RZ 3894 (# ZUR1064-106) 05/03/2016
Computer Science 12 pages

Research Report

USIW: Design and Implementation of Userspace Software iWARP using DPDK

P.I. MacArthur

IBM Research – Zurich
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.
Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Africa • Almaden • Austin • Australia • Brazil • China • Haifa • India • Ireland • Tokyo • Watson • Zurich

USIW: Design and Implementation of Userspace Software iWARP using DPDK

Patrick MacArthur
IBM Research

April 29, 2016

Abstract

Decreasing access latencies of non-volatile memory (NVM) demonstrate the need for a more efficient software storage stack. Remote Direct Memory Access (RDMA) is a useful lightweight mechanism for accessing NVM, as demonstrated by FlashNet. FlashNet is a kernel space solution which integrates the softiwarp RDMA endpoint and SALSA software Flash controller to allow local and remote endpoints to access storage using RDMA. In this study, we use the Data Plane Development Kit (DPDK) to develop a userspace RDMA endpoint called USIW (Userspace SoftiWARP), to be the first component of a userspace analogue to FlashNet. We examine the performance of USIW compared to the kernel softiwarp using standard verbs-level microbenchmarks.

1 Introduction

As new forms of non-volatile memory (NVM) emerge, interest in providing more efficient remote access to this storage also emerges. Existing protocols, such as iSCSI [1], Fibre Channel [6–9], and SRP [12], are heavyweight, requiring connection management and setup for each data transfer. These protocols also assume block access, where data can only be read or written in fixed-size blocks. However, modern NVM is expected to become byte-addressable, making these protocols suboptimal. Thus, new storage protocols are needed. Remote Direct Memory Access (RDMA) provides a base set of semantics that is useful for accessing byte-addressable NVM. FlashNet [23] implements a lightweight protocol to access storage using RDMA.

FlashNet is implemented in the kernel. There is currently no known implementation of fully userspace remote storage access using RDMA. The Data Plane Development Kit (DPDK) [3] leverages I/O virtualization support in the Linux kernel to provide direct NIC access from userspace, in addition to a userspace packet processing framework. Using DPDK, we present USIW, a userspace RDMA implementation using iWARP over UDP [18]. This provides the first part of a userspace analogue to FlashNet.

USIW has the following goals:

- Data transfer performed entirely in userspace using DPDK. A kernel driver is acceptable where required by the verbs interface but must not be involved in the data transfer path.
- Support the standard OpenFabrics verbs interface, which is used by most existing RDMA applications.
- Support out-of-order RDMA READ and RDMA WRITE operations and completions, to improve responsiveness when accessing remote persistent memory.
- Perform at least as well as softiwarp.

1.1 RDMA

Remote Direct Memory Access (RDMA) allows a process to access regions of the virtual memory space of a remote process. This allows data to be transferred in a zero-copy manner, avoiding the intermediate copies done by the kernel in traditional networking stacks. Additionally, the data transfers themselves are done

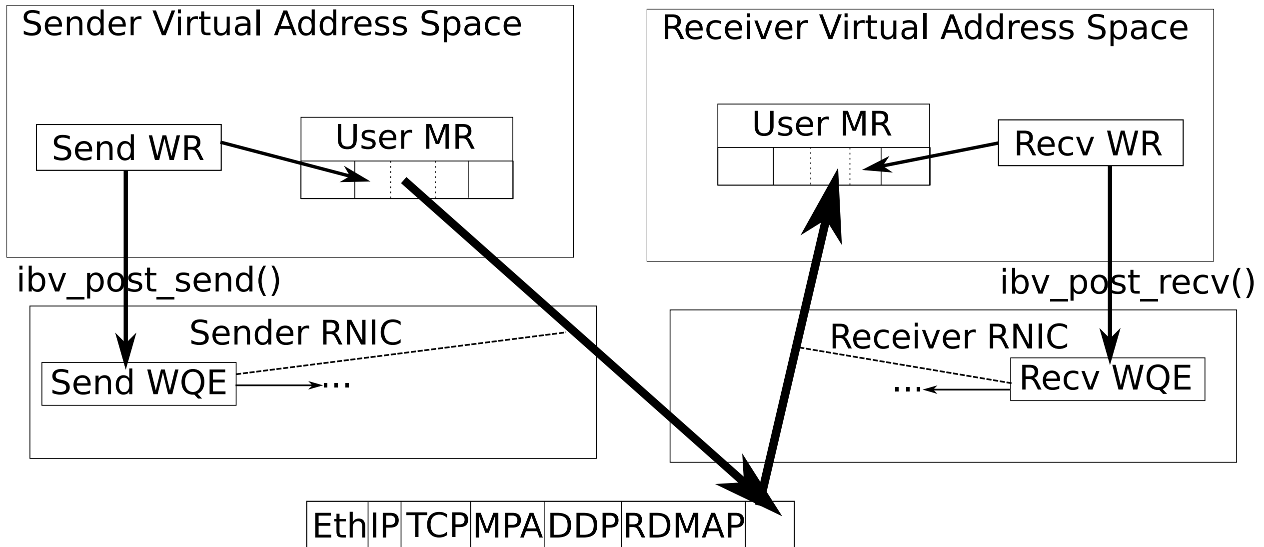


Figure 1: A diagram illustrating the transfer of 1 MTU worth of data using iWARP.

directly from userspace without trapping into the kernel, avoiding the system call and context switching overhead. This allows for higher throughput and lower latency and CPU utilization than is possible with traditional networks.

There are three dominant RDMA technologies today: InfiniBand [12], iWARP [4, 21, 22], and RoCE [10, 11]. The InfiniBand specification includes a set of transfer operations called verbs, described not as an API but as a set of abstract interfaces. The OpenFabrics Alliance has produced an API that provides these operations, also colloquially referred to as verbs, which is used by most RDMA devices today that implement the aforementioned technologies. The primary component is the libibverbs userspace library, which forwards verbs calls to HCA-specific userspace drivers as well as the uverbs Linux kernel module, which provides a mechanism for userspace to communicate with the kernel verbs driver and HCA-specific drivers. Similarly, the librdmacm library provides an interface to the RDMA connection management kernel modules.

Data transfers in RDMA must be done via registered memory regions. Registering a memory region performs two steps. First, it populates the IOMMU on the HCA with the virtual to physical address translations. Second, it pins the virtual pages into physical memory so that the mapping does not change during a transfer operation. This makes memory registration orders of magnitude more expensive than the actual data transfer itself.

We briefly discuss iWARP data transfer, as USIW is built using iWARP. iWARP consists of three protocols on top of TCP: (i) MPA, which provides a framing mechanism to send and receive entire discrete messages over TCP's stream semantics, (ii) DDP, which enables direct placement of incoming data segments into application virtual memory using addressing information placed into the DDP header, and (iii) RDMAP, which provides a set of standard RDMA operations using DDP. The three basic RDMA operations are (i) SEND, which provides a channel semantic in which each message is directly placed into buffers enqueued at the receiver, in order, (ii) RDMA WRITE, which places data directly into a specific virtual memory address at the receiver, and (iii) RDMA READ, in which the responder places data directly from a specific virtual memory address at the sender into the buffer specified by the receiver.

Figure 1 shows the components involved in a basic iWARP data transfer. First, the user application at the receiver posts a work request to the RNIC's receive queue, containing the virtual address at which the next incoming message should be placed. This work request is converted to an internal representation called a Work Queue Element (WQE) and placed onto the queue. Next, the user application at the sender posts a work request to its RNIC's send queue, containing the virtual address of the message to send. The work request is converted into a send WQE. For each segment of the message, the RNIC directly places that portion of the data from the application virtual memory onto the wire, along with the necessary iWARP, TCP/IP, and Ethernet headers. When the RNIC at the receiver receives each segment, it locates the corresponding

receive WQE and directly places the data into the associated memory region.

SoftiWARP [16] is a software-only implementation of iWARP as a Linux kernel module built atop kernel sockets along with a userspace verbs driver. This means that SoftiWARP cannot perform zero-copy receives, but it can be used as a testing engine or as a peer to a hardware iWARP implementation.

1.2 DPDK

The Data Plane Development Kit (DPDK) [3] is a software development kit produced by Intel that allows direct userspace access to standard NICs in Linux and FreeBSD. In Linux, it uses the UIO or VFIO kernel modules to map the device I/O memory and interrupts into userspace. Once mapped into userspace and appropriately configured, an application can transfer packets to/from the NIC in bursts using various poll-mode operations. This allows for high message throughput by minimizing the amount of communication between the NIC and the user application. Several projects, including MICA [14], have used DPDK to great effect.

DPDK makes several large contributions to allow high message throughput. First, it uses poll mode operations: the user application polls for new messages from the NIC rather than waiting for an interrupt. Second, its poll mode operations as well as data structures are designed for bulk operation, including hash tables and ring queues. This reduces contention between cores and communication between the CPU cores and the NIC by removing or placing as many packets as possible in a single operation. DPDK is designed from the ground up to work on non-uniform memory access (NUMA) systems by binding resources and data structures to specific CPU sockets and memory banks. On each CPU core, DPDK creates a single thread called a “logical core”, abbreviated as “lcore”, and provides functionality for launching tasks on each lcore. This makes it easy for an application to make the most efficient use of a NUMA system. Finally, DPDK supports zero-copy transfers by exposing memory buffer pools to the application. Packets are transmitted directly from these memory pools, and the NIC will directly place packets into a memory pool that is bound to each receive queue.

DPDK also supports NIC hardware filtering, including simple n-tuple and ethertype filters as well as Intel’s Flow Director [2]. Hardware filtering allows packets to be directed to different hardware receive queues based on fields within the packet, including source and destination Ethernet [13], IPv4 [19], and UDP addresses as well as arbitrary fields within the packet. Filtering can be used in hash mode to direct the packets to different cores for load balancing, as used by MICA [14], or in perfect match mode to avoid the overhead of classifying packets in software.

DPDK does not address every problem, however. In particular, although DPDK can perform zero-copy transfers for an application specifically written to use it, DPDK has no native support for RDMA memory semantics, as the NIC will only place into the next available memory buffer in its assigned pool. Additionally, although the memory pools are placed into contiguous memory, the packet buffers are interleaved with metadata. This means that an application cannot receive a contiguous message larger than the maximum transmission unit (MTU) of the NIC, without leveraging TCP-specific hardware offload support. Additionally, although DPDK provides some NUMA-aware data structures, these are mostly the data structures used by DPDK itself, and thus some useful data structures such as trees are missing and must be provided by the application.

2 Implementation

We present USIW, a userspace implementation of a subset of the iWARP protocol over UDP. A kernel mode driver is required only for early initialization of libibverbs, and for connection management. The kernel driver is based on the softiwarp code with all of the data transfer code removed. We use DPDK 2.2.0 and Linux 3.17.8.

We chose to base our protocol on iWARP rather than RoCE because RoCE would require implementing the state from the InfiniBand RC protocol, which we would like to avoid. By placing our protocol over UDP, we avoid having to maintain the InfiniBand RC or TCP [20] state machine, at the cost of having to implement reliability ourselves.

We fit each DDP segment in a single UDP datagram, to simplify the implementation, and we explicitly do not support IP fragmentation. This means that we can take advantage of the fact that UDP is message-

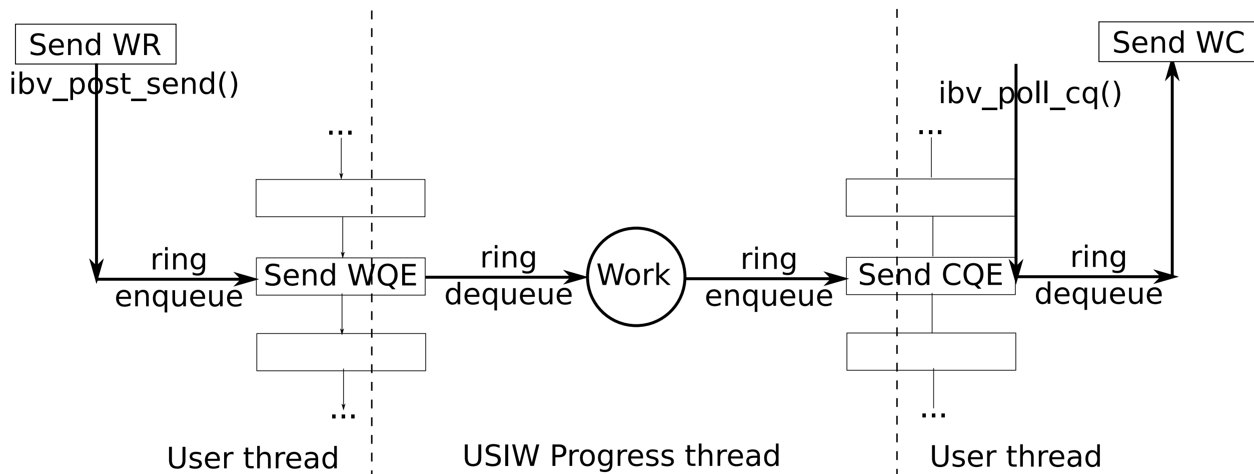


Figure 2: A diagram illustrating work request processing in USIW, including the interaction of user threads with USIW's progress thread via DPDK software ring queues.

oriented and we do not need to support markers, unlike a TCP-based iWARP implementation in which DDP segments may not exactly fall on TCP segment boundaries.

2.1 Work Request Processing

Upon initialization, USIW uses a DPDK logical core to create a background thread which performs all interaction with the NIC and KNI[1]. This is referred to as the *progress thread* through the remainder of this report. The progress thread is required since packets from KNI or the NIC, as well as events from KNI, may arrive while the user application is busy, but we need to react to these packets and events in a timely manner. While the use of multiple threads causes a performance penalty in RDMA applications due to the high cost of synchronization, we mitigate this by only allowing communication with the NIC from the progress thread, and limiting inter-thread communication as much as possible.

The work request processing is illustrated in Figure 2. In order to post a work request, a user thread calling ibv_post_send() will place a WQE into a ring queue. The progress thread eventually dequeues this WQE from the ring queue and places it into a private data structure. Once the WQE is completed, this process is mirrored—the progress thread fills in an empty CQE and places it into a ring queue after freeing the corresponding WQE for reuse. When a user thread eventually calls ibv_poll_cq(), it removes the appropriate number of entries from the ring queue and copies the contents into the work completion array allocated by the user. In this way, the amount of inter-thread communication required for work request processing is minimized.

Each send queue Work Queue Entry (WQE) can be in one of four states: INIT, TRANSFER, WAIT, COMPLETE. Each WQE proceeds through the states in the order given above. When a user posts a work request using ibv_post_send(), the WQE is created in the INIT state and placed onto the pending queue. When the progress thread dequeues the WQE from the pending queue, it is moved to the TRANSFER queue. During this state, each segment of the message is sent in turn. When the last segment is sent, the WQE transitions to the WAIT state, in which the WQE waits for the last segment to be acknowledged [2] Once the last segment has been acknowledged, the WQE transitions to the COMPLETE state. In this final state, once there are no preceding WQEs that have not yet completed, the progress thread removes the WQE from the active set and places a completion queue entry (CQE) onto the completion queue.

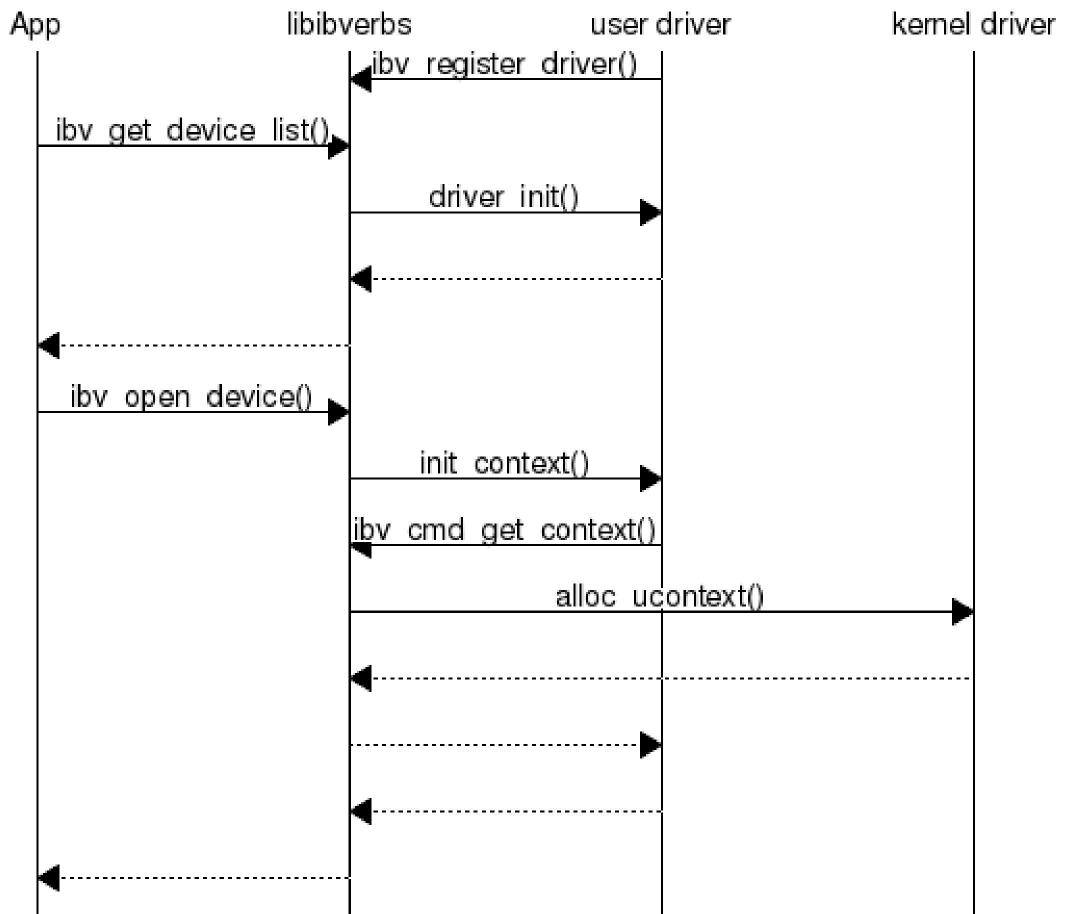


Figure 3: The initialization process for a libibverbs application.

2.2 Kernel Module

For most verbs devices, the device context is created in kernel space and registered with the kernel ib_core module as soon as the kernel module is loaded. When a userspace process using libibverbs first calls ibv_get_device_list(3) (directly or indirectly through librdmacm), the libibverbs initialization opens all uverbs devices visible in sysfs and attempts to associate each with its corresponding userspace driver. To allow this, each verbs driver contains a function marked with the GCC "constructor" attribute, which causes it to run as soon as the driver is loaded into the application, whether at startup or via dlopen(2). The libibverbs initialization process is illustrated in Figure 3.

However, DPDK assumes full control of the network interface card and requires unbinding it from its kernel driver. This means that the kernel device cannot exist until the user process starts and initializes DPDK. Additionally, the librdmacm library used for connection management is simply an interface to the kernel rdma_cm module, which means that RDMA connection management must be performed in kernel space. This requires queue pairs, as well as completion queues and protection domains, to be mirrored in kernel space via uverbs. This complicates the initialization of an application using our verbs driver, and required the development of a USIW kernel module.

The USIW kernel module provides a stripped-down verbs driver that provides only the minimum necessary to interface with uverbs and the rdma_cm. When the module is loaded, it initializes a fixed number of verbs devices which are initially not associated with any hardware device. Our userspace driver registers itself with libibverbs as normal. However, when libibverbs calls our driver_init() function, we initialize the DPDK environment abstraction layer (EAL) and then configure and start each DPDK network interface. We use the kernel native interface (KNI) feature of DPDK to make a virtual Ethernet interface visible to the kernel for each DPDK interface, and then use netlink via libnl-3 [5] to configure the IP and MAC addresses appropriately. The USIW kernel module subscribes to netdev events and binds each KNI interface to a verbs device as the virtual Ethernet device is registered with netdev. At this point, libibverbs and kernel verbs are appropriately set up for connections to be established using our driver.

We use perfect match hardware filters to separate messages destined to each queue pair from messages that are destined to the kernel, falling back to software filtering if the NIC does not support perfect match filtering. Messages that the hardware places in queue 0 are forwarded to the kernel; while messages directed at all other queues remain in userspace.

A design where DPDK was initialized in the GCC constructor function was attempted, but this design did not work because DPDK itself uses GCC constructors. Constructor functions may be executed in an indeterminate order, and the main DPDK initialization function cannot be called until DPDK's constructors have been executed.

Most of the kernel connection management code is boilerplate to implement a connection request/reply handshake. However, we need to initialize the userspace queue pairs once the connection is established, before the user gets a chance to issue operations on the queue pair. To ensure that this happens, we use an anonymous file descriptor to communicate to a userspace thread that the connection was established in kernel space. The kernel driver will not continue connection establishment until it gets a response on the file descriptor, giving userspace time to set up Flow Director rules and other connection state.

2.3 Reliability

We define the Trivial Reliability Protocol (TRP), which is used as a thin layer between UDP and DDP to provide datagram ordering and reliability. The protocol headers are shown in Figure 4. The protocol header has been carefully sized to place the beginning of the DDP header at an 8 byte alignment when preceded by UDP, IPv4, and Ethernet headers, with the Ethernet header aligned at an 8-byte boundary. Packet sequence numbers (PSNs) are assigned in a monotonically increasing manner by the sender, and there is no relationship between PSNs assigned by the sender and the receiver. The ACK PSN indicates the next-expected PSN from the sender's perspective.

For every transmitted datagram, we use a two direct memory buffers and an indirect memory buffer. The first direct buffer is provided by the DDP layer and contains the entire DDP segment. TRP will insert

[1] Section 2.2 discusses our use of DPDK's Kernel Network Interface feature.

[2] As discussed in Section 2.3, acknowledgement of the last segment implies that all previous segments have been received. This mimics the behavior of all popular reliability protocols including TCP [20].

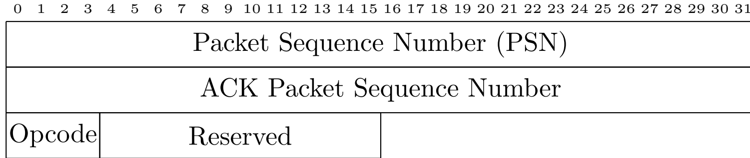


Figure 4: TRP protocol header.

Opcode	Value
0	ULP data segment
1	Connection Request
2	Connection Accept
3	Connection Reject
4	Connection Shutdown
5	SACK

Table 1: TRP protocol opcodes.

metadata into a private data region at the top of the memory buffer, including the packet sequence number and a time at which the packet should be retransmitted if an acknowledgement has not yet been received[3]. TRP then creates an indirect memory buffer which references the one provided by DDP. The indirect buffer ensures that our original memory buffer is not freed after the datagram is transmitted, so we may retransmit it if needed. TRP will then prepend a second direct memory buffer containing the TRP header and which is also used for the UDP, IP, and Ethernet headers.

The header contains a 4 bit opcode. The possible opcodes are shown in Table 1. The connection management opcodes are self-explanatory, and may include RDMA CM private data as payload. For a message with opcode 5 (SACK), the normal meaning of the PSN and ACK PSN fields is overridden—they instead refer to the minimum and maximum PSNs that are included in the selective acknowledgement. Like TCP, a selective acknowledgement will disable retransmissions for the acknowledged datagrams but will not cause them to be removed from the pending set.

2.4 Message Identification

As messages are segmented before being placed onto the wire, we would like to uniquely identify messages. This is especially important for Terminate messages, in order to associate any error with the initial work request to generate an appropriate work completion. At the iWARP level, we can classify messages into two categories: tagged and untagged, each with different reliability concerns.

Untagged SEND messages contain a message sequence number[4], which uniquely identifies the message within a connection. This means that multiple SEND operations can trivially be outstanding on a given connection, as it is trivially easy to track this message. For this reason, we allow many outstanding SEND operations on the wire at once.

Tagged messages may be RDMA READ messages or RDMA WRITE messages, and do not include an unique identifier. The only identifier we can use is the steering tag and tagged offset. Unfortunately, the user may queue many operations using the same steering tag, and these operations may overlap. When transferring messages according to the strict ordering rules of RFC 5040 [21], this is fine since responses may be matched with their requests by their position within the lower-layer protocol stream. However, when accessing non-volatile storage, it may be more efficient to send messages out of order. For example, we may receive two RDMA READ requests in a row, in which the second can be satisfied immediately from DRAM cache but the first must wait for the data to be read from flash storage. In this case, we would like to respond to the second request before the first.

For RDMA WRITE messages, we do not allow more than one outstanding per direction. This means

[3]On retransmission, the original packet sequence number is preserved, but the ACK sequence number is updated. Datagrams may be retransmitted up to five times before the operation fails and the connection is torn down.

[4]These message sequence numbers have no relationship to TRP packet sequence numbers.

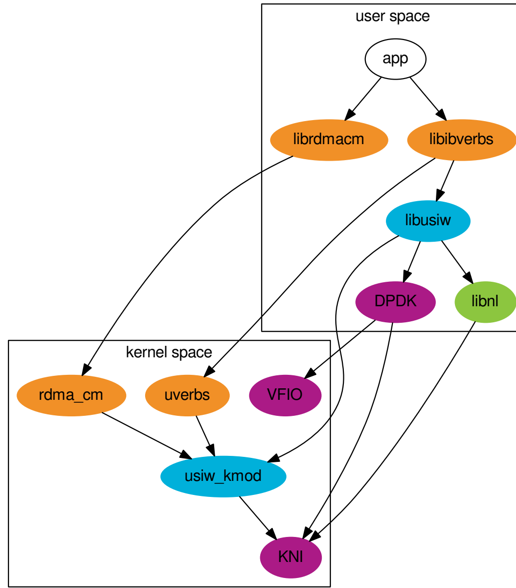


Figure 5: The relationship between USIW and the OpenFabrics Software and DPDK components that it uses. Components in orange are OpenFabrics Software components, those in purple are DPDK components or direct dependencies, those in green are other dependencies, and those in blue are USIW components.

that if the user queues two RDMA WRITE messages, the second will not start until the last segment of the first request has been acknowledged. For RDMA READ requests, we generate a sink stag based on the message sequence number of the RDMA READ request. This stag is used as an alternate name for the stag in the original memory registration, and uniquely identifies the RDMA READ response message. This allows us to have multiple RDMA READ requests outstanding at the responder.

2.5 Integration into OpenFabrics Software stack

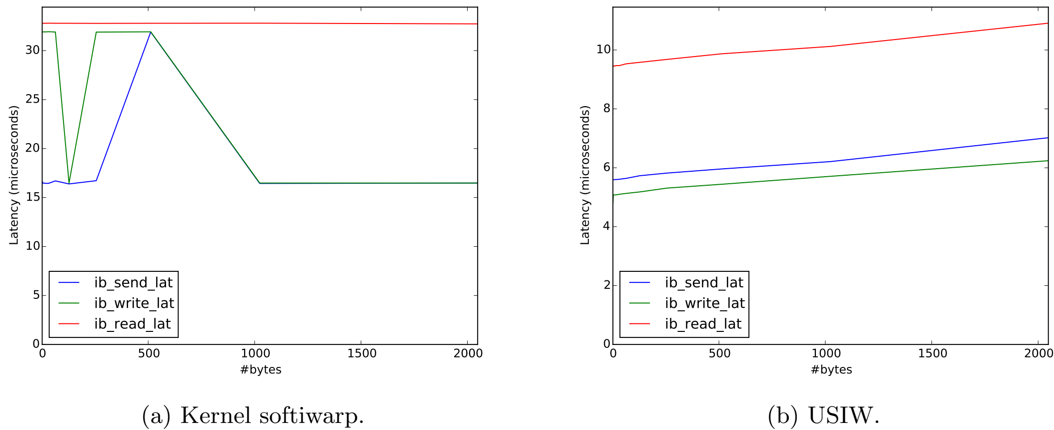
Figure 5 shows the dependencies and overall structure of USIW. USIW is implemented via the libusiw library, which is a libibverbs driver. An application does not call libusiw directly; rather, an application calls libibverbs functions, which in turn dispatch to libusiw functions. USIW, in turn, uses the DPDK libraries to process packets and perform network I/O.

Establishing an iWARP connection cannot be done directly via libibverbs, since connection management is outside of the scope of the InfiniBand verbs specification. Applications use librdmacm to establish connections between libibverbs queue pairs. However, librdmacm does not interact with libusiw—rather, it interacts with its rdma_cm kernel module.

DPDK relies on I/O virtualization to obtain direct access to the NIC from userspace. In Linux, this is provided by the UIO and/or VFIO subsystem. DPDK also provides a module called KNI to create a virtual Ethernet interface into the kernel which can interact with a DPDK process. We use this feature so that we can forward kernel connection management messages to userspace, as discussed in Section 2.2. The libnl library is used to send kernel netlink messages to configure the IP and MAC addresses of these interfaces.

3 Evaluation

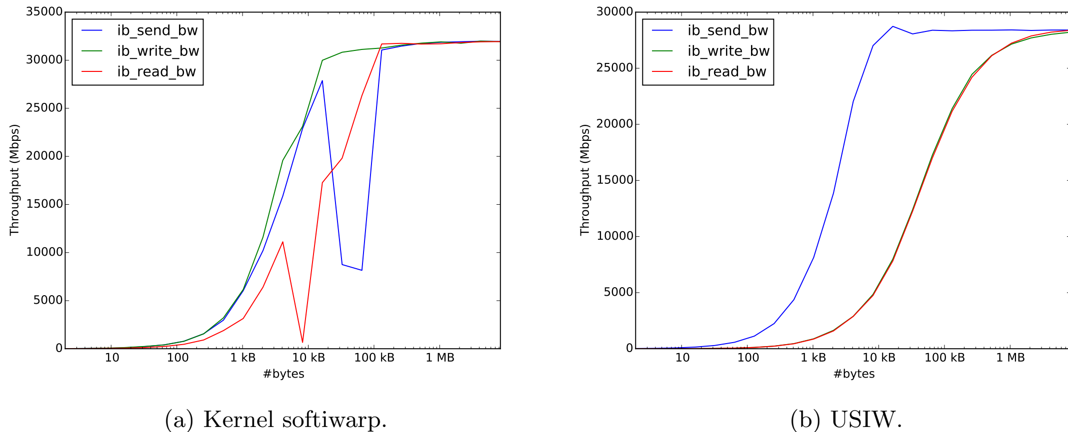
We evaluate the performance using the perftest benchmarks [15] supplied as part of the OpenFabrics Software stack. This study is preliminary, and we have not yet extensively tuned USIW, but these show a baseline of



(a) Kernel softiwarp.

(b) USIW.

Figure 6: Latency for USIW and kernel softiwarp calculated using perftest. Run for 10000 iterations per size.



(a) Kernel softiwarp.

(b) USIW.

Figure 7: Throughput for USIW and kernel softiwarp calculated using perftest. Run for 10000 iterations per size.

how we can expect DPDK to perform when compared to the kernel TCP sockets stack.

We show latency measurements for SEND, RDMA WRITE, and RDMA READ operations in Figures 6a and 6b. The latency reported for USIW is less than half of the reported latency for softiwarp in all cases. Additionally, the latency is stable for USIW while the latency for softiwarp fluctuates. Excluding the fluctuations, the kernel softiwarp latency does not appear to increase with message size while the USIW latency does increase with message size. Since transmission delay is directly proportional to message size, this indicates an extra source of software delay in softiwarp or the kernel TCP stack which masks the transmission delay.

The equivalent throughput measurements are shown in Figures 7a and 7b. For USIW, for all measured sizes up to 8 megabytes the throughput for SEND operations is much higher than the throughput for RDMA READ or RDMA WRITE. For RDMA WRITE operations, this is because USIW only permits a single RDMA WRITE operation on the wire at a time, as explained in Section 2.4. For RDMA READ operations, the reason for this is less clear—although RDMA READ requires a round trip, this should be amortized by the ability to have multiple outstanding RDMA READ Response sequences on the wire simultaneously.

Like the latency numbers, there is some fluctuation in the throughput numbers for softiwarp, although a

general pattern is visible if one ignores the fluctuation. The throughput for all operations is similar, which is expected, since large message sizes will amortize the differences in the implementation of each individual operation. However, the throughput for softiwarp exceeds that of USIW. Since we have yet to perform extensive tuning on USIW, this is not unexpected. While TCP has poor latency due to its congestion control and buffering mechanisms, these same mechanisms allow TCP to achieve high throughput by coalescing data segments and dynamically growing the window size in the absence of packet loss.

4 Related Work

FlashNet [23] is a kernel-space stack which extends SoftiWARP and SALSA to allow efficient remote access to NVM. In this case, a kernel implementation is efficient since incoming network requests are already processed by the kernel, so no context switches are needed. However, softiwarp is built upon the general-purpose kernel sockets stack, which provides many features not needed by an RDMA implementation which affect performance, including buffering, IP fragmentation, and TCP congestion control.

MICA [14] is a key-value store written using DPDK. MICA implements its own micro-optimized network stack, and uses hardware filtering to distribute incoming requests to the core responsible for the given key. However, unlike our implementation which uses perfect-match filtering to direct incoming messages to the correct queue pair, MICA uses hash filtering to distribute individual requests to different cores, regardless of the sender. Hash filtering makes sense for a connectionless request-response protocol; however, RDMA semantics require a flow of packets to have the same destination.

Arrakis [17] is an operating system which performs all I/O directly from userspace using a similar technique to DPDK. Unlike DPDK, Arrakis uses I/O virtualization to isolate processes accessing the same hardware. However, Arrakis focuses on enhancing the performance of traditional sockets applications, and explicitly does not provide RDMA semantics, since RDMA requires a key exchange to access remote memory.

5 Conclusion and Future Work

We have produced an iWARP implementation over UDP which performs data transfer entirely in user space and supports the OpenFabrics verbs interface. We require a kernel driver as required by verbs, but it is used only during initialization and for connection management; the kernel driver has no involvement in the data transfer path. We have provisions to support out-of-order RDMA READ and RDMA WRITE operations. Finally, USIW has less than half of the latency of softiwarp, although the measured throughput of USIW is slightly less than softiwarp. We can reduce and/or eliminate the throughput gap through future performance tuning.

Much work remains to be done. The most obvious piece is integration with storage using SPDK[5], to make a true comparison to FlashNet.

Spark[6] is a cloud application framework that we would like to run using USIW. However, Spark is designed to be used on top of a distributed storage environment, in which data is stored on the compute nodes themselves. We would thus like to be able to have a storage manager and an application share the NIC. Currently this is impossible, since a DPDK process takes exclusive control of the NIC. However, DPDK does have a multi-process mode, in which a designated primary process is in charge of resource allocation, and then direct NIC access can be delegated to any number of secondary processes. This can be leveraged via a daemon which starts as a DPDK primary process. Each verbs application would then set itself up as a secondary process, and request resources from the primary process as appropriate. Since each queue pair already uses separate hardware receive and transmit queues, the processes can directly access these queues without interfering with each other.

Using an individual hardware receive and transmit queue for each queue pair will not scale beyond the limits of the hardware. We can fix this by adding support to multiplex the hardware queues, such that multiple queue pairs can share a hardware receive and transmit queue. This would scale better at the cost of additional software latency on the receive path.

[5] `http://www.spdk.io`
[6] `http://spark.apache.org`

A recent alternative to the OpenFabrics Alliance verbs stack, libfabric[7], provides a more abstract API to access the same functionality, as heavily requested by the MPI community. Unlike the verbs stack, libfabric has no kernel component and does not specify any mechanism for providers to communicate with their kernel support modules. Thus, it is possible to write a libfabric provider entirely in userspace, assuming the NIC hardware resources, including management, can be mapped into userspace. We chose to implement USIW using verbs since our target applications are written against verbs. However, libfabric has the advantage of not assuming or requiring kernel support. Additionally, libfabric has support for reliable datagrams, which are more amenable to Spark applications which must have hundreds or thousands of connections open at the same time when using reliable connected queue pairs.

While we cannot perform zero copy receives, we may be able to perform zero copy sends. This requires pinning memory regions in the kernel and providing the physical addresses to userspace. Then, the DPDK memory buffer structure can be carefully constructed by the application such that DPDK will send the message directly from its original buffer rather than a memory pool. The complicating factor here is that memory regions may be of arbitrary size and are likely not contiguous in physical memory.

Acknowledgements

The author would like to thank Bernard Metzler, Jonas Pfefferle, and Patrick Stuedi for their advice and critique on the USIW implementation and this report.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other products and service names might be trademarks of IBM or other companies.

References

[1] M. Chadalapaka, J. Satran, K. Meth, and D. Black. Internet Small Computer System Interface (iSCSI) Protocol (Consolidated). RFC 7143, RFC Editor, April 2014.

[2] Intel Corporation. 82599 10 GbE Controller Datasheet. Technical report, Intel Corporation, October 2011.

[3] Intel Corporation. *Data Plane Development Kit (DPDK)*, December 2015. Version 2.2.0.

[4] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU Aligned Framing for TCP Specification. RFC 5044, RFC Editor, October 2007. Updated by RFCs 6581, 7146.

[5] Thomas Haller. *libnl3*, October 2015. Version 3.2.27.

[6] INCITS. Fibre Channel Framing and Signaling. Technical Report 470, December 2011.

[7] INCITS. Fibre Channel Protocol for SCSI. Technical Report 481, October 2011.

[8] INCITS. Fibre Channel Generic Services. Technical Report 463, December 2015.

[9] INCITS. Fibre Channel Switch Fabric. Technical Report 461, December 2015.

[10] InfiniBand Trade Association. *InfiniBand Architecture Specification Volume 1, Annex A 16: RoCE*, April 2010.

[11] InfiniBand Trade Association. *InfiniBand Architecture Specification Volume 1, Annex A 17: RoCEv2*, September 2014.

[12] InfiniBand Trade Association. *InfiniBand Architecture Specification Volume 1*, March 2015. Revision 1.3.

[7] `https://ofiwg.github.io/libfabric`

[13] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for Ethernet*, Dec 2012.

[14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[15] Mellanox. *perftest*, December 2015. Version 3.0-0.16.gb2f2e82.

[16] Bernard Metzler. *softiwarp*, August 2015. Commit b453df6dbc395de5133058ce440d5a679b2e2101.

[17] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[18] J. Postel. User Datagram Protocol. RFC 768, RFC Editor, August 1980.

[19] J. Postel. Internet Protocol. RFC 791, RFC Editor, September 1981.

[20] J. Postel. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[21] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040, RFC Editor, October 2007. Updated by RFC 7146.

[22] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports. RFC 5041, RFC Editor, October 2007. Updated by RFC 7146.

[23] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas, and Thomas R. Gross. FlashNet: A Unified High-Performance IO Device. Technical Report RZ 3889, April 2015.