优化程序性能选讲

王慧妍

why@nju.edu.cn

南京大学 软件学院 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];
```

```
 5 set_row:
 6 .LFB0:
 7   .cfi_startproc
 8   endbr64
 9   testq %rcx, %rcx
10   jle .L1
11   imulq %rcx, %rdx
12   leaq  (%rdi,%rdx,8), %rdx
13   movl  $0, %eax
14 .L3:
15   movsd (%rsi,%rax,8), %xmm0
16   movsd %xmm0, (%rdx,%rax,8)
17   addq  $1, %rax
18   cmpq  %rax, %rcx
19   jne .L3
20 .L1:
21   ret
```

```
# 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]

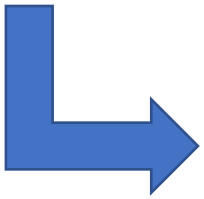
i=0: [3, 8, 16]

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

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

```
//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？

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

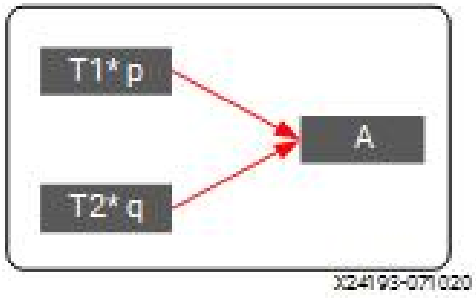
```
*xp = 4 * *xp
```

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

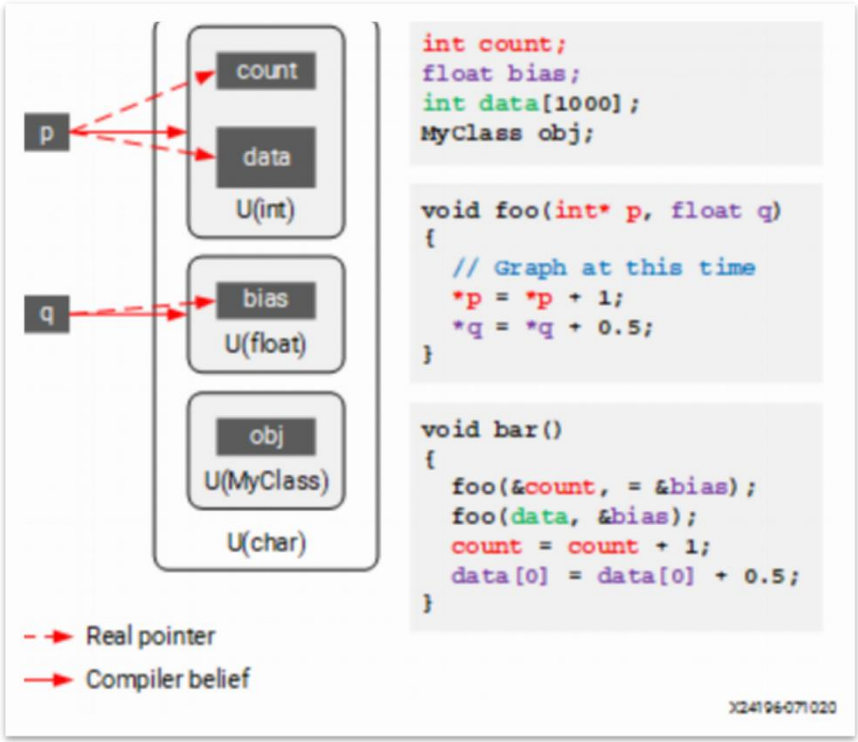
```
*xp = 3 * *xp
```

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

```c

// C program to illustrate aliasing
#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);
```

<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

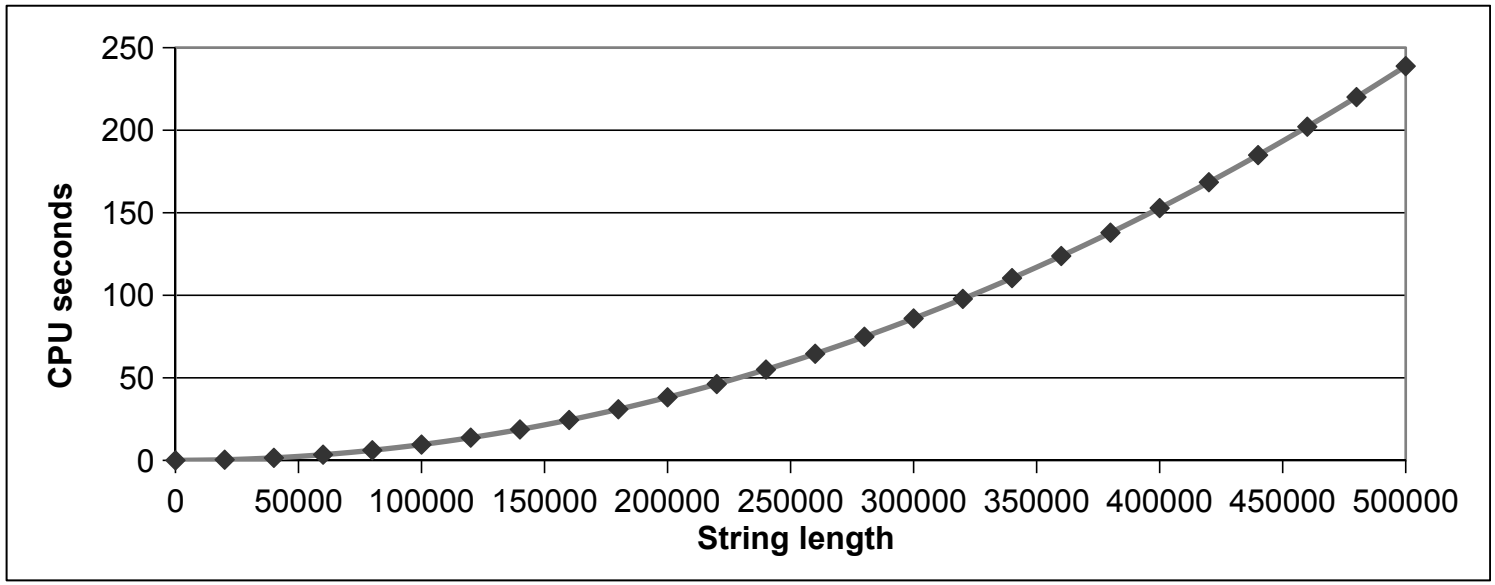
```
  -fstack-reuse=[all|named_vars|none]   all
  -fstdarg-opt                          [enabled]
  -fstore-merging                       [enabled]
  -fstrict-aliasing                     [enabled]
  -fstrict-enums                        [available in C++, ObjC++]
  -fstrict-volatile-bitfields      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

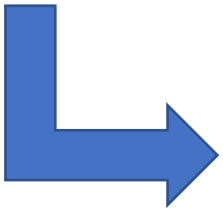
示例程序2.1

```
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’);
}
```

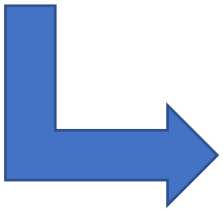


```
void lower(char *s){
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
        if(s[i] >= ‘A’ && s[i] <= ‘Z’)
            s[i] –= (‘A’ – ‘a’);
        i ++;
        if(i< strlen(s))
            goto loop;
done:
}
```

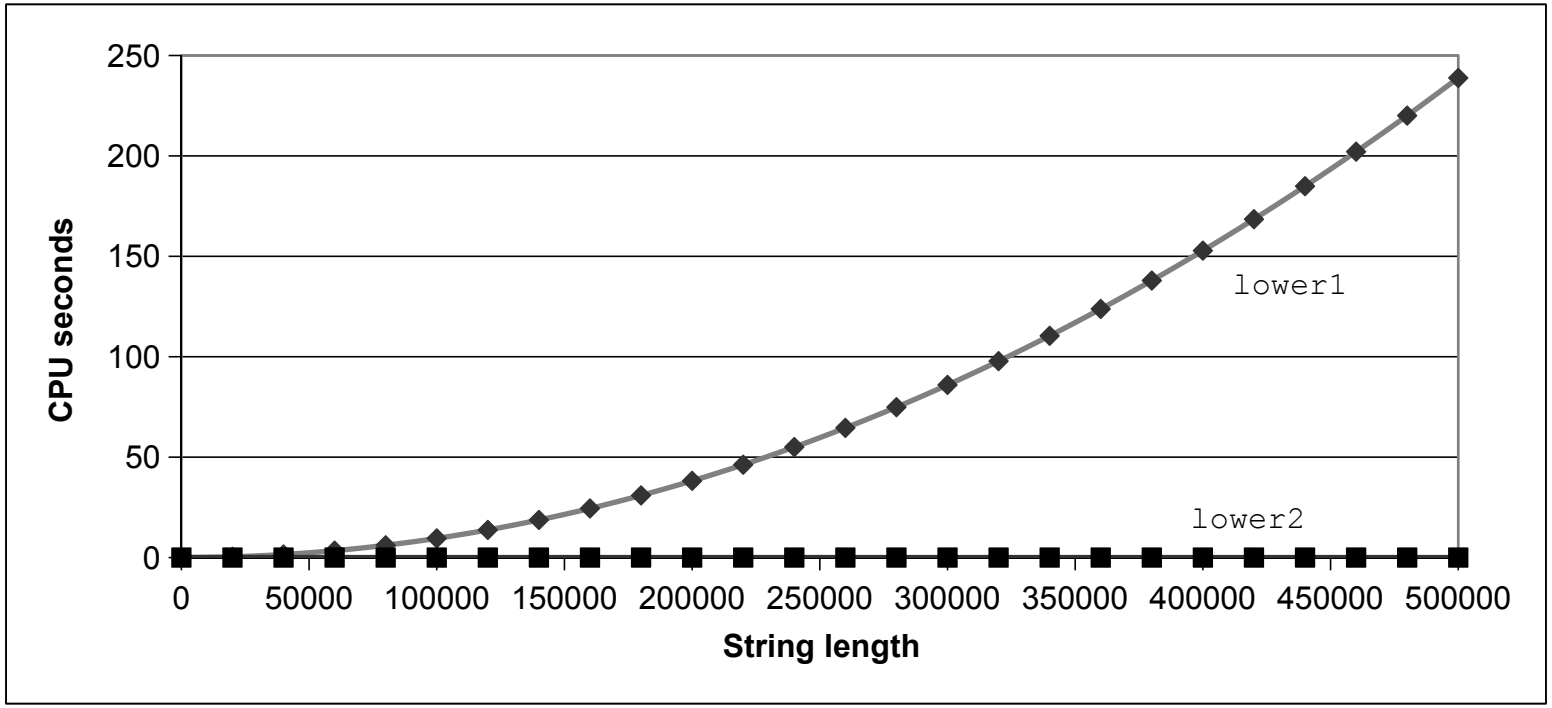
- strlen()：找结束符
  - 线性复杂度
  - N次进入循环，二次复杂度

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
50000
100000
150000
200000
250000
300000
350000
400000
450000
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

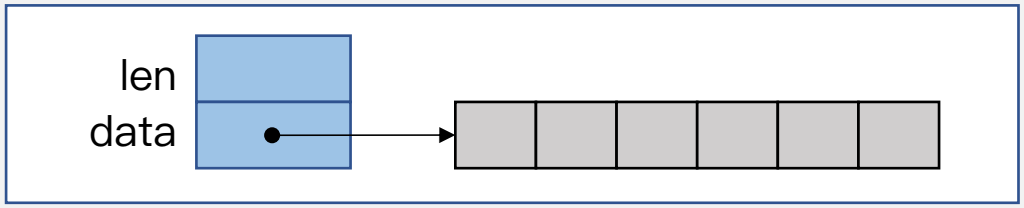
- Function call
  - 过程调用的副作用
    - 全局变量
  - 每次调用返回值不一定一样

- 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;
    }
}
```

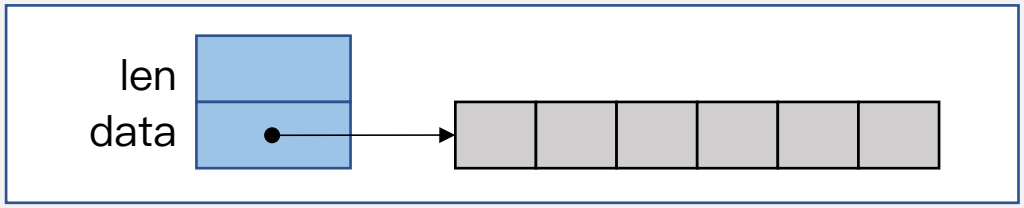


1

- 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;
    }
}
```

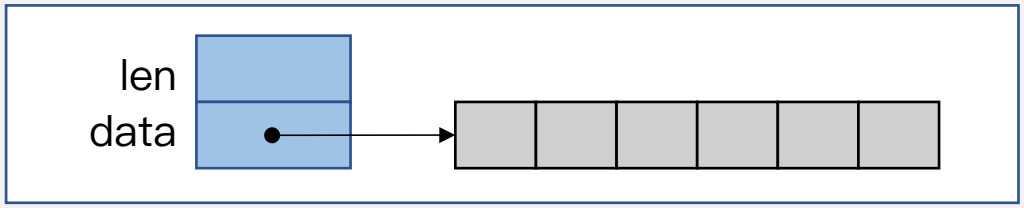


1

- 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;
    }
}
```

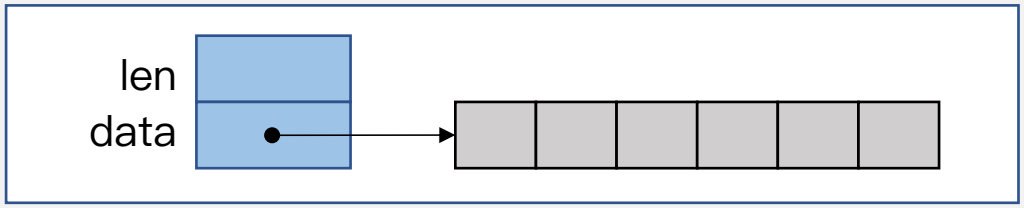


2

- 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];
    }
}
```



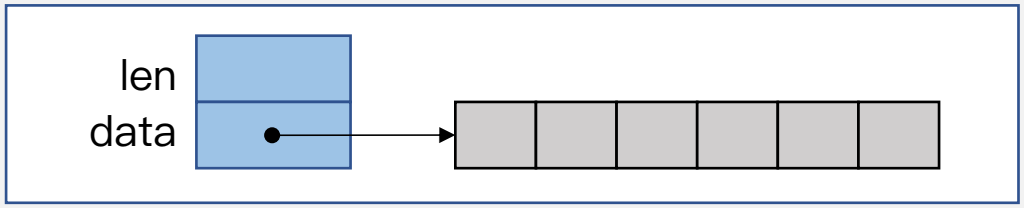
2

- 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



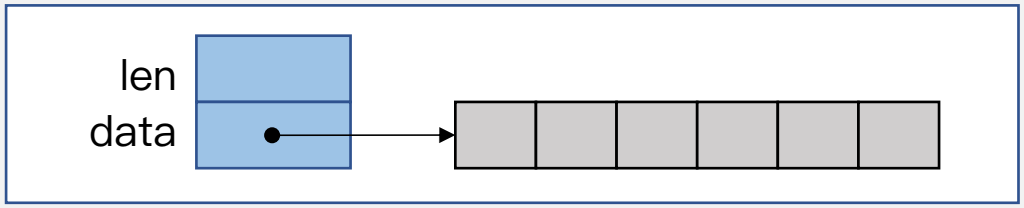

```
.L3:
  movq  (%rax), %rdx
  addq  %rdx, (%rbx)
  addq  $8, %rax
  cmpq  %rcx, %rax
  jne .L3
```

- 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
```

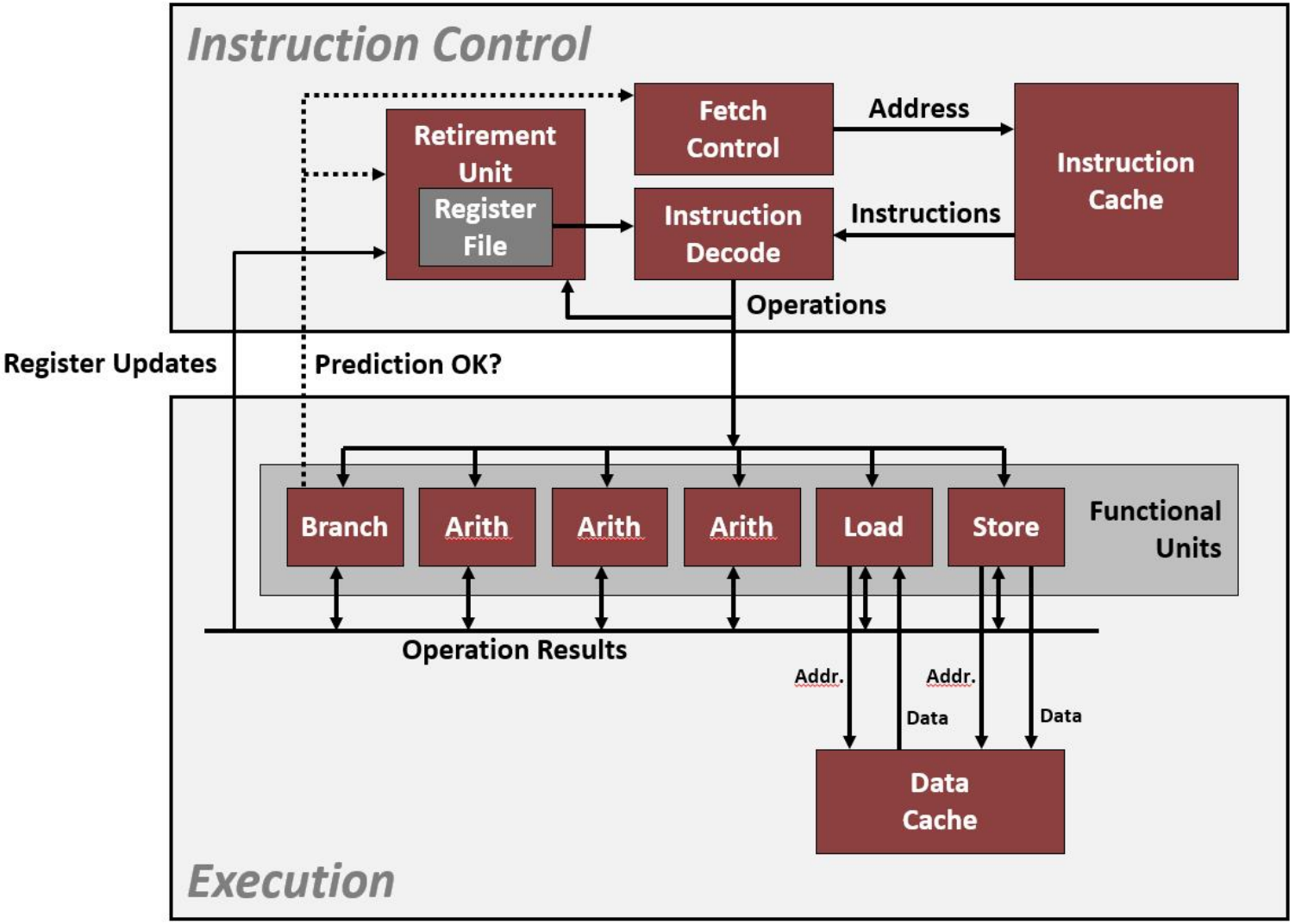
- 消除循环低效
- 减少函数调用
- 减少无需的内存访问

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

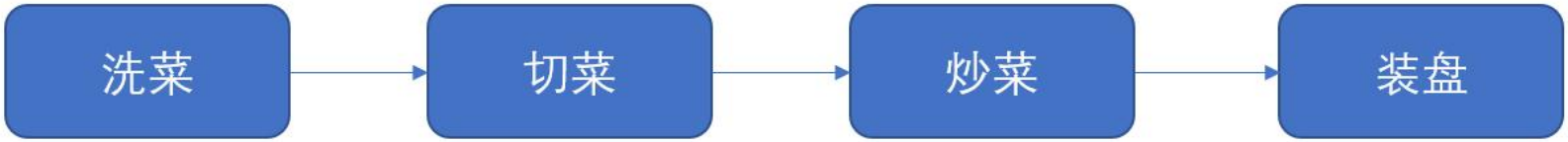


From CMU CSAPP Course

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

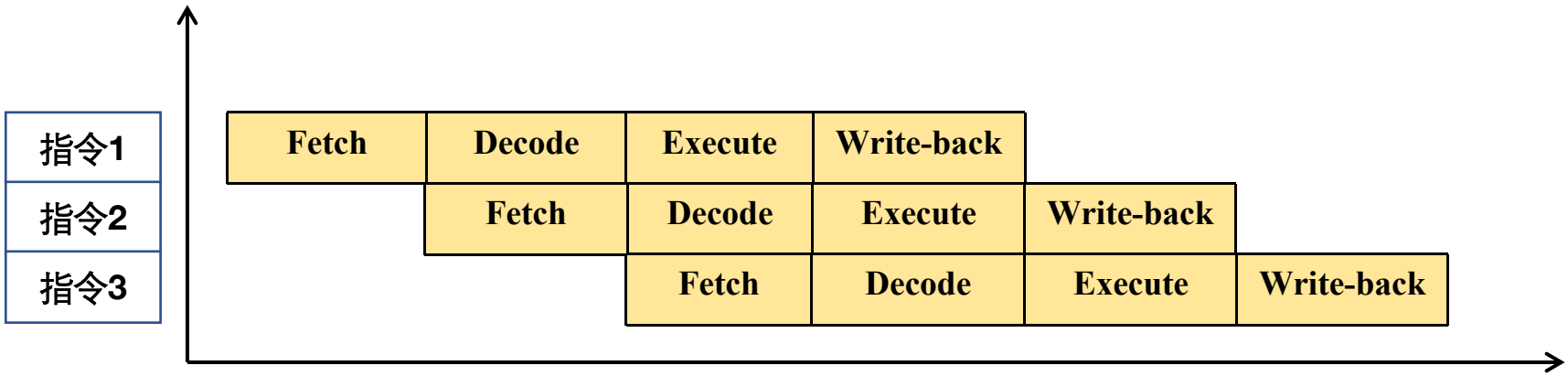
```
long mult_eq(long a, long b, long c){
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



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

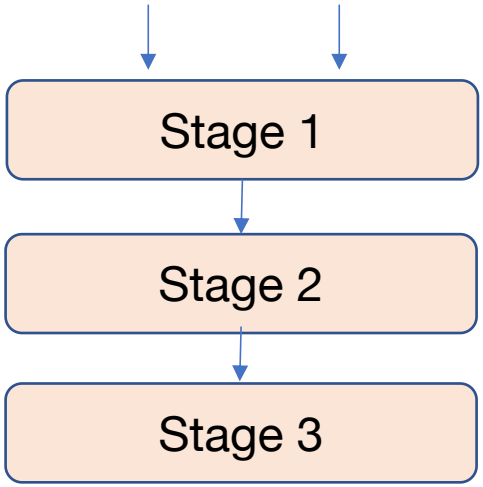
```
long mult_eq(long a, long b, long c){
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



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

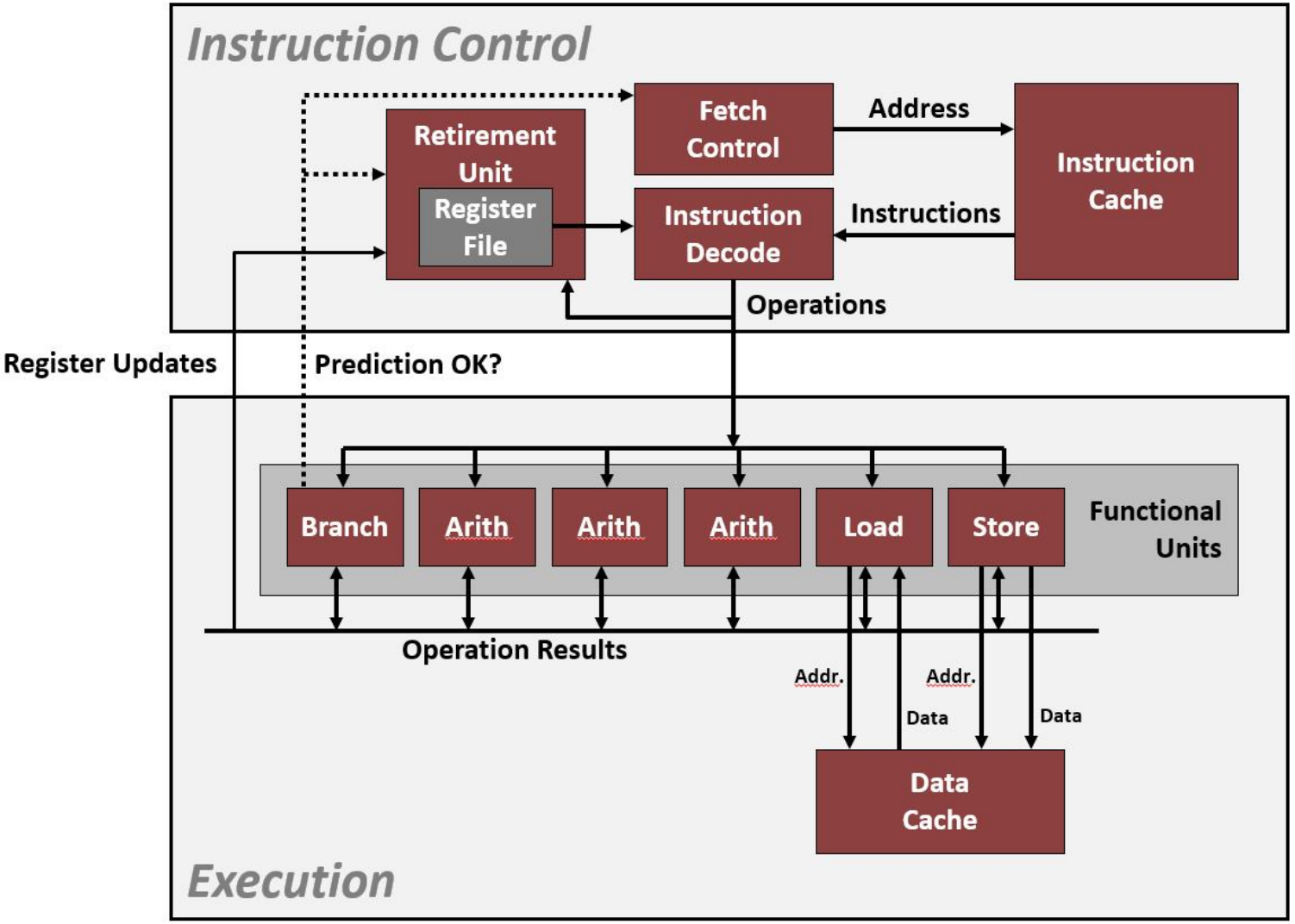
  - java - Why is processing a sorted array faster than processing an unsorted array? - Stack Overflow

Likely and unlikely in Linux Kernel

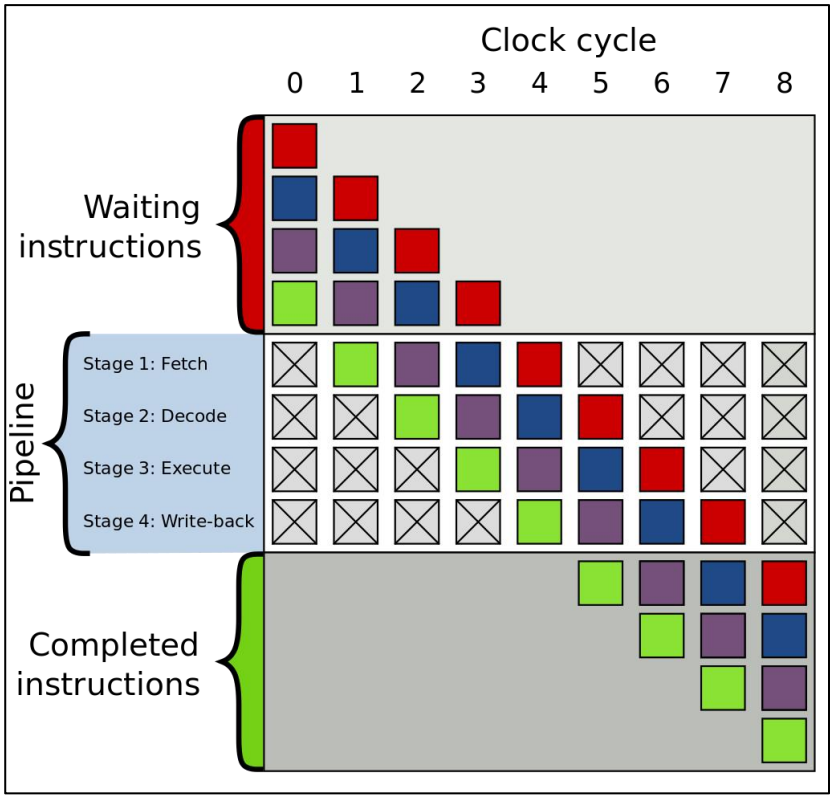
- _builtin_expect由gcc 引入
  - 该函数作用是允许程序员将最有可能执行的分支告诉编译器，来辅助系统进行分支预测

```
//ip 层收到消息后，如果是tcp就调用tcp_v4_rcv作为tcp协议的入口
int tcp_v4_rcv(struct sk_buff *skb)
{
  ...
        if (unlikely(th->doff < sizeof(struct tcphdr) / 4))
                goto bad_packet; //概率很小
        if (!pskb_may_pull(skb, th->doff * 4))
                goto discard_it;

//file: net/ipv4/tcp_input.c
int tcp_rcv_established(struct sock *sk, ...)
{
 if (unlikely(sk->sk_rx_dst == NULL))
  ......
}
//file: include/linux/compiler.h
#define likely(x)   __builtin_expect(!!(x),1)
#define unlikely(x) __builtin_expect(!!(x),0)
```

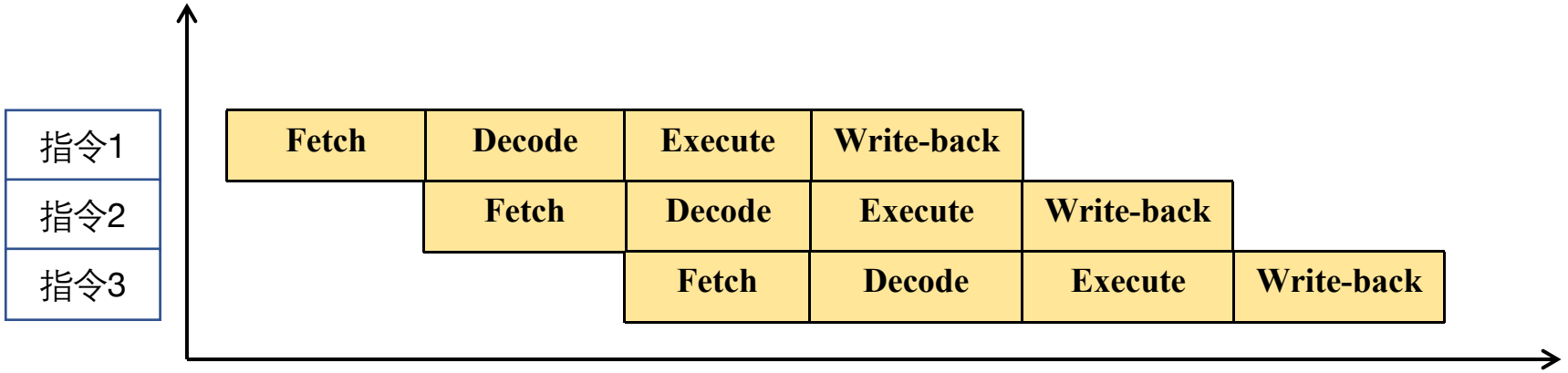
现代处理器



From CMU CSAPP Course




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



GCC 4.6.1 with -O3 or -ftree-vectorize on x64 no difference between the sorted and unsorted data (both are fast)

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

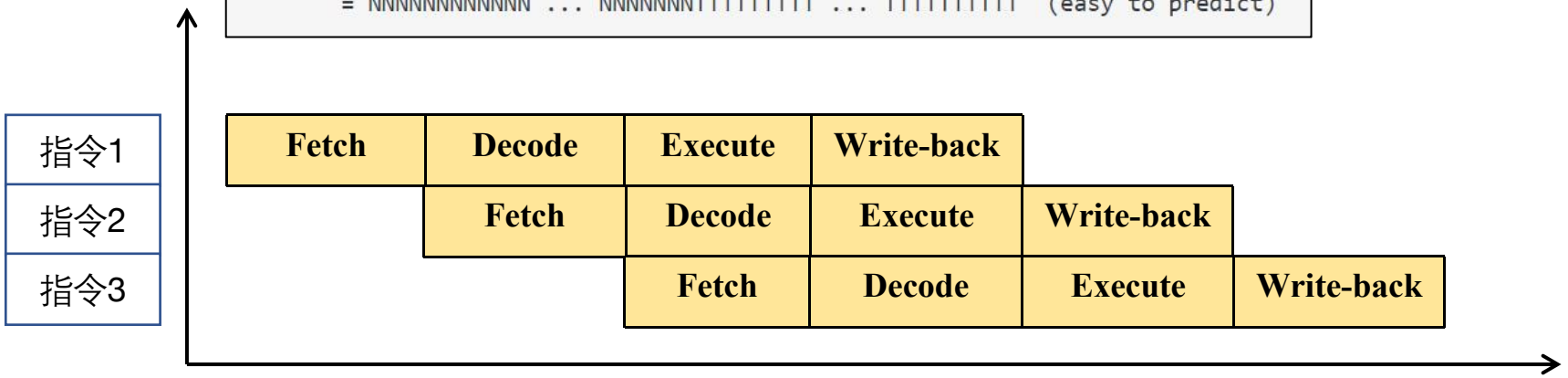
```
int data[N];
int sum = 0;
for (int i = 0; i<N; i++){
      if(data[i] < 128)
            sum += data[i];
}
```

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)
```

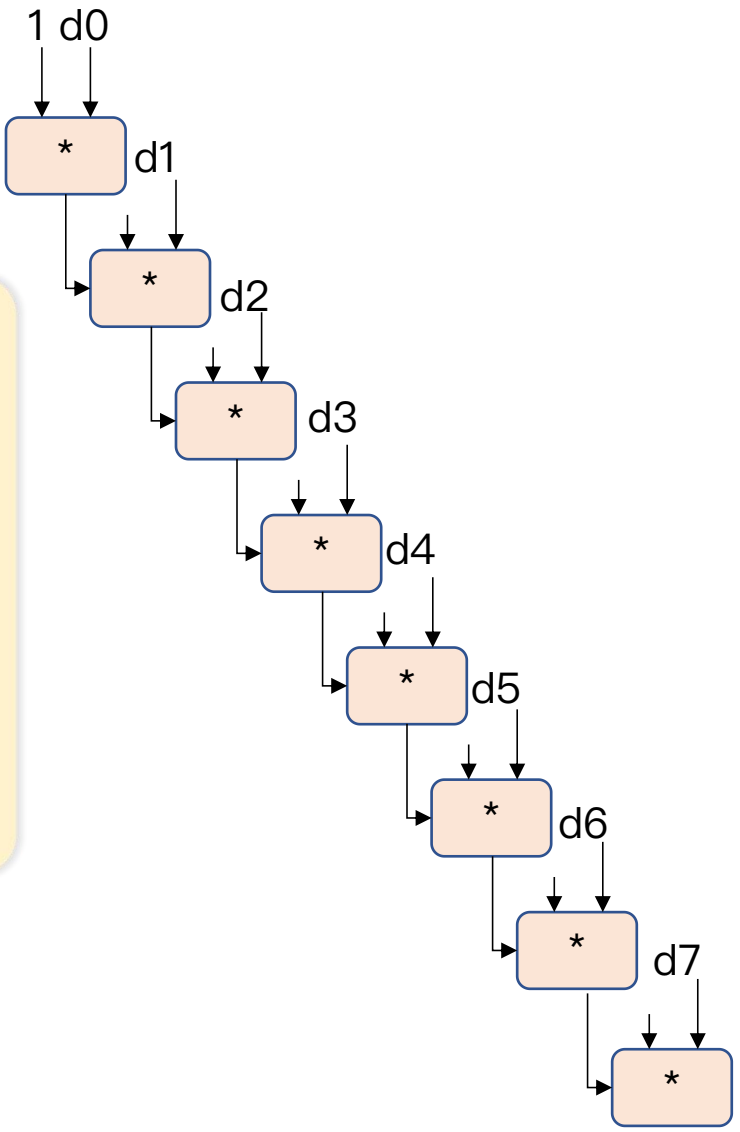
Unsorted:

```
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 ... TTTTTTTTTT  (easy to predict)
```



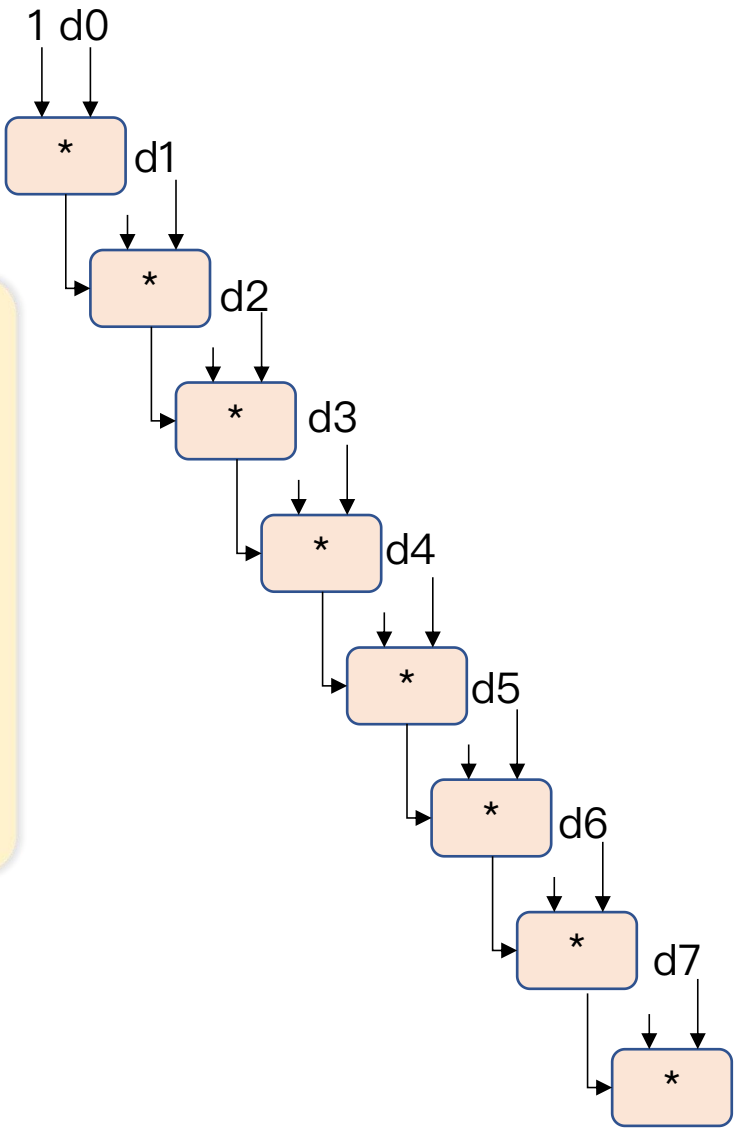
考虑指令并行的优化

```
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;
}
```

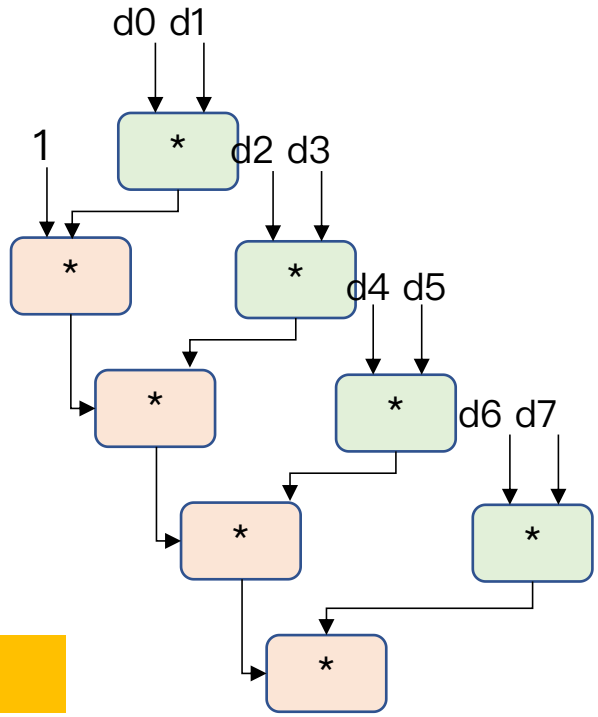


循环展开2

- 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;
}
```

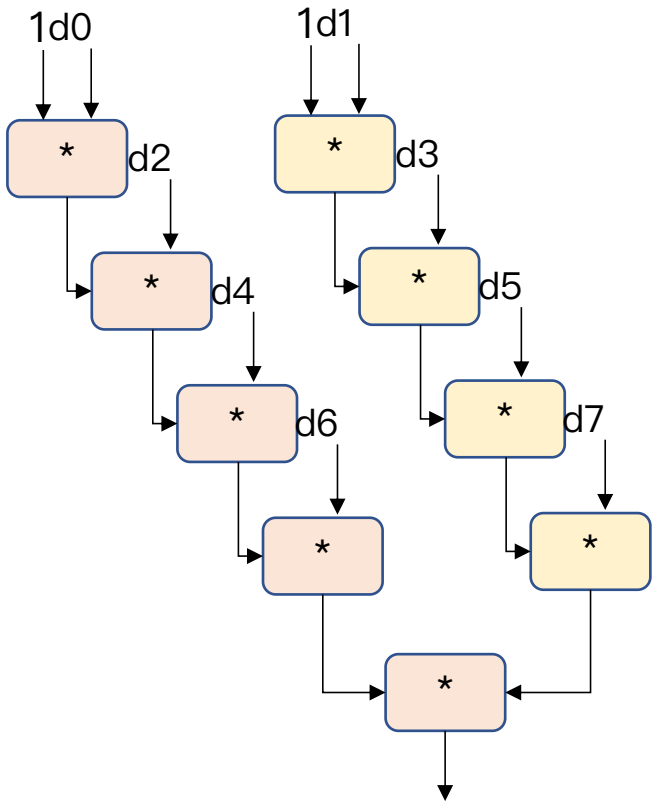
```
acc = acc OP (data[i] OP data[i+1]);
```



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;
}
```



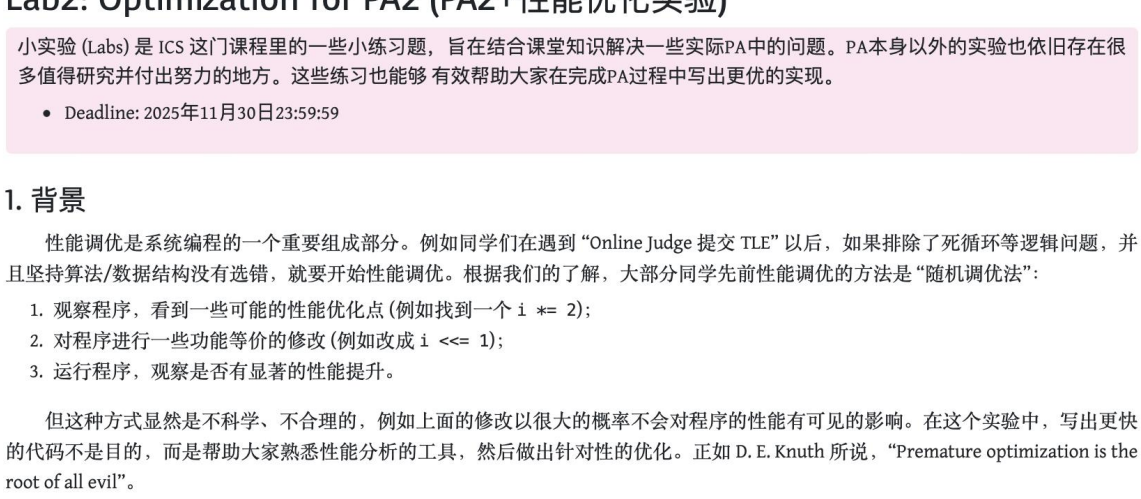
循环展开的思想

- 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];
}
```

性能优化方法: 拍脑袋 vs. 方法论

- Lab2



Lab2: Optimization for PA2 (PA2+性能优化实验)

小实验 (Labs) 是 ICS 这门课程里的一些小练习题，旨在结合课堂知识解决一些实际PA中的问题。PA本身以外的实验也依旧存在很多值得研究并付出努力的地方。这些练习也能够 有效帮助大家在完成PA过程中写出更优的实现。

- Deadline: 2025年11月30日23:59:59

1. 背景

性能调优是系统编程的一个重要组成部分。例如同学们在遇到 “Online Judge 提交 TLE” 以后，如果排除了死循环等逻辑问题，并且坚持算法/数据结构没有选错，就要开始性能调优。根据我们的了解，大部分同学先前性能调优的方法是 “随机调优法”:

1. 观察程序，看到一些可能的性能优化点 (例如找到一个 `i *= 2`);
2. 对程序进行一些功能等价的修改 (例如改成 `i <<= 1`);
3. 运行程序，观察是否有显著的性能提升。

但这种方式显然是不科学、不合理的，例如上面的修改以很大的概率不会对程序的性能有可见的影响。在这个实验中，写出更快的代码不是目的，而是帮助大家熟悉性能分析的工具，然后做出针对性的优化。正如 D. E. Knuth 所说，“Premature optimization is the root of all evil”。

- 如何在一个复杂系统里面优化性能, 是一个正经的科学问题
  - 是科学问题, 就有科学的方法
  - 评估当前的性能
  - 寻找性能瓶颈
  - 采用合适的优化方法

  - 评估优化后的性能, 对比获得的性能提升是否符合预期

往年的性能调优Lab

- 实验要求：对给定源码，通过perf工具定位其性能瓶颈，并修改源码使其保持功能一致前提下，实现10倍以上性能提升。
  - 给定源码为一份 Eratosthenes 筛法求 2,3,…,n 中素数的参考代码

```
// 初始时，is_prime[] 为 true
for (int i = 2; i <= n; i++) {
  for (int j = i + i; j <= n; j += i) {
    is_prime[j] = false;
  }
}
```

优化要求：

- 允许修改数据结构的定义和算法
- 在编译时会开启 -O1 优化开关。除了一些较为激进的优化(如循环展开等)外，编译器依然会尽可能地分析数据流、内联函数等方式优化代码，例如 -fdefer-pop, -finline-functions-called-once 等
- 允许做一些预处理，以及鼓励使用预处理，但限制 sieve.c 的大小不能超过 4096 字节
- 实际运行的测试用例中，n小于10^7因此不必修改n的定义

性能评估第一步：选benchmark

- 拍脑袋 = 凭直觉, 科学方法 = 量化分析
  - 用数据说话是基本素养
  - 需要一个量化的指标来衡量 “性能”

- 性能高 = 跑得快 ≈ 程序的执行时间!
  - 现实中情况有很多，要评估所有情况是不现实的

- 需要跑有代表性的输入 - 越能代表真实的应用场景, 就越合适
  - 代表性 = 优化技术在这些输入上带来的性能收益, 与真实应用场景中的性能收益趋势基本一致

  - 不同的应用场景 -> 性能收益的趋势不同 -> 需要不同的代表性输入 -> 不同的benchmark

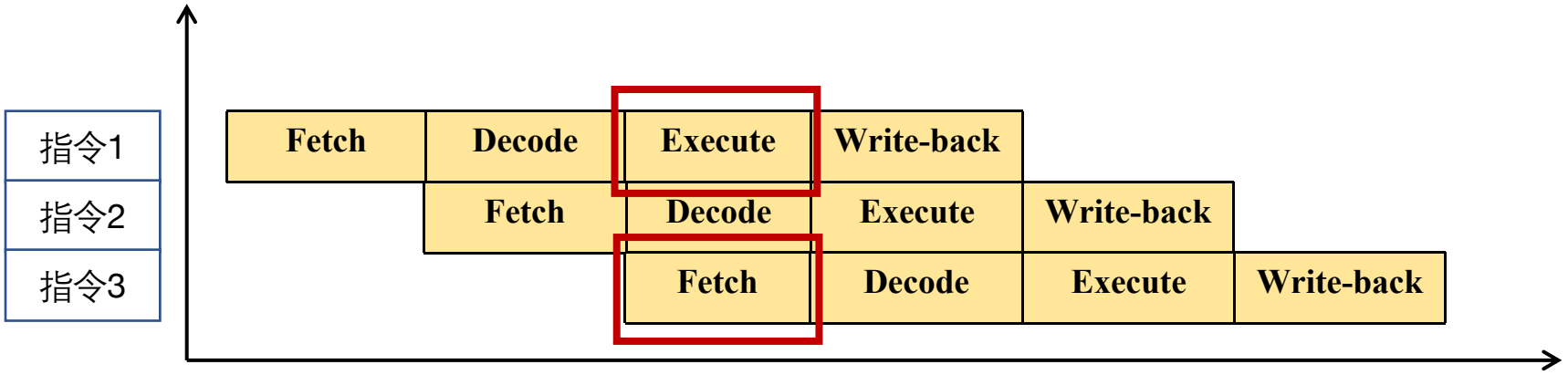
如何优化运行时间?

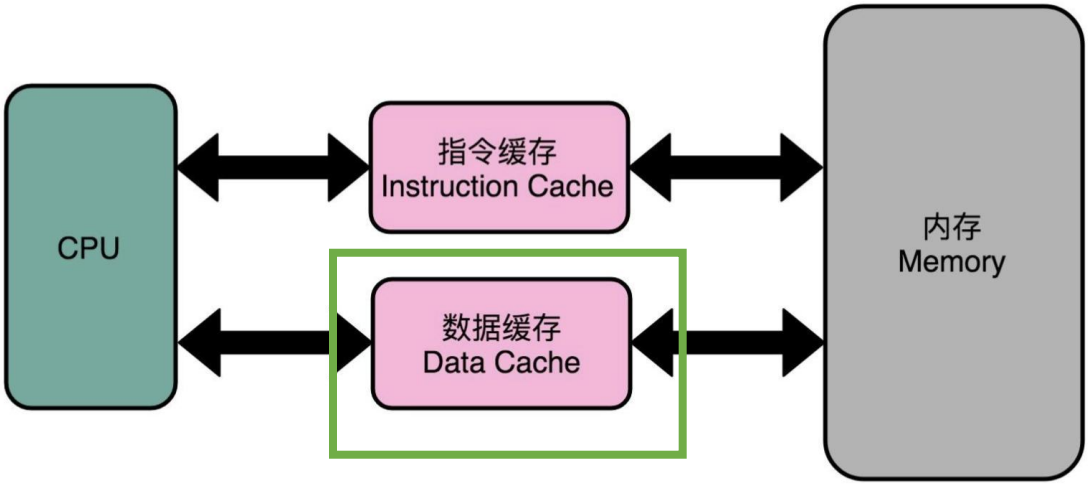
- 需要分析运行时间受哪些因素影响

$$\text{perf} = \frac{\text{time}}{\text{prog}} = \frac{\text{inst}}{\text{prog}} \cdot \frac{\text{cycle}}{\text{inst}} \cdot \frac{\text{time}}{\text{cycle}}$$

- 减少`inst/prog`：即减少执行的指令数，编译阶段
  - 优化措施：改进算法, 编译优化
- 减少`cycle/inst`：即减少 CPU 等待空转的周期，运行阶段
  - 优化措施：CPU 友好的代码，避免等待，~~换更好的 CPU~~
- 减少`time/cycle`：与CPU主频强相关
  - 优化措施：~~换更好的 CPU~~

Cache

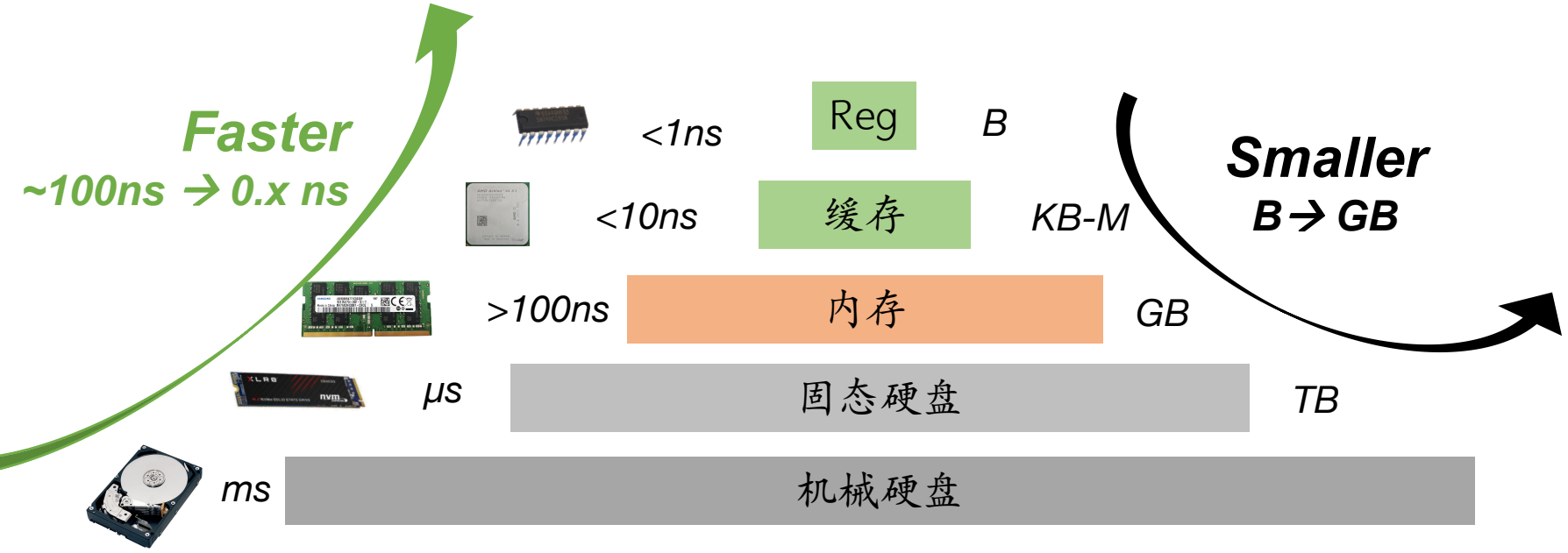




现代CPU的混合架构

存储架构

- 性能优化：尽可能极大利用越高速的存储空间
  - 缓存命中，寄存器使用



局部性原理

- 时间局部性 (temporal locality)
  - 如果程序引用一个数据一次，则很可能在不久的将来再次引用该数据
  - Guideline：保持经常访问的数据（如循环中的计数器或频繁查询的状态变量）在寄存器中，以减少对缓存或主存的访问
- 空间局部性 (spatial locality)
  - 如果程序引用一个数据，则很可能在不久的将来也会引用附近的数据
  - Guideline：程序应尽可能将相关数据紧凑存储在一起
- E.g., bitset, register

如何量化数据？

- 统计CPU在各个阶段耗费的时间：perf
  - 基于事件采样的原理，以性能事件为基础，支持针对处理器相关性能指标与操作系统 相关性能指标的性能剖析，可用于性能瓶颈的查找与热点代码的定位

```
 1,012,981,908      cycles
   751,436,010      instructions
    13,248,364      L1-dcache-loads
    92,143,312      L1-dcache-load-misses
       877,867      LLC-loads
        11,988      LLC-load-misses

  0.257386618 seconds time elapsed
```

```
     236.37 msec task-clock                #    0.998 CPUs utilized
          0      context-switches          #    0.000 /sec
          0      cpu-migrations            #    0.000 /sec
      3,150      page-faults               #   13.327 K/sec
937,451,729      cycles                    #    3.966 GHz
747,161,568      instructions              #    0.80  insn per cycle
196,839,478      branches                  #  832.768 M/sec
    940,069      branch-misses             #    0.48% of all branches
```

寻找性能瓶颈

- 性能瓶颈究竟在哪里? 哪些优化工作是值得做的? 优化工作的预期性能收益是多少?

- Amdahl's law可以告诉我们答案:

  The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

  - 优化系统中某部分所获得的总体性能收益, 受限于那部分实际使用的时间占比.
  - 公式表示: 假设系统某部分实际使用的时间占比是p, 该部分在优化后的加速比是s, 则整个系统的加速比为 f(s) = 1 / (1 - p + p / s)
    - 某程序的运行过程分为独立的两部分A和B, 其中A占80%, B占20%
    - 若将B优化5倍, 则加速比是1/(0.8+0.2/5)=1.1905 若将A优化2倍, 则加速比是1/(0.2+0.8/2)=1.6667

往年的性能优化实验Lab3

- 实验要求：对给定源码，通过perf工具定位其性能瓶颈，并修改源码使其保持功能一致前提下，实现10倍以上性能提升。
  - 给定源码为一份 Eratosthenes 筛法求 2,3,…,n 中素数的参考代码

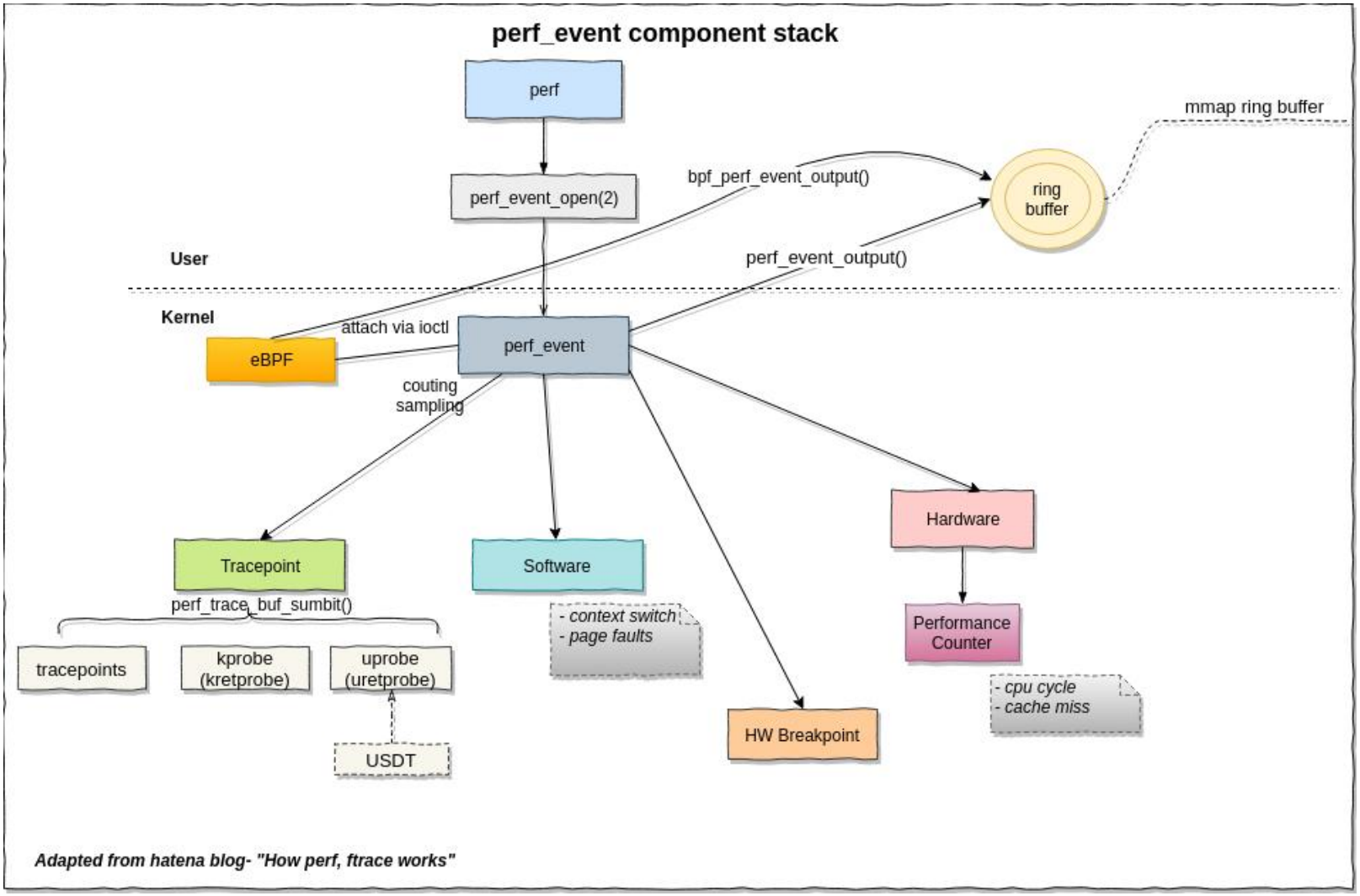
```
// 初始时，is_prime[] 为 true
for (int i = 2; i <= n; i++) {
  for (int j = i + i; j <= n; j += i) {
    is_prime[j] = false;
  }
}
```

优化要求：

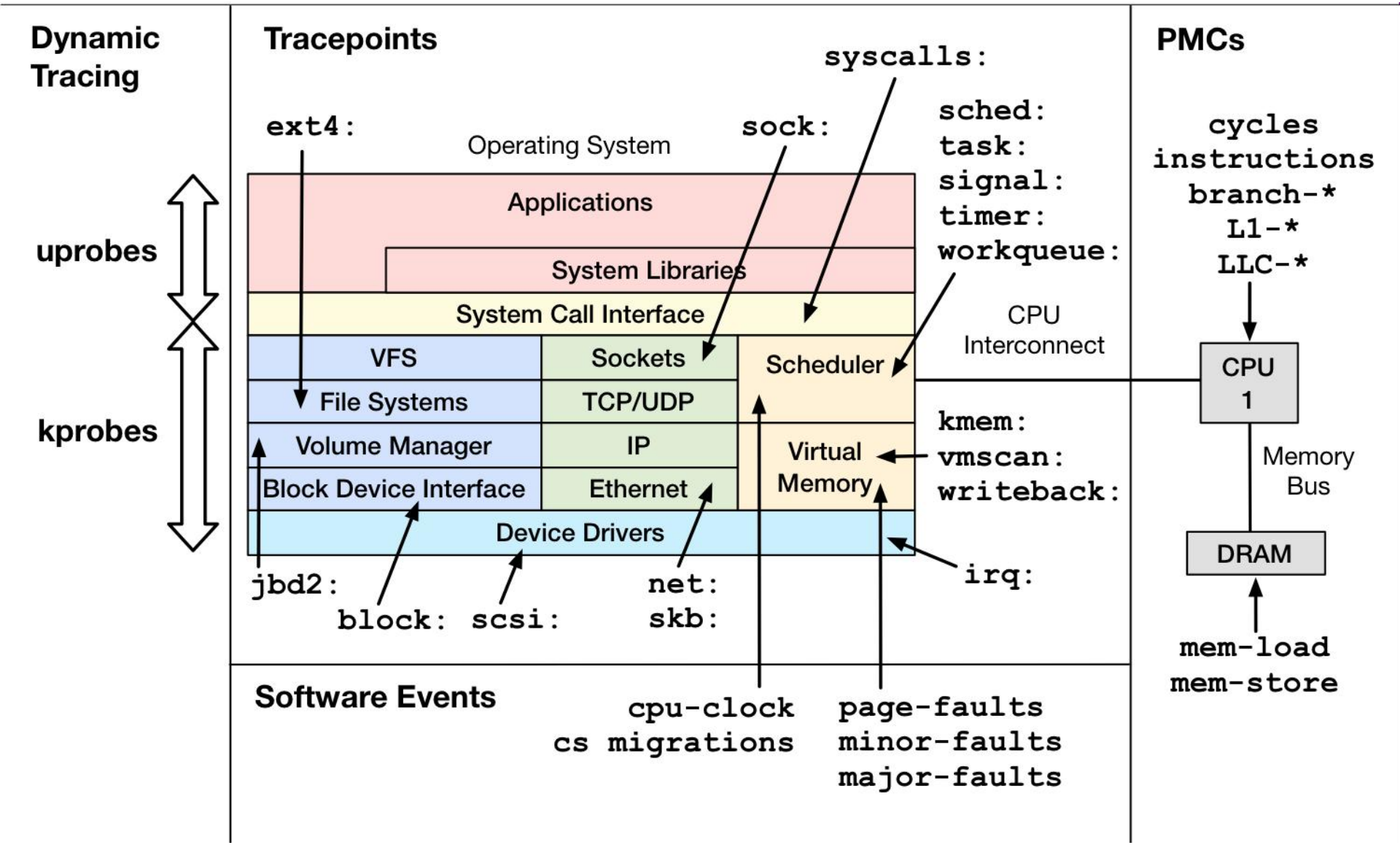
- 允许修改数据结构的定义和算法
- 在编译时会开启 -O1 优化开关。除了一些较为激进的优化(如循环展开等)外，编译器依然会尽可能地分析数据流、内联函数等方式优化代码，例如 -fdefer-pop, -finline-functions-called-once 等
- 允许做一些预处理，以及鼓励使用预处理，但限制 sieve.c 的大小不能超过 4096 字节
- 实际运行的测试用例中，n小于10^7因此不必修改n的定义

Linux Perf

- 基于事件采样的原理，以性能事件为基础，支持针对处理器相关性能指标与操作系统相关性能指标的性能剖析，可用于性能瓶颈的查找与热点代码的定位



Linux perf_events Event Sources



[Linux perf Examples (brendangregg.com)]

如何发现性能瓶颈？

- Perf工具

$man perf

$perf record
$perf report
$perf annotate
$perf stat

```
PERF(1)

NAME
       perf - Performance analysis tools f

SYNOPSIS
       perf [--version] [--help] [OPTIONS]

OPTIONS
       -h, --help
           Run perf help command.

       -v, --version
           Display perf version.

       -vv
           Print the compiled-in status of

       --exec-path
           Display or set exec path.

       --html-path
           Display html documentation path

       -p, --paginate
           Set up pager.

       --no-pager
           Do not set pager.

       --buildid-dir
           Setup buildid cache directory.

       --list-cmds
           List the most commonly used per

       --list-opts
           List available perf options.

 Manual page perf(1) line 1/66 29% (press
```

```
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
  annotate        Read perf.data (created by perf record) and display annotated code
  archive         Create archive with object files with build-ids found in perf.data file
  bench           General framework for benchmark suites
  buildid-cache   Manage build-id cache.
  buildid-list    List the buildids in a perf.data file
  c2c             Shared Data C2C/HITM Analyzer.
  config          Get and set variables in a configuration file.
  daemon          Run record sessions on background
  data            Data file related processing
  diff            Read perf.data files and display the differential profile
  evlist          List the event names in a perf.data file
  ftrace          simple wrapper for kernel's ftrace functionality
  inject          Filter to augment the events stream with additional information
  iostat          Show I/O performance metrics
  kallsyms        Searches running kernel for symbols
  kmem            Tool to trace/measure kernel memory properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  mem             Profile memory accesses
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.
  version         display the version of perf binary
  probe           Define new dynamic tracepoints
  trace           strace inspired tool

See 'perf help COMMAND' for more information on a specific command.
```

回到Lab3

- 带着程序优化的思路，再看看这个程序

```c
#define N 10000000
static bool is_prime[N];
static int  primes[N];

int *sieve(int n) {
  assert(n + 1 < N);
  for (int i = 0; i <= n; i++)
    is_prime[i] = true;

  for (int i = 2; i <= n; i++) {
    for (int j = i + i; j <= n; j += i) {
      is_prime[j] = false;
    }
  }

  int *p = primes;
  for (int i = 2; i <= n; i++)
    if (is_prime[i]) {
      *p++ = i;
    }
  *p = 0;
  return primes;
}
```

$perf stat

```
Performance counter stats for './a.out':

        236.37 msec task-clock                #    0.998 CPUs utilized
             0      context-switches          #    0.000 /sec
             0      cpu-migrations            #    0.000 /sec
         3,150      page-faults               #   13.327 K/sec
   937,451,729      cycles                    #    3.966 GHz
   747,161,568      instructions              #    0.80  insn per cycle
   196,839,478      branches                  #  832.768 M/sec
       940,069      branch-misses             #    0.48% of all branches
```

```
Performance counter stats for './a.out':

    1,012,981,908      cycles
      751,436,010      instructions
       13,248,364      L1-dcache-loads
       92,143,312      L1-dcache-load-misses
          877,867      LLC-loads
           11,988      LLC-load-misses

      0.257386618 seconds time elapsed

      0.257039000 seconds user
      0.000000000 seconds sys


                                              (65.79%)
 #    0.74  insn per cycle                    (82.87%)
                                              (82.85%)
```

回到Lab3

$perf record ./perftune && perf report

```
Samples: 922  of event 'cycles', Event count (approx.): 931927645
Overhead  Command  Shared Object       Symbol
  98.54%  a.out    a.out               [.] sieve
   0.52%  a.out    [kernel.kallsyms]   [k] __mod_node_page_state
   0.28%  a.out    [kernel.kallsyms]   [k] chacha_block_generic
   0.11%  a.out    [kernel.kallsyms]   [k] native_irq_return_iret
```

- 带着程序优化的思路，再看看

```c
#define N 10000000
static bool is_prime[N];
static int  primes[N];

int *sieve(int n) {
  assert(n + 1 < N);
  for (int i = 0; i <= n; i++)
    is_prime[i] = true;

  for (int i = 2; i <= n; i++) {
    for (int j = i + i; j <= n; j += i) {
      is_prime[j] = false;
    }
  }

  int *p = primes;
  for (int i = 2; i <= n; i++)
    if (is_prime[i]) {
      *p++ = i;
    }
  *p = 0;
  return primes;
}
```

```
Samples: 922  of event 'cycles', 4000 Hz, Event cou
sieve  /home/huiyan/Documents/Code/ics-workbench/pe
Percent
         lea     0x1(%rax),%rdx
         add     %rcx,%rdx
         test    %edi,%edi
       ↓ js      2c
  0.84 20:   movb    $0x1,(%rax)
         add     $0x1,%rax
         cmp     %rdx,%rax
  0.60   ↑ jne     20
       2c:   lea     primes,%rdx
         cmp     $0x1,%edi
       ↓ jle     bf
         lea     0x1(%rdi),%r8d
         mov     $0x2,%edx
         mov     $0x4,%esi
         lea     is_prime,%rcx
       51:   cmp     %esi,%edi
  1.19   ↓ jl      63
  0.11     mov     %rsi,%rax
 77.29 58:   movb    $0x0,(%rcx,%rax,1)
  5.09     add     %rdx,%rax
         cmp     %eax,%edi
  9.36   ↑ jge     58
  0.11 63:   add     $0x2,%rsi
  0.22     add     $0x1,%rdx
         cmp     %r8,%rdx
  0.65   ↑ jne     51
         mov     $0x2,%eax
         lea     primes,%rdx
         lea     is_prime,%rcx
       ↓ jmp     b1
       85:   sub     $0x8,%rsp
         lea     __PRETTY_FUNCTION__.0,%rcx
         mov     $0xc,%edx
         lea     _IO_stdin_used+0xd,%rsi
         lea     _IO_stdin_used+0x19,%rdi
       → call    _init
```

算法的改进，整体减少指令数

```
int *sieve(int n) {
  assert(n + 1 < N);
  //register int i = 0;
  int cnt = 0;
  for (int i = 0; i <= n; i++)
    is_prime[i] = true;
//  memset(is_prime, true, sizeof(is_prime));
  is_prime[0] = false;
  is_prime[1] = false;

  for (int i = 2; i <= n; i++) {
    if(is_prime[i] == true){
      primes[cnt++] = i;
    }
    for (int j = 0; j < cnt && i*primes[j] <= n; j++) {
        is_prime[i*primes[j]] = false;
        if(i % primes[j] == 0)
          break;
    }
```

$perf stat -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./perftune

```
213,808,048      cycles                                                  (58.79%)
308,506,769      instructions              #    1.44  insn per cycle     (79.41%)
 33,432,816      L1-dcache-loads                                         (84.23%)
  3,170,234      L1-dcache-load-misses     #    9.48% of all L1-dcache accesses  (86.31%)
     61,666      LLC-loads                                               (86.26%)
      5,720      LLC-load-misses           #    9.28% of all LL-cache accesses  (84.40%)

0.058829288 seconds time elapsed

0.054571000 seconds user
0.004197000 seconds sys
```

紧凑数据排布，去除重复调用

```
for (int i = 2; i <= n; i++) {
  if(is_prime[i] == true){
    primes[cnt++] = i;
  }
  for (int j = 0, tmp = 2; j < cnt && i*tmp <= n; j++) {
      tmp = primes[j];
      is_prime[i*tmp] = false;
      if(i % tmp== 0)
        break;
  }
```

```
for (int i = 3; i <= n; i=i+2) {
  if(is_prime[i] == true){
    primes[cnt++] = i;
  }
```

```
#define BITSET_SIZE N/sizeof(int)+1
typedef struct{
  unsigned int data[BITSET_SIZE];
}BitSet;

static BitSet is_prime;
static inline void initBitSet(BitSet *bitset){
  for(int i = 0; i < BITSET_SIZE; i++){
    bitset->data[i] = 0xFFFFFFFF;//UINT_MAX;
  }
}
static inline void setBit(BitSet *bitset, int index, int offset){
  bitset->data[index] |= (1 << offset);
}

static inline void clearBit(BitSet *bitset, int index, int offset){
  bitset->data[index] &= ~(1 << offset);
}

static inline int getBit(BitSet *bitset, int index, int offset){
  return (bitset->data[index] >> offset) & 1;

}
```

建议：基于sieve调用前后rdtsc指令结果评测加速倍数

```
> [[all-tests-passed]]
> =[[[-speedup-12.5791-]]]=
```

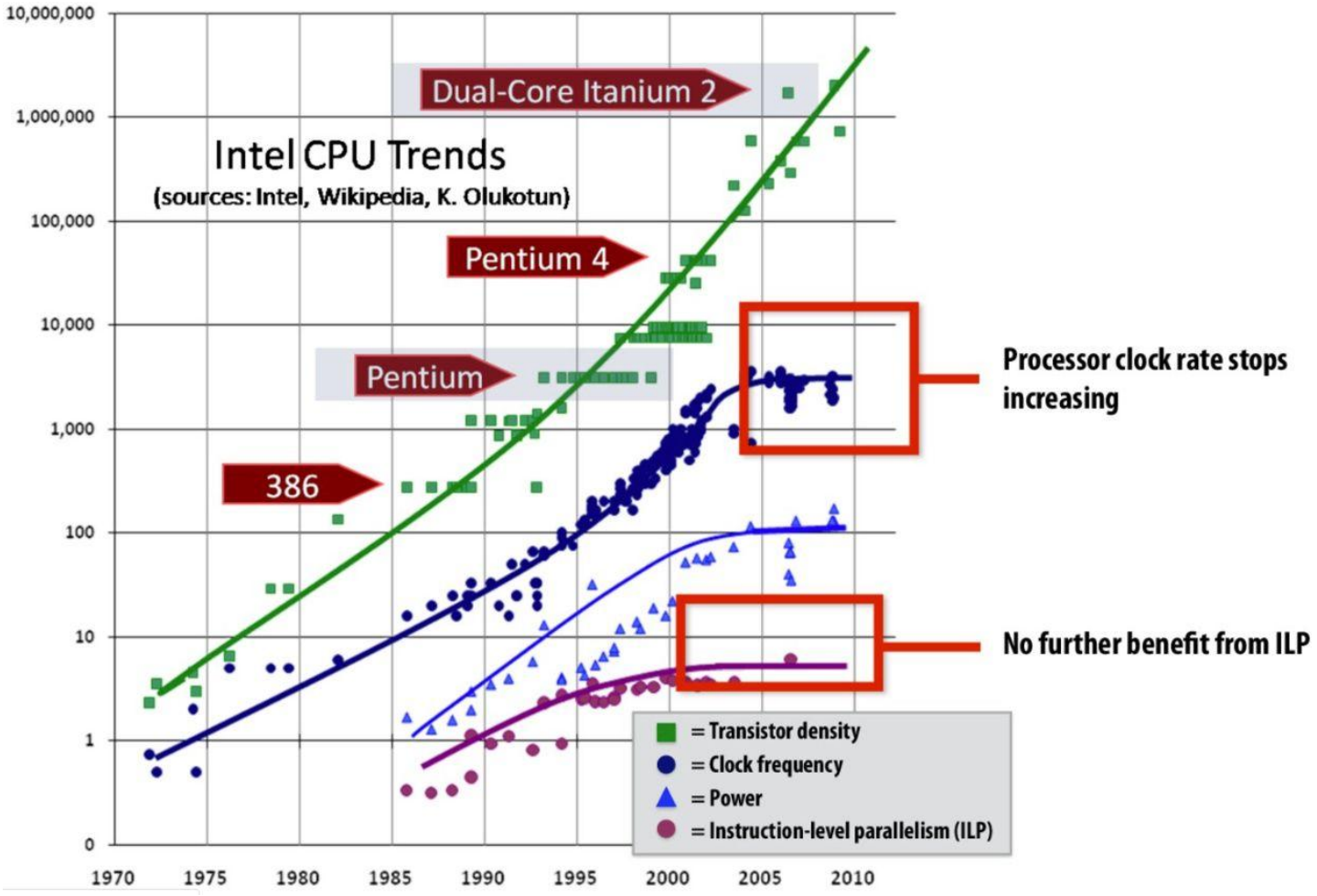
$perf stat -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./perftune

```
         33.05 msec task-clock                #    0.980 CPUs utilized
             0      context-switches          #    0.000 /sec
             0      cpu-migrations            #    0.000 /sec
         3,146      page-faults               #   95.180 K/sec
   112,416,584      cycles                    #    3.401 GHz
   165,998,826      instructions              #    1.48  insn per cycle
    39,935,278      branches                  #    1.208 G/sec
       953,283      branch-misses             #    2.39% of all branches
```

抛开workload谈优化就是耍流氓

- 这是一则来自软件工程领域的忠告
  - 优化占比50%的部分, 和优化占比5%的部分, 效果天差地别
- 启发: 不能依靠直觉来优化觉得哪里有优化机会就改哪里
  - 否则很容易采取了一个没有效果的方案, 甚至会在实际场景中造成性能倒退
- 根据评估数据采取合适的设计方案, 才是科学的做法
  - Amdahl's law = 小学数学应用题
  - 初学者"耍流氓"
  - 缺少相关的专业素养
  - Lab2 里锻炼专业素养比 "会套算法"重要得多

多核时代：parallel thinking



总结

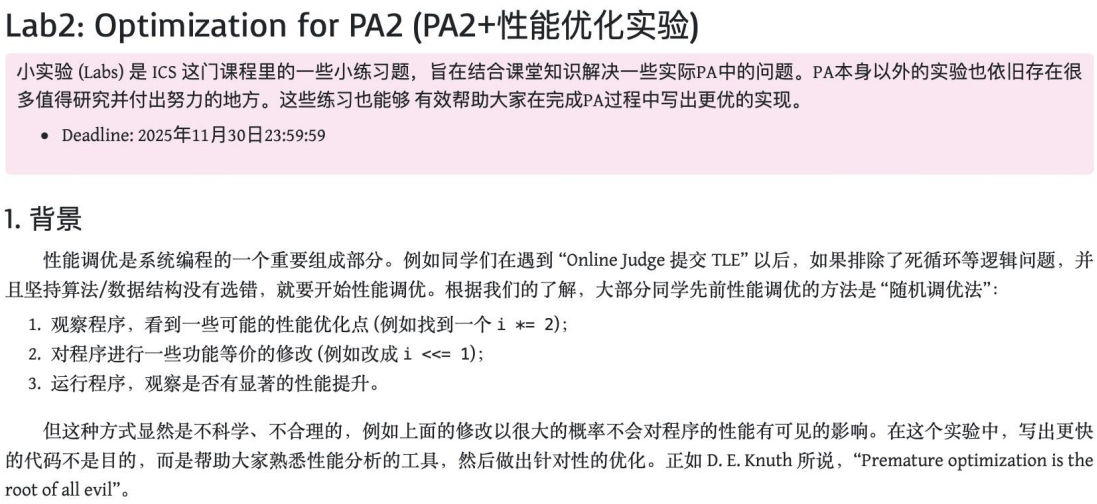
- 程序性能的优化需求明显
  - Optimization重要
  - Readable code更重要

- 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



Lab2: Optimization for PA2 (PA2+性能优化实验)

小实验 (Labs) 是 ICS 这门课程里的一些小练习题，旨在结合课堂知识解决一些实际PA中的问题。PA本身以外的实验也依旧存在很多值得研究并付出努力的地方。这些练习也能够 有效帮助大家在完成PA过程中写出更优的实现。

- Deadline: 2025年11月30日23:59:59

1. 背景

性能调优是系统编程的一个重要组成部分。例如同学们在遇到 “Online Judge 提交 TLE” 以后，如果排除了死循环等逻辑问题，并且坚持算法/数据结构没有选错，就要开始性能调优。根据我们的了解，大部分同学先前性能调优的方法是 “随机调优法”:

1. 观察程序，看到一些可能的性能优化点 (例如找到一个 `i *= 2`);
2. 对程序进行一些功能等价的修改 (例如改成 `i <<= 1`);
3. 运行程序，观察是否有显著的性能提升。

但这种方式显然是不科学、不合理的，例如上面的修改以很大的概率不会对程序的性能有可见的影响。在这个实验中，写出更快的代码不是目的，而是帮助大家熟悉性能分析的工具，然后做出针对性的优化。正如 D. E. Knuth 所说，“Premature optimization is the root of all evil”。