DOI : 10.3745/JIPS.2008.4.4.133

Eager Data Transfer Mechanism for Reducing Communication Latency in User-Level Network Protocols

Chulho Won*, Ben Lee, Kyoung Park*** and Myung-Joon Kim*****

Abstract: Clusters have become a popular alternative for building high-performance parallel computing systems. Today's high-performance system area network (SAN) protocols such as VIA and IBA significantly reduce user-to-user communication latency by implementing protocol stacks outside of operating system kernel. However, emerging parallel applications require a significant improvement in communication latency. Since the time required for transferring data between host memory and network interface (NI) make up a large portion of overall communication latency, the reduction of data transfer time is crucial for achieving low-latency communication. In this paper, Eager Data Transfer (EDT) mechanism is proposed to reduce the time for data transfers between the host and network interface. The EDT employs cache coherence interface hardware to directly transfer data between the host and NI. An EDT-based network interface was modeled and simulated on the Linux-based, complete system simulation environment, Linux/SimOS. Our simulation results show that the EDT approach significantly reduces the data transfer time compared to DMA-based approaches. The EDT-based NI attains 17% to 38% reduction in user-to-user message time compared to the cache-coherent DMA-based NIs for a range of message sizes (64 bytes ~ 4 Kbytes) in a SAN environment.

Keywords: *Data Transfer, Cache Coherence, User-Level, Low-Latency, Network Protocols, Message, VIA*

1. Introduction

Due to the rapid improvements in network and processor performance, cluster computer systems have become the most cost-effective platform for parallel and distributed computing. Advances in technology are closing the performance gap between dedicated parallel computers and cluster computers. 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) [1] and *InfiniBand Architecture* (IBA) [4] significantly reduce user-to-user communication latency compared with traditional 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 the user-level protocols have been successful in lowering communication latency, the demand for even lower communication latency remains high.

Our prior study on the communication performance of VIA showed that the data transfer time between the host and NI constitutes the largest portion of the overall communication latency [10]. Therefore, the reduction of user data transfer time significantly improves communication performance. There are several alternatives for transferring data between the host and NI: Programmed IO (PIO), Direct Memory Access (DMA), Cache-coherent DMA (CC-DMA), and Coherent Network Interface (CNI). *PIO* is a traditional method to access device registers on an NI residing on the I/O bus using uncached loads/stores. Uncached accesses transfer one to eight bytes at a time, which typically results in more bus transactions than using DMA [7, 8]. *DMA* moves data over the memory bus in block transfer mode, which efficiently utilizes the memory bus bandwidth. *CC-DMA* is an advanced form of DMA that does not require the cache to be explicitly flushed before a DMA operation. This is done by using a special logic to detect accesses to memory locations for which there are dirty cache blocks, and either allowing data to be accessed directly from cache or implicitly flushing cache blocks before the data can be accessed from the main memory [8, 19]. *CNI* allows data transfer between NI devices registers and cache memory by relying on the underlying cache coherence protocol [5, 6]. It effectively

Manuscript received 30 September, 2008; accepted 13 October, 2008.
Corresponding Author: Ben Lee
* Electrical and Computer Engineering Dept., California State University, Fresno, CA, USA (chwon@csufresno.edu)
** School of Electrical Engineering and Computer Science, Oregon State University, OR, USA (benl@eecs.oregonstate.edu)
*** Electronics and Telecommunications Research Institute, Daejeon, Korea ({kyoung, joonkim}@etri.re.kr)

uses the bus bandwidth by transferring data in cache-block units and cache invalidations are used as an efficient event-notification mechanism.

Despite these existing data transfer mechanisms, there is an opportunity to further reduce the data transfer time between the host and NI, especially in the context of low-latency, user-level network protocols such as VIA and IBA. In order to understand the opportunity for improvement, consider a message send in VIA. The user first prepares a message in the host memory and notifies the NI of a message send request. The NI copies the user message from host memory to NI buffer and then starts the network transport protocol to inject the message into the network. As can be seen by these steps, a large time gap exists between when the user message is prepared and when it is copied into the NI buffer. Therefore, overlapping the user message preparation and the copying of user message can significantly reduce the overall latency.

This paper proposes a hardware-based speculative approach called *Eager Data Transfer* (EDT) to reduce the data transfer time. EDT employs cache-coherence interface hardware to efficiently transfer data between the host and NI. Since EDT relies on underlying cache-coherence mechanism, it is assumed that the EDT-based NI is located on host memory bus to observe bus transactions (i.e., cache coherence-related bus transfers). The two performance advantages of EDT are (1) efficient use of memory bus bandwidth since data transfers are done in cache block units, and (2) no software overhead since the data transfer process is completely controlled by hardware. In order to evaluate the effectiveness of EDT, an EDT-based NI was modeled and simulated on Linux/SimOS [10], which is a Linux operating system port to a complete system simulator SimOS [14].

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 overviews VIA and its basic data transfer mechanism. Section 4 presents the proposed EDT mechanism. A detailed comparison of EDT versus CNI is presented in Section 5. Section 6 describes a VIA implementation for EDT-based NI. Section 7 presents the simulation results. Finally, Section 8 concludes the paper and discusses future work.

2. Related Work

Banikazemi *et al.* showed that PIO is faster than DMA for transferring small size data (16 bytes or less) [2]. Bhoedjang *et al.* also presented a comparison of data transfer performance between DMA and PIO [8]. Their results shown that PIO using Pentium-Pro™ write combining buffers, which combines multiple write commands over the I/O bus into a single bus transaction, transfers data faster than DMA for data size up to 1,024 bytes because of the DMA start-up cost. However, DMA is more efficient than PIO for large data because data transfers can be performed in bursts and without disturbing the processor.

Since DMA transfers data to and from host memory, both cache and memory must be coherent before a DMA transfer can start. For systems that do not support cache coherent DMA, software is used to explicitly flush the dirty blocks from cache to memory using cache flush instructions, such as cache (MIPS) [16], dcbf (PowerPC) [17], and wbinvd (Intel) [18]. For systems that support cache coherent DMA, there are two methods for maintaining coherency between cache and memory using hardware. The first option is to suspend the current DMA transaction until the cache sends the dirty cache block to the host memory [15]. This method, referred to as *cache coherent DMA with retry*, forces DMA bus request to be retried whenever the requested data is not in the host memory. When the dirty cache block is written back to the host memory, the DMA request is tried again and the data is then read from the host memory. The second option, called *cache coherent DMA with intervention*, is based on using a cache coherent bus [19]. The idea is to have the cache snoop the cache coherent bus and whenever there is a request for a dirty cache block, the cache supplies the requested data. The main advantage of cache coherent DMA with intervention is that the DMA accesses the requested data without having to wait until the cache block is flushed to the host memory.

The work closest to ours is *Coherent Network Interface* (CNI), which was proposed by Mukherjee *et al*. [5-7]. CNI allows a coherent, cacheable memory block implemented as a *Cacheable Device Register* (CDR) to be shared between the host processor and NI. CNI reduces unnecessary bus accesses by transparently transferring data between the host processor and NI in cache blocks rather than words. *Cacheable Queue* is an extension of CDR to represent a contiguous region of memory blocks and is managed by a pair of head and tail pointers. For example, the host processor sends a data by simply writing it to the next free queue entry and incrementing the tail pointer. Then, the cache coherence protocol invalidates the copy of the tail pointer in the NI, which causes the NI to initiate a read request for the block. Mukherjee *et al.* [7] compared regular PIO (i.e., without write combining buffers) and CNI, and showed performance improvement over PIO by 17~53%.

The major advantage of EDT over CNI is the random access capability, which is in contrast to CNI's cacheable queues that require the construction of arrays to be

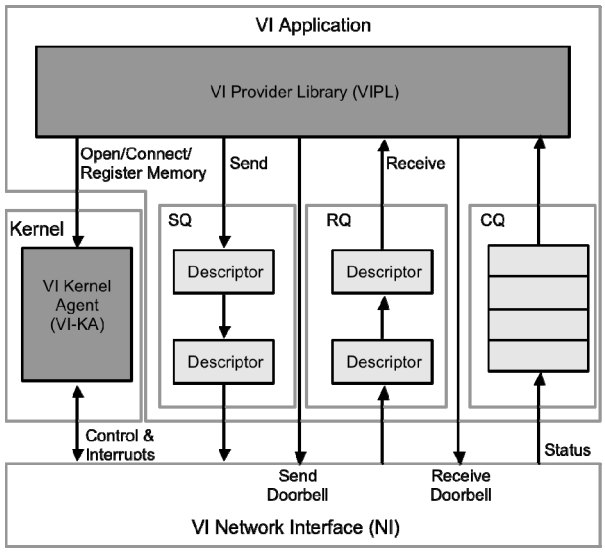


Fig. 1. The VIA architecture.

performed in strict sequential order thereby restricting the programming paradigm. A more detailed comparison between CNI and the proposed EDT mechanism is presented in Section 5.

3. Overview of VI Architecture

Virtual Interface Architecture(VIA) is an industry standard developed by Compaq, Intel, and Microsoft [1]. VIA was designed to provide low-latency, user-level communication over a System Area Network (SAN). Unlike the legacy network protocols of TCP/IP and UDP/IP, which were developed to operate in Wide Area Network or the Internet, VIA is a light-weight protocol that avoids the kernel involvement for time-critical communication services, such as message send/receive, and thus allows user applications to directly access the network. This section briefly describes an implementation of VIA and how data transfers are performed between the host and NI.

Fig. 1 shows the basic components of VIA: Send Queues (SQ) and Receive Queues (RQ), Completion Queues (CQ), VI Network Interface (NI), VI Kernel Agent (VI-KA), VI Provider Library (VIPL), and VI Application [1]. A *work queue pair* SQ and RQ composes a *Virtual Interface* (VI), which is the communication end point that allows an application to submit message requests directly to the communication facility running on the NI hardware. A user application posts requests on the queues in the form of descriptors. A *descriptor* is a data structure that contains all the information needed to process the user request. Each descriptor contains one control segment followed by an optional address segment and zero or more data segments. Each data segment contains the virtual address of the user buffer. The address segment contains the virtual address of the user buffer at the destination node.

Each VI is associated with send and receive *doorbell registers*. A descriptor posting is followed by writing a token to the doorbell 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 either the descriptor or CQ. Thus, CQ merges the completion status of multiple work queues.

The VIA specification requires that a user application register the virtual memory regions that are used to hold VI descriptors, user communication buffers, and CQs. The purpose of the memory registration is to have the VIPL pin down the user's virtual memory in physical memory so that the NI can directly access the user buffers. This eliminates the need to copy data between user buffers and intermediate kernel buffers typically required in traditional network protocols. VIA specifies two types of data communications: the send/receive messaging model and remote direct memory access (RDMA) model. This paper focuses only on the send/receive messaging of VIA.

The detailed operations for VIA message send and receive are shown in Fig. 2, where it is assumed a DMA engine is used to transfer user data between the host memory region and NI. 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 doorbell register to notify the NI of a message send operation (Send3). The doorbell 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 executes a DMA transfer for the descriptor (Send4), followed by another DMA transfer for the message (Send5). After the message is sent out to the network (Message Send), the completion status is set in the CQ (Send6). Finally, user application can check the completion status of the send operation (Send7).

According to the VIA specification, the receiver is required to post a descriptor before a message is sent. Thus, 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 a token (Recv2) to the receive doorbell register, which is followed by DMA transfer of the descriptor (Recv3). The remaining steps are executed when a message arrives (Message

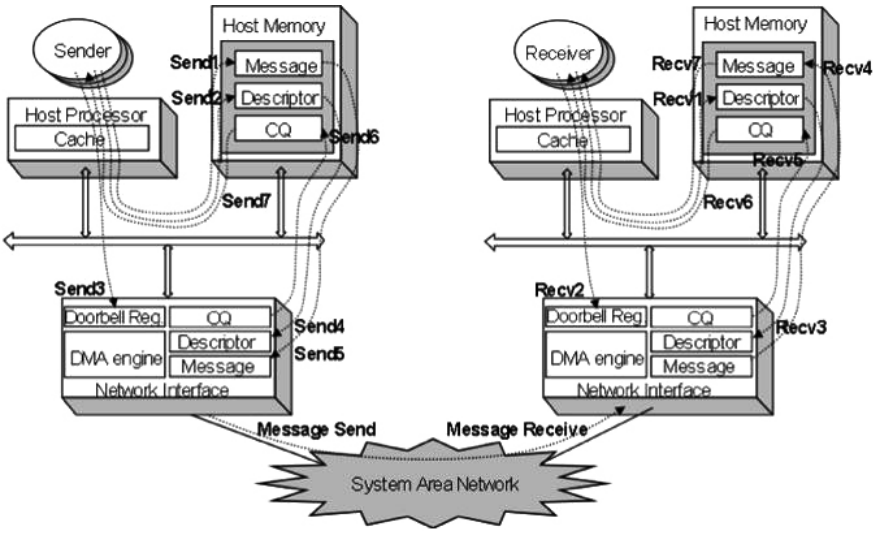


Fig. 2. Message send and receive.

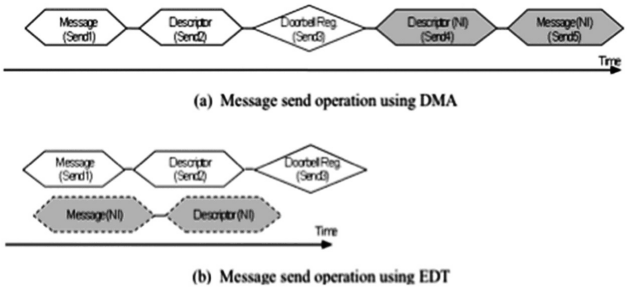


Fig. 3. EDT vs. DMA-based data transfer.

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).

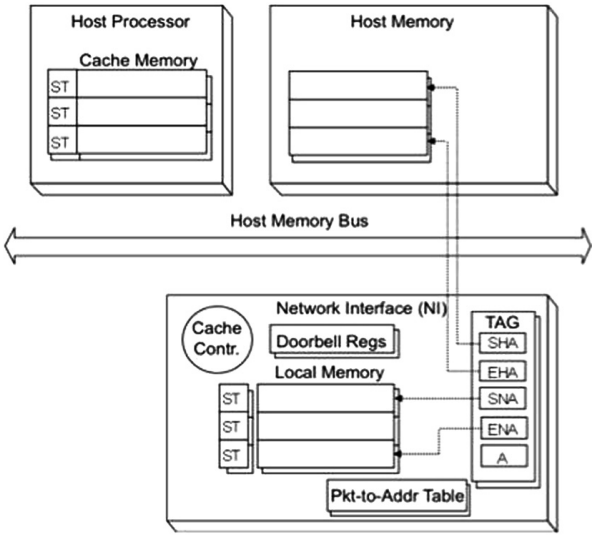
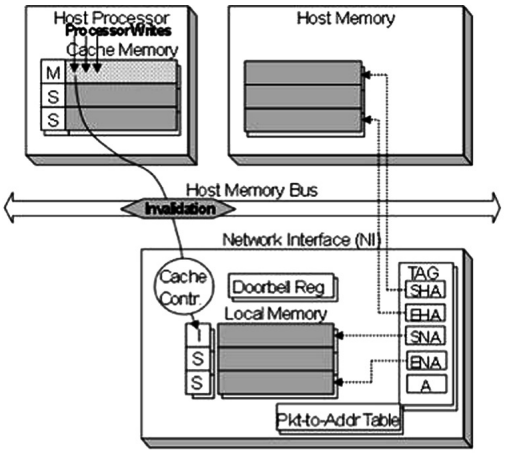


Fig. 4. Hardware architecture of the EDT-based NI.

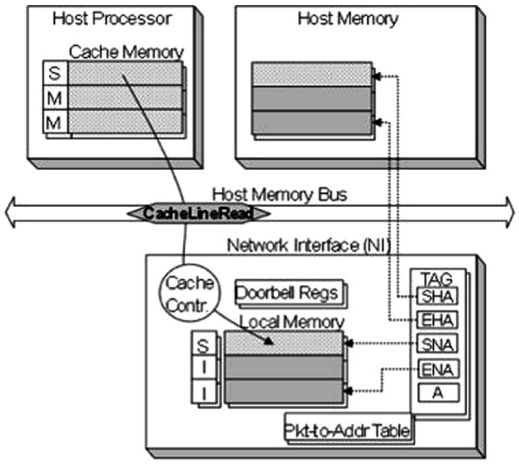
4. Eager Data Transfer

Fig. 3(a) shows the timing of the DMA-based approach for a message send operation. As can be seen in the figure, there is a time interval between when the user process writes data to the message buffer (Send1) and when data is DMA-transferred from the message buffer to NI buffer (Send5). Since this time accounts for a significant portion of the overall latency, the primary motivation of EDT is to overlap the execution of the user data writes (Send1) and the DMA transfer of data to NI buffer (Send5) to reduce the overall latency for message send/receive. Fig. 3(b) illustrates the advantage of the EDT mechanism, where data transfers are performed in cache-block units as the user data is generated by the application. Therefore, as the application builds the message in the user buffer, the entire message is copied into the NI buffer. Thus, right after a token is written into the doorbell register (Send3), the NI can immediately proceed with sending of the message.

To support such an implementation, the EDT-based NI employs a simple cache coherence hardware, which includes a set of tags to hold memory addresses for registered memory regions. The tags are used for monitoring the host memory bus to detect host processor's memory accesses to the registered regions. The bus traffic monitoring and the associated cache coherence control are implemented by observing the subset of the underlying cache coherence protocol. The tags are also used to associate a registered host memory region with a NI memory region. In addition, the NI buffer uses status bit per each cache block to indicate the modification of the data. Memory updates on a host memory region are reflected on the associated NI buffer using the memory association information. The proposed

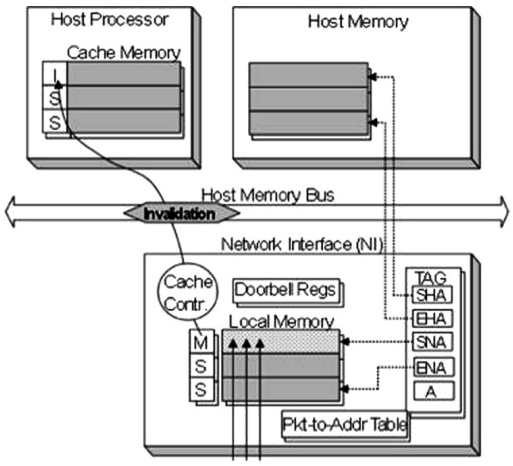


(a) Processor Writes

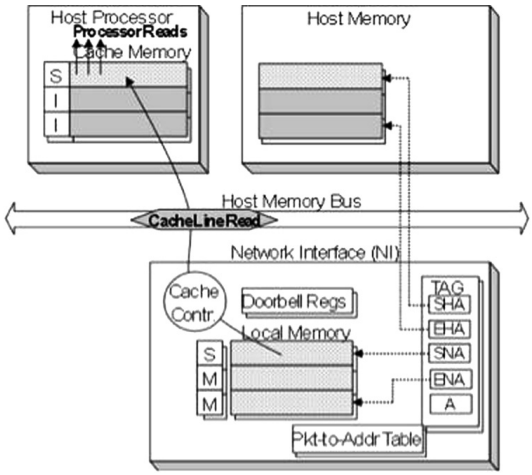


(b) Data Transfer

Fig. 5. EDT operations for message send.



(a) Network message receive



(b) Data transfer

Fig. 6. EDT operations for message receive.

EDT mechanism does not depend on a specific cache coherence protocol. However, the MESI (*Modified*, *Exclusive, Shared,* and *Invalid*) cache coherence protocol is assumed for the sake of discussion [13]

Fig. 4 shows the hardware architecture of the EDT-based NI, which includes doorbell registers, tags, local memory, Packet-to-Address table, and cache controller. During the registration of a host memory region (using VipRegisterMem), the corresponding NI buffer and tag entry are allocated. The allocated NI 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 block size. Each memory block has a *Status bit* (ST) indicating the state of each cache block in the local memory. Each *tag* (TAG) entry consists of five fields: *Start* and *End Host Addresses* (SHA and EHA), *Start* and *End NI Addresses* (SNA and ENA), and *Access bit* (A). The host address pair, SHA and EHA, points to a registered memory region in the host memory. The NI address pair, SNA and ENA, points to the associated buffer in the NI local memory. A-bit is used to represent the validity of the tag. The *Packet-to-Address Table* (Pkt-to-Addr Table) is used during message receive operations to map incoming packets to the local memory. In addition, virtual-to-physical address translation table for the register memory region is generated and copied (DMA) to the NI local memory (see Section 5).

The cache coherence protocol for EDT consists of *Modified* (M), *Shared* (S) and *Invalid* (I) states and they share similar meaning with their MESI protocol counterparts. Once the Tag fields are properly set, the cache blocks for the registered memory regions need to be set to either the S or I state. This step is necessary so that the EDT-based NI can monitor the host processor writes to the registered memory region. There are two ways to accomplish this. First method is to invalidate the cache blocks by flushing the cache. The second method is to have the EDT-based NI initiate read bus transactions for the registered memory region. This is done by an initialization routine implemented

in VI-MSG (see Section 6). The data is copied either from the host cache or from the host memory. If data is copied from the host cache, both the NI buffer blocks and the host cache blocks transition to the S state. However, if the data is copied from the host memory, the data is held only in the NI buffer and the memory blocks transition to the S state (Note that unlike the MESI protocol, EDT does not need to differentiate between E and S states. Thus, for simplicity S state is used instead of E state.). In either case, the initialization overhead is a one-time cost incurred during memory registration and does not affect the communication latency of send/receive operations.

Fig. 5 shows the detailed operations for message send assuming both memory blocks of the NI buffer and the host cache blocks are in the S state (similar steps would be performed if initialized to the I state). When the host processor writes to the registered memory region (Send1 of Fig. 2), there can be either a cache hit or a miss. If a processor write hits on the cache, an invalidation bus transaction is generated to gain exclusive ownership of the cache block. On the other hand, if a processor write misses on the cache, the cache block is first loaded from the host memory and then an invalidation bus transaction is generated. In either case, the cache block and the memory block in the NI buffer transition to the M and I states, respectively. Then, the EDT-based NI 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 S state. As the host processor continues to write to the registered memory region, memory blocks in the NI Local memory are invalidated and the updated cache blocks are read into the NI buffer.

For the message receive case, it is important to note that the host processor read operations do not begin until the completion status is set in the CQ. Therefore, unlike the message send case, the data transfer phase cannot be overlapped with the host processor reads. However, the EDT mechanism can still avoid the use of DMA and thus eliminate DMA startup and interrupt processing overhead resulting in small performance gain. There are two possible EDT implementations for message receive. The first option is to rely on the cache coherence mechanism of EDT, which results in symmetrical behavior for message send and receive. The second method is to bypass the cache coherence and operate the EDT mechanism as a DMA engine. These two methods are described below.

Fig. 6 shows the operations performed by EDT for a message receive using cache coherence. As soon as a message is received, the network-side DMA (not shown in Fig. 6) transfers it to the NI buffer. The Pkt-to-Addr Table is searched to determine the NI buffer address for the message. As the message is moved to the NI buffer, invalidation transactions are generated to invalidate the cache blocks in the host processor's cache (Fig. 6(a)). The completion of the invalidation transactions causes the NI buffer blocks to transition from the S state (initial state) to the M state. Once the message is moved to the NI buffer, VI-MSG starts its processing and 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 cause the blocks to be copied to the host processor's cache and both blocks in the host (I $\rightarrow$ S) and NI (M $\rightarrow$ S) transition to the S state.

If EDT operates as a DMA engine during message receive, simple *Valid*(V)/*Invalid*(I) states are used to indicate the status of the memory blocks. When a message is received, it is moved to the NI buffer and the memory blocks are set to the V state indicating they contain a new message. For each memory block set to the V state, the NI cache controller issues a bus write-back request to flush the memory block to the host memory.

The main difference between the two methods is that when the host processor is ready to read the message, the first method reads it from the NI buffer while the second method reads it from the host memory. Therefore, both methods result in similar performance but the latter method is less complex since it does not rely on cache invalidations.

5. EDT vs. CNI Comparison

Both EDT and CNI rely on the underlying cache coherence mechanism to perform data transfers, but that is where the similarity ends. Therefore, this section provides a more detailed comparison of the two methods and discusses what effect these differences have on their latency and the software abstraction for communication.

EDT implements message buffer as a randomly accessible memory space defined during memory registration. Therefore, data can be written to the message buffer in any order, but sent out to the network only after the doorbell is rung. This is possible because invalidations from the host processor act as notifications to EDT-based NI to issue bus requests to read in the cache blocks. Thus, cache blocks are transferred in the order they are written to. In contrast, CNI implements the message buffer as a queue structure, which requires user message to be written in strict sequential order. CNI polls the cache block at the head of the queue to transfer the blocks. Since both head and tail pointers are needed to determine whether the queue is full or empty, updating of

the tail pointer by the host processor and head pointer by the CNI causes invalidations to ping-pong back and forth increasing the traffic on the bus.

CNI employs three optimizations based on Lazy pointers, valid bits, and sense reverse to reduce the bus traffic [6]. The idea behind *lazy pointers* is to remove the host processor's dependency on the head pointer. This is done by relying on a potentially stale head (called shadow) pointer rather than keeping an actual copy of the head pointer. Thus, the host processor can conservatively check whether the queue is full by comparing the tail and the shadow head pointers. Only when the queue is full, the shadow head pointer is updated with the head pointer. On the other hand, *valid bits* with *sense reverse* are used to eliminate the CNI's dependency on the tail pointer. Valid bits, which are set by the host processor during a message send, indicate the message on the head of the queue is valid. This eliminates the need for CNI to check the tail pointer to determine if the queue is non-empty (i.e., a message exists). Sense reverse eliminates the need to clear valid bits, which causes invalidations, after a message is read from the head of the queue. This is done by alternating the encoding of the valid bits on each pass through the queue. This way both the host processor and CNI keep track of the sense of their current pass, and the message is valid only when the current sense matches with the valid bit in the message.

In terms of the best-case latency, both CNI and EDT would exhibit similar performance. By best case, we mean that data is written in-order and all the words in a cache block are written before it is transfer to NI. Since the bus transaction latency is much longer than the time required to complete writes to the rest of the cache block words, each cache block transfer would require two bus transactions: One invalidation from the host processor to gain exclusive ownership and one bus read request from NI to transfer the block. For CNI with the three optimizations mentioned above, there is additional overhead of polling the head pointer to check whether the queue is empty or full. This incurs only two additional invalidations for each pass through the queue assuming queues are no more than half full on average [6], and thus have minimal effect on performance.

The worst-case latency occurs when non-sequential access patterns cause cache blocks to be read into the NI buffer before the host processor completes all the writes. Note that CNI does not allow non-sequential accesses and therefore this situation does not apply. In particular, consider the case when consecutive writes are performed to fill the message buffer, but with a stride equal to the number of words in a cache block. When the host processor writes to the first word of a cache block, invalidation bus transaction is generated and the cache block transitions from the S or I state to the M state and the memory block of the NI buffer transitions from the S to I state. Then, the EDT-based NI issues bus read transaction to transfer the cache block. If the host processor writes to the second word of the cache block after the bus read transaction, another set of invalidation and bus read request is generated. This causes the cache block to trash back and forth while the host processor's writes are being executed to the cache block. Therefore, the worst-case latency is $2BW$, where B is the number of blocks and W is the number of words in the cache block.

The most important advantage of EDT over CNI is that random access capability of EDT allows for zero-copy communication. In contrast, CNI was not designed with zero-copy in mind due to its FIFO nature. For example, EDT and CNI would serve as a low-level communication facility for high-level message-passing facilities, such as MPI. MPI is the de facto standard for user-level message-passing [20], and there a number of implementations of MPI on VIA [21-24] and InfiniBand [26-27]. In order to facilitate communication, message buffers can be pre-registered to avoid the cost of memory registrations and de-registrations on the fly. To perform a send, the user message is first copied to the registered message buffer and then DMA-transferred to the network interface. Both EDT and CNI would work in this case since user message can be generated non-sequentially but would be copied sequentially. However, implementing zero-copy message transfer requires the user message area to be first registered before the user message is generated. Therefore, unless the user message is generated sequentially, cacheable queue implementation of CNI would not suffice.

There are a number of ways to implement zero-copy using the EDT mechanism. One possible solution is to implement *pinned-down cache* for the registered memory regions [28]. The idea behind pinned-down cache is to take advantage of the fact that often applications programs repeatedly transfer data from the same memory area. Therefore, rather than de-registering the memory region right after a send, the request to de-register is delayed so that the pinned-down area can be reused. A registered memory region is released only when the total area of the registered memory regions exceeds the predetermined maximum size. This way, the cost of a single memory registration can be amortized over multiple sends.

For CNI, the size of Cacheable Queue can be different than the messages being sent. Lazy pointers used to minimize the number of invalidations assume the queue is no more than half full on average, and thus checks the head pointer only twice for each pass around the queue. This causes under utilization of the queue capacity and results in

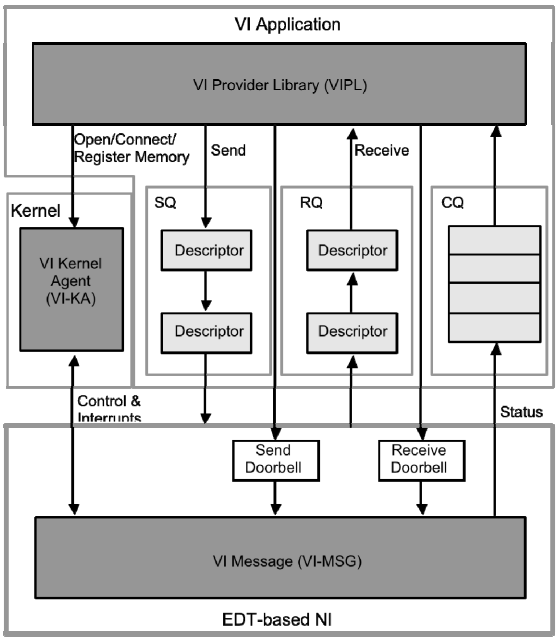


Fig 7. SONIC VIA.

throttling of the message send operation. In contrast, EDT requires a buffer to be allocated in NI's local memory during memory registration that is the same size as the host memory region. Therefore, the number of memory registrations is bounded by the size of the NI memory. However, EDT is proposed for applications requiring low-latency communications, such as parallel applications, which use small message sizes that are typically much smaller than 4 Kbytes [6]. For example, 4 Mbytes of NI memory can accommodate a thousand of registrations with a 4-Kbyte buffer. Considering today's memory technology, hundreds Mbytes of memory can be obtain at a small cost. Therefore, providing a sufficient amount of NI memory for low-latency application programs is not a problem. If memory registration cannot proceed because of lack of free space in the NI memory, it can be postponed until a free space is available or EDT-based NI can fall back to DMA-based transfer.

6. A VIA Implementation for the EDT-based NI

Fig. 7 shows our implementation of VIA for the EDT-based NI, called *SONIC-VIA*, which is based on M-VIA [3]. SONIC-VIA consists of *VI Message* (VI-MSG) layer as well as VIPL and VI-KA layers discussed in Section 3. The protocol layers are distributed over the host and EDT-based NI; i.e., VIPL is compiled into the application, VI-KA is executed as a kernel module, and VI-MSG is executed on the EDT-based NI.

VIPL provides VI library functions for user applications and sends users requests to either VI-KA or VI-MSG. Since user requests are sent to two different layers, VIPL uses two different service callings; doorbell register and IOCTL system call. Doorbell registers are used to notify message send (VipPostSend) and receive (VipPostRecv) requests to VI-MSG. IOCTL system calls are handled by VI-KA, which services calls to VI primitives other than message send and receive, such as VipOpenNI, VipConnectRequest, and VipRegisterMem.

The VI-MSG layer includes routines for message send and receive. When an application program writes a token to a doorbell register, VI-MSG reads the token to determine the user (virtual) address for the descriptor. Then, the user address is mapped to an NI memory address through a two-level translation scheme: User (virtual) address to host memory (physical) address and then the host memory address to NI buffer address. The second memory translation is performed by looking up the tag entries (see Fig. 4). The descriptor is then accessed from the NI buffer. Since the descriptor holds the virtual address of the user data, VI-MSG again goes through address translation and local memory access for user data. Thus, VI-MSG is responsible for maintaining the address translation table, performing address translations, and accessing descriptors and data from the NI buffer. The rest of the VI-MSG operations involve processing packet frames, performing fragmentation/de-fragmentation for user data whose size is over the MTU (Maximum Transfer Unit), and updating the status flag in the VI or CQ.

7. Performance Evaluation

7.1 Simulation Environment

In order to evaluate the performance of the proposed EDT mechanism, SONIC-VIA was implemented and simulated on Linux/SimOS [10]. Since Linux/SimOS provides a real program execution environment, SONIC-VIA and benchmark programs were executed on top of the Linux/SimOS to perform detailed system evaluation in a non-intrusive manner. This allows us to capture all aspects communication performance that includes the effects of application, network protocol, and network interface. The system configuration and parameters used in the simulation study 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 NI processor does not include private caches. This assumption was made

Table 1. System Parameters.

a) ***System Parameters***		
Host Processor	Processor Speed	1 GHz
	L1 Cache Size/Line Size /Assoc./Latency:	32KB/32B /2-way/1 cycle
	L2 Cache Size/Line Size /Assoc./Latency	1M/128B /2-way/10 cycles
	Cache Coherence	MESI, write-invalidate
System Bus	Width	128 bits
	Speed	100 MHz
	Burst-mode BW (peak)	1,600 MB/sec
NI	NI Processor Speed	100 MHz
	Host-side DMA data transfer rate (peak)	1,600 MB/sec
	Local Memory Latency	10 ns

based on the fact that most commercial NI processors, such as the Myrinet LANai processor, do not have caches [31]. The EDT-based NI has a peak DMA rate comparable to the burst-mode 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 EDT mechanism was compared against CC-DMA with retry (CC-DMA$_{retry}$) and CC-DMA with intervention (CC-DMA$_{intv}$). As mentioned earlier, the difference between CC-DMA$_{retry}$ and CC-DMA$_{intv}$ is that the latter scheme can read directly from the cache memory, and thus eliminates the additional bus traffic required to flush the cache blocks to the host memory. 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 the 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.

7.2 Simulation Results

The total execution times for message communication between two user applications are presented in Fig. 8. For each message size, there are three bar graphs representing the latencies (in cycles) of EDT (left), CC-DMA$_{intv}$ (middle), and CC-DMA$_{retry}$ (right). These simulations were run with

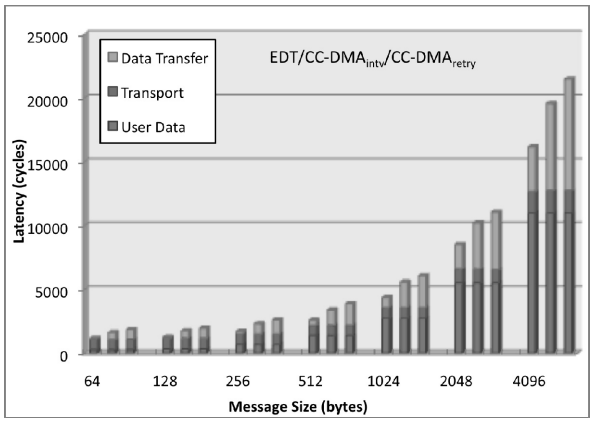


Fig. 8. Communication latency.

a fixed MTU size of 1,500 bytes. The communication latency represents the number of host processor cycles between the library call VipPostSend() from the sender and the return of the library call VipPostRecv() by the receiver. 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 4,096 bytes, which represent a typical range of message sizes for a SAN environment [6]. Fig. 8 shows that the proposed EDT mechanism results in much better performance compared to CC-DMA$_{retry}$ and CC-DMA$_{intv}$. The EDT-based NI attains 17% ~ 38% reduction in the user-to-user messaging latency compared to CC-DMA$_{intv}$. More importantly, the performance improvement becomes more significant as message size increases. It is important to notes that the performance results in Fig. 8 are based on the theoretical peak DMA rate of 1,600 MB/sec. However, a typical DMA rate is much lower due to bus contention and memory architecture, e.g., RAS-to-CAS latency due to a page miss. Our simulation study based on 75% (1,200 MB/sec) and 50% (800 MB/sec) of the peak DMA rate showed improvements of 27% ~ 40% and 34% ~ 42%, respectively.

In order to gain a better understanding of the performance improvement, Fig. 8 also shows the user-to-user communication latency subdivided into three most significant operations: User Data, Data Transfer, and Transport. User Data is the time required for the user program to write a message to the user buffer (Send1 in Fig. 2). Data Transfer includes the time to transfer data between host and NI. For CC-DMA$_{retry}$ and CC-DMA$_{intv}$, this includes the time for DMA setup, DMA operations, and handling interrupts after DMA operations complete. For EDT, this includes the time to perform bus write-back to flush the memory block to the host memory during message receive. Transport represents the time required to run the network protocol (VI-MSG) to service the user

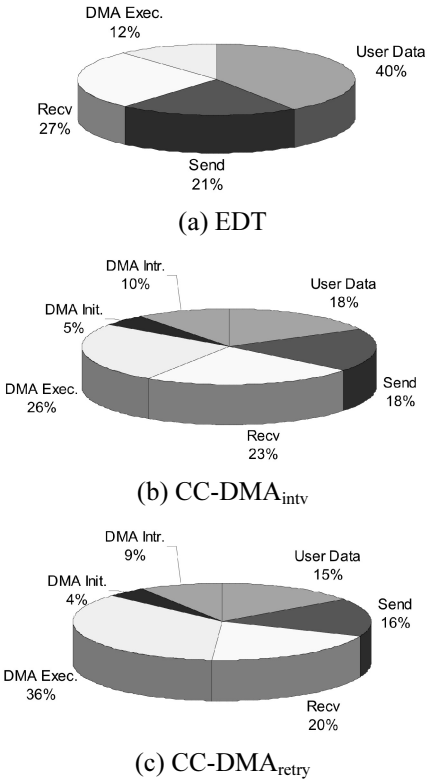


Fig. 9. Breakdown of the communication latency.

send/receive requests.

The breakdown view of Fig. 8 clearly shows how significantly each portion affects the overall latency and increases as data size grows. In particular, the amount of time spent on the Data Transfer portion depends on the underlying data transfer mechanism. The Data Transfer portions for CC-DMA$_{intv}$ are smaller than the ones for CC-DMA$_{retry}$. This is because the CC-DMA$_{intv}$ mechanism supports cache-to-cache transfer so that the user data and descriptor can be moved directly from the cache memory on the host processor. In contrast, the CC-DMA$_{retry}$ mechanism requires two steps to move the user data from the cache memory: The user data in the cache memory has to be first flushed to the host memory and then moved to the NI buffer. The Data Transfer portion for EDT is the smallest because there are no DMA operations and the only cost is the bus write-back operation on the receiver side. The User Data portion increases with the message size, but are the same for all the data transfer mechanisms. Similarly, the Transport sections start to grow as the message size increases beyond the MTU size. This is due to the fact that the network protocol performs fragmentation and de-fragmentation. Again, the Transport sections do not vary with the underlying data transfer mechanism because all three methods were implemented on a common platform, i.e., SONIC-VIA.

The pie charts shown in Fig. 9 give a more detail breakdown of the communication latency for message size of 256 bytes. Transport is further subdivided into Send and Receive portions. Data Transfer for CC-DMA$_{retry}$ and CC-DMA$_{intv}$ is subdivided into DMA initiation (DMA Init), execution (DMA Exec), and interrupt processing (DMA Intr). DMA Init is the time to set up the DMA engine with address and length. DMA Exec is the time for the DMA engine to move user data and descriptor between the host and NI buffers. DMA Intr is the time taken to process the interrupt signaling at the end of a DMA operation. Among the various portions, only the DMA Exec portion increases with increased message size, and the rest of the portions remain constant. As explained earlier, DMA Exec is larger for CC-DMA$_{retry}$ than CC-DMA$_{intv}$ because the user data is transferred directly through host cache memory for CC-DMA$_{intv}$. These results clearly show that the EDT approach significantly reduce the communication latency virtually eliminating DMA operations.

8. Conclusion

This paper proposed the EDT mechanism to reduce communication latency for user-level network protocols. The EDT reduces the time to transfer data between host memory and NI by overlapping writes to the user buffer with the actual transfer of data from user to NI buffer. Our detailed simulation study using Linux/SimOS showed that the EDT reduces the message latency by 17% ~ 41% compared with the CC-DMA based schemes.

There are a number of ways the EDT can be extended. First, more work is needed to measure performance with various parallel and distributed applications. Another challenge is to apply the EDT mechanism to the legacy network protocols such as TCP/IP and UDP/IP. Even though the user-level protocols have become imperative for SANs, reducing the latency of the legacy protocols will continue to be important. Another interesting application of the EDT is on embedded NI for System-on-Chip system(SoC). Because the EDT observes only a subset of cache coherence protocol, the design can be simplified for SOC applications. This will allow EDT-based embedded NI to achieve low-latency communication with less complexity and simpler design.

Reference

[1] Intel, Compaq and Microsoft Corporations, “Virtual Interface Architecture Specification, Version 1.0,” December 1997. Available at http://www.viarch.org.

[2] M. Banikazemi *et al*., “Design Alternatives for Virtual Interface Architecture (VIA) and an Implementation on IBM Netfinity NT Clusters,” *Proc. of the Int'l*

Parallel and Distributed Processing Symposium, May 2000.

[3] NERSC, "M-VIA: A High Performance Modular VIA for Linux," Available from http://www.nersc.gov/research/FTG/via.

[4] Infiniband Trade Association, "Infiniband Architecture Specification, Vol. 1," InfiniBand Trade Association. Available from http://www.infinibandta.org.

[5] S.S. Mukherjee and M.D. Hill, "Making Network Interfaces Less Peripheral," *IEEE Computer*, 31(10):70-76, October 1998.

[6] S.S. Mukherjee *et al*., "Coherent network Interfaces for Fine-Grain Communication," *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.

[7] S.S. Mukherjee *et al*., "The impact of Data Transfer and Buffering Alternatives on Network Interface Design," *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 1998.

[8] 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, November 1998.

[9] R.A.F. Bhoedjang *et al.*, "Reducing Data and Control Transfer Overhead through Network-Interface Support," *First Myrinet User Group Conference (MUG)*, September 2000.

[10] Won, C. *et al.*, "Linux/SimOS - A Simulation Environment for Evaluating High-Speed Communication Systems," *Proceedings of the 2002 international Conference on Parallel Processing (ICPP)*, August 2002. An extended version of this paper appears in "Linux/SimOS: A Complete System simulation Environment for Evaluating High-Speed Communication Systems," *Journal of High Speed Networks*, 2005.

[11] 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.

[12] F. Briggs *et al*., "The Intel 870 Family of Enterprise Chipsets," *Proc. of the Hot Chips XIII*, August 2001.

[13] Intel Corporation, "Pentium(R) Processor Family Developer's Manual," Available at http://developer.intel.com/design/ intarch/manuals/241428.htm.

[14] M. Rosenblum *et al.*, "Using the SimOS Machine Simulator to study Complex Computer Systems," *ACM Transactions on Modeling and Computer Simulations,* 7(1), January 1997.

[15] J. R. Thorpe, "A Machine Independent DMA Framework for NetBSD," *USENIX 1998 Annual Technical Conference*, June 15-19, 1998.

[16] MIPS R1000 Microprocessor User's Manual, Version 2.0. Available from http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/hdwr/bks/SGI_Developer/books/R10K_UM/sgi_html/t5.Ver.2.0.book_396.html.

[17] The PowerPC Architecture: A Specification for a New Family of RISC Processors, Edited by C. May, D. Silha, R. Simpson, and H. Warren, Morgan Kaufmann Publishers, Inc., 1994.

[18] Pentium Processor Family Developer's Manual. Available from http://developer.intel.com/design/pentium/manuals/.

[19] The Alchemy Au1100™ From AMD Internet Edge Processor Data Book. Available from www.sagitron.es/data_sheet/au1100.pdf.

[20] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 8, 994.

[21] MVICH: MPI for Virtual Interface Architecture, Berkeley Lab: http://www.nersc.gov/research/FTG/mvich/.

[22] T. Mehlan *et al., "*Providing a High-Performance VIA-Module for LAM/MPI," *Parallel Computing in Electrical Engineering, International Conference on (PARELEC'04)*, September 07 - 10, 2004

[23] M. Bertozzi, M. Panella, and M. Reggiani, "Design of a VIA based communication protocol for LAM/MPI Suite," *9th Euromicro Workshop on Parallel Distributed Processing*, Sept. 2001.

[24] R. Dimitrov and A. Skjellum, "An efficient MPI implementation for Virtual Interface Architecture -- enabled cluster computing," *Proc. of the 3rd MPI developer's and user's conference*, Atlanta, Georgia, March 1999.

[25] MPICH-A Portable Implementation of MPI: http://www-unix.mcs.anl.gov/mpi/mpich/.

[26] Second Version of MPICH: http://www-unix.mcs.anl.gov/mpi/mpich2/.

[27] MVAPICH: MPI for InfiniBand on VAPI Layer, Ohio State University: http://nowlab.cis.ohio-state.edu/projects/mpi-iba/

[28] H. Tezuka *et al.*, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," *12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

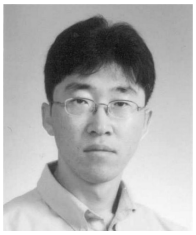
[29] J. Liu *et al.*, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support," *International Parallel and Distributed Processing Symposium (IPDPS 04)*, 2004

[30] R. Grabner, F. Mietke, and W. Rehm, "An MPICH2 Channel Device Implementation over VAPI on InfiniBand," *Proc. of CAC'04, Workshop on Communication Architecture for Clusters* held in conjunction with IPDPS 2004, April 26-30 2004, Santa Fe, New Mexico.

[31] N.J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, 15(1):29-36, February 1995.

Chulho Won
received BS in Electrical Engineering from Korea Aerospace University, MS in Electrical Engineering from KAIST (Korea Advanced Institute of Science and Technology), and Ph.D. in Computer Engineering from Oregon State University, in 1985, 1987, and 2004, respectively. He was a senior technical staff at ETRI (Electronics and Telecommunications Research Institute) between 1987 and 1998. He is an Assistant Professor of Electrical and Computer Engineering Department at California State University, Fresno. His current research interests include networked embedded systems, large-scale wireless sensor networks, and computer networks.

Ben Lee
received his B.E. degree in Electrical Engineering in 1984 from the Department of Electrical Engineering at State University of New York (SUNY) at Stony Brook, and his Ph.D. degree in Computer Engineering in 1991 from the Department of Electrical and Computer Engineering at the Pennsylvania State University. He is currently an Associate Professor of School of Electrical Engineering and Computer Science at Oregon State University. He has published over 70 conference proceedings, book chapters, and journal articles in the areas of embedded systems, computer architecture, multithreading and thread-level speculation, parallel and distributed systems, and wireless networks. He received the Loyd Carter Award for Outstanding and Inspirational Teaching and the Alumni Professor Award for Outstanding Contribution to the College and the University from the OSU College of Engineering in 1994 and 2005, respectively. He also received the HKN Innovative Teaching Award from Eta Kappa Nu, School of Electrical Engineering and Computer Science, 2008. He has been on the program and organizing committees for numerous international conferences, including 2000 International Conference on Parallel Architecture and Compilation Technique (PACT), 2001 and 2004 IEEE Pacific Rim Dependable Computing Conference (PRDC), 2003 International Conference on Parallel and Distributed Computing Systems (PDCS), 2005-2008 IEEE Workshop on Pervasive Wireless Networking (PWN), and 2009 IEEE International Conference on Pervasive Computing and Communications. He is currently the Workshop Chair for PerCom 2009. He was also an invited speaker at the 2007 International Conference on Embedded Software and System. His research interests include embedded systems, computer architecture, multithreading and thread-level speculation, parallel and distributed systems, and wireless networks.



Kyoung Park
received the M.E. degree in Computer Engineering from ChonBuk National University, Korea and the Ph.D. degree from Korea University, Korea in 1993 and 2008, respectively. He joined Electronics and Telecommunications Research Institute (ETRI) in Daejeon, Korea in 1993 and he is serving as a section head of software & contents future technology research team. He developed main memory subsystem of a SMP system called TICOM-III, router switch of a high performance parallel system called SPAX, on-chip multiprocessor called Raptor, and InfiniBand HCA (Host Channel Adapter). Recently, he has been involved with developing Ubiquitous Service Platform as a system architect and a hardware designer. His main interest is computer architecture with a focus on multiprocessor, memory hierarchy, and advanced I/O architecture for next generation computing.

Myung-Joon Kim
received the B.S. degree from Seoul National University, Korea, the M.S. degree from KAIST (Korea Advanced Institute of Science and Technology) and the Ph.D. degree from University of Nancy I, Nancy France in 1978, 1980, and 1986, respectively, all in computer science. He joined ETRI in 1986 and has worked for the development of system software technologies especially database systems and distributed system technologies. He served as head of Database Section (1989-1992). In 1993, he worked at University of Nice Sophia-Antipolis, France as a visiting professor. He also served as Head of Software Engineering Section (1994), Director of System Software Department (1995-1997), Director of Internet Service Department (1998-1999), Vice President of Computer & Software Technology Laboratory (2000-2001), Director of Contents Technology Department (2001), Director of Computer System Department (2002-2003) and Director of Internet Server Group (2004-2008). Currently, he is a Vice President of R&D Strategy Planning Division. His current research interests include database system, real-time event processing, open source software technologies and their deployment for new Internet applications such as ubiquitous computing platform and next generation software architecture.