A Tunable Communications Library for Data Injection

Craig Ulmer and Sudhakar Yalamanchili

Center for Experimental Research in Computer Systems
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
E-mail: {ulmer, sudha}@ece.gatech.edu

Abstract

A key task for providing high performance in cluster computers is efficiently transferring data between cluster resources. This study focuses on one component of the communication pipeline: the host to peripheral card interface. As Moore's Law continues to progress, we are seeing successive generations of clusters with increasing compute power and communications bandwidth, but with roughly the same I/O systems. Communication software is continuously being re-optimized for each succeeding generation of hardware.

In this paper we describe a tunable library for host-to-device communication. The library profiles performance characteristics of the host's hardware environment and utilizes this information to automatically configure host-to-device transfer mechanisms. In addition to taking advantage of CPU-specific features, the library exposes I/O characteristics of individual peripheral devices in data transfer optimizations. The benefit of the library is demonstrated by providing measurements and experiences with three generations of clusters.

1 Motivation

The availability of custom communication software, high-speed networks, and low-cost workstations has resulted in the creation of numerous cluster computers in both academic and industrial establishments. Workstations in these clusters are interconnected with high-performance communication networks so that the hardware can effectively operate as a large multi-processor computer for parallel processing applications. Communication software for such clusters has been designed to overcome I/O performance limitations of commodity workstations in order to achieve low-latency, high-bandwidth transmissions between CPUs in the systems. Additionally, modern communication libraries provide mechanisms that allow peripheral devices such as intelligent storage devices and hardware accelerators to play a more active role in the cluster computing environment [1].

An important challenge with a practical need is the tuning of communication software to operate efficiently in different hardware contexts. From our own experiences we observe two major aspects of clusters that create the need for performance tuning. First, we observe that clusters are built infrequently using state of the art resources. Therefore there is usually a significant difference in the power of host CPUs from one cluster generation to the next. For example, over the last five years Georgia Tech has assembled four Intel x86 clusters with CPUs ranging from Pentium Pros to Pentium IVs. It is desirable for our communication software to be able to exploit the hardware features of each generation of host CPU in order to improve performance. This allows users to migrate between clusters without having to re-optimize their software for each target architecture.

The second aspect of clusters pertains to peripheral devices. In the Active System Area Network (SAN) project at Georgia Tech [2] we have worked with at least six PCI based devices, all of which required custom device drivers with complex interactions between the host and device. While literature discussing PCI performance optimizations [3] is available for some peripheral devices such as Myricom's Myrinet network interface (NI) card [4], other peripheral devices have not benefited from such attention and require custom optimizations. Therefore it is useful to develop mechanisms for tuning performance that are portable across distinct PCI devices.

The work presented in this paper explores the design of a portable and tunable library for the efficient transfer of data between a host CPU and peripheral devices. This library utilizes two techniques for efficiently injecting data from the host CPU into peripheral devices. First, architectural features such as the MMX [5] and SSE [7] units of the x86 family of processors are utilized to boost programmed I/O (PIO) transfers. Second, various DMA techniques are utilized to transfer large messages. The injection library can be tuned to select the best transfer mechanism for an injection based on the amount of data to move, and the specific communication being completed. This library has currently been ported to operate in the context of three commonly used peripheral devices and is known as TPIL: the Tunable PCI Injection Library.

1.1 Data Transfers with Peripherals

Transferring data between host applications and peripheral devices is a common task in cluster computing applications. From a communication library perspective, both shared memory and message passing systems must exchange data efficiently between an application's memory space and the network interface card whenever data is transmitted between cluster nodes. The speed at which an application can move data into the network interface card has a direct effect on the overall performance of the cluster computing system. A second perspective in which host-peripheral device transaction performance is important is in the use of active peripheral devices. As intelligent storage devices (active disks) and accelerator cards (e.g., FPGAs and DSPs) become more available, it is necessary to provide mechanisms for rapidly transferring data with these peripheral devices so that they can play a more integral role in application processing.

Transferring data with peripheral devices is non-trivial for a number of reasons. First the x86 architecture does not contain a CPU-based DMA engine designed to transfer data with the I/O subsystem. Instead data is moved through either programmed I/O (PIO) memory copies or by programming DMA engines located on bus mastering PCI devices as illustrated in Figure 1. Second, transferring data from a user application requires that the address of the data be translated from a virtual to physical address. While this process is handled transparently for PIO transfers, DMA engines operate only with physical addresses. As such the kernel must translate the user's virtual address to a physical address and then pin the pages holding the data until the DMA completes. Third, for large transfers of virtual memory, data may be located in separate non-contiguous pages of physical memory. Therefore the maximum burst size of a single DMA transfer is limited in size to a page of data.

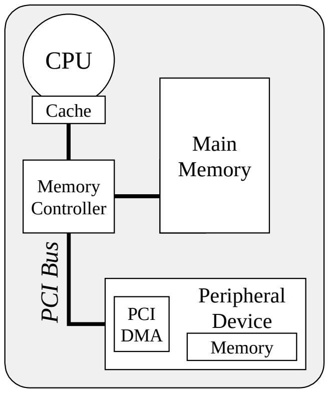

Figure 1: Modern host architecture

The library presented in this paper specializes in transfers from host applications to peripheral devices. While data must flow in both directions between host applications and peripheral devices, we note that card-to-host transfers are more easily performed than host-to-card transfers. This is because card-to-host transfers can be constructed in a push style of operation by having an on-card DMA engine push data directly into a host application's memory space. In contrast, a host-to-card transfer is more complex since the host must either move the data with PIO writes or program the destination card's DMA engines to pull the data to the card.

2 PCI Data Transfer Library

Because of the variety of host CPUs as well as peripheral devices that communication library designers utilize, it is desirable to design a common library for efficiently moving data from host to card that can be used in a number of hardware contexts. The following characteristics are critical to the usefulness of such a library:

- **Easily Tunable:** An injection library needs to be easily tuned by user programs. This allows software to be run on multiple hardware substrates that have different PCI performance characteristics.
- **Customizable:** The library needs to be customized to meet the needs of end applications. For example, for interactions with NI cards, it is desirable to provide simple mechanisms that allow injected data to be transferred to the wire as soon as the first bytes of data are copied to the card.
- **Extensible:** The library must be adapted to utilize architectural enhancements found in emerging CPU and I/O hardware.

A library that successfully addresses these goals must be equipped with multiple mechanisms for moving data across the PCI bus into peripheral devices. These mechanisms are divided into two types of operations: programmed I/O and kernel supported DMA.

2.1 Programmed I/O Transfer Mechanisms

A number of hardware features of the x86 architecture can be utilized to increase PIO memory copies for PCI transactions [6]. These hardware features include:

- **Write-Combining:** The write-combining MTRR registers included in Pentium Pro and higher processors allow stores for memory ranges to take place without strict ordering. This allows multiple writes to consecutive memory addresses to be combined for burst transfers.
- **MMX Registers:** The eight 64-bit MMX registers allow 64-byte blocks to be pulled into the CPU and then written to the I/O subsystem. Potentially this allows data writes to take place in burst operations that are efficiently mapped by the chipset into PCI transactions.
- **SSE Cache Control:** The streaming SIMD extensions (SSE) [7] unit adds features to provide user level control of the cache. In addition to pre-fetching operations, the SSE hardware provides non-temporal stores where writes can bypass cache memory and be flushed directly to memory.

Previous literature [8] has discussed the use of write-combining to improve the host-to-card performance for transfers less than a kilobyte in size. While this greatly reduces the amount of time an application spends injecting data, there are pitfalls that must be addressed. The main hazard with write-combining is that writes can be reordered in the chipset to improve burst transfer performance. For NIs this could result in a race condition where an update to a queue pointer erroneously bypasses the actual placement of data in the queue. Such hazards must be prevented through careful definitions of memory regions that perform write-combining. A second pitfall is that there are a limited number of regions that can be marked for write-combining, and that the definition of such operations is a privileged operation. In our work we resort to using write-combining only as a last resort and instead rely on the MMX and SSE features for speed improvements.

2.2 DMA Transfer Mechanisms

Modern PCI devices employ chained DMA operations for the transfer of large data sets between host and card. In chained DMA either the host or the card establishes a linked list of DMA transfers that need to be performed. When initiated the DMA engine traverses the list linearly until it reaches an end of chain marker. Most chained DMA systems allow users to specify interrupts for notification of completion of each DMA operation. Chained DMAs therefore can efficiently move large amounts of memory even if the memory is on non-contiguous pages, provided the host properly creates the linked list of DMA entries. For our work we have implemented three forms of DMA transfers in the peripheral device drivers:

- **One-Copy**: In this approach user data is copied into a large (128 KB) contiguous buffer. The card then issues a single DMA to move the data. The operation is repeated if application data exceeds the capacity of the transfer buffer.

- **Double-Buffered One-Copy**: Like the previous approach data is copied from user space to a contiguous transfer buffer in host memory. However, this approach splits the buffer in half and overlaps the transfer of data into the buffer with the DMA operation.
- **Zero-Copy**: This approach pins the pages holding user data and configures the DMA engines to transfer data directly from the user pages. While individual DMA transfers are limited to a page in size, this approach removes the need to copy data in host memory, thus greatly improving speed.

Several factors dictate which DMA operation provides the best performance. First, memory speed greatly affects the rate at which the one-copy operations can be performed. For memory controllers that cannot efficiently manage two demanding transfers concurrently, it is unlikely that the system will be able to keep the pipeline of the one-copy approach filled. A second factor affecting DMA operations is the latency for initializing the DMA transfers. If such penalties are high then it is desirable to reduce the number of DMA transfer events. Finally, we note that not all peripheral devices have chained DMA functionality. For these cards it may be beneficial to utilize the one-copy approaches due to host-card synchronization issues. We provide all three mechanisms in this work for comparison and as a means of providing different options for different environments.

3 Tunable PCI Injection Library

The tunable PCI injection library (TPIL) encapsulates both optimized PIO and DMA mechanisms for transferring data from a host to a peripheral device. A list of essential application programming interface calls is presented in Table 1. At runtime a user program must first initialize the TPIL library with configuration information before an individual peripheral device can be utilized with the library. With the *tpil_create()* function call users pass TPIL information such as the device's file handler, a pointer to a memory map of the device, the size of the memory map, and an ioctl identifier for the device's DMA operations. TPIL stores this information and returns an identifier that user applications can reference in subsequent calls to TPIL. Once initialized end users can begin using the *tpil_h2c()* function to transfer blocks of data from user space to card memory. Without additional configuration the library makes conservative estimates as to the thresholds for selecting which mechanism is utilized to transfer data to the card.

Tdev = tpil_create(	device_file_id, device_mmap, mmap_size, device_ioctl)
tpil_h2c(	Tdev, *destination, *source, number_bytes)
Tcfg = tpil_benchmark(	Tdev)
tpil_configure(	Tdev, Tcfg)

Table 1: TPIL API

In order to obtain the best results from TPIL it is necessary to tune the library for the target hardware environment. The *tpil_benchmark()* function benchmarks the hardware environment to determine how much time is required to transfer data of different sizes to the peripheral device using the PIO and DMA transfer mechanisms. Because benchmarking can take several minutes users can export measurement information and then import the information at a later time through the *tpil_configure()* command. This interface also allows users to edit configuration data so that transfer mechanisms can be explicitly specified.

3.1 Hardware Environment

The first environmental characteristic that affects TPIL is the architecture of the host CPU utilized in the target cluster computer. TPIL automatically identifies the hardware features of the host CPU and utilizes this information to determine which PIO techniques can supplement DMA mechanisms in the transfer of data to a peripheral device. The x86 clusters at Georgia Tech are based on Pentium Pro, Pentium II, Pentium III, and Pentium IV Xeon processors. Therefore the older clusters do not have MMX or SSE units to assist in PIO transfers. Likewise memory bandwidth and PCI performance is limited in all of these systems except possibly the Pentium IV Xeon cluster. Therefore we expect to see performance variations between host nodes and require that TPIL examine these differences to extract native hardware performance.

The second characteristic of TPIL is that the library must operate with peripheral devices that have different I/O characteristics. We have implemented a kernel driver and a reusable user space interface for operation with three PCI devices commonly utilized in the Active SAN project at Georgia Tech: the Celoxica RC-1000 FPGA card [9] and two different versions of Myricom's Myrinet NI. The RC-1000 card features 8MB of on-card SRAM and utilizes a PLX9080 PCI controller chip that supports chained DMA operations. The first of the two Myrinet cards is the LANai 9.1b card, featuring chained DMA engines, 64-bit/66MHz PCI, and 2MB of SRAM. The other Myrinet card is the earlier LANai 4.3 32-bit/33MHz card featuring 1MB of SRAM. The older LANai card does not have chained DMA descriptors, but does have a programmable DMA engine. We therefore constructed chained DMAs in the LANai firmware, with the firmware periodically polling DMA descriptors for requests by the host CPU to transfer data.

4 Performance

TPIL is designed to operate with the Linux 2.4 operating system and is implemented as a combination of user, kernel, and device level software. A number of tests were conducted to observe the library's performance in different hardware contexts. In the first series of tests the library was utilized to examine the performance of three different peripheral devices in a 550 MHz Pentium III system. These tests were performed using TPIL's internal benchmarking algorithms, with each transfer mechanism timed for a range of injection sizes. Between measurements the host CPU's cache is intentionally polluted so that the benchmarking favors pessimistic conditions. Each measurement is performed 32 times to remove transient effects.

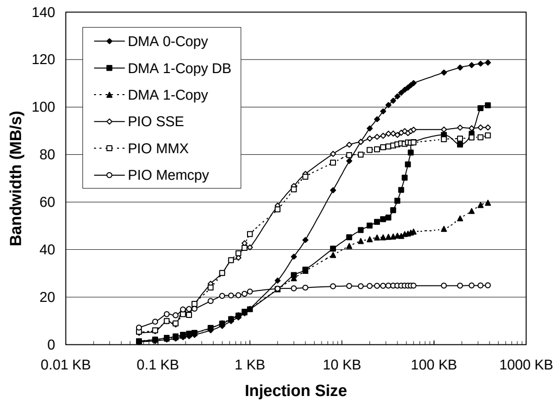

Figure 2: Myrinet LANai 4 PCI performance for Pentium III 550 MHz hosts

Figure 2 illustrates the library's performance for the Myrinet LANai 4 peripheral device. In this test we observe that the PIO transfers are beneficial for transfers up to 16 KB. MMX and SSE PIO transfers perform similarly until approximately 2 KB where the SSE's non-temporal stores and pre-fetching operations have a slight advantage. DMA operations are utilized in transfers larger than 16 KB. Due to limited host memory bandwidth the zero-copy approach is always more efficient than the one-copy techniques. The Myrinet LANai 9 benchmarks yielded similar results to the LANai 4, with slightly better DMA performance.

The benchmarking experiment was performed for the Celoxica RC-1000 card in the same host system, with results depicted in Figure 3. As expected MMX and SSE PIO based transfers provided the best performance for injections less than 3 KB. However, this performance drops to a steady state value of 30 MB/s for transfers larger than 2 KB. We observe that the PCI characteristics of this card may affect performance since the card's PCI chipset has a limited capacity for incoming data from the PCI bus. PIO transfers can therefore be slowed if the host CPU saturates these PCI buffers. For transfers larger than 3 KB the card's DMA engine operates efficiently since the engine pulls data into the card as buffer space allows. The zero-copy approach outperformed other DMA transfer mechanisms.

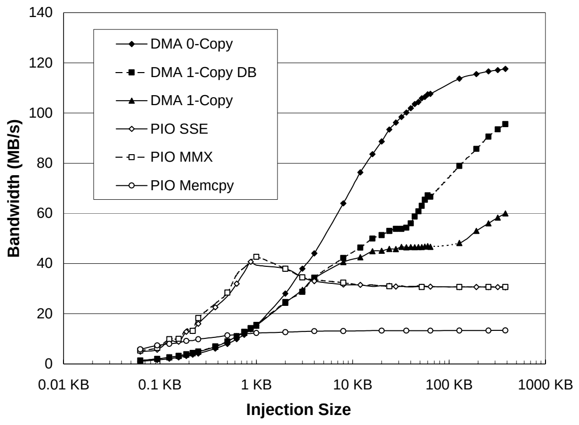

Figure 3: RC-1000 PCI injection performance f or Pentium III 550MHz hosts

A second set of tests was performed to examine how TPIL performed in different host systems. For this work we utilized the Myrinet LANai 9 card due to its ability to deliver good PCI performance. Figure 4 illustrates the performance results of TPIL in three different cluster computer hosts: a 200 MHz Pentium Pro, a 550 MHz Pentium III, and a 1.7 GHz Pentium IV Xeon with 32-bit and 64-bit PCI. Each curve in the figure traces the amount of bandwidth TPIL can obtain for a host using the different transfer techniques discussed in this paper. The Pentium Pro system delivered only 20 MB/s due to its poor PCI implementation. For the Pentium III system the library effectively switched between transfer mechanisms as reported in the previous set of measurements.

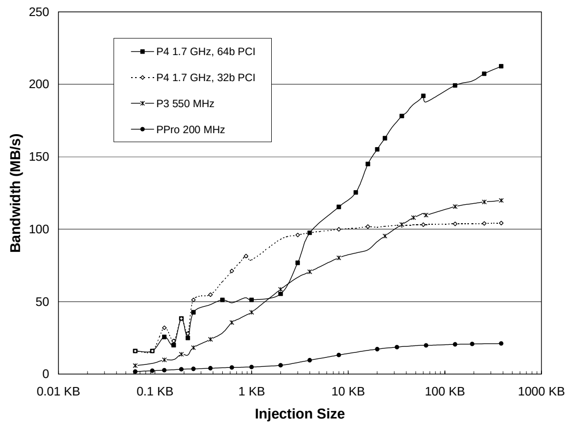

Figure 4: Myrinet LANai 9 PCI injection performance for multiple hosts

The Pentium IV host exhibited unusual performance characteristics for both 32-bit and 64-bit PCI slots. The motherboard is based on the Intel 860 chipset, which has known performance issues. Initially, performance was limited to 140 MB/s until a patch was applied to the chipset to increase PCI buffering [10]. Figure 5 depicts the performance of the more dominant transfer mechanisms for this host. For 32-bit PCI slots, maximum DMA performance was limited to 90 MB/s, resulting in SSE transfers being selected for all injection sizes. For 64-bit slots, MMX and SSE transfers reached an early plateau of 53 MB/s for injections larger than 512 bytes. Interestingly, zero-copy and double-buffered one-copy approaches alternated as the most efficient mechanism for transfers larger than 2 KB to a 64-bit PCI card. The competitiveness of the double-buffered one-copy approach can be attributed to the high memory bandwidth available in the RDRAM-based Pentium IV.

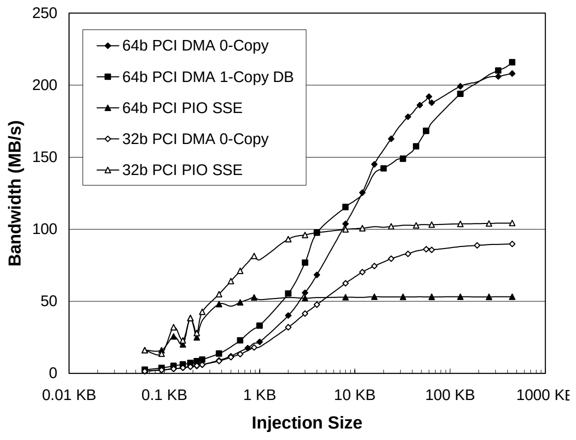

Figure 5: Myrinet LANai 9 PCI injection performance for Intel 860-based Pentium IV

5 Conclusions and Future Work

For institutions with multiple clusters it is important to provide mechanisms that allow applications to be tuned automatically to the characteristics of the underlying hardware. TPIL is a tunable library that accomplishes this in the communication library task of injecting data from the host into peripheral devices. We have demonstrated that this library automatically extracts native performance benefits on different cluster computing platforms, and that it is portable to multiple peripheral devices. As a benefit of benchmarking its hardware environment, TPIL is able to uncover basic performance bottlenecks, such as the saturation of the RC-1000's PCI buffers.

The current implementation of TPIL is open source and available for academic use. We are currently adapting the library to operate with the Intel IXP1200 network processor [11]. While this NI features sophisticated hardware for accelerating network operations, the host cannot directly control the card's DMA engines. For use with TPIL this requires a DMA management agent be implemented in card firmware. This work is similar to what was constructed earlier for the LANai 4 processor. In addition to adapting new peripheral devices, future TPIL work will also include the examination of other CPU-specific optimizations that can be utilized to improve host-to-card transfer performance.

6 Acknowledgements

We thank Ken Mackenzie and Ivan Ganev at Georgia Tech for their assistance and suggestions in this work. We gratefully acknowledge the support for this research by a grant from the Advanced Networking Infrastructure and Research Division of National Science Foundation under grant ANI-9876573, a grant from the Xilinx Corporation, and hardware donations from the Intel Corporation.

References

[1] C. Ulmer and S. Yalamanchili. A Messaging Layer for Heterogeneous Endpoints in Resource Rich Clusters. In *Proceedings of the First Myrinet User Group Conference*, 2000.

[2] C. Ulmer, C. Wood, and S. Yalamanchili. Active SANs: Hardware Support for Integrating Computation and Communication. In *Proceedings of the Workshop on Novel Uses of System Area Networks*, 2002.

[3] K. Yocum, J. Chase, A. Gallatin, and A. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low Latency Messaging. In *Proceedings of IEEE Symposium on High-Performance Distributed Computing*, 1997.

[4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. In *IEEE Micro*, Vol.15, No.1, 1995.

[5] M. Mittal, A. Peleg, and U. Weiser. MMX Technology Architecture Overview. In *Intel Technology Journal*, Q3, 1997.

[6] Intel Corporation. Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual. Intel document 248966-04, 2001.

[7] S. Thakker and T. Huff. The Internet Streaming SIMD Extensions. In *Intel Technology Journal*, Q2, 1999.

[8] R. Bhoedjang, T. Ruhl, and H. Bal. User-Level Network Interface Protocols. In *IEEE Computer*, Vol.31, No.11, P53-60, 1998.

[9] Embedded Solutions, Ltd. RC1000-PP Hardware Reference Manual. Product data sheet, 1999.

[10] Myricom. Myrinet Frequently Asked Questions. http://www.myri.com.

[11] Intel. IXP1200 Network Processor Datasheet. December 2001.