

# Application Note 228

## Implementing DMA on ARM SMP Systems

Document number: ARM DAI 0228 A

Issued: 1<sup>st</sup> August 2009

Copyright ARM Limited 2009



# Application Note 228

## Implementing DMA on SMP Systems

Copyright © 2009 ARM Limited. All rights reserved.

### Release information

#### Change history

Date	Issue	Change
August 2009	A	First release

### Proprietary notice

Words and logos marked with © and ™ are registered trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

### Confidentiality status

This document is Open Access. This document has no restriction on distribution.

### Feedback on this Application Note

If you have any comments on this Application Note, please send email to [errata@arm.com](mailto:errata@arm.com) giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

### ARM web address

<http://www.arm.com>

# Table of Contents

<b>1. Scope .....</b>	<b>1-1</b>
<b>2. Cache coherency in multi-core processing systems.....</b>	<b>2-1</b>
2.1 Coherency Protocols in ARM MPCore processors .....	2-2
2.2 Coherency and non-CPU bus masters.....	2-6
<b>3. Introduction to Direct Memory Access (DMA) .....</b>	<b>3-7</b>
3.1 DMA Basics .....	3-7
3.2 Using DMA in cached systems.....	3-8
<b>4. Considerations on using DMA.....</b>	<b>4-8</b>
4.1 ARM11 MPCore .....	4-8
4.2 Cortex-A9 MPCore .....	4-8
4.3 The Accelerator Coherency Port (ACP) and I/O Coherency.....	4-8
<b>5. Conclusion.....</b>	<b>5-8</b>



# 1. Scope

This Application Note explores the implications associated with performing Direct Memory Access (DMA) operations on an ARM multi-core system such as the ARM11 MPCore and Cortex-A9 MPCore.

The target audience for this document is kernel level programmers, device driver developers and firmware designers that need to program at a low level on a coherent shared memory ARM system. This document highlights features and behaviors of the underlying hardware and provides guideline recommendations so that the developer can use and program the DMA engine efficiently and avoid software design mistakes that would result in poor performance.

Examples given in this document are taken mainly from the Linux 2.6 OS kernel, but the concepts apply to any modern Operating System that allows DMA.

## 2. Cache coherency in multi-core processing systems

The concept of Symmetric Multi-Processing (SMP) refers to a processor composed by two or more equivalent cores sharing main memory with equal access rights to it. Generally an Operating System would be running across all cores, transparently distributing tasks. When these cores feature local caches, a mechanism must be used to keep them coherent.

Processors such as the ARM11 MPCore and the Cortex-A9 MPCore feature a hardware block known as the Snoop Control Unit (SCU). When enabled, the SCU automatically maintains coherency between the data caches local to each CPU.

### 2.1 Coherency Protocols in ARM MPCore processors

In a cached, shared memory, multi-core system, the mechanism implemented to maintain coherency between all CPUs' local caches is called the *cache coherency protocol*. The cache coherency protocol is a state machine that governs the condition of each cache line in each core's cache at a given time. This is implemented by tagging all cache line with an identifier of their state in respect to overall system coherency and cache lines in other cores. A hardware control unit automatically manages the states.

The ARM11 MPCore and Cortex-A9 MPCore processors support the MESI cache coherency protocol. In a correctly configured system, every cache line is dynamically marked with one of the following states:

#### **Modified (M)**

The subject cache line is present only in the current cache and it is dirty (not up to date with the next level of the memory hierarchy, L2 cache or main memory).

#### **Exclusive (E)**

The subject cache line is present only in the current cache, and it is clean (it is up to date with the next level of the memory hierarchy, L2 cache or main memory).

#### **Shared (S)**

Indicates that the subject cache, other than being up to date with the next level of memory hierarchy, is also stored (duplicated) in one or more other core's caches.

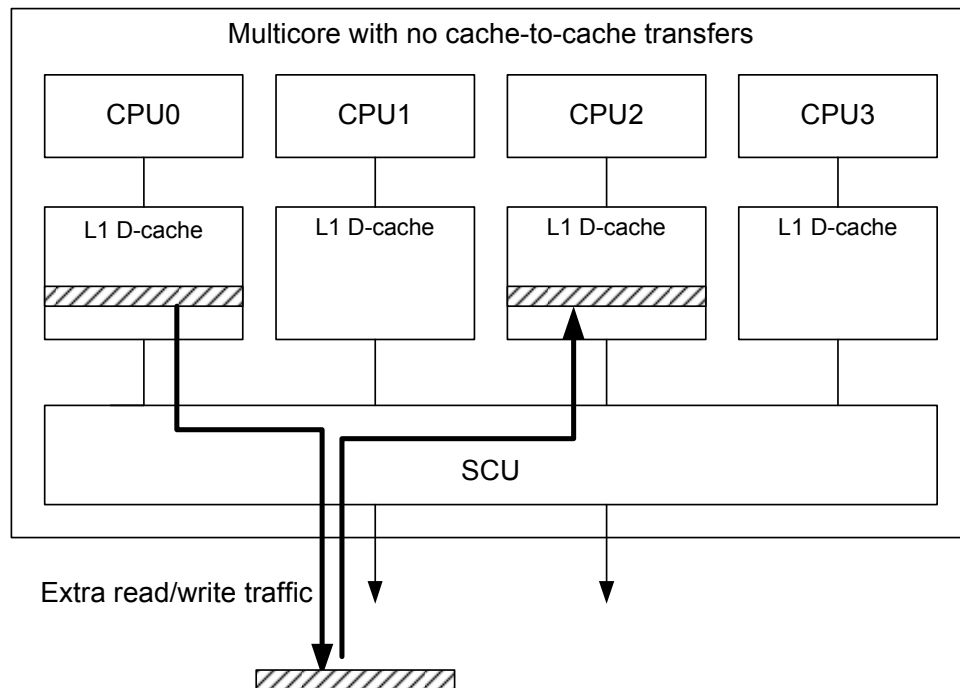
#### **Invalid (I)**

The subject cache line is invalid.

In an ARM MPCore processor, the coherency protocol is implemented and managed by the Snoop Control Unit (SCU). The SCU effectively monitors the traffic between local L1 data caches and the next level of the memory hierarchy. At boot time, each core can select to partake in the *coherency domain*, in which case the SCU will maintain coherency between them.

An important fact about symmetric multi-core processors is that the detail of which process runs on which core is controlled by the Operating System. In reality, unless explicit system calls bound a task to a specific core (this is known as *CPU affinity*), the likelihood is that that task will at some point migrate to a different core, along with its data as it is used.

In a literal implementation of the MESI cache coherence protocol, it is quite inefficient for a migrated task to access memory locations that are stored in the L1 (write-back) cache of another core. First the original core will need to invalidate and clean the relevant cache lines out to the next level of the memory architecture. Once the data is available at a shared level of the memory architecture (e.g. L2 or main memory), then it would be loaded into the new core.

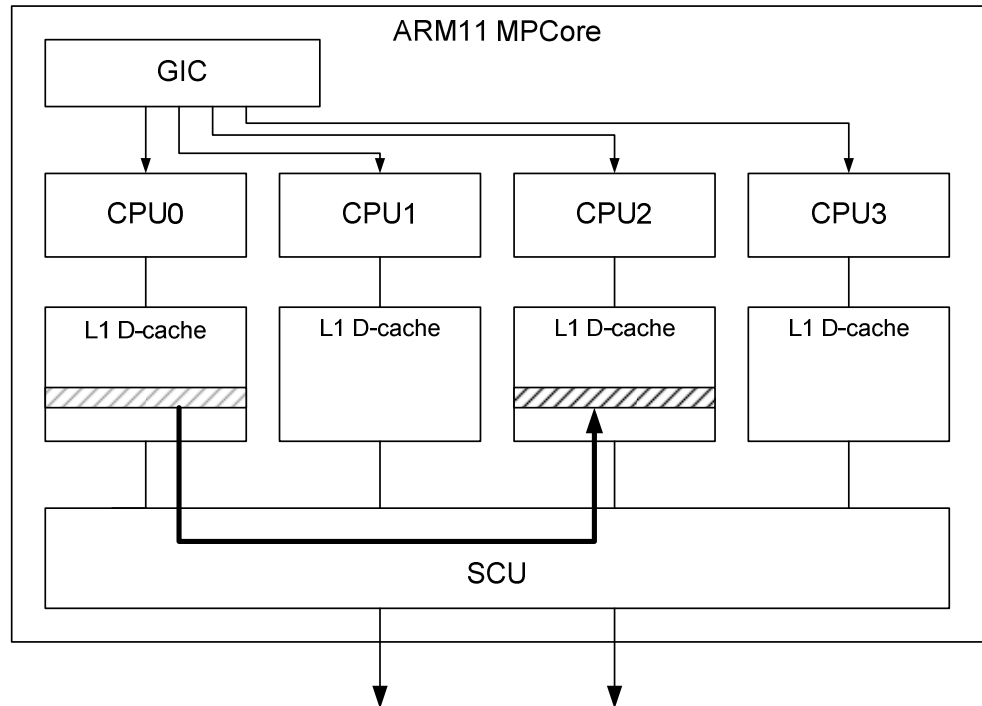


Note the above diagram is an illustration – ARM multi-core designs do not implement MESI in this way.

Whilst maintaining compatibility with the MESI protocol, the ARM11 MPCore and Cortex-A9 MPCore processors implement performance and power optimizations that address this shortcoming.

**Direct Data Intervention (DDI):** The SCU keeps a copy of all cores caches' tag RAMs. This enables it to efficiently detect if a cache line request by a core is in another core in the coherency domain before looking for it in the next level of the memory hierarchy.

**Cache-to-cache Migration:** If the SCU finds that the cache line requested by one CPU is present in another core, it will either copy it (if clean) or move it (if dirty) from the other CPU directly into the requesting one, without interacting with external memory.



In addition to the inherent performance benefits (in particular in a system without L2 cache), such optimizations reduce memory traffic in and out the L1 cache subsystem, in turn reducing the overall load on the interconnect, and reducing power consumption by eliminating interaction with the external memories.

DDI and cache-to-cache transfers particularly benefit the real life example of running an SMP operating system, where tasks and data can migrate between cores. It must be noted that effectively they are a method of reducing the side-effect of poorly designed software: They should not be relied upon when designing software for multi-core systems.

Techniques such as CPU task affinity are a more beneficial approach to improve system performance, particularly for cache intensive activities such as device drivers operating DMA (since they take advantage of warm caches). In reality, modern operating systems such as Linux 2.6 will try to keep tasks running on the same CPU (this practice is known as *soft affinity*).

Since Linux 2.5.80 the following system calls were introduced:

```
#include <sched.h>

int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);

int sched_getaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
```

These system calls are Linux specific, and enable to bound task identified by its process identifier to a subset of cores (using a mask).

Alongside NPTL (Native POSIX Thread Library), glibc version 2.3.4 provides non-standard POSIX extensions to abstract the above system calls:

```
#include <pthread.h>

int pthread_attr_setaffinity_np(pthread_attr_t *, size_t, const cpu_set_t *);

int pthread_attr_getaffinity_np(pthread_attr_t *, size_t, cpu_set_t *);
```

These are not recommended for use since they are non-portable (hence the suffix *-np*).

Most modern Operating Systems and Real Time Operating Systems (RTOSs) provide system calls that implement thread/process affinity.

## 2.2 Coherency and non-CPU bus masters

Modern embedded systems often feature multiple bus masters. Various levels of cache memories are also adopted in order to provide the best performance. In this scenario, maintaining coherency between the CPU and the data generated or consumed by I/O devices can be challenging, with an optimal solution which depends on application and CPU cache subsystem characteristics.

Points to consider are:

- how much of the DMA data is actually processed by the CPU ?
- is the DMA data cached by the CPU (and, whether at L1 or L2 level) ?
- is there any I/O coherency?

Cortex-A9 is the first ARM application processor to offer full I/O coherency via the Accelerator Coherence Port (ACP). This is discussed further in a dedicated section later in this document.

## 3. Introduction to Direct Memory Access (DMA)

### 3.1 DMA Basics

Direct memory access (DMA) is a vital part of many high-end systems. It allows additional bus masters to read or write system memory independently of the CPU(s). DMA channels can transfer blocks of data to or from devices with no CPU overhead.

The CPU manages DMA operations via a DMA controller unit. While the DMA transfer is in progress, the CPU can continue executing code. When the DMA transfer is completed, the DMA controller will signal the CPU with an interrupt.

Typical scenarios of block memory copy where DMA can be useful are network packet routing and video streaming. DMA is a particular advantage in situations where the blocks to be transferred are large or the transfer is a repetitive operation that would consume a large portion of potentially useful CPU processing time.

### 3.2 Using DMA in cached systems

When using DMA in a cached system, the software designer/programmer must pay attention to the behavior of the underlying memory subsystem. Using DMA in a cached system can have some practical implications, both in single and multi-core processor configurations.

Consider a CPU with a cache and a DMA accessible external memory, with a write-back, rather than write-through cache policy. A write-back cache will often contain more recent data than system memory. If the cache is not cleaned before the external DMA engine reads the system memory, the device will receive a stale value. Similarly, if the cached copy of the address is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value.

In order to address the aforementioned scenarios, the following approaches are commonly adopted.

Conventional non-I/O-coherent systems (like the ARM11 MPCore) leave the responsibility to software: the OS must ensure that the cache lines are cleaned before an outgoing DMA transfer is started, and invalidated before a memory range affected by an incoming DMA transfer is accessed. This introduces some overhead to the DMA operation, as most CPU cores require a loop to invalidate each cache line individually. Often, this must be done separately for L1 and L2 memory. In particular L2 cleaning/flushing of a large region can be time consuming. The OS must



also make sure that the memory range is not accessed by any other running threads in the meantime (both in the single and multi-core cases).

I/O coherent systems (e.g. Cortex-A9 with ACP) implement a hardware mechanism where accesses to shared DMA memory regions are routed to the cache controller which will invalidate (for DMA reads) or cleans (for DMA writes) the relevant cache lines.

## 4. Considerations on using DMA

Most modern operating systems, including Linux, provide a DMA API capable of handling coherency between CPUs and external devices accessing the same physical memory.

The following cases are possible in the context of Linux device drivers:

- Use uncached memory mapping from kernel space, usually allocated using `dma_alloc_coherent()`
- Use uncached memory mapping from user space, usually created with `dma_mmap_coherent()` (in the kernel driver)
- Use cached mapping and clean or invalidate it according to the operation needed (`dma_map_single()` and `dma_unmap_single()`)

When opting for un-cached memory mapping, the memory utilized can be configured either as *strongly ordered* or *normal uncached*. Strongly ordered memory is memory configured to be uncached and un-buffered: the changes made by a CPU on a shared strongly order locations of memory are immediately visible to all cores in the system (and do not require barriers). Normal un-cached is buffered memory, so memory barriers will be required. The cache is cleaned and invalidated when the mapping is created so there can be no stale data.

When using cached memory, the driver specifies the direction of the DMA operation (`FROM_DEVICE` or `TO_DEVICE`) so that the cache is cleaned or invalidated accordingly. Using cached memory mapping may be preferred in many circumstances (even where it gives rise to memory coherency issues requiring extra software clean/invalidate operations) as it is likely to result in a much more efficient set of bus accesses and faster overall performance.

*Zero-copy* DMA should be used wherever possible: User pages are mapped for DMA and the transfer takes place from device memory directly into user space. A traditional (non-zero-copy) solution would copy the data across several intermediate buffers between user and kernel spaces, adding considerable overheads. Linux 2.6 supports zero-copy DMA functionality for all block devices.

### 4.1 ARM11 MPCore

The ARM11 MPCore SCU does not handle coherency consequences of CP15 cache operations like clean and invalidate. If these operations are performed on one CPU, they do not affect the state of a cache line on a different CPU. This can result in unexpected behavior if, say, a line is cleaned/invalidated but a subsequent access hits a stale copy in another CPU's L1 through snooping the 'coherency domain'.

The different DMA use cases above require some (initial) memory allocation followed by one or more CP15 cache operations. If the allocated memory was in use by a different CPU, the cache lines might be in modified, exclusive or shared state on that CPU and so might not be present on the current CPU running the DMA device driver.

As CP15 cache maintenance operations on the current CPU don't affect the cache of the other CPU, there are several potential problems. Not all of these scenarios are likely with the Linux use cases but may be seen in other systems.

- Another CPU writes the data to a cached memory buffer. The DMA restart operation is done by the current CPU which does a cache clean operation. None of the modified cache lines on the other CPU are cleaned, causing stale data to be transferred.

- Current CPU invalidates the cache before an incoming DMA transfer writes new data. The CPU then reads the transferred data from the cacheable memory location. If the cache of another CPU contains an address within the DMA buffer, the SCU may take the stale data directly from the cache on that CPU with no access to external memory.
- Current CPU writes to the uncached mapping but cache lines in the modified state on the other CPU may be evicted to Level 2 cache or main memory, corrupting part of the data written by the current CPU.
- An external device writes to memory, but cache lines in the modified state on another CPU may be evicted, over-writing parts of the data written by the external device.

There are several solutions to accommodate DMA on ARM11 MPCore:

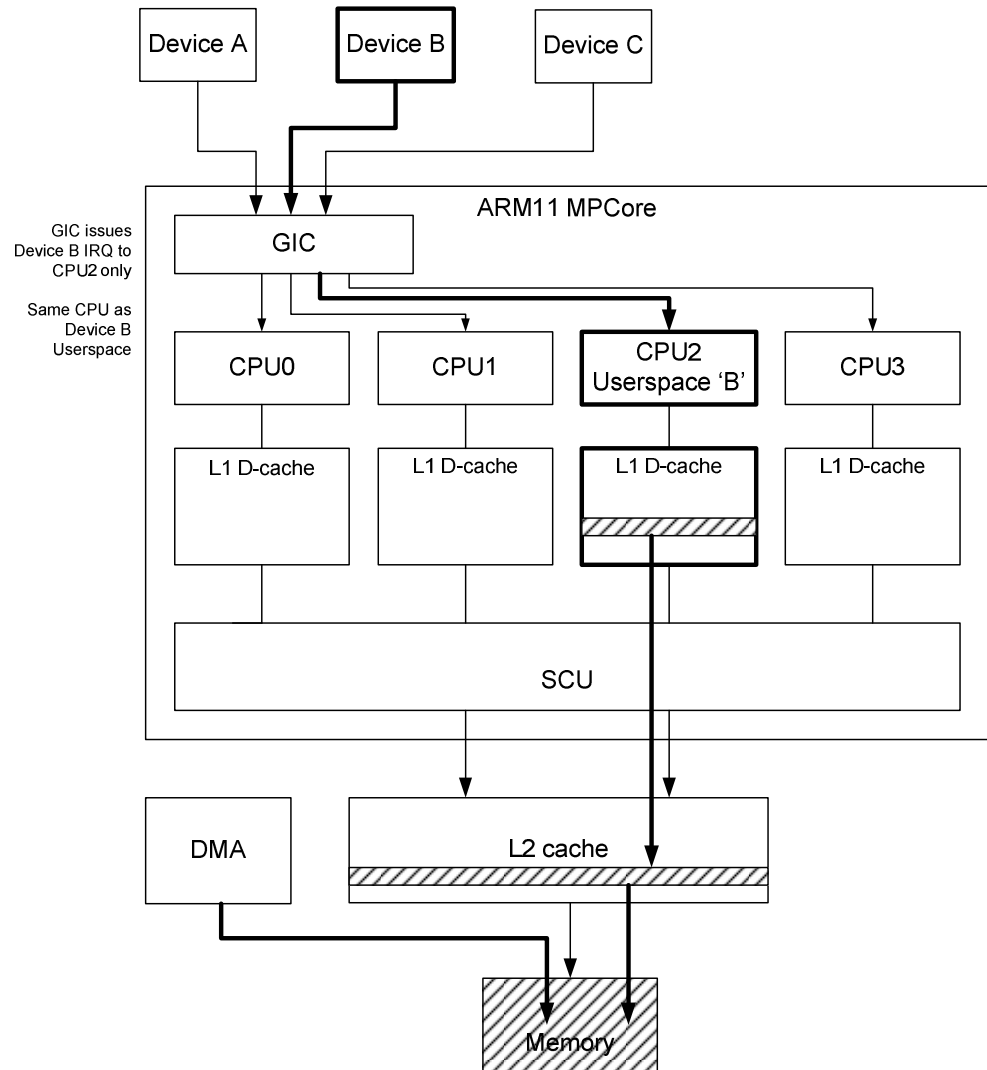
#### **a) Use uncached mappings for DMA memory**

Uncached mappings can have performance implications, but are probably the best solution for small DMA buffers. It is particularly appropriate if the DMA data is not frequently accessed.

#### **b) Use cached mappings but ensure that only one CPU deals with the DMA memory directly ('set affinity').**

The mechanism by which this may be done is OS specific, but in a typical SMP OS will involve setting the CPU affinity of the user-space application associated with the DMA. Any IRQ used by the device if the driver calls cache maintenance operations in the interrupt handler may also need to be restricted to the same CPU.

In addition to the low-level Linux setaffinity calls, Linux cgroups may be usable to set a group affinity without changing the application program.

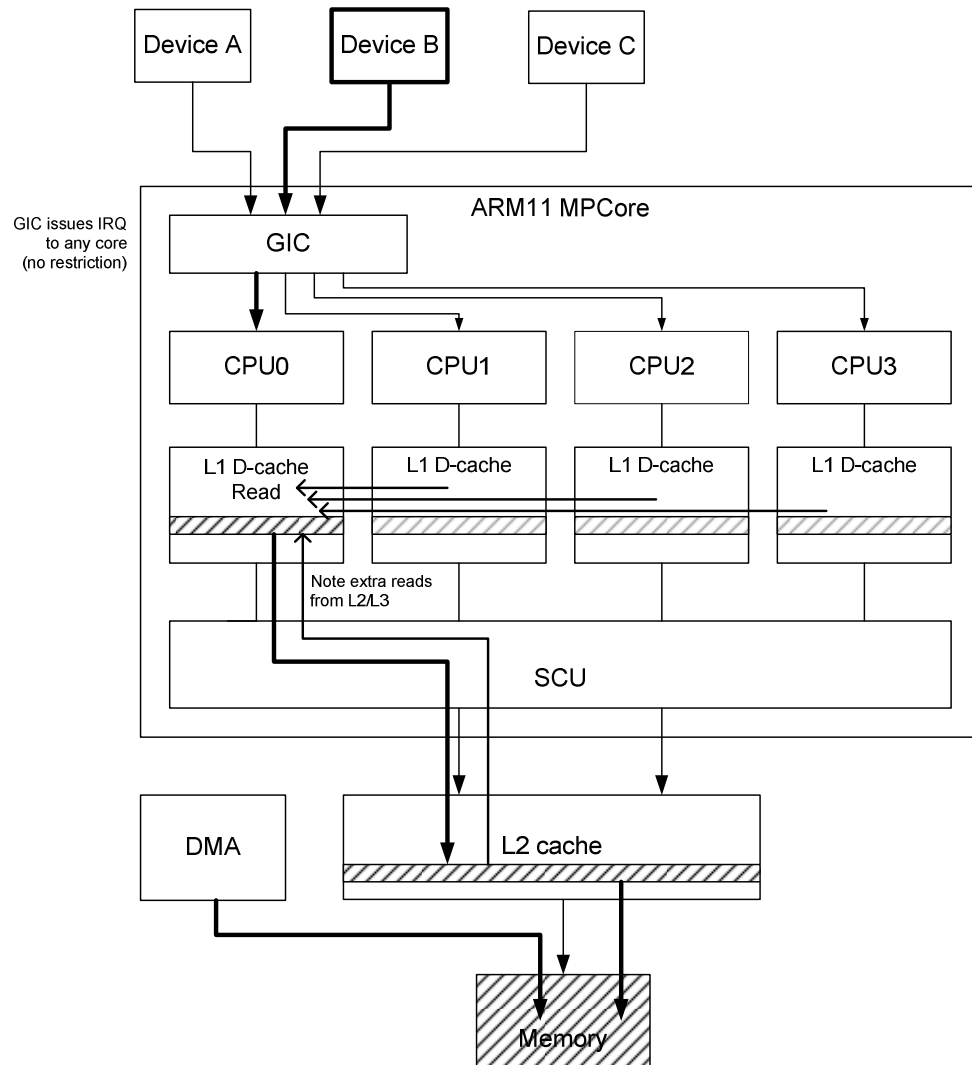


### c) Ensure cache lines are exclusive on driver CPU ('read for ownership')

Use cached mappings but ensure that the relevant cache lines are exclusive to the current CPU or in shared (NOT modified) state if they are on other CPUs. This requires the current CPU to perform a read of every corresponding cache line for the entire DMA block of memory before any clean/invalidate operations. This solution could be time-consuming, and assumes that it can be guaranteed that the other CPU will not modify the buffer contents during or after this read-for-ownership. It does however allow full SMP operation of the driver or application on any CPU.

The disadvantage of this approach is when very large blocks of data (much larger than the L1 cache) are being processed. The entire block must be read since the contents could exist in any CPU's L1 cache. This can cause thrashing in the driver's cache. This technique is much better suited to small blocks.

The following diagram illustrates the TO\_DEVICE case where reads are required:



In comparison, in the FROM\_DEVICE case, a CPU needs to invalidate its caches before the device transfers data into memory so that any dirty cache lines aren't evicted during the transfer (which could corrupt transferred data) and there are no stale entries in the cache. Since dirty cache lines or stale data may be present on other CPUs, a broadcast operation is normally required but the equivalent optimization is to write (rather than read) a word in every cache line before invalidating the cache (writing is needed to ensure that stale data on other CPUs is invalidated). This is similar to the above diagram but with writes instead of reads, which force the lines to become modified in the current CPU's cache.

#### d) Broadcast of cache maintenance operations

Broadcast the cache operations in software to the other CPUs using the Inter-Processor Interrupt (IPI) mechanism. There are a number of hazards to be aware of:

- The CPU which initiates the cache operation must wait for the other CPUs to complete the operation and issue a response to show that they have completed. This causes a delay which can vary depend on the amount of other IRQ processing.

- The IPI uses IRQ and cannot be done if interrupts are disabled on the current CPU. The current Linux implementation allows the DMA operations to be broadcast with interrupts disabled. Sending the IPI is protected by a spinlock. If the CPU sending the IPI cannot acquire the spinlock and interrupts are disabled (it uses `spin_trylock` rather than `spin_lock`), it means that another CPU is sending an IPI and it enters a polling mode for receiving the IPI.

Linux supports this with 2.6.28-arm1 (ARM Ltd stable version on [www.linux-arm.org](http://www.linux-arm.org)) onwards.

An OS could combine support for “c) Ensure cache lines are exclusive on driver CPU” and “d) Broadcast of cache maintenance operations”. For small blocks, (c) is performed, and for larger blocks (d) is performed.

Technique	Recommended for	Not recommended for
<b>a) Uncached mappings</b>	Infrequently or sparsely accessed DMA data blocks  Simplest implementation  Correctness/reliability	Intensively accessed DMA blocks will not benefit from caching, in which case performance can suffer.
<b>b) Set affinity</b>	Minimizing cache line migration overheads	In complex applications, a manual procedure will be required to map the IRQ part of device drivers and user processes to CPUs. Thorough testing will be required.  In complex applications, the enforced mapping of processes to CPUs might mean that the maximum aggregate performance is not achievable due to unbalanced CPU loading.
<b>c) Read for ownership</b>	Small DMA data blocks.  Can be done by OS kernel	Large DMA data blocks – must read/write completely – will cause extra memory traffic and potentially thrash L1 cache.
<b>d) Broadcast of cache maintenance operations</b>	Can be done by OS kernel  Currently done by Linux 2.6.28-arm1	Small DMA blocks – overhead of synchronous IPI is likely to be significant.  Performance – scheduling of IPI depends on IRQ loading – process blocks until all CPUs have responded.

## 4.2 Cortex-A9 MPCore

On Cortex-A9 MPCore, cache maintenance operations can be broadcast by hardware to other CPUs in the inner shareable domain. This is highly recommended so that coherency issues may be avoided.

Cache operations are only broadcast for addresses in the coherent (inner, shareable) domain. Operations on Non-Shared addresses are not broadcast. The CPU on which the Cache operation is performed does the Virtual to Physical address translation. The cache operation is then broadcast to the other CPUs with the Physical address.

A CPU will send broadcast cache maintenance operations only when both SMP and FW bits are set.

A CPU will receive the broadcast operations when its SMP bit is set, regardless of the FW bit value.

The advantage of Cortex-A9's hardware based approach, is that cached mappings can be used, and the broadcast happens in hardware automatically with very low overhead. This increases the performance and simplifies the software.

## 4.3 The Accelerator Coherency Port (ACP) and I/O Coherency

The Accelerator Coherency Port (ACP) is an optional feature of Cortex-A9, which provides an 64-bit AXI slave port that can be connected to a DMA engine, providing the DMA access to the SCU of Cortex-A9. Addresses on the ACP port are physical addresses which can be snooped by the SCU to provide full I/O coherency. Reads on the ACP port will hit in any CPU's L1 D-cache, and writes on the ACP port will invalidate any stale data in L1 and write through to L2. This can give significant system performance benefits and power savings, as well as simplifying driver software.

The ACP allows a device, such as an external DMA, direct access to CPU-coherent data regardless of where the data is in the CPU cache and memory hierarchy. It provides automatic coherency similar to that provided inside the Cortex-A9 MP between the CPU's L1 D-caches.

From a software perspective, writers of device drivers that use ACP do not need to perform cache cleaning or flushing to ensure the L2/L3 memory system is up-to-date. Memory barriers (DMB) may still be required to ensure correct ordering.

ARM Linux kernel can support `arch_is_coherent()`, which means all DMA I/O is coherent. In this case, the internal Linux kernel `clean/flush dma_cache_maint()` calls are not required and the `dma_map_single()`, `dma_unmap_single()` calls become much simpler.

Further discussion of ACP usage is outside the scope of this document.

## 5. Conclusion

Multi-core SMP-based ARM11 MPCore systems need careful implementation of DMA device drivers to ensure highest performance. This application note has explored some possible techniques:

- use uncached mappings
- set CPU affinity for driver/application

- use 'read for ownership' for data block, to ensure lines are migrated to device driver's CPU

There is no single best recommended approach as it depends on the application: DMA block size, access pattern from CPU, etc.

On Cortex-A9 based systems, there are significant enhancements that permit the highest I/O performance using DMA:

- Broadcasting of cache maintenance operations
- ACP port for full coherent I/O
- Programmable prefetch engine (Cortex-A9 r2 and later) to minimize read latencies