Integrating DMA Into the Generic Device Model

James Bottomley

SteelEye Technology

26 July 2003

In the Beginning

- There was Programmed I/O (PIO).
- The processor coped with all the quirky device timings and spoon fed the device a byte/word/quad at a time at the correct rate.
- This was very, very slow.
- And wasted an awful lot of CPU cycles doing timings.
- But then if you have old IDE you already know this.
- Then came...DMA.

What Is DMA?

- Direct Memory Access (DMA) is simply the ability of an I/O device to read or write memory directly *without* the intervention of the Processor.
- Fundamentally, all devices that transfer reasonable amounts of data need to use DMA.
- Because the processor isn't required, it can perform other tasks while the DMA is going on (well, as long as the DMA doesn't lock it off the memory bus).
- Principal problem is that the kernel thinks in terms of "virtual" addresses and DMA must be performed to "physical" addresses.

DMA Just Works, Doesn't It?

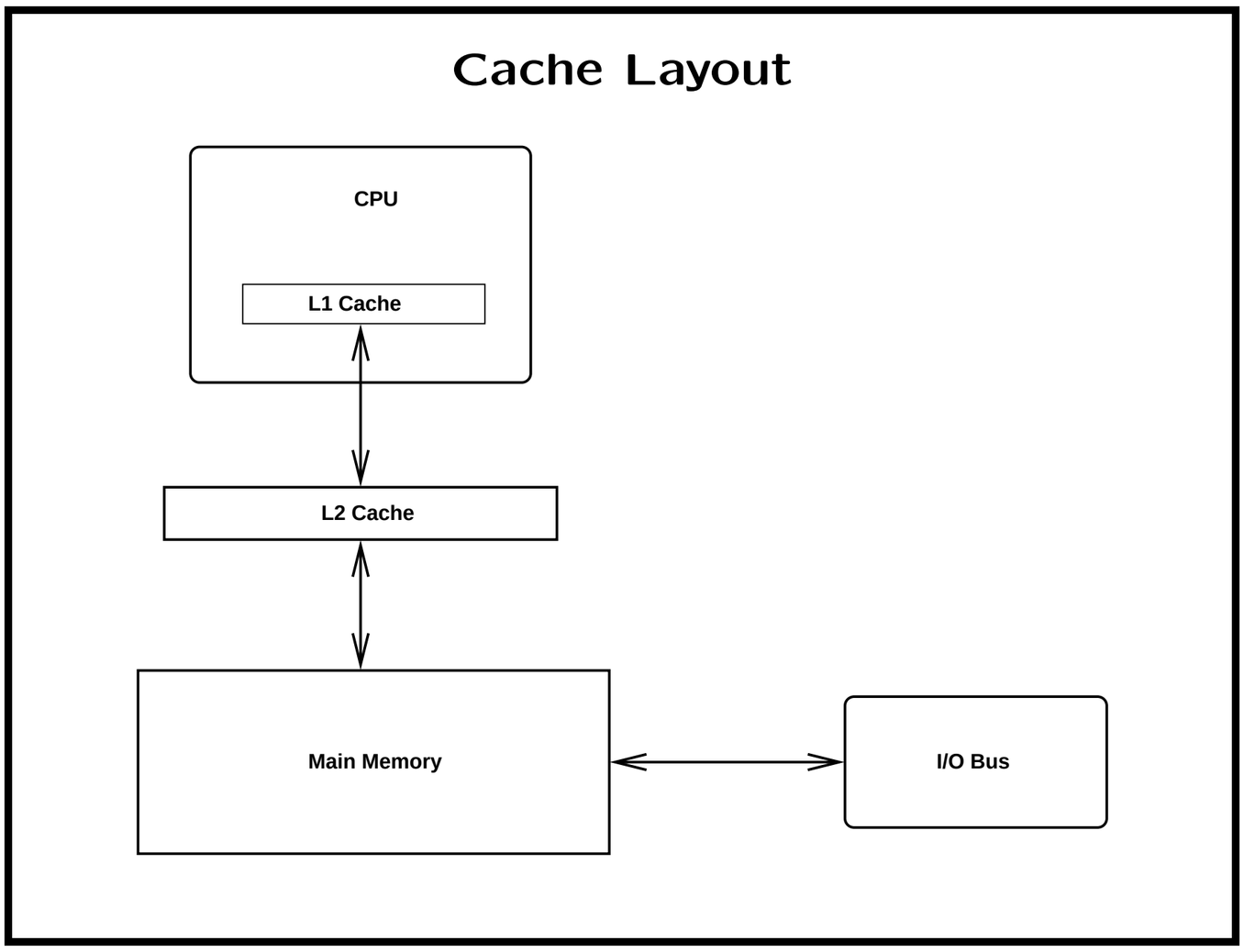
- In 2.2 and early 2.4, DMA was controlled simply by two APIs:
  - `virt_to_bus`
  - `bus_to_virt`
- and it just worked.
- Everyone was happy.
- in later 2.4 an entire new (and complex) DMA Mapping API was introduced, necessitating the conversion of every driver in the kernel.

What is Wrong With the Original Approach?

- The basic problems is that the simple approach really **only** works on x86.
- Worse, it only really works on x86 up to 4GB of memory.
- It only apparently worked for non-x86—a lot of work went on under the covers to give this appearance.
- In 2.4, the decision was finally made to tackle this properly, hence the DMA Mapping API.

What are the Critical Problems?

- Cache Coherency
  - The coherency of the processor cache may need to be managed by the driver (most non-x86 architectures).
- IO-MMUs
  - The whole virtual memory problem occurs because the CPU has a Memory Management Unit to do address translation
  - Can put one of these MMUs across the I/O bus (as well as on the CPU)
  - This means that I/O drivers now need to manage the mappings for this IO-MMU.


Cache Layout
CPU
L1 Cache
L2 Cache
Main Memory
I/O Bus

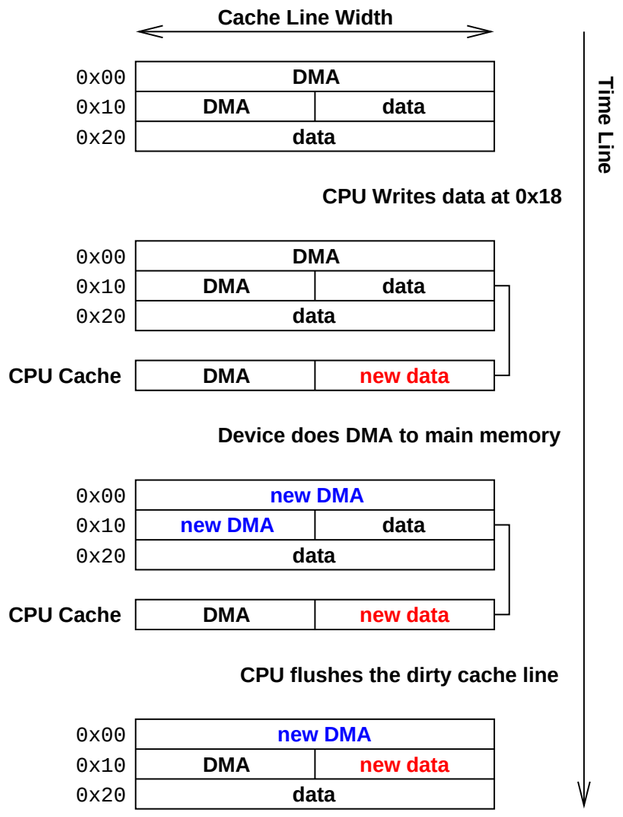
Cache Coherency Problems

- Any write From I/O to memory may change data that is stored in one of the CPUs caches.
- If the CPU doesn't watch for this and manage its caches accordingly, it must explicitly be made aware of the changes using cache management instructions.
  - **invalidate** simply evicts a cache line from the CPU cache.
  - **writeback** causes the CPU to flush a dirty cache line into memory
  - **writeback/invalidate**

Cache Width

- The minimum number of bytes the CPU can read/write from memory into the cache is called the "cache width"
- Sometimes the width is different for read/write (or even for L1/L2 cache fills and evictions).
- However, the largest number is the one called the cache width.
- To PCI people, this is also called the cache line size.
- This quantity is very CPU (and sometimes bus) specific.

Cache Line Interference



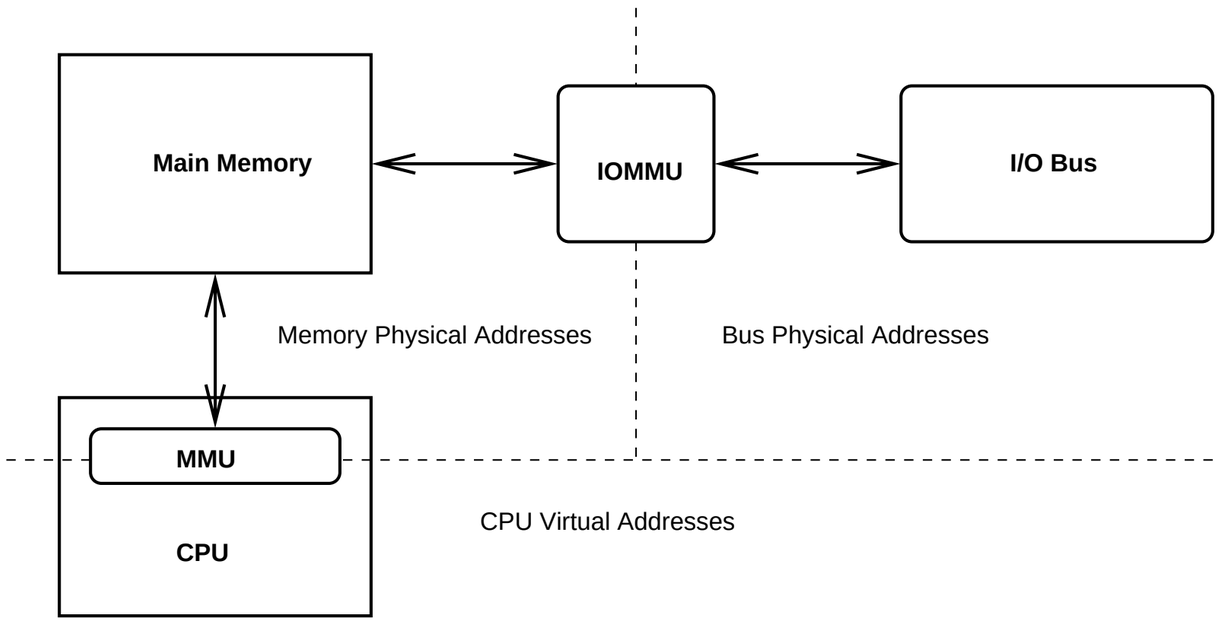
Dealing with Cache Line Interference

- Simplest way to avoid this problem is always to follow the rules:
  1. Never, *never*, **ever** do DMA onto the stack
  2. `kmalloc()` is aware of the cache line constraints and, by and large, never allocates memory which violates them.
  3. Don't mix DMA and non-DMA data in a structure
  4. If you have to break rule 3 keep the DMA and non-DMA areas separate (and make sure they're both cache line aligned).

Inducing Coherency

- On some Architectures, certain areas of memory can be made Coherent.
- Most common IOMMUs can be made to participate in the CPU coherency model, so requiring coherency may sometimes be programmed into it per mapping.
- Even if it cannot, you may be able to fake it by turning off the CPU's cache per page (slower, but works).
- Some CPU's have absolutely no way at all to induce coherency.
- Note: For PCI people
  - Coherent == Consistent
  - Non Coherent == Streaming

IOMMUs



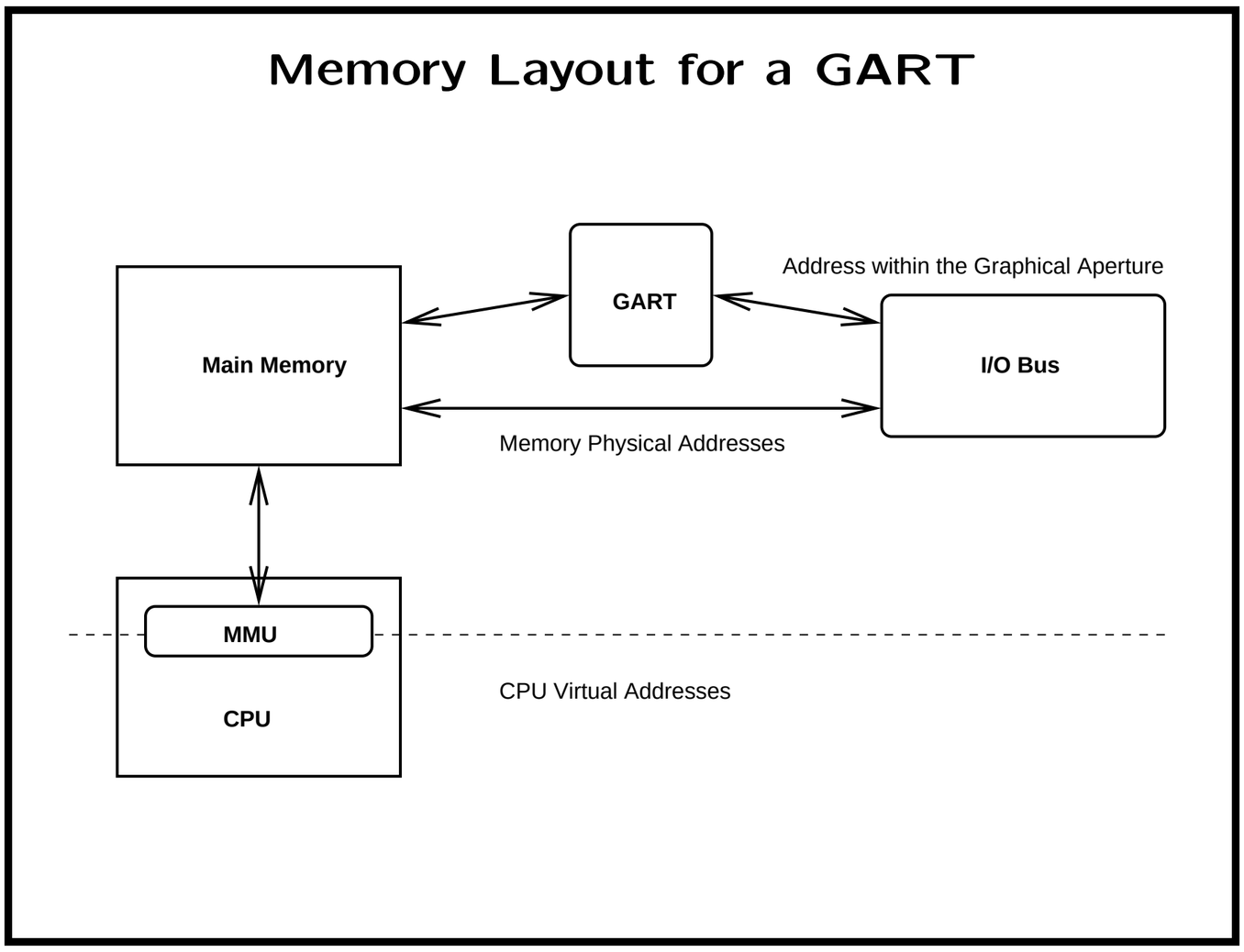
- The IOMMU is a memory mapping unit sitting between the I/O bus and physical memory.

IOMMUs Continued

- In order to begin a DMA transaction, the translations between the Memory and Bus physical addresses mus be programmed into the IOMMU.
- One useful feature of this is that any attempt to DMA to memory ranges outside the programmed translation can be caught (and even before it corrupts memory!)
- At the end of the DMA, the mappings must be removed from the IOMMU (otherwise they'll build up until you run out of mappings)

GARTs

- These are Graphical Aperture Remapping Tables
- Originally designed to give video cards on the AGP bus apparently large contiguous areas of physical memory.
- A GART acts like a simple IOMMU
  - The Aperture may be fixed by the bios or variable
  - It occupies usually a "window" in memory
  - Any bus use of a physical address in this window may be remapped into a different physical memory location.


Memory Layout for a GART
GART
Address within the Graphical Aperture
Main Memory
I/O Bus
Memory Physical Addresses
MMU
CPU
CPU Virtual Addresses

The DMA Mask

- Every device has a specific range of physical memory it can address
- This range may not correspond exactly to the amount of memory available in the system (rendering some memory non-addressable).
  - Old ISA devices can only address up to the first 16MB of memory.
  - Even some modern PCI devices may only be able to address up to the first 4GB of physical memory.
- This range is encoded in a mask, called the DMA mask

Using the DMA Mask

- The DMA mask has two distinct uses depending on whether an IOMMU is present in the system or not.
  - For non-IOMMU systems, it represents the physical maximum address that may be DMA'd to. Once over this address, the data must be copied to/from a lower address to perform DMA (this process is called bouncing).
  - If an IOMMU is present, the device may still DMA to all of physical memory, and the DMA mask is used by the IOMMU to determine what bus physical addresses it should use for the device.

Bouncing

- Bouncing refers to the task of moving data from device inaccessible memory to memory the device can DMA to.
- It is **only** necessary in non-IOMMU systems.
- For block devices, the bouncing is done per page as the bio is processed.
- The network layer has its own bouncing system too.
- And for good measure, IA-64 has an additional bouncing system coded to look like a fake memory management unit.

The PCI DMA API

- Introduced as an API to encapsulate the solution to all of the coherency/IOMMU problems
- Has:
  - Mapping and Unmapping API
  - Cache Synchronisation API
  - Coherent (Consistent) memory allocation API
- To emphasise the PCI nature, the API begins `pci_` and always takes a pointer to a `struct pci_device`.
- Sparc also has a completely equivalent `sbus_` API which takes a `struct sbus_device`.

The PCI DMA API Problems

- It only applies to PCI (and SBUS).
- We have many more bus types that need it (EISA, GSC, MCA, USB, ...)
- Even if we introduced more *<bus>*_ variants, some drivers don't know the underlying bus type (or must drive a large number of buses).
  - Can correct this by making fake PCI devices for these buses (callbacks are architecture specific)
  - if you only need to use the IOMMU, `NULL` works as the `struct pci_dev` pointer.
- Doesn't work at all for fully incoherent architectures.
- There are, actually, several problems with the API itself.

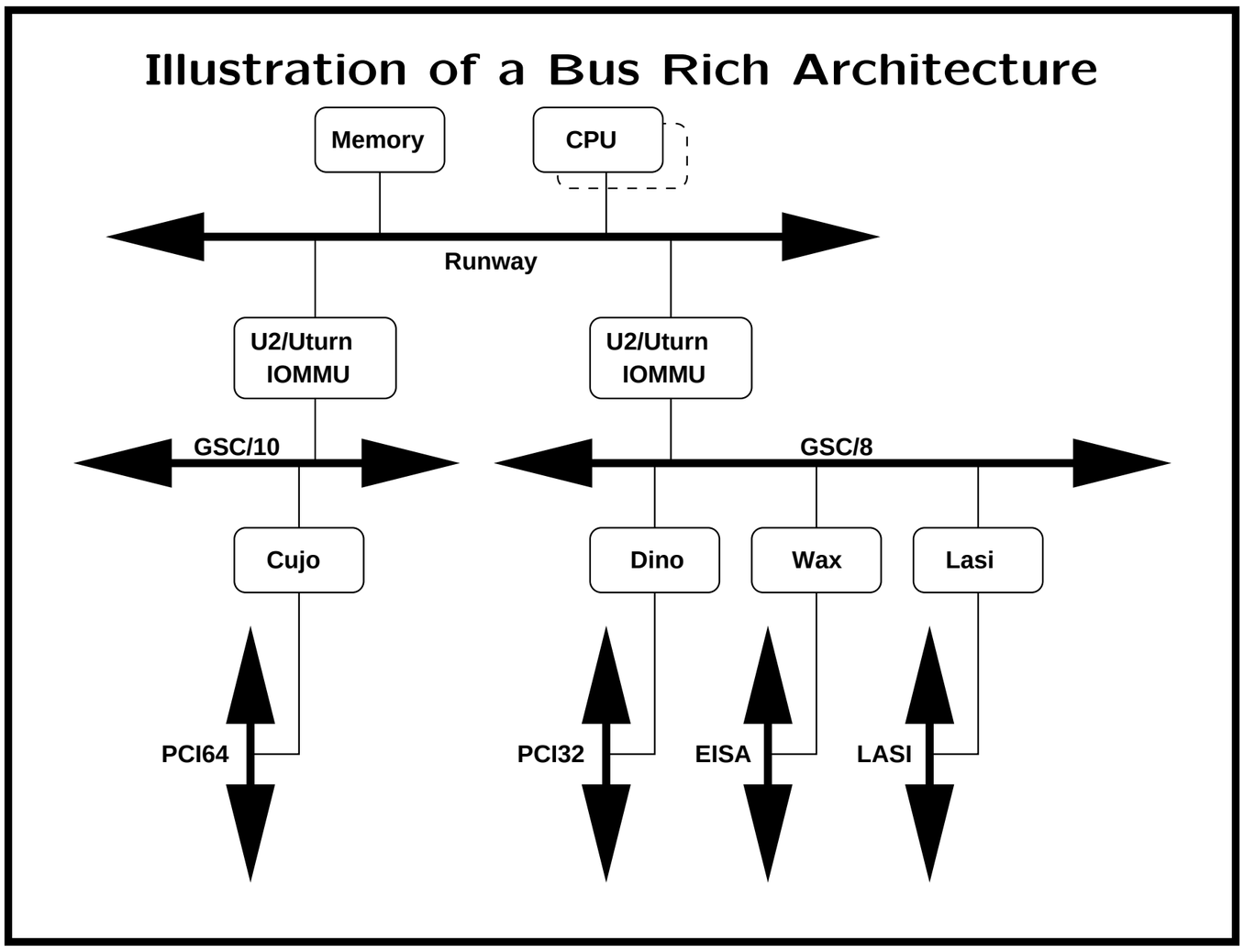


Illustration of a Bus Rich Architecture
Memory
CPU
Runway
U2/Uturn
IOMMU
U2/Uturn
IOMMU
GSC/10
GSC/8
Cujo
Dino
Wax
Lasi
PCI64
PCI32
EISA
LASI

Enter the Generic Device Model

- The generic device model requires that a `struct device` be embedded in every bus specific device type.
- The entire bus tree may be built from the parent/child relationships of the generic devices.
- This provides the ideal framework for building a generic API which simply takes a pointer to the generic device (and leaves any bus specific issues to the underlying architecture)
- Approach greatly simplifies the job of driver writers because the API is the same regardless of the bus.

Illustration—EISA

- EISA is an old bus type, but it is present in several architectures (x86, Alpha, parisc).
- A difficult one, because neither `NULL` PCI device nor a fake one provides the answer.
- However, once the EISA bus type was added (with minimal architecture specific support) it just simply worked.
- Well, OK, the drivers had to be converted as well.
- Good proof of concept.

Non-Coherent Architectures

- Problem is that `pci_alloc_consistent` can fail in two ways:
  1. Because you're out of coherent memory, so you should retry/abort the operation.
  2. Because you're on a non-coherent architecture (the call can never succeed).
- Ideally, you'd like to treat consistent memory allocation failure as always fatal and provide a special API to drivers that know they may be used on non-coherent platforms.
- Without having to do `if(coherent)` switches in the driver

Other Extensions in the New DMA API

- In addition to the incoherent memory API (which is dangerous), there are two other API additions which are classified as extremely dangerous
- And I mean extremely dangerous
- Cache width API (`dma_get_cache_alignment()`) allows returns the correct cache width dynamically. (The `#define` is usually the largest possible value for this)
- Partial sync API. This allows the partial synchronisation of the memory area (instead of fully synchronising it)
- Again, you should never be using either of these unless you really, really know what you're doing.

Current Problems with Both APIs

- There are really three glaring issues with both the generic device and the pci specific memory mapping APIs:
  1. The mapping APIs have no failure returns.
  2. There's no way to tell what an *appropriate* `dma_mask` for the system is
  3. The cache coherency API isn't as clearly usable and implementable as it should be.

Mapping Failures

- Originally, in the IOMMU model, usually up to 4GB of memory may be mapped at any one time, so you can never run out of mappings, right?
- Wrong: in super High End machines, we're already getting into situations where 4GB of data may be in flight at once
- GARTs may have really small apertures (around 128k)
- If you're stuck with a GART as your IOMMU you are almost always going to hit a failure even under normal load.
- Adding error returns is now **required**.

Appropriate DMA mask sizing

- Even on a 64 bit machine, you may have $<$ 4GB of memory.
- Most adapters that can DMA directly into all 64 bits have more than one "descriptor mode"
- e.g. aic79xx has three descriptor types, 64 bit, 39bit and 32bit.
- Often, the I/O can go slightly faster if you use smaller descriptors, so even on a 64bit machine, if it has less than 4GB memory you want to use 32 bit descriptors.
- Therefore, need to expand the return of `dma_set_mask` so that zero is error, but otherwise it returns the mask required by the platform (often just the mask required to reach available memory).

Cache Coherency API

- Current API has three hint flags: bidirectional, from device and to device.
- However, when to use these in the driver requires knowledge of how cache coherency works.
- An ownership model would have been much more appropriate
- Now, The data is either owned by the CPU or owned by the device (you still flag it with the same directions to give cache hints).

Old Cache API

- `dma_map`
- device writes to memory
- `dma_sync(DMA_FROM_DEVICE)`
- CPU snoops data and asks for more
- device writes to memory
- `dma_unmap`

Proposed New Cache API

- `dma_map` — device owns the memory
- device writes to memory
- `dma_sync_to_cpu(DMA_FROM_DEVICE)` — CPU owns the memory
- CPU snoops data and asks for more
- `dma_sync_to_device(DMA_FROM_DEVICE)` — device owns the memory again
- device writes to memory
- `dma_unmap` — CPU owns the memory again

Conclusions

- with the new API life is definitely much easier for some architectures and buses.
- The new API definitely does not solve all the problems
- However, it does solve some of the corner cases of the previous API.
- The API will still change to accommodate the three listed issues.