kernel如何保证cache数据一致性

原创 kerneler_ 于 2015-08-27 10:31:21 发布 20841 收藏 88 版权

分类专栏： linux kernel 内核机制学习笔记 文章标签： 缓存 kernel 嵌入式 cache 存储

在嵌入式系统中，cache位于CPU与DDR之间，是一段SRAM，读写性能远高于DDR，利用cache line提供了预取功能，平衡CPU与DDR之间的性能差异，提高系统的性能。

据我了解，ARM/PPC/MIPS三款主流嵌入式处理器都是**软件管理cache**，即有专门的指令来进行cache操作，如PPC的iccci icbi，ARM的CP15协处理器也提供对cache的操作。

cache的操作有2种：写回和无效。写回操作是将cache中数据写回到DDR中，无效操作是无效掉cache中原有数据，下次读取cache中数据时，需要从DDR中重新读取。这两种操作其实都是为了保证cache数据一致性。
在kernel平台汇编代码中也封装了cache操作函数，这里以ARM v7处理器
3.4.55内核为例，在arch/arm/mm/cache-v7.S中就封装了cache的操作函数，其中v7_dma_flush_range刷新函数是完成了写回和无效2种操作。

其他平台（ARM MIPS）中cache操作函数也类似。

对cache敏感是因为在cache问题上被坑过2次。。。，其中一次在移植uboot时，没有注意cache，网卡dma始终不能工作，表面上dma描述符该配置的我都已经配置了，但是dma就是不能运行，在将uboot整个启动代码都过了一遍后我才考虑到是cache的问题，我写入的描述符数据并没有完全写入DDR中，而是cache住了，uboot对系统运行性能要求不高，索性将cache关掉，发现dma果然正常了。这个问题前后折腾了我半个多月，cache的问题实在不好查，读取写入DDR中的数据是完全正确的（因为读到的其实是cache中的数据），只能靠猜测推断这个问题原因。导致我现在一看到cache操作就紧张。

不扯题外话了，**正因为嵌入式处理器软件管理cache，就需要我们代码主动去操作cache，但在内核开发中却很少会直接进行cache操作，那么kernel是在什么时候进行的cache操作。**

首先想明白一点，为什么要进行cache操作，只能说cache是天使也是魔鬼。

cache在提高了系统性能同时却导致了数据的不一致性。嵌入式处理器软件管理cache的初衷就是保证数据一致性。

那什么地方需要保证数据一致性呢?

对于由CPU完全操作的数据，数据是完全一致的。也就是该数据完全由CPU写读操作，没有对CPU不透明的操作。这种情况下CPU读写的数据都是来自于cache，我们代码（代码由处理器执行，我们就应该站在处理器视角来看这个问题）完全不用考虑cache一致性的问题。

想来想去，我觉得kernel中有2种情况是需要保证数据一致性的：

（1）寄存器地址空间。寄存器是CPU与外设交流的接口，有些状态寄存器是由外设根据自身状态进行改变，这个操作对CPU是不透明的。有可能这次CPU读入该状态寄存器，下次再读时，该状态寄存器已经变了，但是CPU还是读取的cache中缓存的值。但是寄存器操作在kernel中是必须保证一致的，这是kernel控制外设的基础，IO空间通过ioremap进行映射到内核空间。ioremap在映射寄存器地址时页表是配置为uncached的。数据不走cache，直接由地址空间中读取。保证了数据一致性。

（2）DMA缓冲区的地址空间。DMA操作对于CPU来说也是不透明的，DMA导致内存中数据更新，对于CPU来说是完全不可见的。反之亦然，CPU写入数据到DMA缓冲区，其实是写到了cache，这时启动DMA，操作DDR中的数据并不是CPU真正想要操作的。

kernel中对于DMA操作是如何保证cache一致性的呢?

在LDD3的内存映射和DMA一章中详细介绍了通用DMA操作层的一些函数，当时看这一章有些说法看得我挺晕的，现在站在cache的角度再来看就明了了好多。

通用DMA层主要分为2种类型的DMA映射：
（1）一致性映射，代表函数：
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t handle);

（2）流式DMA映射，代表函数：
dma_addr_t dma_map_single(struct device *dev, void *cpu_addr, size_t size, enum dma_data_direction dir)
void dma_unmap_single(struct device *dev, dma_addr_t handle, size_t size, enum dma_data_direction dir)

void dma_sync_single_for_cpu(struct device *dev, dma_addr_t handle, size_t size, enum dma_data_direction dir)
void dma_sync_single_for_device(struct device *dev, dma_addr_t handle, size_t size, enum dma_data_direction dir)

int dma_map_sg(struct device *, struct scatterlist *, int, enum dma_data_direction);
void dma_unmap_sg(struct device *, struct scatterlist *, int, enum dma_data_direction);

首先来看一致性映射如何保证cache数据一致性的。直接看下dma_alloc_coherent实现。

```
void *
dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp)
{
    void *memory;

    if (dma_alloc_from_coherent(dev, size, handle, &memory))
```

```
        return memory;

    return __dma_alloc(dev, size, handle, gfp,
                pgprot_dmacoherent(pgprot_kernel),
                __builtin_return_address(0));
}
```

关键点在于pgprot_dmacoherent，实现如下：

```
#ifdef CONFIG_ARM_DMA_MEM_BUFFERABLE
#define pgprot_dmacoherent(prot) \
    __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_BUFFERABLE | L_PTE_XN)
#define __HAVE_PHYS_MEM_ACCESS_PROT
struct file;
extern pgprot_t phys_mem_access_prot(struct file *file, unsigned long pfn,
                     unsigned long size, pgprot_t vma_prot);
#else
#define pgprot_dmacoherent(prot) \
    __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_UNCACHED | L_PTE_XN)
#endif
```

其实就是修改page property为uncached，这里需要了解，page property是kernel页管理的页面属性，在缺页异常填TLB时，该属性就会写到TLB的存储属性域中。保证了dma_alloc_coherent映射的地址空间是uncached的。
并且__dma_alloc中调用__dma_alloc_buffer，在./arch/arm/mm/dma-mapping.c的100行处，如下：

```
    /*
     * Ensure that the allocated pages are zeroed, and that any data
     * lurking in the kernel direct-mapped region is invalidated.
     */
    ptr = page_address(page);
    memset(ptr, 0, size);
    dmac_flush_range(ptr, ptr + size);
    outer_flush_range(__pa(ptr), __pa(ptr) + size);
```

在分配到缓冲区后，对缓冲区进行刷cache以及可能存在的外部cache（二级cache）的刷新。

dma_alloc_coherent首先对分配到的缓冲区进行cache刷新，之后将该缓冲区的页表修改为uncached，以此来保证之后DMA与CPU操作该块数据的一致性。

LDD3中说一致性映射缓冲区可同时被CPU与DMA访问，就是通过uncached的TLB映射来保证的。

再来看流式DMA映射，以dma_map_single为例，因代码调用太深，这里就只是列出调用关系，跟cache有关的调用如下：

```
dma_map_single ==> __dma_map_page ==> __dma_page_cpu_to_dev ==> ___dma_page_cpu_to_dev
___dma_page_cpu_to_dev实现如下：
void ___dma_page_cpu_to_dev(struct page *page, unsigned long off,
    size_t size, enum dma_data_direction dir)
{
    unsigned long paddr;

    dma_cache_maint_page(page, off, size, dir, dmac_map_area);

    paddr = page_to_phys(page) + off;
    if (dir == DMA_FROM_DEVICE) {
        outer_inv_range(paddr, paddr + size);
    } else {
        outer_clean_range(paddr, paddr + size);
    }
    /* FIXME: non-speculating: flush on bidirectional mappings? */
}
dma_cache_maint_page会对映射地址空间调用dmac_map_area，该函数最终会调用到arch/arm/mm/cache-v7.S中v7处理器的cache处理函数v7_dma_map_area，
/*
 *  dma_map_area(start, size, dir)
 *  - start - kernel virtual start address
 *  - size  - size of region
 *  - dir   - DMA direction
 */
```

```
ENTRY(v7_dma_map_area)                  add r1, r1, r0
    teq r2, #DMA_FROM_DEVICE
    beq v7_dma_inv_range
    b   v7_dma_clean_range
ENDPROC(v7_dma_map_area)
```

指定方向为DMA_FROM_DEVICE，则v7_dma_inv_range无效掉该段地址cache。方向为DMA_TO_DEVICE，则v7_dma_clean_range写回该段地址cache。保证了cache数据一致性。
之后在___dma_page_cpu_to_dev中还对外部cache进行了刷新。

dma_map_single中并没有对cpu_addr指定缓冲区映射的存储属性进行修改，还是cached的，但是对该缓冲区根据数据流向进行了cache写回或者无效，这样也保证了cache数据一致性。

再来看dma_unmap_single，调用关系如下：

```
dma_unmap_single ==> __dma_unmap_page ==> __dma_page_dev_to_cpu ==>
___dma_page_dev_to_cpu ==> dmac_unmap_area ==》v7_dmac_unmap_area
dmac_unmap_area实现如下：
/*
 *  dma_unmap_area(start, size, dir)
 *  - start - kernel virtual start address
 *  - size  - size of region
 *  - dir   - DMA direction
 */
ENTRY(v7_dma_unmap_area)
    add r1, r1, r0
    teq r2, #DMA_TO_DEVICE
    bne v7_dma_inv_range
    mov pc, lr
ENDPROC(v7_dma_unmap_area)
```

指定方向为DMA_TO_DEVICE，不做任何操作。方向为DMA_FROM_DEVICE，则v7_dma_unmap_area无效掉该段地址cache。
我的理解因为对于指定为CPU需要读取的数据，在释放该缓冲区后必须保证cache数据一致性，接下来CPU就要读取数据进行处理了。而对于指定为CPU写的数据，缓冲区释放后CPU不会再去操作该缓冲区，所以不做任何操作。

从这里就可以看出来，LDD3讲到，流式DMA映射对于CPU何时可以操作DMA缓冲区有严格的要求，只能等到dma_unmap_single后CPU才可以操作该缓冲区。
究其原因，是因为流式DMA缓冲区是cached，在map时刷了下cache，在设备DMA完成unmap时再刷cache（根据数据流向写回或者无效），来保证了cache数据一致性，在unmap之前CPU操作缓冲区是不能保证数据一致的。因此kernel需要严格保证操作时序。
当然kernel也提供函数dma_sync_single_for_cpu与dma_sync_single_for_device，可以在未释放时操作缓冲区，很明显这2个函数实现中肯定是再次进行刷新cache的操作保证数据一致性。

到这里DMA的2种类型映射都分析完了，**很清晰的看出一致性映射与流式DMA映射核心区别就是在于缓冲区页表映射是否为cached，一致性映射采用uncached页表保证了CPU与外设都可以同时访问。**
不过这些都是内核为驱动开发者已经封装好的接口函数，驱动开发者并不需要关心cache问题，只需要按照LDD3的规定调用这些接口即可。
这也就是为什么在驱动中很少见到cache操作，内核代码将cache操作做到对驱动不透明了。
这也让我想起了在开发网卡驱动时，DMA描述符的分配是一致性映射，是因为DMA描述符需要CPU与设备同时操作。而数据收发缓冲区分配是流式的，随用随分配，用完释放后CPU才可以操作数据！

到目前为止，我所接触到的内核TLB映射，做了uncached映射的只有2个：寄存器空间（ioremap）和一致性DMA缓冲区（dma_alloc_coherent），其他地址空间都是cached，来保证系统性能。

内核操作cache的时机是在操作DMA缓冲区时，其他时候内核代码不需要关心cache问题。这篇文章就分析到这！