Shared Data in Asymmetric Multiprocessing (AMP) Configurations

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

Abstract **-** Modern system-on-module (SoM) platforms may have heterogeneous processor devices in asymmetric multiprocessing (AMP) configurations. Two widely accepted architectures from NXP are the i.MX 8 QuadMax and i.MX 8 QuadPlus series that include varying counts of ARM Cortex-A and Cortex-M CPU cores, and ST Micro's STM32MP15x lines. These designs are frequently used in multimedia, automotive, and industrial controller applications. The heterogeneous multicore architecture allows the offloading of critical hard real-time tasks to the Cortex M processors for extremely low latency processing, while using the Cortex-A cores for high-performance tasks. In addition, these SoCs integrate shared memory and hardware modules intended to provide communication capabilities between the Cortex-A and Cortex-M cores. Different CPUs usually run instances of different operating systems, e.g., Linux on the A cores and bare-metal applications or real-time OS (FreeRTOS, AutoSAR, etc.) on the M core(s).

Traditionally on-board shared memory was used for communications / message passing only. Yet due to the high complexity and critical nature of applications utilizing the AMP hardware, and the amount of the external memory, real-time data can often be shared between the Cortex-A and Cortex-M cores. This article will discuss the design and challenges of a shared data implementation for AMP configurations, explore implementation options to synchronize access to shared data, and discuss example use cases.

Heterogeneous multicore systems

Heterogeneous multicore systems are becoming ever more popular for automotive and industrial applications due to their high performance and energy efficiency. Heterogeneous systems have two or more cores with different instruction set architectures with more than one operating system running on the device. By dividing tasks between different processors, heterogeneous system designs cover multiple requirements such as energy efficiency, performance and safety, and make them fit for real-time critical applications.

This architecture is known as Asymmetric Multiprocessing (AMP). An AMP system has multiple CPUs, each of which may be a different architecture (but can be the same). Each CPU has its own address space (though some of the memory may be shared with other cores). The system is typically equipped with a communication facility between the CPUs, normally a hardware messaging unit, and DDR shared memory. In addition, more than one operating system runs independently on one or more processors with different architectures. For example, NXP i.MX8x with embedded Linux on the ARM Cortex-A and FreeRTOS or Vector's MicroSAR on the ARM Cortex-M4.

This is in contrast to Symmetric Multi-Processing systems (SMP) that refers to systems with multiple CPUs each of which has the same architecture. SMP designs are used when an embedded application simply needs more CPU power to manage its workload, similar to multi-core CPUs used in desktop computers.

Multiple vendors offer AMP hardware. Two widely accepted architectures are NXP's i.MX 8 QuadMax and i.MX 8 QuadPlus series that include varying counts of ARM Cortex-A and Cortex-M CPU cores. Another popular alternative includes STM's STM32MP157F devices that offer dual-core Arm® Cortex®-A7 in combination with a Cortex® -M4 32-bit RISC core. Throughout this paper we

use the NXP's i.MX8 hardware as our reference. That said, similar provisions apply to other AMP hardware, including the above referenced STM devices.

The shared memory regions integrated with the AMP hardware are directly accessible allowing for communications and/or message passing between the "clusters" (a cluster refers to several cores capable of independent instruction execution and running a separate operating system). Traditionally, shared memory was used for communications / message passing only. Yet due to the high complexity and critical nature of applications utilizing the i.MX8 hardware, and the amount of external memory, real-time data can be shared between the Cortex-A and Cortex-M cores.

The need for a real-time storage system in applications that utilize AMP hardware is quite wide-spread: power grid controllers must share their sensor readings and other real-time data collected by a FreeRTOS-based low-latency application running atop of the Cortex-M clusters with the fast yet complex processing on the Linux side running atop of the Cortex-A clusters; an AMP-based drone system utilizes different clusters for navigation parameters acquisition and for real-time processing running different real-time operating systems on each cluster.

Overview

Conceptually, a database subsystem in the AMP systems can be organized in two ways:

1. A client-server methodology is used when the database is maintained within the scope of a process running on a single cluster. This process has exclusive access to the database. The data is then available to any other processes running within the same cluster or a different cluster through "messages" passed to the "server" process through the shared memory. The message passing is coordinated by the specialized hardware integrated within the SoC. The database management system processes the "messages" through a dedicated "server" application that then writes and reads the storage using conventional methods and sends back a reply to the "client" process. AMP hardware/ firmware normally integrates some means of communication between clusters.

2. An alternative method is to keep the database in the device's shared memory and make it directly accessible from all clusters' processes/applications. The implementation must maintain control over the flow of data, synchronize data access and ensure logical and perhaps temporal consistency of the data, equipping applications with methods to write and read the data, such as a C-call API or other means and makes use of the available hardware resources, e.g., the i.MX8 Messaging Unit.

This paper discusses the second approach. We find it attractive to data-driven applications for several reasons: the performance is higher with direct access to storage, and the storage is better utilized because the entire DDR memory can be used for storage. Directly managing database access from multiple clusters presents greater flexibility to multiple threads by fine-tuning contention resolution. In addition, the absence of a single point of failure – the server process, is essential for data driven safety-critical systems that often make use of the AMP hardware.

The relevant hardware modules

As indicated, the shared data design presented in this article was originally created for the NXP i.MX8 QuadMax device with embedded Linux running on the ARM Cortex-A and FreeRTOS running on the ARM Cortex-M4 side. The DDR memory that is used to maintain the database is deemed "external" from the viewpoint of the clusters' software. The shared access to common storage from multiple processes/threads dictates that the solution coordinates access between those threads through some means of synchronization. When all cores are located within the same "cluster" — running the same operating system, the operating system's synchronization primitives can be used, or serialization and synchronization mechanisms can be created directly through the hardware interrupts. In the AMP scenario all cores are completely independent, so the i.MX8 design presents two hardware modules that facilitate inter-process communication: the hardware semaphore called SEMA42 and the Messaging Unit (MU).

The SEMA42 is a memory-mapped module that provides hardware support for implementing semaphores and a simple mechanism to implement "lock" and "unlock" operations via a single-write, atomic access. The hardware semaphore module provides hardware-enforced "gates" as well as other useful system functions related to the gating mechanisms.

The Messaging Unit (MU) module enables two processors within the device to communicate and coordinate by passing messages (e.g., data, status and control) through the MU interface. The MU also provides the ability for one processor to signal the other processor using interrupts.

Both tools have their own benefits and disadvantages when put in practice and we will discuss the use of them further in this article. Note that the STM analog to SEMA42 — the Hardware semaphore (HSEM) and to the Messaging Unit — the Inter-processor communication controller (IPCC), provide similar functionality and facilitate similar software architecture.

On heterogeneous devices, different CPU architectures and the use of multiple operating systems on different processors presents considerable added complexity regarding synchronization concepts relative to symmetric (SMP) systems. These challenges are analyzed in more detail in the following sections.

Endianness and word size

The internal data structures and the storage layout must be impartial to different endianness and the word size (32- or 64-bit). This was uncomplicated. Both the i.MX8 Cortex-A and Cortex-M processors use little-endian architecture and the Cortex-M is a 32-bit processor. Therefore, the data layout is a 32-bit layout.

Addressing database memory

All cores from different clusters must have the ability to address the same physical memory region (where the shared data is kept). The i.MX8 clusters run different operating systems that implement different memory management unit (MMU) mechanisms. Thus, the shared data must be located in the physical memory region with a contiguous addressing, and the starting address and the size of the region must be known up-front.

Normally, AMP-based operating systems are configured so that not all shared memory is directly utilized by their respective kernels. The memory is divided into several segments. For example, the first segment is assigned and controlled by one OS (e.g., Linux), the second segment is used by another OS (e.g., FreeRTOS). The third segment is not directly used by any of the operating systems, but can be used for data exchange between the operating systems' kernels and other software modules run on different clusters. Therefore, this segment's address space is a perfect fit for a shared database as it can be managed exclusively by the database management system and is never directly "touched" by the operating systems' kernels.

As a rule, operating systems' kernels provide methods for addressing physical memory to software modules. In Linux (and other Unix-based operating systems) the/dev/mem is a character device file that provides access to the system's physical memory (as opposed to the virtual memory). The pseudo-device driver must be configured in the Linux kernel:

```
CONFIG_DEVMEM=y
CONFIG_STRICT_DEVMEM=n
```

and can be used to access a physical address:

```
int fd = open("/dev/mem", O_RDWR|O_SYNC /* oflags */));
void *vaddr =
    mmap(hint, size, PROT_READ | PROT_WRITE,MAP_SHARED, fd, (off_t) paddr);
```

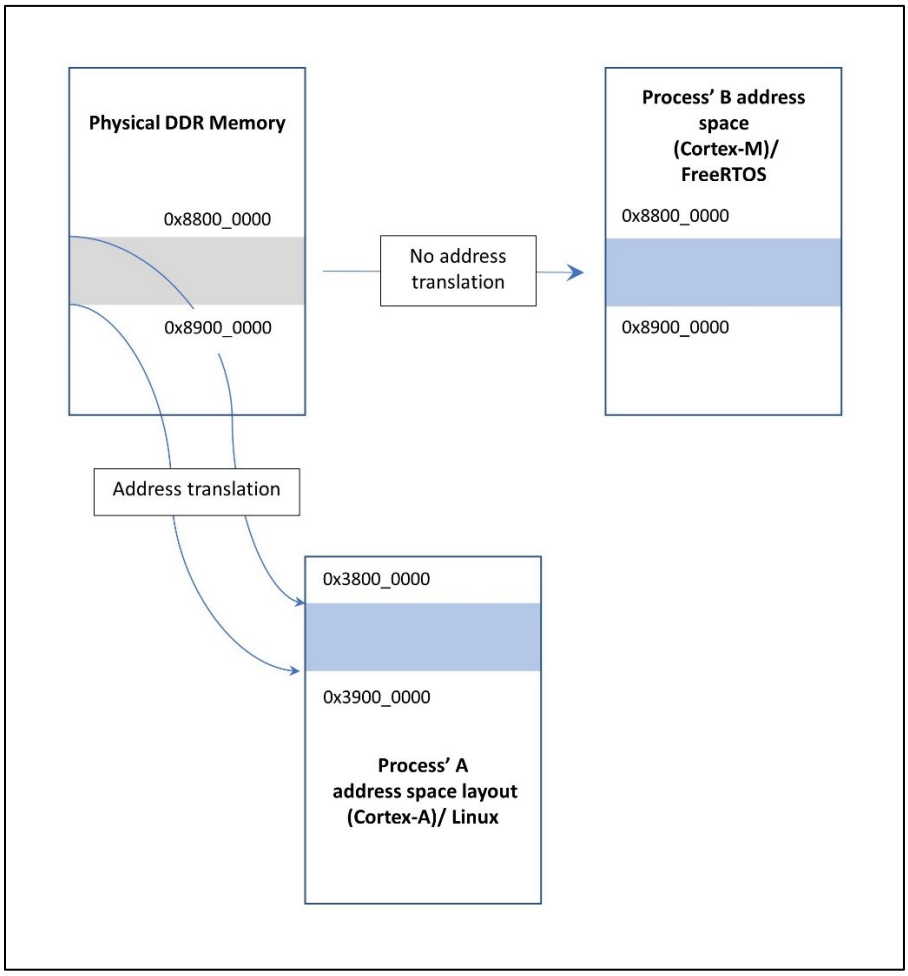
As trivial as it may look, let's review these two lines. In the first line the *open()* function establishes the connection between a file and a file descriptor, which is used for mapping the contents of the file into the processes' address space. Values for oflags are constructed by a bitwise-inclusive-OR of flags defined in <fcntl.h>. The O_RDWR flag indicates that the file is opened for reading and writing. The other flag that is relevant is O_SYNC. It indicates that write operations will flush data and all associated metadata to the underlying hardware bypassing L1/L2 CPU cache. The second line creates a memory mapping in the virtual address space of the calling process. The hint parameter specifies a preferred starting address for the mapping (which could be NULL, in which case the OS picks the address). The paddr is the offset from where the */dev/mem* file mapping is started. Note that from this point on, the performance of accessing a virtual address is roughly the same as accessing any other address within the processes address space (e.g., acquired through malloc()).

Embedded operating systems (e.g., FreeRTOS, Vector's MicroSAR and a number of others) often don't utilize MMUs. The OS kernel, BSPs and the application code are linked into a single

monolithic binary executable and address translation is neither necessary nor provided. In order to access physical memory, it is possible to simply use the physical addresses that are routinely described in the linkers' *.ld* files. For example (note the *m_data* and *m_data2* memory regions):

```
MEMORY
{
 m_interrupts      (RX)  : ORIGIN = 0x88000000, LENGTH = 0x00000A00
 m_text            (RX)  : ORIGIN = 0x88000A00, LENGTH = 0x001FF600
 m_data            (RW)  : ORIGIN = 0x88200000, LENGTH = 0x03000000
 m_data2           (RW)  : ORIGIN = 0x8C000000, LENGTH = 0x04000000
 m_tcml            (RW)  : ORIGIN = 0x1FFE0020, LENGTH = 0x0001FFE0
 m_tcmu            (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00020000
}
```

The same *.ld* file contains other attributes of the physical memory segment such as whether to bypass the L1/L2 cache while accessing the segment.



Picture 1 illustrates the concept, emphasizing the contiguous nature of virtual and physical memory segments.

What is cache coherency?

Coherency means ensuring that all processors within a system have the same view of shared memory and that changes to data held in the cache of one core are visible to the other cores, making it impossible for cores to see stale or old copies of data. In an AMP system each task can have a different view of memory and there is no requirement for hardware-managed cache coherency. The CPUs that belong to different clusters are completely independent and don't have any common cache controller that automatically writes data from memory of a given CPU to the external memory. Thus, memory changes made by processes running in one cluster (e.g., Cortex-A-based) are invisible to processes running in another (Cortex-M-based) and vice versa unless some explicit action is taken create the visibility.

Obviously, modifications made to data in shared memory (and committed to storage) by any process must be made visible to all other processes connected to the data. This can be handled in two ways:

- by simply not caching shared memory locations, and
- by controlling the cache state by "cleaning" and "invalidating" the cache or individual cache lines explicitly

While perceived as a simpler and more reliable solution, disabling caching typically has a high-performance cost. Perhaps not so strikingly noticeable on the Cortex-M / FreeRTOS side because of the relatively slower CPU, but quite dramatic on the Cortex-A/Linux - based cluster (an order of magnitude for an average transaction). Therefore, despite the added complexity the clean/invalidate approach is well justified.

The ARM Developer documentation states that "*...the invalidation of cache or a cache line means clearing it of data. This is done by clearing the valid bit of one or more cache lines*." In other words, if a cache line was invalidated, attempting to read from the address would trigger a read access to memory. Cleaning a cache or cache line means writing the contents of dirty cache lines out to main memory and clearing the dirty bit(s) in the cache line. This makes the contents of the cache line and main memory coherent with each other. Note that the clean and invalidate commands can be applied to a cache line either by its address (virtual or physical), or by the cache line "coordinates" set and way: "*a way is a subdivision of a cache,…. A set consists of the cache lines from all ways sharing a particular index*."

Incidentally, the problem resembles a well-known headache with memory reordering optimizations related to store buffers and invalidation queues that exist in most modern processors. The obstacle is removed through the memory read and write barriers — a necessary evil that enables good performance and scalability by enforcing ordering on memory operations.

Applications to the Database Kernel

Database transactions

Recall that the premise of this paper is that the high complexity and critical nature of applications utilizing AMP architecture manage a significant amount of complex and real-time data that must be shared. Complex data means that the concern is not a single integer, or a single C structure. Rather, the concern is with multiple inter-related structures, and those relationships must be maintained. Normally, this is the work of a database management system. One of the key database management system (DBMS) concepts is that of database transactions, a powerful notion that differentiates database storage from other storage paradigms (such as a file system). A database transaction is a unit of work that is either completed as a unit or undone as a unit. Transaction processing is crucial in maintaining the integrity of a database (meaning that data in the database was collected and stored accurately, and is contextually accurate to the data model). DBMS transactions have two main purposes:

- to provide reliable units of work that allow correct recovery from failures and keep a database consistent, and
- to provide isolation between programs accessing a database concurrently.

A database transaction, by definition, must be atomic (integrity of the entire database transaction, not just a component of it), consistent, isolated and durable. These transactions' characteristics often referred to as *ACID properties*. Cache coherency discussed above is vital to enforcing transactions' ACID properties.

If you're going to exploit the AMP architecture manage a significant amount of related data, you're going to need to implement at least lightweight transactions.

Enforcing cache coherency in transactions

For the purpose of this discussion let's assume a simple implementation that allows one "write" transaction (a transaction that makes any modifications to the database) and multiple "read" transactions to run concurrently. Let's consider the following (high-level) approach to cache synchronization and note that the description below is streamlined for clarity.

1. At the start of any transaction the local cache is invalidated prompting the transaction to read the data from physical memory (see the *read_invalidate()* function below)

2. At the end of a "write" transaction the cache is cleaned, which leads to pushing all modifications from the local cache to the physical memory (see the *write_clean()* function below)

This two-step, brute-force approach of dropping all cache lines at the end of each transaction carries a high-performance penalty. The obvious optimizations are to decrease the number of cache lines participating in the invalidate/clean operations and to minimize the number of *invalidate* and *clean* calls.

At the end of a "write" transaction all modified shared memory ranges (database pages, in database vendors' parlance) should be written to RAM. For that purpose, several internal structures need to be maintained:

- the "write bitmap" (WBM). Each bit of the WBM corresponds to a single database page and the bitmap is also allocated *in the shared storage*, which makes it accessible by all connected tasks.
- variables that keep integer numbers identifying the cluster that was last used to *invalidate* the cache and the cluster that executed the last write transaction These variables are also kept in the shared storage and are accessible by all connected tasks.
- a list of all pages modified by the current transaction.

At the end of the transaction, you need to walk through the list of modified pages and mark them in the "bitmap". The modified list is then released. Note that multiple transactions can mark their modifications in the WBM, in the event that there are two or more sequential write transactions on the same cluster. At the start of a subsequent "read-only" or "write" transaction, you need to check whether the current transaction is running on the same cluster as the last transaction and whether the previous *invalidate* was issued on the current cluster. This verification is done through the cluster identification variables described above. If the cluster is the same, the cache was synchronized locally by the cache controller. However, if the last transaction was run on a different cluster (the cluster_id is not this cluster), the modified pages listed in the WBM should be invalidated. The pseudo code below illustrates the algorithm.

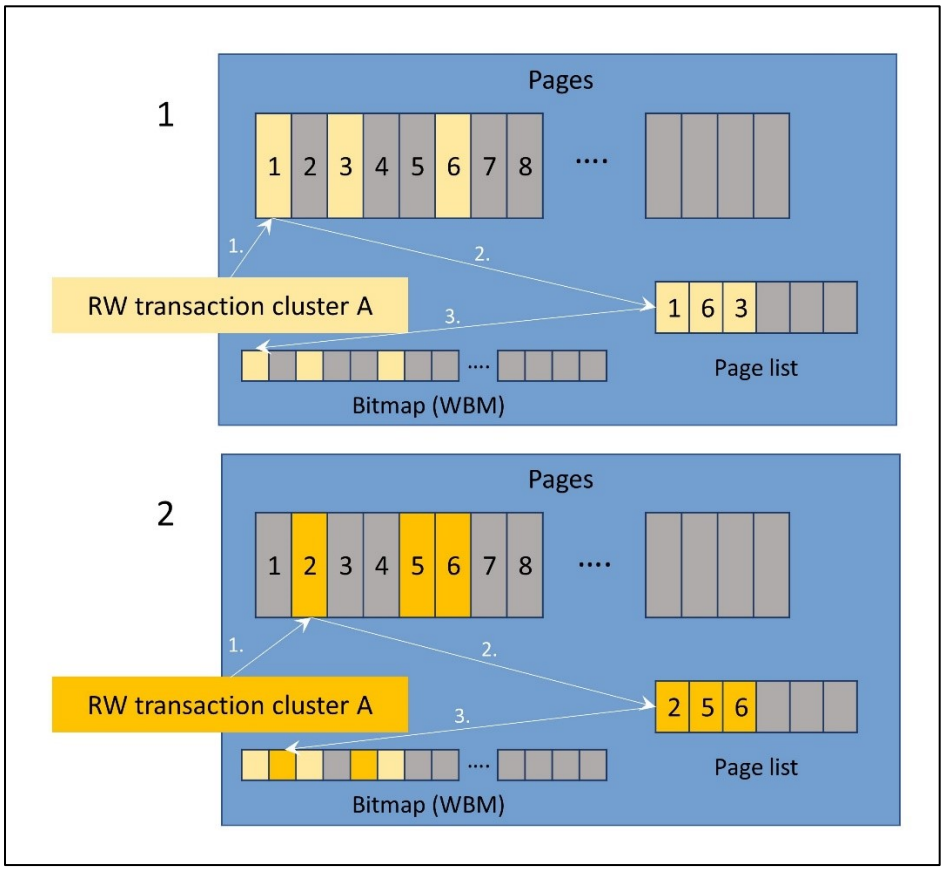
```
read_invalidate() {
  if (wbm->reader_id != current_id && wbm->writer_id != current_id) {
    for (each page in wbm->bitmap) {
      invalidate_page(page);
    }
    wbm->reader_id = current_id;
  }
}

write_clean()
{
  if (wbm->writer_id != current_id || wbm->reader_id != INVALID_ID) {
    mco_memnull(wbm->bitmap);
  }
  for (each page in dirty_list) {
    clean_page(page);
    set_bit(wbm->bitmap, page_num)
  }
  wbm->writer_id = current_id;
  wbm->reader_id = INVALID_ID;
}
```

To summarize, the modified pages are invalidated (forced to be re-read from RAM), at the beginning of each transaction if

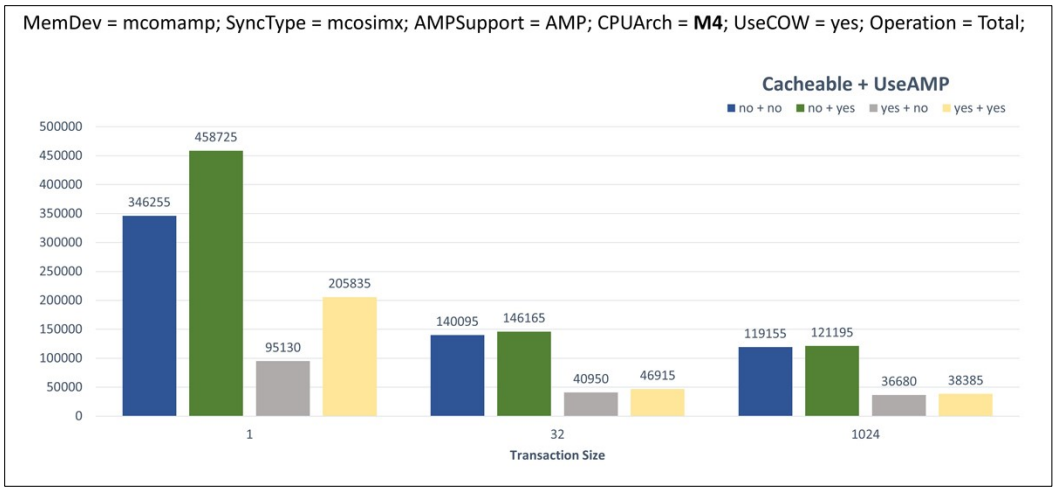
- the last "write" transaction was from a different cluster and
- the current transaction (either read-only or write) is the first one on the current cluster.

The RAM is written to (cleaned) at the end of a write transaction.

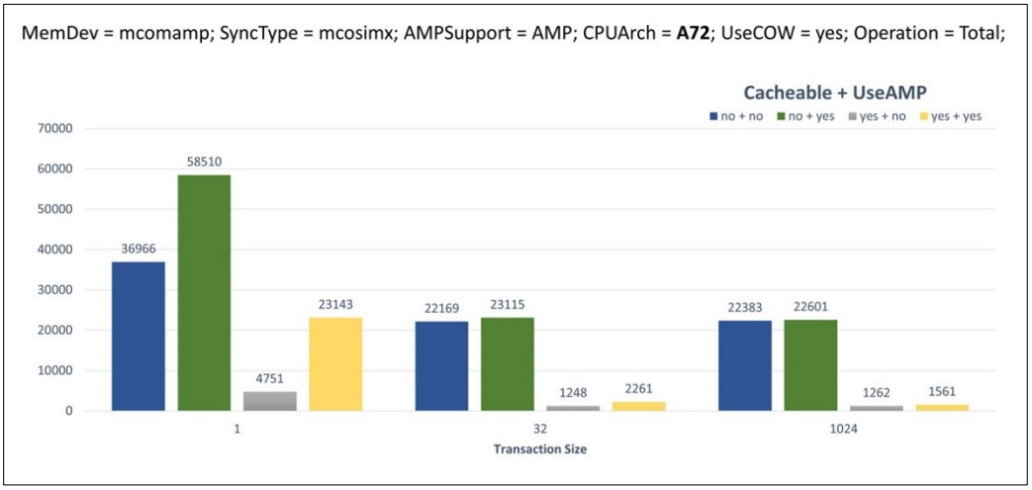
Another caveat is that the WBM itself must be synchronized between clusters too, meaning that the cache clean/invalidate operations must be applied to the bitmap once it has been modified. The bitmap can be rather large (for a gigabyte database with 256 byte pages the bitmap size is 512K). Thus, synchronizing the entire bitmap, especially when the transaction rate is high and the transactions are small (i.e., modify a small number of pages) is wasteful and has a highly negative impact on performance. To mitigate this, we can create a second-level "metamap" (see Picture 2) that bit-marks each cache line of the WBM that needs to be synchronized, reducing the number of addresses that must be invalidated. For example, given the same 1GB database and 256-byte pages and a 64-byte cache line, a 1K metamap (16 cache lines) would serve the 512K bitmap. If a transaction has modified a single database page, the cache operations would impact just 17 cache lines (16 of the metamap and 1 from the WBM) plus the actual modified database page.



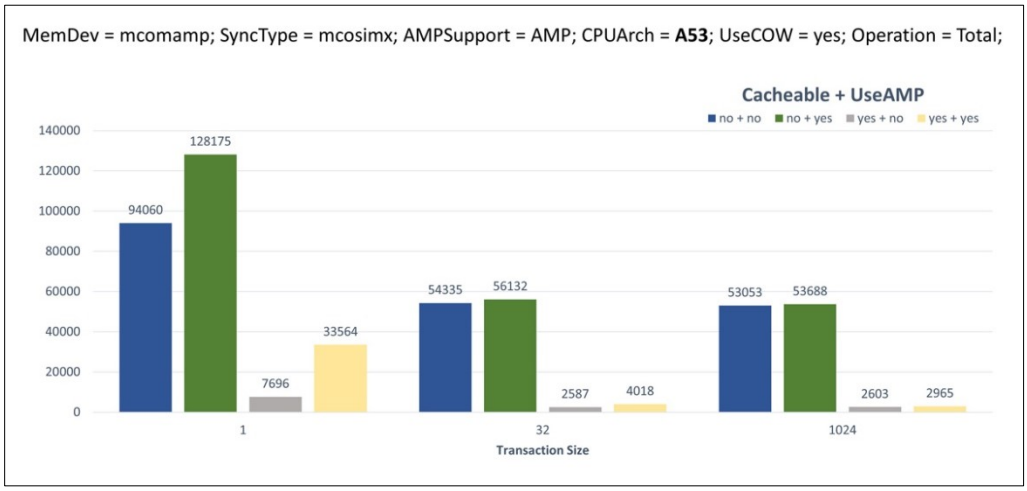
Picture 2. It is worth noticing that the transaction size (the number of objects that are modified) is an important factor in the overall performance of the system. The larger a transaction is, the more beneficial the effect of implementing the WBM and metamap. The diagram on Pictures 3 (for the Cortex M4 core), Picture 4 (Cortex-A72), and Picture 5 (Cortex A-53) illustrate the trend.



Picture 3. Cortex M4 core



Picture 4 Cortex-A72



Picture 5 Cortex A-53

To further complicate things, the database kernel has internal structures that could change even in the context of a read-only transaction. Access to these structures is protect via a synchronization semaphore, which in the AMP environment take a form of a *distributed semaphore* (discussed further in this article). Consequently, these shared structures must be synchronized across multiple AMP clusters: when the semaphore is taken the protected area must be *invalidated* and when it is released the protected memory has to be *cleaned*.

These optimizations are implemented only for environments with two clusters. The implementation can be extended to support multiple clusters.

Instrumentation for Serializing Database Access and the Distributed Semaphore

Inevitably, in implementing the sharing of complex data across cluster in an AMP architecture, you will find it necessary to maintain metadata, and that access to that data must also be synchronized. In the symmetric multi-processing (SMP) environment, a single instance of the operating system is used and this instance runs on all of the CPUs, dividing work between them. In this environment, serializing access is a matter of using high-level operating system-provided synchronization primitives (semaphores, events, barriers, etc.), utilizing hardware-provided atomic instructions, or accessing hardware interrupts directly. In a traditional environment, our job is to exercise good judgment while making use of those "instruments" to avoid excessive delays, while ensuring that necessary protection of the shared resources is in place.

Unlike SMP systems, AMP clusters are controlled by different operating systems. Thus, from the standpoint sharing of data across clusters, the AMP system can be viewed as a distributed network, in which "nodes" communicate with each other via shared memory. Therefore, we need a mechanism to synchronize access to metadata structures and consequently shared storage that is independent of each node's operating system. We call this mechanism a Distributed Semaphore.

Distributed Semaphore

The distributed semaphore is the component that controls multiple tasks' synchronization across the RTOS running atop of Cortex M cores and Linux running on top of the Cortex-A.

In real life, access to the shared data regions is inevitably assisted by special hardware and perhaps low-level OS kernel components. As indicated earlier, our referenced NXP's i.MX8 devices present two hardware modules that assist in these communications: the SEMA42 hardware semaphore and the Messaging Unit (MU). Prior to describing the distributed semaphore implementation let's review the available "instrumentation" options.

SEMA42 hardware semaphore

The SEMA42 module provides hardware support for implementing semaphores, and a simple mechanism for "locking /unlocking" access to shared resources via a single write access. The semaphore supports 16 hardware "gates", each of which can be either locked or unlocked. If the gate is unlocked, one and only one CPU is able to turn it into the locked state, access a shared resource and then unlock the gate.

When multiple CPUs are making attempts to access a shared resource, one of them could lock the gate. Other CPUs would have to wait until the gate is unlocked and repeat the attempt. This trivial logic resembles SMP's widely used Test-And-Set (TAS) instruction. A software module (such as an operating system or a database kernel) would need to poll the gate in a busy loop. Busy loops when used often lead to increased CPU and bus loads and greater latencies while accessing shared resources.

Messaging Unit

The Message Unit is a smarter device. The module integrates several 32-bit TX/RX registers and enables two clusters within the device to communicate and coordinate by passing messages (e.g., data, status and control) through the MU interface. Most importantly, the MU provides the ability for one processor to signal the other processor using interrupts. A cluster's CPU can set control bits in the MU's control-register that generates an interrupt on the other cluster's CPU. Therefore, applications can avoid polling the state of the data register. From our perspective, the high-level logic is for the sender to write the message into the data register and generate the interrupt for the receiver. In turn, the receiver reads the 4-byte register content in the context of an Interrupt Service Routing (ISR) and forwards the data into the application. A user-level process that could have been waiting for the message to arrive could be awoken as well.

The MU provides an API that includes functions to initialize the module, send and receive messages, set MU status flags and interrupts and other miscellaneous functions. The distributed semaphore can be implemented via the MU API.

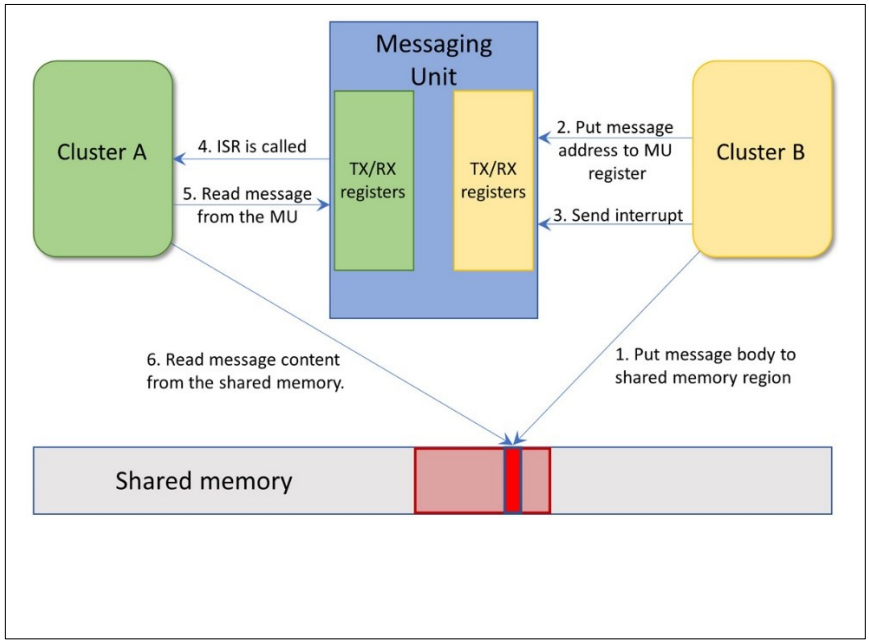
Another higher-level alternative to implementing the distributed semaphore is to use the Remote Processor Messaging protocol.

Remote Processor Messaging

The Remote Processor Messaging (RPMsg) is a software messaging bus that allows interprocess communications between different instances of operating systems, whether Linux or RTOS, running on AMP cores. The RPMsg is a part of the Open Asymmetric Multi Processing (OpenAMP) framework. RPMsg is present in the Linux kernel, integrated with FreeRTOS, Vector's MicroSAR and many other RTOS, and is also available as a stand-alone component for bare metal systems. The RPMsg protocol defines a *standardized binary interface* used to communicate between multiple cores in a heterogeneous multicore system. RTOS often integrate RPMsg-Lite, which is a lightweight implementation of the RPMsg protocol. RPMsg-Lite is an open-source component developed by NXP Semiconductors.

Compared to the RPMsg implementation, RPMsg-Lite offers code size reduction, a simpler API, and improved modularity.

The RPMsg API is relatively generalized across multiple hardware. In practice, it is implemented through similar specialized hardware: NXP implements the RPMsg protocol on top of the MU hardware while STM utilizes IPCC. Picture 6 illustrates the NXP implementation.



Picture 6

RPMsg Use Justification

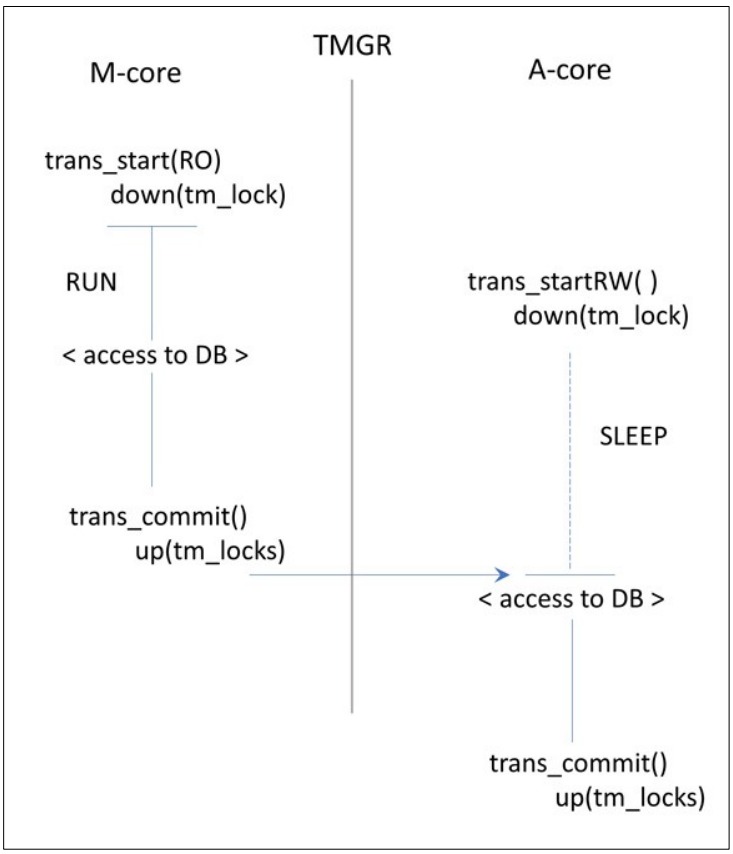
The MU provides enough functionally for the distributed semaphore implementation and so does the RPMsg bus. The advantage of using the MU directly are that it is a lower overhead solution: RPMsg always makes use of shared memory to pass messages around, regardless of their size. The distributed semaphore algorithm needs to pass just a few bytes and uses the MU registers for that. The advantage of using the RPMsg are that the MU-based implementation is hardware-dependent. MU register addressing and overall rules of engagement could be different even across NXP devices and other vendor's hardware could use different means to control messaging across the shared bus. Furthermore, there could be many software components that use the MU directly and hence it would be necessary to coordinate that utilization with those components. For example, the Linux-based RPMsg kernel module uses the MU, so if that module is loaded (used by some applications/drivers), it wouldn't be possible to use the MU directly.

As a result, we recommend implementation of the distributed semaphore via the RPMsg bus. Note that RPMsg implementations vary from one OS to another, but functionally allows message exchange across asymmetric hardware architectures through common DDR memory.

Transaction Flow and Distributed Semaphore

This section describes the implementation of the Distributed Semaphore for NXP's i.MX8 system on chip (SoC).

As described previously, we assume that applications access the shared data in the context of "transactions". For the sake of simplicity to demonstrate the concept, one "write" or multiple "read" transactions operate at a time. While a "read" transaction operates, a "write" transaction will wait for all the read transactions to complete. When a "write" transaction operates, all other transactions (read or write) will wait for the write transaction to complete. Picture 7 illustrates the transaction flow. In the implementation, each task that requires access to the shared data is represented as a structure that lets us put tasks on hold to avoid access conflicts. Specifically, there is a *semaphore* that allows its associated task/transaction to wait. When a task is ready to run, the implementation lifts the semaphore. The unique feature of this setup is the implementation of a distributed semaphore (as opposed to a "local" single OS-based primitive). For the i.MX hardware, the distributed semaphore is implemented so that a task located on the A-core is capable of "waking up" a transaction waiting on the M-core and vice versa.



Picture 7. The M-core starts a RO (read-only) transaction. There are no other transactions in the system, so the TM immediately allows the current transaction to run. While the transaction is running, the A-core attempts to initiate a "read_write" transaction. The transaction waits on the "semaphore".

The implementation on each side (A/M) has a local *imx_semaphore* structure identified by the semaphore ID that is kept in the shared memory. Thus, a distributed semaphore is represented by two structures and a few functions:

```
typedef struct {
   unsigned short      count;
   unsigned short      n_w_local;
   unsigned short      n_w_remote;
} imx_semaphore;
```

The *"count"* defines the number of resources that belong to a given side. This implementation makes use of binary semaphores. That means that, in the beginning, the sum of counts on each side of the semaphore is equal to 1.

The *"n_w_local"* represents the number of local tasks that wait for the semaphore.

The *"n_w_remote"* represents the number of remote tasks waiting for a semaphore (on the opposite side).

If the local count is zero, i.e., the semaphore can't be taken, the kernel sends the *WAIT_INC "*message" (semaphore increment) to the opposite side and the task is put to sleep. The task is awakened when the count becomes positive, either as a result of a local semaphore up() call, or a "reply" from the remote side. If the *WAIT_INC* was sent, the *WAIT_DEC* (semaphore decrement) message is sent.

```
void down(sem)
{
   int req_sent = 0;
   if (sem->count == 0) {
      sem->n_w_local++;
      req_sent = 1;
      rpmsg_send(RPMSG_OP_WAIT_INC);
   }
   while (se->count == 0) {
      task_wait()
   }
   se->count--;
   if (req_sent) {
      sem->n_w_local--;
      rpmsg_send(RPMSG_OP_WAIT_DEC);
   }
}
```

When the WAIT_INC is received the n_w_remote (the number of remote tasks waiting for access) is incremented. If at the same time the local count > 0, the ownership of the semaphore is transferred to the opposite side through the POST message.

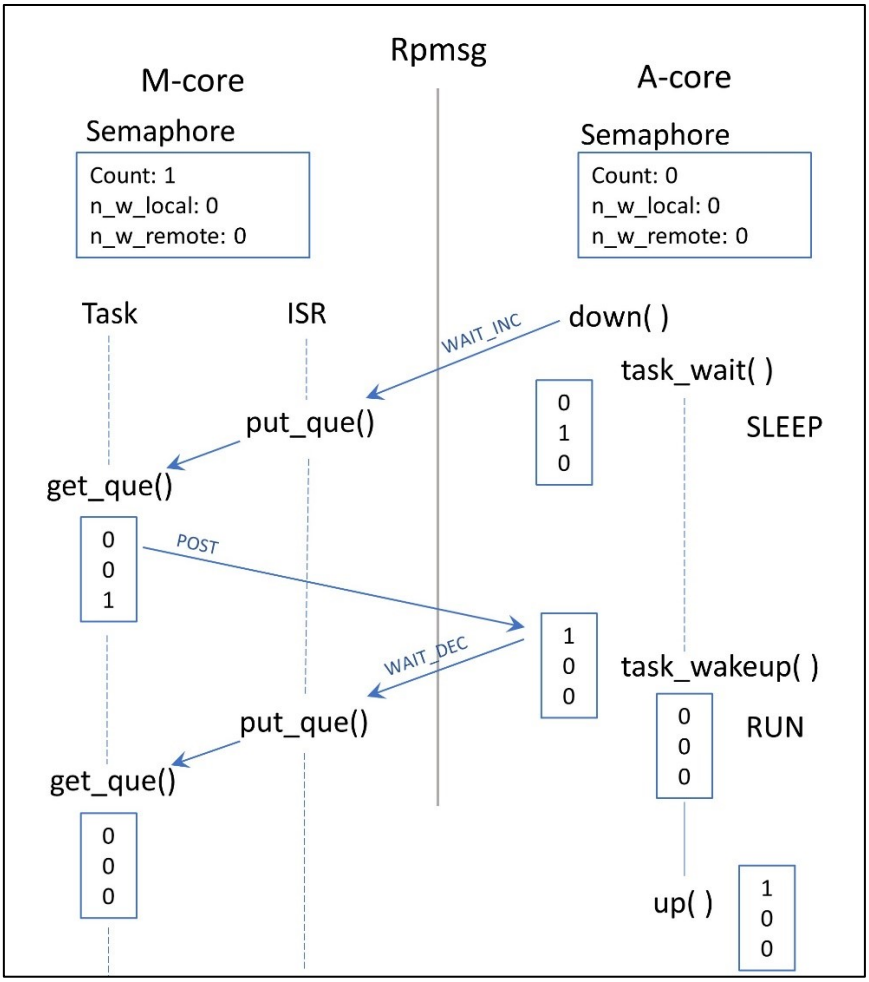
```
void on_WAIT_INC_msg ( sem )
{
   sem->n_w_remote++;
   if (sem->count > 0) {
      sem->count--;
      rpmsg_send(RPMSG_OP_POST);
   }
}


void on_WAIT_DEC_msg(int code, int sem_id)
{
   sem->n_w_remote--;
}
```

When the WAIT_DEC message is received, the number of waiting remote transactions is decremented by one.

```
/*..*/
void on_POST_msg(sem)
{
   if (sem->n_w_remote > 0 && sem->n_w_local == 0) {
      rpmsg_send(RPMSG_OP_POST);
   } else {
      sem->count++;
      if (sem->n_w_local > 0) {
         task_wakeup();
     }
   }
}

/*...*/
on_msg()
{
   while (1) {
       msg = get_queue();
       switch(OP(msg)) {
              case POST:   on_POST_msg(msg); break;
              case WAIT_INC:   on_WAIT_INC_msg(msg); break;
              case WAIT_DEC:   on_WAIT_DEC_msg(msg); break;

             ......
        }
   }
}
```

```
/*..*/
void up(sem)
{
   if (sem->n_w_remote > 0 && sem->n_w_local == 0) {
         rpmsg_send(RPMSG_OP_POST);
   } else {
      sem->count++;
      if (sem->n_w_local > 0) {
                  task_wakeup();
      }
   }
}
```

When a local task lifts the semaphore (calls up()) or receives the POST message from the opposite side, the ownership of the semaphore is transferred if there are no local tasks waiting in the queue and a request for the semaphore had been received. Otherwise, the local "count" is incremented and local tasks are awakened (if there are any). Picture 8 illustrates the flow:



Picture 8 illustrates the scenario when the A-core attempts to acquire a semaphore, but its local count is zero. The WAIT_INC is sent and the A-core task is put to sleep. M-core side's count is equal to 1, so upon receiving the WAIT_INC, the POST is immediately sent and the n_w_remote count is incremented. In turn, the A-side sends the WAIT_DEC and wakes up the waiting task. Since the WAIT_DEC was received, the M-core decremented the n_w_remote. When the A-core's task lifts the semaphore (calls up()), its count becomes 1 and, as a result the semaphore ownership is transferred from the M- to the A-core (until there are no other semaphore requests).

Conclusion

AMP systems and their ability to collect, share and process data in real time are the way forward for applications that need both latency-sensitive and high-performance processing. The hardware itself and the low-level apparatus create an excellent foundation for mission-critical applications, but real-time software — firmware, real-time operating systems and advanced middleware adds substantial value to any design. Real-time data management is a key component to use with AMP hardware for a number of data-driven applications: avionics, automotive, industrial, and virtually all domains that involve significant sensor data fusion.