Software allocated memory or buffers also can be marked as cache coherent or not.

Arm memory regions can have different attributes affecting cache behavior:

* Device Memory: Non-cacheable, used for MMIO registers.
* Normal Memory (Cacheable): Subject to caching, may require explicit cache maintenance.
* Non-cacheable Normal Memory: Access bypasses the cache.

1.2 Arm Cache Maintenance Instructions

To ensure coherency, Arm provides instructions like:

* DC CIVAC (Data Cache Clean and Invalidate by VA to PoC)
* DC IVAC (Data Cache Invalidate by VA to PoC)
* DC CVAC (Data Cache Clean by VA to PoC)
* DC CVAU (Data Cache Clean by VA to PoU)
* DSB (Data Synchronization Barrier)

2. PCIe and Cache Coherency in Arm

PCIe devices operate in a separate memory space, meaning that reads/writes from the device to the CPU memory (or vice versa) can experience coherency issues unless properly managed.

2.1 Challenges in PCIe Cache Coherency

1. CPU Caching of PCIe MMIO Space
   * MMIO registers should always be accessed with non-cacheable memory mappings.
   * Arm uses Device memory attributes to ensure correct access to MMIO regions.
2. DMA Transactions and Cache Coherency
   * PCIe devices performing Direct Memory Access (DMA) can read stale data if CPU caches are not flushed.
   * Arm does not guarantee automatic coherency for external masters (e.g., PCIe devices).
   * Solutions:
     + Software-managed cache maintenance.
     + Using an IOMMU (SMMU in Arm) for address translation and coherency enforcement.

2.2 Ensuring Cache Coherency for PCIe DMA in Linux

2.2.1 DMA API in Linux

Linux offers dma\_map\_\*() and dma\_unmap\_\*() APIs to ensure that data transferred between the CPU and a PCIe device remains coherent.

* Mapping buffers for device access:

dma\_addr\_t dma\_handle;

void \*cpu\_addr = dma\_alloc\_coherent(dev, size, &dma\_handle, GFP\_KERNEL);

* + Ensures CPU and device see the same data.
  + Memory is allocated as non-cacheable.
* Using dma\_map\_single() for existing buffers:

dma\_addr\_t dma\_handle;

dma\_handle = dma\_map\_single(dev, cpu\_addr, size, DMA\_TO\_DEVICE);

* + Ensures cache flushing before the device accesses memory.
* Unmapping and invalidating caches:

dma\_unmap\_single(dev, dma\_handle, size, DMA\_FROM\_DEVICE);

2.2.2 Memory Barriers

Memory barriers (mb(), wmb(), rmb(), dsb()) ensure correct ordering of memory transactions.

* dsb(sy): Ensures completion of all memory operations before execution continues.
* mb(): Prevents reordering of reads/writes across the barrier.

3. IOMMU (SMMU) for PCIe Coherency

Arm systems use System MMU (SMMU) to provide:

* Address translation for PCIe devices.
* Enforcing memory attributes for cache coherency.

Configuring SMMU in Linux

* The Linux kernel driver for SMMU (arm-smmu.c) ensures that devices use the correct memory attributes.
* The SMMU is configured via device tree:

iommu {

compatible = "arm,smmu-v3";

};

* PCIe devices must be linked to SMMU in the DT:

pcie@50000000 {

iommu-map = <0 &smmu 0x0 0x1000>;

};

4. Ensuring Cache Coherency in User-Space

For applications using mmap() on PCIe device memory:

* Use mmap() with PROT\_WRITE | MAP\_SYNC for proper memory synchronization.
* Use sync\_for\_device() and sync\_for\_cpu() operations.

5. Future Enhancements

* Arm v8.4+ supports **PCIe ATS (Address Translation Services),** allowing devices to cache translation results.
* PCIe ECAM (Enhanced Configuration Access Mechanism) may improve memory access efficiency. **What is ECAM blocker and why used ?**

Conclusion

* For MMIO: Use non-cacheable memory.
* For DMA: Use Linux’s dma\_map\_\*() APIs.
* For optimal performance: Enable SMMU and use cache maintenance instructions effectively.

Would you like a more in-depth code example for any specific scenario?

You said:

What is prefetcheable and non-prefetcheable in PCIe

ChatGPT said:

Prefetchable vs Non-Prefetchable Memory in PCIe

PCIe devices interact with system memory using Memory-Mapped I/O (MMIO). In this context, memory regions are classified as prefetchable or non-prefetchable, which affects how the CPU and PCIe device access them.

1. Prefetchable Memory

A PCIe memory region is prefetchable if it can be safely read ahead by the processor or DMA controller without causing issues. This means that consecutive memory accesses can be optimized using burst reads.

Characteristics

* Can be cached: Since the memory region does not contain side-effect-causing registers, the CPU and PCIe device can cache or prefetch it.
* Supports speculative reads: The system can fetch multiple bytes in advance without waiting for a request.

Example Use Cases

* GPU frame buffers
* PCIe SSD memory (mapped storage)
* Network adapter buffers

Linux Representation

In Linux, PCIe prefetchable memory appears in /proc/iomem:

80000000-8fffffff : PCIe Prefetchable Memory

2. Non-Prefetchable Memory

A PCIe memory region is non-prefetchable if it contains device registers or memory with side effects, meaning speculative reads or caching could cause errors.

Characteristics

* Cannot be cached: Every access must go directly to the device.
* No speculative reads: The system must read only when explicitly requested.
* Used for control registers and MMIO: Where writing or reading affects device state.
* Ensures correctness but can be slower: Since every access is explicit and cannot be optimized.

Example Use Cases

* PCIe device control registers
* MMIO regions for hardware settings
* Status registers that change dynamically

Linux Representation

Non-prefetchable memory is also listed in /proc/iomem:

90000000-90ffffff : PCIe Non-Prefetchable Memory

3. PCIe BARs (Base Address Registers) and Prefetchability

When a PCIe device is initialized, it reports memory requirements using Base Address Registers (BARs). Each BAR specifies whether its region is prefetchable or non-prefetchable.

* Prefetchable BAR:
  + Marked as MemPref in lspci -vv
  + Typically used for bulk memory access
  + Example:

Region 0: Memory at 80000000 (64-bit, prefetchable)

* Non-Prefetchable BAR:
  + Marked as Mem in lspci -vv
  + Used for registers and control logic
  + Example:

Region 1: Memory at 90000000 (non-prefetchable)

4. How the CPU Handles Prefetchable vs Non-Prefetchable Memory

|  |  |  |
| --- | --- | --- |
| Feature | Prefetchable Memory | Non-Prefetchable Memory |
| Caching | Allowed | Not Allowed |
| Speculative Reads | Allowed | Not Allowed |
| Latency | Lower | Higher |
| Usage | Bulk data (GPU RAM, SSD) | Control registers (MMIO) |
| Access Method | Normal memory reads | Explicit MMIO reads |

5. Linux Kernel Handling

The Linux PCI subsystem correctly maps prefetchable and non-prefetchable regions:

* ioremap(): Used for non-prefetchable regions.
* ioremap\_wc(): Used for prefetchable memory with write-combining.

Example mapping for a device:

void \_\_iomem \*ctrl\_regs = ioremap(0x90000000, 0x1000); // Non-prefetchable MMIO

void \_\_iomem \*framebuffer = ioremap\_wc(0x80000000, 0x100000); // Prefetchable GPU memory

6. Conclusion

* Use prefetchable memory for high-throughput, cache-friendly operations like frame buffers.
* Use non-prefetchable memory for device control registers to ensure correctness.
* Linux and PCIe BARs distinguish between these two types, affecting how the system interacts with PCIe devices.

1. What Are PCIe BARs?

A Base Address Register (BAR) in a PCIe device specifies:

* The amount of memory the device needs.
* Whether the memory is prefetchable or non-prefetchable.
* Whether the memory region is 32-bit or 64-bit.

Each PCIe device can have up to 6 BARs (BAR0 – BAR5) for memory and I/O mapping.

2. Types of PCIe BARs

(a) Memory BARs (MMIO)

Memory BARs are used for accessing device memory and control registers.

* Prefetchable: Can be cached and accessed in bursts (e.g., GPU frame buffers).
* Non-Prefetchable: Cannot be cached; used for device registers.

(b) I/O BARs (Port I/O)

I/O BARs are used for legacy x86-style port-mapped I/O, but they are rarely used in modern PCIe devices.

3. Checking PCIe BARs in Linux

Use lspci -vvv to check BARs of a PCIe device:

lspci -vvv -s <bus:device.function>

Example Output:

01:00.0 VGA compatible controller: NVIDIA Corporation Device XYZ

Region 0: Memory at 80000000 (64-bit, prefetchable) [size=256M]

Region 1: Memory at 90000000 (non-prefetchable) [size=128K]

Region 2: I/O ports at d000 [size=128]

* Region 0 → Prefetchable GPU memory (64-bit).
* Region 1 → Non-prefetchable control registers.
* Region 2 → I/O space (legacy x86 I/O).

4. Configuring PCIe BARs in Linux Kernel

4.1. Reading PCIe BARs in Kernel

The kernel reads BARs using:

u32 bar;

pci\_read\_config\_dword(dev, PCI\_BASE\_ADDRESS\_0, &bar);

* PCI\_BASE\_ADDRESS\_0 refers to BAR0.
* The same can be done for PCI\_BASE\_ADDRESS\_1, PCI\_BASE\_ADDRESS\_2, etc.

4.2. Mapping PCIe BARs in Linux

Linux maps BARs into the CPU address space using:

void \_\_iomem \*bar\_addr;

bar\_addr = pci\_iomap(pdev, bar\_index, 0);

* bar\_index: Index of the BAR (0-5).
* pdev: PCI device structure.
* 0 means the full region size is mapped.

Example: Mapping a GPU’s frame buffer (prefetchable memory):

void \_\_iomem \*framebuffer;

framebuffer = pci\_iomap(pdev, 0, pci\_resource\_len(pdev, 0));

Example: Mapping non-prefetchable control registers:

void \_\_iomem \*ctrl\_regs;

ctrl\_regs = pci\_iomap(pdev, 1, pci\_resource\_len(pdev, 1));

4.3. Unmapping BARs

pci\_iounmap(pdev, bar\_addr);

* Always unmap when no longer needed.

5. Handling BAR Size and Alignment

Each BAR requests a specific size of memory, and its address must be aligned. The size is determined by writing 0xFFFFFFFF to the BAR and reading back the allocated size.

Example:

pci\_write\_config\_dword(dev, PCI\_BASE\_ADDRESS\_0, 0xFFFFFFFF);

pci\_read\_config\_dword(dev, PCI\_BASE\_ADDRESS\_0, &size);

size = ~(size & PCI\_BASE\_ADDRESS\_MEM\_MASK) + 1;

* PCI\_BASE\_ADDRESS\_MEM\_MASK ensures that only the size bits are extracted.

6. Reassigning PCIe BARs in Linux

To manually change BAR allocations:

1. Disable the device first:

echo 1 > /sys/bus/pci/devices/0000:01:00.0/remove

1. Modify BAR settings in /sys/bus/pci/devices/0000:01:00.0/resource.
2. Re-enable the device:

echo 1 > /sys/bus/pci/rescan

7. IOMMU and BARs

If the device uses an IOMMU (e.g., SMMU in Arm), memory accesses from PCIe devices go through address translation, ensuring:

* Security (preventing unauthorized memory access).
* Coherency with CPU caches.

Enabling IOMMU in Linux

Ensure the kernel is built with:

CONFIG\_IOMMU=y

CONFIG\_ARM\_SMMU=y

8. Summary

|  |  |  |
| --- | --- | --- |
| Feature | Prefetchable BAR | Non-Prefetchable BAR |
| Caching | Allowed | Not Allowed |
| Speculative Reads | Allowed | Not Allowed |
| Usage | GPU RAM, SSD memory | Control registers |
| Mapping in Linux | ioremap\_wc() | ioremap() |

1. Understanding Cache Coherency in PCIe DMA

What is Cache Coherency?

Cache coherency ensures that multiple processors (CPU and PCIe devices) see the most recent version of memory. In the context of PCIe:

* The CPU caches memory for fast access.
* The PCIe device (via DMA) reads/writes directly to system memory.
* If the CPU caches outdated data, it may read incorrect values.

Key Problem Areas

* CPU writes data to a buffer, but it is still in the cache → PCIe device reads stale data.
* PCIe device updates a buffer, but the CPU still sees cached old data.
* Multiple CPU cores accessing the same data can create inconsistencies.

2. PCIe, DMA, and Memory Regions

PCIe devices access memory using DMA, bypassing the CPU cache. However, different memory regions have different behaviors:

|  |  |  |
| --- | --- | --- |
| Memory Type | Cacheable | PCIe Device Access |
| Prefetchable Memory (e.g., GPU VRAM) | Yes | Can be cached, but needs sync |
| Non-Prefetchable Memory (MMIO, Registers) | No | Always accessed directly |
| DMA Buffers (Normal Memory) | Sometimes | Needs explicit cache maintenance |

* Prefetchable Memory: Used for bulk transfers (e.g., GPU buffers).
* Non-Prefetchable Memory: Used for control registers (PCIe MMIO).
* DMA Buffers: Must be managed properly to maintain coherency.

3. Cache Coherency Mechanisms in Arm & Linux

Unlike x86 (which has hardware cache coherency), Arm architecture does not guarantee cache coherency for PCIe devices unless explicitly managed.

(a) Coherent vs. Non-Coherent DMA

|  |  |  |
| --- | --- | --- |
| Type | Description | Usage |
| Coherent DMA | CPU and device always see the same memory state | Requires an IOMMU or hardware cache snooping |
| Non-Coherent DMA | CPU cache and memory can become inconsistent | Requires explicit cache maintenance |

Most embedded Arm platforms do not have hardware cache coherency, meaning software-controlled cache maintenance is required.

4. Handling Cache Coherency in Linux (DMA API)

Linux provides DMA APIs to ensure proper cache synchronization for PCIe DMA transfers.

(a) Allocating DMA-Coherent Memory

If the hardware supports cache-coherent DMA, use:

void \*buf = dma\_alloc\_coherent(dev, size, &dma\_handle, GFP\_KERNEL);

* Ensures CPU and PCIe device always see consistent data.
* No need for manual cache maintenance.

(b) Allocating Non-Coherent DMA Buffers

If the platform is non-coherent, use:

void \*buf = dma\_alloc\_noncoherent(dev, size, &dma\_handle, GFP\_KERNEL);

* Requires explicit cache invalidation and flushing.

5. Cache Maintenance for Non-Coherent DMA

If dma\_alloc\_coherent() is not used, manual cache management is needed.

(a) Before PCIe Device Reads (CPU → Device)

The CPU must flush its cache so the device reads the latest data:

dma\_sync\_single\_for\_device(dev, dma\_handle, size, DMA\_TO\_DEVICE);

(b) After PCIe Device Writes (Device → CPU)

The CPU must invalidate its cache before reading data:

dma\_sync\_single\_for\_cpu(dev, dma\_handle, size, DMA\_FROM\_DEVICE);

(c) Bidirectional DMA (Both Directions)

dma\_sync\_single\_for\_device(dev, dma\_handle, size, DMA\_BIDIRECTIONAL);

* Used when both CPU and device modify the buffer.

6. IOMMU and Cache Coherency

If an IOMMU (like SMMU in Arm) is enabled, it can:

* Ensure cache coherency via snooping.
* Translate virtual addresses for PCIe devices.

To check if the platform supports IOMMU:

dmesg | grep -i smmu

To enable IOMMU in Linux:

echo Y > /sys/module/iommu/parameters/enable

7. Real-World Example (PCIe Network Driver)

A typical PCIe Ethernet driver must handle cache coherency properly:

struct sk\_buff \*skb;

dma\_addr\_t dma\_addr;

/\* Allocate buffer \*/

skb = dev\_alloc\_skb(size);

dma\_addr = dma\_map\_single(dev, skb->data, size, DMA\_FROM\_DEVICE);

/\* PCIe device writes data to DMA buffer \*/

pci\_dma\_write(device, dma\_addr, size);

/\* Ensure CPU reads updated data \*/

dma\_sync\_single\_for\_cpu(dev, dma\_addr, size, DMA\_FROM\_DEVICE);

/\* Process received packet \*/

process\_packet(skb->data);

/\* Unmap the buffer \*/

dma\_unmap\_single(dev, dma\_addr, size, DMA\_FROM\_DEVICE);

8. Summary

|  |  |
| --- | --- |
| Concept | Details |
| PCIe Cache Coherency Issue | CPU cache and PCIe DMA can have stale data. |
| Coherent DMA | No need for manual cache maintenance. |
| Non-Coherent DMA | Requires explicit cache flushing and invalidation. |
| Linux DMA API | dma\_alloc\_coherent(), dma\_sync\_single\_for\_cpu(), etc. |
| IOMMU | Helps maintain coherency on supported platforms. |

Both pci\_iomap() and ioremap() are used to map physical memory regions (e.g., PCIe device memory) into the CPU's virtual address space, but they have key differences in functionality and use cases.

1. pci\_iomap(): pci\_iomap() is a PCI-specific API in Linux that maps a PCIe device's BAR (Base Address Register) into CPU virtual memory.

void \_\_iomem \*bar\_addr;

bar\_addr = pci\_iomap(pdev, bar\_index, 0);

Key Features

✅ Simplifies PCIe BAR mapping

* pci\_iomap() directly takes a PCIe struct pci\_dev and a BAR index, avoiding manual address lookup.
* It automatically retrieves BAR addresses and sizes from pci\_resource\_start() and pci\_resource\_len().

✅ Handles different BAR types automatically

* Works for both prefetchable and non-prefetchable memory.
* Determines if a BAR is MMIO or I/O ports and maps accordingly.

✅ Portable and PCI-aware

* It ensures proper memory type handling for PCIe devices.
* Abstracts low-level details, making driver code more portable.

2. ioremap(): ioremap() is a generic API that maps a physical memory address (not limited to PCIe) into virtual memory for CPU access.

void \_\_iomem \*mapped\_addr;

mapped\_addr = ioremap(phys\_addr, size);

Key Features

✅ Works for any physical memory region

* Unlike pci\_iomap(), it is not limited to PCIe BARs.
* Used for platform devices, memory-mapped registers, and firmware regions.

✅ More control over address mapping

* Requires explicit phys\_addr and size, making it flexible.

✅ Supports different caching modes

* ioremap() → Maps with default non-cacheable attributes.
* ioremap\_wc() → Maps write-combining (for framebuffers).
* ioremap\_cache() → Maps with cacheable attributes.

3. Key Differences

|  |  |  |
| --- | --- | --- |
| Feature | pci\_iomap() | ioremap() |
| Scope | PCIe BARs only | Any physical memory |
| Ease of Use | Simple (just needs pdev and BAR index) | Requires manual address lookup |
| BAR Handling | Automatically detects type and size | Requires manual input |
| Caching Control | Automatically sets appropriate attributes | Allows different caching options |
| Use Case | PCIe drivers | Platform devices, MMIO registers |

4. When to Use Which?

|  |  |
| --- | --- |
| Scenario | Use |
| Mapping a PCIe BAR | pci\_iomap() |
| Mapping a generic MMIO register | ioremap() |
| Mapping firmware regions (e.g., ACPI tables) | ioremap() |
| Mapping a GPU framebuffer | ioremap\_wc() |

5. Example Usage

Using pci\_iomap() (Recommended for PCIe)

#include <linux/pci.h>

#include <linux/io.h>

struct pci\_dev \*pdev;

void \_\_iomem \*bar0;

bar0 = pci\_iomap(pdev, 0, 0);

if (!bar0) {

pr\_err("Failed to map PCIe BAR0\n");

return -ENOMEM;

}

/\* Access the mapped BAR \*/

iowrite32(0x12345678, bar0);

/\* Unmap when done \*/

pci\_iounmap(pdev, bar0);

Using ioremap() (For Generic MMIO)

#include <linux/io.h>

#define MMIO\_PHYS\_ADDR 0xFEDC0000

#define MMIO\_SIZE 0x1000

void \_\_iomem \*mmio\_base;

mmio\_base = ioremap(MMIO\_PHYS\_ADDR, MMIO\_SIZE);

if (!mmio\_base) {

pr\_err("Failed to remap MMIO address\n");

return -ENOMEM;

}

/\* Access the MMIO register \*/

iowrite32(0x87654321, mmio\_base);

/\* Unmap when done \*/

iounmap(mmio\_base);

6. Summary

* Use pci\_iomap() for PCIe BARs (simplifies mapping).
* Use ioremap() for general MMIO regions (requires manual mapping).
* If dealing with cached memory, use ioremap\_cache() or ioremap\_wc().

Would you like an example of ioremap\_wc() for framebuffers? When mapping framebuffers (FBs) or GPU memory for high-speed writes (e.g., rendering), normal ioremap() is inefficient because:

* It does not allow write combining, meaning each write operation is slow.
* It is not optimized for bulk writes, leading to performance bottlenecks.

Instead, ioremap\_wc() (write-combining) should be used to optimize memory writes for framebuffers.

1. Why Use ioremap\_wc() for Framebuffers?

✅ Faster writes → Groups multiple writes before sending to memory.  
✅ Efficient rendering → Reduces CPU overhead for drawing operations.  
✅ Optimized for GPUs and Display Controllers → Common for PCIe GPUs, embedded LCD controllers, and video buffers.

2. Example: Mapping a GPU Framebuffer Using ioremap\_wc()

Scenario

* Suppose the GPU framebuffer is mapped at physical address 0xA0000000.
* We want to map 16 MB of framebuffer memory and write pixels efficiently.

Code Implementation

#include <linux/io.h>

#include <linux/module.h>

#define FB\_PHYS\_ADDR 0xA0000000 // GPU framebuffer base address

#define FB\_SIZE 0x01000000 // 16MB framebuffer size

void \_\_iomem \*fb\_base; // Virtual address of framebuffer

static int \_\_init fb\_map\_init(void)

{

// Map the framebuffer with write-combining attributes

fb\_base = ioremap\_wc(FB\_PHYS\_ADDR, FB\_SIZE);

if (!fb\_base) {

pr\_err("Failed to map framebuffer memory\n");

return -ENOMEM;

}

pr\_info("Framebuffer mapped at virtual address %p\n", fb\_base);

// Example: Set the first pixel to white (assuming 32-bit color)

iowrite32(0xFFFFFFFF, fb\_base);

return 0;

}

static void \_\_exit fb\_map\_exit(void)

{

if (fb\_base) {

iounmap(fb\_base); // Unmap when done

pr\_info("Framebuffer unmapped\n");

}

}

module\_init(fb\_map\_init);

module\_exit(fb\_map\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("Framebuffer Mapping using ioremap\_wc()");

3. Explanation of Code

Initialization (fb\_map\_init)

1. Maps the framebuffer (ioremap\_wc(FB\_PHYS\_ADDR, FB\_SIZE)).
2. Checks if the mapping was successful (returns an error if not).
3. Writes a pixel (0xFFFFFFFF → white) to the first pixel.

Cleanup (fb\_map\_exit)

* Calls iounmap(fb\_base) to release the mapped region when the module is removed.

4. Performance Benefit of ioremap\_wc()

Without ioremap\_wc() (Using ioremap())

* Every iowrite32() triggers immediate write to memory.
* Slower performance due to frequent memory access.

With ioremap\_wc()

* Write operations are buffered and combined before committing.
* Faster rendering, ideal for high-throughput operations like video playback or 2D graphics.

5. Use Cases

|  |  |
| --- | --- |
| Use Case | Recommended API |
| Framebuffer (GPU, LCD, Display Controllers) | ioremap\_wc() |
| Video buffers (camera frame processing) | ioremap\_wc() |
| PCIe device memory (e.g., mapped VRAM) | ioremap\_wc() |
| Regular MMIO registers (Control Registers, PCIe BARs) | ioremap() |

6. Summary

|  |  |  |
| --- | --- | --- |
| API | Use Case | Performance |
| ioremap() | General MMIO (Control registers, PCIe BARs) | Normal (Non-cacheable) |
| ioremap\_wc() | Framebuffers, GPU memory, video buffers | Fast (Write-combining) |
| ioremap\_cache() | Cached memory regions | Very fast (CPU caching enabled) |

Using ioremap\_cache() for Cached Memory Mappings

1. Why Use ioremap\_cache()?

ioremap\_cache() is used when mapping memory regions that benefit from CPU caching. It is ideal for scenarios where frequent read operations are required, as cached memory reduces access latency.

* ✅ High-speed memory access
* ✅ CPU caching enabled (faster reads & writes)
* ✅ Used for RAM-like memory regions (e.g., shared memory, DDR buffers, firmware regions)

2. When to Use ioremap\_cache()?

|  |  |
| --- | --- |
| Use Case | Recommended API |
| Shared memory between CPU & GPU/DSP | ioremap\_cache() |
| Firmware regions (e.g., ACPI tables, device trees) | ioremap\_cache() |
| On-chip SRAM or DDR buffers | ioremap\_cache() |
| Read-heavy MMIO memory (e.g., logs, data buffers) | ioremap\_cache() |

3. Example: Mapping DDR Buffer Using ioremap\_cache()

Scenario

* Assume a shared DDR memory buffer is located at physical address 0xB0000000.
* We need to map 4 MB of this buffer into the CPU’s address space for high-speed reads.

Code Implementation

#include <linux/io.h>

#include <linux/module.h>

#define SHARED\_MEM\_ADDR 0xB0000000 // Physical address of shared DDR buffer

#define SHARED\_MEM\_SIZE 0x00400000 // 4MB buffer size

void \_\_iomem \*shared\_mem\_base; // Virtual address after mapping

static int \_\_init shared\_mem\_init(void)

{

// Map the memory with CPU caching enabled

shared\_mem\_base = ioremap\_cache(SHARED\_MEM\_ADDR, SHARED\_MEM\_SIZE);

if (!shared\_mem\_base) {

pr\_err("Failed to map shared memory\n");

return -ENOMEM;

}

pr\_info("Shared memory mapped at virtual address %p\n", shared\_mem\_base);

// Example: Read the first 32-bit value from the buffer

uint32\_t data = ioread32(shared\_mem\_base);

pr\_info("Read value: 0x%08X\n", data);

return 0;

}

static void \_\_exit shared\_mem\_exit(void)

{

if (shared\_mem\_base) {

iounmap(shared\_mem\_base); // Unmap the memory

pr\_info("Shared memory unmapped\n");

}

}

module\_init(shared\_mem\_init);

module\_exit(shared\_mem\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("Shared DDR Memory Mapping using ioremap\_cache()");

4. Explanation of Code

Initialization (shared\_mem\_init)

1. Calls ioremap\_cache(SHARED\_MEM\_ADDR, SHARED\_MEM\_SIZE) to map the DDR buffer with CPU caching enabled.
2. If mapping fails, it returns an error.
3. Reads the first 32-bit value from the shared memory buffer.

Cleanup (shared\_mem\_exit)

* Calls iounmap(shared\_mem\_base) to release the mapped memory region.

5. Performance Benefit of ioremap\_cache()

Without ioremap\_cache() (Using ioremap())

* Each ioread32() is a slow memory-mapped I/O operation.
* The CPU fetches data directly from RAM every time, leading to latency.

With ioremap\_cache()

* The CPU caches the memory region for faster access.
* Frequent reads are much faster due to caching.

6. Summary: Choosing Between ioremap(), ioremap\_wc(), and ioremap\_cache()

|  |  |  |
| --- | --- | --- |
| API | Use Case | Performance |
| ioremap() | General MMIO (control registers, PCIe BARs) | Normal (Non-cacheable) |
| ioremap\_wc() | Framebuffers, GPU memory, video buffers | Fast (Write-combining) |
| ioremap\_cache() | Shared DDR buffers, firmware regions, read-heavy memory | Very Fast (CPU Caching) |

The Internal Address Translation Unit (IATU) in PCIe is used to map CPU (SoC) address space to PCIe address space and vice versa. It is commonly found in PCIe Root Complex (RC) implementations on Arm-based SoCs.

1. Why is IATU Needed?

✅ Translates between CPU and PCIe address spaces  
✅ Configures inbound & outbound transactions  
✅ Used in PCIe Root Complex (RC) and Endpoint (EP) mode

PCIe does not have a flat address space, so IATU helps map system memory (RAM, MMIO regions) to PCIe’s address space.

2. IATU Transaction Types

|  |  |  |
| --- | --- | --- |
| Direction | Purpose | Example |
| Outbound (OB) | CPU to PCIe | Mapping CPU memory to PCIe device |
| Inbound (IB) | PCIe to CPU | Mapping PCIe device memory to CPU |

3. IATU Outbound Mapping (CPU → PCIe Device)

* When CPU writes to a mapped PCIe device, IATU translates CPU memory regions to PCIe address space.
* Example: Mapping DDR memory (0x80000000 – 0x81000000) to a PCIe device BAR0 (0x50000000).

Outbound IATU Configuration (CPU → PCIe)

|  |  |
| --- | --- |
| Register | Value |
| IATU\_OUT\_LWR\_BASE | 0x80000000 (CPU address) |
| IATU\_OUT\_UPPER\_BASE | 0x00000000 |
| IATU\_OUT\_LIMIT | 0x80FFFFFF |
| IATU\_OUT\_LWR\_TARGET | 0x50000000 (PCIe BAR address) |
| IATU\_OUT\_UPPER\_TARGET | 0x00000000 |

Example: Mapping CPU Memory to PCIe BAR

// Configure IATU Outbound Region 0

writel(0x80000000, IATU\_OUT\_LWR\_BASE);

writel(0x00000000, IATU\_OUT\_UPPER\_BASE);

writel(0x80FFFFFF, IATU\_OUT\_LIMIT);

writel(0x50000000, IATU\_OUT\_LWR\_TARGET);

writel(0x00000000, IATU\_OUT\_UPPER\_TARGET);

After this, any write to CPU address 0x80000000 is translated to PCIe device address 0x50000000.

4. IATU Inbound Mapping (PCIe Device → CPU)

* When a PCIe device performs DMA to system memory, IATU translates PCIe addresses into CPU memory regions.
* Example: Mapping PCIe BAR0 (0x50000000 – 0x500FFFFF) to DDR memory (0x80000000 – 0x800FFFFF).

Inbound IATU Configuration (PCIe → CPU)

|  |  |
| --- | --- |
| Register | Value |
| IATU\_IN\_LWR\_BASE | 0x50000000 (PCIe BAR0) |
| IATU\_IN\_UPPER\_BASE | 0x00000000 |
| IATU\_IN\_LIMIT | 0x500FFFFF |
| IATU\_IN\_LWR\_TARGET | 0x80000000 (CPU RAM) |
| IATU\_IN\_UPPER\_TARGET | 0x00000000 |

Example: Mapping PCIe Device to System RAM

// Configure IATU Inbound Region 0

writel(0x50000000, IATU\_IN\_LWR\_BASE);

writel(0x00000000, IATU\_IN\_UPPER\_BASE);

writel(0x500FFFFF, IATU\_IN\_LIMIT);

writel(0x80000000, IATU\_IN\_LWR\_TARGET);

writel(0x00000000, IATU\_IN\_UPPER\_TARGET);

Now, when a PCIe device writes to 0x50000000, the transaction is routed to CPU RAM at 0x80000000.

The Internal Address Translation Unit (IATU) in PCIe supports different modes for address translation. The two primary modes used in configuring IATU are:

1. BAR Match Mode
2. Address Match Mode

1. BAR Match Mode

Definition

* This mode is used when mapping an inbound transaction from a PCIe endpoint to a specific Base Address Register (BAR) of the Root Complex (RC).
* The IATU checks which BAR the incoming address matches and forwards the transaction accordingly.

How It Works

* PCIe devices expose their memory regions via BARs.
* The IATU is configured to match BAR0–BAR5 and direct accesses to corresponding CPU memory regions.
* This is mostly used for inbound translations, where the PCIe device initiates transactions towards the host (Root Complex).

Example Use Case

* A PCIe device performs DMA writes to system memory via a BAR-mapped region.
* The IATU is configured to detect BAR0 and map its accesses to a specific memory region.

IATU Register Configuration for BAR Match Mode

|  |  |
| --- | --- |
| Register | Value |
| IATU\_REGION\_CTRL\_1 | Set to Inbound + BAR Match Mode |
| IATU\_REGION\_CTRL\_2 | Enable translation |
| IATU\_LIMIT | Specifies the BAR size |
| IATU\_LWR\_TARGET | System memory base address |
| IATU\_UPPER\_TARGET | Upper 32 bits of the memory base address |

// Configure IATU for BAR Match Mode (Inbound)

iowrite32(0x2, iatu\_base + IATU\_REGION\_CTRL\_1); // Set BAR match mode

iowrite32(BAR\_NUMBER, iatu\_base + IATU\_REGION\_CTRL\_2); // Match BAR0, BAR1, etc.

iowrite32(0x90000000, iatu\_base + IATU\_LWR\_TARGET); // Target System RAM

2. Address Match Mode

Definition

* This mode is used to map a specific address range rather than matching a BAR.
* It applies to both inbound and outbound transactions.

How It Works

* Instead of matching a BAR, the IATU checks whether an address falls within a predefined range and maps it accordingly.
* Used for both CPU → PCIe (Outbound) and PCIe → CPU (Inbound) transactions.

Example Use Case

* A CPU wants to access a PCIe device at address 0x50000000.
* The IATU maps 0x80000000 in CPU memory space to 0x50000000 in PCIe address space.
* Any access to 0x80000000 is translated to 0x50000000.

IATU Register Configuration for Address Match Mode

|  |  |
| --- | --- |
| Register | Value |
| IATU\_REGION\_CTRL\_1 | Set to Address Match Mode |
| IATU\_LWR\_BASE | Lower base address of the CPU memory region |
| IATU\_UPPER\_BASE | Upper 32 bits of the CPU memory region |
| IATU\_LIMIT | Upper limit of the mapped address range |
| IATU\_LWR\_TARGET | PCIe memory base address |
| IATU\_UPPER\_TARGET | Upper 32 bits of PCIe memory base address |

// Configure IATU for Address Match Mode (Outbound)

iowrite32(0x0, iatu\_base + IATU\_REGION\_CTRL\_1); // Address match mode

iowrite32(0x80000000, iatu\_base + IATU\_LWR\_BASE); // CPU Memory Start

iowrite32(0x80FFFFFF, iatu\_base + IATU\_LIMIT); // Limit Address

iowrite32(0x50000000, iatu\_base + IATU\_LWR\_TARGET); // PCIe Target Address

3. BAR Match Mode vs. Address Match Mode

|  |  |  |
| --- | --- | --- |
| Feature | BAR Match Mode | Address Match Mode |
| Use Case | Inbound transactions (PCIe device to CPU) | Both Inbound & Outbound transactions |
| Matching Criteria | Matches a PCIe BAR number | Matches a specific address range |
| Best for | Memory-mapped BAR regions | Direct memory-to-memory translations |
| Example | Mapping BAR0 of a PCIe NIC to system RAM | Mapping CPU address 0x80000000 to PCIe device 0x50000000 |

4. Summary

* BAR Match Mode → Matches PCIe BARs and maps them to system memory.
* Address Match Mode → Matches specific address ranges and redirects them.
* Outbound Mode → CPU accessing PCIe device memory.
* Inbound Mode → PCIe device accessing CPU memory (e.g., DMA).

PCIe IATU Driver Example (BAR Match + Address Match)

This example assumes:

* BAR0 of a PCIe device is mapped to system memory using BAR Match Mode.
* A specific CPU address range (0x80000000 - 0x80FFFFFF) is mapped to a PCIe device memory (0x50000000 - 0x50FFFFFF) using Address Match Mode.

Driver Code

#include <linux/module.h>

#include <linux/pci.h>

#include <linux/io.h>

#define IATU\_BASE 0x1000 // Assume IATU base register offset

#define IATU\_REGION\_CTRL\_1 0x00

#define IATU\_REGION\_CTRL\_2 0x04

#define IATU\_LWR\_BASE 0x08

#define IATU\_UPPER\_BASE 0x0C

#define IATU\_LIMIT 0x10

#define IATU\_LWR\_TARGET 0x14

#define IATU\_UPPER\_TARGET 0x18

#define CPU\_ADDR\_START 0x80000000 // CPU Address range to map

#define CPU\_ADDR\_END 0x80FFFFFF // Limit

#define PCIE\_TARGET\_ADDR 0x50000000 // PCIe Device Target Address

static void \_\_iomem \*iatu\_base;

// Function to configure IATU in BAR Match Mode (Inbound)

static void configure\_iatu\_bar\_match\_mode(void)

{

printk(KERN\_INFO "Configuring IATU: BAR Match Mode\n");

// Select Region 0 for BAR Match Mode

iowrite32(0x2, iatu\_base + IATU\_REGION\_CTRL\_1); // Inbound + BAR match mode

iowrite32(0x0, iatu\_base + IATU\_REGION\_CTRL\_2); // Match BAR0

iowrite32(CPU\_ADDR\_START, iatu\_base + IATU\_LWR\_TARGET); // Target System Memory

}

// Function to configure IATU in Address Match Mode (Outbound)

static void configure\_iatu\_address\_match\_mode(void)

{

printk(KERN\_INFO "Configuring IATU: Address Match Mode\n");

// Select Region 1 for Address Match Mode

iowrite32(0x0, iatu\_base + IATU\_REGION\_CTRL\_1); // Outbound + Address match mode

iowrite32(CPU\_ADDR\_START, iatu\_base + IATU\_LWR\_BASE); // Lower CPU Address

iowrite32(CPU\_ADDR\_END, iatu\_base + IATU\_LIMIT); // Address Range Limit

iowrite32(PCIE\_TARGET\_ADDR, iatu\_base + IATU\_LWR\_TARGET); // PCIe Target Address

}

// PCIe Probe Function

static int pcie\_iatu\_probe(struct pci\_dev \*pdev, const struct pci\_device\_id \*id)

{

int ret;

printk(KERN\_INFO "PCIe IATU Driver Probe\n");

ret = pci\_enable\_device(pdev);

if (ret) {

printk(KERN\_ERR "Failed to enable PCIe device\n");

return ret;

}

// Map IATU registers

iatu\_base = pci\_iomap(pdev, 0, 0);

if (!iatu\_base) {

printk(KERN\_ERR "Failed to map IATU registers\n");

return -ENOMEM;

}

// Configure both modes

configure\_iatu\_bar\_match\_mode();

configure\_iatu\_address\_match\_mode();

return 0;

}

// PCIe Remove Function

static void pcie\_iatu\_remove(struct pci\_dev \*pdev)

{

printk(KERN\_INFO "Removing PCIe IATU Driver\n");

if (iatu\_base)

pci\_iounmap(pdev, iatu\_base);

pci\_disable\_device(pdev);

}

// PCI Device Table

static struct pci\_device\_id pcie\_iatu\_ids[] = {

{ PCI\_DEVICE(0x1234, 0x5678) }, // Replace with real Vendor & Device ID

{ 0, }

};

MODULE\_DEVICE\_TABLE(pci, pcie\_iatu\_ids);

// PCI Driver Struct

static struct pci\_driver pcie\_iatu\_driver = {

.name = "pcie\_iatu\_driver",

.id\_table = pcie\_iatu\_ids,

.probe = pcie\_iatu\_probe,

.remove = pcie\_iatu\_remove,

};

// Module Init & Exit

static int \_\_init pcie\_iatu\_init(void)

{

return pci\_register\_driver(&pcie\_iatu\_driver);

}

static void \_\_exit pcie\_iatu\_exit(void)

{

pci\_unregister\_driver(&pcie\_iatu\_driver);

}

module\_init(pcie\_iatu\_init);

module\_exit(pcie\_iatu\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("PCIe IATU Configuration Driver");

Key Takeaways

* BAR Match Mode (Inbound)
  + Used when a PCIe device wants to access the host’s system memory.
  + Matches a specific BAR (e.g., BAR0) and directs traffic.
* Address Match Mode (Outbound)
* Used when the CPU wants to access a specific PCIe device address range.
* Translates a defined CPU memory range (0x80000000 - 0x80FFFFFF) to a PCIe device memory (0x50000000 - 0x50FFFFFF).