窥探CPU Cache

2014/10/12

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](http://msdn.microsoft.com/en-us/library/ms683194(VS.85).aspx) 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：
2. 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。
3. 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。
4. 现在来分析测定结果差异的原因：（结合下图来分析）

**array**

1

3

0

2

cache line

**Cache**

...

1

3

0

2

cache line

**Cache**

...

**array**

**按行循环**

**按列循环**

Index

（0，0）

（0，10239）

（1，0）

Index

（0，0）

（0，1）

（0，1）

...

）

...

）

（0，15）

（0，16）

...

（0，10239）

（2，0）

...

...

1. 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)）。
2. 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：

Memory

Socket 0

L3 Cache

L1 Data

cache

L1 Instruction

Cache

Logical Processor 0

Logical Processor 1

L2 cache

Core 0

z

Core 1

L2 cache

L1 Data

cache

L1 Instruction

Cache

Logical Processor 0

Logical Processor 1

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。

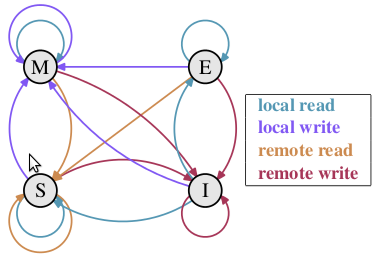
注：可以通过前面介绍的/sys/devices/system/cpu/cpu0/cache/index\*文件下的shared\_cpu\_list参数来验证。

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：

1. 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)
2. 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
3. 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
4. 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

1. **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.
2. **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：
2. 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。
3. 结构体数组在内存布局上也是连续的。
4. 现在我们来分析测定结果差异的原因：（结合下图来分析）

**CPU0**

Thread 0

**CPU1**

Thread 1

**CPU3**

Thread 3

**CPU2**

Thread 2

**Cache**

Cache Line

**Cache**

Cache Line

**Cache**

Cache Line

**Cache**

Cache Line

**Memory**

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：

0

1

2

3

4

5

1

...

Index

3

0

2

cache line

**Memory**

**Cache**

1. 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：

**Memory**

**Cache**

Index

cache line

0

（0，0）

1

（0，1）

2

（1，0）

3

（1，1）

4

5

...

1. 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; |

## 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：

**array**

**Cache**

Cache Sets

Index

**cache line**

步长大小决定进行多少次循环后才能再次访问同一cache set

0

0

第1次

0

1

1

...

）

第2次

2

）

7

2048

第3次

次

次

3

…

…

4096

第N次

次

次

…

1

…

第51200次

次

次

次

…

…

26214400

31

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。