现代 C++ 性能漫谈

吴咏炜

© 2022 吴咏炜

自我介绍

- 学编程超过 35 年
- 30 年 C++ 老兵
- 热爱 C++ 和开源技术
- 多次在 C++ 大会上推广 C++ 新特性
- 对精炼、跨平台的代码有特别偏好

© 2022 吴咏炜

$o(\ldots)$?

© 2022 吴咏炜

影响性能的架构因素

硬件

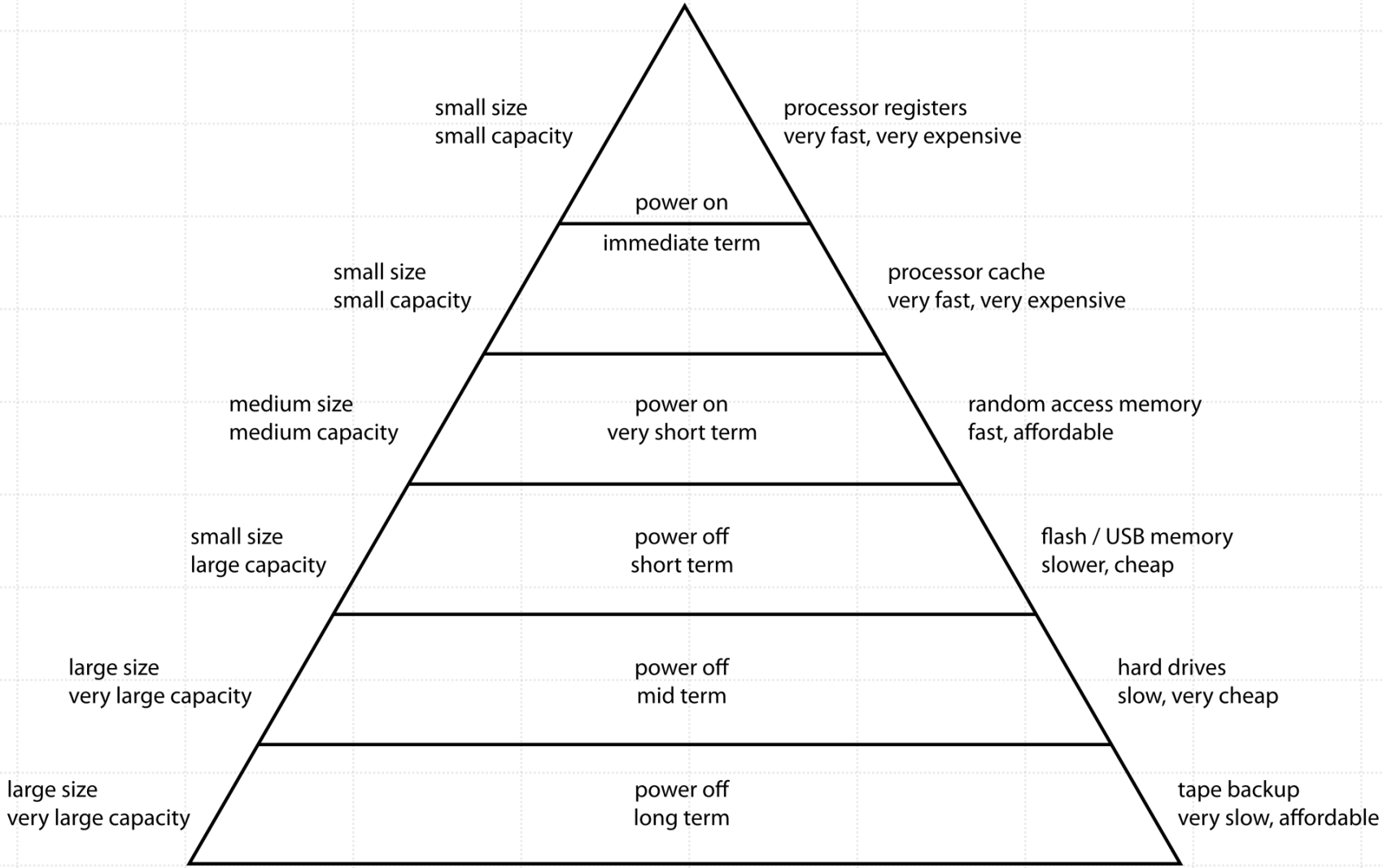
- 存储层次体系
- 处理器的乱序执行和流水线
- 并发

软件

- 系统调用开销
- 编译器优化
- 语言抽象性

© 2022 吴咏炜

Computer Memory Hierarchy



来源： https://en.wikipedia.org/wiki/Memory_hierarchy

© 2022 吴咏炜

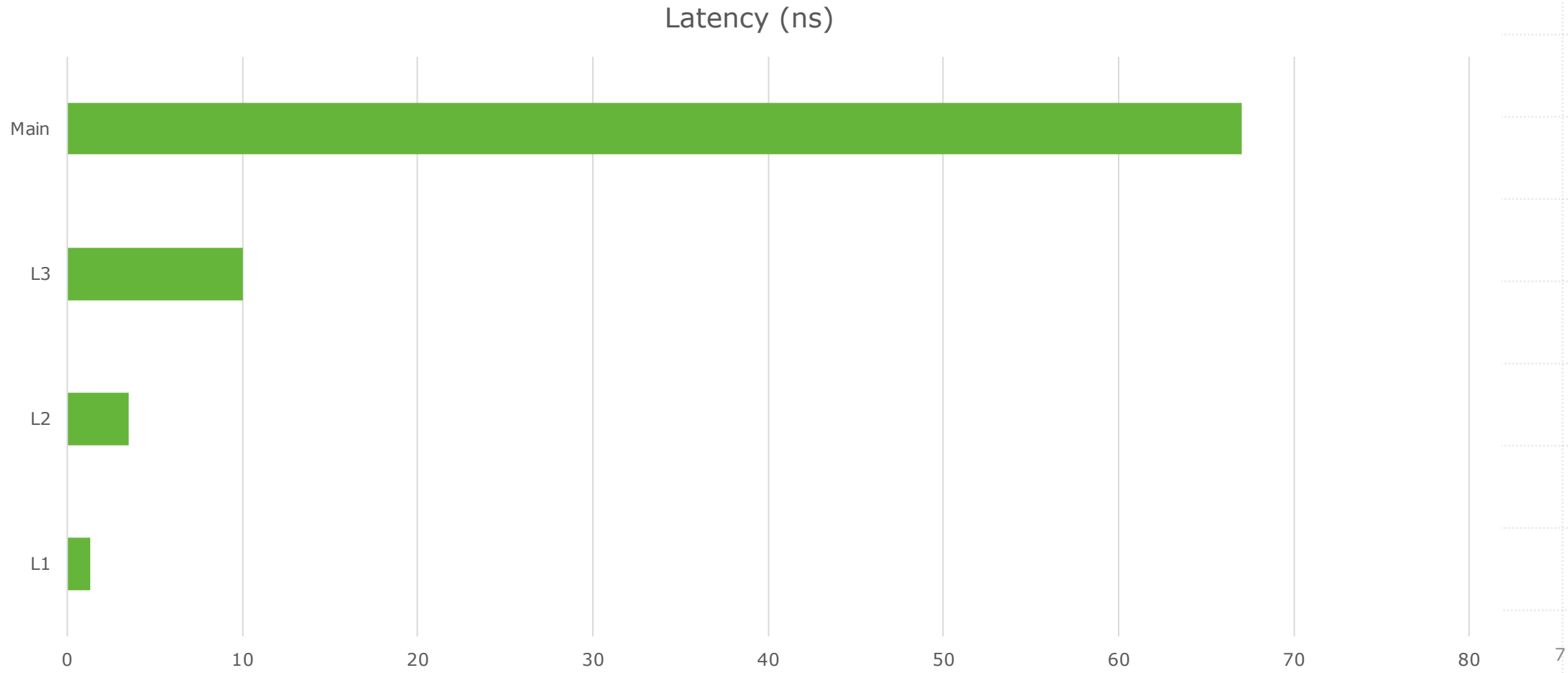
Intel Haswell i7-4770 的存储层次

名称	大小	延迟
寄存器	~1 KB	—
L0（微码）缓存	6 KB	—
L1 缓存	32 KB 数据 + 32 KB 指令	4–5 时钟周期（数据）
L2 缓存	256 KB	12 时钟周期（~4 纳秒）
L3 缓存（4 核共享）	8 MB	36 时钟周期（~10 纳秒）
主存	32 GB（最大；~$3/GB）	~67 纳秒
固态盘	256 GB（典型；~$0.2/GB）	~30 微秒
硬盘	2 TB（典型；~$0.05/GB）	~15 毫秒
磁带	15 TB（典型；~$0.01/GB）	无随机访问能力

* 假设主频为 3.4 GHz，关闭 Turbo Boost

© 2022 吴咏炜

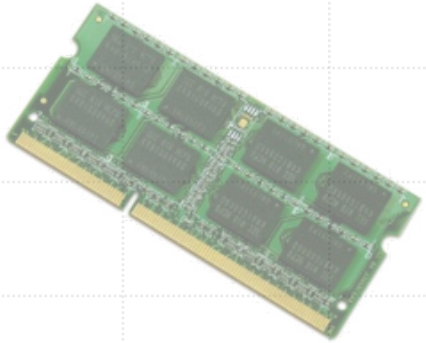
Intel Haswell i7-4770 的内存延迟

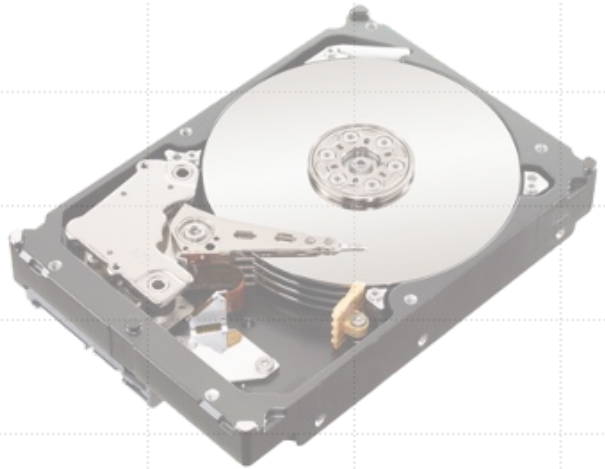


© 2022 吴咏炜

存储访问的基本原则——局域性

- 连续、不跳跃的存储访问最快





© 2022 吴咏炜

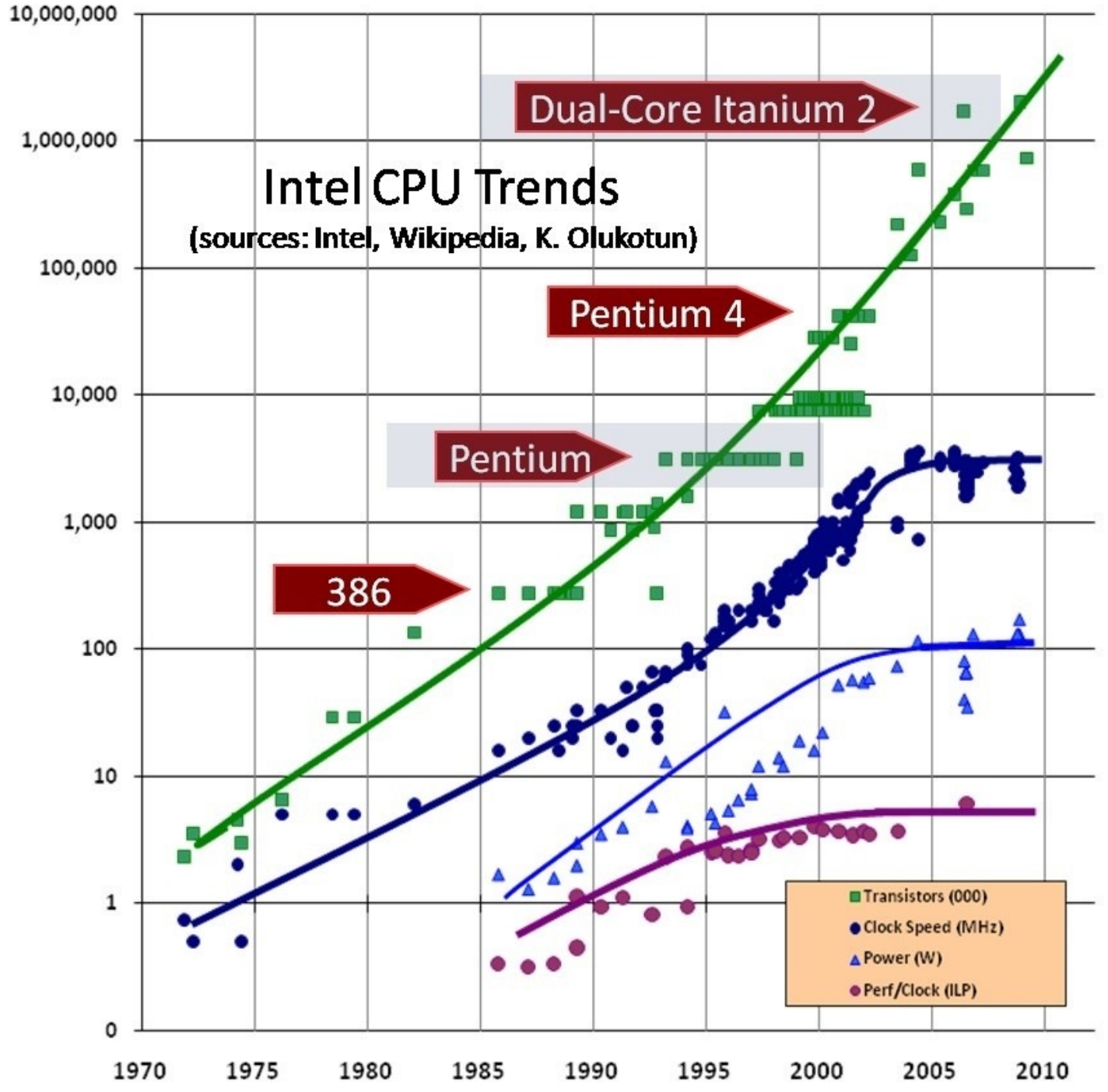
处理器的乱序执行和流水线

流水线和乱序执行提高每个时钟周期的处理能力

分支可能打乱流水线，造成性能下降

© 2022 吴咏炜

摩尔定律的局限性



© 2022 吴咏炜

并发对编程思维的冲击

- 不再能假设有“自然”的完全执行顺序
- 开发人员必须主动利用多核的特性
- 多线程的调度和竞争成为影响性能的关键因素
- 适用于单线程的接口可能不再适用

© 2022 吴咏炜

系统调用开销

- read
- write
- open
- close
- mmap
- gettimeofday
- …

© 2022 吴咏炜

编译器优化

- 可能产生巨大的性能差异
  - 圈复杂度小的代码更明显
- C++ 开启优化性能更明显
  - 标准库的性能依赖于打开优化
- 优化的“副作用”
  - 程序员常常没有意识到“未定义行为”的存在
  - 警告选项和静态扫描可以部分解决问题

© 2022 吴咏炜

编译器对硬件特性的照顾

C++

```
1   int x;
2   int y;
3   int a;
4
5   int main()
6   {
7       x = a;
8       y = 2;
9   }
10
```

x86-64 gcc 4.4.7 -O2

```
1   main:
2           mov     eax, DWORD PTR a[rip]
3           mov     DWORD PTR y[rip], 2
4           mov     DWORD PTR x[rip], eax
5           xor     eax, eax
6           ret
7   x:
8           .zero   4
9   y:
10          .zero   4
11  a:
12          .zero   4
```

https://godbolt.org/z/rbq1q9

© 2022 吴咏炜

语言抽象性

C

```
Obj obj;
```

- 在栈上分配了 `sizeof(Obj)` 字节，$O(1)$ 开销

C++

```
Obj obj;
```

- 在栈上分配了 `sizeof(Obj)` 字节，$O(1)$ 开销
- 调用 `Obj` **构造函数**
- 到达下面的 `}` 时调用**析构函数**

© 2022 吴咏炜

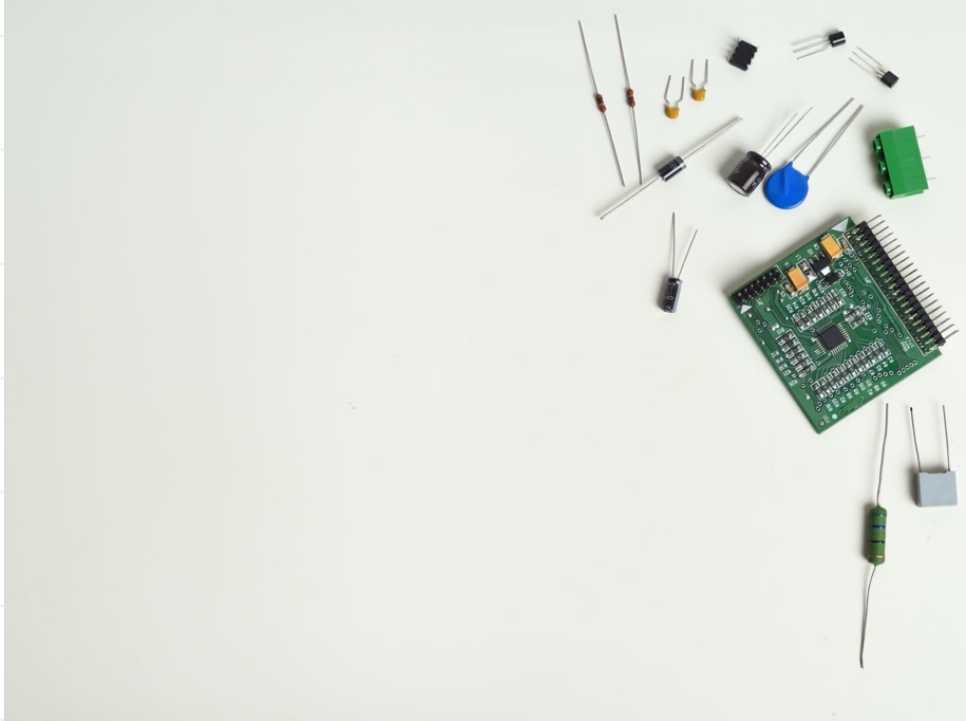
为什么要使用 C++？

贴近硬件

- 使用原生的指令和类型，高性能
- 方便使用新的硬件（包括 GPU、FPGA 等）

零开销抽象

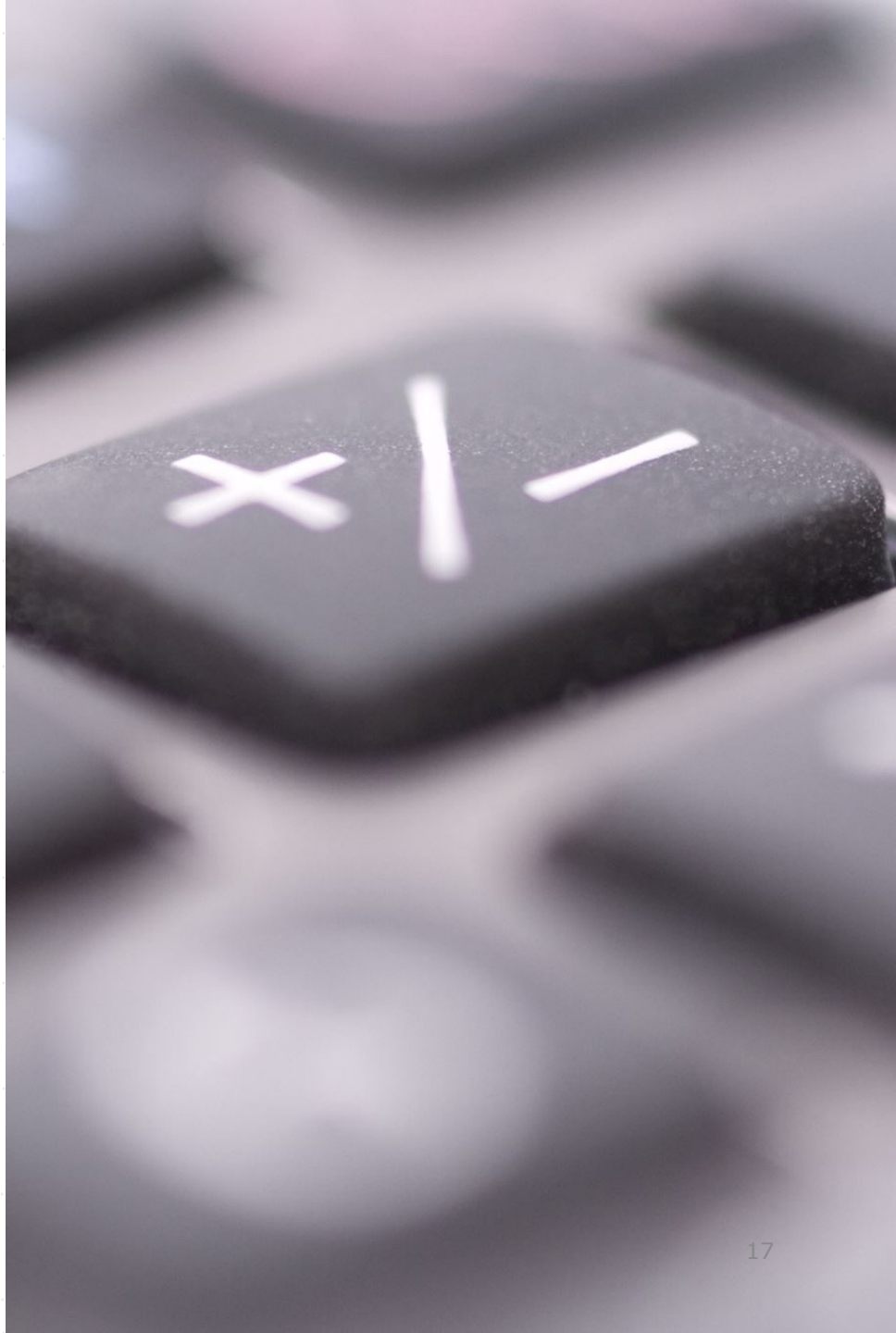
- 类、继承、模板、类型别名……
- 将来：完全的类型和资源安全，概念、模块、协程、契约、静态反射……



© 2022 吴咏炜

成功语言的烦恼

- C++ 太大、太复杂了，请简化！
- 我需要这两个关键功能，请尽快加上！！
- 不管语言怎么变，不要搞砸我的代码！！！



© 2022 吴咏炜

如何学习 C++

像学外语一样持之以恒

- 上手也许很快，真正掌握需要很久

掌握惯用法

- 语言的精髓不是语法，而是积累下来的惯用法

新语言，而不是“C 加上类”

- 学习《C++ 核心指南》（*C++ Core Guidelines*）

© 2022 吴咏炜

Bjarne 的洋葱原则

- 复杂性的管理
  - 简单事情简单做！
- 抽象层次
  - 切得越深，哭得越多……

© 2022 吴咏炜

Premature optimization is the root of all evil.

© 2022 吴咏炜

优化领域的阿姆达尔定律

$$S = \frac{1}{1 - P + \frac{P}{S_P}}$$

© 2022 吴咏炜

性能需要测试！

© 2022 吴咏炜

测不准问题……

© 2022 吴咏炜

小例子：你知道结果吗？

```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
  memset(buffer, 0, sizeof buffer);
}
auto t2 = clock();
printf(
  "%g\n",
  (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
  for (size_t j = 0; j < sizeof buffer;
       ++j) {
    buffer[j] = 0;
  }
}
auto t2 = clock();
printf(
  "%g\n",
  (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

© 2022 吴咏炜

GCC 8 的测试结果

编译选项	memset:手工循环（时间）
-O0	1:55
-O1	1:5
-O2	100000:1

© 2022 吴咏炜

原因：对 buffer 的写入被优化没了！

© 2022 吴咏炜

volatile？

© 2022 吴咏炜

GCC 8 的测试结果（volatile）

编译选项	memset:手工循环（时间）
-O0	1:25
-O1	1:5
-O2	1:5

© 2022 吴咏炜

volatile 本身会妨碍优化……

```
volatile char buffer[80];
…
for (size_t j = 0; j < sizeof buffer; ++j) {
    buffer[j] = 0;
}

; GCC 10 下可能产生的汇编（x86-64）
    xor     eax, eax
.L2:
    mov     BYTE PTR buffer[rax], 0
    add     rax, 1
    cmp     rax, 80
    jne     .L2
```

```
char buffer[80];
…
for (size_t j = 0; j < sizeof buffer; ++j) {
    buffer[j] = 0;
}

; GCC 10 下可能产生的汇编（x86-64）
    pxor    xmm0, xmm0
    movaps  XMMWORD PTR buffer[rip], xmm0
    movaps  XMMWORD PTR buffer[rip+16], xmm0
    movaps  XMMWORD PTR buffer[rip+32], xmm0
    movaps  XMMWORD PTR buffer[rip+48], xmm0
    movaps  XMMWORD PTR buffer[rip+64], xmm0
```

https://godbolt.org/z/vnbEaT

© 2022 吴咏炜

防优化技巧

- 谨慎使用 `volatile`
  - 可防止编译器重排序（不防止处理器重排序）
  - 可能阻止应有的优化
- 使用全局变量
  - 一定会写入（仍有乱序问题和对重复写的优化）
- 使用锁来当作简单的内存屏障
  - 可靠（C++ 内存模型保证），但时间开销较大
- 可使用 `__attribute__((noinline))` 来防止意外内联

© 2022 吴咏炜

Linux 的时钟函数和某次测试结果

函数	精度（微秒）	耗时（时钟周期）
clock	1	~1800
gettimeofday	1	~69
clock_gettime	0.0265(38)	~67
std::chrono::system_clock	0.0274(38)	~68
std::chrono::steady_clock	0.0272(28)	~68
std::chrono::high_resolution_clock	0.0275(20)	~69
rdtsc	0.00965(48)	~24

© 2022 吴咏炜

函数调用和虚函数调用的额外开销

```
0 count_space:
  Call count: 10000
  Call duration: 1360812
  Average duration: 136.081
1 count_space_noinline:
  Call count: 10000
  Call duration: 2540532
  Average duration: 254.053
2 count_space_virtual:
  Call count: 10000
  Call duration: 3260456
  Average duration: 326.046
```

每次函数调用的开销：

$$\frac{254-136}{47} \approx 2.5$$

每次虚函数调用的开销：

$$\frac{326-136}{47} \approx 4.0$$

普通函数和虚函数开销差异本身不大……

© 2022 吴咏炜

两种性能测试方式

采样测试

- 总体开销可控
- 一般不影响程序“热点”
- 基于统计，误差较大
- 适合用来寻找程序的热点
- 可以完全在程序外部进行测试

插桩测试

- 开销随测试范围而变
- 插桩本身可能影响测试结果
- 测试结果可以较为精确、稳定
- 适合对单个函数进行性能调优
- 需要修改源代码或构建过程

© 2022 吴咏炜

性能分析和性能优化

- 90/10 规律
- 生产率 vs 性能
- 过早优化是万恶之源（在 97% 的情况下☺）

© 2022 吴咏炜

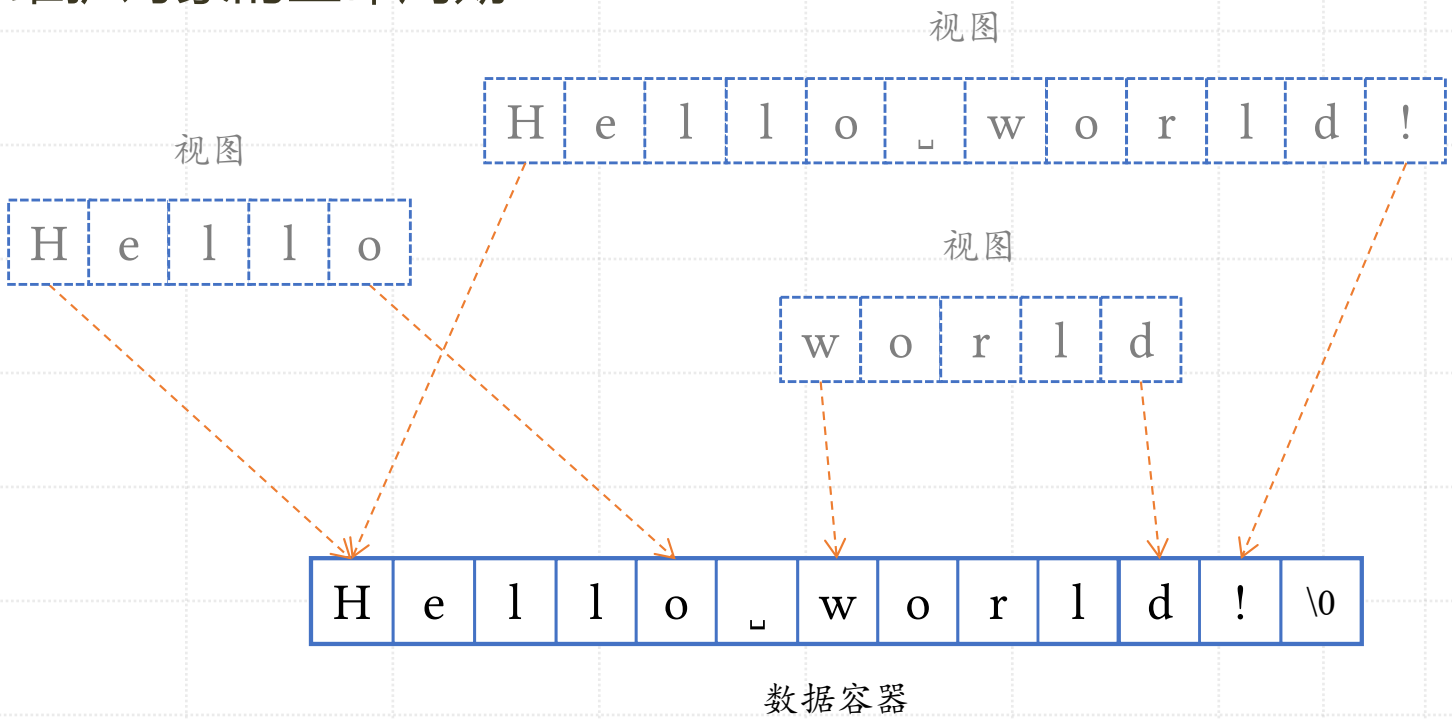
直接和性能相关的 C++ 特性

- 内联
- 移动语义
- 值语义
- 值容器
- 模板
- 视图类型
- 内存模型
- 原子量
- 并行和并发
- 编译期计算
- ……

© 2022 吴咏炜

视图类型

- 不拥有指向的资源，需要在视图外部维护对象的生命周期
- 轻量，常常只是指针加长度
  - 比使用指针更安全，不容易出错
- 可以以 $O(1)$ 开销进行复制
- 常见视图类型
  - `std::string_view`
  - `std::span` 或 `gsl::span`
  - `std::ranges::views` 下面的视图



© 2022 吴咏炜

GCC 编译选项 – 优化级别

- `-O0`（默认）：不开启优化，方便功能调试
- `-Og`：方便调试的优化选项（比 `-O1` 更保守）
- `-O1`：保守的优化选项
  - GCC 11 上打开了 47 个优化选项
- `-Os`：产生小代码的优化选项（比 `-O2` 更保守，并往产生较小代码的方向优化）
- `-O2`：常用的发布优化选项（对错误编码容忍度低）
  - GCC 11 上比 `-O1` 额外打开 51 个优化选项
  - 包括自动内联函数和严格别名规则
- `-O3`：较为激进的优化选项（对错误编码容忍度最低）
  - GCC 11 上比 `-O2` 额外打开 13 个优化选项
- `-Ofast`：打开可导致不符合 IEEE 浮点数等标准的性能优化选项

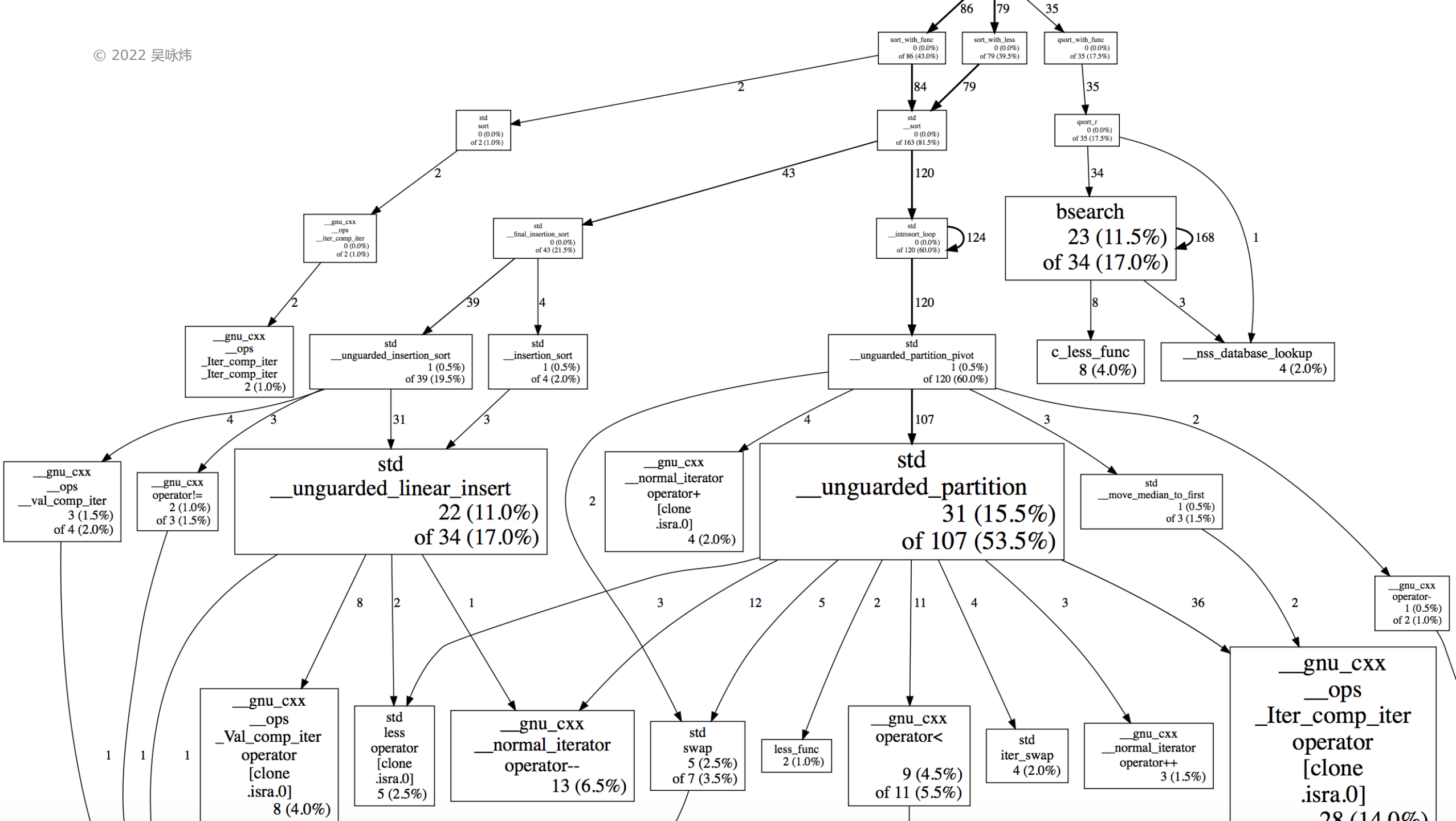
© 2022 吴咏炜

优化和内联的影响

	-O0	-O2 -fno-inline	-O2	-O2 对 -O0 的提升
sort + 函数对象	340981(1717)	197830(675)	24414(333)	14.0x
sort + 普通函数	334601(1843)	209241(609)	46384(220)	7.21x
qsort + 普通函数	133801(1814)	87014(790)	85323(535)	1.57x

单单内联即可产生一个数量级的性能差异！

© 2022 吴咏炜


sort_with_func
0 (0.0%)
of 86 (43.0%)
sort_with_less
0 (0.0%)
of 79 (39.5%)
qsort_with_func
0 (0.0%)
of 35 (17.5%)
std
sort
0 (0.0%)
of 2 (1.0%)
std
__sort
0 (0.0%)
of 163 (81.5%)
qsort_r
0 (0.0%)
of 35 (17.5%)
__gnu_cxx
__ops
__iter_comp_iter
0 (0.0%)
of 2 (1.0%)
std
__final_insertion_sort
0 (0.0%)
of 43 (21.5%)
std
__introsort_loop
0 (0.0%)
of 120 (60.0%)
bsearch
23 (11.5%)
of 34 (17.0%)
__gnu_cxx
__ops
_Iter_comp_iter
_Iter_comp_iter
2 (1.0%)
std
__unguarded_insertion_sort
1 (0.5%)
of 39 (19.5%)
std
__insertion_sort
1 (0.5%)
of 4 (2.0%)
std
__unguarded_partition_pivot
1 (0.5%)
of 120 (60.0%)
c_less_func
8 (4.0%)
__nss_database_lookup
4 (2.0%)
__gnu_cxx
__ops
__val_comp_iter
3 (1.5%)
of 4 (2.0%)
__gnu_cxx
operator!=
2 (1.0%)
of 3 (1.5%)
std
__unguarded_linear_insert
22 (11.0%)
of 34 (17.0%)
__gnu_cxx
__normal_iterator
operator+
[clone
.isra.0]
4 (2.0%)
std
__unguarded_partition
31 (15.5%)
of 107 (53.5%)
std
__move_median_to_first
1 (0.5%)
of 3 (1.5%)
__gnu_cxx
operator-
1 (0.5%)
of 2 (1.0%)
__gnu_cxx
__ops
_Val_comp_iter
operator
[clone
.isra.0]
8 (4.0%)
std
less
operator
[clone
.isra.0]
5 (2.5%)
__gnu_cxx
__normal_iterator
operator--
13 (6.5%)
std
swap
5 (2.5%)
of 7 (3.5%)
less_func
2 (1.0%)
__gnu_cxx
operator<
9 (4.5%)
of 11 (5.5%)
std
iter_swap
4 (2.0%)
__gnu_cxx
__normal_iterator
operator++
3 (1.5%)
__gnu_cxx
__ops
_Iter_comp_iter
operator
[clone
.isra.0]
28 (14.0%)

© 2022 吴咏炜

问题：下面哪个容器比较快？

- `vector<pair<Obj, size_t>>`（保持排序）
- `map<Obj, size_t>`
- `unordered_map<Obj, size_t>`

It depends …

© 2022 吴咏炜

容器选择的考虑因素

- 容器最常用的操作是什么（应用场景问题），它们的算法复杂度（容器）
  - 插入：$O(n)$，$O(\log(n))$，$O(1)$
  - 查询：$O(\log(n))$，$O(\log(n))$，$O(1)$
- Obj 的大小（移动的最小开销）
- Obj 的可移动性（额外的算法复杂度）
- 元素数量（缓存利用、内存分配器开销）
  - 当 Obj 较小时，vector 直到元素数量 1000 左右时仍可能具有全方位的性能优势！

测试（泛型的易用性优势）！

© 2022 吴咏炜

一些通用的优化方法

- 内存优化
- 循环优化
- 算数表达式优化
- 输入输出优化
- 算法优化
- ……

© 2022 吴炳炜

问题和讨论