Proceedings of the
Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team

Introduction to the InfiniBand Core Software

Bob Woodruff
Intel Corporation
robert.j.woodruff@intel.com

Sean Hefty
Intel Corporation
sean.hefty@intel.com

Roland Dreier
Topspin Communications
roland@topspin.com

Hal Rosenstock
Voltaire Corporation
halr@coltaire.com

Abstract

InfiniBand support was added to the kernel in 2.6.11. In this paper, we describe the various modules and interfaces of the InfiniBand core software and provide examples of how and when to use them. The core software consists of the host channel adapter (HCA) driver and a mid-layer that abstracts the InfiniBand device implementation specifics and presents a consistent interface to upper level protocols, such as IP over IB, sockets direct protocol, and the InfiniBand storage protocols. The InfiniBand core software is logically grouped into 5 major areas: HCA resource management, memory management, connection management, work request and completion event processing, and subnet administration. Physically, the core software is currently contained within 6 kernel modules. These include the Mellanox HCA driver, ib_mthca.ko, the core verbs module, ib_core.ko, the connection manager, ib_cm.ko, and the subnet administration support modules, ib_sa.ko, ib_mad.ko, ib_umad.ko. We will also discuss the additional modules that are under development to export the core software interfaces to userspace and allow safe direct access to InfiniBand hardware from userspace.

1 Introduction

This paper describes the core software components of the InfiniBand software that was included in the linux 2.6.11 kernel. The reader is referred to the architectural diagram and foils in the slide set that was provided as part of the paper's presentation at the Ottawa Linux Symposium. It is also assumed that the reader has read at least chapters 3, 10, and 11 of InfiniBand Architecture Specification [IBTA] and is familiar with the concepts and terminology of the InfiniBand Architecture. The goal of the paper is not to educate people on the InfiniBand Architecture, but rather to introduce the reader to the APIs and code that implements the InfiniBand Architecture support in Linux. Note that the InfiniBand code that is in the kernel has been written to comply with the InfiniBand 1.1 specification with some 1.2 extensions, but it is important to note that the code is not yet completely 1.2 compliant.

The InfiniBand code is located in the kernel tree under `linux-2.6.11/drivers/infiniband`. The reader is encouraged to read the code and header files in the kernel tree. Several pieces of the InfiniBand stack that are in the kernel contain good examples of how to use the routines of the core software described in this paper. Another good source of information can be found at the www.openib.org website. This is where the code is developed prior to being submitted to the linux kernel mailing list (lkml) for kernel inclusion. There are several frequently asked question documents plus email lists `<openib-general@openib.org>`. where people can ask questions or submit patches to the InfinBand code.

The remainder of the paper provides a high level overview of the mid-layer routines and provides some examples of their usage. It is targeted at someone that might want to write a kernel module that uses the mid-layer or someone interested in how it is used. The paper is divided into several sections that cover driver initialization and exit, resource management, memory management, subnet administration from the viewpoint of an upper level protocol developer, connection management, and work request and completion event processing. Finally, the paper will present a section on the user-mode infrastructure and how one can safely use the InfiniBand resource directly from userspace applications.

2 Driver initialization and exit

Before using InfiniBand resources, kernel clients must register with the mid-layer. This also provides the way, via callbacks, for the client to discover the available InfiniBand devices that are present in the system. To register with the InfiniBand mid-layer, a client calls the `ib_register_client` routine. The routine takes as a parameter a pointer to a `ib_client` structure, as defined in `linux-2.6.11/drivers/infiniband/include/ib_verbs.h`. The structure takes a pointer to the client's name, plus two function pointers to callback routines that are invoked when an InfiniBand device is added or removed from the system. Below is some sample code that shows how this routine is called:

```
static void my_add_device(
   struct ib_device *device);

static void my_remove_device(
   struct ib_device *device);

static struct ib_client my_client = {
 .name   = "my_name",
 .add    = my_add_device,
 .remove = my_remove_device
};
static int __init my_init(void)
{
 int ret;

 ret = ib_register_client(
       &my_client);
    if (ret)
       printk(KERN_ERR
"my ib_register_client failed\n");
 return ret;
}
static void __exit my_cleanup(void)
{
   ib_unregister_client(
             &my_client);
}
module_init(my_init);
module_exit(my_cleanup);
```

3 InfiniBand resource management

3.1 Miscellaneous Query functions

The mid-layer provides routines that allow a client to query or modify information about the various InfiniBand resources.

```
 ib_query_device
 ib_query_port
 ib_query_gid
 ib_query_pkey
```

```
ib_modify_device
ib_modify_port
```

The `ib_query_device` routine allows a client to retrieve attributes for a given hardware device. The returned `device_attr` structure contains device specific capabilities and limitations, such as the maximum sizes for queue pairs, completion queues, scatter gather entries, etc., and is used when configuring queue pairs and establishing connections.

The `ib_query_port` routine returns information that is needed by the client, such as the state of the port (Active or not), the local identifier (LID) assigned to the port by the subnet manager, the Maximum Transfer Unit (MTU), the LID of the subnet manager, needed for sending SA queries, the partition table length, and the maximum message size.

The `ib_query_pkey` routine allows the client to retrieve the partition keys for a port. Typically, the subnet manager only sets one pkey for the entire subnet, which is the default pkey.

The `ib_modify_device` and `ib_modify_port` routines allow some of the device or port attributes to be modified. Most ULPs do not need to modify any of the port or device attributes. One exception to this would be the communication manager, which sets a bit in the port capabilities mask to indicate the presence of a CM.

Additional query and modify routines are discussed in later sections when a particular resource, such as queue pairs or completion queues, are discussed.

3.2 Protection Domains

Protection domains are a first level of access control provided by InfiniBand. Protection domains are allocated by the client and associated with subsequent InfiniBand resources, such as queue pairs, or memory regions.

Protection domains allow a client to associate multiple resources, such as queue pairs and memory regions, within a domain of trust. The client can then grant access rights for sending/receiving data within the protection domain to others that are on the Infinband fabric.

To allocate a protection domain, clients call the `ib_alloc_pd` routine. The routine takes and pointer to the device structure that was returned when the driver was called back after registering with the mid-layer. For example:

```
my_pd = ib_alloc_pd(device);
```

Once a PD has been allocated, it is used in subsequent calls to allocate other resources, such as creating address handles or queue pairs.

To free a protection domain, the client calls `ib_dealloc_pd`, which is normally only done at driver unload time after all of the other resources associated with the PD have been freed.

```
ib_dealloc_pd(my_pd);
```

3.3 Types of communication in InfiniBand

Several types of communication between end-points are defined by the InfiniBand architecture specification [IBTA]. These include reliable-connected, unreliable-connected, reliable-datagram, and unreliable datagrams. Most clients today only use either unreliable datagrams or reliable connected communications. An analogy in the IP network stack would be that unreliable datagrams are analogous to UDP packets, while a reliable-connected queue pairs provide a connection-oriented type of communication, similar to TCP. But InfiniBand communication is packet-based, rather than stream oriented.

3.4 Address handles

When a client wants to communicate via unreliable datagrams, the client needs to create an address handle that contains the information needed to send packets.

To create an address handle the client calls the routine `ib_create_ah()`. An example code fragment is shown below:

```
struct ib_ah_attr   ah_attr;
struct ib_ah        *remote_ah;

memset(&ah_attr, 0, sizeof ah_attr);
ah_attr.dlid     = remote_lid;
ah_attr.sl       = service_level;
ah_attr.port_num = port->port_num;

remote_ah = ib_create_ah(pd, &ah_attr);
```

In the above example, the pd is the protection domain, the `remote_lid` and `service_level` are obtained from an SA path record query, and the `port_num` was returned in the device structure through the `ib_register_client` callback. Another way to get the `remote_lid` and `service_level` information is from a packet that was received from a remote node.

There are also core verb APIs for destroying the address handles and for retrieving and modifying the address handle attributes.

```
ib_destroy_ah
ib_query_ah
ib_modify_ah
```

Some example code that calls `ib_create_ah` to create an address handle for a multicast group can be found in the IPoIB network driver for InfiniBand, and is located in `linux-2.6.11/drivers/infiniband/ulp/ipoib`.

3.5 Queue Pairs and Completion Queue Allocation

All data communicated over InfiniBand is done via queue pairs. Queue pairs (QPs) contain a send queue, for sending outbound messages and requesting RDMA and atomic operations, and a receive queue for receiving incoming messages or immediate data. Furthermore, completion queues (CQs) must be allocated and associated with a queue pair, and are used to receive completion notifications and events.

Queue pairs and completion queues are allocated by calling the `ib_create_qp` and `ib_create_cq` routines, respectively.

The following sample code allocates separate completion queues to handle send and receive completions, and then allocates a queue pair associated with the two CQs.

```
send_cq = ib_create_cq(device,
          my_cq_event_handler,
          NULL,
          my_context,
          my_send_cq_size);
recv_cq = ib_create_cq(device,
          my_cq_event_handler,
          NULL,
          my_context,
          my_recv_cq_size);

init_attr->cap.max_send_wr  = send_cq_size;
init_attr->cap.max_recv_wr  = recv_cq_size;
init_attr->cap.max_send_sge = LIMIT_SG_SEND;
init_attr->cap.max_recv_sge = LIMIT_SG_RECV;

init_attr->send_cq          = send_cq;
init_attr->recv_cq          = recv_cq;
init_attr->sq_sig_type      = IB_SIGNAL_REQ_WR;
init_attr->qp_type          = IB_QPT_RC;
init_attr->event_handler = my_qp_event_handler;

my_qp = ib_create_qp(pd, init_attr);
```

After a queue pair is created, it can be connected to a remote QP to establish a connection. This is done using the QP modify routine and the communication manager helper functions described in a later section.

There are also mid-layer routines that allow destruction and release of QPs and CQs, along with the routines to query and modify the queue pair attributes and states. These additional core QP and CQ support routines are as follows:

```
ib_modify_qp
ib_query_qp
ib_destroy_qp
ib_destroy_cq
ib_resize_cq
```

Note that ib_resize_cq is not currently implemented in the mthca driver.

An example of kernel code that allocates QPs and CQs for reliable-connected style of communication is the SDP driver [SDP]. It can be found in the subversion tree at openib.org, and will be submitted for kernel inclusion at some point in the near future.

4 InfiniBand memory management

Before a client can transfer data across InfiniBand, it needs to register the corresponding memory buffer with the InfiniBand HCA. The InfiniBand mid-layer assumes that the kernel or ULP has already pinned the pages and has translated the virtual address to a Linux DMA address, i.e., a bus address that can be used by the HCA. For example, the driver could call get_user_pages and then dma_map_sg to get the DMA address.

Memory registration can be done in a couple of different ways. For operations that do not have a scatter/gather list of pages, there is a memory region that can be used that has all of physical memory pre-registered. This can be thought of as getting access to the "Reserved L_key" that is defined in the InfiniBand verbs extensions [IBTA].

To get the memory region structure that has the keys that are needed for data transfers, the client calls the ib_get_dma_mr routine, for example:

```
mr = ib_get_dma_mr(my_pd,
     IB_ACCESS_LOCAL_WRITE);
```

If the client has a list of pages that are not physically contiguous but want to be virtually contiguous with respect to the DMA operation, i.e., scatter/gather, the client can call the ib_reg_phys_mr routine. For example,

```
*iova = &my_buffer1;

buffer_list[0].addr = dma_addr_buffer1;
buffer_list[0].size = buffer1_size;
buffer_list[1].addr = dma_addr_buffer2;
buffer_list[1].size = buffer2_size;

mr = ib_reg_phys_mr(my_pd,
     buffer_list,
     2,
     IB_ACCESS_LOCAL_WRITE |
     IB_ACCESS_REMOTE_READ |
     IB_ACCESS_REMOTE_WRITE,
     iova);
```

The mr structure that is returned contains the necessary local and remote keys, lkey and rkey, needed for sending/receiving messages and performing RDMA operations. For example, the combination of the returned iova and the rkey are used by a remote node for RDMA operations.

Once a client has completed all data transfers to a memory region, e.g., the DMA is completed, the client can release to the resources back to the HCA using the ib_dereg_mr routine, for example:

```
ib_dereg_mr(mr);
```

There is also a verb, ib_rereg_phys_mr that allows the client to modify the attributes of a given memory region. This is similar to doing a de-register followed by a re-register but where possible the HCA reuses the same resources rather than deallocating and then reallocating new ones.

```
status = ib_rereg_phys_mr(mr,
         mr_rereg_mask,
         my_pd,
         buffer_list,
         num_phys_buf,
         mr_access_flags,
        iova_start);
```

There is also a set of routines that allow a technique called fast memory registration. Fast Memory Registration, or FMR, was implemented to allow the re-use of memory regions and to reduce the overhead involved in registration and deregistration with the HCAs. Using the technique of FMR, the client typically allocates a pool of FMRs during initialization. Then when it needs to register memory with the HCA, the client calls a routine that maps the pages using one of the pre-allocated FMRs. Once the DMA is complete, the client can unmap the pages from the FMR and recycle the memory region and use it for another DMA operation. The following routines are used to allocate, map, unmap, and deallocate FMRs.

```
ib_alloc_fmr
ib_unmap_fmr
ib_map_phys_fmr
ib_dealloc_fmr
```

An example of coding using FMRs can be found in the SDP [SDP] driver available at `openib.org`.

NOTE: These FMRs are a Mellanox specific implementation and are NOT the same as the FMRs as defined by the 1.2 InfiniBand verbs extensions [IBTA]. The FMRs that are implemented are based on the Mellanox FMRs that predate the 1.2 specification and so the developers deviated slightly from the InfiniBand specification in this area.

InfiniBand also has the concept of memory windows [IBTA]. Memory windows are a way to bind a set of virtual addresses and attributes to a memory regions by posting an operation to a send queue. It was thought that people might want this dynamic binding/unbinding intermixed with their work request flow. However, it is currently not used, primarily because of poor H/W performance in the existing HCA, and thus is not implemented in the mthca driver in Linux.

However, there are APIs defined in the midlayer for memory windows for when it is implemented in mthca or some future HCA driver. These are as follows:

```
ib_alloc_mw
ib_dealloc_mw
```

5 InfiniBand subnet administration

Communication with subnet administration(SA) is often needed to obtain information for establishing communication or setting up multicast groups. This is accomplished by sending management datagram (MAD) packets to the SA through InfiniBand special QP 1 [IBTA]. The low level routines that are needed to send/receive MADs along with the critical data structures are defined in `linux-2.6.11/drivers/infiniband/include/ib_mad.h`.

Several helper functions have been implemented for obtaining path record information or joining multicast groups. These relieve most clients from having to understand the low level MAD routines. Subnet administration APIs and data structures are located in linux-2.6.11/drivers/infiniband/include/ib_sa.h and the following sections discuss their usage.

5.1 Path Record Queries

To establish connections, certain information is needed, such as the source/destination LIDs, service level, MTU, etc. This information is found in a data structure known as a path record, which contains all relevant information of a path between a source and destination. Path records are managed by the InfiniBand subnet administrator(SA). To obtain a path record, the client can use the helper function:

```
 ib_sa_path_rec_get
```

This function takes the device structure, returned by the register routine callback, the local InfiniBand port to use for the query, a timeout value, which is the time to wait before giving up on the query, and two masks, comp_mask and gfp_mask. The comp_mask specifies the components of the ib_sa_path_rec to perform the query with. The gfp_mask is the mask used for internal memory allocations, e.g., the ones passed to kmalloc, GFP_KERNEL, GFP_USER, GFP_ATOMIC, GFP_USER. The **query parameter is a returned identifier of the query that can be used to cancel it, if needed. For example, given a source and destination InfiniBand global identifier (sgid/dgid) and the partition key, here is an example query call taken from the SDP [SDP] code.

```
query_id = ib_sa_path_rec_get(
     info->ca,
     info->port,
     &info->path,
     (IB_SA_PATH_REC_DGID |
      IB_SA_PATH_REC_SGID |
      IB_SA_PATH_REC_PKEY |
    IB_SA_PATH_REC_NUMB_PATH),
      info->sa_time,
      GFP_KERNEL,
      sdp_link_path_rec_done,
      info,
      &info->query);

if (result < 0) {
    sdp_dbg_warn(NULL,
"Error <%d> restarting path query",
               result);
}
```

In the above example, when the query completes, or times-out, the client is called back at the provided callback routine, sdp_link_path_rec_done. If the query succeeds, the path record(s) information requested is returned along with the context value that was provided with the query.

If the query times out, the client can retry the request by calling the routine again.

Note that in the above example, the caller must provide the DGID, SGID, and PKEY in the info->path structure, In the SDP example, the info->path.dgid, info->path.sgid, and info->path.pkey are set in the SDP routine do_link_path_lookup.

5.2 Cancelling SA Queries

If the client wishes to cancel an SA query, the client uses the returned **query parameter and query function return value (query id), e.g.,

```
 ib_sa_cancel_query(
     query_id,
     query);
```

5.3 Multicast Groups

Multicast groups are administered by the subnet administrator/subnet manager, which configure InfiniBand switches for the multicast group. To participate in a multicast group, a client sends a message to the subnet administrator to join the group. The APIs used to do this are shown below:

```
ib_sa_mcmember_rec_set
ib_sa_mcmember_rec_delete
ib_sa_mcmember_rec_query
```

The `ib_sa_mcmember_rec_set` routine is used to create and/or join the multicast group and the `ib_sa_mcmember_rec_delete` routine is used to leave a multicast group. The `ib_sa_mcmember_rec_query` routine can be called get information on available multicast groups. After joining the multicast group, the client must attach a queue pair to the group to allow sending and receiving multicast messages. Attaching/detaching queue pairs from multicast groups can be done using the API shown below:

```
ib_attach_mcast
ib_detach_mcast
```

The gid and lid in these routines are the multicast gid(mgid) and multicast lid (mlid) of the group. An example of using the multicast routines can be found in the IP over IB code located in `linux-2.6.11/drivers/infiniband/ulp/ipoib`.

5.4 MAD routines

Most upper level protocols do not need to send and receive InfiniBand management datagrams (MADs) directly. For the few operations that require communication with the subnet manager/subnet administrator, such as path record queries or joining multicast groups, helper functions are provided, as discussed in an earlier section.

However, for some modules of the mid-layer itself, such as the communications manager, or for developers wanting to implement management agents using the InfiniBand special queue pairs, MADs may need to be sent and received directly. An example might be someone that wanted to tunnel IPMI [IPMI] or SNMP [SNMP] over InfiniBand for remote server management. Another example is handling some vendor-specific MADs that are implemented by a specific InfiniBand vendor. The MAD routines are defined in `linux-2.6.11/drivers/infiniband/include/ib_mad.h`.

Before being allowed to send or receive MADs, MAD layer clients must register an agent with the MAD layer using the following routines. The `ib_register_mad_snoop` routine can be used to snoop MADs, which is useful for debugging.

```
ib_register_mad_agent
ib_unregister_mad_agent
ib_register_mad_snoop
```

After registering with the MAD layer, the MAD client sends and receives MADs using the following routines.

```
ib_post_send_mad
ib_coalesce_recv_mad
ib_free_recv_mad
ib_cancel_mad
ib_redirect_mad_qp
ib_process_mad_wc
```

The `ib_post_send_mad` routine allows the client to queue a MAD to be sent. After a MAD is received, it is given to a client through their receive handler specified when registering. When a client is done processing an incoming MAD, it frees the MAD buffer by calling `ib_free_recv_mad`. As one would expect, the `ib_cancel_mad` routine is used to cancel an outstanding MAD request.

`ib_coalesce_recv_mad` is a place-holder routine related to the handling of MAD segmentation and reassembly. It will copy received MAD segments into a single data buffer, and will be implemented once the InfiniBand reliable-multi-packet-protocol (RMPP) support is added.

Similarly, the routine `ib_redirect_mad_qp` and the routine `ib_process_mad_wc` are place holders for supporting QP redirection, but are not currently implemented. QP redirection permits a management agent to send and receive MADs on a QP other than the GSI QP (QP 1). As an example, a protocol which was data intensive could use QP redirection to send and receive management datagrams on their own QP, avoiding contention with other users of the GSI QP, such as connection management or SA queries. In this case, the client can re-redirect a particular InfiniBand management class to a dedicated QP using the `ib_redirect_mad_qp` routine. The `ib_process_mad_wc` routine would then be used to complete or continue processing a previously started MAD request on the redirected QP.

6 InfiniBand connection management

The mid-layer provides several helper functions to assist with establishing connections. These are defined in the header file, `linux-2.6.11/drivers/infiniband/include/ib_cm.h` Before initiating a connection request, the client must first register a callback function with the mid-layer for connection events.

```
ib_create_cm_id
ib_destroy_cm_id
```

The `ib_create_id` routine creates a communication id and registers a callback handler for connection events. The `ib_destroy_cm_id` routine can be used to free the communication id and de-register the communication callback routine after the client is finished using their connections.

The communication manager implements a client/server style of connection establishment, using a three-way handshake between the client and server. To establish a connection, the server side listens for incoming connection requests. Clients connect to this server by sending a connection request. After receiving the connection request, the server will send a connection response or reject message back to the client. A client completes the connection setup by sending a ready to use (RTU) message back to the server. The following routines are used to accomplish this:

```
ib_cm_listen
ib_send_cm_req
ib_send_cm_rep
ib_send_cm_rtu
ib_send_cm_rej
ib_send_cm_mra
ib_cm_establish
```

The communication manager is responsible for retrying and timing out connection requests. Clients receiving a connection request may require more time to respond to a request than the timeout used by the sending client. For example, a client tries to connect to a server that provides access to disk storage array. The server may require several seconds to ready the drives before responding to the client. To prevent the client from timing out its connection request, the server would use the `ib_send_cm_mra` routine to send a message received acknowledged (MRA) to notify the client that the request was received and that a longer timeout is necessary.

After a client sends the RTU message, it can begin transferring data on the connection. However, since CM messages are unreliable, the RTU may be delayed or lost. In such cases, receiving a message on the connection notifies the server that the connection has been established. In order for the CM to properly track the connection state, the server calls `ib_cm_establish` to notify the CM that the connection is now active.

Once a client is finished with a connection, it can disconnect using the disconnect request routine (`ib_send_cm_dreq`) shown below. The recipient of a disconnect request sends a disconnect reply.

```
ib_send_cm_dreq
ib_send_cm_drep
```

There are two routines that support path migration to an alternate path. These are:

```
ib_send_cm_lap
ib_send_cm_apr
```

The `ib_send_cm_lap` routine is used to request that an alternate path be loaded. The `ib_send_cm_apr` routine sends a response to the alternative path request, indicating if the alternate path was accepted.

6.1 Service ID Queries

InfiniBand provides a mechanism to allow services to register their existence with the subnet administrator. Other nodes can then query the subnet administrator to locate other nodes that have this service and get information needed to communicate with the other nodes. For example, clients can discover if a node contains a specific UD service. Given the service ID, the client can discover the QP number and QKey of the service on the remote node. This can then be used to send datagrams to the remote service. The communication manager provides the following routines to assist in service ID resolution.

```
ib_send_cm_sidr_req
ib_send_cm_sidr_rep
```

7 InfiniBand work request and completion event processing

Once a client has created QPs and CQs, registered memory, and established a connection or set up the QP for receiving datagrams, it can transfer data using the work request APIs. To send messages, perform RDMA reads or writes, or perform atomic operations, a client posts send work request elements (WQE) to the send queue of the queue pair. The format of the WQEs along with other critical data structures are located in `linux-2.6.11/drivers/infiniband/include/ib_verbs.h`. To allow data to be received, the client must first post receive WQEs to the receive queue of the QP.

```
ib_post_send
ib_post_recv
```

The post routines allow the client to post a list of WQEs that are linked via a linked list. If the format of WQE is bad and the post routine detects the error at post time, the post routines return a pointer to the bad WQE.

To process completions, a client typically sets up a completion callback handler when the completion queue (CQ) is created. The client can then call `ib_req_notify_cq` to request a notification callback on a given CQ. The `ib_req_ncomp_notif` routine allows the completion to be delivered after n WQEs have completed, rather than receiving a callback after a single one.

```
ib_req_notify_cq
ib_req_ncomp_notif
```

The mid-layer also provides routines for polling for completions and peeking to see how many completions are currently pending on the completion queue. These are:

```
ib_poll_cq
ib_peek_cq
```

Finally, there is the possibility that the client might receive an asynchronous event from the InfiniBand device. This happens for certain types of errors or ports coming online or going offline. Readers should refer to section 11.6.3 of the InfiniBand specification [IBTA] for a list of possible asynchronous event types. The mid-layer provides the following routines to register for asynchronous events.

```
ib_register_event_handler
ib_unregister_event_handler
```

8 Userspace InfiniBand Access

The InfiniBand architecture is designed so that multiple userspace processes can share a single InfiniBand adapter at the same time, with each the process using a private context so that fast path operation can access the adapter hardware directly without requiring the overhead of a system call or a copy between kernel space and userspace.

Work is currently underway to add this support to the Linux InfiniBand stack. A kernel module `ib_uverbs.ko` implements character special devices that are used for control path operations, such as allocating userspace contexts and pinning userspace memory as well as creating InfiniBand resources such as queue pairs and completion queues. On the userspace side, a library called libibverbs will provide an API in userspace similar to the kernel API described above.

In addition to adding support for accessing the verbs from userspace, a kernel module (`ib_umad.ko`) allows access to MAD services from userspace.

There also now exists a kernel module to proxy CM services into userspace. The kernel module is called `ib_ucm.ko`.

As the userspace infrastructure is still under construction, it has not yet been incorporated into the main kernel tree, but it is expected to be submitted to lkml in the near future. People that want get early access to the code can download it from the InfiniBand subversion development tree available from `openib.org`.

9 Acknowledgments

We would like to acknowledge the United States Department of Energy for their funding of InfiniBand open source work and for their computing resources used to host the `openib.org` web site and subversion data base. We would also like to acknowledge the DOE for their work in scale-up testing of the InfiniBand code using their large clusters.

We would also like to acknowledge all of the companies of the `openib.org` alliance that have applied resources to the `openib.org` InfiniBand open source project.

Finally we would like the acknowledge the help of all of the individuals in the Linux community that have submitted patches, provided code reviews, and helped with testing to ensure the InfiniBand code is stable.

10 Availability

The latest stable release of the InfiniBand code is available in the Linux releases (starting in 2.6.11) available from `kernel.org`.

```
ftp://kernel.org/pub
```

For those that want to track the latest InfiniBand development tree, it is located in a subversion database at openib.org.

```
        svn checkout
https://openib.org/svn/gen2
```

References

[IBTA] The InfiniBand Architecture Specification, Vol. 1, Release 1.2
`http://www.ibta.org`

[SDP] The SDP driver, developed by Libor Michalek
`http://www.openib.org`

[IPMI] Intelligent Platform Management Interface
`http://developer.intel.com`

[SNMP] Simple Network Management Protocol
`http://www.ietf.org`

[LJ] May 2005 Linux Journal—InfiniBand and Linux
`http://www.linuxjournal.com`