Cache False Sharing Debug

> 本文首发于内核月谈微信号，由吴一昊，杨勇共同完成; 转载时请包含原文或者作者网站链接：http://oliveryang.net

1. 关于本文

本文基于 Joe Mario 的一篇博客 改编而成。 Joe Mario 是 Redhat 公司的 Senior Principal Software Engineer，在系统的性能优化领域颇有建树，他也是本文描述的 `perf c2c` 工具的贡献者之一。 这篇博客行文比较口语化，且假设读者对 CPU 多核架构，Cache Memory 层次结构，以及 Cache 的一致性协议有所了解。 故此，笔者决定放弃照翻原文，并且基于原博客文章做了一些扩展，增加了相关背景知识简介。 本文中若有任何疏漏错误，责任在于编译者。有任何建议和意见，请回复内核月谈微信公众号，或通过 oliver.yang at linux.alibaba.com 反馈。

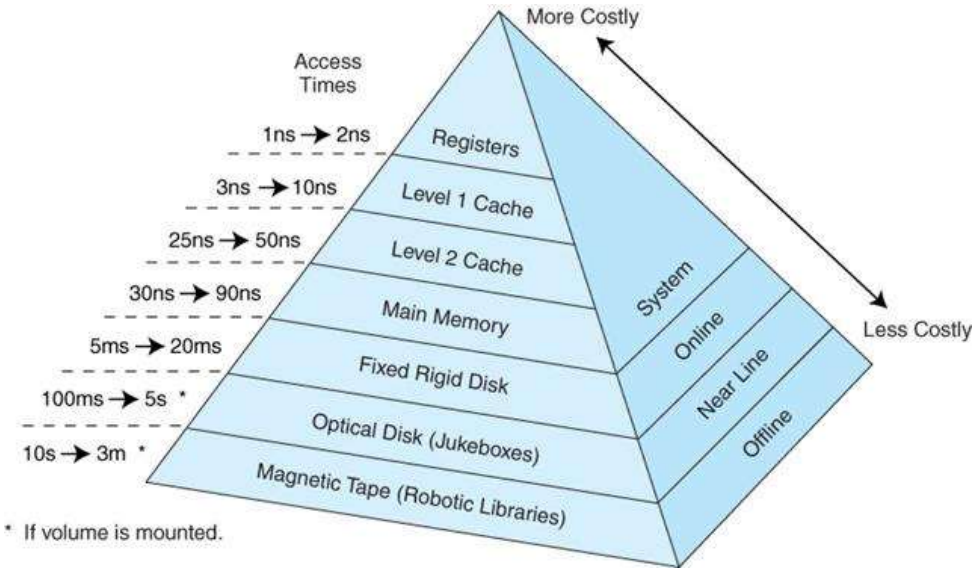
2. 背景知识

要搞清楚 Cache Line 伪共享的概念及其性能影响，需要对现代理器架构和硬件实现有一个基本的了解。 如果读者已经对这些概念已经有所了解，可以跳过本小节，直接了解 perf c2c 发现 Cache Line 伪共享的方法。 (注：本节中的所有图片，均来自与 Google 图片搜索，版权归原作者所有。)

2.1 存储器层次结构

众所周知，现代计算机体系结构，通过存储器层次结构 (Memory Hierarchy) 的设计，使系统在性能，成本和制造工艺之间作出取舍，从而达到一个平衡。 下图给出了不同层次的硬件访问

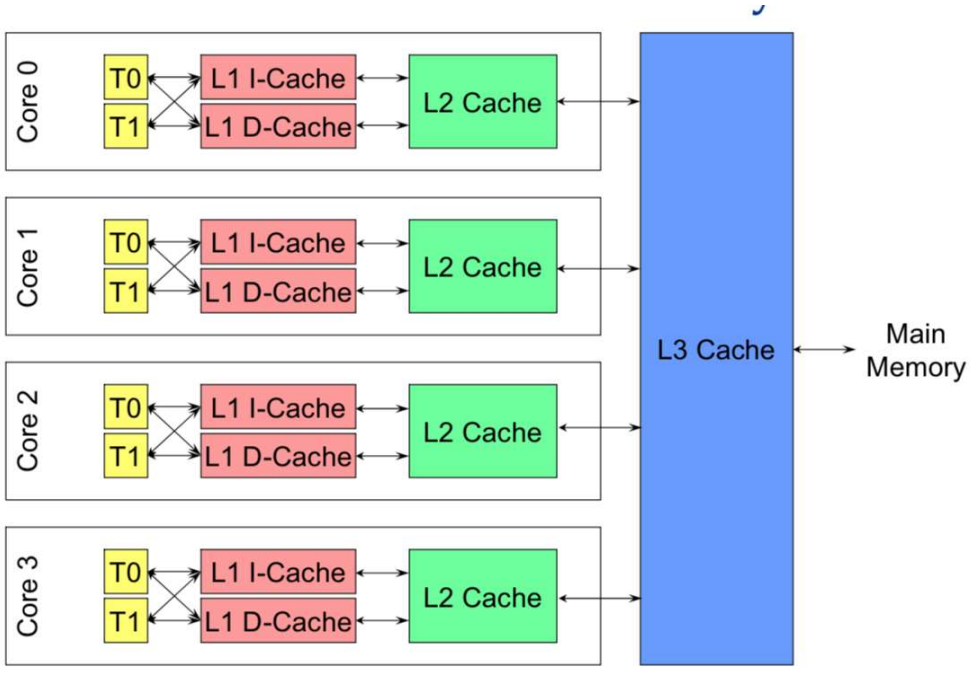
延迟，可以看到，各个层次硬件访问延迟存在数量级上的差异，越高的性能，往往意味着更高的成本和更小的容量：



通过上图，可以对各级存储器 Cache Miss 带来的性能惩罚有个大致的概念。

2.2 多核架构

随着多核架构的普及，对称多处理器 (SMP) 系统成为主流。例如，一个物理 CPU 可以存在多个物理 Core，而每个 Core 又可以存在多个硬件线程。 x86 以下图为例，1 个 x86 CPU 有 4 个物理 Core，每个 Core 有两个 HT (Hyper Thread)，

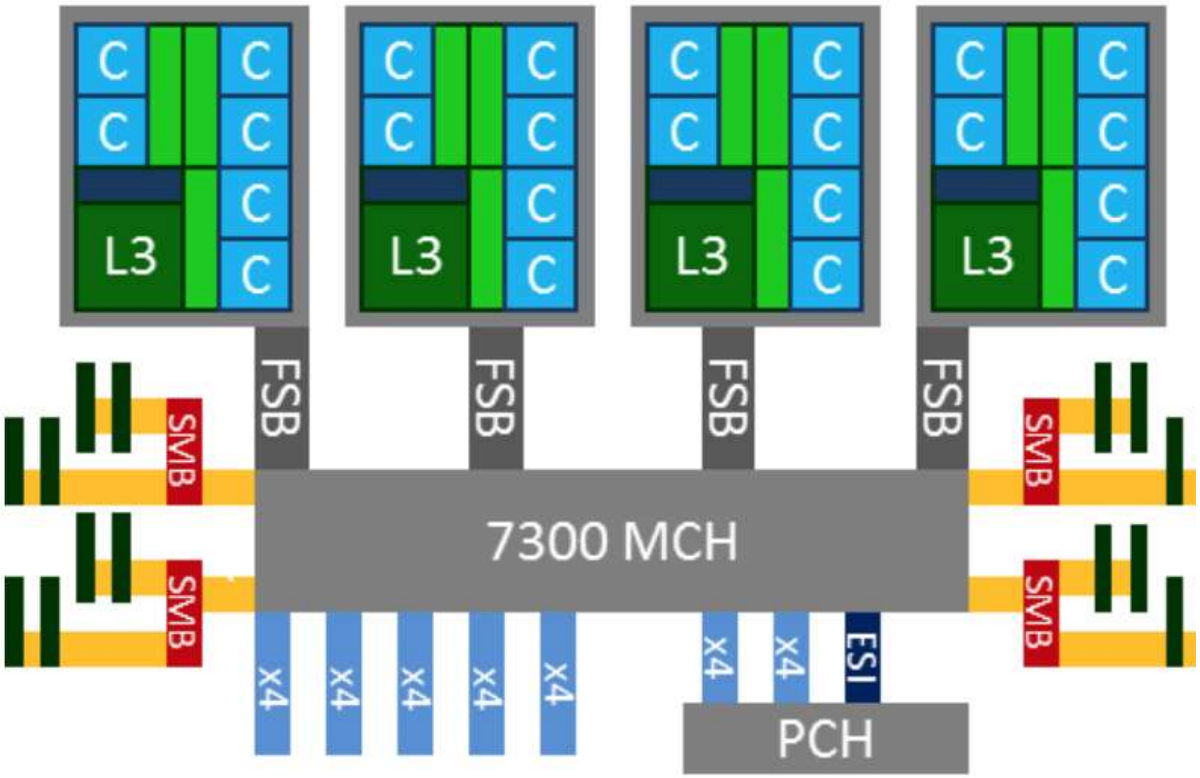


从硬件的角度，上图的 L1 和 L2 Cache 都被两个 HT 共享，且在同一个物理 Core。而 L3 Cache 则在物理 CPU 里，被多个 Core 来共享。 而从 OS 内核角度，每个 HT 都是一个逻辑 CPU，因此，这个处理器在 OS 来看，就是一个 8 个 CPU 的 SMP 系统。

2.3 NUMA 架构

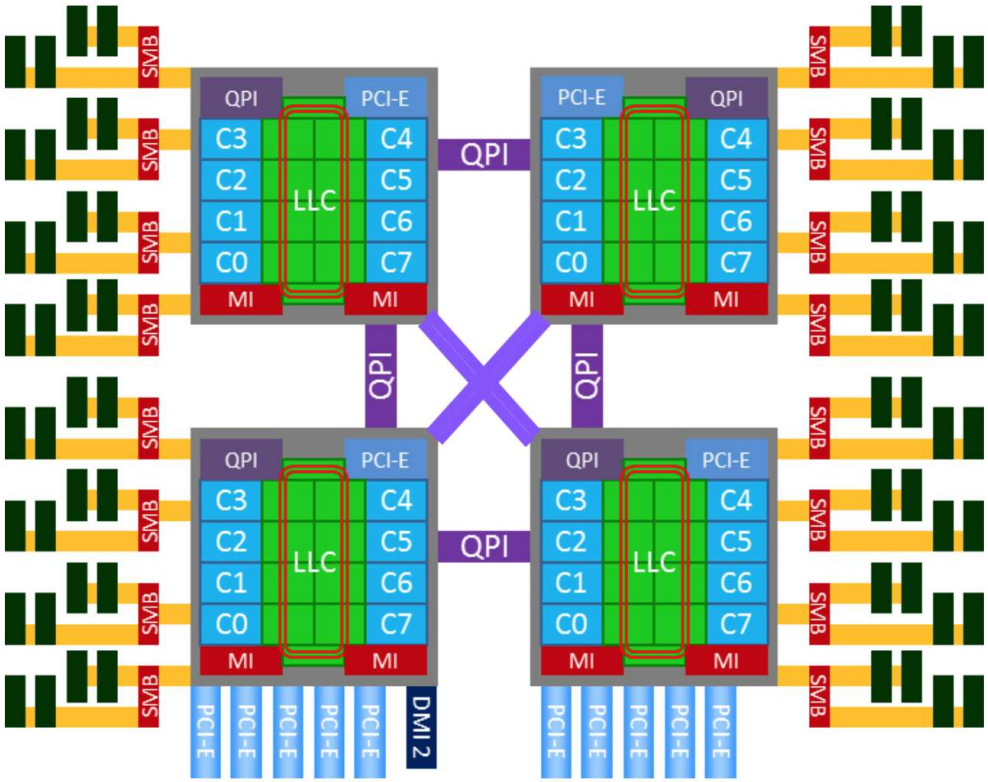
一个 SMP 系统，按照其 CPU 和内存的互连方式，可以分为 UMA (均匀内存访问) 和 NUMA (非均匀内存访问) 两种架构。 其中，在多个物理 CPU 之间保证 Cache 一致性的 NUMA 架构，又被称做 ccNUMA (Cache Coherent NUMA) 架构。

以 x86 为例，早期的 x86 就是典型的 UMA 架构。例如下图，四路处理器通过 FSB (前端系统总线) 和主板上的内存控制器芯片 (MCH) 相连，DRAM 是以 UMA 方式组织的，延迟并无访问差异，



然而，这种架构带来了严重的内存总线的性能瓶颈，影响了 x86 在多路服务器上的可扩展性和性能。

因此，从 Nehalem 架构开始，x86 开始转向 NUMA 架构，内存控制器芯片被集成到处理器内部，多个处理器通过 QPI 链路相连，从此 DRAM 有了远近之分。 而 Sandybridge 架构则更近一步，将片外的 IOH 芯片也集成到了处理器内部，至此，内存控制器和 PCIe Root Complex 全部在处理器内部了。 下图就是一个典型的 x86 的 NUMA 架构：



由于 NUMA 架构的引入，以下主要部件产生了因物理链路的远近带来的延迟差异：

- Cache

  除物理 CPU 有本地的 Cache 的层级结构以外，还存在跨越系统总线（QPI）的远程 Cache 命中访问的情况。需要注意的是，远程的 Cache 命中，对发起 Cache 访问的 CPU 来说，还是被记入了 LLC Cache Miss。

- DRAM

  在两路及以上的服务器，远程 DRAM 的访问延迟，远远高于本地 DRAM 的访问延迟，有些系统可以达到 2 倍的差异。 需要注意的是，即使服务器 BIOS 里关闭了 NUMA 特性，也只是对 OS 内核屏蔽了这个特性，这种延迟差异还是存在的。

- Device

  对 CPU 访问设备内存，及设备发起 DMA 内存的读写活动而言，存在本地 Device 和远程 Device 的差别，有显著的延迟访问差异。

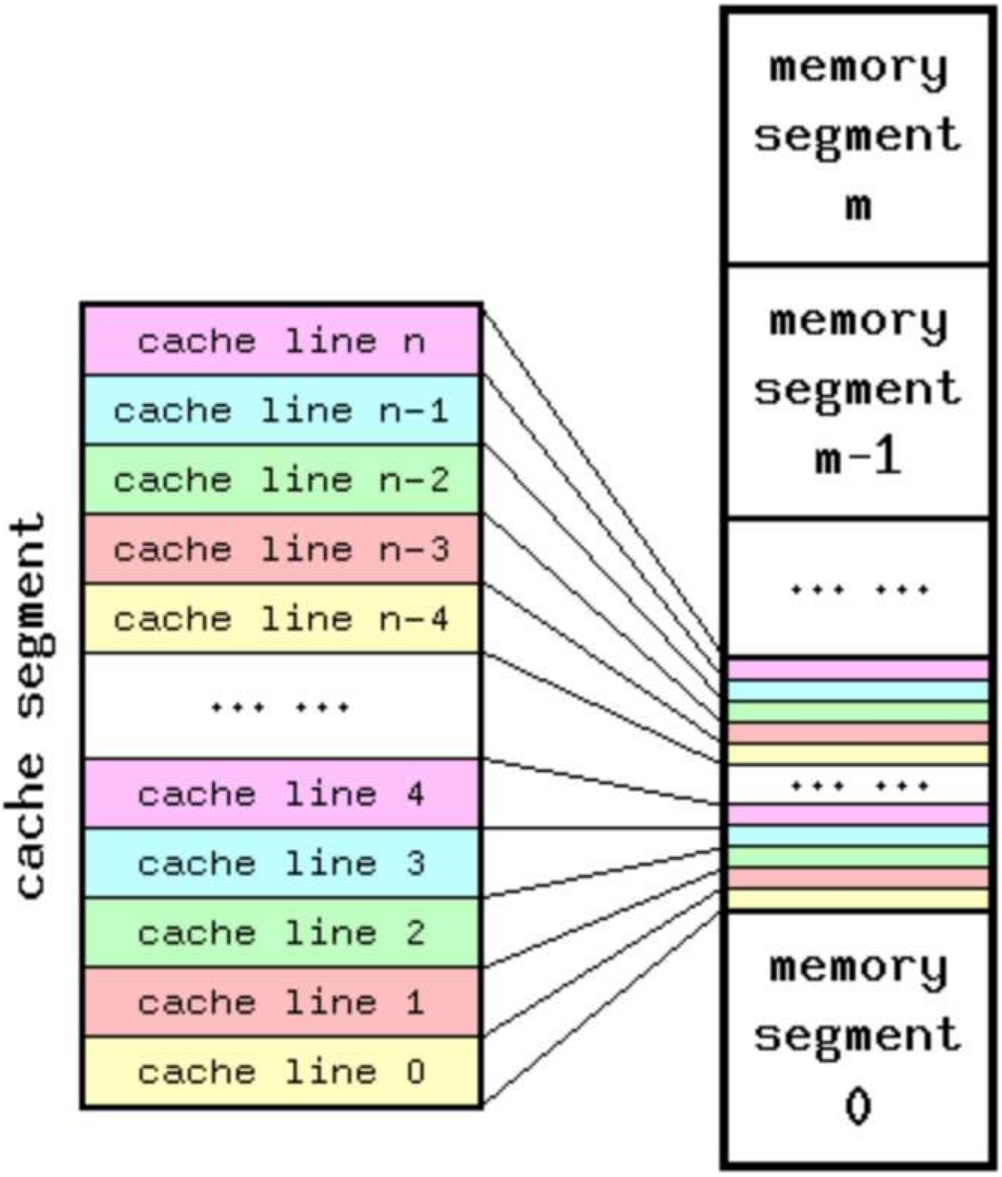
因此，对以上 NUMA 系统，一个 NUMA 节点通常可以被认为是一个物理 CPU 加上它本地的 DRAM 和 Device 组成。那么，四路服务器就拥有四个 NUMA 节点。 如果 BIOS 打开了 NUMA 支持，Linux 内核则会根据 ACPI 提供的表格，针对 NUMA 节点做一系列的 NUMA 亲和性的优化。

在 Linux 上，`numactl --hardware` 可以返回当前系统的 NUMA 节点信息，特别是 CPU 和 NUMA 节点的对应信息。

2.4 Cache Line

Cache Line 是 CPU 和主存之间数据传输的最小单位。当一行 Cache Line 被从内存拷贝到 Cache 里，Cache 里会为这个 Cache Line 创建一个条目。 这个 Cache 条目里既包含了拷贝的内存数据，即 Cache Line，又包含了这行数据在内存里的位置等元数据信息。

由于 Cache 容量远远小于主存，因此，存在多个主存地址可以被映射到同一个 Cache 条目的情况，下图是一个 Cache 和主存映射的概念图：



而这种 Cache 到主存的映射，通常是由内存的虚拟或者物理地址的某几位决定的，取决于 Cache 硬件设计是虚拟地址索引，还是物理地址索引。 然而，由于索引位一般设计为低地址位，通常在物理页的页内偏移以内，因此，不论是内存虚拟或者物理地址，都可以拿来判断两个内存地址，是否在同一个 Cache Line 里。

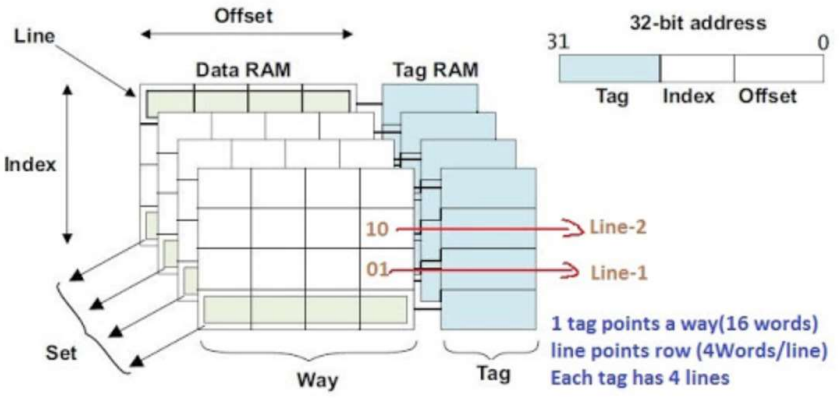
Cache Line 的大小和处理器硬件架构有关。在 Linux 上，通过 `getconf` 就可以拿到 CPU 的 Cache Line 的大小，

```
$getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  15728640
LEVEL3_CACHE_ASSOC                 20
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

2.5 Cache 的结构

前面 Linux `getconf` 命令的输出，除了 `*_LINESIZE` 指示了系统的 Cache Line 的大小是 64 字节外，还给出了 Cache 类别，大小。 其中 `*_ASSOC` 则指示了该 Cache 是几路关联 (Way Associative) 的。

下图很好的说明了 Cache 在 CPU 里的真正的组织结构，



This means that the cache controller will use two bits of the address (bits [3:2]) as the offset to select a word within the line and two bits of the address (bits [5:4]) as the index to select one of the four available lines. The remaining bits of the address (bits [31:6]) will be stored as a tag value.

一个主存的物理或者虚拟地址，可以被分成三部分：高地址位当作 Cache 的 Tag，用来比较选中多路 (Way) Cache 中的某一路 (Way)，而低地址位可以做 Index，用来选中某一个 Cache Set。 在某些架构上，最低的地址位，Block Offset 可以选中在某个 Cache Line 中的某一部份。

因此，Cache Line 的命中，完全依靠地址里的 Tag 和 Index 就可以做到。关于 Cache 结构里的 Way，Set，Tag 的概念，请参考相关文档或者资料。这里就不再赘述。

2.6 Cache 一致性

如前所述，在 SMP 系统里，每个 CPU 都有自己本地的 Cache。因此，同一个变量，或者同一行 Cache Line，有在多个处理器的本地 Cache 里存在多份拷贝的可能性，因此就存在数据一致性问题。 通常，处理器都实现了 Cache 一致性 (Cache Coherence) 协议。如历史上 x86 曾实现了 MESI 协议， 以及 MESIF 协议。

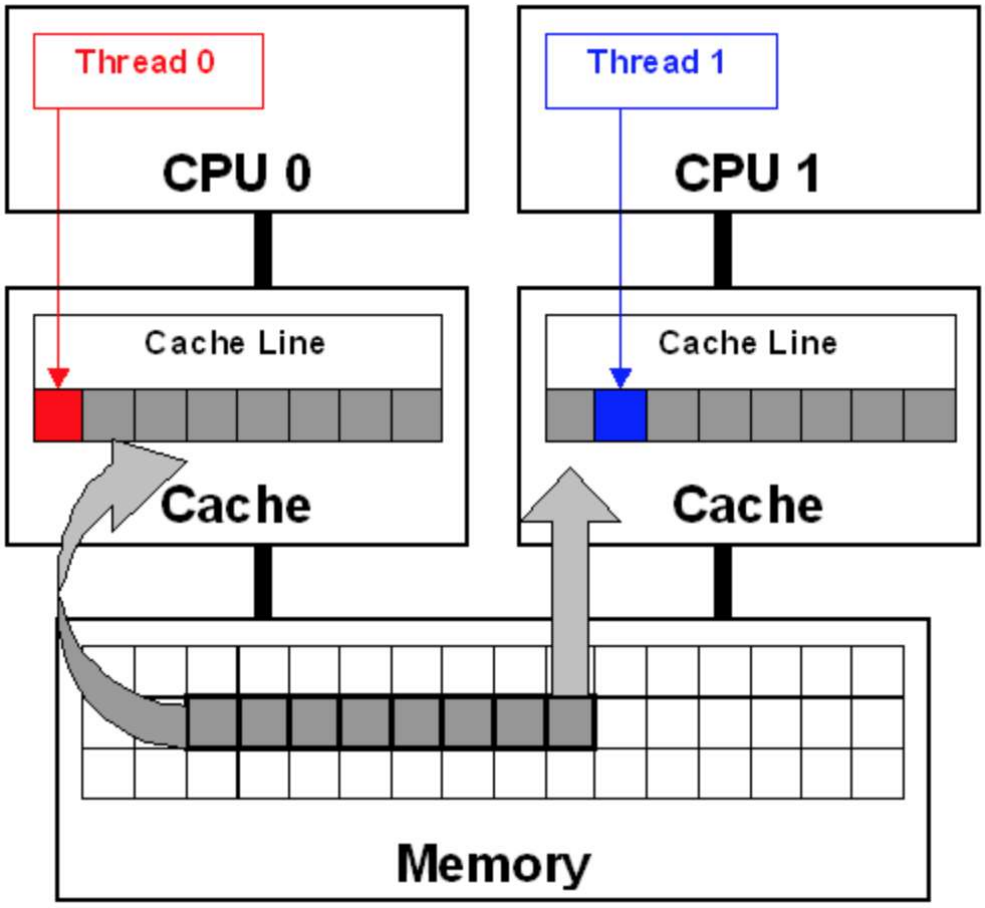
假设两个处理器 A 和 B, 都在各自本地 Cache Line 里有同一个变量的拷贝时，此时该 Cache Line 处于 `Shared` 状态。当处理器 A 在本地修改了变量，除去把本地变量所属的 Cache Line 置为 `Modified` 状态以外， 还必须在另一个处理器 B 读同一个变量前，对该变量所在的 B 处理器本地 Cache Line 发起 Invaidate 操作，标记 B 处理器的那条 Cache Line 为 `Invalidate` 状态。 随后，若处理器 B 在对变量做读写操作时，如果遇到这个标记为 `Invalidate` 的状态的 Cache Line，即会引发 Cache Miss， 从而将内存中最新的数据拷贝到 Cache Line 里，然后处理器 B 再对此 Cache Line 对变量做读写操作。

本文中的 Cache Line 伪共享场景，就基于上述场景来讲解，关于 Cache 一致性协议更多的细节，请参考相关文档。

2.7 Cache Line 伪共享

Cache Line 伪共享问题，就是由多个 CPU 上的多个线程同时修改自己的变量引发的。这些变量表面上是不同的变量，但是实际上却存储在同一条 Cache Line 里。 在这种情况下，由于 Cache 一致性协议，两个处理器都存储有相同的 Cache Line 拷贝的前提下，本地 CPU 变量的修改会导致本地 Cache Line 变成 `Modified` 状态，然后在其它共享此 Cache Line 的 CPU 上， 引发 Cache Line 的 Invaidate 操作，导致 Cache Line 变为 `Invalidate` 状态，从而使 Cache Line 再次被访问时，发生本地 Cache Miss，从而伤害到应用的性能。 在此场景下，多个线程在不同的 CPU 上高频反复访问这种 Cache Line 伪共享的变量，则会因 Cache 颠簸引发严重的性能问题。

下图即为两个线程间的 Cache Line 伪共享问题的示意图，



3. Perf c2c 发现伪共享

当应用在 NUMA 环境中运行，或者应用是多线程的，又或者是多进程间有共享内存，满足其中任意一条，那么这个应用就可能因为 Cache Line 伪共享而性能下降。

但是，要怎样才能知道一个应用是不是受伪共享所害呢？Joe Mario 提交的 patch 能够解决这个问题。Joe 的 patch 是在 Linux 的著名的 perf 工具上，添加了一些新特性，叫做 c2c， 意思是“缓存到缓存” (cache-2-cache)。

Redhat 在很多 Linux 的大型应用上使用了 c2c 的原型，成功地发现了很多热的伪共享的 Cache Line。 Joe 在博客里总结了一下 `perf c2c` 的主要功能：

- 发现伪共享的 `Cache Line`
- 谁在读写上述的 `Cache Line`，以及访问发生处的 `Cache Line` 的内部偏移
- 这些读者和写者分别的 `pid`，`tid`，指令地址，函数名，二进制文件
- 每个读者和写者的源代码文件，代码行号
- 这些热点 `Cache Line` 上的，`load` 操作的平均延迟

- 这些 Cache Line 的样本来自哪些 NUMA 节点， 由哪些 CPU 参与了读写

perf c2c 和 perf 里现有的工具比较类似:

- 先用 perf c2c record 通过采样，收集性能数据
- 再用 perf c2c report 基于采样数据，生成报告

如果想了解 perf c2c 的详细使用，请访问: PERF-C2C(1)

这里还有一个完整的 perf c2c 的输出的样例。

最后，还有一个小程序的源代码，可以产生大量的 Cache Line 伪共享，用以测试体验: Fasle sharing .c src file

3.1 perf c2c 的输出

下面，让我们就之前给出的 perf c2c 的输出样例，做一个详细介绍。

输出里的第一个表，概括了 CPU 在 perf c2c 数据采样期间做的 load 和 store 的样本。能够看到 load 操作都是在哪里取到了数据。

在 perf c2c 输出里，HITM 意为 “Hit In The Modified”，代表 CPU 在 load 操作命中了一条标记为 Modified 状态的 Cache Line。如前所述，伪共享发生的关键就在于此。

而 Remote HITM，意思是跨 NUMA 节点的 HITM，这个是所有 load 操作里代价最高的情况，尤其在读者和写者非常多的情况下，这个代价会变得非常的高。

对应的，Local HITM，则是本地 NUMA 节点内的 HITM，下面是对 perf c2c 输出的详细注解:

```
=================================================
            Trace Event Information
=================================================
  Total records                     :     329219  >> 采样到的 CPU load 和 store
  Locked Load/Store Operations      :      14654
  Load Operations                   :      69679  >> CPU load 操作的样本总数
  Loads - uncacheable               :          0
  Loads - IO                        :          0
  Loads - Miss                      :       3972
  Loads - no mapping                :          0
  Load Fill Buffer Hit              :      11958  >> Load 操作没有命中 L1 Cache,
  Load L1D hit                      :      17235  >> Load 操作命中 L1 Dcache 的
  Load L2D hit                      :         21  >> Load 操作命中 L2 Dcache 的
  Load LLC hit                      :      14219  >> Load 操作命中最后一级 (LLC)
  Load Local HITM                   :       3402  >> Load 操作命中了本地 NUMA 节
  Load Remote HITM                  :      12757  >> Load 操作命中了远程 NUMA 节
  Load Remote HIT                   :       5295  >> Load 操作命中了远程未修改的
  Load Local DRAM                   :        976  >> Load 操作命中了本地 NUMA 节
  Load Remote DRAM                  :       3246  >> Load 操作命中了远程 NUMA 节
  Load MESI State Exclusive         :       4222  >> Load 操作命中 MESI 状态中，处
  Load MESI State Shared            :          0  >> Load 操作命中 MESI 状态中，处
  Load LLC Misses                   :      22274  >> Load 操作产生的本地 NUMA 节
  LLC Misses to Local DRAM          :       4.4% >> Load 操作产生的 LLC Cache Mis
  LLC Misses to Remote DRAM         :      14.6% >> Load 操作产生的 LLC Cache Mis
  LLC Misses to Remote cache (HIT)  :      23.8% >> Load 操作产生的 LLC Cache Miss
  LLC Misses to Remote cache (HITM) :      57.3% >> Load 操作产生的 LLC Cache Miss
  Store Operations                  :     259539  >> CPU store 操作的样本总数
  Store - uncacheable               :          0
  Store - no mapping                :         11
  Store L1D Hit                     :     256696  >> Store 操作命中 L1 Dcache 的
  Store L1D Miss                    :       2832  >> Store 操作命中 L1 Dcache Mis
  No Page Map Rejects               :       2376
  Unable to parse data source       :          1
```

perf c2c 输出的第二个表, 以 Cache Line 维度，全局展示了 CPU load 和 store 活动的情况。 这个表的每一行是一条 Cache Line 的数据，显示了发生伪共享最热的一些 Cache Line。默认按照发生 Remote HITM 的次数比例排序，改下参数也可以按照发生 Local HITM 的次数比例排序。

要检查 Cache Line 伪共享问题，就在这个表里找 `Rmt LLC Load HITM`（即跨 NUMA 节点缓存里取到数据的）次数比较高的，如果有，就得深挖一下。

```
=================================================
            Shared Data Cache Line Table
=================================================
#
#                                        Total        Rmt  ----- LLC Load Hitm -----   ---- .
# Index             Cacheline  records     Hitm      Total       Lcl       Rmt     Tota .
# .....  ..................  .......  .......  .......  .......  .......  .......  .
#
     0           0x602180    149904   77.09%     12103     2269     9834    10950
     1           0x602100     12128   22.20%      3951     1119     2832
     2  0xffff883ffb6a7e80       260    0.09%        15        3       12       1(
     3  0xffffffff81aec000       157    0.07%         9        0        9
     4  0xffffffff81e3f540       179    0.06%         9        1        8        1
```

下面是共享 Cache Line 的 Pareto 百分比分布表，命名取自帕累托法则 (Pareto principle)，即 2/8 法则的喻义，显示了每条内部产生竞争的 Cache Line 的百分比分布的细目信息。 这是最重要的一个表。为了精简，这里只展示了三条 Cache Line 相关的记录，表格里包含了这些信息：

- 其中 71,72 行是列名，每列都解释了Cache Line的一些活动。
- 标号为 76，85，91 的行显示了每条 Cache Line 的 HITM 和 store 活动情况：依次是 CPU load 和 store 活动的计数，以及 Cache Line 的虚拟地址。
- 78 到 83 行，是针对 76 行 Cache Line 访问的细目统计，具体格式如下：
    - 首先是百分比分布，包含了 HITM 中 remote 和 local 的百分比，store 里的 L1 Hit 和 Miss 的百分比分布。注意，这些百分比纵列相加正好是 100%。
    - 然后是数据地址列。上面提到了 76 行显示了 Cache Line 的虚拟地址，而下面几行的这一列则是行内偏移。
    - 下一列显示了pid，或线程id（如果设置了要输出tid）。
    - 接下来是指令地址。
    - 接下来三列，展示了平均load操作的延迟。我常看着里有没有很高的平均延迟。这个平均延迟，可以反映该行的竞争紧张程度。
    - cpu cnt列展示了该行访问的样本采集自多少个cpu。
    - 然后是函数名，二进制文件名，源代码名，和代码行数。
    - 最后一列展示了对于每个节点，样本分别来自于哪些cpu

以下为样例输出：

```
=================================================
      Shared Cache Line Distribution Pareto
=================================================
#
#        ----- HITM -----  -- Store Refs --        Data address
#   Num      Rmt      Lcl   L1 Hit  L1 Miss              Offset      Pid
# .....  .......  .......  .......  .......  ..................  .......  ..........
#
  ------------------------------------------------------------------------------
      0     9834     2269   109036      468              0x602180
  ------------------------------------------------------------------------------
           65.51%   55.88%   75.20%    0.00%                 0x0    14604
            0.41%    0.35%    0.00%    0.00%                 0x0    14604
            0.00%    0.00%   24.80%  100.00%                 0x0    14604
            7.50%    9.92%    0.00%    0.00%                0x20    14604
           17.61%   20.89%    0.00%    0.00%                0x28    14604
            8.97%   12.96%    0.00%    0.00%                0x30    14604

  ------------------------------------------------------------------------------
      1     2832     1119        0        0              0x602100
  ------------------------------------------------------------------------------
           29.13%   36.19%    0.00%    0.00%                0x20    14604
           43.68%   34.41%    0.00%    0.00%                0x28    14604
           27.19%   29.40%    0.00%    0.00%                0x30    14604

  ------------------------------------------------------------------------------
      2       12        3      161        0  0xffff883ffb6a7e80
  ------------------------------------------------------------------------------
           58.33%  100.00%    0.00%    0.00%                 0x0    14604  0xffff
           16.67%    0.00%   98.76%    0.00%                 0x0    14604  0xffff
           16.67%    0.00%    0.00%    0.00%                 0x0    14604  0xffff
            8.33%    0.00%    0.00%    0.00%                 0x8    14604  0xffff
            0.00%    0.00%    1.24%    0.00%                 0x8    14604  0xffff
```

3.2 如何用 perf c2c

下面是常见的 `perf c2c` 使用的命令行：

```
# perf c2c record -F 60000 -a --all-user sleep 5
# perf c2c record -F 60000 -a --all-user sleep 3      // 采样较短时间
# perf c2c record -F 60000 -a --all-kernel sleep 3    // 只采样内核态样本
# perf c2c record -F 60000 -a -u --ldlat 50 sleep 3   // 或者只采集 load 延迟大于 60 个
```

熟悉 `perf` 的读者可能已经注意到，这里的 `-F` 选项指定了非常高的采样频率: 60000。请特别注意：这个采样频率不建议在线上或者生产环境使用，因为这会在高负载机器上带来不可预知的影响。 此外 `perf c2c` 对 CPU load 和 store 操作的采样会不可避免的影响到被采样应用的性能，因此建议在研发测试环境使用 `perf c2c` 去优化应用。

对采样数据的分析，可以使用带图形界面的 tui 来看输出，或者只输出到标准输出

```
# perf report -NN -c pid,iaddr                  // 使用tui交互式界面
# perf report -NN -c pid,iaddr --stdio          // 或者输出到标准输出
# perf report -NN -d lcl -c pid,iaddr --stdio   // 或者按 local time 排序
```

默认情况，为了规范输出格式，符号名被截断为定长，但可以用 “–full-symbols” 参数来显示完整符号名。

例如：

```
# perf c2c report -NN -c pid,iaddr --full-symbols --stdio
```

3.3 找到 Cache Line 访问的调用栈

有的时候，很需要找到读写这些 Cache Line 的调用者是谁。下面是获得调用图信息的方法。但一开始，一般不会一上来就用这个，因为输出太多，难以定位伪共享。一般都是先找到问题，再回过头来使用调用图。

```
# perf c2c record --call-graph dwarf,8192 -F 60000 -a --all-user sleep 5
# perf c2c report -NN -g --call-graph -c pid,iaddr --stdio
```

3.4 如何增加采样频率

为了让采样数据更可靠，会把 `perf` 采样频率提升到 `-F 60000` 或者 `-F 80000`，而系统默认的采样频率是 1000。

提升采样频率，可以短时间获得更丰富，更可靠的采样集合。想提升采样频率，可以用下面的方法。 可以根据 `dmesg` 里有没有 `perf interrupt took too long …` 信息来调整频率。注意，如前所述，**这有一定风险，严禁在线上或者生产环境使用。**

```
# echo     500 > /proc/sys/kernel/perf_cpu_time_max_percent
# echo  100000 > /proc/sys/kernel/perf_event_max_sample_rate
```

然后运行前面讲的 `perf c2c record` 命令。之后再运行，

```
# echo      50 > /proc/sys/kernel/perf_cpu_time_max_percent
```

3.5 如何让避免采样数据过量

在大型系统上（比如有 4,8,16 个物理 CPU 插槽的系统）运行 `perf c2c`，可能会样本太多，消耗大量的CPU时间，perf.data文件也可能明显变大。 对于这个问题，有以下建议（包含但不仅限于）：

- 将 `ldlat` 从 `30` 增加大到 `50`。这使得 `perf` 跳过没有性能问题的 `load` 操作。
- 降低采样频率。
- 缩短 `perf record` 的睡眠时间窗口。比如，从 `sleep 5` 改成 `sleep 3`。

3.6 使用 c2c 优化应用的收获

一般搭建看见性能工具的输出，都会问这些数据意味着什么。Joe 总结了他使用 `c2c` 优化应用时，学到的东西，

- `perf c2c` 采样时间不宜过长。Joe 建议运行 `perf c2c 3` 秒、`5` 秒或 `10` 秒。运行更久，观测到的可能就不是并发的伪共享，而是时间错开的 `Cache Line` 访问。
- 如果对内核样本没有兴趣，只想看用户态的样本，可以指定 `--all-user`。反之使用 `--all-kernel`。
- CPU 很多的系统上（如 `>148` 个），设置 `-ldlat` 为一个较大的值（`50` 甚至 `70`），`perf` 可能能产生更丰富的C2C样本。
- 读最上面那个具有概括性的 `Trace Event` 表，尤其是 `LLC Misses to Remote cache HITM` 的数字。只要不是接近 `0`，就可能有值得追究的伪共享。
- 看 `Pareto` 表时，需要关注的，多半只是最热的两三个 `Cache Line`。
- 有的时候，一段代码，它不在某一行 `Cache Line` 上竞争严重，但是它却在很多 `Cache Line` 上竞争，这样的代码段也是很值得优化的。同理还有多进程程序访问共享内存时的情况。
- 在 `Pareto` 表里，如果发现很长的 `load` 操作平均延迟，常常就表明存在严重的伪共享，影响了性能。
- 接下来去看样本采样自哪些节点和 `CPU`，据此进行优化，将哪些内存或 `Task` 进行 `NUMA` 节点锁存。

最后，`Pareto` 表还能对怎么解决对齐得很不好的Cache Line，提供灵感。 例如：

- 很容易定位到：写地很频繁的变量，这些变量应该在自己独立的 `Cache Line`。可以据此进行对齐调整，让他们不那么竞争，运行更快，也能让其它的共享了该 `Cache Line` 的变量不被拖慢。
- 很容易定位到：没有 `Cache Line` 对齐的，跨越了多个 `Cache Line` 的热的 `Lock` 或 `Mutex`。
- 很容易定位到：读多写少的变量，可以将这些变量组合到相同或相邻的 `Cache Line`。

3.7 使用原始的采样数据

有时直接去看用 `perf c2c record` 命令生成的 `perf.data` 文件，其中原始的采样数据也是有用的。 可以用 `perf script` 命令得到原始样本，`man perf-script` 可以查看这个命令的手册。输出可能是编码过的，但你可以按 `load weight` 排序（第 5 列），看看哪个 load 样本受伪共享影响最严重，有最大的延迟。

4 致谢

最后，在文章末尾，Joe 给出了如下总结，并在博客中致谢了所有的贡献者:

> Linux perf c2c 功能在上游的 4.2 内核已经可用了。这是集体努力的结果。

Oliver Yang / 2018-01-15
Published under (CC) BY-NC-SA in categories Chinese Software Hardware tagged with perf

trace kernel linux memory