

# *Reducing Data Transfer Time in User-Level Network Protocols*

Chulho Won and Jong-Hoon Youn

Department of Computer Science  
University of Nebraska at Omaha  
Omaha, NE 68182

**Abstract** - Emerging parallel applications require a significant improvement in communication latency. Since the time required for transferring data between host memory and network interface (NI) takes up a large portion of overall communication latency, the reduction of data transfer time is crucial for achieving low-latency communication. In this paper, a new data transfer mechanism - called hereafter the DT, is proposed to reduce communication latency for latency demanding applications. The DT employs a cache coherence interface hardware to utilize an eager method for transferring data between the host and NI. Our simulation results show that the DT significantly reduces message latency up to 36 % compared to a Direct Memory Access (DMA) based approach.

**Key Words:** *Data transfer, eager method, user-level, cache coherence, low-latency, network protocols, message.*

## **1. Introduction**

Due to the fast evolving network and processor technology, cluster computer systems have become the most cost-effective platform for parallel and distributed computing. As the popularity of cluster computing grows, there is an increasing demand for low-latency network protocol and intelligent network interface hardware. Since the performance of parallel and distributed applications is greatly dependent on message passing facility, low-latency message processing becomes a main design issue for cluster network protocols and network interface (NI). User-level network protocols such as *Virtual Interface Architecture (VIA)* [4] and *InfiniBand Architecture (IBA)* [12]

significantly reduce user-to-user communication latency compared with legacy network protocols (e.g., TCP/UDP/IP). User-level protocols execute time-critical operations, such as message send and receive, without the kernel involvement, and implement zero-copy data transfer to avoid data copy overhead. Although those recent development efforts have been successful in lowering communication latency, the demand for low-latency communication is still high. Therefore, this paper was motivated to investigate opportunities for further reducing communication latency using special network interface hardware. The previous study [18] on the communication performance of VIA showed that the data transfer time between the host and NI constitutes a significant portion of the overall communication latency.

For example, VIA uses a series of events during a message send described as follows. First, the user first prepares a message in the host memory and notifies the NI of a message send request. Second, the NI copies the user message from host memory to NI buffer. Finally, the network transport protocol starts to send out the message into the network. There is a large time gap between the first step (when the user message is prepared) and the second (when it is copied into the NI buffer). Therefore, an idea is to execute simultaneously those two time-consuming operations: preparation of the user message and copying of the user message.

This paper proposes a hardware-based data transfer mechanism, which is called hereafter DT, to attain the reduction of data transfer time in the execution of communication primitives such as send and receive. The DT employs a cache-coherence interface hardware to efficiently transfer data between the host and NI.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes a model of user-level network protocol and its data transfer mechanism. Section 4 presents the proposed data transfer mechanism. Section 5 explains the performance evaluation. Section 6 presents conclusions.

## 2. Related Work

Banikazemi showed that PIO is faster than DMA for transferring small size data (16 bytes or less) [2]. Bhoedjang presented a comparison of data-transfer performance between DMA and PIO [19], which shows that PIO using Pentium-Pro™ write combining buffers moves data faster than DMA for data up to 1024 bytes because of the DMA start-up cost. Mukherjee proposed Cacheable Network Interface (CNI), which allows a coherent, cacheable memory block implemented as a Cacheable Device Register (CDR) to be shared between the host processor and NI [10], [13], [18]. They compared PIO and CNI to show that CNI has better performance than PIO [18].

## 3. A User-Level Network Protocol model

User-level protocols were motivated to overcome the shortcomings of the legacy network protocols such as TCP/IP and UDP/IP. They avoid the kernel involvement for time-critical communication services, such as message send/receive, and thus allow user applications to directly access the network. Popular user-level protocols include Virtual Interface Architecture (VIA) and InfiniBand Architecture (IBA) [4], [13]. Since VIA architecture was developed as a industry standard and have influenced many of recently developed user-level protocols, our model of user-level protocol shares main features with VIA. This section briefly describes the model and a typical sequence of operations for a data transfer.

As shown in Figure 1, the model includes several key components: Send Queues (SQ), Receive Queues (RQ), Completion Queues (CQ), Notification Registers, and Application Programming Interface Library. User program issues send or receive message requests through the Application Programming Interface Library, which includes

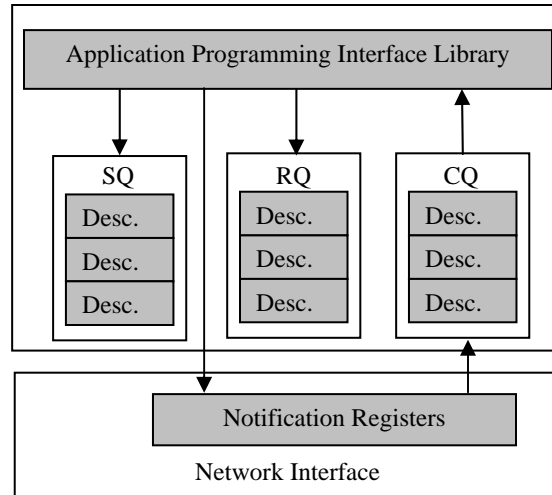


Figure 1: A User-Level Network Protocol

message send and receive primitives. A queue pair SQ and RQ is the communication end point that allows applications to submit message requests directly to the communication facility running on the NI hardware. User applications write request into the queues in the form of descriptors, which is shown Desc in Figure 1. Descriptor is a data structure that contains all the information needed to process the user request including the address of the user buffer. It also contains the address of the user buffer at the destination node.

Descriptor posting is followed by writing a token to the Notification Register, which notifies the NI to process the descriptor. When a user request completes, the associated descriptor in the work queue is updated with a status value, and a notification is inserted into the CQ. Applications can check the completion status of their message request via CQ.

It is assumed that the memory regions for holding the user messages and their descriptors are pinned down through a registration process. The purpose of the memory registration is to fix the user's virtual memory in physical memory so that it avoids memory swap operations. Even though user-level protocols allow different types of communication primitives such as the send/receive messaging and remote direct memory access model, this paper focuses only on the send/receive messaging.

A number of steps are executed for a message

send or receive. In order to perform a send operation, the sender builds a message (**Send1**) and a descriptor (**Send2**) in the registered memory regions. The descriptor includes the address of the message buffer, message size, type of operation, and a status field. Then, a token is written to the send Notification Register to notify the NI of a message send operation (**Send3**). The send Notification Register token includes the address of the descriptor, which in turn holds the address of the user message. Since the address of the user message is contained in the descriptor, the NI reads in the descriptor (**Send4**) and the user message (**Send5**). Finally, after the message is sent out to the network (**Message Send**), the completion status is set in the CQ (**Send6**).

Since a receiver needs to post a descriptor before a message is sent, a message receive is performed in two separate sequences: Posting a descriptor and receiving a message. A descriptor posting for receive follows the same steps as in the send case. The receiver builds a descriptor (**Recv1**) and writes (**Recv2**) to the receive Notification Register, which is followed by the transfer of the descriptor (**Recv3**). The remaining steps are executed when a message arrives (**Message Receive**). The NI moves the message into the registered memory region, which is pointed to by the address held in the descriptor (**Recv4**), and sets the completion status in the CQ (**Recv5**). The receiver polls the CQ to detect a new message arrival (**Recv6**) and reads the message from the registered memory region (**Recv7**).

## 4. Proposed Data Transfer Mechanism

In the earlier description of a series of operations for a message send, there is a time interval between **Send1** and **Send5**. Since this time accounts for a significant portion of the overall latency, the DT executes those time-consuming operations simultaneously. The DT mechanism allows to move data in an eager method. Therefore, as the application builds the message in the user buffer, the message is copied into the NI buffer. In this way, **Send1** and **Send5** are executed at the same time.

Figure 2 shows the architecture of the DT consisting of tag memory and local buffer, which

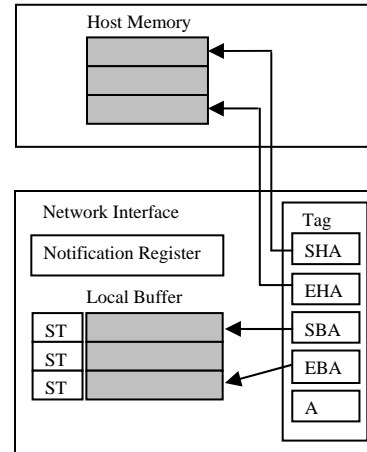


Figure 2: DT Architecture

is also called NI buffer. In the following descriptions, we use DT and NI interchangeably. During the registration of a host memory region, an associated local buffer and a tag entry are allocated. The allocated buffer is the same size as the message buffer in the host memory and is subdivided into memory blocks, where each block is equal to the cache line size. Each memory block has a Status bit (ST) indicating the state of the cache block. Each tag entry consists of five fields: Start and End Host Addresses (SHA and EHA), Start and End Buffer Addresses (SBA and EBA), and Access bit (A). The host address pair, SHA and EHA, points to a registered memory region in the host memory. The buffer address pair, SBA and EBA, points to the associated buffer in the local memory. A-bit is used to represent the validity of the tag. The tags are used for monitoring access attempts to the registered regions. The tags are also used to associate a registered host memory region with a local buffer. In addition, the NI buffer uses status bit per cache block to indicate the modification of the data. Using the memory association information, memory updates on a host memory region are reflected on the associated NI buffer.

To accomplish the simultaneous execution, DT employs a simple cache coherence protocol, which follows the underlying cache coherence protocol. In this paper, it is assumed the host uses MESI cache protocol [19]. The cache coherence protocol uses *Modified* (M), *Shared* (S), and *Invalid* (I) states, which mean the line is modified,

shared, and is not valid, respectively.

Once the Tag fields are properly set, the cache lines for the registered memory regions are initialized to *Shared* (S) state. This step is needed to allow DT to monitor the attempts of writing to the registered memory regions by the host processor.

A brief description of the cache coherence protocol is presented for a message send. When the host processor writes to the registered memory region (*Send1*), there can be a cache hit or miss. If a processor write hits on the cache, an invalidation bus transaction is generated to gain exclusive ownership of the cache line. On the other hand, if a processor write misses on the cache, the cache line is first loaded from the host memory and then an invalidation bus transaction is generated. In either case, the cache line and the memory block in the NI buffer transition to the *Modified* (M) and *Invalid* (I) states, respectively. Once the DT detects the invalidation bus transaction, the corresponding block in the NI buffer changes the state from *Shared* (S) to *Invalid* (I). At this point, DT places a bus read request to read in the cache block modified by the host processor. This causes the requested block to be transferred to the NI local memory, and both the cache blocks in the host processor and the local memory transition to the *Shared* (S) state. As the host processor continues to write to the registered memory region, memory blocks in the local buffer are invalidated and the updated cache lines are read into the buffer.

For the message receive case, the host processor read operations do not begin until the message is completely received. Therefore, the data transfer phase cannot be overlapped with the host processor reads, unlike the case for message send. However, the use of DT mechanism can still avoid the use of DMA and thus eliminate DMA startup and interrupt processing overhead, which results in a small performance gain.

The operations performed by DT for message receive are described below. An additional state *Modified* (M) is used to indicate a message has been received from the network. As soon as a message is received to the local buffer, invalidations are generated to purge the cache lines in the host cache. The completion of the invalidation transactions causes the buffer blocks to change from *Shared* (S) state (the initial state) to *Modified* (M) state. Once the message is moved to the NI

Table 1: Simulation Parameters

Simulation Parameters		
Host Processor	L1 cache	2-way set, 32B/line, 32KB, 1cycle/hit
	L2 Cache	2-way set, 128B/line, 1MB, 10 cycles/hit
	CCP	MESI
Host Memory & Bus	Access Latency	100 cycles
	Data Width	16 B/access
	Max. Transfer Rate	1600 MB/sec
Network Interface (DT)	Local Buffer Access	10 cycles
	Host Memory Access (Max. Rate)	1600 MB/sec

buffer, it notifies the receiver process that a new message has arrived. After the notification, the user process attempts to read the message by placing read requests on the bus. This causes the blocks to be copied to the host processor cache and those cache blocks both in the host processor cache and NI buffer are changed to *Shared* (S) state.

## 5. Performance Evaluation

### 5.1 Simulation Environment

A system simulation environment [15] was used for the evaluation of DT. To set up a full system evaluation environment, a network protocol and benchmark programs were written and executed on top of the simulation environment. The system configuration and parameters are summarized in Table 1.

The host processor model includes L1 and L2 caches and follows the MESI cache coherence protocol to maintain data consistency between cache and memory. The DT has a data transfer speed comparable to the maximum host memory bandwidth because it is connected to the host

memory bus. Since our simulation study focuses only on network protocol processing within a node, our simulation results are based on a no-delay network model.

For performance evaluation, the DT mechanism was compared against a DMA mechanism. A micro-benchmark was used to send and receive messages between two users on different hosts. A sender sends a fixed-size message to the receiver and then waits for a message arrive from the receiver. When the receiver receives a message, it sends a new message back to the original sender. Messages are sent back and forth between sender and receiver for a number of times. To show the impact of data transfer mechanism, message send/receive time was measured as a function of message size.

### 5.2 Simulation Results

The total execution times for message communication between two user applications are presented in Figure 3. The execution time represents the number of host processor cycles between a sender call and the corresponding receiver call. These results do not include the effects of MAC, physical layer operations, and network transfer time. The message size was varied from 64 bytes to 512 bytes, which represent a typical range of message sizes for a SAN environment [13]. Figure 7 shows that the proposed DT mechanism results in much better performance compared to the DMA. The DT attains 34% to 36% reduction in the user-to-user messaging latency compared to the DMA.

The pie charts shown in Figure 4 give a more detail breakdown of the communication latency for 256-byte message. The total execution times are divided into three most significant operations: User Data, Data Transfer, and Transport. The User Data represents the time required for the user program to write a message to the host memory. The Data Transfer includes the time to transfer data between host and the local buffer in the DT (NI). The Transport represents the execution time of the network protocol to service the user send/receive requests. For DMA, both the User Data time and Transport time take significant portions of the total execution time because those two operations are executed sequentially. For DT, in contrast, the Transport takes a smaller portion of the

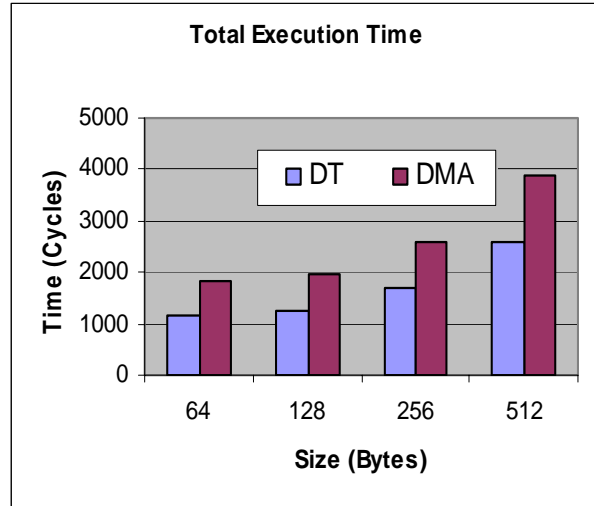


Figure 3: Total execution time

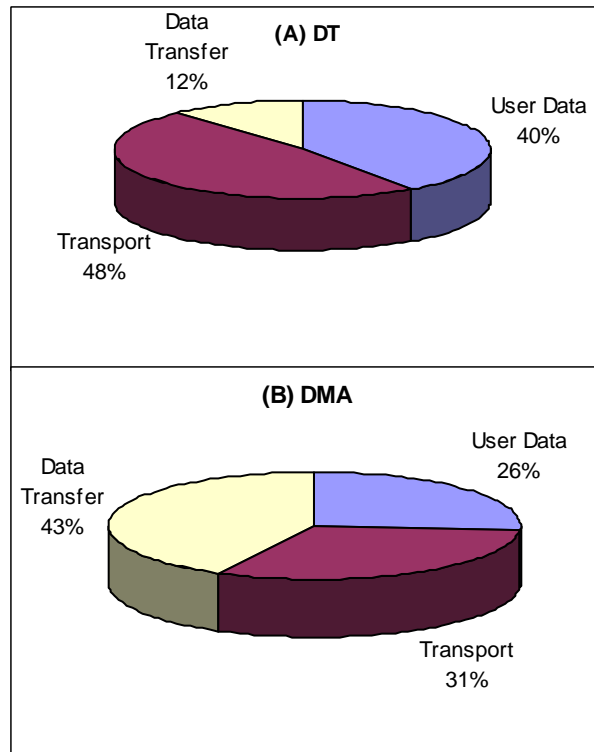


Figure 4: A breakdown of total execution time

total execution time because the Transport and the User Data are executed simultaneously. The charts clearly shows that DT significantly reduce the communication latency by simultaneously executing data transfer.

## 6. Conclusions

This paper proposed a new mechanism called DT to reduce communication latency for user-level network protocols. The DT reduces data transfer time between host memory and NI by overlapping writes to the user buffer with the actual transfer of user data from user buffer to NI buffer. Our detailed simulation study showed that the DT reduces the message latency by 36% compared with the DMA.

## References

- [1] A. Begel *et al.*, "An Analysis of VI Architecture Primitives in Support of Parallel and Distributed Communication," *Concurrency and Computation: Practice and Experience* (14) 1, 2002.
- [2] M. Banikazemi *et al.*, "Design Alternatives for Virtual Interface Architecture (VIA) and an Implementation on IBM Netfinity NT Clusters," *Proceedings of the International Parallel and Distributed Processing Symposium*, 2000.
- [3] NERSC, "M-VIA: A High Performance Modular VIA for Linux," Available from <http://www.nersc.gov/research/FTG/via>.
- [4] Intel, Compaq and Microsoft Corporations, "Virtual Interface Architecture Specification, Version 1.0," Available at <http://www.viarch.org>.
- [5] P. Buonadonna, A. Geweke, and D.E. Culler, "Implementation and Analysis of the Virtual Interface Architecture," *Proceedings of Supercomputing (SC'98)*, November 1998.
- [6] J. R. Thorpe, "A Machine Independent DMA Framework for NetBSD," *Proceedings of USENIX 1998 Annual Technical Conference*, June 15-19, 1998.
- [7] N.J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, 15 (1):29-36, 1995.
- [8] Dave Dunning *et al.*, "The Virtual Interface Architecture," *IEEE Micro*, 18(2):66-75, 1998.
- [9] H. Hellwagner, "Exploring the Performance of VI Architecture Communication Features in the Giganet Cluster LAN," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)*, 2000.
- [10] S.S. Mukherjee and M.D. Hill, "Making Network Interfaces Less Peripheral," *IEEE Computer*, 31(10):70-76, 1998.
- [11] IBM Corporation, "IBM InfiniBlue solutions," Available at <http://www3.ibm.com/chips/products/InfiniBand>.
- [12] Infiniband Trade Association, "Infiniband Architecture Specification, Vol. 1," InfiniBand Trade Association, Available at <http://www.infinibandta.org>.
- [13] S.S. Mukherjee *et al.*, "Coherent network Interfaces for Fine-Grain Communication," *Proceedings of the 23<sup>rd</sup> International Symposium on Computer Architecture (ISCA)*, 1996.
- [14] R.A.F. Bhoedjang *et al.*, "Reducing Data and Control Transfer Overhead through Network-Interface Support," *Proceedings of the First Myrinet User Group Conference (MUG)*, September 2000.
- [15] C. Won *et al.*, "Linux/SimOS - A Simulation Environment for Evaluating High-Speed Communication Systems," *Proceedings of the 2002 international Conference on Parallel Processing (ICPP)*, 2002.
- [16] Intel Corporation, "Pentium(R) Processor Family Developer's Manual," Available at <http://developer.intel.com/design/intarch/manuals/241428.htm>.
- [17] S. Pakin *et al.*, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of Supercomputing '95*, December 1995.
- [18] S. Mukherjee *et al.*, "The impact of Data Transfer and Buffering Alternatives on Network Interface Design," *Proceedings of the 4<sup>th</sup> International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 1998.
- [19] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal, "Design Issues for User-Level Network Interface Protocols on Myrinet," *IEEE Computer*, 31(11):53-60, 1998.