SeaStar Unchained: Multiplying the Performance of the Cray SeaStar Network

David Dillow and Scott Atchley
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
Oak Ridge, TN
{*dillowda,atchleyes*}@*ornl.gov*

***Abstract*—The Oak Ridge Leadership Computing Facility (OLCF) supports many different systems and many different interconnects. The only common programming interfaces across these systems are BSD Sockets and MPI. Due to the design assumptions such as implicit buffering leading to extra copies, Sockets performance is almost universally lower than the native interface. Even in the cases that Sockets provides similar bandwidth as the native interface, it suffers from excessive CPU usage. MPI is the de-facto interface for intra-job communication, but is difficult to use between jobs and provides no ability to communicate with service nodes or off-system nodes (e.g. for I/O forwarding). We have developed the Common Communication Interface (CCI), a programming interface that exposes the advances in interconnect hardware, notably Remote Direct Memory Access (RDMA) and operating system (OS) bypass, while imposing as little overhead as possible. This API directly supports inter-job as well as off-system communication. CCI is a lightweight abstraction layer that provides point-to-point messaging and remote memory access.**

The Cray SeaStar ASIC, with its programmable embedded processor, provides an excellent platform to investigate the properties of various network protocols and programming interfaces. This paper describes our native implementation CCI on the SeaStar platform, and details how we implemented full OS bypass for common operations. We demonstrate a 30% to 50% reduction in latency, more than a six-fold increase in message injection rate, and an almost 7x improvement in bandwidth for small message sizes when compared to the generic Cray Portals implementation.

I. Introduction

The Oak Ridge Leadership Computing Facility (OLCF) supports several systems with differing interconnect technologies. Collaborating with industry partners, we developed the Common Communication Interface (CCI) to provide a programming API that is as simple to use as BSD sockets, but performs with minimal overhead when running on underlying interconnects such as Verbs and Portals [1]. As part of that work, the authors implemented a prototype that runs natively on Cray's SeaStar hardware. This provided a sanity-check on the API and provided renewed respect for the achievable performance of the SeaStar platform.

In this paper, we describe the operation of the SeaStar hardware (Section II) and give a high-level overview of the Portals API and details of its implementation on SeaStar (Section III). We describe the goals and a high-level overview of CCI, provide details of the prototype native SeaStar implementation (Section IV), and present our results (Section V).

II. SeaStar Hardware

The Cray XT series of supercomputers is built around the proprietary SeaStar ASIC [2]. This ASIC combines a high-speed, seven port router with an embedded processor to provide network access and reliability, availability, and serviceability (RAS) functions for each node on the network.

The router portion of the ASIC provides the basic building block of the 3D torus network for the machine. Each of the six network links has been measured to provide slightly more than 3 GB/s of *data payload* bandwidth in each direction of the full-duplex connection [3]. Each 64 byte packet on the link is protected by a 16 bit CRC checksum and uses a retry protocol to ensure reliable transmission over the network. The seventh port from the router connects to the receive engine on the embedded processor, providing inspection access to the NIC and DMA facilities to the host memory space. The ASIC is connected to the AMD Opteron hosts via a 16 bit HyperTransport (HT) link running at 800 MHz, providing over 2 GB/s of effective bandwidth to memory.

The embedded processor is a PowerPC 440 core clocked at 500 MHz. This dual-issue, 32 bit core includes independent 32 KB data and instruction caches, and has direct access to 384 KB of SRAM memory local to the ASIC. Additional hardware on the ASIC allows the embedded processor to access host memory via 15 address translation slots, each providing a 256 MB chunk of translated space.

The embedded processor is responsible for managing the flow of data into and out of the network. For transmission of packets, the TX DMA engine allows up to 31 commands to

This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

be queued for processing. There are four basic commands – load destination, transmit data, end packet, and end message. As the TX engine breaks up long messages into the 64 byte packets used on the network, it must first reset state and know the appropriate destination node on the network. This allows it to mark the first packet of the message with the Start-Of-Message indicator and resets the CRC calculation. Once the TX engine has been initialized for the message, it can be instructed to DMA from host memory using raw bus addresses over the HT link. There is no provision to DMA data from the ASIC's local memory. At the end of the message, the final DMA command will end the message. This command is able to notify the processor of its completion, and may optionally append a 32 bit CRC to the message to provide for end-to-end reliability. The end packet command is similar, and allows to multiple CRCs to be calculated and inserted for a message – for example, the header and payload of the message could have independent CRC checks.

Additionally, the ASIC TX engine has the ability to inject small messages into the data stream using programmed IO. A 256 byte buffer and four command FIFO allow the processor to send short messages independently of the main TX DMA engine. A hardware interlock prevents injection of the PIO messages into middle of a DMA message to the same host – such an event would corrupt the message from the DMA engine.

As the packets flow into the NIC from the router, they are placed into an RX FIFO. If the packet is marked as Start-Of-Message, a flag is set and a pointer to the FIFO entry is provided to the embedded processor. As the packet reaches the head of the FIFO, the source is matched against a 256 entry content-addressable-memory (CAM) to determine its disposition. Each CAM entry has an associated seven entry command queue. If the source does not match an entry in the CAM, or the associated command queue is empty, the RX pipeline is stalled and the embedded processor must take action to resume operations.

Each RX command in the queue may discard the data or DMA it to the the host using raw addresses on the HT bus. Each command handles an assigned length of up to 256 KB, which must be exhausted before moving to the next command. This allows one command to handle multiple packets from the same message. Commands may optionally check the CRC at the end of the length, and notify the embedded processor when the command is retired.

The embedded processor is responsible for noticing the Start-Of-Message flag and setting up the CAM and RX command queues for the message. It is also possible for the processor to read it out of the RX FIFO and discard it once it reaches the head of the queue, thus avoiding the need for a CAM entry for that message. The processor must also handle the DMA completion events, keep the DMA queues full for messages that do not fit in a single command, react to commands from the host, and notify the host upon completion.

Based on the work of Pedretti et al [4], software for the embedded processor is generally coded in C. This provides a good balance of performance and ease-of-development, and this improves along with the state of compiler technology.

III. Portals

Portals was designed and developed at Sandia National Laboratories in collaboration with the University of New Mexico. Beginning with the Sandia Red Storm system in 2002, Cray used Portals 3.3 as the interface to the SeaStar network[5].

A. Portals API Overview

The design of Portals had scalability as its primary goal. Portals did not promise to enable any application to scale, it instead only promised to not limit an application's ability to scale. To enable this scalability, Portals avoids maintaining state. It depends on reliable, in-order delivery of messages; connectionless communication; and all buffering is handled in user-space [6].

The design also was meant to allow zero-copy, OS-bypass, as well as *application bypass*, which allowed for overlap of computation and communication without application involvement. All messages contain match bits (i.e. tags) that will correspond to a posted buffer on the target. All messages are expected and messages without a corresponding buffer are dropped. This design gives the flexibility to the application to use the match bits to implement both one-sided and/or two-sided semantics.

The communication model is a *matching put* and a *matching get*. The match bits are not a specific memory address; they are compared against an application-defined tag in a list of *match entries*. Once matched, Portals looks up the associated *memory descriptor* for the entry. If the operation is a put, the data is placed in that memory location and it may have an optional *acknowledgment*. If it is a get request, the request is matched and then a reply is then returned to the initiator and placed in the initiator's buffer specified in the `PtlGet()` call.

B. Portals Implementation on SeaStar

The Portals 3.3 implementation on SeaStar is split between a network abstraction layer (NAL) that provides the network protocol for the Portals API that is implemented in a common library. The SeaStar NAL (SSNAL) lives in kernel space and provides the entry-points needed to send and receive messages. It provides the interrupt handler used by the NIC to notify the host processor of events requiring attention. Almost all Portal API calls from user-space must be "bridged" to the kernel-space library and NAL implementations, as the processing of inbound requests from the network take place in the kernel in the interrupt handler. Additionally, as portions of the Portal message header contain trusted information such as the source process id and message length, the header must be generated by the kernel in kernel-memory unaccessable to the user process. Allowing the user to control these values could bypass authentication mechanisms or confuse the network protocol.

1) Portals Transmit Path: Posting a message using `PtlPut()` packages up the API call and forwards it into the kernel library. There, the arguments are unpacked and validated. Once the library is assured this is a valid request, it allocates a control structure and creates the message header. Next, a list of DMA commands to pick up the header and message data out of the MD is created. This DMA program is placed into a mail box in the NIC's uncached memory region, where the firmware will pick it up during its next periodic poll for work. It will be added to firmware's list of pending transmit tasks, and once it reaches the head of the queue, the DMA program will be added to the SeaStar's TX queue. If the program is larger than the available space in the queue, the firmware will add as many operations as possible to the TX queue. Any remaining TX operations will cause the firmware to attempt to add more entries to the TX queue on each pass through the control loop.

The final command in the DMA program will be set to notify the firmware of the completion of this TX request. At that point, the firmware will post an event to the host indicating which TX request just completed, and interrupt the host. The kernel library will clean up the structures associated with that request and queue an PTL_EVENT_SEND_END event for the next time the user code calls into `PtlEQPoll()` or `PtlEQWait()`, which also requires a bounce into kernel-space.

2) Portals Receive Path: On the generic receive path, the firmware is notified of a new message when a packet arrives with the Start-Of-Message indicator set. The firmware peeks into the RX FIFO to verify that we are processing a Portals message, and then copies the contents of the Portals header into local memory for further validation. The source of the message is looked up in the list of active sources, and a new tracking structure allocated if it is not found. The destination process is verified to be alive, and a new RX task structure is allocated to handle this message. The message header is then transferred across the HT bus to the host memory using programmed IO, and the host kernel is interrupted to notify it to generate a DMA program.

The host kernel interrupt handler retrieves the match bits from the copied header, and walks the match entries for the targeted portal index. Upon finding a matching entry with room in the memory descriptor, it generates a DMA program to discard the header from the RX FIFO and transfer the body of the message to the appropriate locations in host memory. The firmware is notified of the generation of the program, and links it into the list of pending messages to process from this source. Once the source is assigned a CAM and has room in the associated RX DMA queue, the firmware feeds the program into the queue when it becomes the active receive task.

The final entry in the DMA program will be set to notify the firmware of its completion. The firmware then posts an RX completion event and interrupt to the host, which uses the information to queue the proper event for `PtlEQPoll()`/`PtlEQWait()` such as PTL_EVENT_PUT_END or PTL_EVENT_REPLY_END. Finally, the host will post a command to the firmware to release the tracking structure for this message.

One large difference between transmission and receive processing on SeaStar is that messages from different sources may be progressing through the RX hardware simultaneously. As each message can be many megabytes or even gigabytes in length, it will arrive broken up into 64 byte packets, and these will often be interleaved in the RX FIFO. The RX hardware handles the de-multiplexing of the individual packet streams using the 256 CAM entries to sort the messages into the buckets provided by the host-specified DMA programs. It is possible on large machines to have more than 256 individual sources sending a message to the same node at once; when this happens, resource exhaustion occurs and messages in excess of the available CAM entries will be dropped. Cray's implementation of Portals relies on the Basic End-to-End Reliability and CAM Overflow Protocol to detect and recover from these situations. The details of these protocols are not discussed here.

There is also an accelerated version of Portals for SeaStar that takes advantage of the contiguous process space available under Catamount, documented in [7]. In this model, user-space directly writes transmit commands to a mailbox on the NIC, which can generate the DMA program itself. For receive side, the match list processing is also performed by the embedded processor. When the match lists are short this improves the latency for messages, but the slower embedded processor loses to the host processor as the lists grow.

IV. CCI: The Common Communication Interface

Many HPC projects include a network abstraction layer (NAL) to isolate the application from the underlying network interface. Examples include MPICH's Nemesis, Open-MPI's BTL, Lustre's LNET, PVFS' BMI, and many more. Each project must then update their NAL as new interconnects and their programming interfaces are introduced.

ORNL, with industry collaborators, designed the Common Communication Interface (CCI) as a common NAL, which provides a scalable messaging service and remote memory access (RMA)[1]. The goal is to provide a NAL that could be adopted by new projects that wished to take advantage of common HPC interconnects. Currently, CCI supports Verbs (InfiniBand and RDMA over Converged Ethernet), Cray GNI (Gemini and Aries), Sockets (UDP and TCP), and Ethernet-Direct on Linux, which bypasses the IP stack. There is partial, outdated support for Myricom's MX (Myrinet and Ethernet) and Cray Portals3 (SeaStar).

The remainder of this section will provide a brief overview of the programming interface and the prototype implementation running directly on the SeaStar hardware.

A. Programming Interface

CCI provides access to the network device via an *endpoint*. The endpoint is the container of resources such as buffers and queues, and is bound to a specific device. An application may open one or more endpoints. CCI uses an event driven model and the application polls the endpoint for new events. Events include send completions, receive completions, connection handshakes, etc. When the application no longer needs the event, it returns the event to CCI for reuse.

Semantically, CCI uses *connections* to manage communication between two peers. Although CCI uses connection semantics, it tries to minimize per-peer connection state, especially per-peer buffering, as much as the underlying network allows. CCI provides choices for reliability and ordering to all application developers to choose the best fit for his needs. CCI offers reliable/ordered (RO), reliable/unordered (RU), and unreliable/unordered (UU) connections.

CCI provides two modes for communication, Message (MSG) and Remote Memory Access (RMA). The MSG service uses send/receive (i.e. two-sided) semantics for small messages. Unlike most two-sided interfaces, however, the application never directly posts receive buffers. When the application opens an endpoint, the CCI transport automatically populates the receive queue. After returning a receive event, CCI will then repost the buffer for further use. CCI limits a single message to the size of the underlying MTU or a pseudo-MTU, depending on the network's capabilities. The maximum send size (MSS) ranges from slightly under 9000 bytes on Ethernet down to 128 bytes for Cray GNI. CCI does not provide fragmentation and reassembly and requires the application to manage it if desired. By limiting messages to a single MTU, CCI can take advantage of multiple interfaces and/or multiple paths through the network for RU connections.

MSGs work well for control messages, but bulk data movement is better served by RMA. Available only on reliable connections, RMA provides one-sided, zero-copy semantics for READ and WRITE with an optional FENCE. If the hardware supports RMA, CCI will use it and, if not, emulate it. Applications may optionally include a message that is guaranteed to be delivered to the target after the RMA completes to notify the peer that the operation is complete.

B. CCI on SeaStar

Our initial prototype focused on CCI's message interface. While the wire format could be modified to allow kernel and NIC-based de-multiplexing endpoints to be used interchangeably, we focus on the interface that would most commonly be used on the compute platform – NIC-based. This mode allows for full OS-bypass and zero-copy receives, with multiple processes owned by the same user sharing a command queue to the NIC.

For the purposes of the prototype, we chose 4 KB to be the MSS. This is not an inherent limitation in the protocol, but was the smallest MSS that was thought to be useful. It could be adjusted upward to 8 KB without significant effort, at the cost of increased memory usage. Similarly, a number of the default memory sizes could be adjusted upward. The remaining discussion assumes a MSS of 4 KB.

Network resources are owned by the CCI endpoint. For user-mode CCI, each endpoint maps a number of memory regions into its address space:

- *TX areas.* Under CCI, each message is buffered for transmission. Each endpoint has its own pool of transmit buffers, currently 32 areas of 16 KB each, or 512 KB. This allows for 96 full-size messages to be queued, as we must account for the header and do not allow a message to span buffers.
- *RX areas.* Each endpoint owns a number of receive buffers. The current implementation allows up to 8,192 areas of 16 KB each, or 128 MB. This allows for 24,576 full-size messages to be queued, accounting for headers.
- *RX ready ring.* Used to provide the NIC with information on which RX areas are available for reception at this time.
- *Completion queue.* Used to receive notification from the NIC of command completions and newly received messages. The completion queues live in host memory, and empty slots are initialized to zero.
- *Command queue.* Used to instruct the NIC to transmit a message, update RX ready ring indexes, and return completion queue credits to the NIC for future processing. The command queue lives in uncached memory on the NIC, and empty slots are initialized to zero.
- *Host-side controls.* Shared page used to coordinate access to the command queue by multiple processes. The current number of entries available on the command ring and the next entry to fill are maintained in this area.

The CCI message header is described in Figure 1. The first word contains the message type, destination endpoint, and message length. As SeaStar does not have a mechanism to notify the recipient that a message has ended – only that a new one has started – it is vital that the length is correct. For CCI, a table of message types, destinations, and length is constructed in kernel-controlled memory. The NIC uses these tables to DMA the correct header for the message it is sending, then DMAs the rest of the message as built by untrusted user space. These tables currently require 292 KB

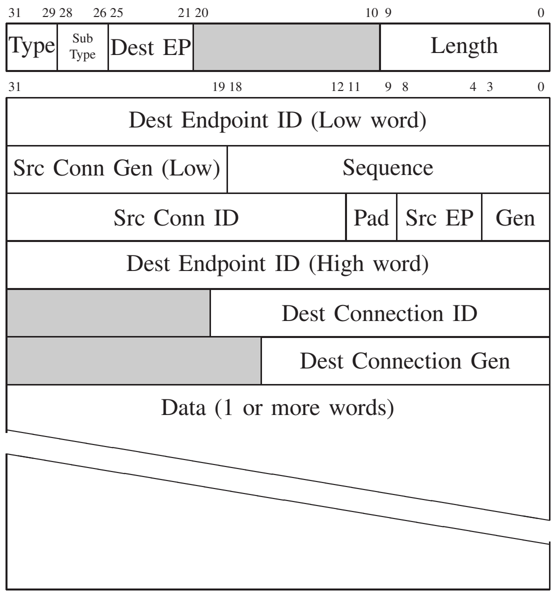

Figure 1: CCI message header

of kernel memory to handle 32 user-level endpoints for each of the various CCI message types.

As the rest of the message is not trusted, the header carries several fields to help ensure that each message is valid for a given destination. The 64 bit nonce is included to ensure that we properly established a connection with the receiving endpoint; the connection id and generation allow for lookup of the connection and verification that it is still alive. The sequence number and source connection information are only used for reliable connections; these fields allow the NIC to easily generate acknowledgments to the sender once the message has been placed in the host buffer.

1) Message Transmission: In CCI, messages are buffered on send. While adding a copy to the cost of sending a message, this design choice simplifies message lifetime for the user – similar to the BSD sockets interface – and allows an implementation to pre-register memory to avoid overhead of repeated register/unregister cycles. The native CCI implementation on SeaStar leverages this approach, with the kernel informing the NIC of the location of the transmit buffers during endpoint creation. DMA program generation for message transmission becomes a simple table lookup and a small amount of math.

To process a `cci_send()` request, the CCI library allocates a tracking structure from the local pool on the host. This tracking structure also holds the completion event that will be given to the user once the message has been sent. Once we have a tracking structure, we then allocate buffer space from our TX areas. We populate the header in that area based on the information contained in the connection structure, and copy the user message after the header.

With the message properly built, we then atomically claim a command credit from the host-side control area and advance the command index by one. We write the launch TX command to the NIC command queue, perform any bookkeeping needed for reliability, and return to the user.

The NIC picks up the command by noticing a non-zero value at the current command consumer index, verifies that the source endpoint is active, and that a credit is available for the completion queue to hold the event when the command is finished. A task structure is allocated in local NIC memory to track the request, and the TX DMA engine is programmed if there are three entries available in the ring. If room is not available, the request is queued until another transmit completes and frees up additional space. Once bookkeeping has been completed for the transmission, the command location is zeroed to indicate that the slot is empty, and the command consumer index increased to point at the next slot to be filled with a command.

Once the TX DMA engine has sent the message, we use our pre-reserved completion queue credit and put our notification in the next available slot for the source endpoint's queue. User space will poll this location during a `cci_get_event()` call and discover the completion. If the message was not sent on a reliable connection, we can immediately give the user the event from the tracking structure. Otherwise, we queue the message to await acknowledgment. Once the user returns the event, we place the tracking structure back in the free pool for re-use.

2) Message Reception: Message reception begins once the NIC firmware is notified of the arrival of a packet with the Start-Of-Message indicator set. The firmware allocates a source and tracking structure and inspects the first word of the message to determine the processing needed – is this a CCI message, IP packet, or Portals message? The CCI message handler retrieves the destination endpoint id and message length from this same word, minimizing the number of accesses required to the hardware RX FIFO.

Once the destination endpoint is known, we verify that there is enough room in the completion queue to notify the host, and that we have room in the RX area for the specified length. Implicit in these checks is verification that the endpoint is still alive – if not, then the "resources-not-ready" response is converted to a "refused" response to differentiate the error.

Given the sufficient resources, we attempt to program the RX DMA engine if we are the active task and space is available for the two entries required for the message – we discard the first word of the header, as that information is transferred in the completion queue entry. If no room is available, the task is queued until previous tasks complete and make space. Whether or not we programmed the DMA engine, we pre-construct the completion event and store it for use when the DMA completes. When the message has been safely transferred to host memory, we are notified of the DMA completion and remove our task from the queue.

We post the pre-computed event to the completion queue and release our tracking structure to the free pool, ending the firmware's involvement.

The user-space library checks the next slot in the completion queue during `cci_get_event()` and upon finding it non-zero, begins processing the message. We calculate a pointer to the message based on which RX area was used and the offset into that chunk of memory. The library then extracts the endpoint id and validates it against our endpoint nonce. We verify that the specified connection is active and that the source of the message matches the expected remote. For reliable connections, the sequence number is verified, and recovery performed as needed. We then retrieve an event from the free pool and prepare it for the application's consumption. The entry in the completion queue is zeroed out for future notifications. Once the application returns the receive event to the library, we process the receive area for potential reuse by the firmware and put the event structure back in the free pool.

The firmware code works from one receive area at a time. If the message being received does not fit in the current area, the firmware sends a completion event that notifies the library of which area is closed out and how many messages it handled during its life. As receive events are returned to the library, it checks if the area is inactive and if we have returned all outstanding events. Once this has happened, the area is ready to be re-used and its index is placed in the ready-ring. We normally notify the firmware of additional ready areas by piggy-backing on transmit requests, but if the library has several pending entries to send it will send a dedicated command to expedite the transaction, preventing stalls in a receive-heavy workload.

The CCI firmware also implements CAM swapping to prevent this limited resource from being exhausted. Providing storage for the out-going CAM contents requires approximately 128 bytes per node, or 2.5 MB for 19200 nodes. While this does not fit into SeaStar memory, a small amount is kept in local memory and the rest is mapped from host memory as cached address space. This is intended to provide quick access for the common case of a reasonable number of sending nodes, but continue to work in the extreme case. This code has not been fully excercised, and its effectiveness is unknown.

C. Caveats

As a prototype, the current implementation of CCI on native SeaStar has a few limitations:

- *RMA currently unimplemented.* Work has focused on the messaging functionality, and little work has been done to implement memory region registration or to assess how much of the processing can be effectively offloaded to the NIC.
- *Command queue fairness.* There is currently one user-level command queue shared by cooperating processes on a host. The kernel ensures that all of these processes are owned by the same user, but it is possible that one of the processes could starve the others by flooding commands into the queue. Further, commands owned by a process that crashes are not recoverable unless all other processes sharing the queue exit. These issues are not expected to be significant impediments to the expected scientific computing use-cases on compute nodes, but would need further work to resolve on service nodes.
- *Transmit scalability.* We currently allow for only 96 full-size messages to be queued at once for each endpoint, though it is possible to fit 8,192 messages in the buffers if they each contain 40 bytes or less of user payload. Messages sent on a reliable connection must currently occupy a slot in the transmission area until acknowledged. This is likely to lead to stalls on large machines. A possible approach is to double-buffer the transmitted messages, but this would require an extra copy if CCI_FLAG_NO_COPY is not used for the `cci_send()` call.
- *User-driven progress.* The current library requires the application to periodically call `cci_get_event()` to ensure progress is made. This is acceptable for the current transmit handling, but would need to be addressed if double-buffering is added.

V. SeaStar Performance

To evaluate the performance potential of the native CCI implementation on SeaStar, tests were performed on OLCF's test and development machine, "chester". At the time of the testing, chester was a single cabinet Cray XT5. Each compute node contained two 2.6 GHz Opteron processors, each providing six cores for a total of 12 cores per node. Each node contained 16 GB of DDR2 800 MHz memory. All tests were performed on two compute nodes adjacent on a single module; this ensured a consistent network latency between different tests.

The system ran the Cray Linux Environment 2.2UP03 during the Portals testing. This includes Cray's customized 2.6.16 Linux kernel, Portals 3.02.0001, and SeaStar firmware 03.12.2246. The CCI native testing was performed on a modified Linux kernel 2.6.37 and custom firmware.

A. Latency

To test latency, we implemented a ping-pong test for each stack. All binaries were compiled with `-O3` and `-fno-builtin`. For each message size, we conducted an average of 500,000 iterations. For the Portals testing, we implemented two strategies for transmission. One stategy bound the buffer, executed the `PtlPut()`, and unbound the buffer for each message. The second strategy bound the buffer once, and copied each message into the buffer prior to the `PtlPut()` call. We found that the version that copied

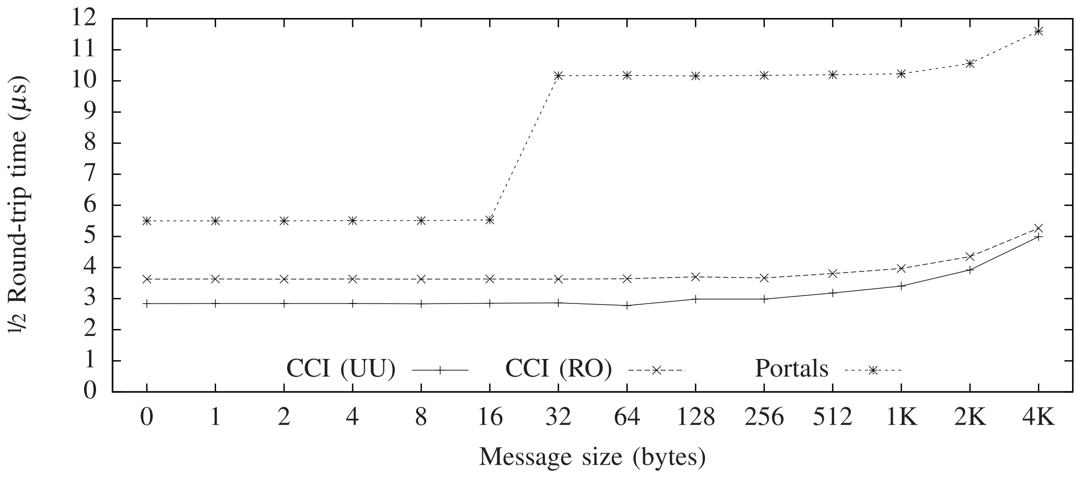

Figure 2: SeaStar Ping-pong Latency

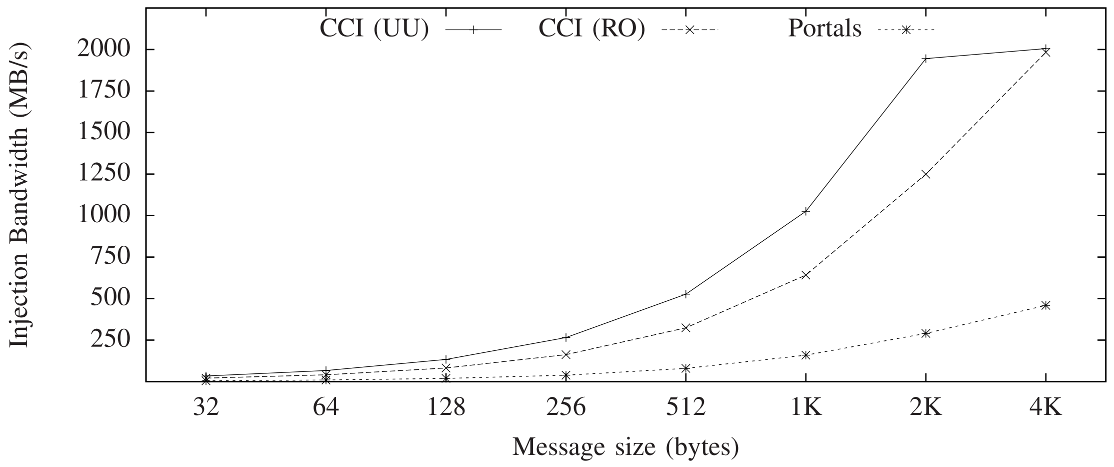

Figure 3: SeaStar Streaming Bandwidth

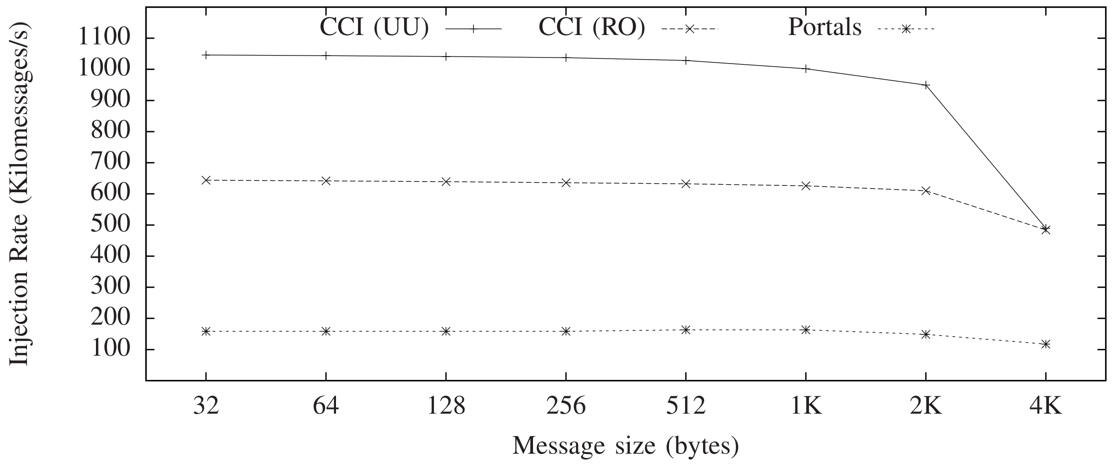

Figure 4: SeaStar Streaming Message Rate

produced lower latencies for messages up to 8 KB, and use those results here. All processes were bound to the same core on the socket closest to the NIC.

As shown in Figure 2, the round-trip through the kernel for transmission, and interrupting the host on reception add up. The jump in latency for the Portals implementation at 32 bytes denotes the point at which the message body no longer fits in the Portals header, requiring the NIC to wait for the DMA program from the host, and necessitating a second interrupt to indicate completion of the message. While not shown in the figure, requesting an ACK on a Portals message adds 4.4 to 6 μs to the latency of the reported non-ACK'd results.

By avoiding these overheads, CCI half-round trip latency starts at 2.8 μs and stays under 3 μs until we reach a packet size of 512 bytes on an unreliable/unordered (UU) connection. CCI latency tops out at just under 5 μs at the MSS for UU. Adding the ACK processing adds approximately 800 ns to the latency until the time to actually transmit a message begins to hide the processing time.

B. Streaming

To test message injection rate and bandwidth, we implemented a streaming test. After an initial burst of transmissions to achieve a specified number of messages in flight, additional messages are sent as previous transmissions complete. The transmit side sends messages for a specified length of time, and the client reports how many were received at the end. For both implementations, we tuned the number of messages in flight for maximum results for that stack.

As shown in Figure 3, CCI begins to run out of bandwidth at 2 KB messages, and caps out at 2 GB/s when using the maximum message size. Portals single-node bandwidth has been measured at over 1700 MB/s with 1 MB messages in our other work, so we believe these results may be dominated by the receive processing, and sending to multiple destinations could improve Portals performance.

More interesting are the message injection rate shown in Figure 4. CCI is able to send over one million messages per second – a message every 960 ns – from a single node for message sizes up to 1 KB. As we approach our maximum message size, our rate drops to slightly under 500,000 messages per second as we start to hit bus bandwidth and transaction limitations. As expected, the injection rate for reliable connections is lower until we reach the maximum message size. This reflects the additional latency and processing required for the acknowledgment packets.

VI. Conclusions and Future Work

While subject to a number of limitations, the prototype implementation of CCI on SeaStar produced promising results. We demonstrated half-round trip latencies of 2.83 μs (UU) and 2.99 μs (RO) with zero-byte payloads. These are 30% to 50% of the latency achieved by the stock Cray Portals implementation. As we reach the 4 KB maximum message size for CCI, our latency begins to be driven by bandwidth and we hit 4.99 μs (UU) and 5.26 μs (RO). In the streaming message tests, CCI was able to inject over one million messages per second (UU), or over 640 thousand messages per second for RO connections. This is a six-fold increase in injection rate, and leads to similar bandwidth gains for message traffic under 2 KB.

Of course, this is a prototype implementation, and imposes a number of trade-offs when compared to a fully-optimized production implementation. Addressing the issues of transmission scalability and thread safety will no doubt reduce some of the gains, though it is hoped that spending time optimizing the code would partially offset the costs.

Any work to complete the implementation would not be performed at OLCF. The SeaStar hardware has reached its end-of-life – indeed, Cray has since released two generations of improved hardware in its Gemini and Aries ASICs – and OLCF has no longer operates any XT class machines. The authors do have access to a personally-owned XT cabinet, and may eventually complete the implementation as a hobby and for the educational experience.

References

[1] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The common communication interface (CCI)," in *19th Annual IEEE Symposium on High-Performance Interconnects*, August 2011.

[2] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "Seastar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, pp. 41–57, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MM.2006.65

[3] D. A. Dillow, G. M. Shipman, S. Oral, Z. Zhang, and Y. Kim, "Enhancing i/o throughput via efficient routing and placement for large-sscale parallel file systems," *IEEE International Performance Computing and Communications Conference*, vol. 0, pp. 1–9, 2011.

[4] K. T. Pedretti and T. Hudson, "Developing Custom Firmware for the Red Storm SeaStar Network Interface," in *Proceedings of the Cray User Group Conference*, 2005.

[5] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. D. Underwood, "Implementation and performance of Portals 3.3 on the Cray XT3," in *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, 2005.

[6] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe, "Portals 3.0: Protocol building blocks for low overhead communication," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, 2002.

[7] R. Brightwell, T. Hudson, K. Pedretti, and K. D. Underwood, "An Accelerated Implementation of Portals on the Cray SeaStar," in *Proceedings of the Cray User Group Conference*, 2006.