o2iblnd memory mapping

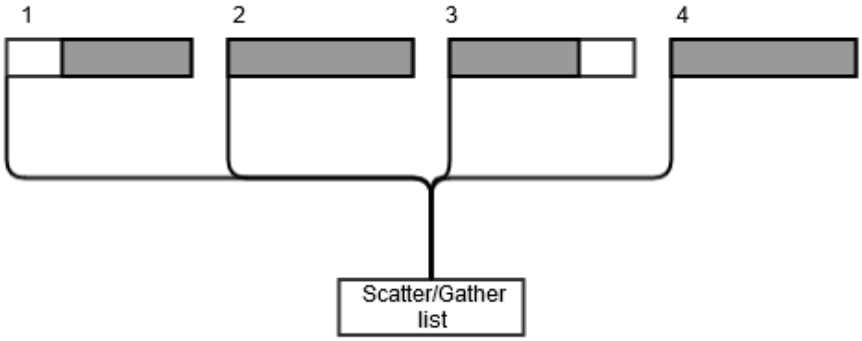
LU-9983 Problem Statement

Changes in OSP sent non-contiguous buffers to the o2iblnd. The assumption in o2iblnd was that for FMR and FastReg (FRMR) it only accepts an offset for the first fragment. Each fragment has a maximum size of PAGE_SIZE (4K on x86_64 systems). However it supported it for global memory regions. For global memory regions each Fragment is described to the peer in the kib_rdma_desc_t structure. However, for FMR and FRMR o2iblnd only used 1 fragment and relied on FMR/FRMR to understand the rest of the fragments. FMR does not support non-contiguous data (as far as I know). FRMR supports gaps in the buffers to be RDMAed only if IB_MR_TYPE_SG_GAPS is used when allocating memory regions. However, Cray testing showed that using this option introduced a memory drop.

Originally, LNet didn't do any sanity checking on the buffers before it attempted to RDMA them, therefore when using FMR/FRMR not all the data would be RDMAd properly and we'd end-up with strange corrupted data.

This resulted in several fixes which introduced some major behavior changes to the o2iblnd:

- LU-9983 ko2iblnd: allow for discontiguous fragments
- LU-10089 o2iblnd: use IB_MR_TYPE_SG_GAPS
- LU-10129 lnd: set device capabilities
- LU-10129 lnd: rework map_on_demand behavior
- LU-10394 lnd: default to using MEM_REG



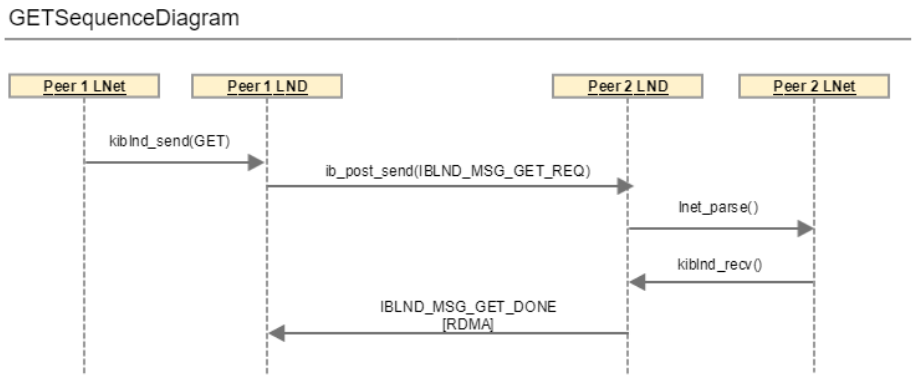
The above diagram represents the issue at play. Page 3 is not the last page and it has a gap. When this gets RDMAed using FMR/FRMR, then some data is lost from page 4 (need to confirm)

Overview

To understand the above fixes we need to have an understanding of how memory is mapped by the o2iblnd.

MD memory passed down to the LND in IOV or KIOV needs to be mapped in order for it to be DMAed.

Let's look at the GET case for IOV with FMR:



- kiblnd_send() is called for LNET_MSG_GET
  - tx = kiblnd_get_idle_tx(ni, target.nid);
  - ibmsg = txtx_msg; # ibmsg is of type kib_msg_t. That's the structure that goes over the wire

        - rd = &ibmsgibm_u.get.ibgm_rd; # rd is of type kib_rdma_desc_t, which also gets sent to the peer. This describes the memory in which the data will be RDMAd in
        - rc = kiblnd_setup_rd_iov(...); # this maps the iov to make it DMAable
            - go through the iov and collect all the pages (struct page) into a scatter gather list.
                - for each page there could be a page_offset.
                - The data does not necessarily fill the entire page. This is recorded by: sg_set_page(sg, page, fragnob, page_offset);
            - Once we have collected all the pages in a scatter gather list, we need to map the transmit by calling: kiblnd_map_tx(ni, tx, rd, sg - txtx_frags);
            - kiblnd_map_tx() maps the pages given the scatter gather list into the device's dma addresses.
                - For good documentation take a look at: linux/Documentation/DMA-API-HOWTO.txt
                - kiblnd_dma_map_sg() is called to map the pages gathered in the scatter/gather list to the device's DMA addresses.
                    - The dma address is the page physical address + the sg_offset stored. Look at the implementation of sg_phys()
                - When this function completes then we call sg_dma_address() and sg_dma_length() macros to return the dma_address and length.
                - These values are stored in the rd (kib_rdma_desc_t)
                - If we are using FMR or FastReg, determined by if (net->ibn_fmr_ps != NULL), then call kiblnd_fmr_map_tx()
                - kiblnd_fmr_map_tx()
                    - kiblnd_fmr_pool_map() is called. The main work here is done when ib_fmr_pool_map_phys() is called. This basically maps the pages where the data will be written into the FMR pools.
                    - The rd that gets sent over to the peer, in the FMR case, will need to describe a relative addresses within the memory region, in order for the remote to know where to RDMA the data.
- on the receiving peer we call kiblnd_handle_rx() for IBLND_MSG_GET_REQ. This will call up in lnet_parse, which will match the MD pointing to the data to be RDMAed to the peer
    - kiblnd_reply() is called, which calls kiblnd_setup_rd_iov(). Same logic is done is described above.
    - kilbnd_init_rdma() is then called to set up the work requests (wr) which is use in the call to ib_post_send()
        - Key point in this function is that it sets the sge (txtx_sge[]) to have the address for each fragment. Look at kiblnd_rd_frag_addr()
        - The sge addr will have a relative address to the local source MD. That's used in ib_post_send() to RDMA the data into the buffers which are also described via relative addresses by the destination rd.

One difference between FMR and FastReg is that in FMR the pages are mapped to the FMR pool, and therefore the rd will need to be set to relative addresses. In the FastReg case, wrwr.wr.fast_reg.page_list, points to the physical pages, and therefore the source rd, must describe the actual page addresses, and not just a relative offset.

Fast Registration

Peer 1 sends the RDMA descriptor (rd) with only 1 fragment and the dma_address of the memory. Peer 2 receives the rd and sets up its own memory to RDMA by mapping the memory (for fast reg case). Peer2 does an RDMA write given the remote address. It appears for the fast reg this remote address needs to be the actual dma address on peer1. A possible explanation is that peer 1 when it maps the memory to dma data in, it uses the starting memory and from there it knows how to place the data. So when peer 2 sends it just needs to give it the start dma address

Resources

Slide deck - Verbs Overview