Fast-Reg Memory Registration in Mellanox OFED 3.3

By Austin Pohlmann

I Introduction

Mellanox's RDMA drivers provide both user level and kernel level APIs for users to develop RDMA applications. However, the kernel level API is severely lacking in documentation, mostly due to all the changes between versions. The kernel API provides a few methods of registering memory for use with rdma transfers, and the most recent drivers (4.0+) have changed the way users do this. This paper is to serve as a reference to anyone who wishes to use version 3.3 (and possibly 3.4) of Mellanox's OFED drivers. All of this is the way I perceived the usage of the fast reg memory system, and could possibly be incorrect.

The fast reg system provides a fast and asynchronous way to prepare and register memory regions for use with rdma operations. Most of this information has been inferred after looking through the source code for the drivers and reading through the appropriate header files. However, with all the assumptions I made, I was able to successfully able to register and use memory with the methods described here. All bolded words are ones that will most likely show up when trying to use this registration.

II Parameters

The functions referenced are in the **ib_verbs.h** file in directory where the driver was installed to. The use must first allocate the **ib_fast_reg_page_list** structure via a call to

ib_alloc_fast_reg_page_list(struct ib_device *device, int page_list_len)

The device can be found using a previously allocated protection domain as follows:

pd->device

The **page_list_length** should be set to the number of pages covered by a contiguous buffer that was previously allocated with **kmalloc**. The returned structure will have the following members:

struct ib_device	***device;**
u64	***page_list;**
unsigned int	**max_page_list_len;**

max_page_list_len is the length of the array **page_list**, which will be at LEAST the size specified upon allocation. The page list is to be populated by the DMA address that are returned from the **ib_dma_*** functions. Also, the page list must contain the address of the PAGES. A simple way to do this is to map the entire **kmalloc**'ed buffer using **ib_dma_map_single**, which will give you the DMA address of the buffer. With this address, you can loop through the page list buffer size/page size times, setting each entry to the sum of the previous entry and the page size.

NOTE: the original DMA address might not be on a boundry, so be sure logical AND the DMA address with the **PAGE_MASK** macro supplied by the kernel. This will effectively set any offset bits to zero.

Alongside allocating the page list, you must also allocate the memory region to be associated with the page list. A simple call to

ib_alloc_fast_reg_mr(struct ib_pd *pd, int max_page_list_len)

will return the **ib_mr** structure needed to proceed. Note that the **max_page_list_length** should be the value that was stored in the page list structure earlier.

Before continuing, be sure that both sides are connected. On the server, this is accomplished after calling **rdma_accept**, and on the client, this is accomplished after the event handler processes the **RDMA_CM_EVENT_ESTABLISHED** event.

The final step before the buffer is registered is the **fast_reg** work request. The **ib_send_wr** structure's union, **wr**, contains the **fast_reg** portion that we are concerned with.

iova_start

This value is the start of your buffer, and should be set to the DMA address returned from **ib_dma_map_single**

page_list

This is the same **page_list** found in the structure we allocated earlier.

page_shift

This is set to the **PAGE_SHIFT** macro provided by the kernel. Its value is the number of bits that are used as the offset in the address you provided in the page list.

page_list_length

This is the value that YOU specified to the first function call mentioned earlier, not **max_page_list_length**. This is how many of the entries in **page_list** that you populated.

length

This is the length of the buffer that starts at the address **iova_start**.

access_flags

The access rights for the memory being registered. These are the same flags used when registering a DMA mr.

rkey

This is the **rkey** contained in the **mr** structure allocated earlier

After the work request has been filled out, a call to **ib_post_send** will asynchronously complete the registration (don't forget to set the **send_flags** to **IB_SEND_SIGNALED** in the **ib_send_wr**). When it is complete, the completion queue will be notified and the completion's opcode will be **IB_WC_FAST_REG_MR**. When the currently running code tries to use the registered memory for the first time, it should sleep if the request has not yet been completed. The buffer should not be touched until after the work is completed. When the memory is no longer needed, please free everything in the following order:

ib_dma_unmap_single
ib_dereg_mr
ib_free_fast_reg_page_list
kfree

Final notes: if you wish to use the memory without having to reregister it, please look into the **ib_update_fast_reg_key** function and the **invalidate_rkey** value in **ib_send_wr**. You must then post another **fast_reg** work request with the new **rkey** to notify the rdma device of the change.

III Example

The following code is a functioning example using the Mellanox OFED 3.3 drivers on centos 6.9 with the 3.10.87 kernel. The structs and global variables used in the code are shown in the beginning. This function would take place after a connection has already been established, and relies on the completion handler function (defined when creating the completion queue) to sleep until conditions are met. The fast reg region is created, and the relevant information for using it is sent to the remote side. What is not shown is the deregistration process, as it happens outside of the function. The error markers at the end can be examined in order to see what the deregistration would look like in case of an error at any point of the process

```
enum crdma_state {
      WAITING,    // state is set to WAITING before going to sleep
      CONNECT_REQUEST,  // appears when a client requests to connect to a server
      CONNECTED,  // appears when a client and server successfully connect
      ROUTE_RESOLVED,   // client has successfully resolved the route to server
      ADDR_RESOLVED,    // client has resolved the host's address
      FRMR_COMPLETE,    // the fast reg mr has been successfully registered
      RDMA_WRITE_COMPLETE,    // These a used when a completion for the
      RDMA_SEND_COMPLETE,     // relevant operations have been processed
      RDMA_RECV_COMPLETE,
      RDMA_READ_COMPLETE,
      DISCONNECT, // the remote side has disconnected
      ERROR // an error has occured
} state;
struct crdma_cb {
      u8 addr[4]; //holds the address of the server (ipv4)
      uint16_t port;    // the port that is being used to communicate
      struct rdma_cm_id *cmid, *child;    // the cm ids, child is used for the
client's connection on the server's side and cmid is for self reference
      struct ib_cq *cq; // the completion queue
      struct ib_pd *pd; // the protection domain
      struct ib_qp *qp; // the queue pair
      struct ib_mr *mr; // the mr that I used before setting up the fast reg mr
      struct ib_mr *frmr;     // the mr used for fast reg
      struct ib_fast_reg_page_list *frpl; // the page list for fast reg
      struct ib_send_wr send_wr;    // the wr used for any type of send
      struct ib_send_wr fast_wr;    // unused, I used send_wr instead of this
      struct ib_recv_wr recv_wr;    // the wr used for any type of receive
      struct ib_send_wr *bad_send;  // supplied as an argument to ib_post_send
      struct ib_recv_wr *bad_recv;  // supplied to ib_post_recv
      struct ib_sge test_sge[2];    // 0 is recieve, 1 is send
      void *buffs[3];   // 0 is recieve, 1 is send, 2 is the fast reg region
```

```
	u64 dma[3]; // array of dma addresses
	uint32_t remote_rkey;	// the rkey for the remote mr
	uint64_t remote_addr;	// the remote dma address of the mr
	uint32_t remote_len;	// the length of the remote mr
	wait_queue_head_t wqueue;	// used for sleeping
};
static int crdma_fr(struct crdma_cb *cb){
	int ret=0,i;	// return variable and for loop variable
	// allocate the page list structure
	cb->frpl = ib_alloc_fast_reg_page_list(cb->pd->device, 8);
	pr_info("Page list: %p\nMax page list length: %u\n", cb->frpl, cb->frpl->max_page_list_len);
	// see if we got a valid address. If not, there was an error
	if(IS_ERR(cb->frpl)){
		pr_err("Failed to allocate the page list!!!\n");
		return PTR_ERR(cb->frpl);
	}
	// Allocate the fast reg mr and see if we got a valid address
	cb->frmr = ib_alloc_fast_reg_mr(cb->pd,
				cb->frpl->max_page_list_len);
	if (IS_ERR(cb->frmr)) {
		pr_err("fast_reg_mr failed\n");
		ret = PTR_ERR(cb->frmr);
		goto error0;
	}
	pr_info("Fast_reg rkey: %lu\n", (long unsigned)cb->frmr->rkey);
	// get the memory we want to register and map it to a dma address
	cb->buffs[2] = kmalloc(8*4096, GFP_KERNEL);
	cb->dma[2] = ib_dma_map_single(cb->pd->device, cb->buffs[2], 8*4096, DMA_BIDIRECTIONAL);
	// make sure everything mapped okay
	if(ib_dma_mapping_error(cb->pd->device, cb->dma[2])){
		pr_err("Error mapping fast reg buffer\n");
```

```
            ret = 1;

            goto error1;

      }

      // fill the page list by removing the offset values from every page in
our buffer

      for(i=0; i<7; i++){

            cb->frpl->page_list[i] = (cb->dma[2] + i*4096) & PAGE_MASK;

      }

      pr_info("Page mask: %llx\n", PAGE_MASK);

      // fill out the fast reg work request

      cb->send_wr.opcode = IB_WR_FAST_REG_MR;

      cb->send_wr.num_sge = 0;

      cb->send_wr.sg_list =NULL;

      cb->send_wr.next = NULL;

      cb->send_wr.send_flags = IB_SEND_SIGNALED;

      cb->send_wr.wr.fast_reg.access_flags =

            IB_ACCESS_LOCAL_WRITE        |        IB_ACCESS_REMOTE_READ        |
IB_ACCESS_REMOTE_WRITE;

      cb->send_wr.wr.fast_reg.page_list = cb->frpl;

      cb->send_wr.wr.fast_reg.rkey = cb->frmr->rkey;

      cb->send_wr.wr.fast_reg.page_shift = PAGE_SHIFT;

      cb->send_wr.wr.fast_reg.page_list_len = 8;

      cb->send_wr.wr.fast_reg.iova_start=cb->dma[2];

      cb->send_wr.wr.fast_reg.length = 8*4096;

      // sleep until the work request finishes/fails

      state = WAITING;

      if(ib_post_send(cb->qp, &cb->send_wr, &cb->bad_send)){

            pr_err("Failed to post work request to send queue!!!\n");

            goto error2;

      }

      if(wait_event_interruptible(cb->wqueue, state >=FRMR_COMPLETE)){

            pr_info("Interrupted\n");

      }
```

```
	pr_info("State after wakikng: %u\n", state);
	if(state >= DISCONNECT)
		goto error2;
	state = WAITING;
	// state was set to waiting for our upcoming sleep
	// fill out the work requests
	cb->test_sge[0].lkey = cb->frmr->lkey;
	cb->test_sge[1].lkey = cb->frmr->lkey;
	cb->test_sge[0].addr = cb->dma[2];
	cb->test_sge[1].addr = cb->dma[2] + 4096;
	cb->test_sge[0].length = 4*1024;
	cb->test_sge[1].length = 4*1024;
	// I mess with the recv here because I used it AFTER this function. It
can be ignored
	cb->recv_wr.next = NULL;
	cb->recv_wr.num_sge = 1;
	cb->recv_wr.sg_list = cb->test_sge;
	// copy the dma addr and rkey to be send to the remote side
	memcpy(cb->buffs[2] + 4096, &cb->dma[2], sizeof(cb->dma[2]));
	memcpy(cb->buffs[2]    +    4096+sizeof(cb->dma[2]),    &cb->frmr->rkey,
sizeof(cb->frmr->rkey));
	pr_info("Address: %llx\n",(long long unsigned)cb->dma[2]);
	cb->send_wr.next = NULL;
	cb->send_wr.sg_list = &cb->test_sge[1];
	cb->send_wr.num_sge = 1;
	cb->send_wr.opcode = IB_WR_SEND;
	cb->send_wr.send_flags = IB_SEND_SIGNALED;
	if(ib_post_send(cb->qp, &cb->send_wr, &cb->bad_send)){
		pr_err("Failed to post work request to send queue!!!\n");
		goto error2;
	}
	if(wait_event_interruptible(cb->wqueue, state >=RDMA_SEND_COMPLETE)){
		pr_info("Interrupted\n");
```

```
	}
	return 0;
error2:
	ib_dma_unmap_single(cb->pd->device,		cb->dma[2],		8*4*1024,
DMA_BIDIRECTIONAL);
error1:
	ib_dereg_mr(cb->frmr);
error0:
	ib_free_fast_reg_page_list(cb->frpl);

	return ret;

}
```