OpenAMP Structured Buffer Exchange

(aka the Big Data problem)

Bill Mills

2018-03-19

Overview

- Structured Buffer Exchange is a framework to allow rpmsg messages to carry references to other buffers
  - Other buffers can be any size
  - Other buffers can be allocated by multiple means and may be contagious memory or normal paged memory
  - Other buffers can be normal cached memory even for non-coherent co-processors
- Memory areas to hold buffers are pre-negotiated between app and co-processor
  - normally at app start-up time but can be done more dynamically
- All work for the buffer exchange is described in a single kernel call
- Outline
  - Allocation and Sharing via ION & DMABUF
  - Structured Buffer Exchange (2 slides)
  - Multi-context model
  - Paged Allocation & Sharing via VFIO
  - Sub-allocation of device carve-out memory
  - Optional add on: SBE with “dumb” firmware

Allocation and Sharing via ION and DMABUF

- Use ION published info in sysfs and app config to determine ION heap to use
  - ION may have heaps for multiple CMA regions
  - ION may have secure buffer heaps
  - ION may have a heap of huge (2MB) or gigantic (1GB) pages
- Use ION device and ioctl API to allocate memory from selected heap
- Use ION device and ioctl API to export memory as a DMABUF
- Use ION to map memory to userspace (optional, not used for secure buffers)
- Open rp device
- Use rp ioctl API to import DMABUF
  - RP framework will track buffer, size and handle
  - RP framework may use IOMMU to map buffer for co-processor access
  - RP driver may send message to co-processor to allow it to map its MMU for access
- Memory Area can now be used in structured buffer exchange (described next)

Structured Buffer Exchange

- Enhance userspace rpmsg API to optionally support “SBE mode”
  - Could be an ioctl to set a mode and then use normal reads & writes or it could be new ioctls to do transfers
  - Any mode should be per end-point
- SBE API requests contain:
  - Bytes of actual message to send to co-processor
    - May be omitted to just do buffer sync operations
  - Request Flags: Blocking/non-blocking, others?
  - Optional Transaction ID
    - Allows blocking RPC mode
  - Meta data to describe buffers being given to co-processor
  - The meta data for each buffer includes:
    - Memory area handle (DMABUF filehandle or other token prearranged with remoteproc framework)
    - Offset & size within the Buffer handle
    - Direction of buffer ownership: IN (co-processor input), OUT (co-processor output), or IN/OUT (both in & out)
    - Q: should we distinguish input followed by output from SHARED where both parties read and write the buffer at the same time? (SHARED case is only possible with coherent buffers)

Structured Buffer Exchange - Operation

- SBE eliminates the need for address translation as all addresses are referenced by offsets within pre-arranged memory area IDs
- SBE allows cache synchronization (if needed) to be done in-line with the actual data transmission or reception
  - IN & IN/OUT buffers are flushed before TX
  - OUT and IN/OUT buffers are invalidated before use on RX
  - (See memtype presentation from 2017 for details)
- SBE can be used in Request / Response model
  - Supports blocking or non-blocking for the response
  - Supports Transaction ID for response mapping in either mode
  - OUT & IN/OUT buffers are described in Request and Response
    - Could match 1:1 between Request and Response or not depending on the specific message protocol
    - IN buffers are described only in response
  - Either side can be Requester per transaction
- SBE can be used in pure message mode
  - Each side gives IN buffers
  - Transaction ID of 0 reserved for non-transaction

Multi-context model - usage

- Multi-context model is required when multiple applications are using the same co-processor and sharing their data with the co-processor.
- Multi-context model allows one application to share memory areas with the co-processor and be assured that a different application will not be able to access them via malicious (or just erroneous) requests to the same co-processor
- If all code on the co-processor is trusted to the same degree as the Linux kernel, then Multi-context support can be implemented using just the Linux kernel
- If untrusted code will be used on the co-processor (OpenCL etc) then co-processor should have an MPU or MMU and do its own enforcement
- Multi-context model will be described with a privileged daemon to create a new context but other models may work as well

Multi-context model - operation

- Application requests daemon to create a new context
- If the co-processor supports runtime multi-context,
  - Daemon does message exchange with co-processor kernel/supervisor to create a new context and negotiate a context ID
- Daemon creates new rpmsg handle, locks it to contextID and passes it to application
- Every Memory Area exchange is done with a contextID
  - 0 is the default global context for the that co-processor when multi-context support is not used
- Linux kernel ensures that buffers used in SBE are associated with the correct context
- If co-processor supports runtime multi-context,
  - its kernel will setup local MMU to ensure each context only has access to the memory areas associated with the context of the SBE request

Allocation and Sharing via VFIO (fuzzy)

- Use normal application allocation (malloc) or hugeTLBfs to allocate memory to your userspace app
- Open rp device
- Use rp VFIO framework to map portions of your userspace for use by the remote processor
  - VFIO puts user space in charge of allocating addresses for mapping
  - Q: How does userspace know how to pick areas that will not interfere with RP firmware etc
  - Q: How do we associate different areas with a handle or ID for structured buffer exchange?
- RP Framework actions
  - RP framework will mapped areas via the VFIO framework
  - RP framework may use IOMMU to map buffer for co-processor access
  - RP driver may send message to co-processor to allow it to map its MMU for access
- Buffer can now be used in structured buffer exchange (described later)

Sub-Allocation of Device Memories and Carve-outs (TBD)

- Slide not complete
- Do we wish to have a method to allocate buffers in the memories associated with the rproc?
- What method? Do we describe the areas available in the resource table?
- What allocation logic is supported? Do we add DMABUF export support to rproc?

Structured Buffer Exchange with “dumb” firmware (TBD)

- Slide not complete
- In slides above, both sides of the conversation know they are involved in a SBE transaction or message
- We could optionally add support for firmware that does not know about SBE
- Such a system would include “fixup” meta data in the request
  - The fixup data would patch the outgoing message to use the actual DA (16/32/64 bit) of the buffer
  - The fixup data would also describe how to match the corresponding response
- Is this useful?

Peer to Peer, IPC only, & Lifecycle Management

Overview

- Independent Master & Slave
- True Peer to Peer
- IPC only
- Lifecycle management topics collection

Independent & Resilient Master/Slave

- Master and Slave come up in any order
- Master or Slave can die and restart at any time and other side syncs back up
- Master does not need to load firmware if not required (IPC only)
- Linux should be capable to be either Master or Slave
- Master
  - Allocates vrings and buffers
  - TX on VRING0, RX on VRING1, Pulls from USED, Pushes to AVIAL
- Slave
  - TX on VRING1, RX on VRING0, Pulls from AVAIL, Pushes to USED
- OK to assume tokenized “kick” and notify
  - Many implementations use a mailbox for kick (TX Data available) and notify (RX Data available)
  - Mailbox can deliver a 32 bit (or 64 bit) token to indicate multiple events
  - Special events can be used to say
    - I restarted
    - I am ready to communicate
    - I crashed or am shutting down

True Peer to Peer

- No need to declare one side Master?
- Completely symmetrical?
- Each Peer
  - Allocates its TX VRING and buffers, and signals ready
  - Waits for peer to allocate its VRING and buffers and signal ready
  - On TX Pulls from USED and pushes to AVAIL
  - On RX Pulls from AVAIL and pushes to USED
- Still need to designate
  - Where VRINGs are located (via firmware or DT or mailbox message)
  - Who will use “master” side of Mailbox
- Possible? (Probably) Needed? (Maybe?) Will be accepted by VirtIO maintainers? (Maybe??)
- Notes
  - Not compatible with existing VirtIO protocol and Virt Device drivers (VirtBlk, VirtNet)
  - If Firmware loading is to be done, asymmetry likely exists anyway

IPC only

- No firmware loading
- Still need some sort of “null-remoteproc”
  - Memory attach, IOMMU handling, carve-outs for vrings, mailbox (kick & IRQ)
- Perhaps we can make one SOC independent null-remoteproc based on mailbox framework
  - Might serve 80% of the cases

Life Cycle Management topics (Brain Storming)

- HW Resources
  - Basics: Memory, IRQ, Clocks, Power Domains
  - Devices: Timers, UART, GPIO, SPI, I2C
  - Infrastructure: IOMMU StreamIDs, DMA Channels, Bus BW
- Co-processor life cycle
  - Suspend / Resume, requested by Host or Co-processor
  - Co-processor frequency control
  - Exit / Shutdown, requested by Host or Co-processor
  - Crash, dump, recovery
  - Restart: declared by co-processor or initiated by host
- Run time control
  - Create / Destroy of context for multi-context model
  - Create / Destroy of new VirtIO channel for rpmsg, or direct virtIO device