Cluster Computing (2019) 22:S1815–S1826
https://doi.org/10.1007/s10586-018-1707-0

Design and implementation of initial OpenSHMEM on PCIe NTB based cloud computing

Cheol Shim[1] · Kwang-ho Cha[2] · Min Choi[1]

Received: 4 December 2016 / Revised: 1 September 2017 / Accepted: 3 January 2018 / Published online: 1 February 2018

Abstract
Cloud computing services are provided by key roles of data centers in which homogeneous and heterogeneous computation nodes are connected by high speed interconnection network. The rapid development of cloud-based services and applications has made data center networks more difficult. The PCI Express is a widely used system bus technology that connects processors and peripheral I/O devices. So, the PCI Express is regarded as a de-facto standard in system area interconnection network. It is currently validating the possibility of using PCI Express as a system interconnection network in areas such as high-performance computers and cluster/cloud computing. With the development of PCI Express non-transparent bridge (NTB) technology, the PCI Express has become available as a system interconnection network. NTB allows two PCI Express subsystems to be interconnected and, if necessary, isolated from each other. Partitioned global address space (PGAS) is one of the shared address space programming models. Due to the recent spread of multicore processors, PGAS has been attracting attention as a parallel computing framework. We make use of the PCI Express NTB to realize the PGAS shared address space model. In this paper, we designed and implemented the interconnection network using PCI Express x8 using a RDK, the PEX8749 based PCI Express evaluation board. We performed some Openshmem applications from Github to verify the accuracy of our initial OpenSHMEM API implementation.

Keywords PCI Express · Non-transparent bridge · Interconnection network · RDMA · One-sided communication

1 Introduction

Data centers have become increasingly part of in the cloud computing. The rapid development of cloud-based services and applications has made data center networks more difficult. Traditional electronic interconnect networks can hardly meet all requirements for bandwidth, device cost, and power dissipation. To solve this problem, a PCI Express interconnect network characterized by high-speed interconnect networks, low cost, and high bandwidth has been proposed as shown in Fig. 1. Recently, it has become a topic in the field of research. Nowadays, hundreds of applications have been implemented using high-speed networking and local networking. In these applications, we can achieve more benefits of high-performance, low power and reduced costs when it is used in field of interconnection network.

✉ Min Choi
mchoi@cbnu.ac.kr

1 Department of Information and Communication Engineering, Chungbuk National University, Cheongju 28644, Korea

2 Center for Supercomputer Development, Korea Institute of Science and Technology Information, Daejon, Korea

As the computing technology and network technology development, the amount of data to be processed by processors was increased, resulting in high-performance computing. Recent high-performance computing system are based on a system interconnect technology such as Infiniband and Ethernet, and PCI Express.

PCI Express bus is originally a technology for the connection among processors, memory, and pe-ripheral I/O devices. PCI Express has been developed from 2.5Gbps Gen1 specification to Gen3 specification technology. With the increase of speeds and performance needs, there were many requests to make use of bus bandwidth efficiently on system interconnection network [1,2].

In field of high performance computing and cluster computing, interconnection network technology has been widely studied for a long time. Majority of interconnection network technologies are Infiniband and Ethernet. Heymian [3–5] use the PCI Express non-transparent bridge (NTB) to implement

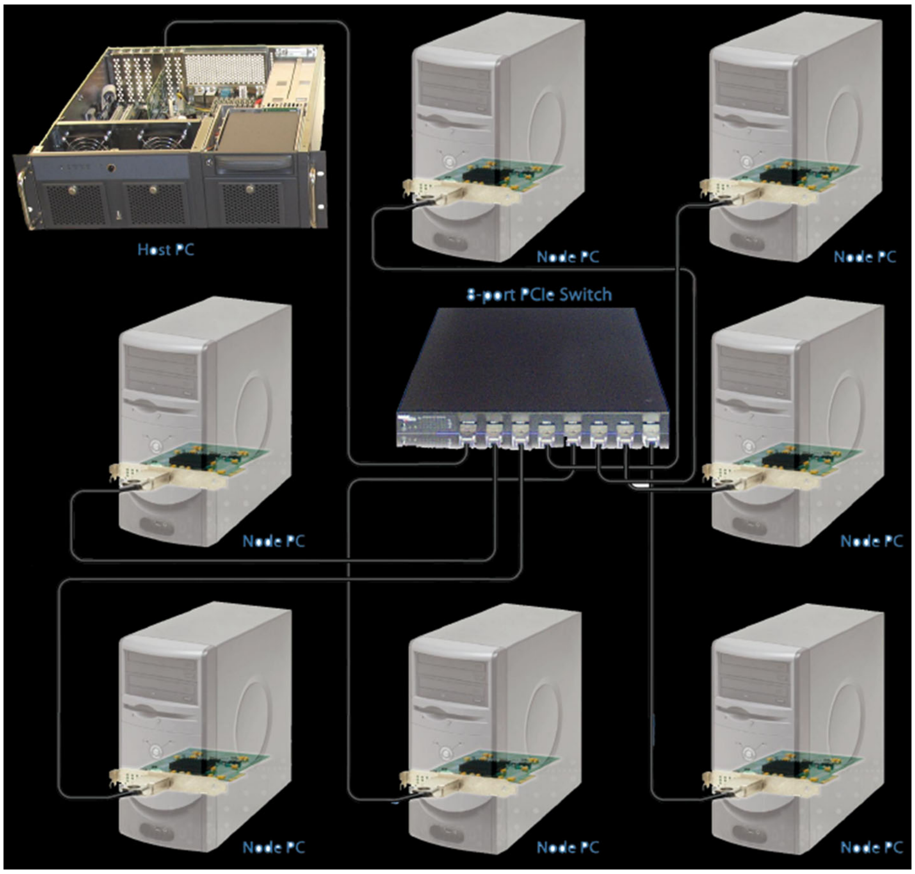


Fig. 1 Technology trends of cloud data center interconnection [1]

a multi-host system. The PCI Express NTB solves clock synchronization issues and provides isolation between hosts. RCT Magazine [6] implemented the DMA transfer with a contiguous buffer using the PCI Express bus [7–9]. This approach enables a DMA transfer for non-contiguous memory. This paper design and implemented PC cluster system based on PCI Express interconnection net-work technology. In this paper, we use the PCI Express non-transparent bridge to implement a DMA transfer of non-contiguous memory between two systems.

This paper is organized as follows: Sect. 1 provides introduction, In Sect. 2, we show our design and implementation of OpenSHMEM on PCI Express. In Sect. 3, we analyze and compare the existing and proposed scheme. Section 4 concludes this paper.

2 Initial design of OpenSHMEM on PCIe NTB

PCI Express is a further improved version than PCI bus technology developed by INTEL. PCI Ex-press is a technology by which can be connected with various peripherals or host, support high I/O scalability and high-performance, high-speed communication. PCI express non-transparent bridge is widely used in the multi-host and host-failover. Figure 2 shows the system architecture in which two hosts are connected through PCI Express non-transparent bridge [4,10].

2.1 System architecture

PCI Express is transmitting and receiving data (full-duplex) through an independent link tied to an integrated line in one lanes of transmit and receive. PCI Express may be used by a plurality of lanes (x2, x4, x8, x16) to control the links from a single lane width (X1). PCI Express has the 8Gbps performance of a single-lane Gen3 data rate. Table 1 represents the data transfer rates depending on the generation and link width [6,11,12].

PCI Express non-transparent bridge is used for the mutual isolation in the PCI Express based inter-connection network [13]. NTB has integrated two conceptual interfaces(endpoints) for two roles in one port. We cannot see the fact that the two systems are connected in the BIOS emulation process due to the logically separated interface of PCI Express NTB. Figure 3 depicts an example in which two systems are connected through PCI Express based interconnection network [14].

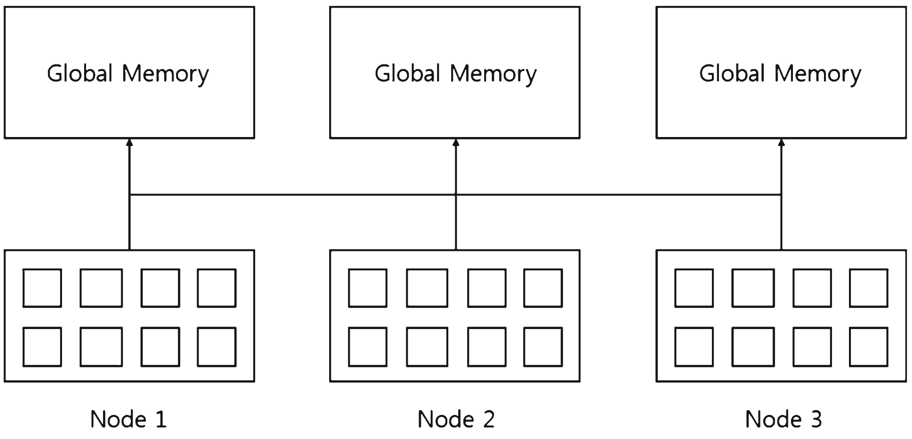


Fig. 2 Multi-host system hierarchy

Table 1 Transfer speed of PCI Express

Generation	Lane (width)				
	X1	X2	X4	X8	X16
Gen1 (Gbps)	2.5	5	10	20	30
Gen2 (Gbps)	5	10	20	40	60
Gen3 (Gbps)	8	16	32	64	96

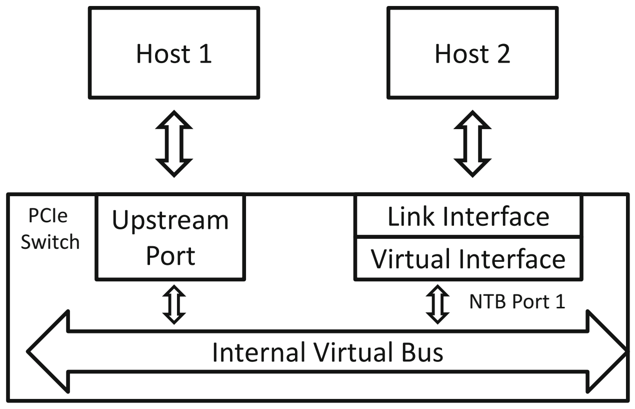


Fig. 3 Interconnect system with non-transparent bridge

2.2 Address translation

In the left side of Fig. 4, Host A and B can communicate via logical endpoint interface of virtual side and link side through address translation. It is necessary to know the address of the external memory on the address translation in the host. There are some scratchpad registers that can be accessed from each of the interface. Two hosts may share data with each other via the scratchpad register. It can also cause an interrupt to the other party through the Doorbell register. The interconnection network determines where the packet is directed through the bus and device information field in the packet header.

The bus number and device number are different, so the interconnection network returns a proper ID depending on the bus number and device number during address translation. In order to do this, there is an address-translation lookup table (LUT) to which we can register an ID, lookup the ID by the index. In the right side of the Fig. 4, each host is located behind its own NTB BAR(s). Each N host address range within the global space is also divided into Nx segments. The figure illustrates a host A slice of the address range, going to each Host. The host A A-LUT points to one segment within each address range, at a fixed host A offset [15,16].

Figure 5 shows the address translation process of a PCI Express packet [15]. There are also necessary to translate memory address on remote DMA transfer, because the destination address on target machine has totally different address management from that on source machine. We make use of direct address translation in that the offset address are maintained but only the base address is replaced by that of the target machine as shown in Fig. 5.

After the address translation in Figs. 4 and 5, we can communicate each other through Remote Direct Memory Access (RDMA) as shown in Fig. 6. The standard interface of RDMA is described in red colored in Fig. 6, for example, shmem_int_put(), shmem_double_put(), shmem_int_get(), and shmem_doule_get(). After those APIs are called, the first task is attempt to allocate a physically contiguous, page-locked buffer which is safe for use with DMA operation. Another step is to map a previously allocated physical memory into user virtual space. Then, the memory allocation and mapping are done, so the block DMATransfer() function will be called and executed.

2.3 Symmetric heap

The symmetric heap of OpenSHMEM consists of a symmetric data object which is accessible from a remote location and a private data object which is accessible only to the local PE [17,18]. A symmetric data object is allocated to a symmetric heap area. The symmetric heap area are physically allocated to a different address space for each PE node. A symmetric

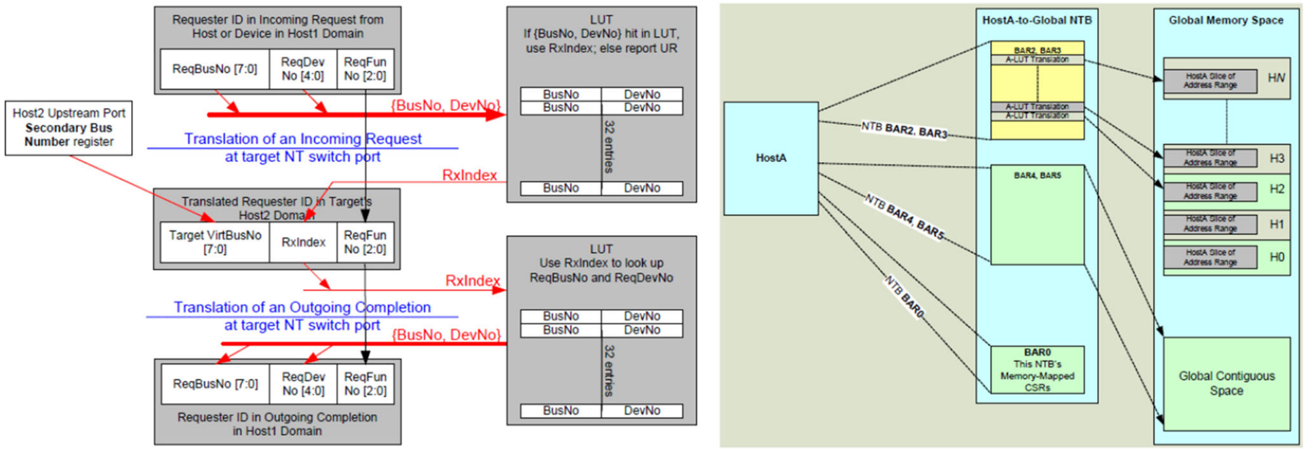
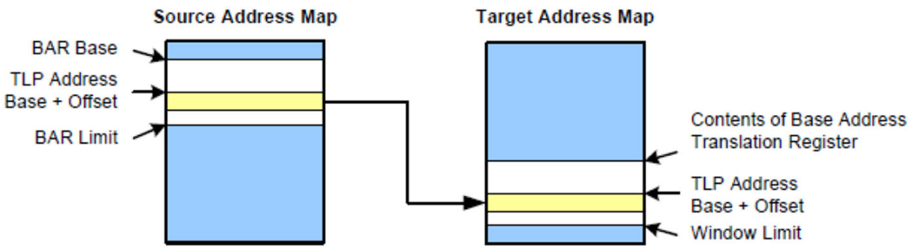

Requester ID in Incoming Request from Host or Device in Host1 Domain
ReqBusNo [7:0]
ReqDev No [4:0]
ReqFun No [2:0]
Host2 Upstream Port Secondary Bus Number register
{BusNo, DevNo}
Translation of an Incoming Request at target NT switch port
RxIndex
Translated Requester ID in Target's Host2 Domain
Target VirtBusNo [7:0]
RxIndex
ReqFun No [2:0]
Translation of an Outgoing Completion at target NT switch port
Requester ID in Outgoing Completion in Host1 Domain
LUT If {BusNo, DevNo} hit in LUT, use RxIndex; else report UR
LUT Use RxIndex to look up ReqBusNo and ReqDevNo
BusNo
DevNo
32 entries
HostA
HostA-to-Global NTB
NTB BAR2, BAR3
NTB BAR4, BAR5
NTB BAR0
BAR2, BAR3
A-LUT Translation
BAR4, BAR5
BAR0 This NTB's Memory-Mapped CSRs
Global Memory Space
HostA Slice of Address Range
HN
H3
H2
H1
H0
Global Contiguous Space

Fig. 4 Requester ID translation sequence on PCIe bus

Fig. 5 Memory address translation sequence


Source Address Map
Target Address Map
BAR Base
TLP Address Base + Offset
BAR Limit
Contents of Base Address Translation Register
TLP Address Base + Offset
Window Limit

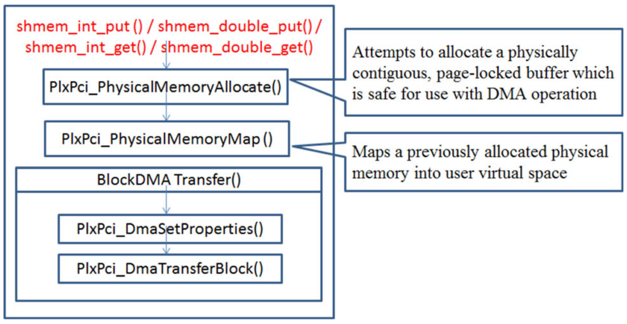

shmem_int_put () / shmem_double_put() / shmem_int_get() / shmem_double_get()
PlxPci_PhysicalMemoryAllocate()
PlxPci_PhysicalMemoryMap ()
BlockDMA Transfer()
PlxPci_DmaSetProperties()
PlxPci_DmaTransferBlock()
Attempts to allocate a physically contiguous, page-locked buffer which is safe for use with DMA operation
Maps a previously allocated physical memory into user virtual space

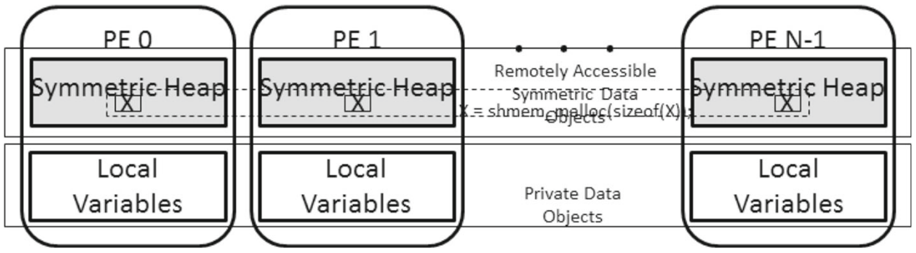
Fig. 6 Our RDMA implementation

data object is created with the same name/size/type through the memory allocation function. The shmem_malloc function assigns a common symmetric data object to each node by calling all PEs together. The offset in the symmetric heap is the same, even though the physical address of the symmetric data object assigned to each PE is different.

Figure 7 describes the design of memory allocation on OpenSHMEM and Fig. 8 shows the flow charts of the barrier implementation. Since the symmetric variables must be assigned to the symmetric heap, it is designed to check if the symmetric heap is already created first by calling a malloc function on each PE. If a symmetric heap does not exist, we create a new symmetric heap. After the allocation of the symmetric heap, each PE shares the address of the allocated symmetric heap to access the symmetric variable in the data transfer function. If a symmetric heap already exists, we check the remaining symmetric heap space to see if we can allocate additional symmetric variables.

If there is not enough memory left in the symmetric heap, a new symmetric heap is allocated. The symmetric variable assigned to each PE has the same offset in the symmetric heap of each PE. Therefore, symmetric variables are allocated in order from the start address of the symmetric heap to the size of the requested symmetric variable.

Fig. 7 Symmetric heap architecture


PE 0
PE 1
PE N-1
Symmetric Heap
X
Remotely Accessible Symmetric Data Objects
X = shmem_malloc(sizeof(X))
Local Variables
Private Data Objects

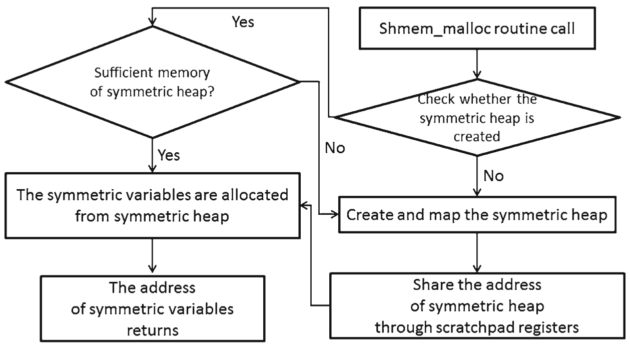

Shmem_malloc routine call
Check whether the symmetric heap is created
No
Create and map the symmetric heap
Share the address of symmetric heap through scratchpad registers
Yes
Sufficient memory of symmetric heap?
No
Yes
The symmetric variables are allocated from symmetric heap
The address of symmetric variables returns

Fig. 8 Flow chart of barrier implementation

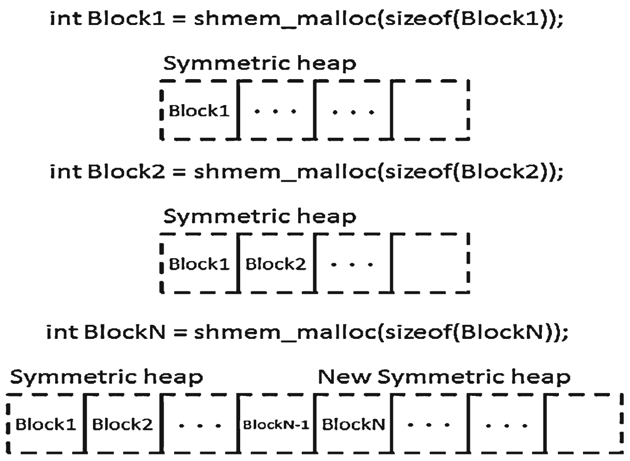

int Block1 = shmem_malloc(sizeof(Block1));
Symmetric heap
Block1
int Block2 = shmem_malloc(sizeof(Block2));
Symmetric heap
Block1
Block2
int BlockN = shmem_malloc(sizeof(BlockN));
Symmetric heap
New Symmetric heap
Block1
Block2
BlockN-1
BlockN

Fig. 9 Symmetric memory implementation for supporting large sized symmetric variable using small chunks

The Fig. 9 shows how symmetric variables are assigned to each PE when calling the malloc function. First, we call the malloc function to allocate the symmetric variable block1 (assuming that the malloc function was called first). Then, we allocate from the start address of the symmetric heap as much as the requested size from the memory of the symmetric heap. However, the malloc function is called to allocate the nth symmetric variable, but the memory of the original symmetric heap is exhausted and there is no remaining memory. We create a new unit size symmetric heap, and we allocate additional symmetric variables by taking up memory from the starting address of the newly created symmetric heap. This process is the same for all PEs because all PEs call shmem_malloc at the same time.

2.4 Synchronization primitive: barrier

A barrier is a type of synchronization method. As shown in Fig. 10, a barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier. Figure 10 is a conceptual representation of barrier. In

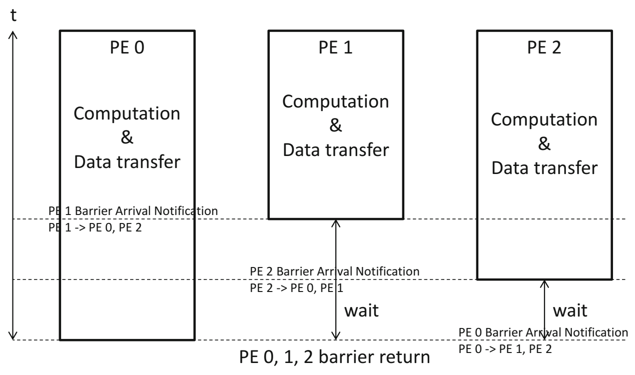

t
PE 0
Computation & Data transfer
PE 1
Computation & Data transfer
PE 2
Computation & Data transfer
PE 1 Barrier Arrival Notification
PE 1 -> PE 0, PE 2
PE 2 Barrier Arrival Notification
PE 2 -> PE 0, PE 1
wait
wait
PE 0 Barrier Arrival Notification
PE 0 -> PE 1, PE 2
PE 0, 1, 2 barrier return

Fig. 10 Conceptual description of barrier

Table 2 Classical barrier implementation

Barrier implementation (ACM TOCS91')
1. Centralized barrier
2. Software combining tree barrier
3. Dissemination barrier (butterfly barrier)
4. Tournament barrier

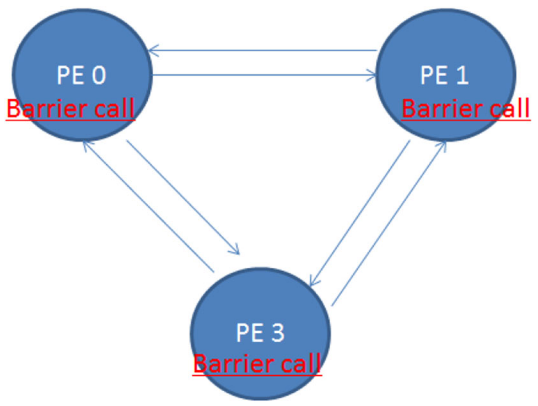


Fig. 11 Our implementation: the centralized barrier

this study, when the DMA transfer is terminated, it notifies other PEs that the barrier function has been performed at the PE and waits until other PEs perform the barrier call.

As shown above, each PE may have different execution time for computation and data transfer. PE 1 confirms that the data transfer is completed after the computation is completed, notifies PE 0 and PE 2 that the barrier has been called, and waits for the barrier arrival of PE 0 and PE 1. When all PE calls the barrier, the synchronization is completed and the process can proceed to the next phase.

In this study, one of the 4 methods, shown in Table 2, presented in the ACM TOCS paper [19] was used to implement the barrier. The Centralized Barrier method is the most typical barrier implementation method as shown in Fig. 11. When each PE (processing element) reaches the barrier, the central-

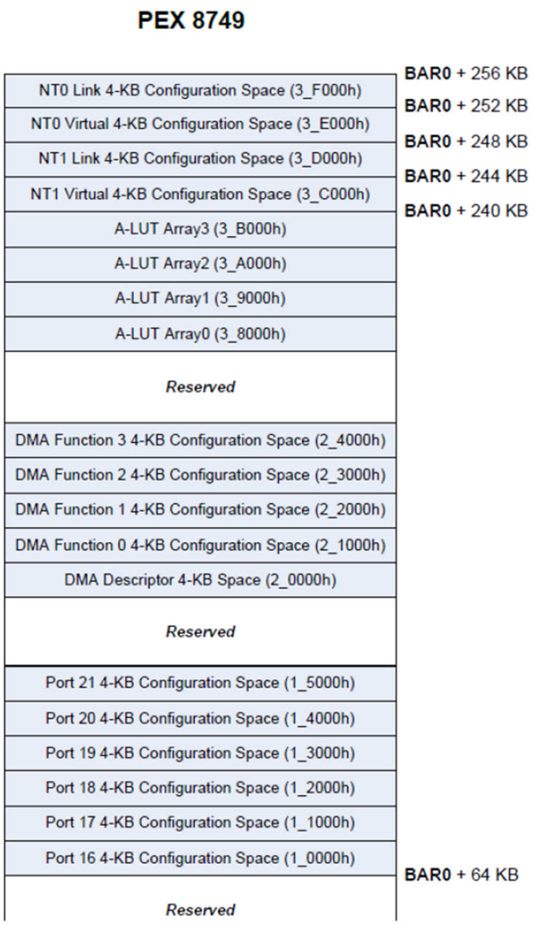

PEX 8749
NT0 Link 4-KB Configuration Space (3_F000h)
NT0 Virtual 4-KB Configuration Space (3_E000h)
NT1 Link 4-KB Configuration Space (3_D000h)
NT1 Virtual 4-KB Configuration Space (3_C000h)
A-LUT Array3 (3_B000h)
A-LUT Array2 (3_A000h)
A-LUT Array1 (3_9000h)
A-LUT Array0 (3_8000h)
Reserved
DMA Function 3 4-KB Configuration Space (2_4000h)
DMA Function 2 4-KB Configuration Space (2_3000h)
DMA Function 1 4-KB Configuration Space (2_2000h)
DMA Function 0 4-KB Configuration Space (2_1000h)
DMA Descriptor 4-KB Space (2_0000h)
Reserved
Port 21 4-KB Configuration Space (1_5000h)
Port 20 4-KB Configuration Space (1_4000h)
Port 19 4-KB Configuration Space (1_3000h)
Port 18 4-KB Configuration Space (1_2000h)
Port 17 4-KB Configuration Space (1_1000h)
Port 16 4-KB Configuration Space (1_0000h)
Reserved
BAR0 + 256 KB
BAR0 + 252 KB
BAR0 + 248 KB
BAR0 + 244 KB
BAR0 + 240 KB
BAR0 + 64 KB

Fig. 12 Interrupt generation through the doorbell register

ized barrier sends an interrupt to all the PE nodes except for itself. Each node can proceed through the barrier if it receives an interrupt of the total number of nodes (N)-1.

To implement a barrier in a PCI Express-based interconnection network, the doorbell interrupt of the configuration space register in Fig. 12 should be used. Doorbell interrupts enable interrupt transfer between each PE node. Before counting the number of doorbell interrupts, we first need to check that the DMA transfer has been completed by PlxPci_BarrierDMANotificationWait. Then, an interrupt signal is generated by writing to the doorbell IRQ register through the function PlxPci_PlxMappedRegisterWrite. The function PlxPci_BarrierDBNotificationWait is implemented to check the number of door bell interrupts received from other PEs.

We also implemented interrupt messaging mechanism using doorbell interrupt. To realize this functionality, we have to access the NT0 link side configuration space and NT0 virtual side configuration space as shown in Fig. 12. For sending an interrupt on link side, we make use of address 0x3FC5C. For sending an interrupt on virtual side, we make use of address 0x3EC4C.

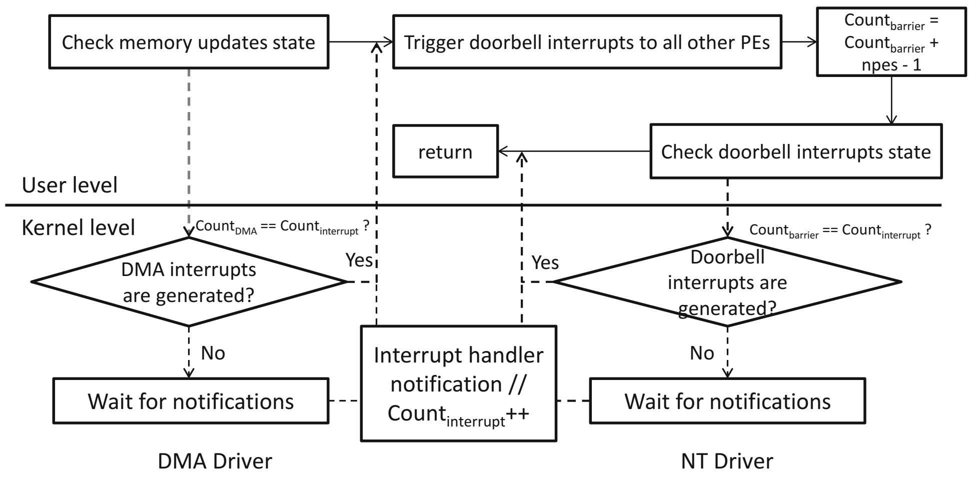
Figure 13 depicts an overall flow of barrier implementation. The PlxPci_BarrierDMANotificationWait function in Fig. 14 checks whether the DMA transfer is completed. Next, to count the number of doorbell interrupts, an interrupt is generated by controlling the doorbell IRQ register with the function PlxPci_PlxMappedRegisterWrite. Plx_BarrierDBNotificationWait() function calls PlxIoMessage(), the IOCTL function. The IOCTL function executes system call of operating system, especially IOCTL_BARRIER_DB_NOTIFICATION_WAIT.

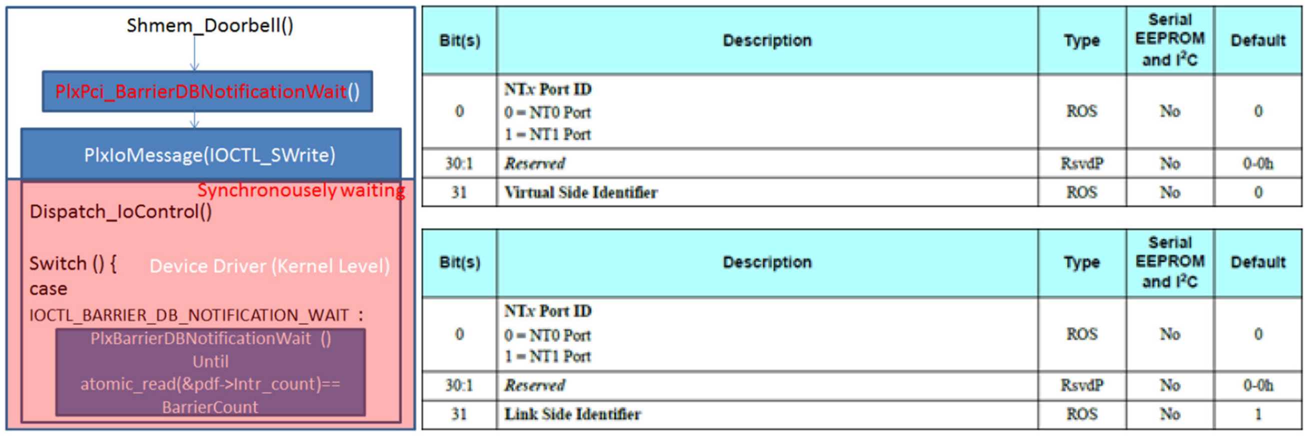
The PlxMappedRegisterRead () function uses the device driver to read the value of a register at a specific address. And this function checks the VIRTUAL_IRQ_SET and LINK_IRQ_SET bits.

3 Experimental analysis

This section provides the design and implementation of evaluation system. Table 3 shows the ex-perimental environment of the system. We make use of Core-i5 3.2 GHz, with 2 GB 1333 MHz DDR memory. The OS is Centos Linux 7

Fig. 13 Overall flow of barrier implementation


Check memory updates state
Trigger doorbell interrupts to all other PEs
Count_barrier = Count_barrier + npes - 1
return
Check doorbell interrupts state
User level
Kernel level
Count_DMA == Count_interrupt ?
DMA interrupts are generated?
Yes
No
Wait for notifications
Interrupt handler notification // Count_interrupt++
Yes
Count_barrier == Count_interrupt ?
Doorbell interrupts are generated?
No
Wait for notifications
DMA Driver
NT Driver


Shmem_Doorbell()
PlxPci_BarrierDBNotificationWait()
PlxIoMessage(IOCTL_SWrite)
Synchronously waiting
Dispatch_IoControl()
Switch () {
case
IOCTL_BARRIER_DB_NOTIFICATION_WAIT :
Device Driver (Kernel Level)
PlxBarrierDBNotificationWait ()
Until
atomic_read(&pdf->Intr_count)==
BarrierCount

Bit(s)	Description	Type	Serial EEPROM and I²C	Default
0	NTx Port ID 0 = NT0 Port 1 = NT1 Port	ROS	No	0
30:1	*Reserved*	RsvdP	No	0-0h
31	Virtual Side Identifier	ROS	No	0

Bit(s)	Description	Type	Serial EEPROM and I²C	Default
0	NTx Port ID 0 = NT0 Port 1 = NT1 Port	ROS	No	0
30:1	*Reserved*	RsvdP	No	0-0h
31	Link Side Identifier	ROS	No	1

Fig. 14 PxlPci_BarrierDBNotificationWait() register

Table 3 Testbed specification

Host PC hardware environment	
M/B	GIGABYTE GA-H61M-DS2V
Processor	Intel® CoreTM i5-3470 CPU @ 3.20GHz X 4
Memory	Samsung DDR3 1333MHz 2GB
O/S	Centos Linux 7 64bit Kernel : 3.10.0-327.el7
Interconnect hardware environment	
PCI Express NTB evaluation board	PLX PEX8749 RDK 48 Lane, 18 Port, PCI Express Gen 3 Switch
NIC	PLX PEX8732 Cable Adapter X 4
Driver	PLX SDK Device Driver

Table 4 Execution command

```
gcc –c rotget.c
gcc –o rotget rotget.o ./Library/openshmem.a –lm –ldl
```

64bit, kernel 3.10.0. The interconnection network is based on PEX8749 Chipset of PLX Technology.

Figure 15 depicts the experimental environment. Each PE make use of the shared file system by network file system (NFS) in order to see the same files and directories. We compile application source code such as rotget.c using gcc as shown in Table 4. We run the same compiled binaries on each PE through the NFC shared file system.

This section shows the performance of PCI Express link. These days we usually utilize the PCI Express generation 3, so we can get link performance up to 8 Gbit/sec, theoretically. But, the actual performance depends on variable situation such as link status, protocol overhead and so on.

To connect a host PC to PEX8749 evaluation board, we need a PEX8732 chipset based network interface card for each side. Since the evaluation host supports up to PCI Express generation 2 specification, we make use of generation 2 specification both for host and PCI Express NTB evaluation board. Evaluation software runs on Centos 7.0 linux operating system (kernel 3.10.0). For device control, PEX8749 device driver was installed as shown in Fig. 16.

First, we show the performance difference between non-DMA transfer using ordinary memcpy and PCI Express

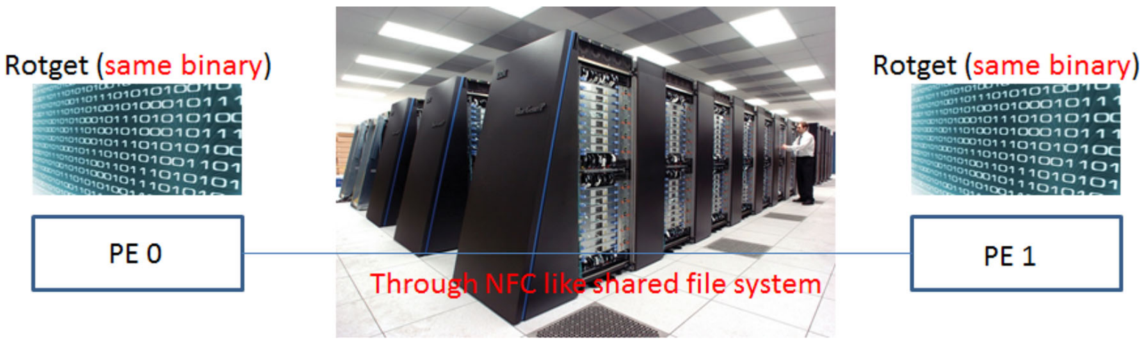


Fig. 15 Experimental environment

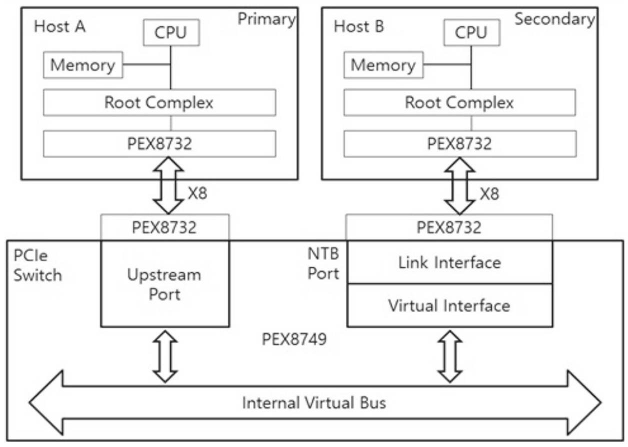


Fig. 16 Implemented interconnection network system

RDMA transfer and we also evaluate the performance improvement between Block RDMA transfer and scatter-gather RDMA.

In each experiment, each host transmits the data through buffer located in BAR (base address reg-ister). The actual physical address, the virtual address in kernel, and buffer size are send and received to each other via scratchpad register in host. In this case, the host buffer ware assigned to the same size. We transmit the buffered data to the target host after address translation by using the BAR register.

We limit the buffer size to a maximum of 1 MB. In this experiment, we transfer the data to Host A from Host B by using one DMA channel. We measured the transmission rate with respect to the size and number of buffers. When an interrupt occurs at the end of each transfer, we measured the average data transfer rate as a way to send data back and forth.

The left side of Fig. 17 depicts the evaluation between non-DMA transfer and block RDMA transfer and between Block DMA and SGDMA. The x axis shows the size of transfer data and the y axis represents the transfer rate as MB/s, commonly in both graphs. When the size of the buffer is small, memcpy and DMA transfer shows low performance at the same time. The larger the size of the buffer gets better performance in DMA transfer. This is because the data transfer at a time to increase. When the data size is below 2 kB, memcpy showed higher transmission rate than DMA transfer. Even if we increase the buffer size in memcpy, it tends to be satisfied at the transfer rate of up to 150 MB/s. In more than 4 kB buffer size, we got a relatively high performance of up to about 500 MB/s in DMA transfer.

The SGDMA transfer descriptor has information which is required for DMA such as source address, destination address, and byte count. This descriptor enables us to transmit multiple DMA buffers or discontinuous address spaces. The right side of Fig. 17 shows the performance of SGDMA transfer. The right side of Fig. 17 shows the evaluation between block RDMA transfer and scatter-gather RDMA transfer. The x axis shows the size of transfer data and the y axis represents the transfer rate as MB/s, commonly in both graphs. We used the 8, 32, 128, 256 descriptors for SGDMA transfer. We specify the data size for each descriptor to a value obtained by dividing evenly the same size to the buffer size. The smaller the size of the buffer, the more the number of descriptor, we got lower the transmission rate of the SG DMA. As a result of execution with 8 descriptors, we got up to 520 MB/s speed. This is slightly higher than the performance Block DMA. Evaluation result shows that the non-DMA transfer performance is 150 MB/s and the Block RDMA transfer is improved to 500 MB/s. Finally, scatter-gather DMA performance is shown up to 520 MB/s. In this paper, it delivers high-performance computing and widely used Interconnect Technology of the Non-Transparent PCI Express Switch and connect two PC interconnect system using cluster computing Bridging that the DMA transfer performance were analyzed between two PC.

Non-DMA (memcpy) and DMA Block exhibited a maximum transmission speed of each to increase the larger the

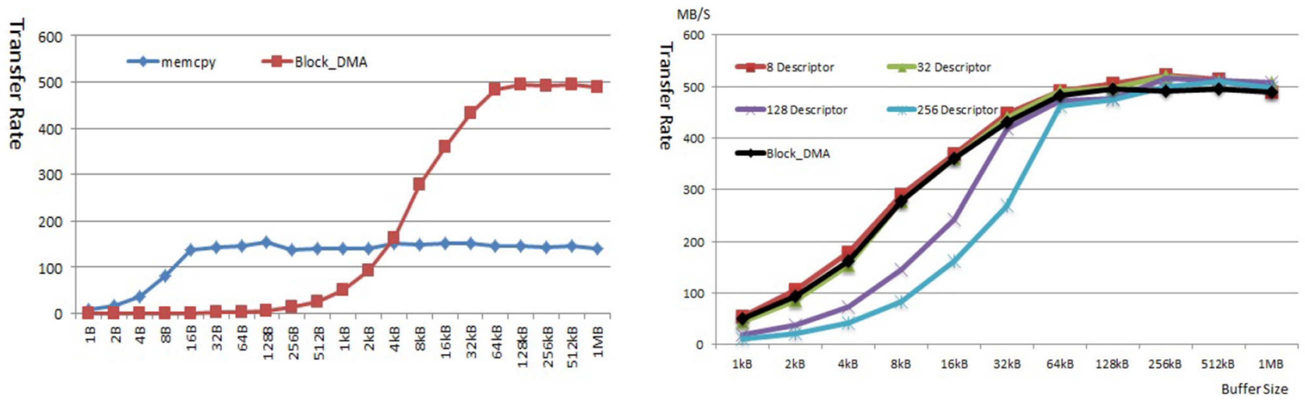
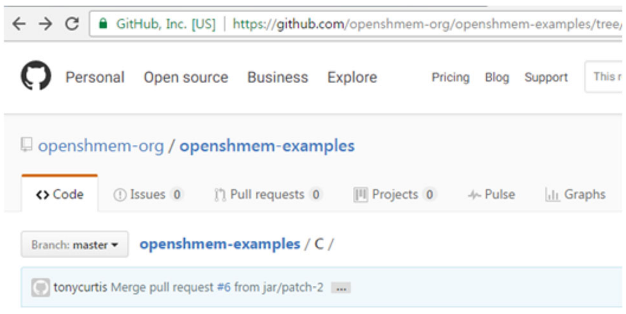


Fig. 17 Evaluation result of non-DMA and block DMA transfer, block DMA and SGDMA (MB/sec)



```
← → C  GitHub, Inc. [US] | https://github.com/openshmem-org/openshmem-examples/tree/
Personal  Open source  Business  Explore          Pricing  Blog  Support  This r
openshmem-org / openshmem-examples
<> Code   Issues 0   Pull requests 0   Projects 0   Pulse   Graphs
Branch: master ▾   openshmem-examples / C /
tonycurtis Merge pull request #6 from jar/patch-2  ...
```

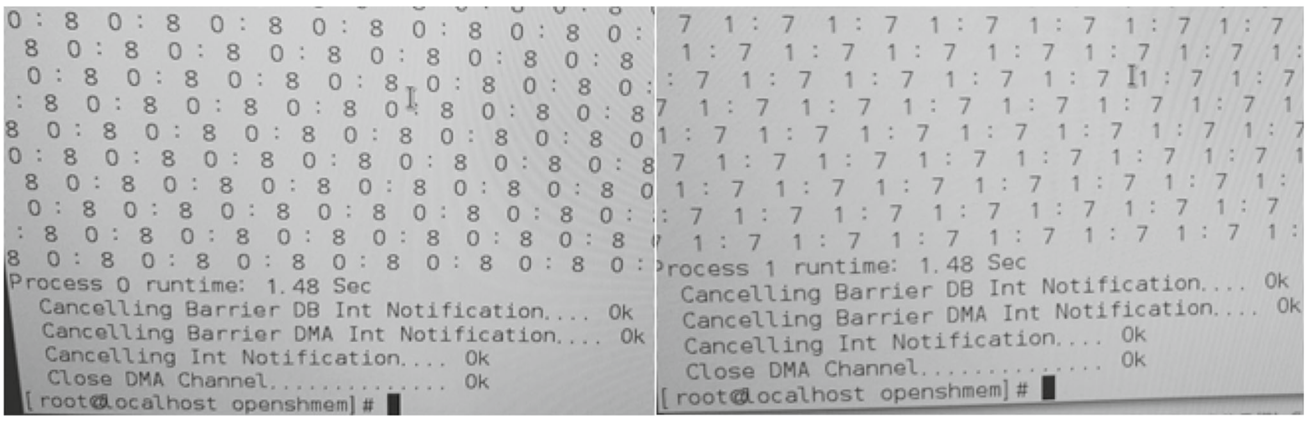
Fig. 18 Result of block DMA and SGDMA transfer (MB/sec)

size of the buffer transfer rate of about 150, 500 MB/s. For SGDMA larger the size of the buffer, the number of descriptor was confirmed to represent a slightly better performance than the DMA Block The transfer rate is low in transmission rate of up to about 520 MB/s. Due to the limitation of allocation for physically contiguous buffer the performance resulted in a lower performance than theoretical bound. By reducing the overhead due to buffer copy and non-contiguous physical buffer, we can achieve higher bandwidth up to theoretically bound.

Figure 18 shows the benchmark application which we used in this experiment. 7 applications of github.com/openshmem-examples are working on our platform : rotget.c, rotput.c, randput.c, shmem_all.c, dip.c, swap.c, and matrix multiplication.c. The reason why the rest of applications on the github.com are not working is because of limited APIs support at the initial stage of our implementation.

Figures 19, 20, and 21 provide the results of execution the applications mention above. The rotput.c rotates PE id to right neighbor (dest), with wrap-around. The rotget.c gets from right neighbor, with wrap-around. randput.c gets from 0 (master) to some other random PE. The shmem_all.c shows how to use shmem_put to simulate MPI_Alltoall. Each processor send/rec a different random number to/from other processors. The dip.c is a double value put application from PE 0 to PE 1 to shmem_malloc'ed variable. The swap.c swap values between odd numbered PEs and their right (modulo) neighbor. The shmem_matrix.c calculates the product of 2 matrices A and B based on block distribution. This is adopted from the mpi implementation of matrix multiplication based on 1D block-column distribution.

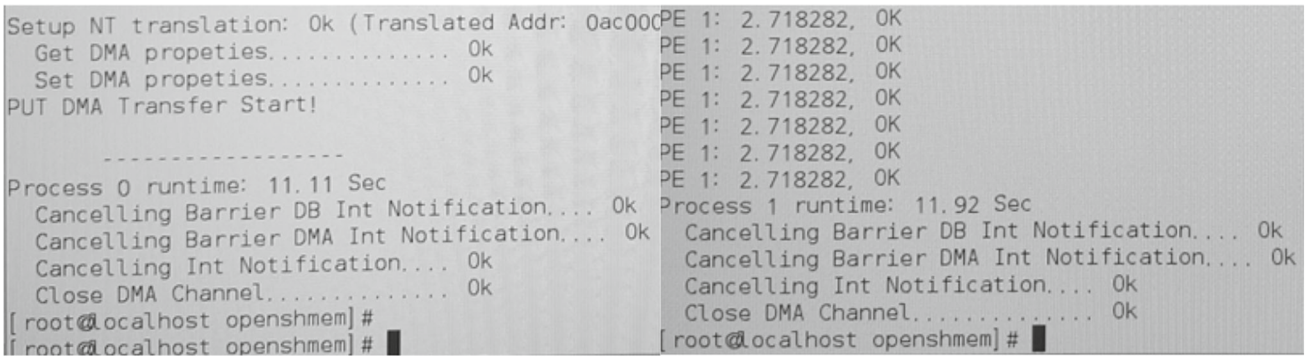
From Figs. 22, 23, 24, 25, 26 and 27 are the result of execution. The x-axis is block size and y-axis is the proportional execution time. All applications, with exception of shmem_all, commonly provide that larger block size will result in longer execution time. The shmem_all application



```
0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 :
 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8
 0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 :
: 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8
8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0
0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8
 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0
 0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 :
: 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8
8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 : 8  0 :
Process 0 runtime:  1.48 Sec
   Cancelling Barrier DB Int Notification....  Ok
   Cancelling Barrier DMA Int Notification....  Ok
   Cancelling Int Notification....  Ok
   Close DMA Channel...............  Ok
[root@localhost openshmem]# 
```

```
 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7
 1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 :
: 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7
7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1
1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7
 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1
 1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 :
: 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7
  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 : 7  1 :
Process 1 runtime:  1.48 Sec
   Cancelling Barrier DB Int Notification....  Ok
   Cancelling Barrier DMA Int Notification....  Ok
   Cancelling Int Notification....  Ok
   Close DMA Channel...............  Ok
[root@localhost openshmem]# 
```

Fig. 19 Result of execution rotget.c

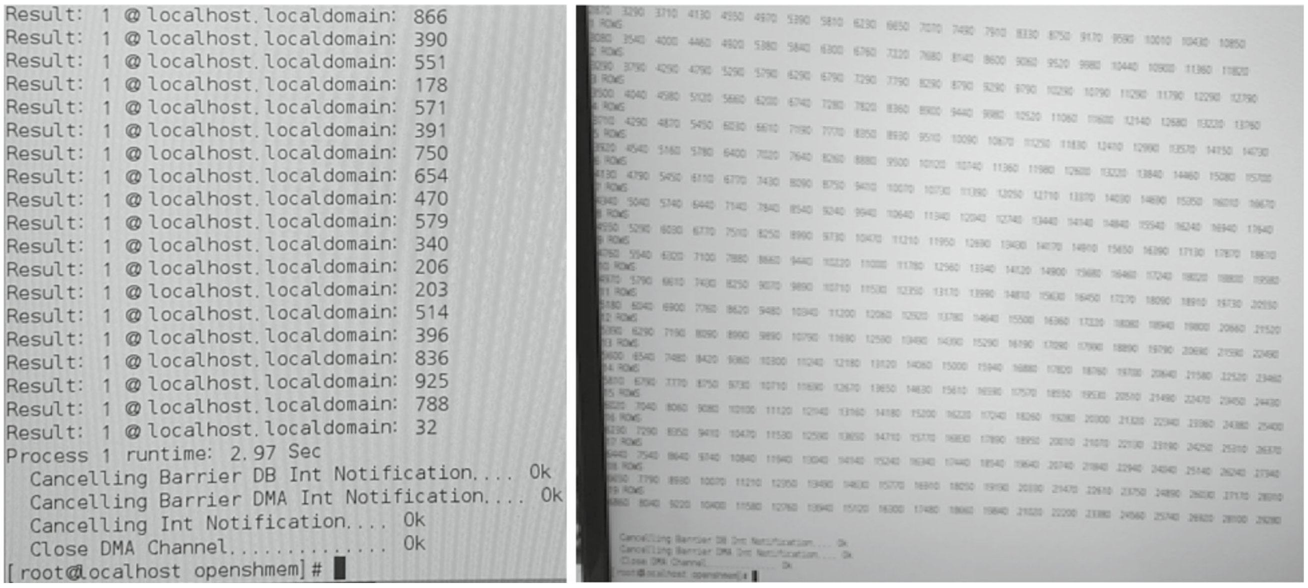


```
Setup NT translation: Ok (Translated Addr: 0ac00C
  Get DMA propeties..............  Ok
  Set DMA propeties..............  Ok
PUT DMA Transfer Start!

       ------------------
Process 0 runtime:  11.11 Sec
  Cancelling Barrier DB Int Notification....  Ok
  Cancelling Barrier DMA Int Notification....  Ok
  Cancelling Int Notification....  Ok
  Close DMA Channel..............  Ok
[root@localhost openshmem]#
[root@localhost openshmem]# 
```

```
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
PE 1:  2.718282,  OK
Process 1 runtime:  11.92 Sec
  Cancelling Barrier DB Int Notification....  Ok
  Cancelling Barrier DMA Int Notification....  Ok
  Cancelling Int Notification....  Ok
  Close DMA Channel..............  Ok
[root@localhost openshmem]# 
```

Fig. 20 Result of execution dip.c



```
Result:  1 @localhost.localdomain:  866
Result:  1 @localhost.localdomain:  390
Result:  1 @localhost.localdomain:  551
Result:  1 @localhost.localdomain:  178
Result:  1 @localhost.localdomain:  571
Result:  1 @localhost.localdomain:  391
Result:  1 @localhost.localdomain:  750
Result:  1 @localhost.localdomain:  654
Result:  1 @localhost.localdomain:  470
Result:  1 @localhost.localdomain:  579
Result:  1 @localhost.localdomain:  340
Result:  1 @localhost.localdomain:  206
Result:  1 @localhost.localdomain:  203
Result:  1 @localhost.localdomain:  514
Result:  1 @localhost.localdomain:  396
Result:  1 @localhost.localdomain:  836
Result:  1 @localhost.localdomain:  925
Result:  1 @localhost.localdomain:  788
Result:  1 @localhost.localdomain:  32
Process 1 runtime: 2.97 Sec
  Cancelling Barrier DB Int Notification.... Ok
  Cancelling Barrier DMA Int Notification.... Ok
  Cancelling Int Notification.... Ok
  Close DMA Channel.............. Ok
[root@localhost openshmem]# 
```

Fig. 21 Result of execution randput.c and matrix multiplication.c

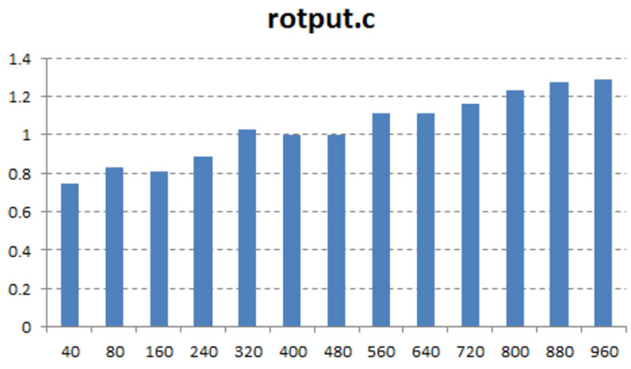


Fig. 22 Execution result of rotput.c

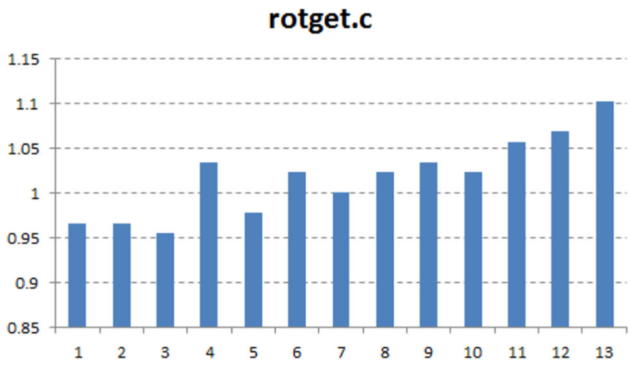


Fig. 23 Execution result of rotget.c

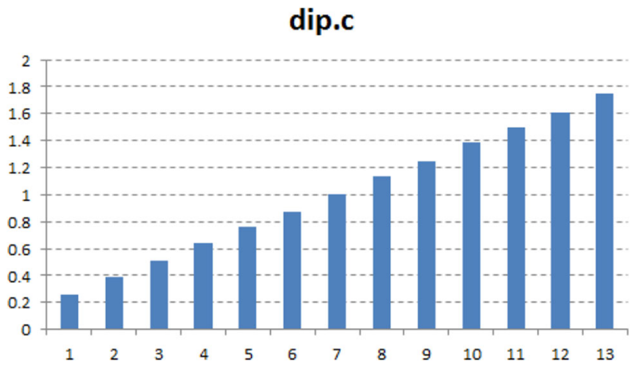


Fig. 24 Execution result of dip.c

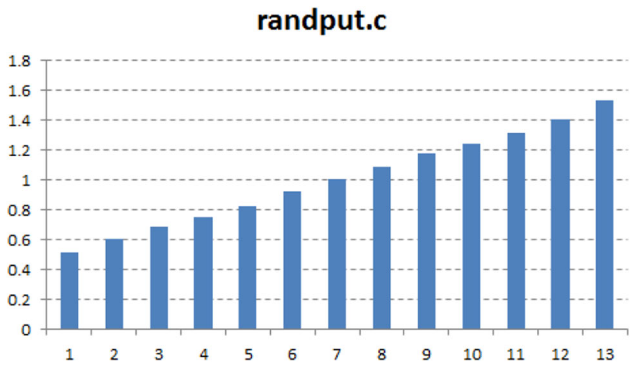


Fig. 25 Execution result of randput.c

maintains almost same level of performance with the increasing block size. This is because the shmem_all is collective operation such as MPI_Alltoall(). In shmem_all operation, the contents of send buffer on each PE are distributed in group of some elements to all process. So, each process receives some elements into receive buffer. Therefore, shmem_all requires many send/receive and synchronization, not directly proportional to RDMA performance.

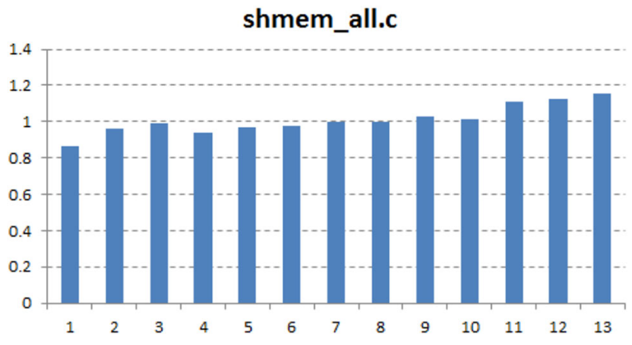


Fig. 26 Execution result of shmem_all.c

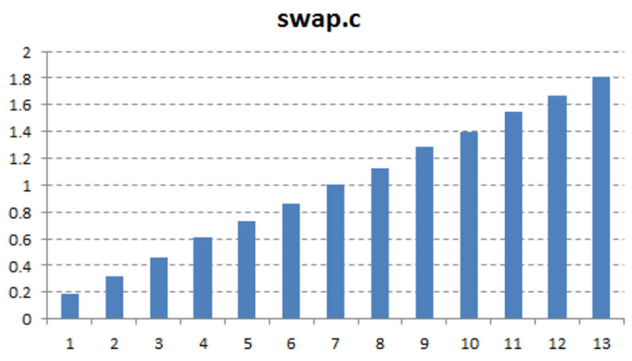


Fig. 27 Execution result of swap.c

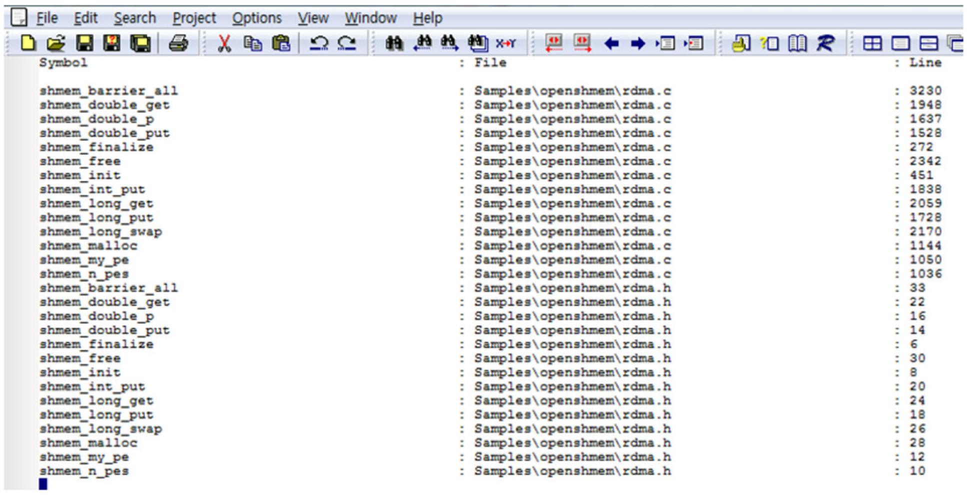
The API we have implemented is shown in the following Fig. 28. Since the result of retrieving the API from the project source code, the same API was searched once in the c file and once in the .h file.It indicates that the initial version of GPI / OpenSHMEM works well in the PCIe interconnection network as originally intended.

4 Conclusions

Although PCI Express-based interconnection network environment is not popular now, we have established a PCI Express-based interconnection network experimental environment. OpenSHMEM technology has already become widespread in HPC interconnection networks such as Infiniband and RoCE. However, research on PCI Express-based HPC interconnection networks is only the beginning of the world. We conducted this study in case the PCI Express NTB-based HCA adapter is widely available. Through this research, we propose the possibility of using it as the OpenSHMEM framework of the next generation Interconnection Network. This research is applicable to HCA for HPC interconnection network. This study is also meaningful in terms of securing next generation system software technology. We have learned the driving and control technology of the PLX PEX8749. We designed and implemented the initial OpenSHMEM framework and performed verification of the correctness and compatibility of the framework. Finally, a few APIs of our initial OpenSHMEM implementation are demonstrated by running 7 OpenSHMEM examples of github.com.

Acknowledgements This research has been performed as a subproject of Project No. K-17-L01-C01 (Development of key enabling technologies for massively parallel and high-density computing) supported by the KOREA INSTITUTE of SCIENCE and TECHNOLOGY INFORMATION (KISTI). This research is also jointly Supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2016R1C1B1012189).

Fig. 28 Supported APIs of our initial OpenSHMEM



```
Symbol                                   : File                            : Line

shmem_barrier_all                        : Samples\openshmem\rdma.c        : 3230
shmem_double_get                         : Samples\openshmem\rdma.c        : 1948
shmem_double_p                           : Samples\openshmem\rdma.c        : 1637
shmem_double_put                         : Samples\openshmem\rdma.c        : 1528
shmem_finalize                           : Samples\openshmem\rdma.c        : 272
shmem_free                               : Samples\openshmem\rdma.c        : 2342
shmem_init                               : Samples\openshmem\rdma.c        : 451
shmem_int_put                            : Samples\openshmem\rdma.c        : 1838
shmem_long_get                           : Samples\openshmem\rdma.c        : 2059
shmem_long_put                           : Samples\openshmem\rdma.c        : 1728
shmem_long_swap                          : Samples\openshmem\rdma.c        : 2170
shmem_malloc                             : Samples\openshmem\rdma.c        : 1144
shmem_my_pe                              : Samples\openshmem\rdma.c        : 1050
shmem_n_pes                              : Samples\openshmem\rdma.c        : 1036
shmem_barrier_all                        : Samples\openshmem\rdma.h        : 33
shmem_double_get                         : Samples\openshmem\rdma.h        : 22
shmem_double_p                           : Samples\openshmem\rdma.h        : 16
shmem_double_put                         : Samples\openshmem\rdma.h        : 14
shmem_finalize                           : Samples\openshmem\rdma.h        : 6
shmem_free                               : Samples\openshmem\rdma.h        : 30
shmem_init                               : Samples\openshmem\rdma.h        : 8
shmem_int_put                            : Samples\openshmem\rdma.h        : 20
shmem_long_get                           : Samples\openshmem\rdma.h        : 24
shmem_long_put                           : Samples\openshmem\rdma.h        : 18
shmem_long_swap                          : Samples\openshmem\rdma.h        : 26
shmem_malloc                             : Samples\openshmem\rdma.h        : 28
shmem_my_pe                              : Samples\openshmem\rdma.h        : 12
shmem_n_pes                              : Samples\openshmem\rdma.h        : 10
```

References

1. Helal, A.A., Kim, Y.W., Ren, Y., Choi, W.H.: Design and implementation of an alternate system inter-connect based on PCI Express. J. Inst. Electron. Inform. Eng. **52**(8), 74–85 (2015)
2. Liu, J., Mamidala, A., Vishnu, A., Panda, D.K.: Evaluating infiniband performance with PCI Express. IEEE Micro **24**(1), 20–29 (2005)
3. Heymian, W.: PCI Express multi-root switch reconfiguration during system operation. M. Eng. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge (2011)
4. Krishnan, V.: Towards an integrated IO and clustering solution using PCI express. 2007 IEEE International Conference on Cluster Computing [Online]. pp. 259–266. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4629239 (2007)
5. Mohrmann, L., Tongen, J., Friedman, M., Wetzel, M.: Creating multicomputer test systems using PCI and PCI Express. IEEE AUTOTESTCON [Online]. pp. 7–10. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5314043 (2009)
6. RCT Magazine. Enabling Multi-Host System Designs with PCI Express Technology [Internet]. http://www.rtcmagazine.com/articles/view/100015.8
7. Jo, H., Jeong, J., Lee, M., Choi, D.: Exploiting GPUs in Virtual Machine for BioCloud, BioMed Research International, vol. 2013, Article ID 939460 (2013)
8. Vienne, J., Chen, J., Wasi-ur-Rahman, M., Islam, N., Subramoni, H., Panda, D.: Performance Analaysis and Evaluation of Infiniband FDR and 40GigE RoCE on HPC and Cloud Computing Systems, IEEE 20th Annual Symposium on High-Performance Interconnects (2012)
9. Choi, M., Park, J.H.: Feasibility and performance analysis of RDMA transfer through PCI Express. J. Inform. Process. Syst. **13**(1), 95–103 (2017)
10. Top500.org.: Interconnect Family Statistics [Internet]. http://top500.org/statistics/list (2015)
11. Chen, L., Zhou, W., Wu, Q.: A design of high-speed image acquisition card based on PCI EXPRESS. 2010 International Conference on Computer Application and System Modeling (ICCASM 2010) [Online]. vol, 3, pp. V3-551–V3-554. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5620696 (2010)
12. Kavianipour, H., Bohm, C.:. High performance FPGA-based scat-ter/gather DMA interface for PCIe. Nuclear Science Symposium and Medical Imaging Confer-ence (NSS/MIC), 2012 IEEE [Online]. pp. 1517–1520. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6551364 (Oct. 27 2012–Nov. 3 2012)
13. Buschettu, A., Sanna, D., Concas, G., Eros, F.: A platform based on kanban to build taxonomies and folksonomies for DMS and CSS. J. Converg. (JoC) **6**(1), 1–8 (2015)
14. Rota, L., Caselle, M., Chilingaryan, S., Kopmann, A., Weber, M.: A new DMA PCIe architecture for Gigabyte data transmission. Real Time Conference (RT), 2014 19th IEEE-NPSS [Online]. pp. 1–2. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7097561 (2014)
15. ExpressLane PEX8749 PCI Express Gen 3 Multi-Root Switch with DMA Data Book, PLX Technology (2013)
16. Richter, A., Herber, C., Wild, T., Herkersdorf, A.: Resolveing Performance Interference in SR-IOV Setups with PCIe Quality-of-Service Extensions. Euromicro Conference on Digital System Design (2016)
17. Sinha, A., Lobiyal, D.K.: Performance evaluation of data aggregation for cluster wireless sensor network. Human-centric Comput. Inform. Sci. (HCIS) **3**(13), 2–17 (2013)
18. Naoui, M., Mahmoudi, S., Belalem, G.: Feasibility study of a distributed and parallel environment for implementing the standard version of AAM model. J. Inform. Process. Syst. (JIPS) **12**(1), 149–168 (2016)
19. Mellor-crummey, J.M.: Algorithm for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)

Cheol Shim was born in Cheongju-si, Republic of Korea, in 1990. He received the B.S. degree in Computer Engineering from the Chungbuk National University, Cheongju-si, Republic of Korea, in 2016, and He is currently in the Master's course of Information and Communication Engineering at the Chungbuk National University, Cheongju-si, Republic of Korea. His research interests include Cloud Computing and Embedded Software, Real-Time Operating System, Internet of Things in Embedded systems and software application in Web Application Server.

Kwangho Cha received the B.S. degree in Computer Science from Soongsil University, Korea in 1999 and the M.E. degree in Information and Computer Engineering from Information and Communications University (ICU), Korea in 2002 and the Ph.D. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2012. Since 2002, he has been with Division of Supercomputing of Korea Institute of Science and Technology Information, where he is currently a senior research staff. His research interests include parallel and cluster computing, parallel file system, and high-performance computer systems.

Min Choi received the B.S. degree in Computer Science from Kwangwoon University, Korea, in 2001, and the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2003 and 2009, respectively. From 2008 to 2010, he worked for Samsung Electronics as a Senior Engineer. Since 2011 he has been a faculty member of Department of Information and Communication of Chungbuk National University. His current research interests include high performance computing, cloud computing, interconnection network, and embedded computing.