窥探 CPU Cache

2016/5/26

renyl

介绍

1) CPU 负责程序指令的执行，内存负责数据的存储，但 CPU 的执行速度远大于内存的访问速度，为了缓和两者之间的速度差异，于是 CPU Cache 就应运而生了。

2) CPU Cache 由硬件实现，速度介于寄存器和内存之间，系统会把经常使用的数据放到 Cache 中，当对相同数据进行多次操作时，就可以避免从内存中获取数据，而直接从 CPU Cache 中获取数据，这样就会提高程序性能。

3) CPU Cache 在高级语言（C，C++，Java 等）程序员的角度来看，它是透明的，无法直接干预它，也无法察觉它是如何运行的，因为它是完全依赖硬件设施来实现的。但是，我们可以利用它，在并发编程中善于运用 CPU Cache，会给程序性能带来“质”的提升。

本文主要通过 3 个知识点入口，每个知识点将结合一个程序来说明 CPU Cache 的工作原理及相关技术。

1 cache-miss（缓存失效）

该部分主要介绍 CPU Cache 的一些基础知识以及 cache-miss 对程序性能的影响。

1.1 背景

现代计算机通常具有两级或三级 Cache，一般叫做 L1、L2 和 L3 Cache。 Cache 级别越小，容量也越小，所在物理位置也越接近 CPU，速度也越快。如 L1 Cache 是最接近 CPU 的，速度最快，容量最小。

当 CPU 对内存发出访问请求时，会先查看 Cache 内是否有请求数据。如果存在，则不需要访问内存直接返回该数据，该过程称为 cache-hit（缓存命中）；如果不存在，则需要先把内存中的相应数据载入到Cache 中，再将其返回CPU，该过程称为cache-miss(缓存失效)。

Cache 之所以能够有效地减少程序访问内存的时间，有如下两个原因：
1） 程序运行时对内存的访问呈现空间局部性（Spatial Locality）。
2） 程序运行时对内存的访问呈现时间局部性（Temporal Locality）。

如果程序能有效利用这两种局部性，cache hit 就可以达到极高的命中率，程序性能也就有很大的提升。

操作系统为了使 CPU Cache 更加高效的工作，并不是简单的将单条数据写入到 Cache 中，而是以 cache line 为单位写入到 Cache 中。（cache line 的大小由下面将介绍的参数 coherency_line_size 指定）

接下来，介绍下如何查看系统中各级别 Cache 的大小及相关信息：
1) 在 Windows 下可以通过调用 API GetLogicalProcessorInfo 来查看。
2) 在 Linux 下，可以有多种方法查看机器上 CPU Cache 信息，下面主要介绍两种方法来查看 CPU Cache 的相关信息：

A：lscpu 命令

```
[root@localhost testdir]# lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             2
Vendor ID:             GenuineIntel
```

```
CPU family:            15
Model:                 6
Stepping:              4
CPU MHz:               2992.461
BogoMIPS:              5984.28
Virtualization:        VT-x
L1d cache:             16K
L2 cache:              2048K
```

B：查看/sys/devices/system/cpu/cpu0/cache/目录下的文件，显示如下：

```
[root@localhost testdir]#  cat /sys/devices/system/cpu/cpu0/cache/index
index0/ index1/
```

一般情况下：
index0 表示：1 级数据 cache
index1 表示：1 级指令 cache
index2 表示：2 级 cache
index3 表示：3 级 cache （对应/proc/cpuinfo/文件显示的 cache_size）
注：我的机器上只有 L1 和 L2 Cache，没有 L3 Cache。因此，只有 index0 和 index1 两个目录。

目录里的各文件描述 cache 的相关信息，以本机的 cpu0/index0 为例进行说明：

```
[root@localhost testdir]#  cat /sys/devices/system/cpu/cpu0/cache/index0/
coherency_line_size       number_of_sets                  shared_cpu_list
size                      ways_of_associativity      level
physical_line_partition   shared_cpu_map           type
```

各个参数的意义与值如下表所示：

文件	内容	说明	
type	Data	Date 表示数据 Cache，Instruction 表示指令 Cache	
level	1	1 表示 L1 Cache，2 表示 L2 Cache，3 表示 L3 Cache	
size	16K	大小为 16KB	
coherency_line_size	64	cache line 大小为 64B	64B*1*8*32=16KB
physical_line_partition	1	-	
ways_of_associativity	8	8 路组关联	
number_of_sets	32	-	
shared_cpu_list	0	表示这个 Cache 只被 CPU0 共享	
shared_cpu_map	00000001	同上，十六进制表示法	

接下来，我们通过一小段代码来说明 cache-miss 对程序性能的影响。

1.2 源码

1）源码说明：

对一个两维数组进行按行或按例循环访问

2）源码显示：

cache1.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5
 6
 7 int main(int argc, char* argv[])
 8 {
 9
10     int  count=10240;
11     int **array=(int**)malloc(count*sizeof(int));
12
13     for ( int i=0; i<10240;i++ )
14     {
15         array[i]=(int *)malloc(count*sizeof(int));
16         memset(array[i],1,count*sizeof(int));
17     }
18
19     for ( int i=0; i<count; i++ )
20     {
21         for ( int j=0; j<count; j++ )
22         {
23             array[i][j]+=j;
24 //          array[j][i]+=j;
25         }
26     }
27
28     return 0;
29 }
```

1.3 结果

1）编译后运行，结果如下：

```
[root@localhost testdir]# gcc -std=c99 cache1.c
[root@localhost testdir]# time ./a.out
real    0m1.290s
user    0m0.676s
sys     0m0.615s
```

2）将 24 行的注释取消，将 23 行注释，编译后再次运行，结果如下：

```
[root@localhost testdir]# gcc -std=c99 cache1.c
[root@localhost testdir]# time ./a.out
real    0m8.059s
user    0m7.455s
sys     0m0.602s
```

1.4 疑问

同样是对一个二维数组循环，一个是按行循环，一个是按列循环，循环次数一样，运行时间却相差近 7 倍。

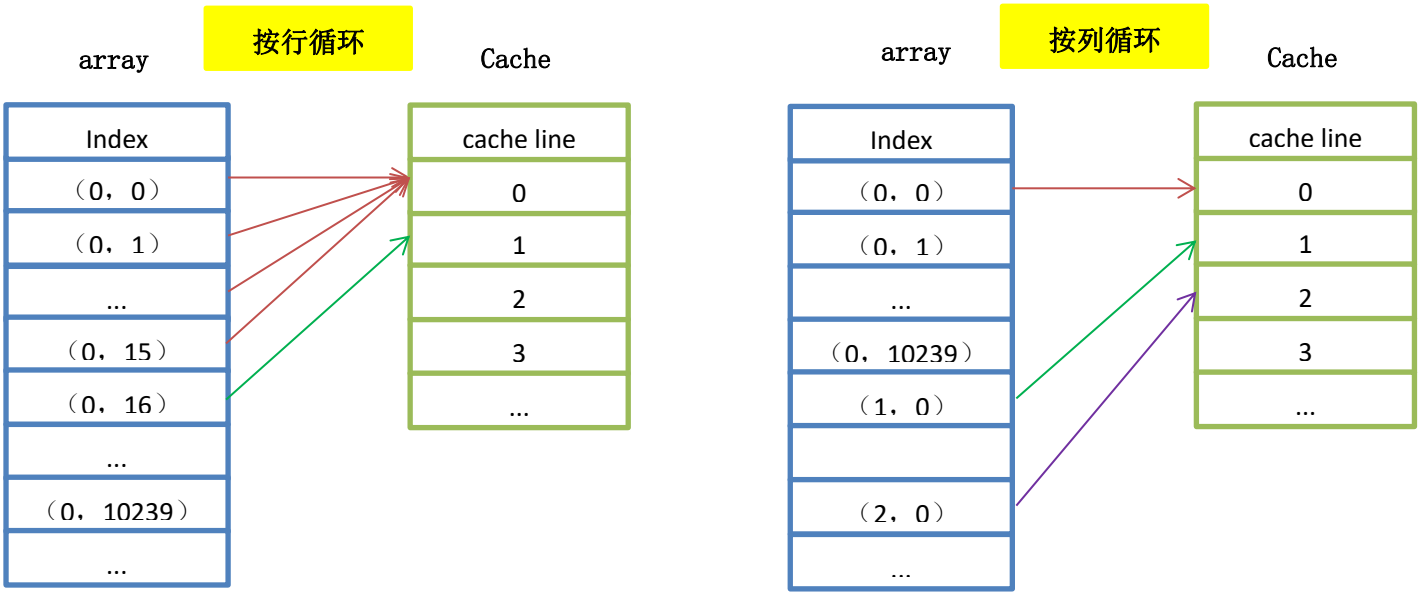
1.5 分析

1） 针对上面的测定结果，有 2 点需要说明：

A. 二维组数在内存布局上是连续的，即每行数据在内存中都是连续的。

B. 程序运行时间的长短，并不是由数据进行多少次的加法操作运算决定，而是由访问内存的次数决定。

2） 现在来分析测定结果差异的原因：（结合下图来分析）



A. 按行循环：当访问第一个数据 array[0][0]时，系统会从内存中把 array[0][0]，array[0][1]，…，arrar[0][15]共 16 个数据（int 占 4 个字节）放到一个 cache line 中，当访问 array[0][1]到 array[0][15]时，就可以直接从 Cache 中获取数据，而不必再次从内存中获取。访问 Cache 的速度大约是访问内存速度的 100 倍，因此总的访问内存时间减少约为 94%（1-115/(16*100)）。

B. 按列循环：访问第一个数据为 array[0][0]，访问第二个数据为 array[1][0]，访问第三个数据为 array[2][0]，….。每次访问相邻数据在内存布局上都相差 4096 个字节（1024*sizeof（int））。因此，对二维组数的每次访问都无法从 Cache 中获取数据，每次都要从内存中获取数据。

注：这里还涉及到 ways_of_associativity，暂时不考虑，后面会专门分析。

现在我们知道，“按行循环”速度快是由于 Cache 的命中率高，即 cache-misses 数比“按列循环”少。接下来，就用 perf 工具进行验证一下。

按行循环：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses  ./a.out
 Performance counter stats for './a.out'

         14,423,110 L1-dcache-misses

       1.289017444 seconds time elapsed
```

按列循环：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses  ./a.out
 Performance counter stats for './a.out':

          142,794,723 L1-dcache-misses

       8.267403173 seconds time elapsed
```

可以看到，“按行循环”要比“按例循环”的 cache-misses 数少近 90%。

2 false sharing（伪共享）

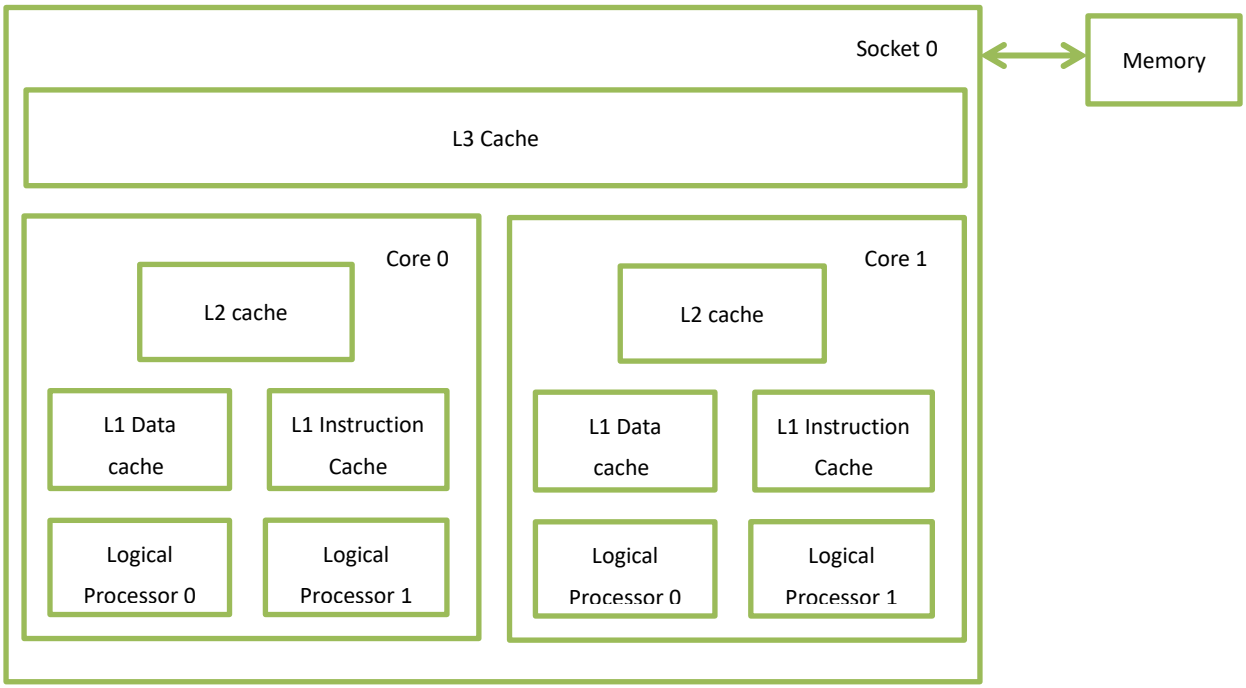
该部分主要介绍 CPU Cache 伪共享的一些基础知识以及在多核程序中对性能的影响。

2.1 背景

CPU Cache 是以 cache line 为单位组成，当两个或以上 CPU 都要操作同一个 cache line，这时就会发生冲突现象。但是，有时多个 CPU 访问的是同一个 cache line 中的不同内容， 由于 cache line 的存在与限制，多个 CPU 即使访问不同的内容，但访问的是同一个 cache line，这种现象称之为“伪共享”。

举例来说，当两个 Thread 操作同一个 int 型数组 int array[2]时。如果 Thread 0 只访问 array[0]，Thread 1 只访问 array[1]，那么 Thread 0 和 Thread 1 之间不应该发生数据共享。但是，由于一个 cache line 可以包含多个 int，那么 array[0]与 array[1]被放置在同一个 cache line 里。这样，当 Thread 0 和 Thread 1 同时对数组 array 进行操作时，系统需要花费额外资源和时间运用控制协议来协调这个 cache line。但这是不必要的，因为在这种情况下，把每个数组元素单独放在一个 cache line 里程序性能会更好。

接下来，先通过下面的一张图来理解下 CPU 之间 L1、L2 和 L3 Cache 的共享情况：



注：Logical Processor 为开启 HyperThread 后所显示的两个逻辑 CPU。

对上图进行说明：

1) 每个 Core 都有自己私有的 L1 和 L2 Cache，Core 之间共享 L3 Cache。
2) Socket（物理 CPU）之间不共享任何 Cache。

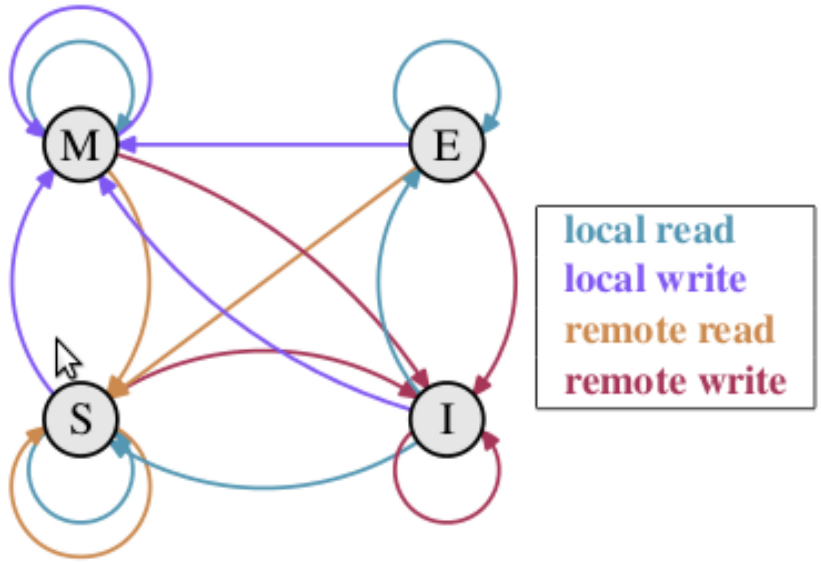
注：可以通过前面介绍的/sys/devices/system/cpu/cpu0/cache/index*文件下的 shared_cpu_list 参数来验证。

最后，分析下多个 CPU 访问同一个 cache line 时，系统如何进行协调控。

假设如下：当 Core0 想要访问 Core1 Cache（L1 或 L2）（或者别的 Socket 上的 L1、L2 和 L3 Cache）时，系统会把 Core1 中的 Cache 数据发送给 Core0。但是，如果 Core0 对这份数据进行了修改，如何通知 Core1 这份数据已经修改了呢？这里就会出现 Cache 的同步问题了。

MESI 协议和 RFO 请求就应运而生了，接下来简单介绍下：

先看如下一张图：



说明如下：

a) M(修改, Modified)：本地 Core 已经修改 cache line（即是脏行），它的内容与内存中的内容不一样，且此 cache line 只有本地一个拷贝(专有)
b) E（专有，Exclusive)：cache line 内容和内存中的一样，且其它 Core 都没有这行数据
c) S（共享，Shared)：cache line 内容和内存中的一样，有可能其它 Core 也存在此 cache line 的拷贝
d) I(无效，Invalid)：cache line 失效，不能使用

Cache 的四种状态转换过程如下：

1) **初始**：开始时，cache line 里没有加载任何数据，因此它处于 I 状态
2) **本地写**(Local Write)：如果本地 Core 写数据至处于 I 状态的 cache line，则 cache line 的状态变成 M
3) **本地读**(Local Read)：如果本地 Core 读取处于 I 状态的 cache line，由于此 Cache 没有数据给它。此时分两种情况：
   A：其他 Core 的 Cache 里也没有此 cache line 数据，则从内存加载数据到 cache line 后再将它设成 E 状态（表示只有本 Core 有此这条数据，其它 Core 都没有）

B:其它 Core 的 Cache 有此行数据，则将此 cache line 的状态设为 S 状态

4) **远程读**(Remote Read)：假设 Core5 要读 Core0 的 cache line 内容（Core5 和 Core0 不在同一个 socket 上)， Core0 需要把它 cache line 的内容通过内存控制器(Memory Controller)发送给 Core5， Core5 接到后将相应的 cache line 状态设为 S。在设置之前，内存也得从总线上得到这份数据并保存.

5) **远程写**(Remote Write)： Core1 得到 Core0 的 cache line 后，不是为了读，而是为了写（确切地说不是远程写，算是本地写)。但是 Core0 也拥有这份数据的拷贝，这该怎么办呢? Core1 将发出一个 RFO(Request For Owner)请求，它需要拥有此 cache line 数据的权限，其它 Core 的相应 cache line 设为 I，除了这个 Core， 其他 Core 都不能读写此 Cache line，这保证了数据的安全，同时处理 RFO 请求以及设置 I 的过程将给写操作带来很大的性能消耗。

注：如果处于 M 状态的缓存行，再由本地 Core 写入/读出，状态是不会改变的

2.2 源码

1）源码说明：

4 个线程分别访问结构体数组的不同内容

2）源码显示：

Cache2.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5 #include <pthread.h>
 6
 7
 8 typedef struct tag_volatile_int {
 9
10      volatile int value;  //read data don't from register
11      int useless[15];
12
13 } volatile_int;
14
15
16 void *thread_function(void *arg)
17 {
18
19     volatile_int *data = (volatile_int *) arg;
20
21     for (int i = 0; i < 10000; i++) {
22         for (int j = 0; j < 10000; j++)
23             data->value++;
```

```
24      }
25
26      return NULL;
27  }
28
29  int main(int argc, char *argv[])
30  {
31
32      volatile_int data[4];
33      memset(data, 0, sizeof(data));
34
35      pthread_t tid[4];
36      int err;
37
38      for (int i = 0; i < 4; i++) {
39
40        err = pthread_create(&tid[i], NULL, thread_function, (void *) &data[i]);
41        if (err != 0) {
42              printf("Can't create thread:%s\n", strerror(err));
43              exit(1);
44          }
45      }
46
47      for (int i = 0; i < 4; i++)
48          pthread_join(tid[i], NULL);
49
50      return 0;
```

2.3 结果

1）编译后运行，结果如下：

```
[root@localhost testdir]# gcc -std=c99 cache2.c -lpthread
[root@localhost testdir]# time ./a.out
real    0m0.272s
user    0m1.074s
sys     0m0.004s
```

2）将第 11 行注释，编译后再次运行，结果如下：

```
[root@localhost testdir]# gcc -std=c99 cache2.c -lpthread
[root@localhost testdir]# time ./a.out
real    0m3.391s
user    0m13.386s
sys     0m0.006s
```

2.4 疑问

两个逻辑一模一样的程序，运行时间相差近 12 倍。

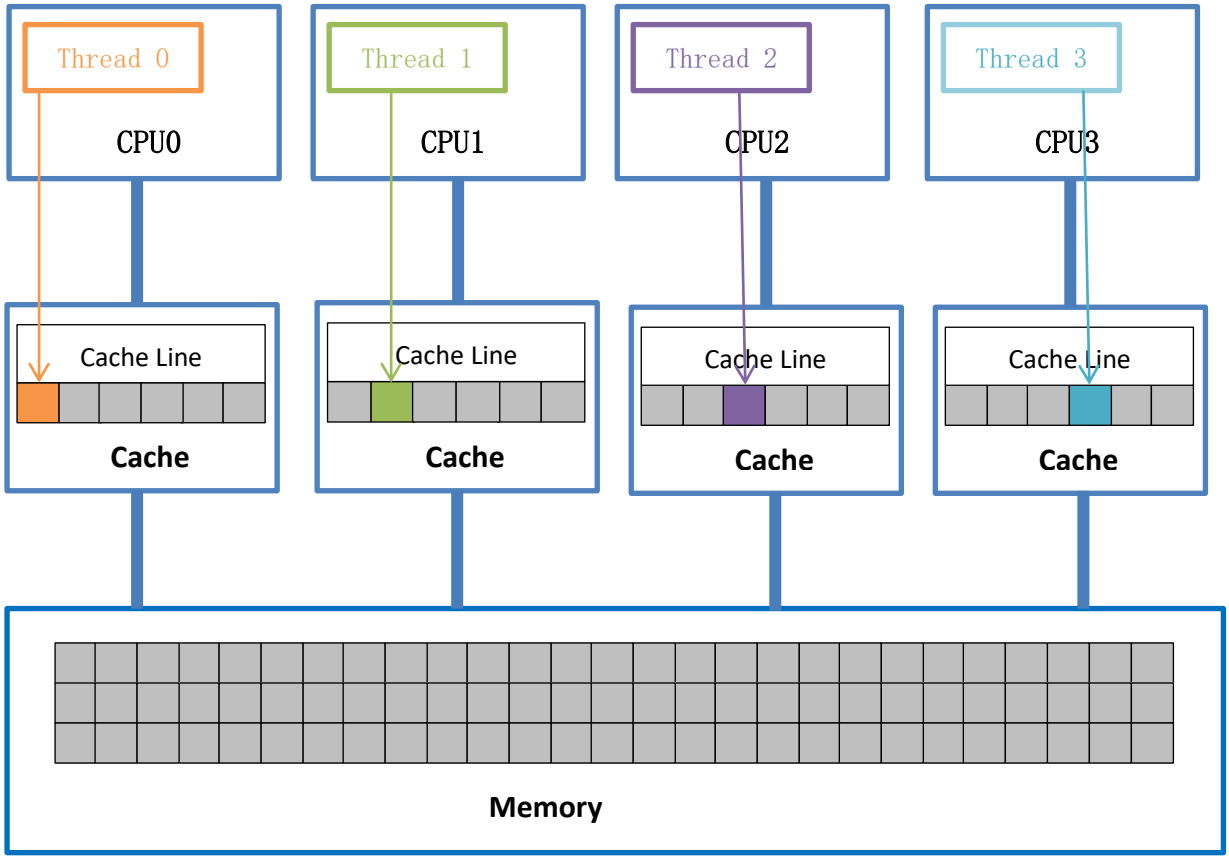
2.5 分析

1） 针对上面的测定结果，有两点需要说明：

A. 当两个不同的 Core 需要操作相同的 cache line 时，会发送 RFO 请求。而从上面的“知识要点”中，我们知道当需要发送大量 RFO 消息时，会对写操作性能带来损耗。

B. 结构体数组在内存布局上也是连续的。

2） 现在我们来分析测定结果差异的原因：（结合下图来分析）



1） Case2 中的 4 个子线程分别运行在不同的 Core 上，Thread 0 操作 data[0]数据，Thread1 操作 data[1]数据，Thread2 操作 data[2]数据，Thread3 操作 data[3]数据。

2） 当数据结构 volatile_int 大小为 4 个字节，Thread 0 读取 data[0]数据时，会把&data[0]地址之后的 64 个字节全部放到 cache line 里（即 data[0]，data[1]，data[2]，data[3]都放到同一个 cache line 里），这样当 ThreadX 读取并修改 data[X] （1<=X<=3）数据时，会轮番发送 RFO 信息，因此程序运行速度慢，性能差。

3） 当数据结构 volatile_int 大小为 64 个字节时（为一个 cache line 大小），Thread 0 读取 data[0]数据时，data[0]数据正好填满一个 cache line。Thread 1 读取 data[1]时，由于 data[1]不在 cache line 中，会同样从内存中把 data[1]数据放到 cache line 中（Thread2 和 Thread3 类似）。这样由于每个 Thread 操作的数据都不在同一个 cache line 里，因此也就不需要发送 RFO 消息，因此程序运行速度快，性能好。

接下来，再用 perf 工具来对比两种不同情况下 L1-dcache-load-misses 情况：

A：结构体 volatile_int 为 64 字节时：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses ./a.out

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

             112,235 L1-dcache-misses

       0.271763162 seconds time elapsed
```

B：结构体 volatile_int 为 4 字节时：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses  ./a.out

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

        70,279,415 L1-dcache-misses

       3.381110320 seconds time elapsed
```

可以看到，当结构体 volatile_int 为 64 字节和 4 字节时，两者的 L1-dcache-load-misses 数相差近 600 倍。

3 associativity（缓存关联性）

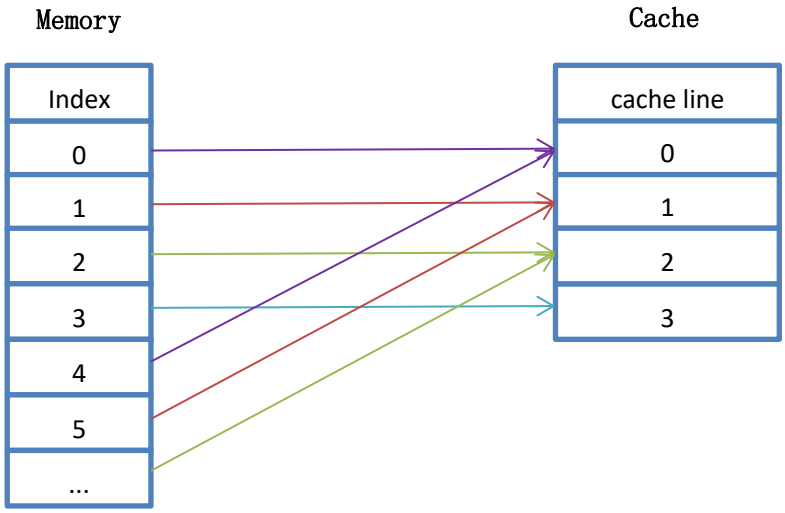
该部分主要介绍缓存关联性的一些基础知识以及对程序性能的影响。

3.1 背景

缓存关联性可通过内存块映射到 Cache 的位置来说明，Cache 的设计方式分为 3 种：

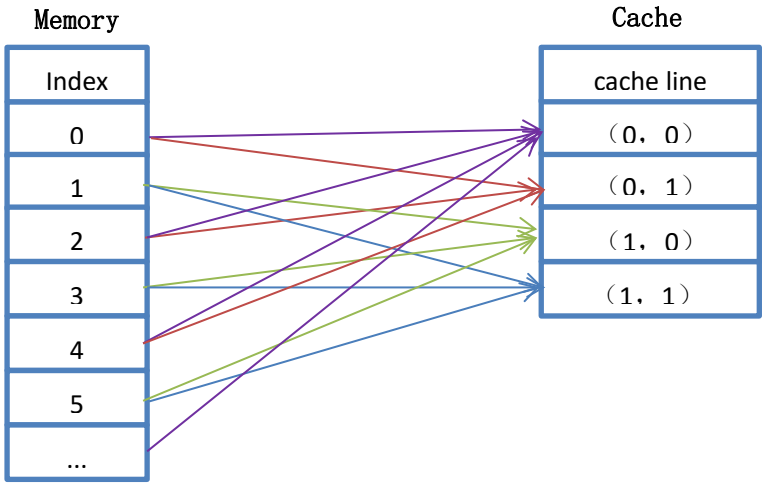
1) 直接映射（Direct mapped cache）：每个内存块只能映射到一个特定的 Cache line 里，被映射到同一个 Cache line 的两个内存块是不能同时放入 Cache line 里的。

直接映射示意图如下所示：



2) N 路组关联（N-way set associative cache）：每个内存块能被映射到 N 个 cache line 中的任何一个（比如一个 8 路组关联，每个内存块能够被映射到 8 个 cache line 中的任何一个），前面介绍的参数 ways_of_associativity 就表示多少路组关联。

2 路组关联映射示意图如下所示：



3) 完全关联(Fully associative cache):每个内存块能够被映射到任意一个 cache line。操作效果上相当于一个散列表。

目前，大多数 CPU 用的都是 N 路组关联映射 Cache，有如下 3 个原因：
1） 直接映射 Cache 容易引发冲突，当多个内存块竞争同一个 cache line 时，它们将互相驱逐对方，导致 cache-miss rate 高。
2） 完全映射 Cache 设计复杂，硬件实现上价格高。
3） N 路组关联映射 Cache 在电路实现简化和 cache-miss rate 低之间取得了良好的折中。

接下来，就以本人机器中的 CPU 来对 Cache 的关联性进行简要说明：

首先，查看 cat /sys/devices/system/cpu/cpu0/cache/index0/目录下的各个文件，得出 L1 Cache 大小为 16KB，为 8 路组关联映射，有 32 个 Cache 子集（注:因为是 8 路组关联映射，所以每个子集里有 8 个 cache line)。

接着，根据以上信息，可以得出，物理地址凡是 2048 字节（32*64，子集数乘以 cache line 占用的字节数）的倍数都将映射到同一个 Cache 子集，而每个子集里只有 8 个 cache line，这样，当超过 8 个内存块同时映射到同一个子集的话，它们将互相驱逐对方。

最后，我们通过一小段代码来说明缓存关联性对程序性能的影响。

3.2 源码

cache3.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5
 6 int main(int argc, char *argv[])
 7 {
 8
 9
10     int span = atoi(argv[1]);
11
12     int *array = (int *) malloc(1024 * 1024 * 100);  //100MB
13     memset(array, 0, 1024 * 1024 * 100);
14
15     long long iteration = 256 * 1024 * 1024;
16     long long array_length = 1024 * 1024 * 100 / sizeof(int) - 1;
17
18     long long loop = 0;
19
20     for (loop; loop < iteration; loop++)
```

```
21          array[(loop * span) % array_length]++;
22
23
24      return 0;
25
```

3.3 结果

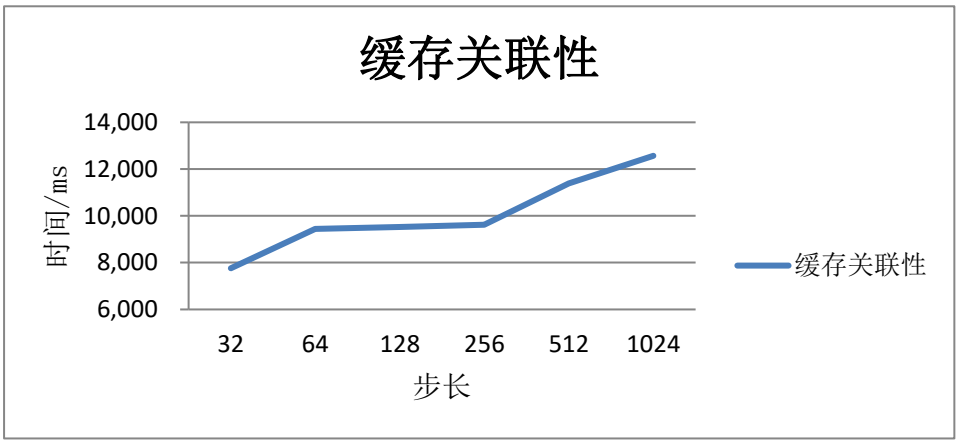
编译后，以 32 为步长运行程序，结果如下：

```
[root@localhost testdir]# gcc -std=c99 cache3.c
[root@localhost testdir]# time ./a.out  32
real    0m7.762s
user    0m7.641s
sys     0m0.031s
```

接着再以 64、128、256、512、1024 为步长运行，对结果整理如下表所示：

步长	32	64	128	256	512	1024
时间	7,762ms	9,445ms	9,529ms	9,615ms	11,388ms	12,569ms

为了更加直观的看出运行时间随步长的变化，根据表的数据制作下图：

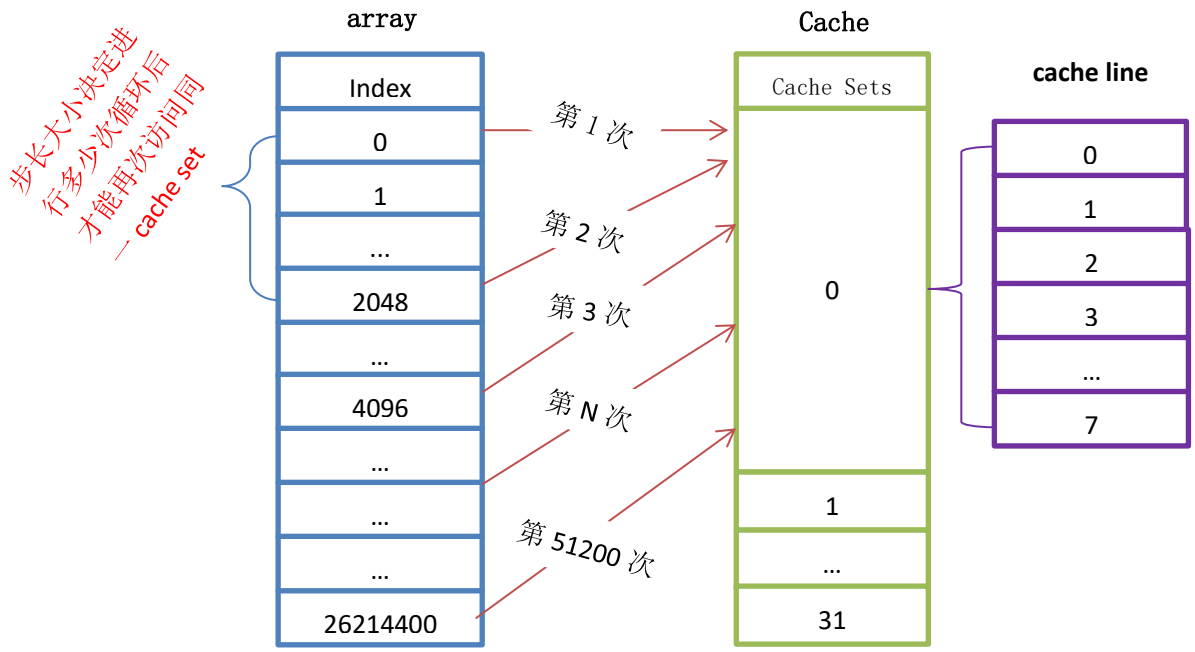


3.4 疑问

由于每次的步长最少为 128 个字节（32*sizeof（int）），大于一个 cache line（64 个字节）的大小，这样每次访问数据都要发生 cache-miss，且不同步长的循环次数一样，运行时间却存在差异。

3.5 分析

结合下图来分析：



1) 步长为 32：Case 中数组大小为 100MB 时，因为 128（32*sizeof（int））个字节能够整除 2048 个字节，所以会存在 51200 个元素（100MB/2048B）竞争同一个 L1 Cache 子集。

2) 步长为 64：Case 中数组大小为 100MB 时，因为 256（64*sizeof（int））个字节能够整除 2048 个字节，所以会存在 51200 个元素（100MB/2048B）竞争同一个 L1 Cache 子集。

3) 但是，步长为 32 的访问到同一个 L1 Cache 子集需要经过 16（2048/128）次循环，步长为 64 的访问到同一个 Cache 子集需要经过 8(2048/128)次循环。那么当循环数为 2^28 时，步长为 32 的访问了 2^24 次存在竞争的 Cache 子集，而步长为 64 的访问了 2^25 次存在竞争的 Cache 子集，竞争数整整是步长 32 的 2 倍。步长为其他大小时类似，即在相同循环次数下，步长越小，访问存在竞争 Cache 子集的次数越少，运行速度越快。

我们再用 perf 工具对步长为 32 和 512 的 L1-dcache-load-misses 数进行统计：

步长为 32：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses  ./a.out  32
 Performance counter stats for './a.out 32':

         271173354 L1-dcache-load-misses

       7.738250970 seconds time elapsed
```

步长为 512：

```
[root@localhost testdir]# perf stat -e L1-dcache-load-misses  ./a.out  512
 Performance counter stats for './a.out 512':

         270674902 L1-dcache-load-misses

      11.389850021 seconds time elapsed
```

可以看出，L1 cache-misses 数与之前的分析是吻合的。由于每次访问数据步长都大于 64 个字节，于是每次数据访问都会出现 L1 cache-miss，即步长为 32 和 512 的 L1 cache-misses 数一样多。

这里也可以看出， L1 cache-misses 数不是决定程序性能的唯一标准，即相同程序 L1 cache-misses 数一样多，程序的性能也会存在差异。

总结

1） Case1 主要讨论了 cache-miss 产生的时机以及对性能的影响。在实际编程中，是会经常会遇到的，合理的改善代码将会提高程序的性能。

2） Case2 主要讨论了伪共享在多核编程中的应用。在实际编程中，遇到的比较多，在内存大小充沛的情况下可以利用此特性来提高程序性能。

3） Case3 主要讨论了 Cache 的关联性。在实际编程中应该不会考虑该特性的，但是 Cache 的关联性理解起来有趣而且确能够被证实。