Atitit.提升系统性能 Cache

# 3                 CPU Cache工作原理

## 3.1              CPU Cache简介

Cache是CPU和内存之间的一个中转站。由于目前CPU的频率（速度）已经大大超过内存，往往CPU会为了读取或存储数据白白浪费几十个时钟周期。这造成了巨大的资源浪费。于是Cache的设计思想被提上日程，几经实验修改后，逐渐形成了现在所能够看到的Cache架构。  
    在现代CPU设计中，设计师们要解决的最主要问题，就是找到一个在CPU和内存之间平衡的均点。Cache作为CPU--->内存的中转站，在其中发挥了巨大的作用。CPU在请求数据或指令时，除了常规的在内存中进行查找外，还会在Cache中进行查找。一旦命中，就可以直接从Cache中读取，节约大量时间。正因为如此，Cache在现代CPU中显得越来越重要。

## 3.2              Cache原理

Cache属于SRAM（Satic Random Access Memory），它利用晶体管的逻辑开关状态来存取数据。也正因为如此，SRAM内部的电路构造比起常见的DRAM（Dynamic Random Memory）要复杂得多，导致了成本的巨增。这也是SRAM不能普及的一个重要原因。  
    Cache在计算机存储系统中没有编配固定的地址，这样程序员在写程序时就不用考虑指令是运行在内存中还是Cache中，Cache对于计算机上层来说是完全透明的。  
    CPU在读取数据时，会首先向内存和Cache都发送一个查找指令。如果所需要的数据在Cache中（命中），则直接从Cache读取数据，以节约时间和资源。CPU对Cache的搜索叫做Tag search，即通过Cache中的[CAM](http://blog.vsharing.com/Tag/CAM" \t "http://bbs.vsharing.com/Information/Soft/_blank)（Content Addressed Memory）对希望得到的Tag数据进行搜索。CAM是一种存储芯片，延迟很低，常用于网络设备中用作路由选择。  
    CPU进行Tag search的过程是这样的：在Cache中数据或指令是以行为单位存储的，一行又包含了很多字。如现在主流的设计是一行包含64Byte。每一行拥有一个Tag。因此，假设CPU需要一个标为Tag 1的行中的数据，它会通过CAM对Cache中的行进行查找，一旦找到相同Tag的行，就对其中的数据进行读取。  
  
    在当前流行的系统中，虽然Cache的容量一直在增涨，但大多数处理器中Cache最大的也不过4MB，设计师们是如何保证在这小小的Cache中保存的数据或指令就一定是CPU需要的呢？这就要利用到CPU运行时的两个基本局限性：时间局限性和空间局限性。

****时间局限性****

所谓时间局限性，是指CPU在某一时刻使用到的数据或指令，在下一时刻也会被重复利用。比如3D游戏中，同一场景会在不同时间被渲染多次，如果在第一次渲染中Cache存储了相关指令、数据，那么在下一次需要重复渲染时，就能够直接从Cache中读取相关内容。

****空间局限性****

空间局限性，指的是CPU在读取某一地址的数据时，也有可能会用到该地址附近的数据。也就是说，CPU需要的数据在很多时候是连续的。例如在听歌或看电影时，数据流总是连续的（正常播放状态下）。这样的应用对于CPU来说是很有利的，数据预读取单元也能够发挥最大作用。  
    Cache正是利用了上述两个局限性，得已实现并工作。设计师们能够充分利用这两个局限，在容量较小的Cache中存入CPU在将来某时刻需要的内容。需要指出的是，很多程序在执行指令或数据时，所呈现出来的局限性是不同的。有可能执行指令的时候呈现出时间局限性，而数据呈现出空间局限性，因此设计师们把L1 Cache分成了Instruction Cache（指令缓存）和Data Cache（数据缓存）。

## 3.3              Cache运行原理

Cache的数据存储是以行（Line）为单位的，每一行又包含64Byte。行是存储在“块”（Block frame）这种数据容器中的，而块则直接与内存相对应。很明显，Cache中可能包含数个块。那么这些Cache块是怎么与内存相对应相联系的呢？有三种办法。  
    ****完全相联法****  
    完全相联法，即内存中的数据可以存储在任何Cache块中，同一数据也可以存储在不同的块中。 这样数据的存储相当灵活，CPU在查找时也很方便：只需在块中对比找出需要的Tag行，即实现命中，显著的提升了命中率。然而这样做的缺陷也很明显：对于容量较大的Cache来说，CPU需要在成百的块中查找需要的Tag行，延迟大大增加。因此这种设计方式只适用于容量较小的Cache。

****直接映像法****

由于完全相联法的这种局限性，设计师们很快提出了另一种旨在降低延迟的组织方式：直接映象法。和完全相联不同，在直接映象中内存会将数据存入的Cache块地址“记住”，以后再次存储时就只能使用该块。这样做的好处是使CPU只需要进行一次Tag search，在以后的读取操作中就可以直接找到所需Tag行所在的框架，从而达到降低延迟的目的。  
    而至于内存会将数据存入Cache的哪个块中，这有个算法——块地址与整个块数的同余。比如，有个1K的缓存，块大小为64字节，则总共有16个缓存块，那在内存中首地址为12480的内存块应该保存在缓存的哪个块中呢？12480/64=195，195%16=3,则它应该放入第4个块中。这样一来，内存中的数据能很快的读取到缓存中的某个块中，CPU也能很快的在这个块中找到所要的数据，这样就省下了对比各个块的时间，自然延迟就小了，但是，如果第4个块中装入了内存块195的数据，而其它同余依然是3的35，51，67等这些块就不能装入了，这样，当CPU需要35，51，67这些块的时候，就会发生冲突（collision），导致出现Cache miss的情况，大大的降低了命中率。

****路组相联法****

直接映象法和完全相联法都只解决了Cache运行问题的一个方面。这以后设计师们又设计了另一种综合前两者优点的方法----路组相联法。先将Cache分成不同的组，每个组中放入不同的块。内存数据的存储对于组来说采用直接映像法---这就有效控制了延迟。而每个块中的行又按完全相联的方法，灵活存储---这又提高了命中率。

显然，一组中放入的块越少，那么它的命中率和延迟都能够控制得越好。组中的块称为“路”，有几个块就叫做几路。

## 3.4              Cache写入策略

前面说了这么多，似乎都在谈Cache的读取，而忘了它的写入是怎么样的？在一个常见的应用程序中，其50%的指令是与数据存取相关的，而其中又有近30%的指令与读取有关。也就是说，CPU在运行中进行的读取操作频率要远远大于写入操作。Cache的写入则相关简单，依赖于三种写入策略：

****写回法****

当CPU更新Cache时，并不同时更新内存中的相应数据。这种方法减少了访问内存的次数，缩短了时间。但在保持与内存内容的一致性上存在在隐患，并且使用写回法，必须为每个缓存块设置一个修改位，来反映此块是否被CPU修改过。

****全写法****

和写回法相反，当Cache数据更新时，也同时更新内存数据。Cache不用设置修改位或相应的判断器。这种方法的好处是，当Cache命中时，由于缓存和内存是同时写入的，所以可以很好的保持缓存和内存内容的一致性，但缺点也很明显，由于每次写入操作都要更新所有的存储体，如果一次有大量的数据要更新，就要占用大量的内存带宽，而内存带宽本来就不宽裕，如果写操作占用太多带宽的话，那主要的读操作就会受到比较大的影响。

****写一次法****

这是一种基于上面两种方法的写策略，它的特点是，除了第一次更新Cache的时候要同时更新内存，其它时候都和写回法一样，只修改Cache。这就在内存一致性和延迟中找到了一个较好的平衡。

## 3.5              Cache替换策略

所谓替换策略，是指Cache中数据的更新方法。无论如何，Cache的容量还是比较小的。要保证在这样的容量下其中的数据是CPU需要的，就需要不停地对Cache进行更新，摈弃不需要的旧数据，加入新内容。  
    目前常见的Cache替换策略有三种：

****FIFO****

先进先出（First In First Out，FIFO），即替换最早进入Cache的数据。这种算法在早期的Cache里使用较多，那时候的Cache的容量还很小，数据在Cache的时间都不会太久，经常是CPU一用完就不得不被替换下来，以保证CPU所需要的其它数据能在Cache中找到。但这样命中率也会降低，因为这种算法所依据的条件是数据在Cache中的时间，而不是其在Cache中的使用情况。

****LFU****

最不经常使用（Least Frequency Used，LFU），即替换被CPU访问次数最少的行。LFU算法是将每个行设置计数器，起始为0，每被CPU访问一次，就加1，当需要替换时，找到那个计数最小的替换出来，同时将其它行的计数置0。这种算法利用了时间局限性原理，但是每次替换完都把其它行置0，使得把这种局限性的时间限定在了两次替换之间的时间间隔内。由于替换太频繁，让这时间间隔太短了，并不能完全反映出CPU近期的访问情况。

****LRU****

近期最少使用（Least Recently Used，LRU），即替换在近段时间里，被CPU访问次数最少的行，它是LFU的拓宽版本。其原理是在每个行中设置一个计数器，哪一行被CPU访问，则这行置0，其它增1，在一段时间内，如此循环，待到要替换时，把计数值最大的替换出去。这种算法相当于延长了替换时间，从而更能本质地反应行的使用情况。有一点要说明的是，有时候行被替换出，并不代表它一定用不到了，而是Cache容量不够了。这种算法是目前最优秀的，大部分的Cache的替换策略都采用这种算法。

## 3.6              Cache对齐

一个L1 CACHE相当于一块小的内存，我们假设它为16K大，它会与物理内存交互。它和内存交互一般一次传输一个Cache Line的大小，比如16或者32个字节,也就是: CACHE 字节0-15一次写到/读取物理内存 ，字节16-31一次写到/读取物理内存32-47 ... ...这些一次被传输的字节被称为cache line。 另外，cache写到物理内存的位置不是任意的，我们假定内存为64K,那么cache地址0的数值只能和物理内存的地址0, 16K, 32K交互；cache地址1的数值只能和物理内存的地址1, 16K+1, 32K+1交互…，cache地址16K-1的数值只能和物理内存的地址16K-1, 16K+16K-1, 32K+16K -1交互. 这说明了: 假设对象A的一个字段长为16个字节，如果它放在物理地址 0-15,那么它将和cache的第一个cache line 交互，如果放在物理地址 8-23,那么如果CPU要访问这个字段，必须将第一个和第二个cache line 都读入，才能获得这个字段的信息，显然这样速度慢，所以一般字段需要cache line对齐，在这里就是16个字节对齐。

在Linux系统中，经常会在数据结构后面跟上一个\_\_\_\_cacheline\_aligned宏，通常这个宏定义如下：

#define CACHELINE\_SIZE 32

#define \_\_\_\_cacheline\_aligned \_\_attribute\_\_((\_\_aligned\_\_(CACHELINE\_SIZE)))

所以，实际上是要求数据结构的起始地址要32字节对齐，以便于CPU访问此数据结构的时候Cache命中率比较高。

## 3.7              Cache效果指标

衡量Cache的效果，主要指标是Cache命中率。若命中率为90%, 不命中时需要另花10个周期，则平均访存时间为：1+10%\*10 = 2 周期，即存储系统的速度是cache速度的1/2。

[分享]提升系统性能方法初步探讨 - 软件行业 - 畅享论坛.html