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)

Cache-Line Interference

The situation where two sets of independent data lie in the same cache line, potentially leading to the data destruction detailed above, is termed *cache-line interference*. If you are laying out data structures in memory, the general rule to avoid this situation is never, ever have data that can be modified outside of the caches mixed with data the CPU may ordinarily use. If you absolutely have to violate this rule, make sure all externally modifiable elements of the structure are aligned *L1_CACHE_BYTES*, a value set at compile time to the largest possible cache width value for all the processors on which your code may run. The best thing to do is use kmalloc to allocate two separate regions. kmalloc never allocates two regions that overlap in a cache line.

Cache Management Instructions

The most basic instruction, called an invalidate, simply ejects the nominated line from all caches. Any reference to data in that line causes it to be re-fetched from main memory. Thus, the stale data problem may be resolved by invalidating the cache line before reading the data. In Linux, such an invalidation is done with:

```
void
dma_cache_inv(unsigned long address
              unsigned long size);
```

where *address* is the virtual address on which to begin, and *size* is the length of data to invalidate. Note that *size* is rounded up automatically to a multiple of the cache line width.

For write back caches, any dirty cache line may be written out, or flushed, to main memory using:

```
void
dma_cache_wback(unsigned long address,
                unsigned long size);
```

This flushing must be done before anything changes the main memory under the dirty cache line. You therefore must issue the flush before an external entity (such as a PCI card) modifies main memory and issue an invalidate after this flush but before the CPU accesses any data that has changed.

In theory, for write back caches an invalidate kills the cache line without actually writing the data out, thus destroying the data in the cache. A safer thing to do in this case is issue a flush and invalidate instruction:

```
void
dma_cache_wback_inv(unsigned long address,
```

```
                unsigned long size);
```

This flushes the cache lines to main memory and then invalidates them from the cache.

Types of Caches

This section explains how a cache actually works. The only vital piece of information you need from this section is a property called aliasing, which means that the same physical address in memory may be cached in multiple distinct cache lines. How the kernel actually manages the aliases is discussed in the following section.

In a directly mapped cache, as shown in Figure 3, the cache is divided up into cache lines of known width (four in the example). Each line in the cache is characterized by a unique index, so every byte in the cache is addressed by the index of the line and offset into the line. Each index of the cache also possesses a hidden number called the tag.

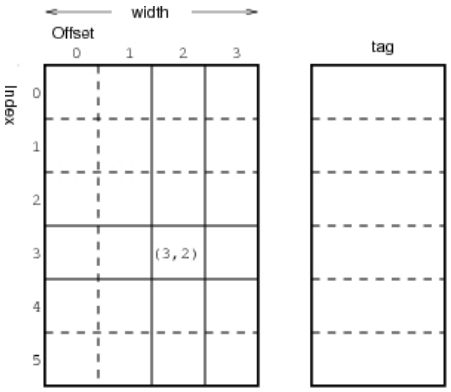

Figure 3. A Directly Mapped Cache

Every address in the system is divided into three pieces—tag, index and offset—along a power of two boundary (Figure 4). When a line is brought into the cache, the tag and index are extracted from the address. The line is stored in the cache at the required index, and the hidden tag is stored along with the line data. When the CPU makes reference to a particular address, the cache is consulted at the given index. If the tags match, the offset into the line is fetched to satisfy the address reference. If the tags do not match, the current line may be flushed back to main memory and the correct line brought into the cache.

Every cache-able address has one and only one corresponding index line, which can cause problems. For instance, if the processor reads a sequence of addresses that accidentally happen to correspond to the same cache index, the cache line must be evicted and re-fetched on each read. Such a situation easily can happen in, say, a for loop reading elements of a structure that happens to be about the same size as the cache. For directly mapped caches, the index sometimes is called the cache color, and this problem is called the cache-line coloring problem.

To get around the coloring problem of directly mapped caches, cache circuitry sometimes is arranged so that a cache lookup can compare tags simultaneously in more than one cache line. In an N-way associative cache, each index corresponds to N cache lines (and tags); thus, we can have up to N addresses with the same index simultaneously in the cache. The added parallel cache lookup circuitry tends to increase the cost of associativity somewhat, so it usually is found only in higher-end CPUs.

At the very top of the range, you might find a fully associative cache. This type of cache has no index at all, and all lines are consulted at once when looking for a particular tag.

All modern CPUs handle address translation, which means the virtual address used by the kernel or application to refer to memory isn't the same as the physical address where the data actually resides. The caches can be placed before or after address translation, and sometimes in a hierarchical cache there is a mixture of placements. Each of these placements has different properties and features as described below.

In physically indexed, physically tagged (PIPT) caches, the tag and index of the cache are both in physical memory, that is, after virtual address translation has been done. This process is nice and simple, but the disadvantage of PIPT caches is that a valid address translation must be in the TLB (translation lookaside buffer) of the CPU. If such a TLB entry needs to be fetched from memory before the address translation can be done, the advantage of caching the data is lost. Even if a TLB entry is present, the TLB lookup and the cache lookup must be done sequentially, making these caches slow.

In virtually indexed, virtually tagged (VIVT) caches, on the other hand, both the index and tag are in the virtual address space in which the CPU currently is operating. Although this makes cache lookups much faster (no address translation needs to be done before the lookup or after, if the cache lookup is successful), they suffer from several other problems:

1. Virtual address translations usually are changed as part of normal kernel operation, so the cache must pay careful attention to changes in TLB entries (and changes in address space) and flush cache lines whose translations have changed.

2. Even in a single address space, multiple virtual addresses may exist for the same physical address. Each of these virtual addresses would be cached separately, even though they represent the same data. This is called the cache-line aliasing problem.

Generally, though, the added lookup speed of a VIVT cache outweighs its disadvantages, making it the predominant cache type for non-x86 CPUs.

A type of hybrid cache designed to overcome some of the shortcomings of the VIVT cache without sacrificing too much of its speed advantage is virtually indexed, physically tagged (VIPT) caching. The index is on the virtual address, but the tag is on the physical address, so the combination (tag, offset) must specify the full physical address. This requirement causes the tags to be larger than the tags for other cache types.

The VIPT cache gains its speed advantage over PIPT because the address translation and the cache lookup now can be done in parallel. The CPU doesn't know if the cache line is valid (the tags match), however, until the address translation has completed.

The disadvantages of VIVT are overcome because the tag is physical, thus the VIPT cache automatically detects aliasing when it sees that two tags are identical in the cache. Thus, a VIPT cache may be constructed in such a fashion that cache-line aliasing never occurs.

This fourth theoretical type of cache, physically indexed, virtually tagged (PIVT), is basically useless and is not discussed further.

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!) :#)