Developing Custom Firmware for the Red Storm SeaStar Network Interface

Kevin T. Pedretti
Scalable Computing Systems Department
Sandia National Laboratories*
P.O. Box 5800
Albuquerque, NM 87185-1110
ktpedre@sandia.gov

Trammell Hudson
OS Research
1527 16th NW #5
Washington, DC 20036
hudson+cug@osresearch.net

May 16, 2005

Abstract

The Red Storm SeaStar network interface contains an embedded PowerPC 440 CPU that, out-of-the-box, is used solely to handle network protocol processing. Since this is a fully programmable general purpose CPU, network researchers may wish to program it to perform additional tasks such as NIC-based collective operations and NIC-level computation. This paper describes our experiences developing custom firmware for the SeaStar. In order to make the SeaStar more accessible to non-experts, we have developed a C version of the assembly-based firmware provided by Cray. This high-level language firmware should be much easier to understand and to quickly extend with new features. A detailed overview of the SeaStar programming environment and our C firmware will be presented along with optimization techniques that we have found beneficial.

KEYWORDS:
SeaStar, firmware development, PowerPC 440

1 Introduction

This paper describes our experiences developing custom firmware for the Red Storm SeaStar network interface using the C programming language [10]. While using a C-based SeaStar firmware may not deliver 100% of the theoretical performance, it should be much easier for other network researchers to understand and extend than the assembly-based firmware provided by Cray. It has also been more amenable to system wide optimizations that are available through modern C compilers. It is hoped that the groundwork we have done to enable programming SeaStar firmware in C will enhance the attractiveness of Red Storm as a network research platform and perhaps lead to novel uses of SeaStar.

In the past, programmable network interfaces have been leveraged to accelerate collective communication and to perform a variety of other NIC-level computation [5, 7, 12, 15]. Underpowered NIC processors are often cited as a limitation preventing more advanced uses of programmable network interfaces. The level of performance provided by SeaStar's embedded PowerPC 440 CPU, coupled with versatile send and receive DMA engines, may allow new applications to be offloaded to the network.

We believe C strikes a good balance of usability and performance. C was originally designed to be a systems programming language and, as such, excels at interfacing with hardware. In C, there is no need for the programmer to manage register allocation and instruction scheduling by hand, as there is in assembly. Although this may sacrifice some performance, an optimizing C compiler can often approach the efficiency of hand-tuned assembly code for these tasks. An optimizing compiler can also reduce the overhead of managing the C call stack by using optimizations such as function inlining, tail recursive

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

call identification, and passing function arguments in registers. In any case, the use of C does not preclude writing (or rewriting) performance critical sections of code in assembly when bottlenecks are apparent.

The rest of this paper is organized as follows. The next section provides a brief overview of SeaStar. In Section 3, we describe the tools that we have assembled which enable C-based SeaStar firmware development. Section 4 describes the C firmware in detail, followed by a discussion of optimization techniques in Section 5. Conclusions and final remarks are given in Section 6.

2 SeaStar Overview

SeaStar is a system-on-a-chip ASIC that was developed by Cray for Red Storm [1]. It provides a number of node support functions, including a programmable network interface and an interface to the Reliability, Accessibility, and Serviceability (RAS) system. A detailed description of SeaStar is provided in [3]. Here, we focus on the aspects of SeaStar relevant to firmware development.

Figure 1 shows a high-level view of a Red Storm node. Each node consists of an Opteron CPU, some amount of host memory, and a SeaStar that attaches the host to the high-speed 3-D mesh network. SeaStar is directly attached to the Opteron by an 800 MHz DDR, 16-bit wide HyperTransport connection, which enables a relatively high-bandwidth, low-latency path to host memory. Messages flow into and out of a node under the control of a firmware program that executes on the SeaStar's embedded PowerPC 440 CPU (PPC) [8], which operates at 500 MHz. A 6.4 GB/s Processor Local Bus (PLB) connects the PPC to a small local SRAM, transmit and receive DMA engines, a HyperTransport interface, and a high-speed serial interface to the RAS system. Each of these devices have their control registers mapped into the PLB so that the firmware can control them with normal load and store operations. The DMA engines have dedicated ports to the HyperTransport interface that allow them to read and write host memory without crossing the PLB.

The firmware uses the DMA engines to transmit and receive messages from host memory. Transmitting a message involves sending a series of DMA commands to the transmit (TX) DMA engine. Each non-contiguous region of host memory requires a distinct DMA command. Similarly, receiving a message involves sending DMA commands to the receive (RX) DMA engine. Once all of the DMA commands for a message have been enqueued, no additional firmware processing is required. The DMA engines perform the necessary packetization and end-to-end CRC verification autonomously. Once the message transfer is complete, the firmware receives a completion notification. If this notification indicates that an error has occurred, the firmware must take action to handle it.[1] This may involve retransmitting the message or possibly halting the node if the error cannot be handled.

Two types of memory are accessible by the firmware. First, a small 384 KB bank of local SRAM is available to store the firmware and its associated data structures. This memory is statically mapped into the top 384 KB of the PLB's address space so that it can be used in booting. Second, the firmware can access host memory through 15 windows of 256 MB each by configuring mapping registers in the HyperTransport interface. Reading host memory from the PPC should be avoided if possible—our measurements indicate that reading host memory has approximately ten times higher latency than reading local SRAM. The latency of writing to host memory can be hidden somewhat because the PPC supports up to four outstanding writes (and reads) without blocking. It is interesting to note that any memory mapped into the PLB, including device control registers and host memory, can be accessed by the RAS system via the SeaStar's high-speed serial interface. This provides a useful back-door for debugging.

3 Development Tools

3.1 C Compiler

One of our first tasks when we received our development hardware was to find a suitable C compiler for the SeaStar's embedded PowerPC 440 core. We considered a number of possibilities but ultimately chose GCC since it was what we were most familiar with and had PowerPC 440 support. Additionally, GCC may be freely modified and distributed to others under the terms of the GNU Public License [6].

GCC tool-chains are typically built from source. This requires downloading Binutils and GCC distributions from the GNU web site, configuring them, and then building them in the proper order. To facilitate this, we have created a pre-packaged SeaStar

[1] In practice, we have not observed any errors that were not due to faulty hardware or firmware bugs.

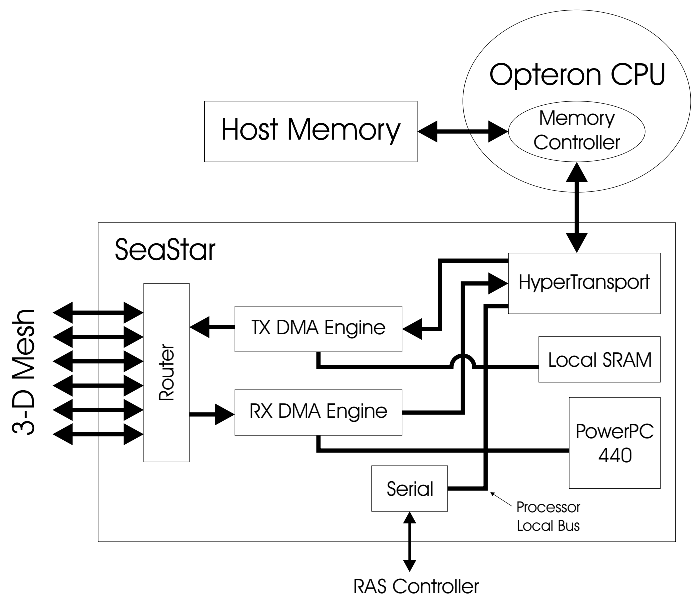


Figure 1: High-level view of Red Storm node

tool-chain distribution that is built using a simple build script. The resulting GCC compiler executes on whatever platform it was built on and generates output targeting the PowerPC 440 (i.e., it is a cross-compiler). A number of supporting GNU tools are also built, including an assembler, a linker, and an object disassembler.

The PowerPC 440 supports both big-endian and little-endian operation. Traditionally, PowerPC architectures store the most significant byte first in memory (big-endian) and x86-based architectures, such as the Opteron, store the least significant byte first (little-endian). This presents a choice as to what endianess the C compiler should target. If the C compiler targets big-endian, all data structures exchanged with the Opteron need to be encoded and decoded due to the differing byte orders. Targeting little endian eliminates the need for this translation, however it has the potential to generate less efficient code because GCC's little-endian support for PowerPC is less mature than its big-endian support. There are also some instructions, such as `lmw` that are not supported in little-endian mode, so we were unable to make use of them in block copy routines.

We chose to target little-endian because it greatly simplifies the interaction with the Opteron. In practice, most of the inefficiencies that we have observed in the generated assembly can be worked around by using simple techniques, such as adding a GCC-specific alignment attribute to all structure definitions.

3.2 Linking and Loading

The object code produced by the C compiler has to be linked into a single image and loaded into the SeaStar's local SRAM. Linking is accomplished by using the GNU linker, `ld`, along with a custom linker script that we have developed. Loading takes place using the standard mechanisms provided by Cray.

By convention, SeaStar's 384 KB of local SRAM is mapped at the top of the PPC's virtual memory space. The firmware code and data structures are placed into this region by the linker according to the commands in a linker script. Regions are defined for uncached data, cached data, text (the firmware code), and the stack. The uncached region is used to store data that is shared with the Opteron. Data that is only accessed by the PPC is stored in the cached region. Both the text and stack regions may also be cached since they are only accessed by the PPC.

The linking step produces an ELF image of the firmware. Two post-processing steps are required to convert this into a format compatible with the Cray loader. First, GNU objcopy is used to create a flat binary image from the ELF image. This removes a significant amount of unnecessary information—for our current firmware, a 166 KB ELF image is reduced to a 22 KB binary image. Second, an endian swap is performed on the binary image. This is necessary to compensate for an endian swap built into the Cray loader. The net result is that the image loaded into the SeaStar is the original, non-endian swapped binary image.

To prepare for load, the default firmware image provided by Cray is replaced with the C firmware image. The Cray boot mechanism is then invoked as normal. Several steps are performed by the Cray loader in order to start the firmware executing. First, while holding the PPC in reset, it uses the SSI interface to load the firmware image and a Cray-developed initialization shim into SeaStar SRAM. Next, reset is de-asserted and the initialization shim begins to execute. It performs a number early initialization tasks, including configuring the PPC's TLB to map all of the SeaStar SRAM (out of reset, only the top 4 KB is mapped [8]). When complete, the loader shim performs an unconditional jump to the address where the C firmware image was loaded. Finally, the firmware must set a flag at a well-known location in uncached SRAM to notify the Cray loader that it has begun executing.

3.3 Tracing

We have developed a low-overhead tracing mechanism to enable C-based firmware to be debugged and profiled. A trace point is created by calling either `trace(void)` or `trace_val(uint8_t value)` at the desired location, the later taking a user supplied value that will be stored in the trace record.

When a trace point is hit, an eight-byte trace record is created and stored in a ring buffer located in uncached SeaStar SRAM. Each trace record contains the value of the program counter when the trace point was hit, a timestamp, and possibly an eight-bit user value.

At any point in time, a program may be run on the Service Management Workstation (SMW) to fetch and decode any node's trace buffer. Figure 2 shows an example of the output of this tool. Each line is a decoded trace record. The first and second columns are the timestamp and user value, respectively. The next column is the function name and offset of the trace point in that function. Finally, the time delta between the current trace point and the previous trace point is displayed.

When tracing is turned on at compile time, each trace point incurs about 10 ns of latency. If it is turned off, there is no penalty at all for the trace points. We have kept the trace penalty low by dedicating a PPC register for maintaining the current index into the trace ring buffer and by using the PPC's link register to obtain the address of the trace point (i.e., the return address from the tracing function). The PPC supports up to four outstanding memory write operations so there is no need to wait for the trace record to land in memory before proceeding.

As with many debugging and tracing techniques, compiler optimizations such as inlining and tail recursion elimination may cause errors in the trace output's symbol resolution. The user supplied `value` can be used as a unique identifier, or it can be used to provide a small amount of data to the user.

4 C Portals Firmware

This section describes the operation of the C-based SeaStar firmware that we have developed. This firmware provides the low-level support needed to implement the Portals message passing API [14], which is the lowest-level communication mechanism on Red Storm.

Portals is an RDMA-like protocol with the primary difference being that addressing is by a set of maskable match bits rather than by a fixed address or remote-key. The receiver determines where in host memory to put an incoming message by walking a list of *Memory Descriptors* (MD), each containing a set of match bits and ignore bits. A match is found when an MD is encountered that has the same match bits as the incoming message, ignoring any bit positions masked by the MD's ignore bits, and that also matches the source node ID and process ID. MD's can be auto-unlinked from the list when they have received a specified number of transactions or when they have been filled to capacity. This style of matching semantics is well suited for implementing MPI [13] and has proven flexible enough for other upper-level protocols, such as Lustre [2] and the Catamount [9] job launch protocol.

The C firmware was originally derived from the Cray assembly-based firmware in November, 2004. Since then, we have made a number of architectural changes and performance optimizations. Some of these changes have been incorporated into the Cray firmware and, similarly, we have merged several of Cray's changes into the C firmware. We believe the diversity introduced by having two distinct firmware

```
e4681e8b:  24: 18ff0c4c mainloop+0014 (178 ns)
e4681ee4:  99: 63ff313c rx_complete+001c (84 ns)
e4681f0e:   0: 00ff318c rx_complete+006c (24 ns)
e4681f1a:   0: 00ff3198 rx_complete+0078 (374 ns)
e4681fd5:   5: 05ff12b4 handle_command+0030 (156 ns)
e4682023:   0: 00ff24b0 goaccel_tx_command+001c (80 ns)
e468204b:   0: 00ff24c0 goaccel_tx_command+002c (40 ns)
e468205f:   0: 00ff2124 resolve_source+0030 (18 ns)
e4682068:   0: 00ff212c resolve_source+0038 (102 ns)
e468209b:   0: 00ff2504 goaccel_tx_command+0070 (290 ns)
```

Figure 2: Example of tracedump tool output

implementations has been beneficial to both teams. Here, we discuss only the C firmware, however, much of this information also applies to the Cray firmware.

The C firmware currently consists of 3,434 source lines of C code and 253 source lines of assembly code, according to Wheeler's SLOCCount tool [4]. When compiled with GCC 4.0 using optimization level three (-O3), the resulting firmware image is 22 KB in size.

4.1 General Architecture

Figure 3 shows a high-level view of the host interface to the C firmware. On the host, there are a number of processes that use the Portals API to send and receive messages. These processes are split into two groups, termed *generic* and *accelerated*. Generic processes forward all of their Portals API calls to the OS kernel, which multiplexes them to a single firmware mailbox. Accelerated processes, on the other hand, send some of their Portals API commands directly to a dedicated firmware mailbox. Such commands are said to be *offloaded* to the NIC. Some commands, such as those related to process initialization, cannot be offloaded and are always forwarded to the OS kernel.

The firmware processes commands that it receives in its mailboxes. Each mailbox contains a command and result FIFO. The host posts commands to the command FIFO by incrementing the tail index in the NIC's mailbox structure. If the command returns a result, the host busy-waits until the firmware posts the result to the result FIFO. The use of FIFOs allows the host to post multiple commands before waiting for a result in some cases. In particular, commands that do not return an immediate result (e.g., transmit message[2]) can be efficiently streamed to the firmware.

Each accelerated process and the generic Portals implementation in the kernel contain an Event Queue (EQ) for the firmware to post asynchronous events into. Examples of asynchronous events are 'message transmit complete' and 'message reception complete'. Accelerated processes poll the EQ, if necessary, when the user-level Portals library is entered. The generic Portals implementation in the kernel is interrupt driven and only checks the EQ when the firmware has raised an interrupt. In order to reduce the number of interrupts, the Portals interrupt handler processes all of the new events in the generic EQ each time it is invoked. Individual events are small enough that they can be posted atomically by the firmware, allowing the host to simply read the next EQ slot to determine if a new event has arrived.

Limited NIC resources and OS limitations prevent all processes from operating in accelerated mode. Typically, there will be a small number of accelerated processes (one or two on each Catamount compute node) and the remaining processes will operate in generic mode. Supporting accelerated mode for Linux processes is particularly difficult because of memory paging—accelerated mode relies on message buffers being physically contiguous in memory. Catamount places application memory in physically contiguous regions so it is straightforward to support accelerated mode.

Currently, only generic mode has been fully implemented in the C firmware. It was implemented first because it was absolutely necessary for Linux nodes,

[2]Transmit commands can be thought of as returning a result much later in the form of a 'message transmit complete' event in the process' event queue, but this may be hundreds of microseconds after the command was posted, so it is not efficient to busy wait for completion.

where there are often many Portals clients, and it would function correctly for Catamount nodes until accelerated mode could be completed. Additionally, since generic mode Portals operates in the OS kernel and is interrupt driven, it allowed Cray to reuse an existing reference implementation of Portals provided by Sandia. Accelerated mode requires that the offloaded portions of Portals be reimplemented for the SeaStar. Much of the infrastructure for accelerated mode is already in place and we are actively working to complete it.

4.2 Data Structures

The firmware manages a number of data structures needed to transmit and receive messages. Figure 4 depicts an abstract view of the most important of these structures. First, there is one NIC Control block that contains global information concerning the entire firmware. Next, the accelerated host process and the generic Portals implementation in the OS kernel each have a dedicated process structure and mailbox structure allocated to them. These structures are one-to-one mapped and are split due to caching requirements—the mailbox must be uncached so that coherency is maintained with the host while the data in the process structure is accessed only by the firmware so it can be stored in cached memory. Each process structure has a pool of pendings, split into upper and lower portions, that are used to track in-progress message transmissions and receptions. Finally, each node that the firmware is sending a message to or receiving a message from has a source structure allocated to it. There is one pool of source structures for the entire firmware (i.e., all processes on each node).

Each pending is split into lower and upper portions, which are one-to-one mapped. The lower pending structure is located in cached SeaStar local SRAM and contains all of the information needed to progress and complete the message it represents. The upper pending structure is located in host memory and contains all of the information needed by the host regarding the message. In normal operation, the firmware never reads data from the upper pending structure because doing so requires a high latency round-trip across the HyperTransport link. The firmware does write information that is needed by the host into the upper pending. The upper pending structures are stored in cached host memory and are automatically kept coherent with respect to firmware writes by the Opteron's memory controller.

Every firmware-level process has two pools of pending structures, one managed by the firmware and the other managed by the host. The firmware managed pool is used for message receptions. When a new message arrives, the firmware allocates a pending from the target process' `RX pending free list`. The host managed pool is used for message transmissions. To prepare to send a message, a host process (i.e., an accelerated process or the generic Portals implementation in the kernel) allocates a pending structure from a free list that it maintains.

There is no dynamic allocation of any data structures by the firmware. All structures are pre-allocated at initialization time and inserted into free lists or slab caches, from which they may be rapidly allocated for use. While this introduces compile time constraints on the number of outstanding messages, in practice sizing these constants has not been too difficult. Resource exhaustion is addressed more fully in section 4.3.3.

The two primary consumers of SeaStar local SRAM are the source structures and the lower pending structures. The memory occupied by these structures can be calculated by

$$M = (S * S_{size}) + \sum_{i=1}^{N} (P_i * P_{size})$$

where S is the number of sources, S_{size} is the size of each source structure, N is the number of firmware-level processes, P_i is the number of pendings associated with process i, P_{size} is the size of each pending structure, and M is the SRAM occupied. For the current firmware, there are 1,024 global source structures and 1,274 pending structures allocated to the generic process (N is currently 1). These structures are small enough that several more similarly sized pending pools can be supported for additional firmware-level processes.

4.3 Firmware Processing

When idle, the firmware executes in a tight polling loop waiting for events. When an event occurs, the corresponding event handler is dispatched. The firmware is single threaded so handlers execute until they return, at which point a new event can be processed.

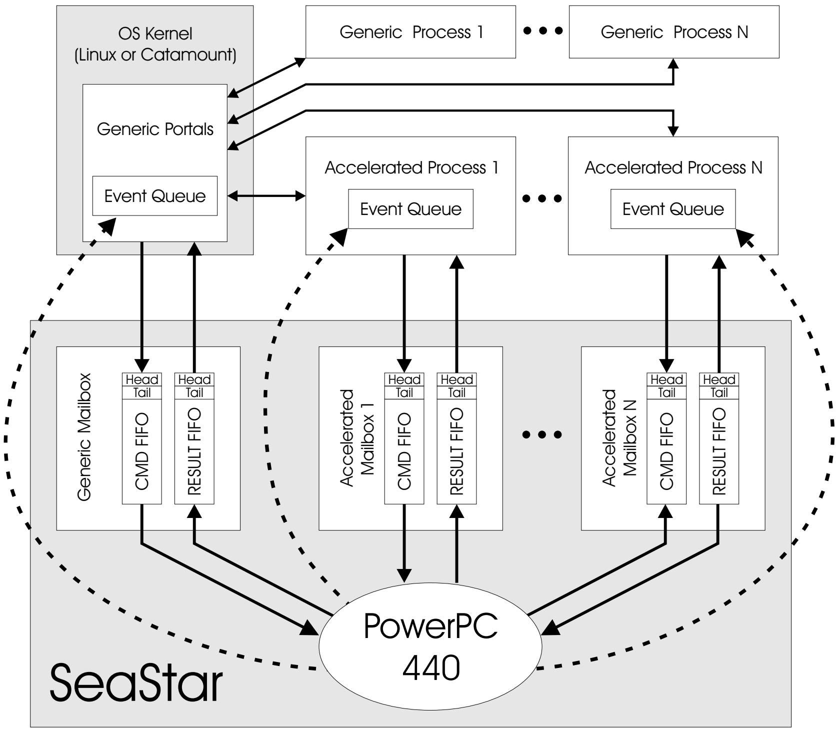


Figure 3: Firmware host interface

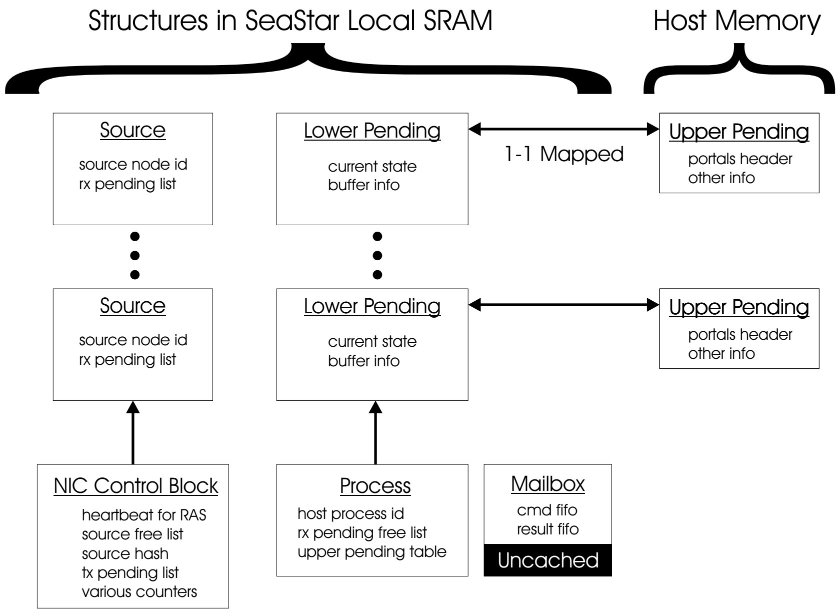


Figure 4: Firmware Data Structures

4.3.1 Message Transmission

In order to transmit a message, the host must setup the message header and then send a transmit command to the firmware. First, the host allocates a pending structure from the pool that it manages (i.e., the transmit pool). The host then stores the Portals header in the upper portion of the pending structure. Next, the host sends a transmit command to the firmware, including the message's pending ID, target node ID, payload address in main memory, and the number of bytes to transmit. If the message buffer is not physically contiguous, the host must pre-compute the commands for the TX DMA engine and pass them to the firmware. DMA commands for physically contiguous messages are generated by the firmware.

When the firmware receives the transmit command, it looks up the lower pending structure using the pending ID that the host pushed down and initializes it. If there is no source structure for the destination node, a new one is allocated and initialized. The lower pending structure is then enqueued at the tail of the `TX pending list` in the NIC control block. All transmits, regardless of destination or process type, are serialized through a single TX FIFO.

Once the pending reaches the head of the list, the firmware programs the TX DMA engine to transmit the message. The header is first DMA'ed out of the upper pending, followed by the payload DMA'ed directly from main memory.[3] If the message does not fit into the TX FIFO, the transmit state machine will yield and return to the mainloop until there is more room in the FIFO. Finally, when the message has been completely sent, the firmware unlinks the lower pending from the `TX pending list` and posts a completion event to the host process' event queue. This completes the transmit from the firmware's perspective. The host posts the Portals completion event to the application and then returns the pending to its free list.

4.3.2 Message Reception

The RX DMA engine notifies the firmware when a new message arrives from the network. In response, the firmware inspects the message header to determine the source node ID and the target host process ID. The source node ID is used to retrieve the corresponding source structure from a hash table of active sources. If no source structure is found, the firmware allocates a new one from its free list and inserts it into the hash table. The host process ID in the message header is used to look up the target firmware-level process. Once identified, the firmware allocates a pending from the target process' `RX pending free list`.

At this point, the future actions of the firmware depend on whether the target process is a generic process or an accelerated process. For generic processes, the firmware writes the entire Portals header into the upper pending, posts an event to the generic Portals event queue on the host, and then raises an interrupt. When the host receives the interrupt, it reads the event from the event queue and uses it to lookup the upper pending structure containing the Portals header. The header is then used to perform Portals matching on the host. Once the target memory descriptor has been identified, the host sends a receive command to the firmware, including the message's pending ID, the payload address in main memory, and the number of bytes to receive (and implicitly the number of bytes to discard). Like the transmit case, message buffers that are not physically contiguous have to have their receive DMA engine commands pre-computed by the host. The firmware uses the target buffer information in the receive command to setup the lower pending structure.

For accelerated processes, Portals matching is performed by the firmware. Therefore, there is no need for the firmware to raise an interrupt to ask the host where to put an incoming message. Once the target memory descriptor has been matched, the lower pending structure can be setup immediately. Like the generic case, the firmware writes the entire Portals header into the upper pending structure. This information is needed by the user-level Portals library when the firmware posts the message reception complete event.

Once the lower pending structure has been setup, the firmware links it to the tail of the target source structure's `RX pending list`. When the pending reaches the head of this list, the firmware programs the receive DMA engine to deposit the message directly into the target buffer in host memory. Once complete, the firmware posts a completion event to the host process' event queue. The host then uses the information stored in the upper pending to post the Portals completion event. Finally, the host sends a release pending command to the firmware to indi-

[3] This is often referred to as zero-copy.

cate that it is done with the upper pending structure and that the firmware can return the pending to the appropriate free list. This completes the receive from the firmware's perspective.

Unlike message transmission, there can be multiple receives in progress simultaneously—one from each source node. The packets of multiple incoming message streams arrive interleaved from the network. Normally, the RX DMA engine can transparently handle de-multiplexing the interleaved packets to the correct target buffers, based on the commands programmed by the firmware. In exceptional cases, there may be too many incoming messages for the RX DMA engine to handle. This is treated as a resource exhaustion case, described in the next section.

4.3.3 Resource Exhaustion

There are a number of NIC level resources that can be exhausted. For example, there may not be an unused pending structure available to handle a new message. Similarly, there may be too many sources trying to send to a node simultaneously. When this occurs, the firmware should become involved to resolve the situation.

The C firmware currently assumes that resource exhaustion does not occur. This has been sufficient to run several of Sandia's applications on approximately 7,700 nodes of Red Storm, which at the time of this writing is the maximum sized system partition that we have had access to. We have carefully monitored firmware resource usage and have never observed anything approaching dangerous levels. However, we expect that production-level use will occasionally trigger resource exhaustion. We are currently working on a simple go-back-n protocol to resolve resource exhaustion gracefully. The current approach is to panic the node, which results in the application failing.

5 Optimizations

Over time, we have added a number of optimizations to the C firmware. This has resulted in MPI zero-byte message latency falling from approximately 30 μs in the original C firmware to 4.9 μs currently. This section describes several of the more significant optimizations that may be be useful to others who are developing SeaStar firmware. All performance results in this section were obtained using MPI-level micro-benchmarks available from the Network-Based Computing Lab at Ohio State University [11].

5.1 Caching

One of the most significant optimizations that we have made is to enable the PPC's instruction and data caches. The SeaStar PowerPC 440 CPU implementation contains a 32 KB instruction cache and a 32 KB data cache. In general, data that is accessed only by the firmware may be placed in cached memory. Data that is accessed by both the firmware and external sources must normally be placed in uncached memory. Write-through techniques, described in Section 5.3, can sometimes be used to reduce the performance impact of using uncached memory.

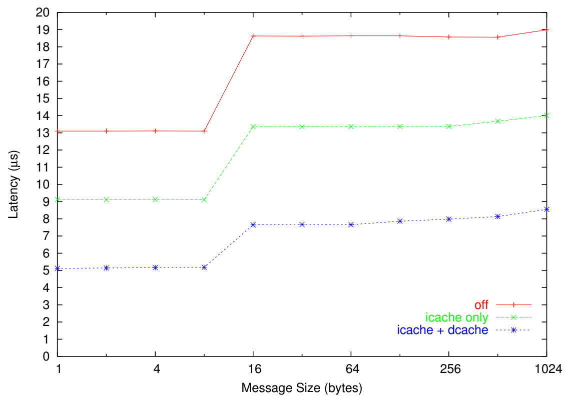
Enabling the PPC caches required us to gain a detailed understanding of Cray's firmware loading mechanism (Section 3.2). After much experimentation, we discovered that caching could be enabled without crashing the PPC by making use of the PPC's second address space. Immediately after it starts to execute, the C firmware creates an entirely new virtual memory space in the PPC's second address space.[4] Several of the memory regions in this space are configured to be cacheable. The C firmware then switches to the newly created address space. From this point on, the firmware executes in the newly created virtual memory space, which does not conflict with the original memory space setup by the Cray loader.

Figure 5 shows the performance of the C firmware with no caching, only the instruction cache enabled, and both instruction and data caching enabled. Caching reduces small-message latency between approximately 7.75 μs and 11 μs. The large jump from 8 byte to 16 byte messages is due to a protocol switch (Section 5.2). After the switch, there is more firmware processing performed per message, which increases the overall advantage of caching. The bandwidth curves for the three configurations eventually hit the same asymptote at around 1.1 GB/s.

5.2 Interrupt Elimination

Interrupting the host is a high latency operation and should be avoided if possible. We added an opti-

[4]The PPC typically uses one of its address spaces for the operating system and the other for user-level.



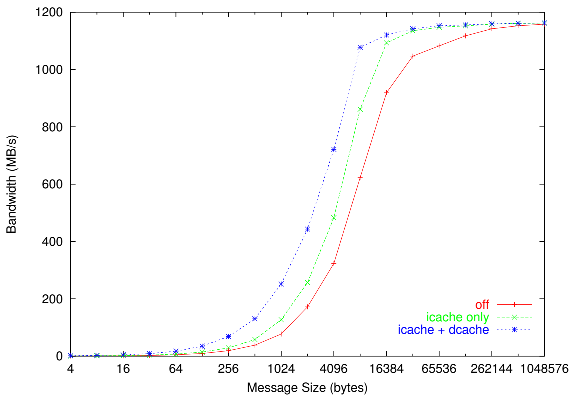


Figure 5: Effect of PowerPC caching on latency and bandwidth

mization for very small messages that eliminates one interrupt from each generic-mode message reception. Normally, each generic-mode receive requires two interrupts. The first is raised by the firmware when a new message arrives to ask the host where to put the message and the second is raised when the message reception is complete. The very small message optimization avoids the second interrupt by copying the data payload up to the host before raising the first interrupt. The host can then copy the message to its final location immediately after performing Portals matching.

Currently, we only perform this optimization for messages that can be transfered in-line with the Portals header. Due to packet size restrictions, the payload capacity of the Portals header is limited to 12 bytes. The very small optimization could be applied to larger message sizes with some additional work, however eventually the cost of the extra copy will negate any savings.

Figure 6 shows the effect of the very small message optimization. The optimization saves approximately 2.7 μs for messages of less than 12 bytes. This implies that the cost of an interrupt to the host is approximately the same—suggesting that accelerated mode, which uses no interrupts, may have small message latency in the two microsecond range.

5.3 Write-through Techniques

The firmware must store some data structures, such as the NIC Control Block and the mailboxes, in uncached SeaStar local SRAM. It is often possible to reduce the overhead of reading from these structures by maintaining shadow copies of them in cacheable

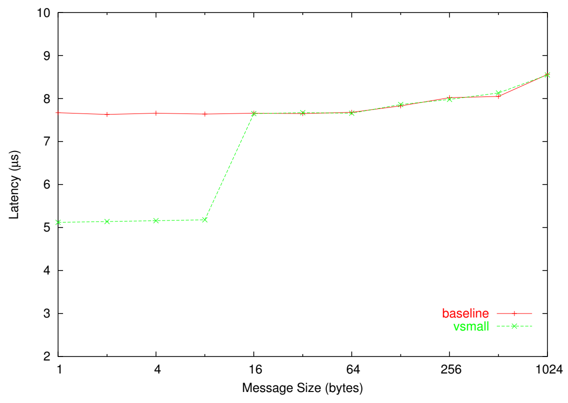


Figure 6: Effect of very small message optimization on latency

memory. Fields accessed by external sources are read and modified using the cached shadow copy and then *written-through* to the uncached structure.

The NIC Control Block provides an illustrative example of this technique. The NIC Control Block contains a heartbeat field that is monitored by the RAS system. If the firmware fails to increment the heartbeat value on a timely basis, the RAS system will mark the node as dead. In order to increment the heartbeat, the firmware must read its present value, add one, and then store the updated value. To eliminate the uncached read in this sequence, the C firmware maintains a cached copy of the NIC Control Block. The cached copy is always used to read and update the heartbeat value. As a final step, the updated value is written to the uncached NIC Control Block. The C code for this looks like:

```
niccb.heartbeat = ++cached_niccb.heartbeat;
```

All of the counters in the NIC Control Block are updated in this way. Fields in the NIC control block that are are only accessed by the firmware are always accessed using the cached copy.

The write-through technique was also used to optimize the firmware's FIFO manipulation routines. The DMA engine command FIFOs are mapped into the firmware's virtual memory space using an uncached memory region. Each FIFO is managed by head and tail pointers that are also mapped into the memory region. The firmware updates the tail pointer after it has written a new command to the FIFO and the DMA engine updates the head pointer when it is finished processing a command. A FIFO is full when incrementing the tail pointer would result in it equaling the head pointer. In order to optimize the firmware's FIFO manipulation functions, cached copies of the head and tail pointers are maintained by the firmware. The tail pointer is always read and updated using the cached copy and then written through to the uncached tail pointer memory location. The firmware uses the cached head pointer until there is a chance that the target FIFO might be full. When this occurs, the firmware refreshes its cached copy of the head pointer by performing an uncached read.

When this style of FIFO optimization was applied to all of the FIFOs manipulated by the firmware and by the host, we measured a net savings of approximately 6 μs per message. The optimization was particularly effective for the host's manipulation of the mailbox FIFOs since it reduces the number round-trip transactions across the HyperTransport link.

5.4 Pinning Globals in Registers

GCC provides a compiler-specific extension that allows global variables to be pinned in specific registers. This technique can lead to more efficient and smaller object code because many of the load and store instructions related to the pinned global variables can be eliminated.

All C source files need to be aware of which registers have been reserved for globals, perhaps by including the same header file. The following C code defines three global variables pinned in registers r31, r30, and r29, respectively:

```
register process_t *process asm ("r31");
register source_t  *source  asm ("r30");
register pending_t *pending asm ("r29");
```

Care must be taken to avoid using registers that are reserved by the target platform's Application Binary Interface (ABI), such as registers used to pass function arguments. Also, if too many registers are reserved in this way, the resulting register pressure will cause GCC to generate inefficient object code. The PowerPC 440 has 32 general purpose registers, of which we have used up to ten registers (r22–r31) for pinned global variables with positive results.

6 Conclusion

This paper has described our experiences developing firmware for the SeaStar network interface using the C programming language. We have presented the development tools that we have assembled in order to enable C-based firmware development for SeaStar, including a GCC toolchain and a tracing tool that we have developed. A detailed description of a C-based firmware for SeaStar with functionality similar to Cray's assembly-based firmware has been given. It is hoped that this firmware will be a good starting point for other researchers who whish to create custom firmware for SeaStar. Finally, we have presented several of the optimization techniques that we have found beneficial. In the coming months, we plan to further reduce the latency of the C firmware by offloading much of the Portals API from the host to the SeaStar.

7 Acknowledgments

We gratefully acknowledge the assistance of the members of the Scalable Computing Systems and Scalable Systems Integration departments at Sandia, especially Ron Brightwell, Doug Doerfler, Sue Kelly, Jim Laros, Mike Levenhagen, and Keith Underwood. In addition, we thank Cray for supporting our efforts and participating in many useful design discussions.

References

[1] Bob Alverson. Red Storm: A 10,000 Node System with Reliable High Bandwidth, Low Latency, Interconnect. In *Fifteenth Symposium on High-Performance Chips*, August 2003.

[2] Cluster File Systems, Inc. *Lustre: A Scalable, High-Performance File System*, November 2002. Available from `http://www.lustre.org/docs/whitepaper.pdf`.

[3] Cray Inc. *Seastar System Chip Specification*, 1.6 edition, January 2004.

[4] David A. Wheeler. *SLOCCount*. Available from `http://www.dwheeler.com/sloccount`.

[5] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-fast resource management. In *SC '02: Proceedings of the IEEE/ACM SC2002 Conference*, page 46. IEEE Computer Society, 2002. Avaliable from `http://sc-2002.org/paperpdfs/pap.pap297.pdf`.

[6] Free Software Foundation, Inc. *The GNU Public License, Version 2*, 1991. Available from `http://www.gnu.org/licenses/info/GPLv2.html`.

[7] Abhishek Gulati, Dhabaleswar K. Panda, P. Sadayappan, and Pete Wyckoff. NIC-based rate control for proportional bandwidth allocation in myrinet clusters. In *Int'l Conference on Parallel Processing*, September 2001. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/icpp01_rate_control.pdf`.

[8] IBM. *PPC440 CPU Core User's Manual*, July 2002.

[9] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Cray User Group*, Albuquerque, NM, May 2005.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[11] Network-Based Computing Lab, Ohio State University. *OSU Benchmarks, version 2.0*, 2004. Available from `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/benchmarks.html`.

[12] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, February 2002.

[13] A.B. Maccabe R. Brightwell, R. Riesen. Design, implementation, and performance of MPI on Portals 3.0. *International Journal of High Performance Computing Applications*, 17(1):7–19, March 2003. Available from `ftp://ftp.cs.sandia.gov/pub/papers/bright/p3-mpi-journal.pdf`.

[14] Sandia National Laboratories. *The Portals 3.3 Message Passing Interface, Document Revision 2.0*, December 2004.

[15] Adam Wagner, Hyun-Wook Jin, Dhabaleswar K. Panda, and Rolf Riesen. NIC-based offload of dynamic user-defined modules for myrinet clusters. In *IEEE Cluster Computing*, San Diego, CA, September 2004. Available from `http://nowlab.cis.ohio-state.edu/publications/conf-papers/2004/wagnera_cluster04.pdf`.