


# 這篇CPU Cache，估計也沒人看

方志朋 2021-12-17 09:35

歡迎關注方志朋的博客 · 回复“666”獲面試寶典



方志朋

號主為CSDN博客之星，博客訪問量突破一千萬，著有暢銷書《深入理解SpringCloud...

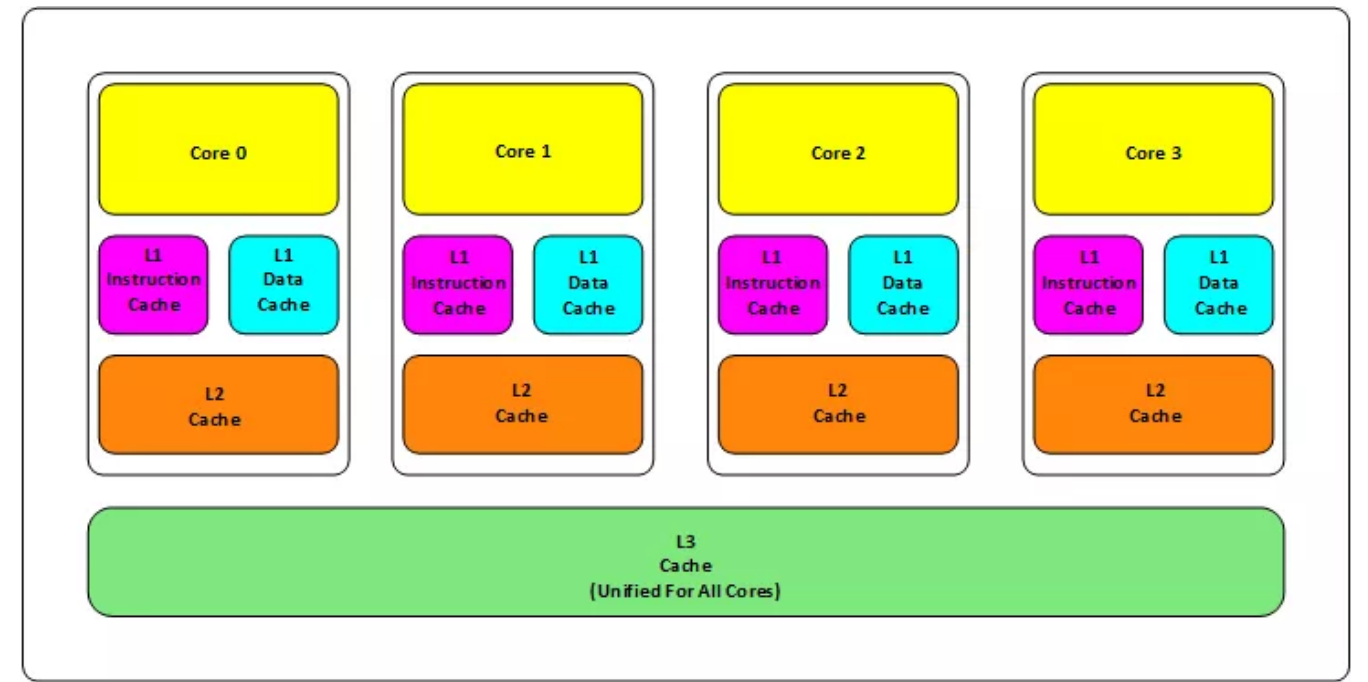
97篇原創內容

公眾號

無論你寫什麼樣的代碼都會交給CPU 來執行，所以，如果你想寫出性能比較高的代碼，這篇文章中提到的技術還是值得認真學習的。另外，千萬別覺得這些東西沒用，這些東西非常有用，十多年前就是這些知識在性能調優上幫了我的很多大忙，從而跟很多人拉開了差距.....

## 基礎知識

首先，我們都知道現在的CPU 多核技術，都會有幾級緩存，老的CPU 會有兩級內存（L1 和 L2），新的CPU會有三級內存（L1，L2，L3），如下圖所示：



其中：

- L1 緩存分成兩種，一種是指令緩存，一種是數據緩存。L2 緩存和L3 緩存不分指令和數據。

- L1 和L2 緩存在每一個CPU 核中，L3 則是所有CPU 核心共享的內存。
- L1、L2、L3 的越離CPU近就越小，速度也越快，越離CPU 遠，速度也越慢。

再往後面就是內存，內存的後面就是硬盤。我們來看一些他們的速度：

- L1 的存取速度：**4 個CPU時鐘週期**
- L2 的存取速度：**11 個CPU時鐘週期**
- L3 的存取速度：**39 個CPU時鐘週期**
- RAM內存的存取速度：**107 個CPU時鐘週期**

我們可以看到，L1 的速度是RAM 的27 倍，但是L1/L2 的大小基本上也就是KB 級別的，L3 會是MB 級別的。例如：Intel Core i7-8700K，是一個6 核的CPU，每核上的L1 是64KB（數據和指令各32KB），L2 是256K，L3 有2MB（我的蘋果電腦是Intel Core i9-8950HK，和Core i7-8700K 的Cache大小一樣）。

我們的數據就從內存向上，先到L3，再到L2，再到L1，最後到寄存器進行CPU 計算。為什麼會設計成三層？這裡有下面幾個方面的考慮：

- 一個方面是物理速度，如果要更大的容量就需要更多的晶體管，除了芯片的體積會變大，更重要的是大量的晶體管會導致速度下降，因為訪問速度和要訪問的晶體管所在的位置成反比，也就是當信號路徑變長時，通信速度會變慢。這部分是物理問題。
- 另外一個問題是，多核技術中，數據的狀態需要在多個CPU中進行同步，並且，我們可以看到，cache 和RAM 的速度差距太大，所以，多級不同尺寸的緩存有利於提高整體的性能。

這個世界永遠是平衡的，一面變得有多光鮮，另一面也會變得有多黑暗。建立這麼多級的緩存，一定就會引入其它的問題，這裡有兩個比較重要的問題，

- 一個是比較簡單的緩存的命中率的問題。
- 另一個是比較複雜的緩存更新的一致性問題。

尤其是第二個問題，在多核技術下，這就很像分佈式的系統了，要對多個地方進行更新。

## 緩存的命中

在說明這兩個問題之前。我們需要要解一個術語Cache Line。緩存基本上來說就是把後面的數據加載到離自己近的地方，對於CPU 來說，它是不會一個字節一個字節的加載的，因為這非常沒有效率，一般來說都是要一塊一塊的加載的，對於這樣的一塊一塊的數據單位，術語叫 **Cache Line**，

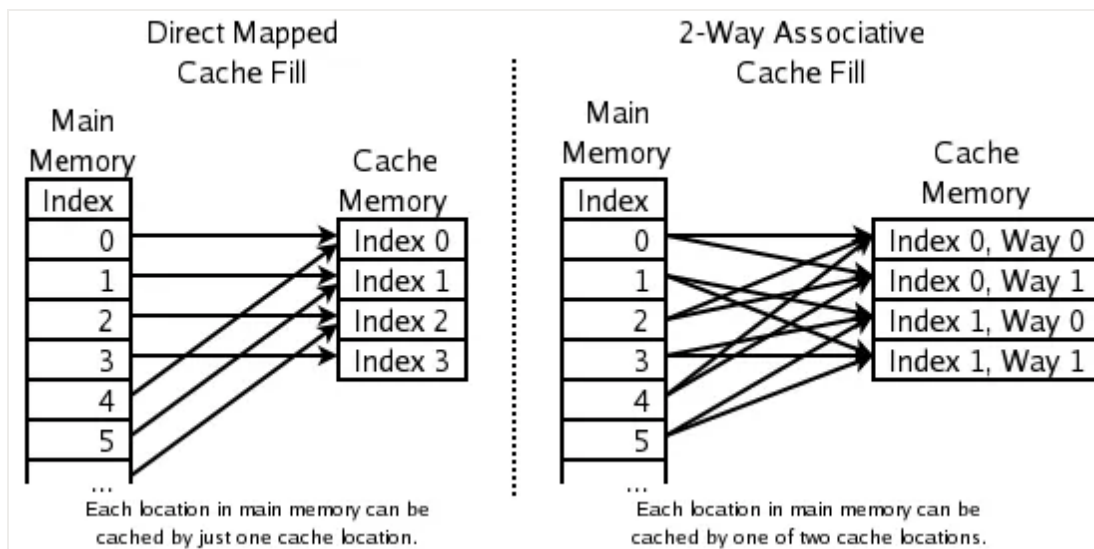
一般來說，一個主流的CPU的Cache Line是64 Bytes（也有的CPU用32Bytes和128Bytes），64 Bytes也就是16個32位的整型，這就是CPU從內存中撈數據上來的最小數據單位。

比如：Cache Line是最小單位（64Bytes），所以先把Cache分佈多個Cache Line，比如：L1有32KB，那麼， $32KB/64B = 512$  個Cache Line。

一方面，緩存需要把內存裡的數據放到放進來，英文叫CPU Associativity。Cache的數據放置的策略決定了內存中的數據塊會拷貝到CPU Cache中的哪個位置上，因為Cache的大小遠遠小於內存，所以，需要有一種地址關聯的算法，能夠讓內存中的數據可以被映射到Cache中來。這個有點像內存地址從邏輯地址向物理地址映射的方法，但不完全一樣。

基本上來說，我們會有如下的一些方法。

- 一種方法是，任何一個內存地址的數據可以被緩存在任何一個Cache Line裡，這種方法是最靈活的，但是，如果我們要知道一個內存是否存在於Cache中，我們就需要進行 $O(n)$ 複雜度的Cache遍歷，這是很沒有效率的。
- 另一種方法，為了降低緩存搜索算法，我們需要使用像Hash Table這樣的數據結構，最簡單的hash table就是做求模運算，比如：我們的L1 Cache有512個Cache Line，那麼，公式： $(\text{內存地址} \bmod 512) * 64$  就可以直接找到所在的Cache地址的偏移了。但是，這樣的方式需要我們的程序對內存地址的訪問要非常地平均，不然衝突就會非常嚴重。這成了一種非常理想的情況了。
- 為了避免上述的兩種方案的問題，於是就要容忍一定的hash衝突，也就出現了N-Way 關聯。也就是把連續的N個Cache Line綁成一組，然後，先把找到相關的組，然後再在這個組內找到相關的Cache Line。這叫Set Associativity。如下圖所示。



對於N-Way 組關聯，可能有點不好理解，這裡個例子，並多說一些細節（不然後面的代碼你會不能理解），Intel 大多數處理器的L1 Cache 都是32KB，8-Way 組相聯，Cache Line 是64 Bytes。這意味著，

- 32KB的可以分成， $32KB / 64 = 512$  條Cache Line。

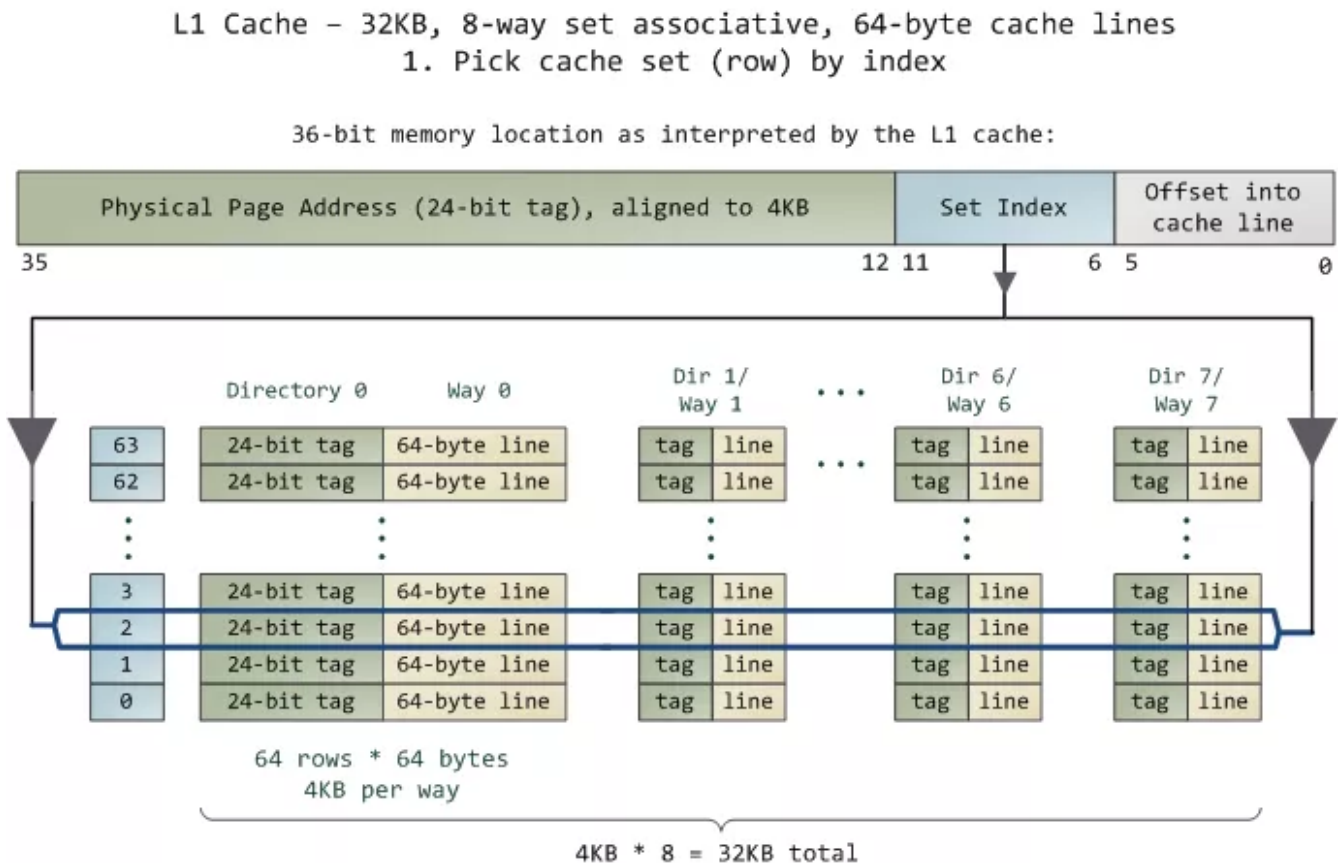
- 因為有8 Way，於是會每一Way有 $512 / 8 = 64$  條Cache Line。
- 於是每一路就有 $64 \times 64 = 4096$  Bytes 的內存。

為了方便索引內存地址，

- **Tag**：每條Cache Line 前都會有一個獨立分配的24 bits來存的tag，其就是內存地址的前24bits
- **Index**：內存地址後續的6 個bits 則是在這一Way 的是Cache Line 索引， $2^6 = 64$  剛好可以索引64條Cache Line
- **Offset**：再往後的6bits 用於表示在Cache Line 裡的偏移量

如下圖所示：（圖片來自《Cache: a place for concealment and safekeeping》）

當拿到一個內存地址的時候，先拿出中間的6bits 來，找到是哪組。

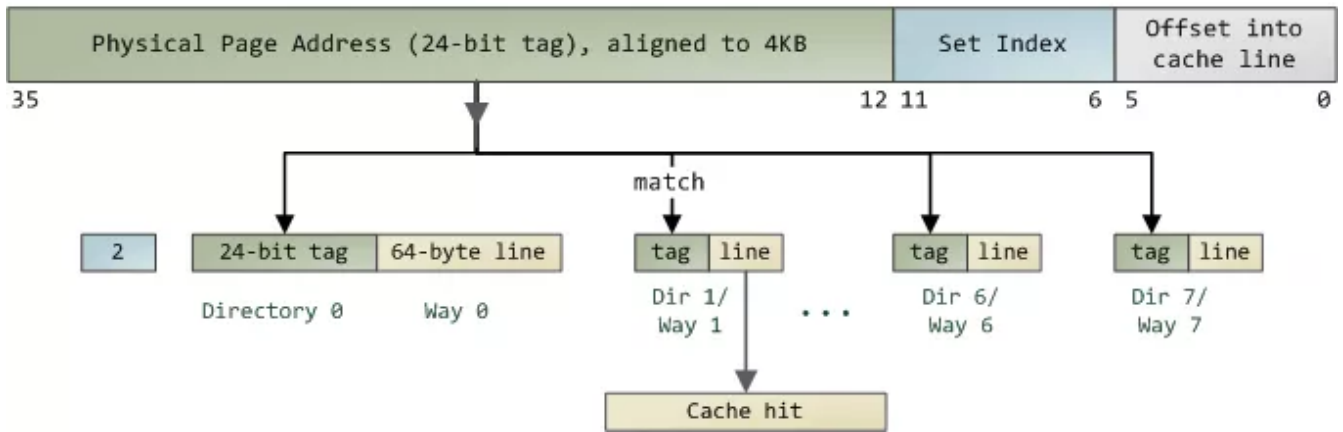


然後，在這一個8 組的cache line 中，再進行 $O(n)$   $n=8$  的遍歷，主是要匹配前24bits 的tag。如果匹配中了，就算命中，如果沒有匹配到，那就是cache miss，如果是讀操作，就需要進向後面的緩存進行訪問了。

L2/L3 同樣是這樣的算法。而淘汰算法有兩種，一種是隨機一種是LRU。現在一般都是以LRU 的算法（通過增加一個訪問計數器來實現）

## 2. Search for matching tag in the set

36-bit memory location as interpreted by the L1 cache:



這也意味著：

- L1 Cache 可映射36bits 的內存地址，一共 $2^{36} = 64\text{GB}$  的內存
- 當CPU 要訪問一個內存的時候，通過這個內存中間的6bits 定位是哪個set，通過前24bits 定位相應的Cache Line。
- 就像一個hash Table 的數據結構一樣，先是 $O(1)$ 的索引，然後進入衝突搜索。
- 因為中間的6bits 決定了一個同一個set，所以，對於一段連續的內存來說，每隔4096 的內存會被放在同一個組內，導致緩存衝突。

此外，當有數據沒有命中緩存的時候，CPU 就會以最小為Cache Line 的單元向內存更新數據。當然，CPU 並不一定只是更新64Bytes，因為訪問主存實在是太慢了，所以，一般都會多更新一些。好的CPU 會有一些預測的技術，如果找到一種pattern 的話，就會預先加載更多的內存，包括指令也可以預加載。

這叫Prefetching 技術（參看，Wikipedia 的Cache Prefetching 和紐約州立大學的Memory Prefetching）。比如，你在for-loop訪問一個連續的數組，你的步長是一個固定的數，內存就可以做到prefetching。（注：指令也是以預加載的方式執行）

了解這些細節，會有利於我們知道在什麼情況下有可以導致緩存的失效。

## 緩存的一致性

對於主流的CPU 來說，緩存的寫操作基本上是兩種策略，

- 一種是Write Back，寫操作只要在cache 上，然後再flush 到內存上。
- 一種是Write Through，寫操作同時寫到cache 和內存上。

為了提高寫的性能，一般來說，主流的CPU（如：Intel Core i7/i9）採用的是Write Back 的策略，因為直接寫內存實在是太慢了。

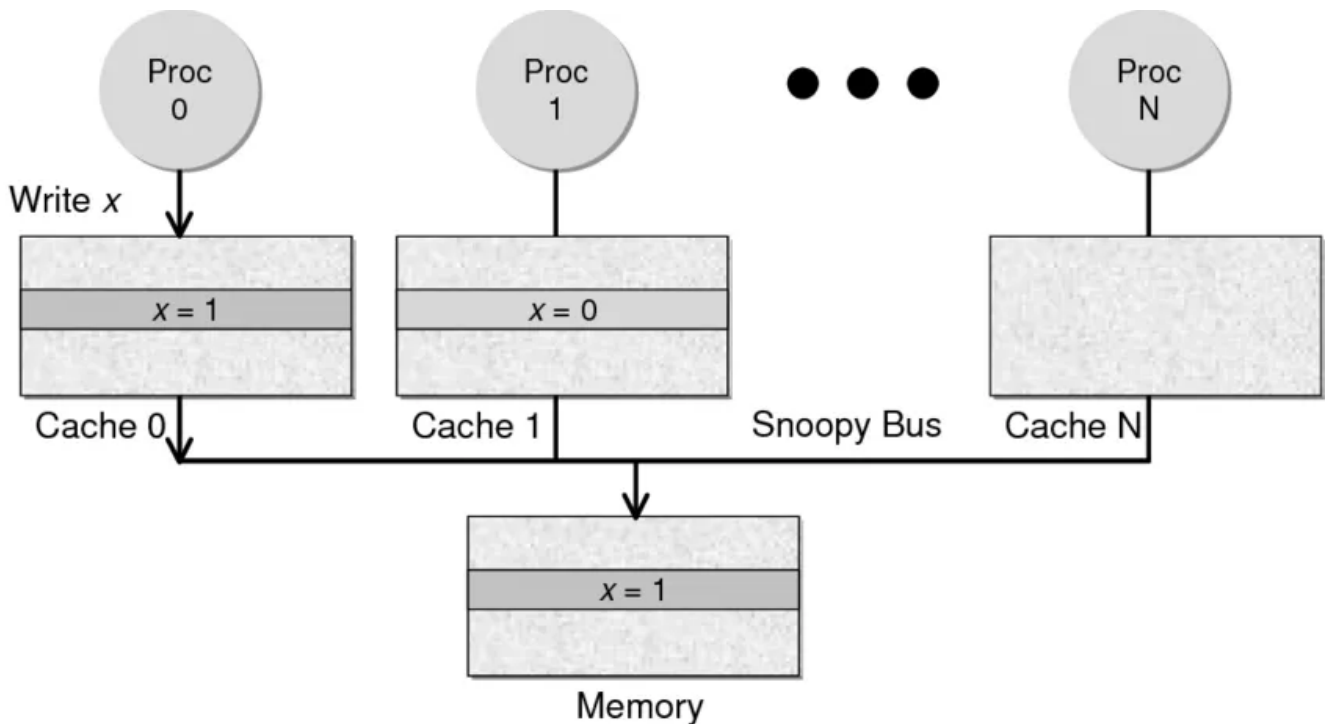
好了，現在問題來了，如果有一個數據x 在CPU 第0 核的緩存上被更新了，那麼其它CPU 核上對於這個數據x 的值也要被更新，這就是緩存一致性的問題。（當然，對於我們上層的程序我



們不用關心CPU 多個核的緩存是怎麼同步的，這對上層的代碼來說都是透明的）

一般來說，在CPU 硬件上，會有兩種方法來解決這個問題。

- **Directory 協議**。這種方法的典型實現是要設計一個集中式控制器，它是主存儲器控制器的一部分。其中有一個目錄存儲在主存儲器中，其中包含有關各種本地緩存內容的全局狀態信息。當單個CPU Cache 發出讀寫請求時，這個集中式控制器會檢查並發出必要的命令，以在主存和CPU Cache之間或在CPU Cache自身之間進行數據同步和傳輸。
- **Snoopy 協議**。這種協議更像是一種數據通知的總線型的技術。CPU Cache 通過這個協議可以識別其它Cache上的數據狀態。如果有數據共享的話，可以通過廣播機制將共享數據的狀態通知給其它CPU Cache。這個協議要求每個CPU Cache 都可以窺探數據事件的通知並做出相應的反應。如下圖所示，有一個Snoopy Bus 的總線。



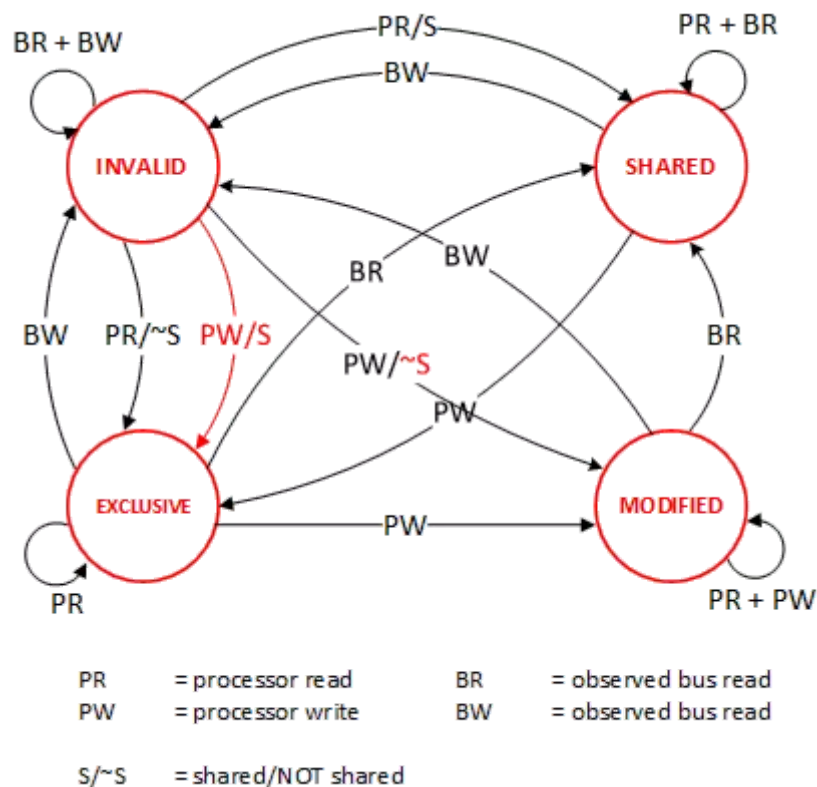
因為Directory 協議是一個中心式的，會有性能瓶頸，而且會增加整體設計的複雜度。而 Snoopy 協議更像是微服務+消息通訊，所以，現在基本都是使用Snoopy 的總線的設計。

這裡，我想多寫一些細節，因為這種微觀的東西，讓人不自然地就會跟分佈式系統關聯起來，在分佈式系統中我們一般用Paxos/Raft 這樣的分佈式一致性的算法。

而在CPU 的微觀世界裡，則不必使用這樣的算法，原因是因為CPU 的多個核的硬件不必考慮網絡會斷會延遲的問題。所以，CPU 的多核心緩存間的同步的核心就是要管理好數據的狀態就好了。

這裡介紹幾個狀態協議，先從最簡單的開始，MESI 協議，這個協議跟那個著名的足球運動員梅西沒什麼關係，其主要表示緩存數據有四個狀態：Modified (已修改)，Exclusive (獨占的)，Shared (共享的)，Invalid (無效的)。

這些狀態的狀態機如下所示（有點複雜，你可以先不看，這個圖就是想告訴你狀態控制有多複雜）：



下面是個示例（如果你想看一下動畫演示的話，這裡有一個網頁（MESI Interactive Animations），你可以進行交互操作，這個動畫演示中使用的Write Through 算法）：

當前操作	CPU0	CPU1	Memory	說明
1) CPU0 read(x)	x=1 (E)		x=1	只有一個CPU有x 變量，所以，狀態是 Exclusive
2) CPU1 read(x)	x=1 (S)	x=1(S)	x=1	有兩個CPU都讀取x 變量，所以狀態變成 Shared
3) CPU0 write(x,9)	x=9 (M)	x=1(I)	x=1	變量改變，在CPU0中狀態變成Modified，在CPU1中狀態變成Invalid
4) 變量x 寫回內存	x=9 (M)	X=1(I)	x=9	目前的狀態不變
5) CPU1 read(x)	x=9 (S)	x=9(S)	x=9	變量同步到所有的Cache中，狀態回到 Shared

MESI 這種協議在數據更新後，會標記其它共享的CPU 緩存的數據拷貝為Invalid 狀態，然後當其它CPU 再次read 的時候，就會出現cache miss 的問題，此時再從內存中更新數據。從內存中更新數據意味著20 倍速度的降低。

我們能不能直接從我隔壁的CPU 緩存中更新？是的，這就可以增加很多速度了，但是狀態控制也就變麻煩了。還需要多來一個狀態：**Owner(宿主)**，用於標記，我是更新數據的源。於是，出現了**MOESI** 協議

**MOESI** 協議的狀態機和演示示例我就不貼了（有興趣可以上Berkeley上看看相關的課件），我們只需要理解**MOESI**協議允許**CPU Cache** 間同步數據，於是也降低了對內存的操作，性能是非常大的提升，但是控制邏輯也非常複雜。

順便說一下，與**MOESI** 協議類似的一個協議是**MESIF**，其中的F 是Forward，同樣是把更新過的數據轉發給別的CPU Cache 但是，**MOESI** 中的Owner 狀態和**MESIF** 中的Forward 狀態有一個非常大的不一樣—— **Owner** 狀態下的數據是**dirty** 的，還沒有寫回內存，**Forward** 狀態下的數據是**clean**的，可以丟棄而不用另行通知。

需要說明的是，AMD 用**MOESI**，Intel 用**MESIF**。所以，F 狀態主要是針對CPU L3 Cache 設計的（前面我們說過，L3 是所有CPU 核心共享的）。（相關的比較可以參看StackOverflow上這個問題的答案）

## 程序性能

了解了我們上面的這些東西後，我們來看一下對於程序的影響。

### 示例一

首先，假設我們有一個64M長的數組，設想一下下面的兩個循環：

```
const int LEN = 64*1024*1024;
int *arr = new int[LEN];

for (int i = 0; i < LEN; i += 2) arr[i] *= i;

for (int i = 0; i < LEN; i += 8) arr[i] *= i;
```

按我們的想法來看，第二個循環要比第一個循環少4倍的計算量，其應該也是要快4倍的。但實際跑下來並不是，在我的機器上，第一個循環需要127 毫秒，第二個循環則需要121 毫秒，相差無幾。

這裡最主要的原因就是Cache Line，因為CPU 會以一個Cache Line 64Bytes 最小時單位加載，也就是16 個32bits 的整型，所以，無論你步長是2 還是8，都差不多。而後面的乘法其實是不耗CPU 時間的。



## 示例二

我們再來看一個與緩存命中率有關的代碼，我們以一定的步長 `increment` 來訪問一個連續的數組。

```
for (int i = 0; i < 10000000; i++) {
    for (int j = 0; j < size; j += increment) {
        memory[j] += j;
    }
}
```

我們測試一下，在下表中，表頭是步長，也就是每次跳多少個整數，而縱向是這個數組可以跳幾次（你可以理解為要幾條Cache Line），於是表中的任何一項代表了這個數組有多少，而且步長是多少。

比如：橫軸是512，縱軸是4，意思是，這個數組有  $4 * 512 = 2048$  個長度，訪問時按512步長訪問，也就是訪問其中的這幾項：`[0, 512, 1024, 1536]` 這四項。

表中同的項是，是循環1000 萬次的時間，單位是“微秒”（除以1000後是毫秒）

count	1	16	512	1024
1	17539	16726	15143	14477
2	15420	14648	13552	13343
3	14716	14463	15086	17509
4	18976	18829	18961	21645
5	23693	23436	74349	29796
6	23264	23707	27005	44103
7	28574	28979	33169	58759
8	33155	34405	39339	65182
9	37088	37788	49863	156745
10	41543	42103	58533	215278
11	47638	50329	66620	335603
12	49759	51228	75087	305075
13	53938	53924	77790	366879
14	58422	59565	90501	466368
15	62161	64129	90814	525780
16	67061	66663	98734	440558
17	71132	69753	171203	506631
18	74102	73130	293947	550920

我們可以看到，從 `[9, 1024]` 以後，時間顯著上升。包括 `[17, 512]` 和 `[18, 512]` 也顯著上升。這是因為，我機器的L1 Cache 是32KB, 8 Way 的，前面說過，8 Way 的有64 組，每組8

個Cache Line，當for-loop步長超過1024 個整型，也就是正好4096 Bytes 時，也就是導致內存地址的變化是變化在高位的24bits 上，

而低位的12bits 變化不大，尤其是中間6bits沒有變化，導致全部命中同一組set，導致大量的cache 衝突，導致性能下降，時間上升。而[16, 512]也是一樣的，其中的幾步開始導致L1 Cache開始衝突失效。

### 示例三

接下來，我們再來看個示例。下面是一個二維數組的兩種遍歷方式，一個逐行遍歷，一個是逐列遍歷，這兩種方式在理論上來說，尋址和計算量都是一樣的，執行時間應該也是一樣的。

```
const int row = 1024;
const int col = 512
int matrix[row][col];

//逐行遍歷
int sum_row=0;
for(int _r=0; _r<row; _r++) {
    for(int _c=0; _c<col; _c++){
        sum_row += matrix[_r][_c];
    }
}

//逐列遍歷
int sum_col=0;
for(int _c=0; _c<col; _c++) {
    for(int _r=0; _r<row; _r++){
        sum_col += matrix[_r][_c];
    }
}
```

然而，並不是，在我的機器上，得到下面的結果。

- 逐行遍歷：0.081ms
- 逐列遍歷：1.069ms

執行時間有十几倍的差距。其中的原因，就是逐列遍歷對於 CPU Cache 的運作方式並不友好，所以，付出巨大的代價。

### 示例四

接下来，我们来看一下多核下的性能问题，参看如下的代码。两个线程在操作一个数组的两个不同的元素（无需加锁），线程循环1000万次，做加法操作。在下面的代码中，我高亮了一行，就是 `p2` 指针，要么是 `p[1]`，或是 `p[30]`，理论上来说，无论访问哪两个数组元素，都应该是一样的执行时间。

```
void fn (int* data) {
    for(int i = 0; i < 10*1024*1024; ++i)
        *data += rand();
}

int p[32];

int *p1 = &p[0];
int *p2 = &p[1]; // int *p2 = &p[30];

thread t1(fn, p1);
thread t2(fn, p2);
```

然而，并不是，在我的机器上执行下来的结果是：

- 对于 `p[0]` 和 `p[1]` ：560ms
- 对于 `p[0]` 和 `p[30]` ：104ms

这是因为 `p[0]` 和 `p[1]` 在同一条 Cache Line 上，而 `p[0]` 和 `p[30]` 则不可能在同一条 Cache Line 上，CPU 的缓存最小的更新单位是 Cache Line，所以，这导致虽然两个线程在写不同的数据，但是因为这两个数据在同一条 Cache Line 上，就会导致缓存需要不断进在两个 CPU 的 L1/L2 中进行同步，从而导致了 5 倍的时间差异。

## 示例五

接下来，我们再来看一下另外一段代码：我们想统计一下一个数组中的奇数个数，但是这个数组太大了，我们希望能用多线程来完成这个统计。下面的代码中，我们为每一个线程传入一个 `id`，然后通过这个 `id` 来完成对应数组段的统计任务。这样可以加快整个处理速度。

```
int total_size = 16 * 1024 * 1024; //数组长度
int* test_data = new test_data[total_size]; //数组
int nthread = 6; //线程数（因为我的机器是6核的）
int result[nthread]; //收集结果的数组

void thread_func (int id) {
    result[id] = 0;
    int chunk_size = total_size / nthread + 1;
```

```

int start = id * chunk_size;
int end = min(start + chunk_size, total_size);

for ( int i = start; i < end; ++i ) {
    if (test_data[i] % 2 != 0 ) ++result[id];
}
}

```

然而，在执行过程中，你会发现，**6 个线程居然跑不过 1 个线程**。因为根据上面的例子你知道 `result[]` 这个数组中的数据在一个 Cache Line 中，所以，所有的线程都会对这个 Cache Line 进行写操作，导致所有的线程都在不断地重新同步 `result[]` 所在的 Cache Line，所以，导致 6 个线程还跑不过一个线程的结果。这叫 **False Sharing**。

优化也很简单，使用一个线程内的变量。

```

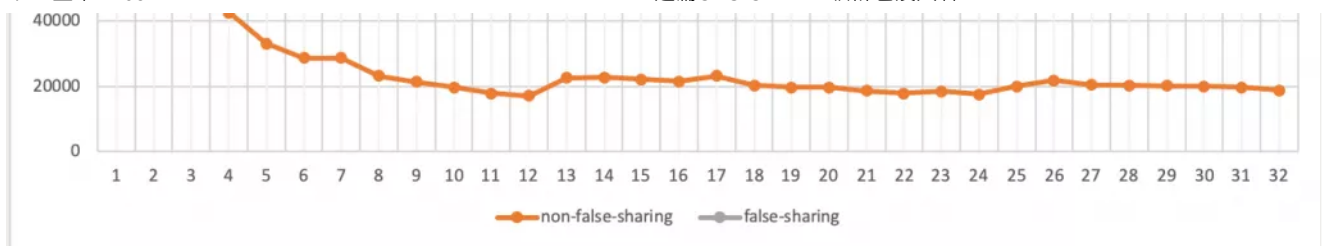
void thread_func (int id) {
    result[id] = 0;
    int chunk_size = total_size / nthread + 1;
    int start = id * chunk_size;
    int end = min(start + chunk_size, total_size);

    int c = 0; //使用临时变量，没有cache line的同步了
    for ( int i = start; i < end; ++i ) {
        if (test_data[i] % 2 != 0 ) ++c;
    }
    result[id] = c;
}

```

我们把两个程序分别在 1 到 32 个线程上跑一下，得出的结果画一张图如下所示（横轴是线程数，纵轴是完成统计的时间，单位是微秒）：





上图中，我们可以看到，灰色的曲线就是第一种方法，橙色的就是第二种（用局部变量的）方法。当只有一个线程的时候，两个方法相当，基本没有什么差别，但是在线程数增加的时候，你会发现，第二种方法的性能提高的非常快。直到到达 6 个线程的时候，开始变得稳定（前面说过，我的 CPU 是 6 核的）。

而第一种方法无论加多少线程也没有办法超过第二种方法。因为第一种方法不是 CPU Cache 友好的。也就是说，第二种方法，只要我的 CPU 核数足够多，就可以做到线性的性能扩展，让每一个 CPU 核都跑起来，而第一种则不能。

转自：程序员cxuan

<https://mp.weixin.qq.com/s/s9w--YRkyAvQi4LQcenq4g>

热门内容：

- [Spring Boot 实现万能文件在线预览](#)
- [突发！Log4j 爆“核弹级”漏洞，Flink、Kafka等至少十多个项目受影响](#)
- [国内最牛逼的笔记，不接受反驳！！](#)
- [顶配版阿里大佬面试笔记+300道硬核面试题，跪着啃完了。。。。](#)
- [阿里二面：GET 请求能传图片吗？](#)

## 方志朋的专栏

专注于Java、SpringBoot、  
SpringCloud、微服务、Docker、  
Kubernetes、持续集成等领域



▲长按图片识别二维码关注

最近面試BAT，整理一份面試資料《Java面試BAT通關手冊》，覆蓋了Java核心技術、JVM、Java並發、SSM、微服務、數據庫、數據結構等等。

獲取方式：點“**在看**”，關注公眾號並回復 **666** 領取，更多內容陸續奉上。

明天見(。·ω·。)/♡

喜歡此內容的人還喜歡

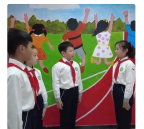
Spring Boot 實現萬能文件在線預覽

方志朋



銘記抗美援朝歷史傳承抗美援朝精神——山東省濟南市章丘區清照小學開展“我心目中最可愛的人”主題少先隊活動課

紅領巾集結號



襯衫先別急著收起來，冬天這樣搭配保暖又時髦

視覺藝術部

