OpenAMP Memory Types TI POV*

*** Well Bill's POV anyway**

Bill Mills

2017-09-22

Co-processor attached memory

- Memory tightly associated with a co-processor
- Examples:
  - TI PRU's each have their own Instruction and Data memories, a pair of PRU's also share a third data memory
  - TI C6x DSP's each have their own L1I, L1D, and L2 memories
    - These memories or only valid when associated cache is not configured to be 100% cache
  - TI ARM Cortex-M's often have dedicated memory
    - These processors are used for various roles including: chip control (suspend/resume helper, resource manager), multimedia control proc, or general user application processor
- Properties
  - The sizes of these memories are either fixed in the HW IP and hard coded in the rproc driver or vary from SOC to SOC and the rproc driver takes the size from DT data (or other boot config data)
  - These memories often appear at fixed location in the device's memory map (DA)
  - Multiple instances of each type often appear in an SOC, each instance has its own PA from the POV of the Cortex-A processor
  - These memories are good for program code & data as they naturally work with multiple instances and are fast for co-proc accesses
  - These memories are also good for vrings because they are fast to check

Co-processor private memory

- Memory associated with a co-processor that is not accessible to the master core
- (This is included mainly to give a name to this concept)
- Examples:
  - TI has no examples of this in HW but hypervisors and/or firewalls could create this situation
  - Two Linux OS'es under a hypervisor would be an example of this. Most of the memory of the two OS'es would not be accessible to the other
- Properties
  - Mostly this is out of scope for OpenAMP kernel & library components
  - These memories may be used for BSS or dynamic memory of firmware loaded by the rproc driver. BSS would be zero'ed by the firmware startup
  - This situation can arise when master core does not do the firmware loading
    - Ex: Firmware loading is done by hypervisor and this "master" is only doing communication
  - If master does require code and data to be loaded into these memories, it must be handled by loading a bootloader or monitor/kernel into accessible memory and then using communication facilities to pass data to this bootloader. *** This usage is out of scope for the in-kernel rproc framework and would have to be addressed by a user-space loader ***

Boot time associated DDR / SRAM

- General SOC memory that is then associated with a co-processor via boottime configuration
- Examples:
  - This is the most common memory type for all TI co-processors
  - Many co-processors get both a chunk of DRR and a chunk of on-chip SRAM
  - Some processor may need more than one chunk of each memory for different uses
    - Ex: A co-processor may need Secure DDR & non-secure DDR
- Properties
  - Carved out at OS boot time, need to reboot to resize
  - Memory may be completely reserved or given to CMA as a dedicated pool to allow OS use when co-processor is not active (true?)
  - Each chunk is a contiguous block
  - Alignment and size may have to meet specific platform constraints such as IOMMU page sizes
  - A specific PA range is defined
  - Co-processors w/o MMU or IOMMU in path can still use absolutely placed ELF segments because address is fixed

Dynamically allocated DDR / SRAM

- Sub Type by Allocation time
  - Allocated when firmware is loaded
    - Size of allocated memory is based on firmware image properties
  - Allocated at runtime under user space control
    - User space program does allocation (ex: ION) and then “gives” it to the rproc driver (DMABUF export / import)
- Sub Type by Page size:
  - Contiguous memory: memory is all in one PA range
    - Need to allocate via CMA
  - Paged memory: memory is made up of N OS native page size blocks (ex: 4K or 64K)
    - Standard page allocation, either kernel (kernel firmware load) or user space (other cases)
  - Huge page or Gigantic page memory: memory is N chunks of some higher order page size such as OS huge pages (2M) or even higher like 512M or 1G.
    - Mechanism TBD, dedicated CMA pool? hugeTLB-FS?
- 6 combinations of above
  - Runtime allocation is out of scope for the resource table
  - Non-contiguous memory w/o an IOMMU requires new resources types, can define these later
  - **Leaves contiguous memory and paged IOMMU memory allocated at firmware load**

Firmware associated contiguous DDR / SRAM

- Contiguous blocks of general SOC memory that is then associated with a co-processor at firmware load time
- Examples:
  - This model is desired when memory sizes for each co-processor is not known ahead of time and/or varies for each of N predefined use cases
- Properties
  - Memory is allocated from CMA or other special allocator based on sizes in firmware tables
  - Each chunk is a contiguous block so that it is easy to describe to the co-processor
  - Alignment and size may have to meet specific platform constraints such as IOMMU page sizes
  - Actual PA range will be unknown ahead of time
  - If Firmware needs to put program loaded code and data into these memory blocks, then one of the following is required:
    - Co-processor MMU
    - SOC IOMMU in the co-processor path
    - ELF relocations (assumed out of scope for in-kernel rproc loader)

Firmware associated Paged DDR [/ SRAM] via IOMMU

- General SOC memory that is then associated with a co-processor via boottime configuration
- Examples:
  - This model is desired because it allows a more flexible memory allocation.
  - If OS native pages are used then OS standard page allocation can be used
  - Even if large pages are used, an allocator that gives fixed size pages can reduce or eliminate allocation failures
- Properties
  - Memory is allocated from OS page allocator or special large page allocator
  - IOMMU is programmed to collect the various pages into a single contiguous DA range, so it is easy to describe to the co-processor
  - Master CPU MMU is programmed to collect the various pages into a single contiguous VA range for master
  - Alignment and size will need to meet platform IOMMU and OS page size constraints
  - Firmware can put program loaded code and data into these memory blocks as DA range is known

PRU specific problems

- PRU cores are true Harvard arch:
  - The first word of Instruction RAM is code address 0
  - The first word of Data RAM is data address 0
- PRU cores have small amounts of memory
  - AM572x: 12K code RAM, 8K data ram per core, 32K data shared between two cores
  - Other SOCs have smaller RAMs for PRUs
  - Don’t want to take up space in target memory for resource tables
  - Don’t want rpmsg proto in all firmware images but don’t want to preclude it where it makes sense

Cache Handling

Cache Handling

- In practice there are 3 memory map types we need to consider
  - Non-cached normal memory
    - The CPU does not cache memory accesses
  - Cache coherent
    - The CPU cache is used but access by other agents update or access the cache data or state to get the correct values
  - Cached with SW managed coherency
    - The CPU cache is used
    - Other agents cannot or will not consult the CPU cache
    - The CPU must perform cache maintenance operations on buffers as they pass between master and co-processor
    - (Detail on other slide)
- Barriers are still needed with all three types
  - Used to ensure ordering or completion

Cache Handling - out of scope types

- Other memory map types
  - Device or strongly ordered
    - Used for memory mapped registers and other HW
    - If used for memory, can be treated as non-cached normal memory for this discussion
    - Requirement for barriers is reduced or eliminated with the use of strongly ordered but this usage is discouraged in Linux
  - Write through
    - Won't be considered
    - Most SMP CPUs don't actually support write-through
    - Write through is only a partial solution for AMP
    - It allows two CPUs to write to two disjoint variables in the same cache line,
      however each CPU must still invalidate its cache to see the updates done by the other CPU

SW Managed Coherency -- detail

- SW & communication protocol must maintain a state for each buffer
  - Owned by Master
  - Owned by Co-processor for reading
  - Owned by Co-processor for writing
  - Owned by Co-processor for reading & writing
- Requirements
  - The SW must flush the cache lines to ensure its writes are visible to other agents
  - The SW must invalidate the cache lines to ensure writes by other agents are visible
  - The SW must ensure stale dirty lines are not evicted while the buffer is owned by the co-processor
  - The SW must ensure that any speculative pre-fetches done by the CPU before the buffer was passed back to the master processor are discarded
- Implementation
  - Many CPUs only implement flush & invalidate
  - Buffers passed to the co-processor need F&I to flush data (co-processor read or r&w) or to ensure no stale dirty line is evicted while buffer is owned by co-processor (co-processor write)
  - Buffers passed back to the master need F&I after buffer return to discard prefetched data
    - This step can be skipped if the buffer was passed for co-processor reading only
    - This step in practice should only invalidate; it should not generate any flushed data unless the SW has violated the buffer passing protocol
  - In the Linux kernel this is all handled by the DMA protocol

Cache Handling Combinations

- In some real world system the two sides use different cache mapping types. Here are a few notable combinations. (Not all of these can be done with two Cortex-A cores.)
- Side A cache coherent, Side B non-cached but IO coherent
  - This is sometimes called IO coherent and is the normal case in a coherent system when dealing with a device that does not have a cache such as most SATA controllers, etc
  - The accesses from side B will consult the side A cache
  - No extra work in SW is required w/ manual In practice there are 3 memory map types we need to consider
- Side A cache w/ SW managed coherency, Side B non-cached
  - This is sometimes is the normal case in a non-coherent system when dealing with a device that does not have a cache
  - Side B access to do not consult the side A cache
  - Side A must use the SW managed coherency protocol, Side B must follow the buffer ownership rules but does not need to do any cache operations
- Side A cache coherent, Side B cached with SW managed coherency
  - This is the case when a device with a cache is in a cache coherent system but does not participate in the coherency protocol
  - Side B access after the cache consult the side A cache, Side A accesses never consult the side B cache
  - Side A does not need to do cache ops but Side B does
  - The buffer ownership passing rules need to be used