Understanding Caching

Jan 01, 2004 By James Bottomley (/user/801293)
in

Architectures that support Linux differ in how they handle caching at the hardware level. Here's how the kernel gets the best possible use out of every cache design.

(/issue/117)

From Issue #117 January 2004 (/issue/117)

Since the earliest days of microprocessors, system designers have been plagued by a problem in which the speed of the CPU's operation exceeded the bandwidth of the memory subsystem to which it was connected. To avoid wasting CPU cycles while waiting for the memory to fetch the requested data, the universally adopted solution was to use an area of faster (and thus more expensive) memory to cache main memory data. This solution allowed the CPU to operate at its natural speed as long as the data it required was available in the cache.

The purpose of this article is to explain caching from the point of view of a kernel programmer. I also explain some of the common terms used to describe caches. This article is divided into sections whose kernel programming relevance is indicated; that is, some sections explain that cache properties are irrelevant to understanding the essentials of how the kernel handles caching. If you're coming from an Intel IA32 background, caching is practically transparent to you. In order to write kernel code that operates correctly on all the architectures Linux supports, however, you need to know the essentials of how caching works in general.

A Cache and Its Properties

Simply put, a cache is a place that buffers memory accesses and may have a copy of the data you are requesting. Usually one thinks of caches (there may be more than one) as being stacked; the CPU is at the top, followed by layers of one or more caches and then the main memory. In this hierarchy, caches are quantified by their level. The cache closest to the CPU is called level one, L1 for short, and caches increase in level until the main memory is reached.

A cache line is the smallest unit of memory that can be transferred to or from a cache. The essential elements that quantify a cache are called the read and write line widths. These signify the minimum amount of data the cache must read or write from the memory or cache below it. Frequently, these quantities are the same, so caches often are quantified simply by the line width. Even if they differ, the longest width usually is called the line width.

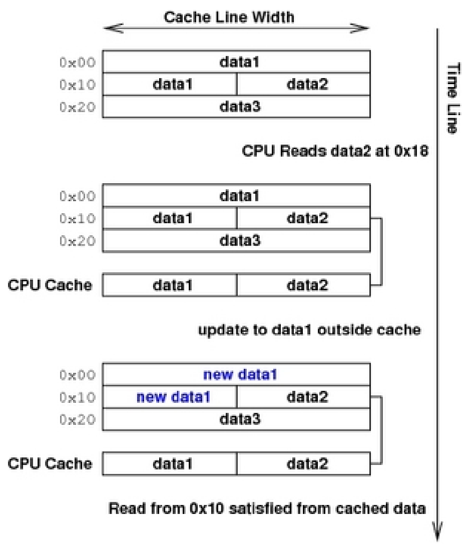
The next property that quantifies a cache is its size. This number is an indication of how much data could be stored in the cache. Often, the performance rule of thumb is the bigger the cache, the better the benchmarks.

A multilevel cache can be either inclusive or exclusive. Exclusive means a particular cache line may be present in exactly one of the cache levels and no more than one. Inclusive means the line may be present simultaneously in more than one level of cache. Nothing prevents the line widths from being different in differing cache levels.

Finally, a particular cache can be either write through or write back. Write through means the cache may store a copy of the data, but the write must be completed at the next level down before it can be signaled as complete to the layer above. Write back means a write may be considered complete as soon as the data is stored in the cache. For a write back cache, as long as the written data is not transmitted, the cache line is considered dirty, because it ultimately must be written out to the level below.

Cache Management and Coherency

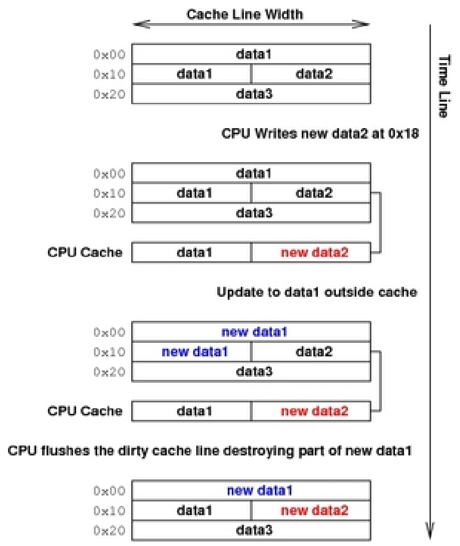
One of the most basic problems with caches is coherency. A cache line is termed coherent when the data in the line is identical to the data stored in the main memory being cached. If this is not true, the cache line is termed incoherent. Lack of coherency can cause two particular problems. The first problem, which may occur for all caches, is stale data. In this situation, data has changed in main memory but the cache hasn't been updated to reflect the change. This usually manifests itself as an incorrect read, as illustrated in Figure 1. This is a transient error, because the correct data is sitting in main memory; the cache simply needs to be told to bring it in.



(/files/linuxjournal.com/linuxjournal/articles/071/7105/7105f1.jpg)

Figure 1. Stale Data Problem

The second problem, which occurs only with write back caches, can cause actual destruction of data and is much more insidious. As illustrated in Figure 2, the data has been changed in memory, and it also has been changed separately by a CPU write to the cache. Because the cache must write out one line at a time, there now is no way to reconcile the changes—either the cache line must be purged without being written, losing the CPU's change, or the line must be written out, thus losing the changes made to main memory. All programmers must avoid reaching the point where data destruction becomes inevitable; they can do this through the judicious use of the various cache management APIs.


Cache Line Width
0x00
0x10
0x20
data1
data2
data3
Time Line
CPU Writes new data2 at 0x18
CPU Cache
new data2
Update to data1 outside cache
new data1
CPU flushes the dirty cache line destroying part of new data1

(/files/linuxjournal.com/linuxjournal/articles/071/7105/7105f2.jpg)

Figure 2. Data Destruction by Dirty Cache Lines

Comments

Comment viewing options

Threaded list - expanded ▼ | Date - newest first ▼ | 50 comments per page ▼ | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

Very good article, but ... wh (/article/7105#comment-27905)

Submitted by Anonymous (not verified) on Thu, 04/07/2005 - 04:27.

Very good article, but ... where is figure 4 ?

figure 4 is just like the (/article/7105#comment-320027)

Submitted by Anonymous (not verified) on Wed, 03/05/2008 - 04:16.

figure 4 is just like the tag(hidden!) :#)