从编译器以及系统结构的角度谈优化

关于性能优化，编译器和CPU一直在给我们负重前行，几行简简单单的代码背后可能是一套套令人拍案叫绝的理论和工程。

学习它们能深刻加强我们对计算机体系结构的理解，但并不是简单一篇文章就能揭开出所有的面纱。所以这篇文章可能更“功利”些，虽然会以系统结构、编译理论的视角出发，但只局限于那些可能对上层开发者而言不那么透明、需要关注的一些要点。

栈区内存的偏见

实用度：★★★★★，日常开发应该养成习惯

有这么三段代码：

```
void foo1 () {
        thread_local char buf[BUF_SIZE]; // BUF_SIZE <= 4KB
        // do something with buf
}

void foo2 () {
        static char buf[BUF_SIZE]; // BUF_SIZE <= 4KB
        // do something with buf
}

void foo3 () {
        char buf[BUF_SIZE]; // BUF_SIZE <= 4KB
        // do something with buf
}
```

你觉得哪段代码更优?

我估计有些同学脱口而出：多线程场景下foo1，单线程场景下foo2。至于原因就是foo3每次调用都要分配一大块内存，而foo2用static将这块缓冲区存放在静态区减少内存分配，如果是多线程场景foo1的thread_local又可以保证线程安全。

但事实并非如此，foo3**在栈上分配内存的代码反而是更优的。**

从学计算机开始，很多人包括我自己一直被告知malloc堆区分配内存是很昂贵的，从而打上思想钢印，认为所有的分配内存操作都昂贵，包括栈内存。今天我们得打破这个偏见。

必须得先明确一点，**堆是操作系统级别的概念，而栈是cpu级别的。**

cpu会提供两个寄存器sp和bp，bp和sp分别存取栈底和栈顶的地址。栈是从高地址向低地址增长的，所以减小sp代表增大栈空间，增大sp代表减少栈空间。

而对于一个函数，该函数需要多少栈空间是编译期间决定的，不会对运行时造成任何开销。你需要栈空间的大小，仅仅是改变了sub sp分配栈空间时操作数的变化而已。

比如对于c++代码:

```
// 避免编译器做过多优化

```

```
3  void nothing() {}
4
5  void foo() {
6      uint64_t c = 0;
7      nothing();
8  }
9
10 void bar() {
11     uint64_t c = 0;
12     char buf[1024];
13     nothing();
14 }
```

g++ xxx.cc -S -O0

编译得到的汇编(节选):

```
1  foo:
2          pushq   %rbp
3          movq    %rsp, %rbp
4          subq    $16, %rsp # 开辟栈空间
5          movq    $0, -8(%rbp)
6          call    nothing
7          nop
8          leave
9          ret
10
11 bar:
12         pushq   %rbp
13         movq    %rsp, %rbp
14         subq    $1040, %rsp # 开辟栈空间
15         movq    $0, -8(%rbp)
16         call    nothing
17         nop
18         leave
19         ret
```

foo、bar虽然开辟的栈空间不同，但指令条数没有任何变化，仅仅只是第4和第14行指令操作数的变化而已。栈内存分配不像堆区内存分配器一样需要考虑线程安全、内存碎片、元数据以及块管理等等复杂的问题，仅仅是改变一个sp指针而已。

x64下linux默认给线程分配的栈是8MB，如果在你能保证不爆栈的情况下，请打破偏见，**务必优先在栈上分配内存空间，它是零开销的。**

函数内联

实用度：★★★★★，日常开发应该养成习惯

函数调用开销

有这么一段代码:

```
1  int add(int a, int b) {return a + b;}   // callee
2
```

```
3 | add(1,2);       // caller
```

实现一个函数调用需要考虑哪些问题：1. 参数该怎么传。 2. 函数调用完后CPU怎么从callee返回到caller的上下文。 3. 如何保证caller上下文的寄存器不被callee覆盖。

我们一一解答：

参数传递

函数被调用者callee和调用者caller处于同一个CPU，它们**共享寄存器，也因此callee和caller也共享同一个栈内存空间。**

所以**寄存器**和**内存栈**都可以作为函数传参的方式，比如x86下主流的c语言编译器都采用栈穿参，x64下因为通用寄存器更多，为了规避额外写内存开销，这时会优先使用寄存器传参。

不同的语言、不同的编译器甚至不同的硬件平台，都可能导致不同的ABI调用规则，我们这里不过多讨论细节，只需知道，如果想要传参，要么写内存、要么写寄存器，这个准备数据的过程是必须支付的开销。

比如上面第3行调用add函数：

```
1 | // 调用add前需要设置对应寄存器
2 | movl $2, %esi
3 | movl $1, %edi
```

函数如何返回

作为一个callee，如果没有一些措施，它无法知道是哪些caller调用自己，更无从知道自己逻辑执行完后怎么再return到caller的上下文。

解决方案是需要caller主动告知callee。caller每次在调用callee前，需要将自己当前指令的地址压入栈中(这个地址叫做return address)，然后再jump到callee去，callee返回时再jump到栈上保存的return address即可。

caller的操作可以总结为push return address + jump两条指令，CPU专门用一条call指令完成这个过程：

```
1 | movl $2, %esi
2 | movl $1, %edi
3 | call add
```

还有一点注意的是，jump永远是对cache不友好的。

ok，目前为止咱们又多了条call指令的开销。

如何保证caller上下文的寄存器不被callee覆盖。

前面提到caller和callee共享寄存器，假设caller在某些寄存器存取了数据，这时执行call指令跳转到callee的代码，callee正好用到了这些寄存器，就会导致caller在寄存器中的值被覆盖，如下面代码：

```
1 | callee:
2 |         movl $3, %eax           # overwrite eax
3 |         movl $2, %edx
4 |         addl %edx, %eax
5 |         ret
6 |
7 | caller:
8 |         movl $1, %eax
```

```
 9          call callee
10          # %eax is not 1 now!
```

为了避免这种情况的发生，很简单的方式就是caller在调用callee之前，将寄存器压栈保存即可，调用完后再退栈。

那么改写的情况应该如此：

```
caller:
        movl $1, %eax
        pushl %eax      # 压栈
        call callee
        popl  %eax      # 出栈
```

当然实际情况下为了性能考虑，并不是让caller一个人保存所有寄存器，而是由caller和callee各自负责一些。

比如对于callee foo：

```
long foo() {
    register long a = 0;
    return a;
}
```

生成的汇编代码(不要开启编译优化)：

```
foo:
        pushq %rbp
        movq  %rsp, %rbp
        pushq %rbx              # 保存rbx
        movl  $0, %ebx
        movq  %rbx, %rax
        popq  %rbx              # 恢复rbx
        popq  %rbp
        ret
```

因为callee要使用rbx寄存器，那么根据ABI规则，它就有义务去保存rbx旧值。在使用完寄存器后再恢复旧值。

具体是由谁负责保存哪些寄存器，也是ABI规则的一部分，但不管如何都可能引入额外的**寄存器压栈访存开销。**

综上所属，我们已经列举了三种开销了，所有有没有方式避免呢？ **函数内联**！

内联原理

函数内联的本质就是代码复制拷贝。它会直接将callee的代码内容复制到caller的上下文中。

比如这样的代码：

```
int add(int a, int b) {return a + b;}
int caller() {return add(1, 2);}
```

caller在将add内联后会变成这样：

```
int caller() {return 1 + 2;}
```

caller将调用add过程替换成了add内部的代码。

当然内联做的事情不会仅仅这么简单，咱们后面再谈～

benchmark

前面只是理论上定性分析的，来个定量的benchmark：

对于一个简单的add方法，内联和非内联版本：

```
1  ------------------------------------------------------
2  Benchmark               Time            CPU             Iterations
3  ------------------------------------------------------
4  BM_InlineAdd    0.400 ns        0.400 ns        1000000000
5  BM_NoInlineAdd  1.99 ns         1.99 ns         347818890
```

相差5倍，这还是因为函数过于简单没有额外寄存器压栈开销的情况。

内联带来的额外收益

你以为函数内联就只是省略掉函数调用开销吗？ 不仅仅如此！

继续举个例子，防御性编程一直是我们推荐鼓励的，因此作为一个**库开发者**，我可能实现了这么一个函数：

```
1  void add_ptr(int* a, int* b) {if (a && b) *a = *a + *b;}
```

使用前对指针判空是个好习惯。正好此时存在一个库的**使用者**，想使用这个函数：

```
1  int* ptr1 = get_from_somewhere();
2  int* ptr2 = get_from_somewhere();
3  if (ptr1 && ptr2) add_ptr(ptr1, ptr2);
```

库的使用者在不确定库源码是否对空指针做防御性处理的情况下，在第3行也对ptr1和ptr2做了防御性检查，这样也很好，代码再怎么小心也不为过。

但问题是，这是以性能作为代价的！我们**对ptr1和ptr2的检查做了两次**！

重点来了：函数内联可以在不改代码逻辑的情况下，帮助你省略掉这个过程，它会先将上面代码优化成这样：

```
1  if (ptr1 && ptr2) {
2      if (ptr1 && ptr2) *ptr1 = *ptr1 + *ptr2;
3  }
```

最后编译器发现了重复的代码，然后优化成这样：

```
1  if (ptr1 && ptr2) *ptr1 = *ptr1 + *ptr2;
```

到最后，皆大欢喜，库使用者放心、开发者放心，性能还没任何损耗！

所以最后来个内联收益的总结：

- 省略函数调用开销。
- 将函数的代码拷贝到caller上下文，从而编译器能做更多优化。

QA

内联有没有坏处?

当然有，内联的原理是复制拷贝，内联前callee的代码仅仅只有一份，任何caller需要调用就jump过来即可。而内联后是在任何调用者处对函数代码做了拷贝，因此可能会导致生成的二进制更大。可以理解为空间换时间。

什么情况下内联函数?

函数本身逻辑比较简单，且被大量调用时，这时候就应该选择让其内联。

如何将函数内联?

c和c++具有inline关键字修饰函数，但它只是对编译器的**建议值**。

编译器有自己的一套规则(函数是否复杂比如是否有for循环等)衡量是否将函数内联，也就意味着编译器可能给未inline的函数内联，也可能给修饰了inline的函数不内联。当然不同编译器有不同接口让开发者对函数强制内联或者不内联，读者可自行了解。所以一般而言我们不需要刻意用inline修饰函数。

但有一点确定的是，如果你将一个函数实现在1.c文件，另外一个2.c文件链接函数这种情况肯定是无法内联的。因为函数内联的原理是将函数代码拷贝到调用者上下文，如果2.c都无法看到函数对应的代码，更不谈内联了。

所以如果想内联函数，c语言做法是将函数的实现放在header文件中，并使用static inline修饰函数。对于c++就无需如此，在**头文件类内实现的方法**默认就是static inline的，所以只需把类的方法实现在头文件的类内即可。

有些时候我并不想在头文件暴露函数具体实现细节，但依旧想消除上面防御性检查的double check损耗，怎么办呢?

一种可行的方式是，把防御性检查代码放在头文件，实现细节依旧放在源码文件中：

比如add_ptr.h：

```
class XXX {
 public:
  void add_ptr(int* a, int* b) {if (a && b) add_ptr_detail(a, b);}

 private:
  void add_ptr_detail(int*a, int* b);
};
```

add_ptr.cc:

```
void XXX::add_ptr_detail(int* a, int* b) {*a = *a + *b;}
```

这样，就算库用户自身也做了防御性检查，但因为对add_ptr的内联，编译器可以省略掉不必要的步骤。

内存对齐

实用度：★★★★★，日常开发应该养成习惯

一个忠告：请仔细编排类和结构体成员的顺序!

什么是内存对齐

假设要实现一个结构，这个结构需要存取8个KV对，key为char类型，val为uint64_t类型，有两种实现方式：

```
// sizeof(BucketWithoutPadding) = 72
struct BucketWithoutPadding {
  char keys[8];
  uint64_t vals[8];
};

// sizeof(BucketWithPadding) = 128
struct BucketWithPaddding {
  struct {
      char key;
      uint64_t val;
  } pairs[8];
};
```

会发现，一个占用72字节一个占用128字节，两者之间占用内存大小天差地别，这就是**内存对齐**导致的副作用！

编译器倾向于将每个结构体成员的地址放在其大小的整数倍上，如uint64_t存取的地址就是8*n*，*uint32_t存取的地址是4*n。这就意味着编译器会隐式在结构体内部插入一些padding，BucketWithPadding会被编译器改写为如下：

```
struct BucketWithPadding {
  struct {
      char key;
      char padding[7];
      uint64_t val;
  } pairs[8];
};
```

为什么需要内存对齐

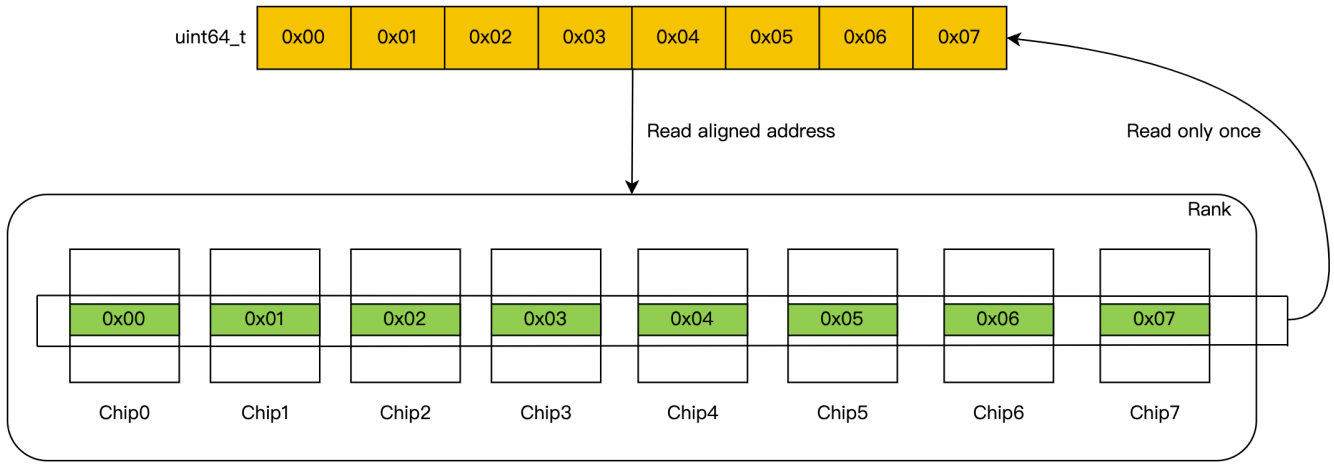
编译器为什么这么做？

第一，有些精简指令集架构的CPU load和store的地址就ni要求是内存地址对齐的，这是物理限制。

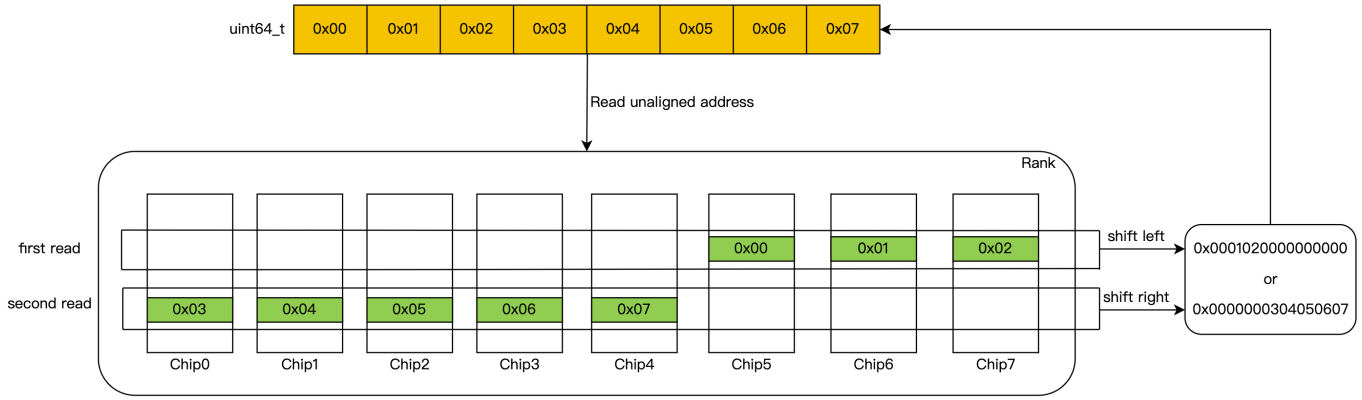
第二出于性能考虑。这点我们细讲下。

在理解之前需要一个前置知识：我们一个内存Rank是由多个内存颗粒Chip组成的，在物理地址上连续的8个字节并不是按照顺序的方式全部存在某一个Chip上，而是将这8B分成8份，分别存在8个Chip上。

如果正好我们要访存的8B是内存对齐的，那么这每个字节就处在8个Chip的同行同列，CPU可以并行的一次将数据读出，如下图：



而在非内存对齐的情况，需要两次内存IO，然后对前后两次读取做结果拼接:



非内存对齐会导致两次内存IO！

> *当然这只是理论上，实际上非内存对齐的访问不会有这么大的惩罚，因为CPU只有Cache有脏数据需要刷内存或者出现Read miss时，CPU才会访存，且访存的最小单位是Cache行的大小(一般64B)，这样已经保证了CPU送上物理地址总线上的永远是内存对齐的地址。只有当访问的非对齐内存的数据跨了Cache line的边界，比如前部分属于上一个Cache line，后一个部分属于另一个Cache line，并且都发生Cache miss时才会发生两次内存IO。而正好编译器和运行时内存分配器都会尽最大努力保证你所访问的内存数据不会跨Cache line，所以**非内存对齐访问的开销远比你想的小。***

虽然我们可以通过一些特殊指令让编译器不对结构体做对齐，但一般不推荐这么做，因为内存对齐的益处上文也讲到了。

我们能做的是尽量是减少内存对齐导致的空间放大负面影响，比如:

```
struct Foo1 {
  bool a;
  uint64_t b;
  bool c;
};

struct Foo2 {
  bool a;
  bool c;
  uint64_t b;
};
```

Foo1需要24字节，而Foo2只需要16字节，显然Foo2更优。

内存布局是门学问，后面讲的False Sharing会进一步讲这个问题。

False Sharing-并行编程的陷阱

实用度：★★★★★，日常开发应该养成习惯

前面内存对齐那一节简要提了一嘴False Sharing，在这里我们进一步探讨内存布局对于性能的影响。

从例子出发

给你一段这样的伪代码：

```
const int thread_num = 4;
int counters[thread_num] = {0};

for i in (0..thread_num) {
  create_thread {
    do {++counters[i]} until (counters[i] == 1000000);
  }
}

wait_for_all_threads_exit();
print_used_time();
```

假设机器的物理核数必定>=thread_num，也即满足任何一个线程能不发生上下文切换独占一个物理CPU。如果不断的增加thread_num，整个过程的执行耗时会不会有明显变化？

让我们简单分析下。线程之间存在资源共享吗？不存在，每个线程独享对应的counter变量。

所以你很自信地说：执行耗时不会变化，我们的系统能够以极为理想的方式线性拓展。

梦想是美好的，但现实很骨感，咱们实际写个benchmark：

```
const uint64_t limit = 100000;

template <size_t ThreadNum>
void BenchmarkFalseSharing(benchmark::State& state) {
  for (auto _ : state) {
    std::vector<std::thread> threads;
    threads.reserve(ThreadNum);
    __attribute__((aligned(64))) uint64_t nums[ThreadNum] = {0};
    for (size_t i = 0; i < ThreadNum; ++i) {
      // volatile防止编译器把while循环优化为直接给num赋值成limit
      volatile uint64_t& num = nums[i];
      threads.emplace_back([&] () {
        while(++num != limit) ;
      });
     }
    for (auto& t : threads) t.join();
  }
}

static void BM_FalseSharingThread1(benchmark::State& state) {
  BenchmarkFalseSharing<1>(state);
```

```
}
BENCHMARK(BM_FalseSharingThread1);
static void BM_FalseSharingThread2(benchmark::State& state) {
  BenchmarkFalseSharing<2>(state);
}
BENCHMARK(BM_FalseSharingThread2);
static void BM_FalseSharingThread4(benchmark::State& state) {
  BenchmarkFalseSharing<4>(state);
}
BENCHMARK(BM_FalseSharingThread4);
static void BM_FalseSharingThread8(benchmark::State& state) {
  BenchmarkFalseSharing<8>(state);
}
BENCHMARK(BM_FalseSharingThread8);
```

分别测试了1、2、4、8个线程下的结果：

```
-----------------------------------------------------------
Benchmark                               Time            CPU             Iterations
-----------------------------------------------------------
BM_FalseSharingThread1  367638 ns       34799 ns        10000
BM_FalseSharingThread2  1421570 ns      65114 ns        10331
BM_FalseSharingThread4  3785955 ns      112065 ns       1000
BM_FalseSharingThread8  6849201 ns      301093 ns       1000
```

结果大跌眼镜，随着线程个数的增多，系统的性能反而看不到任何提升，甚至出现下降的情况。

系好安全带，马上发车揭晓答案。

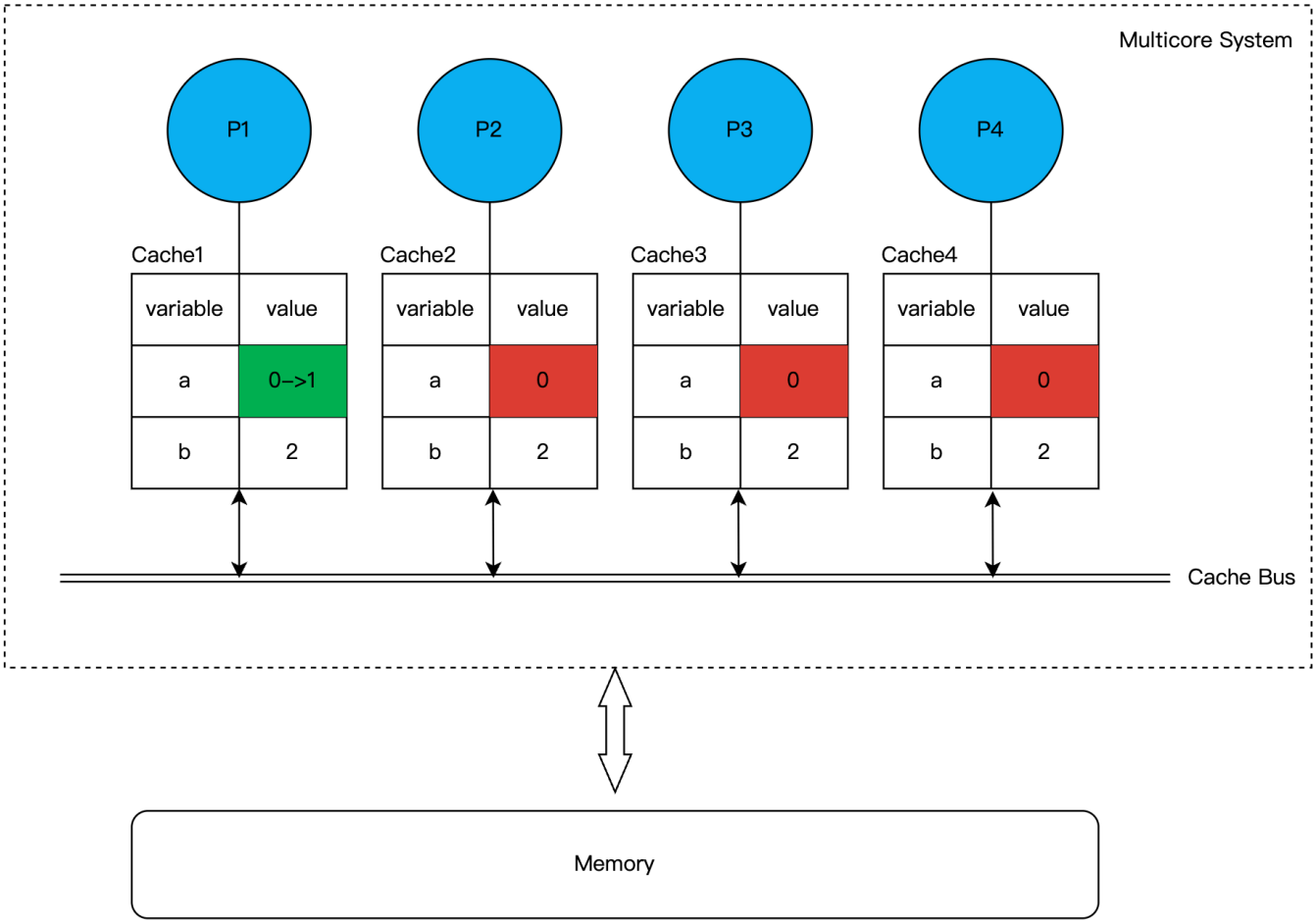
什么是False Sharing

我们还是从一个美好的偏见开始：这么多年的计算机教育告诉我们，Cache是个好东西，它很快，能解决CPU与内存速率不匹配的问题，但它空间很小，所以我们得小心且努力地提升它的命中率，比如对数据做内存对齐、逻辑上连续访问的数据尽量排列在连续内存啊之类。

但我现在告诉你不合适的Cache命中反而会成为拖累性能的罪魁祸首，你会不会开始怀疑"物理学不存在了" 。

在进一步之前，咱们再铺垫点前置知识。在计算机体系中，CPU物理核心配有多级sram cache，我们简化问题，如果只看L1缓存的话，可以理解为**每个CPU核都具有一个独占的cache。**

要知道cpu都是**优先读写自己的cache**的，然后才是访存，那么必然存在一致性问题，比如下图中P1处理器将变量a从0改为1并写到了自己的cache，而没立马将数据刷回内存，这时候其他处理器读到就还是旧值0这个脏数据，这就出现了不一致的问题。



为了解决一致性问题，硬件工程师们就引入了Cache间通信协议来解决。其中运用最广泛的是基于Cache总线监听的MESI协议。从其出发可以牵扯出大量的问题，比如Memory order、内存屏障等，高效无锁数据结构的设计离不开这些理论基础，但我们不会深入很多，仅仅讲述本例需要的知识即可。

我们只需知道的是，根据MESI协议，如果P1更改了变量a，为了保证一致性，p1需要向Cache Bus上发送一个Invalid Message，保证变量a所处的Cache Line在其他处理器的Cache中失效。说着有点拗口，说人话就是，P1霸道地宣布了对a所处Cache Line的所有权，其他的处理器原先存的旧数据全部都得失效，你们不得访问。

那么其他处理器要再次访问变量a怎么办呢？没错，和你想的一样，它们需要向P1发送Read Miss消息提交申请，然后P1才负责将最新Cache line同步到其他处理器或者写到内存。通过这种设计就保证每个Cache就能看到最新写入的值啦。

显而易见，不同cpu之间频繁对同一个数据写会触发频繁的广播，导致CPU之间Cache Line频繁失效。这也是为什么shared_ptr的引用计数会导致性能下降的一个很大原因。

> *读者：你也说了是对共享数据写会有开销，但上面benchmark例子里每个线程更改的都是自身独占的变量，访问的也不是同一份数据，你这答非所问啊？*
>
> *我：我知道你很急，但请你别急。*

请抓住重点，**Cache之间沟通的最小单位是一个Cache Line大小，现代主流CPU都是64字节。**所以Cache之间失效的不会是单纯一个变量，而是整个变量所处的这个Cache行。

咱贴心的把代码再贴过来，你不用向上翻：

```
std::vector<std::thread> threads;
threads.reserve(8);
// 靓仔们请把目光聚集在nums这个数组上
```

```
__attribute__((aligned(64))) uint64_t nums[8] = {0};
for (size_t i = 0; i < 8; ++i) {
  volatile uint64_t& num = nums[i];
  threads.emplace_back([&] () {
    while(++num != limit) ;
  });
}
```

反应比较快的同学可能已经立马醒悟过来了。**虽然每个线程的确是访问nums中不同的uint64，但8个uint64却共享的是同一个Cache Line**！

*8 * sizeof(uint64_t)正好是一个Cache Line大小。*

因此任何一个线程对uint64数据的更改会影响到其他线程，从而阻碍系统得不到理想中的线性提升。

这种错误的内存排列现象就叫做**False Sharing。**

解决方案

什么，你还要我给解决方案？那就手动插入一些padding，将类型占用空间扩展到Cache Line大小呗！

一个简单示例，其中CacheLineHolder会插入64-sizeof(T)大小的padding保证不同的value处在不同Cache Line之中：

```
template <typename T>
struct CacheLineHolder {
  CacheLineHolder(): value() {}
  T value;
  char padding[64 - sizeof(T)];
};

static_assert(sizeof(CacheLineHolder<uint64_t>) == 64);

template <size_t ThreadNum>
void BenchmarkNoSharing(benchmark::State& state) {
  for (auto _ : state) {
    std::vector<std::thread> threads;
    threads.reserve(ThreadNum);
    __attribute__((aligned(64))) CacheLineHolder<uint64_t> nums[ThreadNum];
    for (size_t i = 0; i < ThreadNum; ++i) {
      // volatile防止编译器把while循环优化为直接给num赋值limit
      volatile uint64_t& num = nums[i].value;
      threads.emplace_back([&] () {
        while(++num != limit) ;
      });
    }
    for (auto& t : threads) t.join();
  }
}
```

直接给压测结果：

```
-----------------------------------------------------------------
Benchmark                               Time           CPU             Iterations
-----------------------------------------------------------------
BM_NoSharingThread1     367777 ns       34898 ns        10000
```

```
BM_FalseSharingThread1  367638 ns       34799 ns        10000
BM_NoSharingThread2     383127 ns       59513 ns        12331
BM_FalseSharingThread2  1421570 ns      65114 ns        10331
BM_NoSharingThread4     420252 ns       109613 ns       6189
BM_FalseSharingThread4  3785955 ns      112065 ns       1000
BM_NoSharingThread8     622051 ns       338591 ns       2136
BM_FalseSharingThread8  6849201 ns      301093 ns       1000
```

可以发现规避伪共享后才得到了理想的结果，8线程下性能可以相差10倍。

无分支编程

流水线

先来个灵魂拷问：对于n条排列的机器指令，cpu是对每条指令一个一个串行执行吗？

答案是否。

我们知道在冯诺依曼架构计算机下，指令和数据都存在内存中，对于一条指令的执行肯定是分很多过程的，以最经典的五级流水线为例，一条指令分为如下阶段：

1. **取址(IF)**：从内存中读取出指令。
2. **译码(ID)**：咱们得先知道取的是什么指令。
3. **执行(EX)。**
4. **访存(MEM)**：指令总有操作数嘛，访存获取它们或者将计算结果写入到内存。
5. **写回(WB)**：将执行结果写入到目标寄存器。

每个阶段可以简单理解为CPU的一个部件。

如果I1和I2指令之间完全**串行执行**：

Stage/Cycle	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
IF	I1	I2								
ID	I1	I2								
EX	I1	I2								
MEM	I1	I2								
WB	I1	I2								

你会发现这种情况下，在每个的时钟周期，CPU只有一个部件在运行，其他四个部件都在浑水摸鱼。

这可不行，生产队的驴都不敢这么歇，为了更好的压榨每个部件的，CPU引入了流水化技术。这时下一条指令可不会等上一条完全执行完才开始，CPU只要发现IF部件空闲就立马取下一条指令发射：

Stage/Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
IF	I1	I2	I3	I4	I5	I6				
ID	I1	I2	I3	I4	I5	I6				
EX	I1	I2	I3	I4	I5	I6				
MEM	I1	I2	I3	I4	I5	I6				
WB	I1	I2	I3	I4	I5	I6				

同样是10个时钟周期，串行执行模型却只能执行两条指令，而流水化后却能执行6条指令。通过流水化技术，虽然一个指令需要多个时钟周期，但系统的最理想IPC(每个时钟周期执行的指令)可以到达1！

读者：你讲的这个东西跟标题的branchless program有什么联系啊，作者标题党!

别急，后面有你急的:-)。

流水化的核心原理就是上一条指令立马发射，就接着取下一条指令。咱们的CPU也都以这种方式快乐地工作着。

但我说有没有可能上一条指令刚发射，CPU无法立马确定下一条该执行哪条指令呢？

接着就可以顺理成章的引出今天的主角**条件分支**。假设有如下代码：

```
bool cond;
if (cond) return 1; else return 0;
```

生成汇编(使用-O0关闭所有优化)：

```
  cmpb $0, -1(%rbp)
  je   .L4              # 如果cond=0，则跳转到.L4
  movl $1, %eax
  jmp  .L5
.L4:
  movl $0, %eax
.L5:
  ret
```

第1行cmpb指令就对应条件判断，第3行代表cond为true分支，第6行代表false分支。

在CPU发射了cmpb指令后，在这时CPU会面临一个问题，往后具有两个分支，一个是紧邻的代码，另一个是.L4分支，它该选择哪一个分支的指令做发射呢？对不起，CPU不知道，除非cmpb指令走到了EX执行周期得到了明确的if条件计算结果

那怎么办？最简单的方式那就是等呗(像极了我在等c++编译时摸鱼的样子)，让流水线停下来，等到有结果了再干活，大概要停顿IF取址周期到EX执行周期这么长的时间。

但可惜的是在现代动辄15级流水线往上走的处理器上，这个停顿的开销可不是能接受的。

分支预测

急，怎么办，在线等，有没有办法？当然有，那就是分支预测！

CPU当然可以依旧让流水线跑起来，那就是在条件判断结果出来前，先不管三七二十一随机选个分支的指令发射，猜对了人生巅峰，猜错了就当场短路、回厂重开。

当然上面是开个玩笑，但有一点是对的，CPU会提前选择一个分支的指令发射，选对了就无任何开销，选错了就得revert错误分支指令，然后重新走正确分支。

知名的武术学家马老师曾说过：他可不是乱打的啊，他是有备而来。

同理，具体选哪个分支，CPU也不是乱选一气，也是有所依据的！

什么依据呢？一句概括：**基于分支的历史信息。**

我估计有读者要暴起了：后文呢？你继续讲啊。

诶，咱们得打住话题，具体的分支预测实现细节可是"又臭又长"，咱们可不能偏题，你只需要知道**分支的条件越具有规律性，那么CPU对分支预测的越准确**就行了。

除此之外，还需要知道的一点就是，分支预测失败是具备一定惩罚的。

如下图所示，CPU在cmp执行到EX周期发现预测失败后，CPU则需要把错误分支上预取的指令全部清空(如I1和I2)，然后重新发射正确分支的M1, M2, M3指令：

Stage/Cycle	**2**	**2**	**3**	**4**	**5**	**6**
IF	cmp	I1	I2	M1	M2	M3
ID	cmp	I1	M1	M2		
EX	cmp(predict wrong)	M1				
MEM						
WB						

不仅仅是清空流水线这么简单，有些工作还需要undo，比如清除已经写入到write buffer的脏数据。

预测的成功率直接决定了分支代码的性能，如果你想查看项目的代码分支预测失败率，可以使用perf工具，最后输出的profile数据中就有对应的结果。

咱们依旧是写个benchmark定量测量预测失败对性能影响：

```
// 不允许内联，防止编译器看到过多上下文而做了我们不想要的优化
__attribute__ ((noinline))
std::vector<bool> GenConditions(size_t N, bool random) {
  srand(time(NULL));
  std::vector<bool> conds(N);
  for (size_t i = 0; i < N; ++i) conds[i] = random ? (rand() & 0x01) : true;
  return conds;
}

static const size_t N = 10000;
static void BM_WrongPrediction(benchmark::State& state) {
  auto conds = GenConditions(N, true); // 条件全是随机的
  for (auto _ : state) {
    uint64_t v1 = 0, v2 = 0;
    for (size_t i = 0; i < N; ++i) {
      if (conds[i]) v1 += 2; else v2 += 2;
```

```
    }
    benchmark::DoNotOptimize(v1); benchmark::DoNotOptimize(v2);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(N * state.iterations());
}
BENCHMARK(BM_WrongPrediction);

static void BM_CorrectPrediction(benchmark::State& state) {
  auto conds = GenConditions(N, false); // 条件是具有规律的
  for (auto _ : state) {
    uint64_t v1 = 0, v2 = 0;
    for (size_t i = 0; i < N; ++i) {
      if (conds[i]) v1 += 2; else v2 += 2;
    }
    benchmark::DoNotOptimize(v1); benchmark::DoNotOptimize(v2);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(N * state.iterations());
}
BENCHMARK(BM_CorrectPrediction);
```

一个样本的分支条件完全随机(CPU无法预测)，另一个则总是true(CPU预测总是成功)，结果：

```
-----------------------------------------------------------------
Benchmark                               Time            CPU             Iterations UserCounters...
-----------------------------------------------------------------
BM_WrongPrediction              31151 ns        31151 ns        10000 items_per_second=321.023M/s
BM_CorrectPrediction    8253 ns         8253 ns         10000 items_per_second=1.21165G/s
```

性能相差近4倍!

天呐，要是预测失败的惩罚这么高，你这说的我以后都不敢写条件判断了。

不必过分担心，再给个结论，其实在绝大多数场景下，CPU都能够极为精确的预测分支，基本上不需要我们关心太多。

*没错，敏锐的你已经发现我说的是**绝大多数**，本节讲的无分支编程就是去应对CPU处理不了的小case情况。*

Case 1

Case1讨论的场景：分支很难被CPU预测，走哪个分支完全是随机无规律的，比如上面benchmark中的WrongPrediction。

我们目前只是验证了分支预测失败带来的性能后果，但解决方案是什么呢?

核心思想：既然CPU猜不对，那么我就直接消除条件分支。

可以这样改进BM_WrongPrediction代码(两种方式)：

```
BM_WrongPredictionBranchless1:
  uint64_t v1 = 0, v2 = 0;
  for (size_t i = 0; i < N; ++i) {
    v1 += conds[i] * 2;
    v2 += (!conds[i]) * 2;
```

```
  }

BM_WrongPredictionBranchless2:
  uint64_t v1 = 0, v2 = 0;
  for (size_t i = 0; i < N; ++i) {
    const uint64_t d1[2] = {0, 2};
    const uint64_t d2[2] = {2, 0};
    v1 += d1[conds[i]]; v2 += d2[conds[i]];
  }
```

把条件当作运算操作符，是消除分支的核心思想。还有一种方式是条件move指令，但一般而言编译器会给我们优化，无需我们关心，这里不深谈。

结果如下：

```
-----------------------------------------------------------------
Benchmark                                       Time            CPU             Iterations UserCounters...
-----------------------------------------------------------------
BM_WrongPrediction                           29266 ns    29265 ns          23920 items_per_second=341.708M/s
BM_CorrectPrediction                       7982 ns     7982 ns          88066 items_per_second=1.25279G/s
BM_WrongPredictionBranchless1  19723 ns   19723 ns              35897 items_per_second=507.021M/s
BM_WrongPredictionBranchless2  12662 ns   12661 ns              54433 items_per_second=789.802M/s
```

第二种方式更优，这是因为乘法会慢一些，但都比使用条件判断的方式好。

归纳通用方法：

```
if (cond) variable operator exp1; else variable operator exp2;
=================================>
term[2] = {exp1, exp2};
variable operator term[cond];
```

但一定得记住，虽然能减少分支，但exp1和exp2都会被计算一遍，所以在使用此方法前，请确保额外计算exp的开销不大。

除此之外，你得在性能与可读性之间做一些权衡。

Case 2

上个Case可以说是跟CPU板手腕，这个Case就是跟编译器正面交锋。

在讲下去之前，我不知道你有没有看过这样的代码：

```
// a和b都是bool变量
if (a | b) do_something; else do_something_else();
```

我估计你的内心os：切，不就是个青铜菜鸟吗，连逻辑或运算符||都打成算术或运算符|。

但我想说的是，这个人可能是个王者。

诶，你别不信，看官先往下看。

假设有如下代码：

```
if (a || b) ++i; else --i;
```

我问你，如果a || b总是true，CPU能不能做好分支预测？

按照我们前面讲的，如果**一个分支的条件是具有规律的话，那CPU肯定能优秀地做到预测分支**，更何况条件还必定是true。所以你很快给出答案：是的，CPU一定能不负众望！

嘿嘿，上面加粗的话确实没啥问题，但很可惜的是，CPU还真不一定能做好这个工作，有些事情确实反直觉。

还是先压测，以数据说话：

```
__attribute__ ((noinline))
std::vector<bool> LogicNot(const std::vector<bool>& input) {
  std::vector<bool> output(input.size());
  for (int i = 0; i < output.size(); ++i) output[i] = !output[i];
  return output;
}

static void BM_ShortCircuit(benchmark::State& state) {
  // conds1随机的，conds2[k] = !conds1[k]
  auto conds1 = GenConditions(N, true);
  auto conds2 = LogicNot(conds1);
  for (auto _ : state) {
    uint64_t v1 = 0, v2 = 0;
    for (size_t i = 0; i < N; ++i) {
      // if分支必定成功
      if (conds1[i] || conds2[i]) v1 += 2; else v2 += 2;
    }
    benchmark::DoNotOptimize(v1); benchmark::DoNotOptimize(v2);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(N * state.iterations());
}
```

conds1[i]和conds2[i]中必定有一个是true，两者逻辑或肯定为true，满足我给的前置条件吧，揭晓下真实的结果：

```
-----------------------------------------------------------------
Benchmark                                       Time            CPU             Iterations UserCounters...
-----------------------------------------------------------------
BM_WrongPrediction                          29266 ns   29265 ns          23920 items_per_second=341.708M/s
BM_CorrectPrediction                     7982 ns    7982 ns          88066 items_per_second=1.25279G/s
BM_WrongPredictionBranchless1  19723 ns   19723 ns              35897 items_per_second=507.021M/s
BM_WrongPredictionBranchless2  12662 ns   12661 ns              54433 items_per_second=789.802M/s
BM_ShortCircuit                             49006 ns   49006 ns      14181 items_per_second=204.055M/s
```

明明分支必定成功，但性能依旧很低！(此刻我的脸上洋溢着得胜的笑容)

揭晓答案

这其实是个视角问题，在我们的开发者眼里里，上面的代码似乎仅有一个分支。但如果你真那么认为，那么你就已经掉入到编译器埋下的陷阱了，因为或语句是具有短路功能的，为了实现短路语义，编译器会将其编译为二个分支：

```
        cmpb    $0, -1(%rbp)    // if (a)
    jne .L4
    cmpb        $0, -2(r%rbp)   // if (b)
    je  .L5
 .L4:
    addl        $1, -8(%rbp)    // ++i
    jmp  .L6
```

```
.L5:                                            // else
   subl    $1, -8(%rbp)        // --i
.L6:
   ret
```

对a的判断和对b的判断是分别放在两个分支上的，然后在上面代码的实现里，a单独看是随机的，b单独看也是随机的，目前CPU做的分支预测只会基于某个分支自身的历史信息分析，而不会联合几个分支一起考虑（我不确定是否存在cpu会这样做），所以自然CPU就无法做好分支预测喽。

如何改进？

这种情况就不要使用逻辑或，而是使用算术或运算符，当然加法也ok，但略慢一点：

```
static void BM_NoShortCircuit(benchmark::State& state) {
  auto conds1 = GenConditions(N, true);
  auto conds2 = LogicNot(conds1);
  for (auto _ : state) {
    uint64_t v1 = 0, v2 = 0;
    for (size_t i = 0; i < N; ++i) {
      // 或运算
      if (conds1[i] | conds2[i]) v1 += 2; else v2 += 2;
    }
    benchmark::DoNotOptimize(v1); benchmark::DoNotOptimize(v2);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(N * state.iterations());
}
```

压测结果：

```
-----------------------------------------------------------------
Benchmark                                       Time            CPU             Iterations UserCounters...
-----------------------------------------------------------------
BM_WrongPrediction                         29266 ns   29265 ns          23920 items_per_second=341.708M/s
BM_CorrectPrediction                     7982 ns    7982 ns          88066 items_per_second=1.25279G/s
BM_WrongPredictionBranchless1  19723 ns   19723 ns              35897 items_per_second=507.021M/s
BM_WrongPredictionBranchless2  12662 ns   12661 ns              54433 items_per_second=789.802M/s
BM_ShortCircuit                             49006 ns   49006 ns      14181 items_per_second=204.055M/s
BM_NoShortCircuit                            8419 ns        8419 ns              84292 items_per_second=1.18783G/s
```

得到了近6倍的提升。

记得当年第一次了解到这点时，我失态大呼：“妖术，这是妖术！“，思绪起伏，久久不能平复，这可能就是计算机魅力无穷的奥秘，你永远不知道平凡的代码下究竟隐藏了多少魔鬼细节。

但不要得意忘形，请keep in mind，你必须明确你

- 不需要短路语义
- 具有确切的profile数据证明有性能提升
- 愿意支付一定的可读性
- 并且愿意承担别人看了代码后认为你是菜鸡的风险

时，才使用这种优化:-)