What is Direct Memory Access (DMA) and Why Should We Know About it?

John Franco
Electrical Engineering and Computing Systems
University of Cincinnati

Introduction

What is Direct Memory Access?

Hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor.

Why is Direct Memory Access important?

Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated

What is the downside?

Hardware support is required – DMA controllers
DMA “steals” cycles from the processor
Synchronization mechanisms must be provided to avoid accessing non-updated information from RAM

Introduction

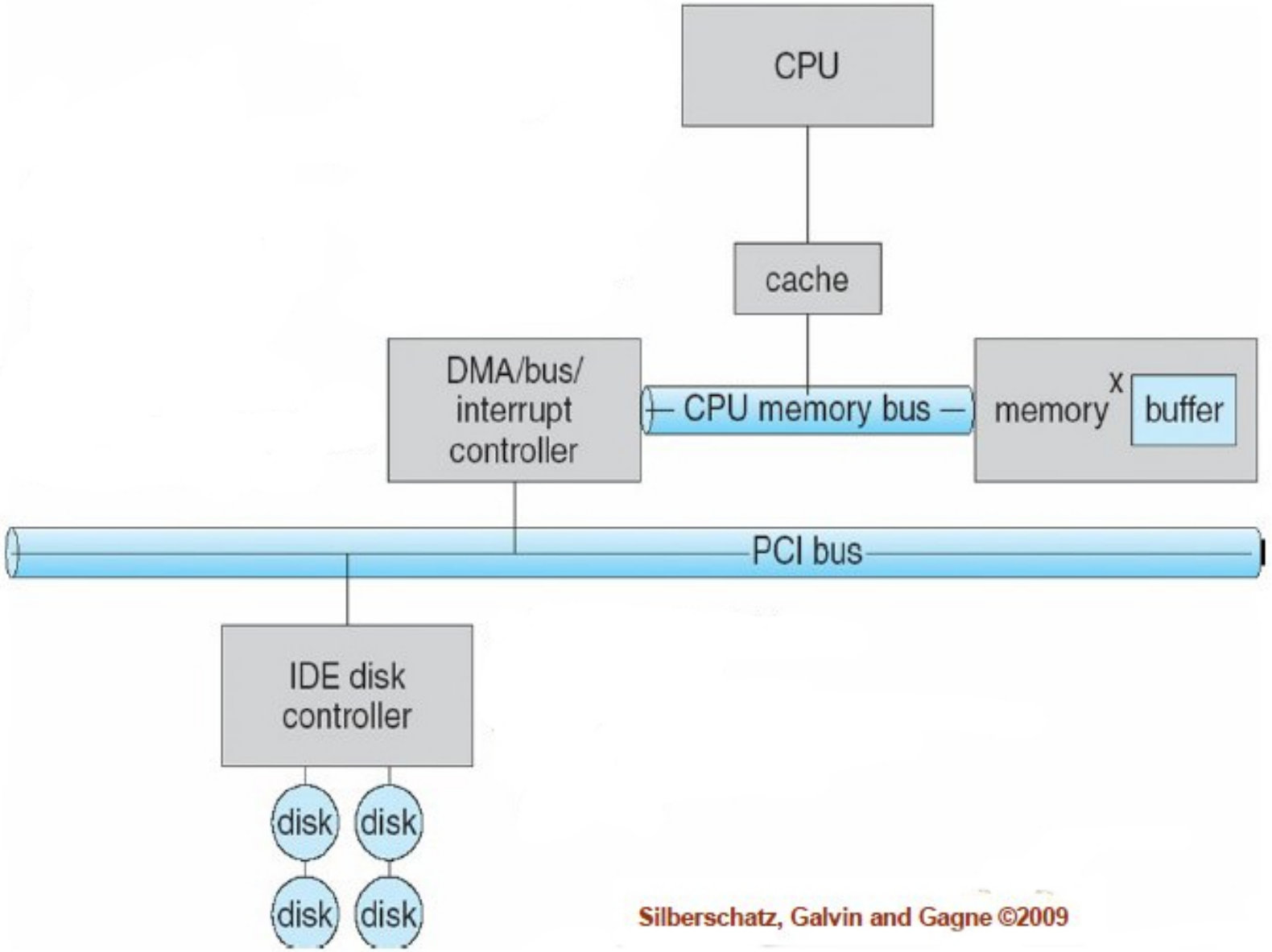
What is Direct Memory Access?

DMA is a capability provided by some computer bus architectures, including PCI, PCMCIA and CardBus, which allows data to be sent directly from an attached device to the memory on the host, freeing the CPU from involvement with the data transfer and thus improving the host's performance.

DMA Programming

The programming of a device's DMA controller is hardware specific. Normally, the OS needs to have the local device address, the physical memory address on the PC, and the size of the memory block to transfer.
Then the register that initiates the transfer is set.

Introduction



Introduction

Elaboration

DMA transfers overcome the problem of occupying the CPU for the entire time it's performing a transfer.

The CPU initiates the transfer, then it executes other ops while the transfer is in progress, finally it receives an interrupt from the DMA controller when the transfer is done

Hardware using DMA: disk drives, graphics cards, network cards, sound cards

DMA can lead to cache coherency problems
If a CPU has a cache and external memory, then the data the DMA controller has access to (stored in RAM) may not be updated with the correct data stored in the cache.

Introduction

Cache Coherency Solutions

Cache-coherent systems:

external writes are signaled to the cache controller which performs a cache invalidation for incoming DMA transfers or cache flush for outgoing DMA transfers (done by hardware)

Non-coherent systems:

OS ensures that the cache lines are flushed before an outgoing DMA transfer is started and invalidated before a memory range affected by an incoming DMA transfer is accessed. The OS makes sure that the memory range is not accessed by any running threads in the meantime.

DMA Transfer

Overview (input example)

- software: asks for data (e.g. read called)

1. Device driver allocates a DMA buffer, sends signal to device indicating where to send the data, sleeps
2. Device writes data to DMA buffer, raises interrupt when finished
3. Interrupt handler gets data from DMA buffer, acknowledges interrupt, awakens software to process the data

DMA Transfer

Overview (input example)

- hardware: asynchronously pushes data to the system
  May push data even if no process is listening!

1. hardware raises an interrupt to announce that new data has arrived
2. interrupt handler allocates a buffer, tells the hardware where to transfer the data
3. device writes the data to the buffer, raises another interrupt when transfer is done
4. interrupt handler dispatches the new data, awakens any relevant process, and takes care of housekeeping

DMA Transfer

Overview (input example – network transfers)

- hardware: must deal with continuous data flow

  Use a circular (ring) buffer in memory shared by device and the processor

  1. incoming packet placed in next available buffer In the ring
  2. interrupt is raised by the device
  3. device driver sends packet to kernel code that will process it
  4. device driver inserts a new buffer into the ring (note: buffer allocation occurs at initialization so the insertion of a buffer is just the selection of an already allocated buffer for the ring)

DMA Transfer

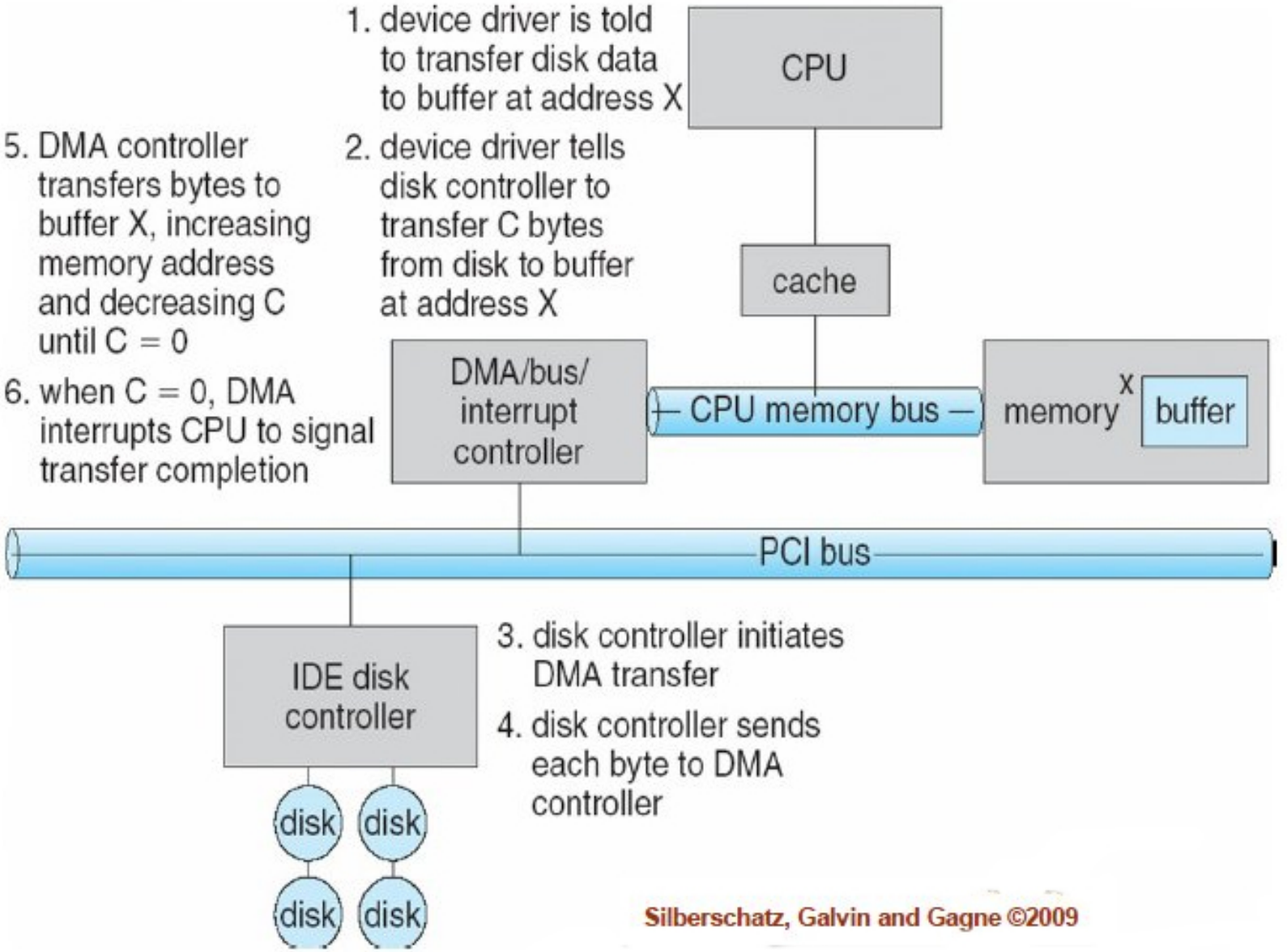
Overview (input example – network transfers)

- pseudo code for read using circular buffer

```
Err_Type Procedure read(data_type data) {
  if (empty_flag) return (ERR_BUF_EMPTY);
  else {
    memory(read_ptr) = data;
    full_flag = false;
    read_ptr = (((read_ptr-B)+1)mod N)+B;
    if (read_ptr == write_ptr) empty_flag = true;
    return(Err-OK);
  }
}
```

DMA Transfer

Graphically:



Allocating a DMA Buffer

Allocation Buffers:

contiguous buffer: contiguous block of memory allocated

scatter/gather: allocated buffer can be fragmented in the physical memory and does not need to be allocated contiguously. The allocated physical memory blocks are mapped to a contiguous buffer in the calling process's virtual address space, thus enabling easy access to the allocated physical memory blocks.

bounce buffer: to allow devices with limited addressing to access all of virtual address space. A bounce buffer resides in memory low enough for a device to copy from and write data to. It is then copied to the desired user page in high memory

Allocating a DMA Buffer

Considerations PCI:

- Must occupy contiguous pages in physical memory due to PCI bus requirement using physical addresses
- Note: other buses (e.g. Sbus) use virtual memory addresses for the transfer
- The right kind of memory must be allocated since not all memory zones are suitable: high memory may not work with some devices because of limited address space
- Memory can be allocated at boot or runtime but drivers can only allocate at runtime

Allocating a DMA Buffer

How to Allocate (low level):

- Use mem=xxx to reserve memory beginning at xxx
  - done at boot

  Then use

  ```
  dmabuf = ioremap(0xFF00000, 0x100000);
  ```

  to create the buffer in the driver

- Allocate from the DMA zone (16 MB)

  ```
  struct page *buf;
  buf = alloc_pages(__GFP_DMA, 3);
  ```

  where 3 is the order and means (1 << 3) pages (from the buddy system allocator)

- Use scatter/gather I/O if the device supports it

Allocating a DMA Buffer

How to Allocate (generic DMA layer):

- DMA based hardware uses bus addresses
  - these do not match with PCI physical addresses
- Can convert at low-level but then risk inconsistency across systems
- Also, different systems have different expectations of cache coherency – hence allocate at generic layer
- Device may have DMA limitations – 32 bit addressing is assumed but may be changed with

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

(e.g. for 24 bit addressing: mask = 0xFFFFFF)

Allocating a DMA Buffer

Creating a buffer and a bus address for the device:

- If the device has an IOMMU a set of mapping registers is provided
- A *bounce buffer* may be necessary – if a driver tries to perform DMA on an address that is not reachable by dev
- Cache coherency:
  copies of recently accessed memory areas are in cache
  if device writes to memory, cache area is invalidated
  so it will have to be paged in
  If device reads data from memory, cache flushed out first
- Generic DMA layer ensures all of above are not a problem over many architectures provided some rules are obeyed

Allocating a DMA Buffer

Rules for generic DMA layer – coherent mapping:

- Data type `dma_addr_t` represents a bus address
  It *must not* be manipulated by the driver – it can be passed to the device
- Set up the mapping (and buffer) with this:

```
void *dma_alloc_coherent(struct device *, size_t, dma_addr_t *, int);
```

  allocates uncached, unbuffered memory for a device for performing DMA. Allocates pages, returns the CPU-viewed (virtual) address, and sets the third arg to the device-viewed address. Buffer is automatically placed where the device can get at it.

- Free the mapping with this:

```
void dma_free_coherent(struct device *, size_t, void *, dma_addr_t);
```

Allocating a DMA Buffer

Rules for generic DMA layer – streaming DMA mapping:

- Different from coherent mapping because the mappings deal with addresses that were chosen apriori. If IOMMU exists, registers may hogged up by coherent mapping, but streaming is single shot, registers are released
- Set up a single buffer with this:

```
dma_addr_t dma_map_single(struct device*, void*, size_t,
                          enum dma_data_direction);
```

Direction choices: DMA_TO_DEVICE, DMA_FROM_DEVICE DMA_BIDIRECTIONAL
`void*` is the buffer virtual address, `dma_addr_t` is the Returned bus address for the device

- Free the mapping with this:

```
void dma_unmap_single(struct device*, dma_addr_t, size_t,
                      enum dma_data_direction);
```

Allocating a DMA Buffer

Rules for generic DMA layer – streaming DMA mapping:

- Buffer can only be used in the direction specified
- A mapped buffer belongs to the device, not the processor
  The device driver *must* keep hands off the buffer until it is unmapped
- A buffer used to send data to a device *must* contain the data before it is mapped
- The buffer *must not* be unmapped while DMA is still active, or serious system instability is guaranteed.

Allocating a DMA Buffer

Rules for generic DMA layer – scatter/gather mapping:

- Send contents of several buffers over DMA
  Could send then one at a time: map each
  Or with scatter/gather, can send them all at once (speed)
- Many devices can accept a scatterlist of array pointers and lengths
- But scatterlist entries *must* be of page size (except ends)

Using a scatterlist DMA transfer

Steps:

- ```
  struct scatterlist { /* in asm/scatterlist.h */
      unsigned long  page_link;
      unsigned int   offset;
      unsigned int   length;
      dma_addr_t     dma_address;
      unsigned int   dma_length;
  };
  ```

- Create array `sg` of scatterlist entries and fill them in
  Let `nent` be the number of entries

- Call

  ```
  int dma_map_sg(device, sg, nent, direction)
  ```

  returns the number of DMA buffers to send (<= `nent`)

- When done do this:

  ```
  void dma_unmap_sg(device, sg, nent, direction);
  ```
  ```

PCI Double Address Cycle (DAC)

- The PCI bus supports 64-bit addressing (DAC)
- The generic DMA layer does not support this mode
  - it is a PCI-specific feature
  - many implementations of DAC are buggy
  - DAC is slower than a regular 32-bit DMA

- But DAC could be useful with large buffer in high memory then PCI-specific routines must be used

- Use a mask `m` (must return 0 to continue)

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 m);
```

- Call this to map:

```
dma64_addr_t pci_dac_page_to_dma
  (struct pci_dev *pdev, struct page *page,
   unsigned long offset, int direction);
```

- No need to unmap – no external resources used (mapping pages directly)

Examples

- dad.c
- usb_skel.c