Integrating DMA Into the Generic Device Model

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com
James.Bottomley@steeleye.com

Abstract

This paper will introduce the new DMA API for the generic device model, illustrating how it works and explaining the enhancements over the previous DMA Mapping API. In a later section we will explain (using illustrations from the PA-RISC platform) how conversion to the new API may be achieved hand in hand with a complete implementation of the generic device API for that platform.

1 Introduction

Back in 2001, a group of people working on non-x86 architectures first began discussing radical changes to the way device drivers make use of DMA. The essence of the proposal was to mandate a new DMA Mapping API[1] which would be portable to all architectures then supported by Linux. One of the forces driving the adoption of this new API was the fact that the PCI bus had expanded beyond the x86 architecture and being embraced by non-x86 hardware manufacturers. Thus, one of the goals was that any driver using the DMA Mapping API should work on *any* PCI bus independent of the underlying microprocessor architecture. Therefore, the API was phrased entirely in terms of the PCI bus, since PCI driver compatibility across architectures was viewed as a desirable end result.

1.1 Legacy Issues

One of the issues left unaddressed by the DMA Mapping API was that of legacy buses: Most non-x86 architectures had developed other bus types prior to the adoption of PCI (e.g. sbus for the sparc; lasi and gsc bus for PA-RISC) which were usually still present in PCI based machines. Further, there were other buses that migrated across architectures prior to PCI, the most prominent being EISA. Finally, some manufacturers of I/O chips designed them not to be bus based (the LSI 53c7xx series of SCSI chips being a good example). These chips made an appearance in an astonishing variety of cards with an equal variety of bus interconnects.

The major headache for people who write drivers for non-PCI or multiple bus devices is that there was no standard for non-PCI based DMA, even though many of the problems encountered were addressed by the DMA Mapping API. This gave rise to a whole hotchpotch of solutions that differed from architecture to architecture: On Sparc, the DMA Mapping API has a completely equivalent SBUS API; on PA-RISC, one may obtain a "fake" PCI object for a device residing on a non-PCI bus which may be passed straight into the PCI based DMA API.

1.2 The Solution

The solution was to re-implement the DMA Mapping API to be non bus specific. This goal was vastly facilitated by the new generic device architecture[5] which was also being implemented in the 2.5 Linux kernel and which finally permitted the complete description of device and bus interconnections using a generic template.

1.3 Why A New API

After all, apart from legacy buses, PCI is the one bus to replace all others, right? so an API based on it must be universally applicable?

This is incorrect on two counts. Firstly, support for legacy devices and buses is important to Linux, since being able to boot on older hardware that may have no further use encourages others who would not otherwise try Linux to play with it, and secondly there are other

new non-PCI buses support for which is currently being implemented (like USB and firewire).

1.4 Layout

This paper will describe the problems caused by CPU caches in section 2, move on to introducing new struct device based DMA API[2] in section 3 and describe how it solves the problems, and finally in section 4 describe how the new API may be implemented by platform maintainers giving specific examples from the PA-RISC conversion to the new API.

2 Problems Caused By DMA

The definition of DMA: Direct Memory Access means exactly that: direct access to memory (without the aid of the CPU) by a device transferring data. Although the concept sounds simple, it is fraught with problems induced by the way a CPU interacts with memory.

2.1 Virtual Address Translation

Almost every complex CPU designed for modern Operating Systems does some form of Virtual Address Translation. This translation, which is usually done inside the CPU, means that every task running on that CPU may utilise memory as though it were the only such task. The CPU transparently assigns each task overlapping memory in virtual space, but quietly maps it to unique locations in the physical address space (the memory address appearing on the bus) using an integral component called a MMU (Memory Management Unit).

Unfortunately for driver writers, since DMA transfers occur without CPU intervention, when a device transfers data directly to or from memory, it must use the physical memory address (because the CPU isn't available to translate any virtual addresses). This problem isn't new, and was solved on the x86 by using functions which performed the same lookups as the CPU's MMU and could translate virtual to physical addresses and vice versa so that the driver could give the correct addresses physical addresses to the hardware and interpret any addresses returned by the hardware device back into the CPU's virtual space.

However, the problems don't end there. With the advent of 64 bit chips it became apparent that they would still have to provide support for the older 32 bit (and even 24 bit) I/O buses for a while. Rather than cause inconvenience to driver writers by arbitrarily limiting the physical memory addresses to which DMA transfers could be done, some of the platform manufacturers came up with another solution: Add an additional MMU between the I/O buses and the processor buses (This MMU is usually called the IOMMU). Now the physical address limitation of the older 32 bit buses can be hidden because the IOMMU can be programmed to map the physical address space of the bus to anywhere in the `physical` (not virtual) memory of the platform. The disadvantage to this approach is that now this bus physical to memory physical address mapping must be programmed into the IOMMU and must also be managed by the device driver.

There may even be multiple IOMMUs in the system, so a physical address (mapped by a particular IOMMU) given to a device might not even be unique (another device may use the same bus address via a different IOMMU).

2.2 Caching

On most modern architectures, the speed of the processor vastly exceeds the speed of the available memory (or, rather, it would be phenomenally expensive to use memory matched to the speed of the CPU). Thus, almost all CPUs come equipped with a cache (called the Level 1 [L1] cache). In addition, they usually expose logic to drive a larger external cache (called the Level 2 [L2] cache).

In order to simplify cache management, most caches operate at a minimum size called the cache width[1]. All reads and writes from main memory to the cache must occur in integer multiples of the cache width. A common value for the cache width is sixteen bytes; however, higher (and sometimes for embedded processors, lower) values are also known.

The effect of the processor cache on DMA can be extremely subtle. For example, consider a hypothetical processor with a cache width of sixteen bytes. Referring to figure 1, supposing I read a byte of data at address `0x18`. Because of the cache burst requirement, this will bring the address range `0x10` to `0x1f` into the cache. Thus, any subsequent read of say `0x11` will be satisfied from the cache without any reference to main mem-

[1] also called the "cache line size" in the PCI specification

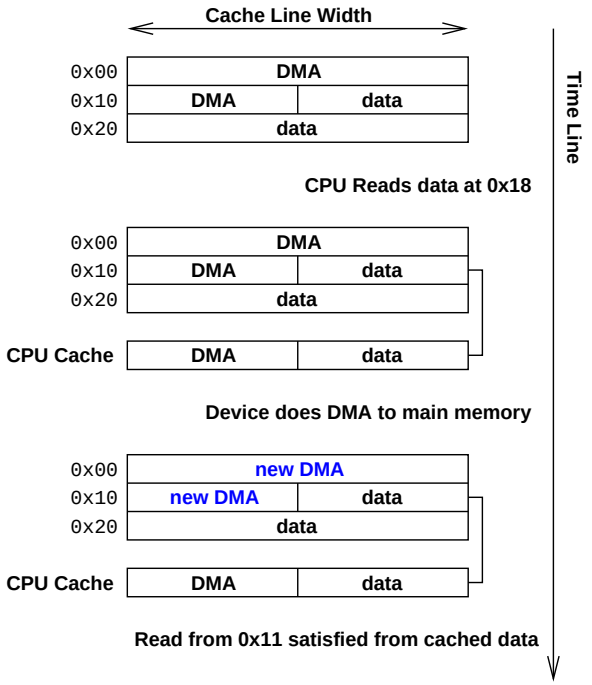

Figure 1: Incorrect data read because of cache effects

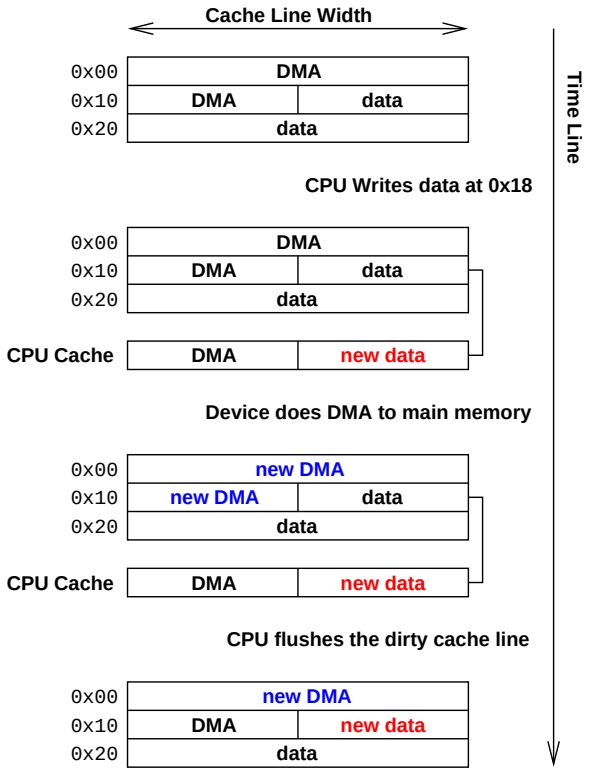

Figure 2: Data destruction by interfering device and cache line writes

ory. Unfortunately, If I am reading from `0x11` because I programmed a device to deposit data there via DMA, the value that I read will not be the value that the device placed in main memory because the CPU believes the data in the cache to be still current. Thus I read incorrect data.

Worse, referring to figure 2, supposing I have designated the region `0x00` to `0x17` for DMA from a device, but then I write a byte of data to `0x19`. The CPU will probably modify the data in cache and mark the cache line `0x10` to `0x1f` dirty, but not write its contents to main memory. Now, supposing the device writes data into `0x00` to `0x17` by DMA (the cache still is not aware of this). However, subsequently the CPU decides to flush the dirty cache line from `0x10` to `0x1f`. This flush will overwrite (and thus destroy) part of the the data that was placed at `0x10` to `0x17` by the DMA from the device.

The above is only illustrative of some of the problems. There are obviously many other scenarios where cache interference effects may corrupt DMA data transfers.

2.3 Cache Coherency

In order to avoid the catastrophic consequences of caching on DMA data, certain processors exhibit a property called “coherency”. This means that they take action to ensure that data in the CPU cache and data in main memory is identical (usually this is done by snooping DMA transactions on the memory bus and taking corrective cache action before a problem is caused).

Even if a CPU isn’t fully coherent, it can usually designate ranges of memory to be coherent (in the simplest case by marking the memory as uncacheable by the CPU). Such architectures are called “partially coherent”.

Finally there is a tiny subset of CPUs that cannot be made coherent by any means, and thus the driver itself must manage the cache so as to avoid the unfortunate problems described in section 2.2.

2.4 Cache Management Instructions

Every CPU that is not fully coherent includes cache management instructions in its repertoire. Although the actual instruction format varies, they usually operate at the level of the cache line and they usually perform one of three operations:

1. Writeback (or flush): causes the cache line to be synced back to main memory (however, the data in the cache remains valid).
2. Invalidate: causes the cache line to be eliminated from the cache (often for a dirty cache line this will cause the contents to be erased). When the cpu next references data in that cache line it will be brought in fresh from main memory.
3. Writeback and Invalidate: causes an atomic sequence of Writeback followed by Invalidate (on some architectures, this is the only form of cache manipulation instruction implemented).

When DMA is done to or from memory that is not coherent, the above instructions must be used to avoid problems with the CPU cache. The DMA API contains an abstraction which facilitates this.

2.5 Other Caches

Although the DMA APIs (both the PCI and generic device ones) are concerned *exclusively* with the CPU cache, there may be other caches in the I/O system which a driver may need to manage. The most obvious one is on the I/O bus: Buses, like PCI, may possess a cache which they use to consolidate incoming writes. Such behaviour is termed "posting". Since there are no cache management instructions that can be used to control the PCI cache (the cache management instructions only work with the CPU cache), the PCI cache employs special posting rules to allow driver writers to control its behaviour. The rules are essentially:

1. Caching will not occur on I/O mapped spaces.
2. The sequence of reads and writes to the memory mapped space will be preserved (although the cache may consolidate write operations).

One important point to note is that there is no defined time limit to hold writes in the bus cache, so if you want a write to be seen by the device it must be followed by a read.

Suffice it to say that the DMA API does not address PCI posting in any form. The basic reason is that in order to flush a write from the cache, a read of somewhere in the device's memory space would have to be done, but only the device (and its driver) know what areas are safe to read.

3 The New DMA API for Driver Writers

By "driver writer", we mean any person who wishes to use the API to manage DMA coherency without wanting to worry about the underlying bus (and architecture) implementation.

This part of the API and its relation to the PCI DMA Mapping API is fairly well described in [2] and also in other published articles [3].

3.1 DMA Masks, Bouncing and IOMMUs

Every device has a particular attachment to a bus. For most, this manifests itself in the number of address lines on the bus that the device is connected to. For example, old ISA type buses were only connected (directly) to the first twenty four address lines. Thus they could only directly access the first sixteen megabytes of memory. If you needed to do I/O to an address greater than this, the transfer had to go via a bounce buffer: e.g. data was received into a buffer guaranteed to be lower than sixteen megabytes and then copied into the correct place. This distinction is the reason why Linux reserves a special memory zone for legacy (24 bit) DMA which may be accessed using flag (`GFP_DMA`) or'd into the allocation flags.

Similarly, a 32 bit PCI bus can only access up to the first four gigabytes of main memory (and even on a 64 bit PCI bus, the device may be only physically connected to the first 32 or 24 address lines).

Thus, the concept of a *dma mask* is used to convey this information. The API for setting the dma mask is:

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

- `dev`—a pointer to the generic device.
- `mask`—a representation of the bus connection. It is a bitmap where a 1 means the line is connected and a zero means it isn't. Thus, if the device is only connected to the first 24 lines, the `mask` will be `0xffffff`.
- returns true if the bus accepted the mask.

Note also that if you are driving a device capable of addressing up to 64 bits, you must also be aware that the

bus it is attached to may not support this (i.e. you may be a 64 bit PCI card in a 32 bit slot). So when setting the DMA mask, you must start with the value you want but be prepared that you may get a failure because it isn't supported by the bus. For 64 bit devices, the convention is to try 64 bits first but if that fails set the mask to 32 bits.

Once you have set the DMA mask, the troubles aren't ended. If you need transfers to or from memory outside of your DMA mask to be bounced, you must tell the block[2] layer using the

```
void blk_queue_bounce_limit (re-
quest_queue_t *q, u64 mask)
```

- `q`—the request queue your driver is connected to.
- `mask`—the mask (again 1s for connected addresses) that an I/O transfer will be bounced if it falls outside of (i.e. `address & mask != address`)

function that it should take care of the bouncing. Note, however, that bouncing *only* needs to occur for buses without an IOMMU. For buses with an IOMMU, the mask serves only as an indication to the IOMMU of what the range of physical addresses available to the device is. The IOMMU is assumed to be able to address the full width of the memory bus and therefore transfers to the device need not be bounced by the block layer.

Thus, the driver writer must know whether the bus is mapped through an IOMMU or not. The means for doing this is to test the global `PCI_DMA_BUS_IS_PHYS` macro. If it is true, the system generally has no IOMMU[3] and you should feed the mask into the block bounce limit. If it is false, then you should supply `BLK_BOUNCE_ANY` (informing the block layer that no bouncing is required).

3.2 Managing block layer DMA transfers

It is the responsibility of the device driver writer to manage the coherency problems in the CPU cache when transferring data to or from a device and also mapping between the CPU virtual address and the device physical address (including programming the IOMMU if such is required).

By and large, most block devices are simply transports: they move data from user applications to and from storage without much concern for the actual contents of the data. Thus they can generally rely on the CPU cache management implicit in the APIs for DMA setup and tear-down. They only need to use explicit cache management operations if they actually wish to access the data they are transferring. The use of device private areas for status and messaging is covered in sections 3.5 and 3.6.

For setup of a single *physically contiguous*[4] DMA region, the function is

```
dma_addr_t dma_map_single (struct
device *dev, void *ptr, size_t
size, enum dma_data_direction di-
rection)
```

- `ptr`—pointer to the physically contiguous data (virtual address)
- `size`–the size of the physically contiguous region
- `direction`—the direction, either to the device, from the device or bidirectional (see section 3.4 for a complete description)
- returns a bus physical address which may be passed to the device as the location for the transfer

and the corresponding tear-down after the transfer is complete is achieved via

```
void dma_unmap_single (struct
device *dev, dma_addr_t
dma_addr, size_t size, enum
dma_data_direction direction)
```

- `dma_addr`—the physical address returned by the mapping setup function
- `size`, `direction`—the exact values passed into the corresponding mapping setup function

[2]Character and Network devices have their own ways of doing bouncing, but we will consider only the block layer in the following

[3]this is an inherent weakness of the macro. Obviously, it is possible to build a system where some buses go via an IOMMU and some do not. In a future revision of the DMA API, this may be made a device specific macro

[4]physically contiguous regions of memory for DMA can be obtained from `kmalloc()` and `__get_free_pages()`. They may specifically *not* be allocated on the stack (because data destruction may be caused by overlapping cache lines, see section 2.2) or `vmalloc()`

The setup and tear-down functions also take care of all the necessary cache flushes associated with the DMA transaction[5].

3.3 Scatter-Gather Transfers

By and large, almost any transfer that crosses a page boundary will not be contiguous in physical memory space (because each contiguous page in virtual memory may be mapped to a non-contiguous page in physical memory) and thus may not be mapped using the API of section 3.2. However, the block layer can construct a list of each separate page and length in the transfer. Such a list is called a Scatter-Gather (SG) list. The device driver writer must map each element of the block layer's SG list into a device physical address.

The API to set up a SG transfer for a given `struct request *req` is

```
int blk_rq_map_sg (request_queue_t
*q, struct request *req, struct
scatterlist *sg)
```

- `q`—the queue the request belongs to
- `sg`—a pointer to a pre allocated physical scatterlist which must be at least `req->nr_phys_segments` in size.
- returns the number of entries in `sg` which were actually used

Once this is done, the SG list may be mapped for use by the device:

```
int dma_map_sg(struct device *dev,
struct scatterlist *sg, int nents,
enum dma_data_direction direction)
```

- `sg`—a pointer to a physical scatterlist which was filled in by
- `nents`—the allocated size of the SG list.
- `direction`—as per the API in section 3.2
- returns the number of entries of the `sg` list actually used. This value must be less than or equal to `nents` but is otherwise not constrained (if the system has an IOMMU, it may chose to do all SG inside the IOMMU mappings and thus always return just a single entry).
- returns zero if the mapping failed.

Once you have mapped the SG list, you may loop over the number of entries using the following macros to extract the busy physical addresses and lengths

```
dma_addr_t sg_dma_address (struct
scatterlist *sge)
```

- `sge`–pointer to the desired entry in the SG list
- returns the busy physical DMA address for the given SG entry

```
unsigned int sg_dma_len (struct
scatterlist *sge)
```

- returns the length of the given SG entry

and program them into the device's SG hardware controller. Once the SG transfer has completed, it may be torn down with

```
void dma_unmap_sg (struct device
*dev, struct scatterlist *sg, int
nents, enum dma_data_direction
direction)
```

- `nents` should be the number of entries passed in to `dma_map_sg()` *not* the number of entries returned.

3.4 Accessing the Data Between Mapping and Unmapping

Since the cache coherency is normally managed by the mapping and unmapping API, you may not access the data between the map and unmap without first synchronizing the CPU caches. This is done using the DMA synchronization API. The first

```
void dma_sync_single(struct
device *dev, dma_addr_t
dma_handle, size_t size, enum
dma_data_direction direction)
```

[5] they use the `direction` parameter to get this right, so be careful when assigning directions to ensure that they are correct

- dma_handle—the physical address of the region obtained by the mapping function.
- size—The size of the region passed into the mapping function.

synchronizes only areas mapped by the dma_map_single API, and thus only works for physically contiguous areas of memory. The other

```
void dma_sync_sg (struct device
*dev, struct scatterlist *sg, int
nelems, enum dma_data_direction
direction)
```

- The parameters should be identical to those passed in to dma_map_sg

synchronizes completely a given SG list (and is, therefore, rather an expensive operation). The correct point in the driver code to invoke these APIs depends on the direction parameter:

- DMA_TO_DEVICE—Usually flushes the CPU cache. Must be called *after* you last modify the data and *before* the device begins using it.
- DMA_FROM_DEVICE—Usually invalidates the CPU cache. Must be called *after* the device has finished transferring the data and *before* you first try to read it.
- DMA_BIDIRECTIONAL—Usually does a write-back/invalidate of the CPU cache. Must be called *both* after you finish writing it but before you hand the data to the device *and* after the device finishes with it but before you read it.

3.5 API for coherent and partially coherent architectures

Most devices require a control structure (or a set of control structures), to facilitate communication between the device and its driver. For the driver and its device to operate correctly on an arbitrary platform, the driver would have to insert the correct cache flushing and invalidate instructions when exchanging data using this.

However, almost every modern platform has the ability to designate an area of memory as coherent between the processor and the I/O device. Using such a coherent area, the driver writer doesn't have to worry about synchronising the memory. The API for obtaining such an area is

```
void * dma_alloc_coherent
(struct device *dev, size_t size,
dma_addr_t *dma_handle, int flag)
```

- dev—a pointer to the generic device
- size—requested size of the area
- dma_handle—a pointer to the area the physically usable address will be placed (i.e. this should be the address given to the device for the area)
- flag—a memory allocation flag. Either GFP_KERNEL if the allocation may sleep while finding memory or GFP_ATOMIC if the allocation may not sleep
- returns the virtual address of the area or NULL if no coherent memory could be allocated

Note that coherent memory may be a constrained system resource and thus NULL may be returned even for GFP_KERNEL allocations.

It should also be noted that the tricks platforms use to obtain coherent memory may be quite expensive, so it is better to minimize the allocation and freeing of these areas where possible.

Usually, device drivers allocate coherent memory at start of day, both for the above reason and so an in-flight transaction will not run into difficulties because the system is out of coherent memory.

The corresponding API for releasing the coherent memory is

```
void dma_free_coherent (struct
device *dev, size_t size, void
*vaddr, dma_addr_t dma_handle)
```

- vaddr—the virtual address returned by dma_alloc_coherent
- dma_handle—the device physical address filled in at allocation time

3.6 API for fully incoherent architectures

This part of the API has no correspondance with any piece of the old DMA Mapping API. Some platforms (fortunately usually only older ones) are incapable of producing any coherent memory at all. Even worse, drivers which may be required to operate on these platforms usually tend to have to also operate on platforms which can produce coherent memory (and which may operate more efficiently if it were used). In the old API, this meant it was necessary to try to allocate coherent memory, and if that failed allocate and map ordinary memory. If ordinary memory is used, the driver must remember that it also needs to enforce the sync points. This leads to driver code which looks like

```
memory = pci_alloc_coherent(...);
if (!memory) {
  dev->memory_is_not_coherent = 1;
  memory = kmalloc(...);
  if (!memory)
    goto fail;
  pci_map_single(...);
}

....

if (dev->memory_is_not_coherent)
  pci_dma_sync_single(...);
```

Which cannot be optimized away. The noncoherent allocation additions are designed to make this code more efficient, and to be optimized away at compile time on platforms that can allocate coherent memory.

```
void *dma_alloc_noncoherent
(struct device *dev, size_t size,
dma_addr_t *dma_handle, int flag)
```

- the parameters are identical to those of `dma_alloc_coherent` in section 3.5.

The difference here is that the driver *must* use a special synchronization API[6] to synchronize this area between data transfers

```
dma_cache_sync (void
*vaddr, size_t size, enum
dma_data_direction direction)
```

- `vaddr`—the virtual address of the memory to sync (this need not be at the beginning of the allocated region)
- `size`—the size of the region to sync (again, this may be less than the allocated size)
- `direction`—see section 3.4 for a discussion of how to use this.

Note that the placement of these synchronization points should be exactly as described in section 3.4. The platform implementation will choose whether coherent memory is actually returned. However, if coherent memory is returned, the implementation will take care of making sure the synchronizations become nops (on a fully coherent platform, the synchronizations will compile away to nothing).

Using this API, the above driver example becomes

```
memory = dma_alloc_noncoherent(...);
if (!memory)
  goto fail;

...

dma_cache_sync(...);
```

The (possibly) non-coherent memory area is freed using

```
void dma_free_noncoherent(struct
device *dev, size_t size, void
*vaddr, dma_addr_t dma_handle)
```

- The parameters are identical to those of `dma_free_coherent` in section 3.4

Since there are very few drivers that need to function on fully non-coherent platforms, this API is of little use in modern systems.

3.7 Other Extensions in the New API

There are two other extensions over the old DMA Mapping API. They are

[6] The driver could also use the API of section 3.4, but the point of having a separate one is that it may be optimized away on platforms that are partially non-coherent

```
int dma_get_cache_alignment (void)
```

- returns the cache alignment width of the platform (see section 2.2). Note, the value returned guarantees only to be a power of two and greater than or equal to the current processor cache width. Thus its value may be relied on to separate data variables where I/O caching effects would destroy data.

```
int dma_is_consistent (dma_addr_t
dma_handle)
```

- returns true if the physical memory area at `dma_handle` is coherent

And

```
void dma_sync_single_range(struct
device *dev, dma_addr_t
dma_handle, unsigned long
offset, size_t size, enum
dma_data_direction direction)
```

- `offset`—the offset from the `dma_handle`
- `size`—the size of the region to be synchronized

which allows a partial synchronization of a mapped region. This is useful because on most CPUs, the cost of doing a synchronization is directly proportional to the size of the region. Using this API allows the synchronization to be restricted only to the necessary parts of the data.

4 Implementing the DMA API for a Platform

In this section we will explore how the DMA API should be implemented from a platform maintainer's point of view. Since implementation is highly platform specific, we will concentrate on how the implementation was done for the HP PA-RISC[4] platform.

4.1 Brief Overview of PA-RISC

An abridged version of a specific PA-RISC architecture[7] is given in figure 3. It is particularly instructive to note

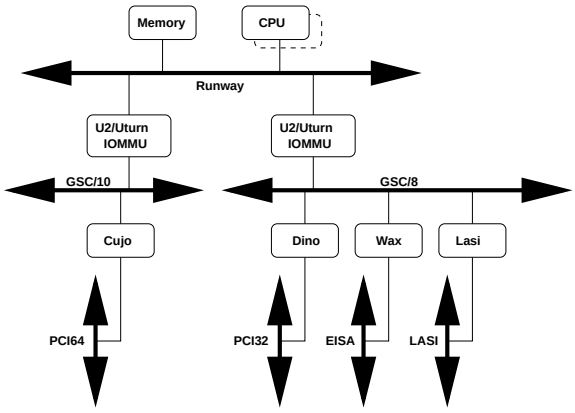

Figure 3: An abridged example of PA-RISC architecture

that the actual chips involved (Cujo, U2/Uturn, Wax etc.) vary from machine to machine, as do the physical bus connections, so the first step that was required for PA-RISC was to build a complete model of the device layout from the runway bus on down in the generic device model[5].

4.2 Converting to the Device Model

Since PA-RISC already had its own device type (`struct parisc_device`), it was fairly simple to embed a generic device in this, assign it to a new `parisc_bus_type` and build up the correct device structure. For brevity, instead of giving all the PA-RISC specific buses (like Lasi, GSC, etc) their own bus type, they were all simply assigned to the `parisc_bus_type`.

The next problem was that of attaching the PCI bus. Previously, PCI buses could only have other PCI buses as parents, so a new API[8] was introduced to allow parenting a PCI bus to an arbitrary generic device. At the same time, others were working on bringing the EISA bus under the generic device umbrella [6].

With all these core and architecture specific changes, PA-RISC now has a complete generic device model layout of all of its I/O components and is ready for full conversion to the new DMA API.

[7] The illustration is actually from a C360 machine, and does not show every bus in the machine

[8] `pci_scan_bus_parented()`

4.3 Converting to the DMA API

Previously for PA-RISC, the U2/Uturn (IOMMU) information was cached in the PCI device `sysdata` field and was placed there at bus scan time. Since the bus scanning was done from the PA-RISC specific code, it knew which IOMMU the bus was connected to. Unfortunately, this scheme doesn't work for any other bus type, so an API[9] was introduced to obtain a fake PCI device for a given `parisc_device` and place the correct IOMMU in the `sysdata` field. PA-RISC actually has an architecture switch (see `struct hppa_dma_ops` in `asm-parisc/dma-mapping.h`) for the DMA functions. All the DMA functions really need to know is which IOMMU the device is connected to and the device's `dma_mask`, making conversion quite easy since the only PCI specific piece was extracting the IOMMU data.

Obviously, in the generic device model, the field `platform_data` is the one that we can use for caching the IOMMU information. Unfortunately there is no code in any of the scanned buses to allow this to be populated at scan time. The alternative scheme we implemented was to take the generic device passed into the DMA operations switch and, if `platform_data` was `NULL`, walk up the `parent` fields until the IOMMU was found, at which point it was cached in the `platform_data` field. Since this will now work for every device that has a generic device (which is now every device in the PA-RISC system), the fake PCI device scheme can be eliminated, and we have a fully implemented DMA API.

4.4 The Last Wrinkle—Non-Coherency

There are particular PA-RISC chips (the PCX-S and PCX-T) which are incapable of allocating any coherent memory at all. Fortunately, none of these chips was placed into a modern system,or indeed into a system with an IOMMU, so all the buses are directly connected.

Thus, an extra pair of functions was added to the DMA API switch for this platform which implemented noncoherent allocations as a `kmalloc()` followed by a map, and also for the `dma_cache_sync()` API. On the platforms that are able to allocate coherent memory, the noncoherent allocator is simply the coherent one, and the cache sync API is a nop.

[9] `ccio_get_fake()`

5 Future Directions

The new DMA API has been successful on platforms that need to unify the DMA view of disparate buses. However, the API as designed is really driver writer direct to platform. There are buses (USB being a prime example) which would like to place hooks to intercept the DMA transactions for programming DMA bridging devices that are bus specific rather than platform specific. Work still needs doing to integrate this need into the current framework.

Acknowledgements

I would like to thank Grant Grundler and Matthew Wilcox from the PA-RISC linux porting team for their help and support transitioning PA-Linux to the generic device model and subsequently the new DMA API. I would also like to thank HP for their donation of a PA-RISC C360 machine and the many people who contributed to the email thread[7] that began all of this.

References

[1] David S. Miller Richard Henderson Jakub Jelinek *Dynamic DMA Mapping* Linux Kernel 2.5 `Documentation/DMA-mapping.txt`

[2] James E.J. Bottomley *Dynamic DMA mapping using the generic device* Linux Kernel 2.5 `Documentation/DMA-API.txt`

[3] Jonathan Corbet *Driver Porting: DMA Changes* Linux Weekly News `http://lwn.net/Articles/28092`

[4] The PA-RISC linux team `http://www.parisc-linux.org`

[5] Patrick Mochel *The (New) Linux Kernel Driver Model* Linux Kernel 2.5 `Documentation/driver-model/*.txt`

[6] Marc Zyngier *sysfs stuff for EISA bus* `http://marc.theaimsgroup.com/?t=103696564400002`

[7] James Bottomley *[RFC] generic device DMA implementation* `http://marc.theaimsgroup.com/?t=103902433500007`