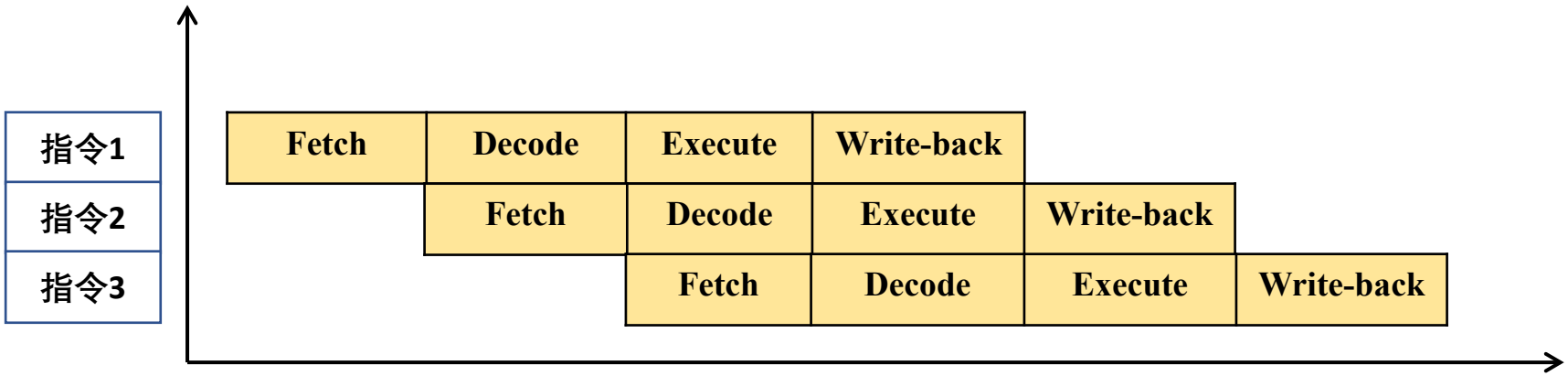
```
long mult_eq(long a, long b, long c){
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

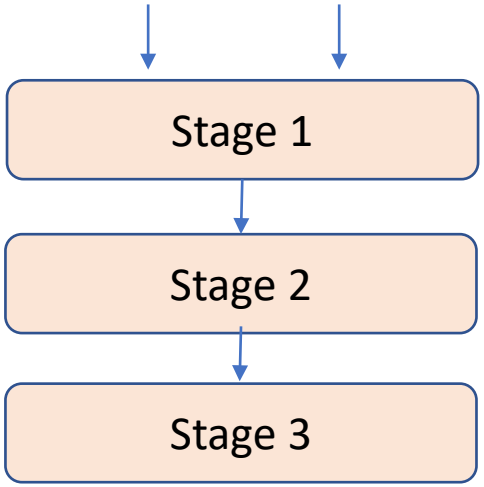
```
long mult_eq(long a, long b, long c){
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

```
long mult_eq(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

超标量处理器

Superscalar Issue (Pentium)

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Execute			Write			
$Instr_2$	Fetch	Decode	Wait			Execute	Write		
$Instr_3$		Fetch	Decode	Execute	Write				
$Instr_4$		Fetch	Decode	Wait			Execute	Write	
$Instr_5$			Fetch	Decode	Execute	Write			
$Instr_6$			Fetch	Decode	Execute	Write			
$Instr_7$				Fetch	Decode	Execute	Write		
$Instr_8$				Fetch	Decode	Execute	Write		

- Superscalar issue allows multiple instructions to be issued at the same time

分支预测

- 预测分支的跳转

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

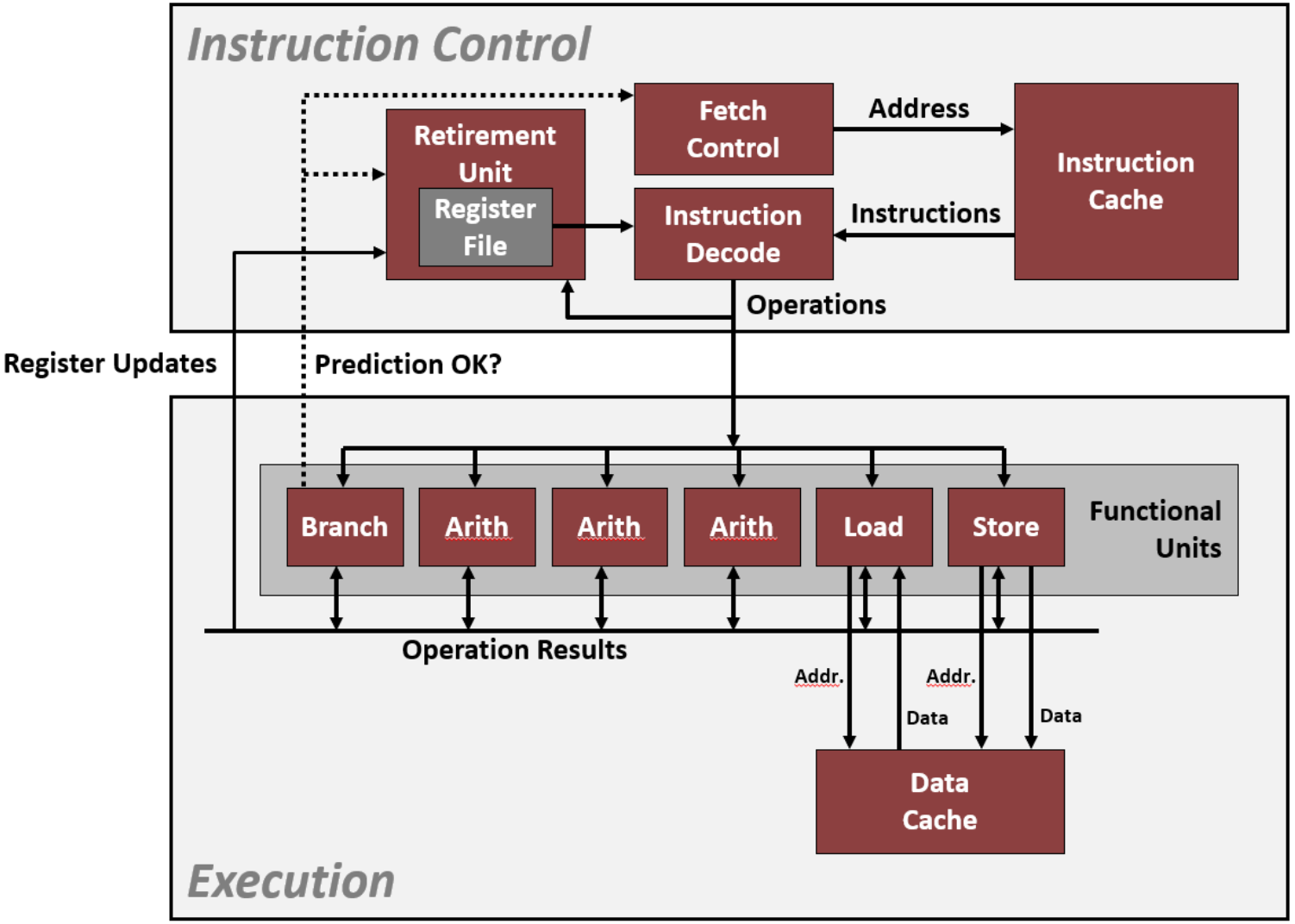
Executing

How to continue?

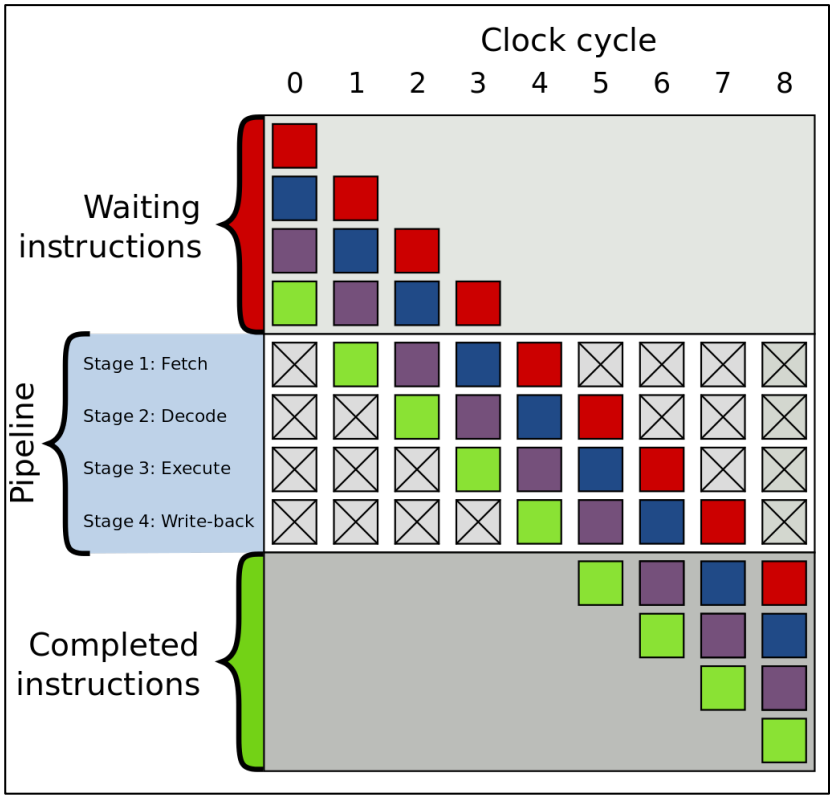
  - 本质：猜测并预测分支的走向

  - java - Why is processing a sorted array faster than processing an unsorted array? - Stack Overflow

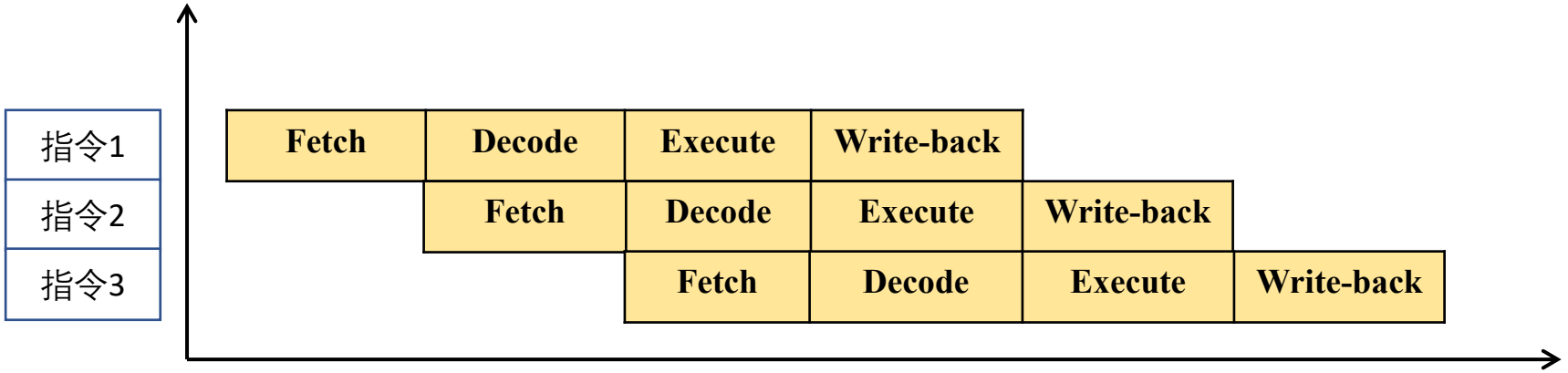
现代处理器



From CMU CSAPP Course




```
int data[N];
int sum = 0;
for (int i = 0; i<N; i++){
    if(data[i] < 128)
        sum += data[i];
}
```



```
int data[N];
int sum = 0;
for (int i = 0; i<N; i++){
    if(data[i] < 128)
        sum += data[i];
}
```

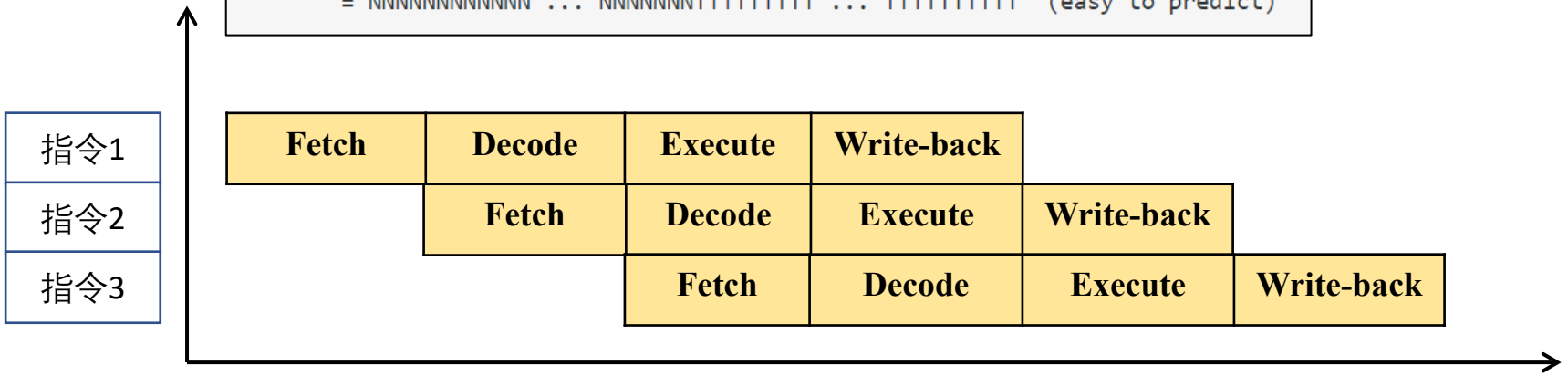
```
int t = (data[i]-128)>>31;
sum+=~t & data[c];
```

Sorted:

```
data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118,  14, 150, 177, 182, ..
branch =   T,   T,   N,   T,   T,   T,   T,  N,   T,   N,   N,   T,   T,   T  ..

       = TTNTTTTNTNNTTT ...   (completely random - impossible to predict)
```

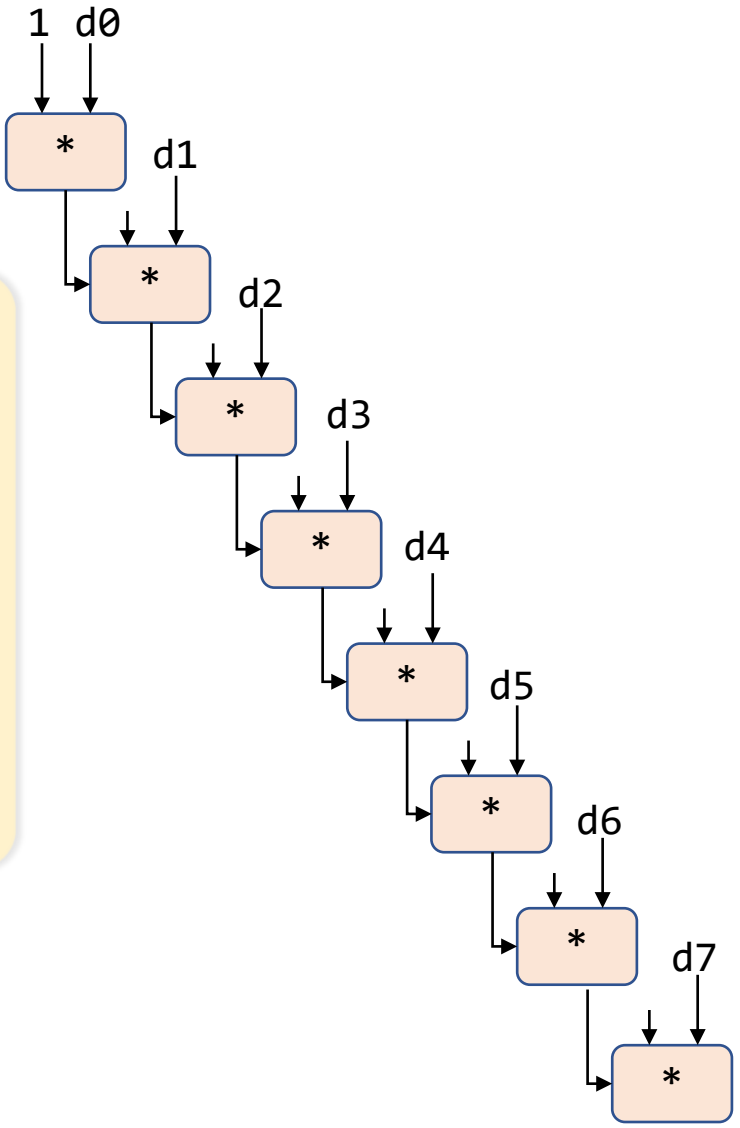
Unsorted:

```
T = branch taken
N = branch not taken

data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, 251, 252, ...
branch = N  N  N  N  N  ...   N    N    T    T    T  ...   T    T    T  ...

       = NNNNNNNNNNNN ... NNNNNNNTTTTTTTTT ... TTTTTTTTT  (easy to predict)
```



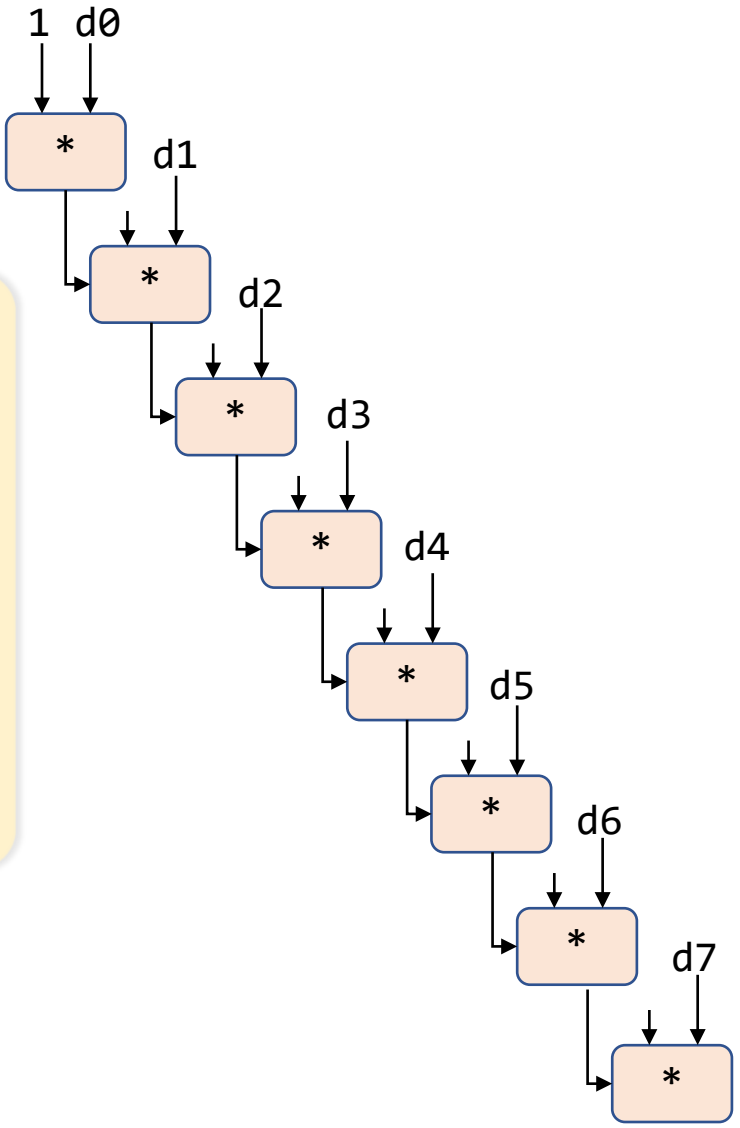
考虑指令并行的优化

```
void combine1 (vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT; //0 for +, 1 for *
    for (i =0; i < length; i++){
        acc = acc OP data[i];
    }
    *dest = acc;
}
```



循环展开1

```
void combine1 (vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT; //0 for +, 1 for *
    for (i =0; i < length; i+=2){
        acc = (acc OP data[i]) OP data[i+1];
    }
    *dest = acc;
}
```

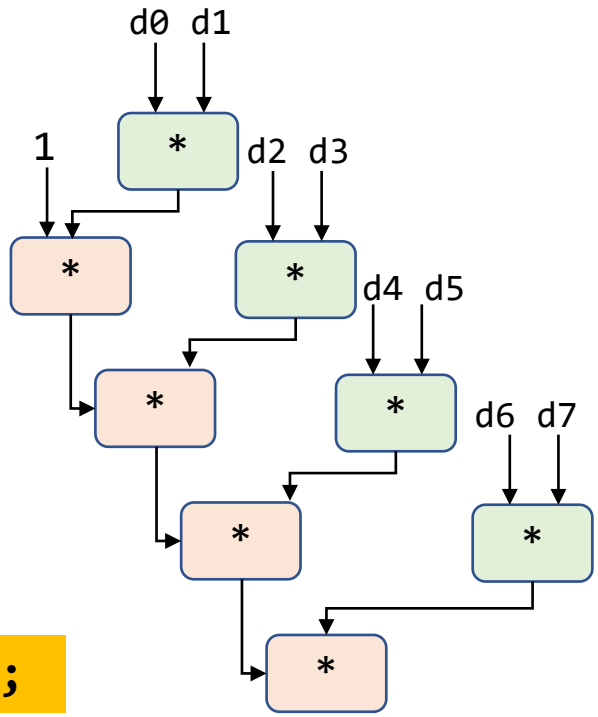


循环展开2

- 打破指令内在的顺序依赖关系

```
void combine1 (vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT; //0 for +, 1 for *
    for (i =0; i < length; i+=2){
        acc = (acc OP data[i]) OP data[i+1];
    }
    *dest = acc;
}
```

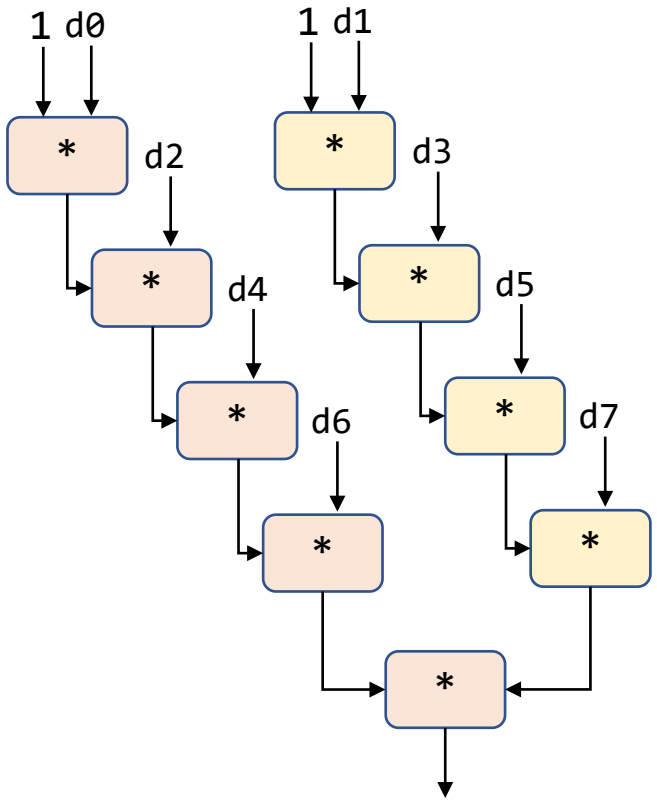
```
acc = acc OP (data[i] OP data[i+1]);
```



循环展开3-multiple accumulator

- 打破指令内在的顺序依赖关系

```
void combine1 (vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    long limit = length – 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT; //0 for +, 1 for *
    //combine 2 elements at a time
    for (i =0; i < limit; i+=2){
        x0 = x0 OP data[i];
        x1 = x1 OP data[i+1];
    }
    //finish any remaining element
    for(; i < length; i++)
        x0 = x0 OP data[i];
    *dest = x0 OP x1;
}
```



循环展开的思想

- 尝试通过控制参数，接近吞吐量的最优化
  - Unrolling
  - Accumulating

```
for (i =0; i < limit; i+=2){
    x0 = x0 OP data[i];
    x1 = x1 OP data[i+1];
}
```

SIMD

- ICS Lecture 06

单指令多数据

`&, |, ~, ...` 对于整数里的每一个 bit 来说是独立（并行）的

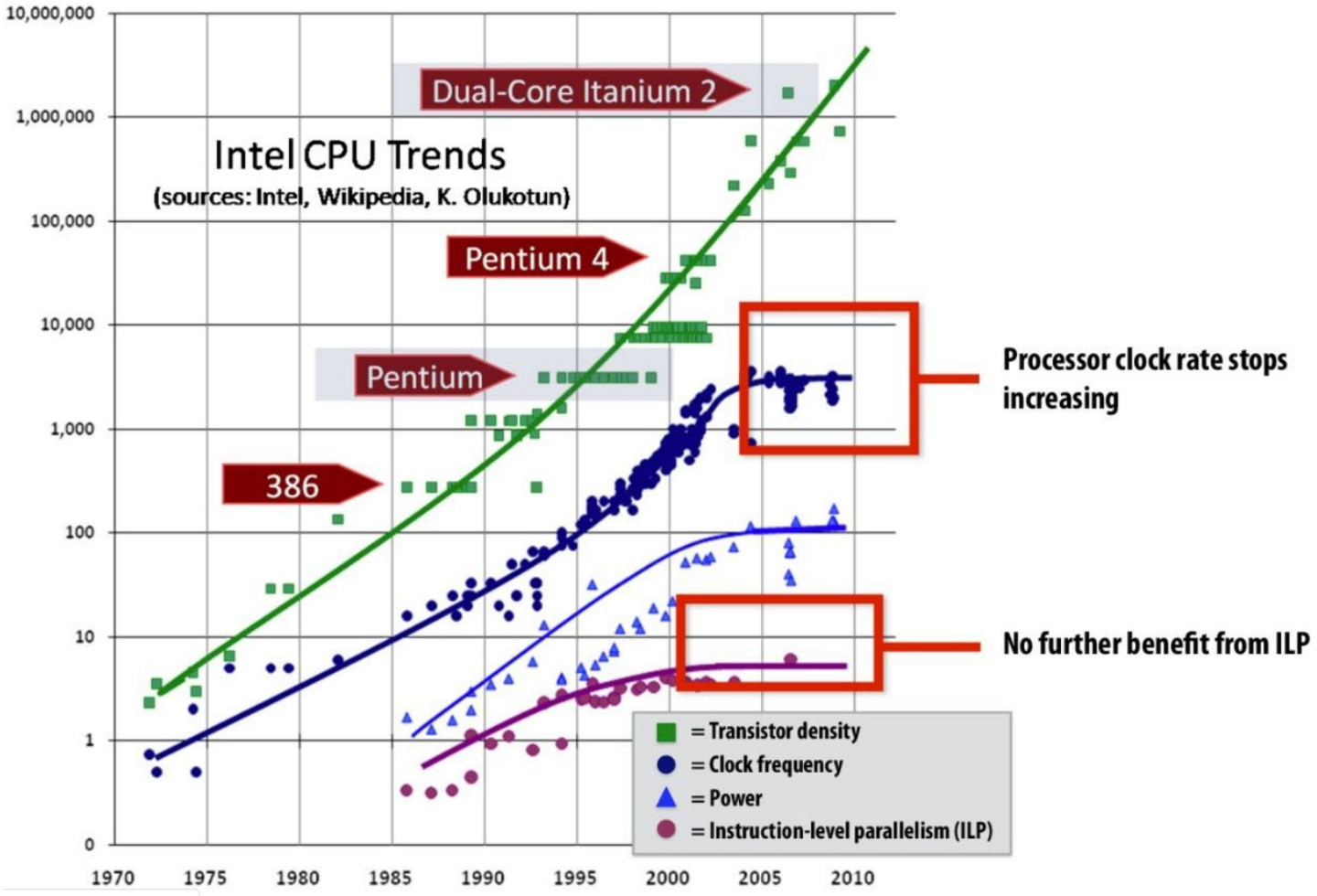
- 如果我们操作的对象刚好每一个 bit 是独立的
  - 我们在一条指令里就实现了多个操作
  - SIMD (Single Instruction, Multiple Data)

- 例子：Bit Set
  - 32-bit 整数 $x \to S \subseteq \{0, 1, 2, 3, \ldots, 31\}$
  - 位运算是对所有 bit 同时完成的
    - C++ 中有 bitset，性能非常可观

```
5 -> 0101
```

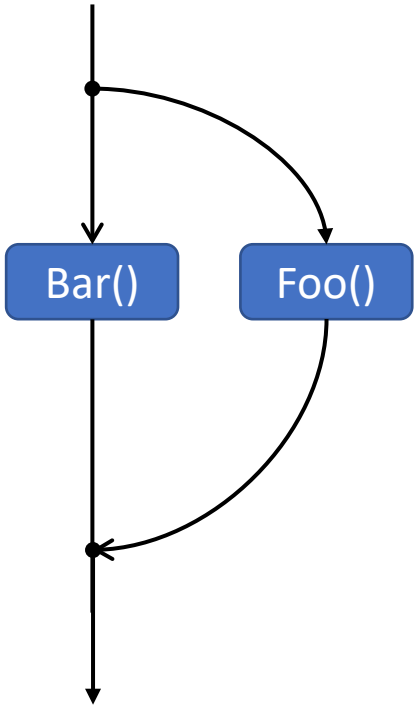
```
S: {0,2}
```

多核时代：parallel thinking



parallel thinking

```
//part A

Click_spawn Foo();
Bar();
Click_sync;

//part B
```



- E.g., QuickSort
  - Parallel
    - QuickSort($A_{1...S}$)
    - QuickSort($A_{S+1...|A|}$),

总结

- 程序性能的优化需求明显
  - Optimization重要
  - Readable code更重要

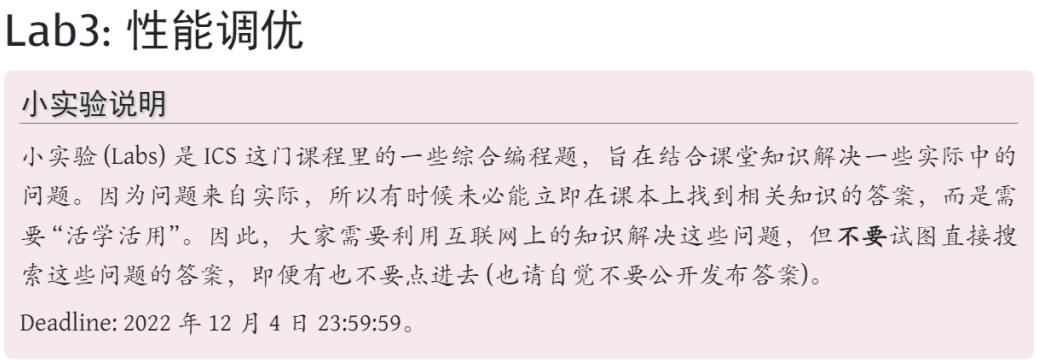
- 推荐的资料
  - CMU: 15-853: Algorithms in the “Real World”
    - A graduate-level course that provides many useful links to parallel algorithms and I/O-efficient algorithm
  - MIT 6.172: Performance Engineering of Software Systems
    - A more thorough explanation on performance analysis on parallel (multi-core) and distributed setting
  - CMU 15-210: Parallel and Sequential Data Structures and Algorithms
    - An overview of basic algorithms and data structures that makes no distinction between sequential and parallel

总结

- 选择适合的算法和数据结构
  - 算法课
  - Parallel thinking

- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 理解编译器优化的能力和局限
  - 基本编码原则和低级优化

- 现实的性能诊断任务，用好trace和profiler工具



Life in real world

Lab3: 性能调优

小实验说明

小实验 (Labs) 是 ICS 这门课程里的一些综合编程题，旨在结合课堂知识解决一些实际中的问题。因为问题来自实际，所以有时候未必能立即在课本上找到相关知识的答案，而是需要“活学活用”。因此，大家需要利用互联网上的知识解决这些问题，但**不要**试图直接搜索这些问题的答案，即便有也不要点进去 (也请自觉不要公开发布答案)。

Deadline: 2022 年 12 月 4 日 23:59:59。

- 用好trace和profiler工具

PA + Lab

PA大作业

每次实验前，请仔细阅读实验须知/提交方法和PA实验指南。

- PA0: 环境安装与配置 (已截止，DDL: 2022年9月20日23:59:59 (extended))
- PA1: 监视器 (已发布，DDL：2022年10月13日23:59:59 (extended))
- PA2: 模拟指令运行（已发布，DDL：2022年11月20日23:59:59）
- PA3: 中断与异常（已发布，DDL：2022年12月18日23:59:59）
- PA4: 分时多任务（已发布，DDL：2023年1月15日23:59:59）

Lab小作业

每次实验前，请仔细阅读实验须知/提交方法。

- Lab1: 大整数运算（已截止，DDL: 2022年10月16日23:59:59）
- Lab2: x86-64内联汇编（已截止，DDL：2022年11月13日23:59:59）
- Lab3: 性能调优（已发布，DDL：2022年12月4日）
- Lab4: 缓存模拟器（已发布，DDL：2022年12月25日）