PCIComm: A Linux Device Driver for Communication over PCI Shared Memory

Gerald Britton
Vanu, Inc.
gbritton@vanu.com

Abstract

Computing clusters are often comprised of a set of full computers connected by a network. Several technologies are emerging to more tightly couple the CPUs in clusters. This paper addresses a Linux device driver for an inter-node communication system. It describes a system utilizing commonplace existing hardware to perform I/O between embedded computers communicating over a shared PCI bus. The system utilizes the kiobuf architecture provided by the Linux kernel to abstract memory regions and allow for zero-copy I/O to and from user application memory. PCIComm is split into a generic management system and hardware specific modules. An initial implementation of the system is discussed pending further development of the Linux kiobuf architecture, which will permit the full functionality of this system to be implemented.

1 Introduction

Many computing environments require a collection of processors to economically build a system capable of performing the required computations. These clustering environments often utilize common networking protocols for communication between nodes. In many implementations, the physical medium for this communication is some form of Ethernet (including Gigabit Ethernet for very high performance systems). Clusters can also be constructed from several embedded computers connected to each other and a master host via PCI or some other system bus (see Figure 1). PCIComm provides a low latency, high bandwidth physical layer on which to build a cluster communication network. This paper discusses integration of this embedded communication layer into the Linux kernel and providing several access layers atop that for different clustering applications.

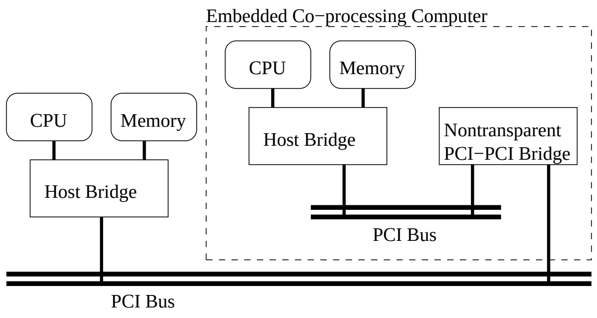
Embedded Co−processing Computer
CPU
Memory
Host Bridge
Nontransparent PCI−PCI Bridge
PCI Bus
CPU
Memory
Host Bridge
PCI Bus

Figure 1: **Example Embedded Cluster Setup**

Figure 1 displays the intended operating environment for this device. A host computer with a PCI bus contains one or more embedded computers on the host's PCI bus. These are full computers with their own system memory and potentially private devices. Their communication is to be handled through shared memory across the PCI bus. This system was initially intended to be used with Total Impact's Total Power MP cards 4. These cards combine 4 PowerPC G3 or G4 CPUs with memory on a PCI card and are ideal for use in this style of clustering environment. As part of this project, an attempt was made to port Linux with Symmetric Multiprocessing (SMP) to these cards. At this time, some uniprocessor work has been completed and SMP support is being worked on. Due to memory coherency problems, SMP support is proving difficult to complete.

The communication system described in this paper is intended to provide the required communication between these cards and the host. As the intended target of this system is incomplete at this time, the initial implementation will include a hardware simulation module for testing purposes. Once the system is complete, further testing will be possible.

The system described here must support all the common needs of a clustering environment. It must be designed within the confines of the Linux kernel, and must provide efficient, high bandwidth movement of data between processes running on separate nodes of the cluster. It must also be well abstracted, providing simple user and kernel level APIs with independent hardware modules actually performing operations. The system must also provide extensibility, allowing it to relatively easily integrate with new kernel infrastructures as they become available.

2 Related Works

One of the primary problems with I/O speed is the traditional copy nature of Unix read/write syscalls. Moving data between user space and a device involves first copying it into a kernel buffer and then using either DMA or PIO to write the data out to the device. Several systems have been developed to eliminate some of the copies from this operation.

Fbufs [1] provide a solution to a problem similar to the one presented here. Fbufs dynamically alter process memory mappings allowing memory pages to be transfered between processes. This would be permissible in this clustering system except that the pages would be existing on physically different nodes on the PCI bus. This would likely make memory reads from the regions inefficient as the process is expecting memory to be in the local main system memory.

Container Shipping [2, 3] provides an infrastructure similar to the Linux kiobuf infrastructure used in this design. Containers hold memory references which can then be passed around within the kernel instead of copying data. As with all systems which share memory between processes, memory protection must be considered to ensure that the intended operation is guaranteed. The user level API also requires process page table alterations when shipping containers into and out of a process.

Comparing to Fbufs and Container Shipping, PCI-Comm eliminates modification of the process page tables. The only memory subsystem operation performed is the pinning of memory pages into physical memory after they have been faulted in. This optimization does not provide a huge savings in single processor systems. In multi-cpu systems, synchronizing the memory subsystems of the cpus is expensive, and for high performance systems, this optimization may help significantly.

3 Other Architectures

The Virtual Interface (VI) Architecture [8] is a hardware independent architecture for distributed computing. It provides a "Remote DMA" operation which allows for data transfers to be modeled as handing memory regions between systems. This is very similar to the system described here. It does not, however, does not specify hardware requirements, or implementations. The system described in this paper is designed with specific hardware requirements. Requiring only simple shared memory regions and interrupts, hardware modules for PCI-Comm can be easily implemented for most existing hardware.

Infiniband [9] was announced after the conception of this project. It is a potential successor to PCI and is intended for bus sharing between peers and will likely offer other better primitives to implementing a system similar to this one. Again, the system described in this paper is intended for existing PCI hardware rather than a redesign of the entire bus architecture as Infiniband does.

AG Electronics [10] has an embedded coprocessor card and supports communication with it over the PCI bus using PCI I_2O messaging. This system is intensive on interrupts and limits the hardware that can be used to implement a generalized communication system.

Montivista Software supports using Hard Hat Net [11] to provide a network connection between embedded computers using the Intel 21554 Non-Transparent PCI-PCI bridge. The system depends on a very specific configuration of the bridge and only provides a network interface. For cluster data processing applications, high-bandwidth is key and the overhead of the network stack will likely limit the bandwidth and cause more processor overhead.

There are several other specialized communication systems for low bandwidth transfers between hosts and embedded computers. Most of these systems cannot be generalized to support a wide variety of hardware as the system described in this paper does.

4 Design Considerations

This design makes few assumptions about the underlying hardware and does not give a specific hardware implementation. A general API is provided by a base device driver. The low level hardware is acted on by a pluggable hardware module allowing for multiple independent hardware implementations. This hardware abstraction requires only a few simple primitive operations and can be implemented with a minimum of assumptions about the available hardware on the PCI bus.

4.1 PCI Shared Resources

PCI devices can export resources to the busses on which they reside. These resources are memory, memory-mapped I/O or standard I/O. In this design we assume two main things about the hardware available between the two kernels.

We assume that the hardware provides a method for sharing the entire memory core of each node onto the PCI bus of the remote nodes. This allows for all nodes to freely read and write from each other's memory.

We assume that the hardware provides at least one interrupt in each direction between nodes which intend to communicate directly. This is necessary to allow for low latency transactions allowing the remote node to operate without the need to poll a memory location for a completion notification. Messages will be sent primarily by causing an interrupt to the remote node. Care must be taken in the interrupt handlers to prevent dead- and livelock situations which can arise depending on the action required by the interrupt.

4.2 Linux `kiobuf` structures

One of the main problems with high bandwidth data processing is the significant cost in memory copying. Eliminating as many copies as possible is one of the main goals of PCIComm. Some work has been done in the Linux kernel to support zero-copy I/O for several subsystems. Primarily this exists to allow direct access to block devices (such as hard disks) allowing database software to perform raw I/O for increased performance and reliability. It will also help provide for zero-copy in the network layers, which is slated for addition to the Linux kernel in the next 1-2 years.

The I/O system described in this paper will take advantage of the primitives currently provided for performing zero-copy I/O. The Linux `kiobuf` structure (similar to the container structures used by Fbufs [1] and Container Shipping [3]) allows for arbitrary user or kernel space pages to be passed around between kernel subsystems. These pages can then be pinned down in physical memory by an end level device drivers which will perform DMA to or from the memory. After this is completed, the page is unlocked and is again free to be paged out of memory by the memory management subsystem.

Since the granularity of these `kiobuf` structures is arbitrary, and all pages can be accessed independently, a physically discontinuous region of memory can be processed easily into either separate I/O operations, or into a scatter-gather list for hardware which supports that level of I/O. The `kiobuf` mechanism also provides for callbacks when I/O completes or all references to a `kiobuf` vanish. This allows for these buffers to be essentially ignored once I/O has started. Notification can automatically be given upon completion allowing I/O to be performed in an asynchronous fashion and lazily if desired.

5 System Design

PCIComm is designed to be a device driver with a simplistic user level API providing connections and two call read/write mechanisms. The data movement mechanism allows for preposting of receive buffers for asynchronous receive without requiring separate kernel buffers.

5.1 User Level Interface

The user level interface is designed to be simple to use. A single handle per connection may be bound to a single incoming or outgoing connection which will then be used to pass pages of data across the link in both directions. A opening a single device allows user programs to obtain this handle into the driver. This handle is then used with purely `ioctl()` calls for functions into the driver. These functions are wrappers around the `ioctl()` calls.

5.1.1 Connection Management Functions

Connections between two systems are created asymmetrically for simplicity in this initial implementation. There are four states a connection can exist in: LISTENING, CONNECTING, CONNECTED, and CLOSING. A connection will remain in a CLOSING state until all pending I/O requests are complete at which point it will be destroyed and its resources freed.

`void pcic_listen(int fd, char *dev, int cid)`
: Create a listening connection on one end of a device.

`int pcic_connect(int fd, char *dev, int cid)`
: Initiate a connection with a listening connection on the other end of the device.

`void pcic_disconnect(int fd)`
: Terminate a connection.

5.1.2 Data Transfer Functions

The data transfer functions all use the same general form. They return the number of bytes processed upon success, and -1 on failure, setting errno to the appropriate error code. Each operation either takes or returns a memory address, and takes a length. Operations which move pages into the kernel's control (`pcic_prepost_recv`, and `pcic_send`) will move the contiguous region of memory pointed to by `data` of length `len`. Operations returning data will set the pointer pointed to by `data` to a region of length `len`. If a contiguous region this size does not exist, these will return the actual length of the region existing at the returned address. Transmitted buffers are recycled back to user space in the same order they were provided to the kernel. Assuming consistency is used with lengths in sending and receiving, contiguous regions can be expected with the returns. The data return functions will sleep if memory is not ready to be returned to user space.

`int pcic_prepost_recv(int fd, void *data, int len)`
: Places a memory region under kernel control for receive purposes.

`int pcic_recv(int fd, void **data, int len)`
: Returns received data memory regions back to user control.

`int pcic_send(int fd, void *data, int len)`
: Places a memory region under kernel control to be transmitted to the other end of the link.

`int pcic_send_recycle(int fd, void **data, int len)`
: Recycles a memory region back to user control after the kernel has finished transmitting the data out of the block.

5.2 Kernel architecture

This system design is discussed from the point of view of a single node. The communication method is symmetrical and all nodes communicate peer-to-peer. The device maintains two queues locally, a queue of kiobufs representing data to be transmitted, and a queue of remote pages to transfer data into. It also maintains buffers for passing control messages between nodes.

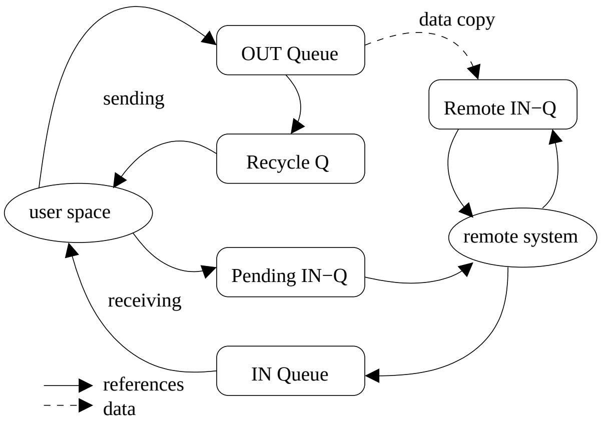

Figure 2: **Data/Page Reference Flow**

5.2.1 Outgoing `kiobuf` queue

A queue of outgoing buffers is maintained in kiobuf form (Figure 2. These buffers are user space buffers passed into the kernel. The user is expected NOT to modify these buffers while the kernel is holding them else the data sent will be unpredictable. In the final system, arbitrary sized and aligned data will be permitted to be placed in the queue. As user space memory addresses may differ from the corresponding kernel addresses, the queue will keep track of pages as viewed by the kernel as well as the user space virtual memory mapping of that memory.

This allows the kernel to recycle these buffers back to the user after I/O has completed.

5.2.2 Exported Incoming `kiobuf` queue

A system supporting partially asynchronous zero-copy receive requires preposting of buffers into which data will be received. These buffers are acquired from user space in the same way as transmit buffers. They are initially placed in the same style of queue as described above (Figure 2). Pages containing the user space memory are pinned in physical memory. The page references are sent to the remote system where they are translated into an equivalent queue. This exported queue is a queue of pointers into the PCI exported memory of a remote system.

Simple memory copies can be used to move data between the systems. Copying memory from the outgoing kiobuf queue (faulting in pages automatically if necessary) to the exported queue (which contains locked memory in the remote system) fills the incoming queue of the remote machine with data. The pages are then pulled off the exported queue and "returned" to their origin system. The pinned pages are then unlocked allowing normal paging to resume. Through the entire process, only a single data copy occurs. All other movement is done by passing page references to the memory pages.

5.2.3 Messages

The PCIComm drivers in independent kernels communicate with each other with messages. Messages are passed between kernels by placing a message in local memory and triggering an interrupt to the remote device corresponding to the local one. The message is retrieved and acknowledged by the remote device. The local device can optionally wait for this acknowledge before proceeding for synchronization or return value purposes. Asynchronous messages still require acknowledgement to prevent multiple messages from being sent simultaneously given a single message control block. A future enhancement could be done involving a queue of operations allowing multiple operations to be sent simultaneously. This would require more advanced inter-machine locking, however.

An optional data block may also accompany a message. This data block is copied into a buffer in the remote machine prior to the message being passed. This allows for faster messages as the data block is potentially much larger than the message control block, and the data push operation is likely to be faster than a pull operation due to available optimizations for PCI bus writes.

Connect
: Sent to signal a request a connection. If the receiver has a listening connection, it should transition to connected and be associated with the connection making the request. This is a synchronous operation as the error code is required to be passed up to the user.

Disconnect
: Sent to indicate that a connection is to be terminated. The sender will destroy all reference to the connection upon acknowledge of this message. This is an synchronous operation as proper operation requires blocking until the disconnect is complete. If the disconnect operation was asynchronous, pages of the remote system might be written to asynchronously by the local system if the connected status of the two sides of the connection are out of sync. The disconnect message is sent asynchronously and the connection is left in a CLOSING state while the disconnect operation sleeps pending all in progress I/O completing.

Export kiobufs
: Sent to transfer memory references of incoming data queue. This exports the queue to the other end of the connection. This is an asynchronous operation. The kiobufs are translated and queued on the remote end to be filled with data and "returned" later.

Send outgoing data
: Sent to acknowledge competition of data transfers and dequeue incoming queue buffers back to the receiving end. This is the "return" operation sending kiobufs back to the original system. This is an asynchronous operation.

Neither the kiobuf export operation, nor the data transmit operation requires acknowledgement as there is no error to be returned. This allows the sender to process more information while the operation completes on the remote system.

5.3 Node Discovery

Discovering other nodes on the network is both a passive and active process. It is also a hardware dependent process. Hardware is flagged with a magic value to indicate that it is operating in this system mode. A node can search available hardware looking for this value, and if it finds it, send a discovery transaction to the remote node informing it to rescan for available nodes which should find the newly activated node. In the initial implementation, discovery will not be supported. For simplicity, nodes will be informed of remote hardware by the user.

5.4 Interrupt Handling

Much of the real work of the device takes place as the result of interrupts being received. Interrupts signal transfers of page references and potentially cause memory copying to occur as the send queue becomes runnable. These manipulations often require allocation of memory and moving of potentially large quantities of data. The interrupt handlers must be fast to avoid slowing other system functions such as data acquisition or output. Allocating memory is also much more reliable if sleeping is allowed (an operation not possible from within an interrupt context). The interrupt handler in the device should simply schedule a thread to run with a process context. This allows it to schedule (permitting sleeping in memory allocation) and to take up as little processor resources as possible. The Linux `keventd` system provides a nice interface to this functionality by allowing single functions to be easily scheduled to be run.

5.5 TCP/IP Networking

Embedded systems lack many of the devices typical computers have. Nearly all storage, user and computation related I/O must be performed over the system described in this paper. Many of these can be provided via a network. Since the system in general provides for a serial communication device, layering PPP atop the serial link will provide a quick solution to providing a TCP/IP network layer. Alternatively, using the Linux generic TUN/TAP interface, a simple generic network layer can be trivially placed atop this serial link providing a link between two systems. It would also be relatively easy to design an independent network device in kernel space atop PCIComm. This would potentially provide packetizing optimizations for improved latency over the network. An advantage to this choice would be its availability early in the Linux boot process. This would allow an embedded system access to the network for a root filesystem for example.

6 Implementation Considerations

6.1 PCI Transaction Types

Many of the high bandwidth PCI transactions are performed by devices other than the CPU. The exception to this is the case in which a video card has a frame buffer of shared memory exported to the PCI bus. The common case with PCI transactions originating at the CPU is a single location memory read or write. Using this method for large quantities of data is very inefficient. Some architectures provide transaction merging capabilities for specified memory ranges. The Memory Type Range Registers [5] provided in the Intel Pentium Pro/Pentium II architecture allow for a range to be specified as "write-combining." This causes the host to combine adjacent memory transactions into a single burst transaction, thus utilizing the PCI and host busses more efficiently. The PowerPC has a system called "store-gathering" [7, 6] which isn't as robust as the Intel method, but performs roughly the same purpose. Another option, if the hardware provides for it, is to use one of the devices in the data path to perform DMA independently of the CPUs. This would require one of the host bridges, or the PCI-PCI bridge to provide this type of functionality.

6.2 Interrupt Latency & Frequency

Thought must be given to the frequency of interrupts as their processing will slow down processing on the node receiving the interrupt. The latency requirements of the interrupt handler must also be considered as this could potentially slow down the entire system instead of just a single node.

7 Initial Implementation

As much of the kernel infrastructure necessary to support the full intent of PCIComm is currently under development by other kernel developers, a full implementation of this system is not currently functional. The primary kernel element this system relies upon (kiobufs) is currently under a lot of debate among kernel developers. This system was designed with the likely final functionality of this subsystem in mind. The current functionality is a bit lacking and some workarounds were taken in this initial implementation.

Implementing PCIComm was fairly straightforward given the simple device abstractions and the runtime loadable module interfaces of the Linux kernel. There was some trouble working around the rapidly changing interfaces in the constantly under-development kernel, especially with the kiobuf infrastructure. As this was intended as only an initial implementation, an arbitrary kernel version was chosen as a baseline for the implementation (Linux 2.4.1, current as of late January, 2001). As the Linux kernel is designed primarily with C as the development language of choice, this driver was implemented in straight C following the Linux coding guidelines. The hardware independent device code totaled roughly 1200 lines of code. The simulated loopback hardware dependent module totaled under 200 lines as only a few functions are required in the hardware abstraction.

The current implementation was intended as a short proof of concept and is not useful for final benchmarking as it provides only a simple loopback hardware abstraction on a single machine. The implementation successfully transfers single pages between processes using a simple memory copy. It makes use of software interrupts (with handlers run upon return of every syscall, and periodically via the timer interrupt) to simulate real interrupts which can potentially cause lockup problems with the current process scheduling system. A better implementation would likely spawn a separate kernel thread for this processing to avoid this potential problem.

The primary limitation of this initial implementation is the limitation of page-sized I/O blocks. This is due to inefficient mechanisms of performing splitting and merging of kiobufs in their current incarnation. In the next revisions of the kiobuf subsystem, this will be better supported and the interfaces to these operations will change. This release is due out within the next two weeks and it was deemed inefficient to implement splitting and merging of the current kiobufs for anything other than the page-sized I/O required for a proof of concept. The newer kiobuf implementation will also provide for splitting and merging of kiobufs with significantly less memory allocation and copying overhead than is used in this initial implementation.

Another limitation of this initial implementation is a dependency on 32 bit systems. The generic portion of the driver supports extensions for endian byte swapping routines, but does not provide support for 64 bit addressing. This should likely be solved by placing more of the system in the hardware specific portions of the driver. Providing methods for passing whole messages to the remote side of a connection rather than simple interrupt methods.

8 Discussion

PCIComm as described is data-push with data copied out by the originator. It could also be designed to be a pull system with data copied in by the destination as local space becomes available. This would require redesigning the `kiobuf` queues, and adding several transaction types. It is also likely to degrade system efficiency as most DMA optimizations for memory transfers to the PCI bus are only applicable to write operations.

The system in its current state requires interrupts in both directions for correct and fast operation. It could be modified to perform polling to allow for operation in hardware systems where interrupts are not available. This would likely incur a latency cost much more than a bandwidth cost. Polling can be performed periodically and whenever a user operation on the driver occurs.

9 Evaluation

PCIComm attempts to minimize the number of copies necessary to move data between nodes in a cluster. It also tries to be as efficient as possible in modifying process memory tables and other memory state. Comparing to a traditional Unix read/write device, this implementation eliminates all possible memory copies. The only data copy remaining is the necessary copy over the PCI bus to move the data to a remote node. Even this can potentially be replaced with a hardware memory copy freeing up the CPU. A traditional Unix device would still have this memory copy, but in addition would require a copy from user space and to user space in the write and read system calls.

10 Conclusions

PCIComm provides an abstraction for user space to user space I/O transactions between Linux nodes in a clustering environment provided these nodes are capable of sharing memory and interrupt or otherwise passing messages to each other. A complete implementation of PCIComm is not currently possible due to the incomplete development of the kiobuf architecture. However, a partial implementation proves that it works and it is likely to be bandwidth limited primarily by the memory transfer rates across the PCI bus used to connect nodes. Latency limitations will be dependent on system load. Further implementation and testing will be necessary to fully determine the efficiency of this design once the necessary kernel infrastructure becomes available.

11 Acknowledgements

Zach Brown, Zero-Knowledge Systems for support, ideas, and pointers during the design stage of this project.

Philipp Rumpf for support and design sanity checking.

Mike Ismert, Vanu, Inc. for support and thoughts during the conception and design stages of this project.

Jacob Strauss, Vanu, Inc. for sanity checking during design and implementation.

Frans Kaashoek, MIT for guidance and supervision of this project.

References

1. P. Druschel and L.L. Peterson. "Fbufs: A high bandwidth cross-domain transfer facility." In Proc. 14th ACM Symp. on Operating System Principles, pages 189–202, 1993.

2. Anderson, E.W.: "Container Shipping: a Uniform Interface for Fast, Efficient, High-Bandwidth I/O," PhD Thesis, Computer Science and Engineering Department, University of California, San Diego, CA, USA, 1995

3. Eric Anderson, and P. Keith Muller, "Container Shipping: Operating System Support for I/O-Intensive Applications," IEEE Computer, Volume 27, Number 3, pp. 84-93, March 1994.

4. Total Impact Total Power MP Multiprocessing Cards. `http://www.totalimpact.com/G3_MP.html`

5. Memory Type Range Registers, `http://www.linuxhq.com/kernel/v2.4/doc/mtrr.txt.html`

6. MPC106 User Manual, Section 8.1.2.2 "Processor-to PCI-Write Buffers," `http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/MPC1XX/MPC106UM.pdf`

7. MPC7400 User Manual, Section 6.4.5.2 "Integer Store Gathering," `http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/MPC7XX/MPC7400UM.pdf`

8. Virtual Interface Architecture, `http://www.viarch.org/`

9. Infiniband Architecture, `http://developer.intel.com/design/servers/future_server_io/`

10. AG Electronics TPE3, "PCI I_2O Messaging," `http://www.agelectronics.co.uk/download.html`

11. Montivista Software, "Hard Hat Net," `http://www.mvista.com/products/hardhat.html`