Kernel Korner

DMA makes I/O go faster by letting devices read and write memory without bothering the CPU.

by James Bottomley

DMA stands for direct memory access and refers to the ability of devices or other entities in a computing system to modify main memory contents without going through the CPU. The desirability of DMA lies in not troubling the CPU; the system simply can request that the data be fetched into a particular memory region and continue with other tasks until the data is ready. Most of the problems in DMA, however, are due to the lack of CPU involvement.

The problems with DMA are threefold. First, the CPU probably is operating a memory management unit. Therefore, the address the CPU uses to describe the memory region is not the same as the physical address of main memory. Second, because the transfer is to main memory, the caches between that memory and the CPU probably are not coherent (see “Understanding Caching”, LJ, January 2004. Third, there also may be a memory management unit on the I/O bus (called an IOMMU). This means the bus address the device uses to transfer the data may not be the same as the physical memory address or the CPU's virtual memory address. This concept is alien to most x86 people. Even here, though, the use of GARTs (graphical aperture remapping tables) for the AGP bus is making the x86 refusal of IOMMUs less strong than it once was.

The API that manages DMA in the Linux kernel must take into account and solve all three of these problems. In addition, because most DMA is done from devices on an external bus, three additional problems may occur. First, the I/O device addressing width may be different from the address width of physical memory. For instance, an ISA device is limited to addressing 24 bits, and some PCI devices in 64-bit systems are limited to addressing 32 bits. Second, the I/O bus controller circuitry itself may cache requests. This occurs mainly on the PCI bus, where write requests may be held in the PCI controller in the hope that it may accumulate them for rapid transfer to the device. This phenomenon is called PCI posting. Third, the operating system may request a transfer to a region that is contiguous in its virtual memory space but fragmented in the memory's physical space, usually because the requested transfer crosses multiple pages. Such a transfer must be accomplished using scatter/gather (SG) lists.

This article deals strictly with the DMA API for devices. The new generic device model in Linux 2.6 provides a nice way of describing device characteristics and finding their bus properties using a hierarchical tree. The interfaces described have undergone considerable revision in the transition from 2.4 to 2.6. Although the general principles of this article apply to 2.4, the API described and the kernel capabilities apply only to the 2.6 kernel.

SG Lists

For any DMA transfer, the first problem to consider is the user may request a large transfer (kilobytes to megabytes) to a given buffer. Because of the way virtual memory is managed, however, this area, which is contiguous in virtual space, may be composed of a sequence of pages fragmented all over physical memory. Linux expects that any transfer above a page size (4KB on an x86 system) needs to be described by an SG list. Ordinarily, these lists are constructed by the block I/O (BIO) layer. A key job of the device driver is to parameterize the BIO layer in the way it may divide up the I/O into SG list elements.

Almost every device that transfers large amounts of data is designed to accept these transfers as some form of SG list. Although the exact form of this list is likely to differ from the one supplied by the kernel,
usually is trivial.

I/O Memory Management Units (IOMMUs)

Figure 1. Address Domains in DMA

An IOMMU is a memory management unit that goes between the I/O bus (or hierarchy of buses) and the main memory. This MMU is separate from the IOMMU built in to the CPU. In order to effect a transfer from the device to main memory, the IOMMU must be programmed with the address translations for the transfer in almost exactly the same way as the CPU's MMU would be programmed. One of the advantages of doing this is an SG list generated by the BIO layer can be programmed into the IOMMU such that the memory region appears to be contiguous again to the device on the bus.

GARTs and IOMMU Bypass

A GART basically is like a simple IOMMU. It consists of a window in physical memory and a list of pages. Its job is to remap physical addresses in the window to physical pages in the list. The window typically is narrow, only about 128MB or so, and any accesses to physical memory outside this window are not remapped. This insufficiency exposes a weakness in the way the Linux kernel currently handles DMA: none of the DMA APIs have a failure return for failing to map the memory. A GART has a limited amount of remapping space, however, and once that is exhausted nothing may be mapped until some I/O completes and frees up mapping space.

Sometimes, like a GART, an IOMMU may be programmed not to do address remapping between the I/O bus and the memory in certain windows. This is called bypass mode and may not be possible for all types of IOMMU. Bypass mode is desirable sometimes, because the act of remapping adds a performance hit to the transfer, so lifting the IOMMU out of the way can achieve an increase in throughput.

The BIO layer, however, assumes that if an IOMMU is present, it is being used, and it calculates the space needed for the device SG list accordingly. Currently, no way exists to inform the BIO layer that the device
wishes to bypass the IOMMU. A problem occurs if the BIO layer assumes the presence of an IOMMU; it also assumes SG entries are being coalesced by the IOMMU. Thus, if the device driver decides to bypass the IOMMU, it may find itself with more SG entries than the device allows.

Both of these issues are being worked on in the 2.6 kernel. A fix for the IOMMU bypass already is under consideration and will be invisible to driver writers, because the platform code will choose when to do the bypass. The fix for the inability to map probably will consist of making the mapping APIs return failure. Because this fix affects every DMA driver in the system, implementing it is going to be slow.

**Bus Widths and DMA Masks**

In order to communicate the maximum addressing width, every generic device has a parameter, called the DMA mask, that contains a map of set bits corresponding to the accessible address lines that must be set up by the device driver. The DMA width has two separate meanings depending on whether an IOMMU is in use. If there is an IOMMU, the DMA mask simply represents a limitation on the bus addresses that may be mapped, but through the IOMMU, the device is able to reach every part of physical memory. If there is no IOMMU, the DMA mask represents a fundamental limit of the device. It is impossible for the device to transfer to any region of physical memory outside this mask.

The block layer uses the DMA mask when building a scatter/gather list to determine whether the page needs to be **bounced**. By bounce, I mean the block layer takes a page from a region within the DMA mask and copies all the data to it from the out-of-range page. When the DMA has completed, the block layer copies it back again to the out-of-range page and releases the bounce page. Obviously, this copying back and forth is inefficient, so most manufacturers try to ensure that the devices with which their server-type machines ship don't have DMA mask limitations.

**DMA and Coherency**

DMA occurs without using the CPU, so the kernel has to provide an API to bring the CPU caches into sync with the memory changed by the DMA. One thing to remember is the DMA API brings the CPU caches up to date only with respect to the kernel virtual addresses. You must use a separate API, described in my article “Understanding Caching”, to update the caches with respect to user space.

**Bus Posting (Caching)**

Sometimes high-end bus chips also come with caching circuitry. The idea behind this is that writes from the CPU to the chipset are fast, but writes across the bus are slow, so if the bus controller caches the writes, the CPU doesn't need to wait for them to complete. The problem with bus posting, as this type of caching is called, is that no CPU instruction is present to flush the bus cache, so bus cache flushes work according to a strict set of rules to ensure proper ordering. First, the rules are that only memory-based writes may be cached. Writes that go through I/O space are not cached. Second, the ordering of memory-based reads and writes must be preserved strictly, even if the writes are cached. This last property allows a driver writer to flush the cache. If you issue a memory-based read to any part of the device's memory region, all cached writes are guaranteed to be issued before the read begins.

No API is available to help with posting, so driver writers need to remember to obey the bus posting rules when reading and writing a device's memory region. A good trick to remember is if you really can't think of a necessary read to flush the pending writes, simply read a piece of information from the device's bus configuration space.
Using the DMA API in a Device Driver

The API is documented thoroughly in the kernel documentation directory (Documentation/DMA-API.txt). The generic DMA API also has a counterpart that applies only to PCI devices and is described in Documentation/DMA-mapping.txt. The intent of this section is to provide a high-level overview of all the steps necessary to get DMA working correctly. For detailed instructions, you also should read the above-mentioned documentation.

To start, when the device driver is initialized, the DMA mask must be set:

```c
int
dma_set_mask(struct device *dev, u64 mask);
```

where `dev` is the generic device and `mask` is the mask you are trying to set. The function returns true if the mask has been accepted and false if not. The mask may be rejected if the actual system width is narrower; that is, a 32-bit system may reject a 64-bit mask. Thus, if your device is capable of addressing all 64 bits, you first should try a 64-bit mask and fall back to a 32-bit mask if setting the 64-bit mask fails.

Next, you need to allocate and initialize the queue. This process is somewhat beyond the scope of this article, but it is documented in Documentation/block/. Once you have a queue, two vital parameters need to be adjusted. First, allow for the largest size of your SG table (or tell it to accept an arbitrarily big one) with:

```c
void
blk_queue_max_hw_segments(request_queue_t *q,
                         unsigned short max_segments);
```

Second, (if you need it), the overall maximum size:

```c
void
blk_queue_max_sectors(request_queue_t *q,
                   unsigned short max_sectors);
```

Finally, the DMA mask must be programmed into the queue:

```c
void
blk_queue_bounce_limit(request_queue_t *q,
                          u64 max_address);
```

Usually, you set `max_address` to the DMA mask. If an IOMMU is being used, however, `max_address` should be set to BLK_BOUNCE_ANY to tell the block layer not to do any bouncing.

Device Operation

To operate a device, it must have a request function (see the BIO documentation) whose job it is to loop around and pull requests from the device queue using the command:

```c
struct request
  *elv_next_request(request_queue_t *q);
```

The number of mapping entries required by the request are located in `req->nr_phys_segments`. You need to allocate an interim table of this size in units of `sizeof(struct scatterlist)`. Next, do the interim mapping with:

```c
int
blk_rq_map_sg(request_queue_t *q,
               struct request *req,
               struct scatterlist *sglist);
```
This returns the number of SG list entries used.

The following command provides the interim table supplied by the block layer, which finally is mapped using:

```c
int
dma_map_sg(struct device *dev,
        struct scatterlist *sglist, int count,
        enum dma_data_direction dir);
```

where `count` is the value returned and `sglist` is the same list passed into the function `blk_rq_map_sg`. The return value is the number of actual SG list entries needed by the request. The SG list is reused and filled up with the actual entries that need to be programmed into the device's SG entries. The `dir` provides a hint about how to cope correctly with cache coherency. It can have three values:

1. DMA_TO_DEVICE: the data is being transferred from memory to the device.
2. DMA_FROM_DEVICE: the device is transferring data into main memory only.
3. DMA_BIDIRECTIONAL: no hint is given about the transfer direction.

Two macros should be used when traversing the SG list to program the device's SG table:

```c
dma_addr_t
sg_dma_address(struct scatterlist *sglist_entry);

and:

int
sg_dma_len(struct scatterlist *sglist_entry);
```

which return the bus physical address and segment lengths, respectively, of each entry.

The reason for this two-stage mapping of the request is because the BIO layer is designed to be generic code and has no direct interaction with the platform layer, which knows how to program the IOMMU. Thus, the only thing the BIO layer can calculate is the number of SG segments the IOMMU makes for the request. The BIO layer doesn't know the bus addresses the IOMMU assigns to these segments, so it has to pass in a list of the physical memory addresses of all the pages that need to be mapped. It is the `dma_map_sg` function that communicates with the platform layer, programs the IOMMU and retrieves the bus physical list of addresses. This too is why the number of elements the BIO layer needs for its list may be longer than the number returned by the DMA API.

When the DMA has completed, the DMA transaction must be torn down with:

```c
int
dma_unmap_sg(struct device *dev,
        struct scatterlist *sglist,
        int hwcount,
        enum dma_data_direction dir);
```

where all the parameters are the same as those passed into `dma_map_sg` except for `hwcount`, which should be the value returned by that function. And finally, the SG list you allocated may be freed and the request completed.

**Accessing Data in the DMA region**
Usually, the device driver operates without touching any of the data it is transferring. Occasionally, however, the device driver may need to modify or inspect the data before handing it back to the block layer. To do this, the CPU caches must be made coherent with the data by using:

```c
int dma_sync_sg(struct device *dev,  
    struct scatterlist *sglist,  
    int hwcount,  
    enum dma_data_direction dir);
```

where the arguments are identical to `dma_unmap_sg`.

The most important factor in accessing data is when you do it. The rules for accessing depend on `dir`:

- **DMA_TO_DEVICE**: the API must be called after modifying the data but before sending it to the device.
- **DMA_FROM_DEVICE**: the API must be called after the device has returned the data but before the driver attempts to read it.
- **DMA_BIDIRECTIONAL**: the API may need to be called twice, after modifying the data but before sending it to the device and after the device finishes with it but before the driver accesses it again.

### Allocating Coherent Memory

Most devices use mailbox-type regions of memory for communication between the device and the driver. The usual characteristic of this mailbox region is that it is never used beyond the device driver. Managing the coherency of the mailbox using the previous API would be quite a chore, so the kernel provides a method for allocating a region of memory guaranteed to be coherent at all times between the device and the CPU:

```c
void *dma_alloc_coherent(struct device *dev, size_t size,  
    dma_addr_t *physaddr, int flag);
```

This returns the virtual address of a coherent region of size that also has a bus physical address (`physaddr`) to the device. The flag is used to specify the allocation type `GFP_KERNEL` to indicate the allocation may sleep to obtain the memory and `GFP_ATOMIC` to indicate the allocation may not sleep and may return NULL if it cannot obtain the memory. All memory allocated by this API also is guaranteed to be contiguous both in virtual and bus physical memory. There is an absolute requirement that `size` be less than 128KB.

As part of driver removal, the coherent region of memory must be freed with:

```c
void dma_free_coherent(struct device *dev, size_t size,  
    void *virtaddr,  
    dma_addr_t *physaddr);
```

where `size` is the size of the coherent region and `virtaddr` and `physaddr` are the CPU virtual and bus physical addresses, respectively, returned for the coherent region.

### Conclusions

The article offers a lightning-quick overview of how the block layer interacts with device drivers to produce SG lists for programming devices. You may find several additional pieces of the DMA API useful, including
APIs that handle unfragmented regions of physical memory. If this article whets your appetite, you're now ready to move on to reading the kernel Documents and source.

James Bottomley is the Software Architect for SteelEye. He also is an active member of the Open Source community. He maintains the SCSI subsystem, the Linux Voyager port and the 53c700 driver and has made contributions to PA-RISC Linux development in the area of DMA/device model abstraction.