Regular Paper

Architecture and Performance of Dynamic Offloading Mechanism for Maestro2 Cluster Network

Keiichi Aoki,† Koichi Wada,† Hiroki Maruoka† and Masaaki Ono†

In this paper, an architecture of software environment to offload user-defined software modules to Maestro2 cluster network, named Maestro dynamic offloading mechanism (MDO), is described. Maestro2 is a high-performance network for clusters. The network interface and the switch of Maestro2 have a general-purpose processor tightly coupled with a dedicated communication hardware. MDO enables the users to offload software modules to both the network interface and the switch. MDO includes a wrapper library with which offload modules can be executed on a host machine without rewriting the program. The overhead and the effectiveness of MDO are evaluated by offloading collective communications.

1. Introduction

Several networks for cluster computing have been developed so far such as Myrinet [1] or QsNet [2]. These networks are built with a gigabit-class physical layer and provide low latency and high throughput performance. To achieve high communication performance between application layers with these networks, several techniques have been proposed for communication software. For example, a specialized communication library for high performance network such as GM [3] and PM [4] introduces zero-copy communication [5] for reducing latency. On the other hand, a host machine consumes much computing resources to process communication protocol or to control network devices [6),7]. One of the promising technique to remedy this problem is to offload communication related processing to network devices.

We can find several researches or products that propose offloading communication protocol to network devices; such as TOE [8)~10)] or offloaded-MPI [11]. Although they have succeeded in increasing communication performance, the host machine still has to control network hardware to handle communication frequently because the unit of offloading is small. By offloading whole communication library or a part of the application, the host machine can be released from the burden of the communication. This increases an opportunity of overlapping computation with communication.

† Department of Computer Science, University of Tsukuba

To offload large software modules that used to be processed on a host machine to network devices, network devices are required to introduce a high performance processor instead of special purpose processor such as a network processor, and a high-capacity memory. We have developed a high performance cluster network called Maestro2 [12]. Maestro2 is composed of network interfaces and switch boxes. Both the network interface and the switch box include a general purpose processor and a high-capacity memory so that user-defined software modules can be executed.

To provide users with an efficient and flexible way for developing offload modules, the offloading mechanism for such a network is required to have the capabilities of 1) dynamic loading/unloading of modules, 2) request handling between the host and the network devices, 3) low-overhead invoking of communication, and 4) support for debugging modules.

In this paper, a dynamic offloading mechanism for Maestro2, which can offload user-defined software modules dynamically to the network devices, is proposed. This mechanism includes a wrapper library that enables the offload modules to be executed on the host machines for debugging the modules.

The rest of this paper is organized as follows. In Section 2, the architecture of the Maestro2 network is described. Section 3 proposes a dynamic offloading mechanism for Maestro2. The results of the evaluation are shown and discussed in Section 4. Section 5 presents related works. Finally, Section 6 presents concluding remarks and plans for further work.

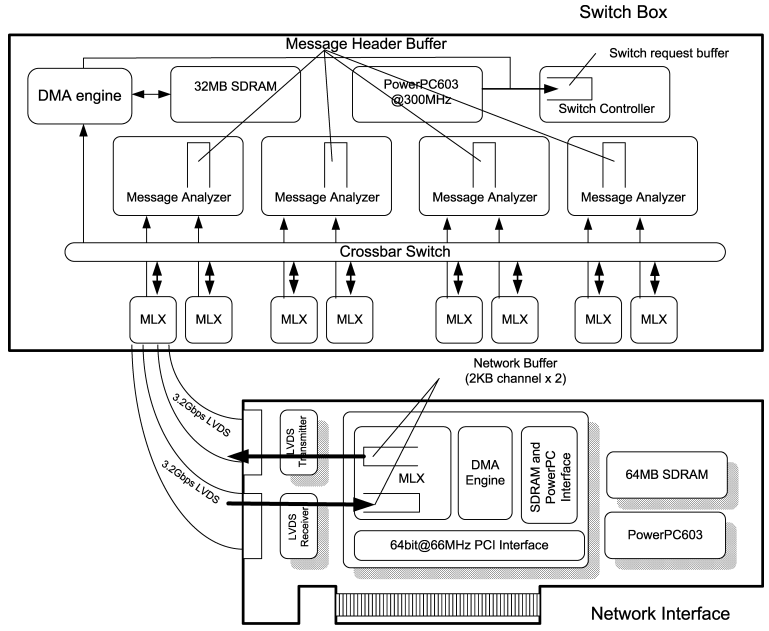

Fig. 1 Architecture of Maestro2.

2. Maestro2 Cluster Network

2.1 Architecture Overview

As described in the previous section, we have developed Maestro2 cluster network which has the capability of executing user-defined software. In this section, we will describe briefly the architecture and implementation of the Maestro2 cluster network.

The Maestro2 cluster network consists of network interfaces and a switch box as shown in **Fig. 1**. Network interfaces are connected to each host machine via the PCI bus and exchange messages with the host machine. The switch box is connected to up to eight network interfaces via an LVDS (Low Voltage Differential Signaling) physical layer [13] and is responsible for switching messages.

(1) Network interface
The network interface (NI) is composed of an LVDS transmitter/receiver, 8 K-byte network buffer, MLC-X link layer protocol, a PCI interface, a PowerPC603e [14] 300 MHz and a 64 M-byte SDRAM. The MLC-X, the PCI interface, and the network buffer are implemented in a Xilinx VirtexII FPGA [15]. The MLC-X controls the LVDS transmitter/receiver bidirectionally. The LVDS transmitter/receiver is connected to the switch box and transfers data at 6.4 Gbps (3.2 Gbps + 3.2 Gbps full-duplex). The PCI interface maps the address of the SDRAM into the host machine's address and the host machine's memory address into the PowerPC's address.

(2) Switch box
The switch box (SB) currently includes eight LVDS transmitter/receiver, four SB interfaces, a crossbar switch, a PowerPC603e 300 MHz, a 32 M-byte SDRAM, and a switch controller. The SB interfaces are composed of a message analyzer, two MLC-X link layer protocol, and a 16 K-byte network buffer. It communicates with network interfaces via LVDS. MLC-Xs and network buffers in the switch box are similar to the ones in the network interface. Each message analyzer is connected to two MLC-Xs. It picks up headers from messages and passes them to the PowerPC processor. The PowerPC analyses the headers received from the message analyzer and controls the switch controller.

As described above, both of the network interface and the switch box of Maestro2 cluster network have a general purpose processor and large-capacity RAM on board. By using this processor, Maestro2 cluster network can control hardware and handle protocol or other communication processing independently of host ma-

chines.

2.2 Message Passing Library

We have also developed a message passing library that can extract maximal performance of the network, named MMP [16]. This library is implemented based on a protocol offloading technology, and includes functions for peer-to-peer message transfer among host machines and synchronization.

The firmware of MMP controls the network hardware and handles the protocol processing such as an addition of a message header. The application interface functions of MMP are designed as non-blocking to overlap computation in application programs with communication processing. We confirmed that this design freed the host processor and the host bus from communication processing and increased the throughput of communication.

3. Maestro Dynamic Offloading Mechanism

3.1 Architecture of MDO

Maestro dynamic offloading mechanism (MDO) is a software environment for offloading user-defined modules to the Maestro2 cluster network. MDO is composed of one or more user-defined modules, the MDO library, the module wrapper library, and the firmware for the network interface and the switch box. **Figure 2** summarizes the architecture of MDO.

The MDO firmware is currently implemented as a part of the MMP firmware for the network interface and the switch box. It loads the modules transferred from the MDO library. Additionally it calls the modules when it receives a request for the module from the MDO library or receives the message for the module from network.

The MDO library is the library for application programs on a host machine. It includes a dynamic linker/loader to load user-defined modules dynamically into network devices and a requester to handle requests for modules issued from application programs.

A user-defined module includes a native executable binary for the processor on the network interface and/or the switch box. It also includes the information for relocation. A user can develop modules by a compiler that can generate ELF (Executable and Linking Format) files, which include information for relocation. It is called from the firmware when the MDO firmware detects a request from the MDO library, or the MDO firmware receives message(s) for the module. It controls the peripheral network hardware or performs computations after being called. A native binary in the module is called by the firmware and controls the hardware on the network device directly to invoke communication with low overhead.

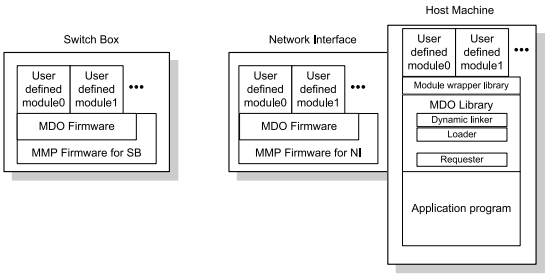

Fig. 2 Architecture of MDO.

3.2 Interface Functions of MDO

The MDO library provides interfaces to application programs for loading modules and issuing requests to them. We will detail about these interfaces below:

(1) `MMP_Regist_mod_NI(...)`
This function relocates the specified module and transfers it to an unused memory region on the network interface. To relocate modules, this function investigates the address information of the MDO firmware and of the modules that have already loaded on the network interface, and builds the global symbol table. Then it resolves symbols' addresses in the module by referring the table. Finally, it transfers the relocated module and registers the information of that module such as the functions' entry addresses to the MDO firmware on the network interface.

(2) `MMP_Regist_mod_SB(...)`
This function registers the module to the switch box. It relocates the module in the same method as in `MMP_Regist_mod_NI()`. Then it transmits the relocated module to the switch box through the network interface.

(3) `MMP_Mod_req(...)`
This function is called when an application program issues a request to a module on the network interface. The arguments passed from the application program are stored in the request queue on the network interface.

(4) `MMP_Mod_recv_post(...)`
This function is used to notify a mod-

ule of the target address of the host machine's memory for DMA, in order for modules to transfer messages from network buffer to the memory of the host machine. The application program passes the address information, which is either a base address or an offset, to modules by this function. The target address is calculated from the address information by modules.

(5) `MMP_Get_mod_req_stat(...)`
This function enables an application program to receive calculation results from the module or to wait for the completion of the execution of the module.

Meanwhile, the MDO module has to contain callback functions to handle requests from an application and messages from other modules. Details of these functions are as follows:

(1) `NI_mod_hostreq_handler(...)`
Depending on the requests issued from an application program by `MMP_Mod_req()`, this function in the module loaded on network interface is called.

(2) `NI_mod_recvmsg_handler(...)`
When the firmware on a network interface receives a message, it calls this function. Generic information such as the size of the message and the source address of the message is passed as arguments. Pre-post information for message reception provided by using `MMP_Mod_recv_post()` is also passed.

(3) `SB_mod_recvheader_handler(...)`
The MDO firmware on a switch box calls this function whenever it receives a message for the offloaded module on the switch box. The module controls the hardware on the switch box to transfer the message to other network interface(s) or to the memory on the switch box to process data included in the message.

3.3 Inter-process Communication in MDO

In MDO, inter-process communication can be categorized into two types: a) request exchange between an application program on a host machine and a module on a network interface. b) message exchange among modules on network interfaces or between modules on a network interface and on a switch box. **Figure 3** depicts details of the inter-process communication.

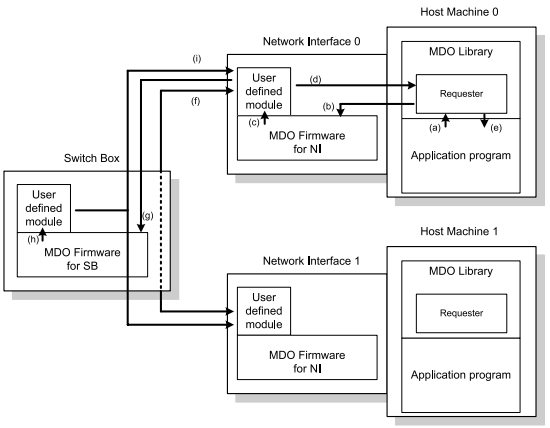

Fig. 3 Inter-process communication in MDO.

An application program on a host machine can issue requests with arguments through MDO library to the module on the network interface (Fig. 3 (a), (b)). When the application program issues the requests to the module, the MDO firmware on the network interface calls the function with the specific symbol name in the module with arguments passed from the application (Fig. 3 (c)). The module may transmit several messages per request if necessary, or it may transmit nothing and do only the computation. If the module needs to return the result to the application program, the result can be transferred via the MDO library (Fig. 3 (d), (e)).

Modules on network interfaces and on switch boxes can exchange messages among themselves without requests from host machines. Similarly, modules on network interfaces can exchange messages among themselves through the switch box (Fig. 3 (f)). Thus, with MDO, host machine can be freed from the communication burden of relaying messages performed in collective communication. This reduces the traffic in the host machine's bus or overheads in frequent interactions between the communication library and an application program, and increases the communication performance.

As described above, the module on the network interface and the one on the switch box can also exchange messages in MDO. When the MDO firmware on the switch box receives messages from the module on the network interface, it calls the module on the switch box (Fig. 3 (g), (h)). The callee module can transmit messages to one or more modules on the network interface after executing designated procedures (Fig. 3 (i)). This feature is effective in reduc-

ing the number of messages in the network and the number of phases of communication when a module or an application program needs to broadcast, or sum up the calculation results in multiple host machines.

3.4 Module Wrapper Library

By only being recompiled and linked with the module wrapper library, the user-defined modules for the network interface can be executed on the host machine without modifying their source codes. This is useful for debugging and examining adequate load partitioning between the host machine and the network interface. When a module is compiled for a host machine, the MDO library calls it via the module wrapper library. The MDO library also hooks function calls in modules that control network hardware directly. Then, MDO translates a hardware-dependent value, such as a physical address, to make the module to work correctly on a host machine.

4. Evaluation

4.1 Experimental Environment

We prepared the environment as shown in **Table 1** for experiments. We performed four experiments on this environment. We will evaluate the performance of MDO by comparing with the performance of the conventional protocol offloading. In this evaluation, MDO is compared with the message passing library MMP, in which a lower communication protocol is offloaded statically. We use `gettimeofday()` function to measure the time on the host machine in all experiments.

4.2 Performance of Strided Data Transfer

We compared the throughput of strided data transfer, which is frequently observed in parallel processing when arrays are partitioned in block-cyclic manner and assigned to the multiple hosts, when using MDO with the case when using MMP. In this experiment, we measured throughputs when the experimental program transfers 100 non-contiguous blocks from a host machine to another host machine with varying block size. The application program on host machine passes the block size and the number of blocks to transfer to the module on the network interface with MDO. Then the module on the network interface calculates the addresses of the blocks and controls the DMA hardware to transfer these blocks. The application program using MMP calls communication functions of MMP to transfer each block.

Table 1 Experimental environment.

CPU	Intel Xeon 2.8 GHz
Memory	1 G-byte
Network	Maestro2 cluster network
OS	Debian GNU/Linux 3.1

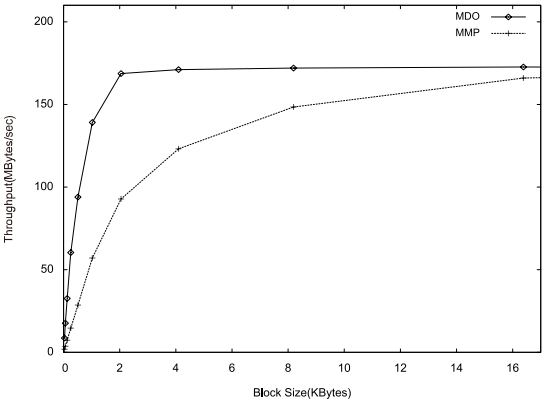

Fig. 4 Throughput comparison of strided data transfer using MDO and MMP with varying data size.

Figure 4 shows the comparison of the throughputs. The horizontal axis of the graph shows the size of one block to be transferred.

As Fig. 4 shows, MDO achieves a higher throughput than MMP for all block sizes. We confirmed that the throughput of MDO is 4.8 times as high as one of MMP at maximum.

The experimental program using MMP needs to issue send/receive requests to the firmware on the network interface for every single transfer of blocks. In contrast, the module on the network interface and the experimental program on the host machine exchange a request once at the beginning of the program. Then, the module on the network interface decides regions to be transferred and controls the DMA hardware. The firmware can set DMA parameters for the subsequent transfer soon after one block transfer has completed. This reduces the idle time of the DMA engine and increases the throughput of strided data transfer.

4.3 Offloading Collective Communication

We have implemented three collective communications using MDO for this evaluation: synchronization, broadcast, and allreduce.

When a host machine exchanges messages with several other host machines by using MDO in these communications, the module on the switch box communicates with the modules on network interfaces. In the case of synchroniza-

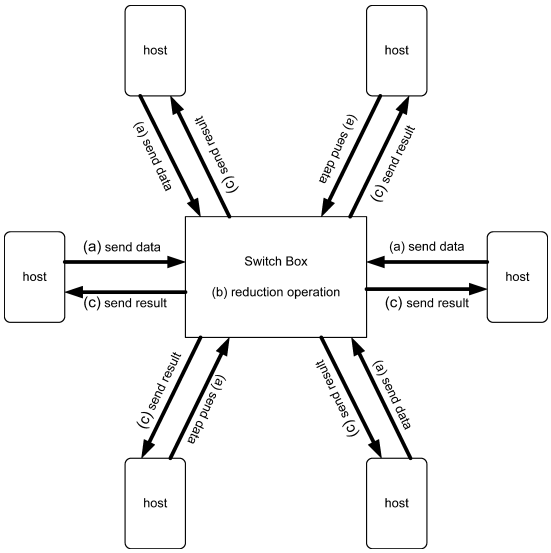

Fig. 5 Allreduce communication with MDO.

tion, a module on a network interface sends a message, which notifies the module on a switch of the arrival of a synchronization point. When the module on a switch has received messages from all the modules on network interfaces that joined the synchronization, it controls the crossbar to broadcast the release messages. In broadcast, the module on the switch box receives messages from the initiator's module. Then it controls the crossbar switch on the switch box to broadcast the messages to receivers' modules.

Figure 5 shows the communication flow in the allreduce communication with MDO. When the application program invokes a collective communication, host machines send data to the module on the switch box. Upon receiving the data, the module performs calculation. Finally the module transmits the results to all the host machines simultaneously. Therefore, the number for message needed in the allreduce communication is decreased compared with the peer-to-peer communication between host machines.

Additionally, computations that are necessary for allreduce can be overlapped with computations of the application program on a host machine, because modules on network devices can compute independently of host machines.

4.3.1 Overheads of MDO

To evaluate the overheads of MDO, three programs are executed that synchronize all processes on multiple host machines. The first one uses MMP's built-in function to synchronize processes. The second one uses the module

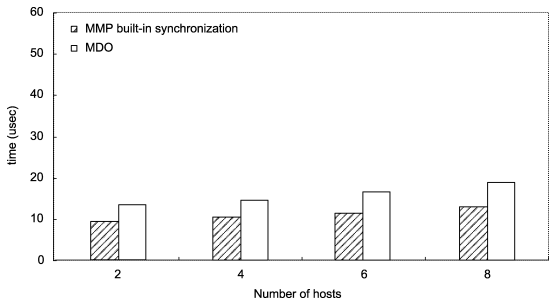

Fig. 6 Time of synchronization using MMP built-in function and MDO.

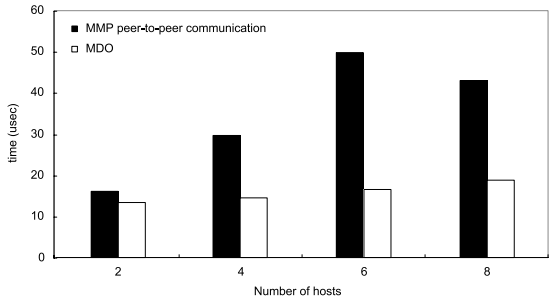

Fig. 7 Time of synchronization using MDO and MMP peer-to-peer function.

for synchronization and MDO.

The results are shown in **Fig. 6**. Although the MMP's built-in function used in the first program performs synchronization in the same manner as in MDO, it is implemented in a monolithic firmware and the library of MMP. On the other hand, synchronization requests in the second program are processed by the dynamically loaded module of MDO. Therefore, the difference between the results of the first and the second programs show the overhead of calling module in MDO. The results show that the overhead is about 30% of the whole synchronization time.

4.3.2 Performance of Synchronization

To see the advantages of MDO, we compare the performance of MDO with the performance using peer-to-peer communication.

Figure 7 shows the results. The performance of MDO is same as in the previous experiment. Another one shows the performance of experimental program using MMP. This program uses the recursive doubling algorithm based on peer-to-peer communications used in MPICH [17]. The number of communication steps required in this algorithm is optimized for the case where the number of hosts is a power of two. The number of communication

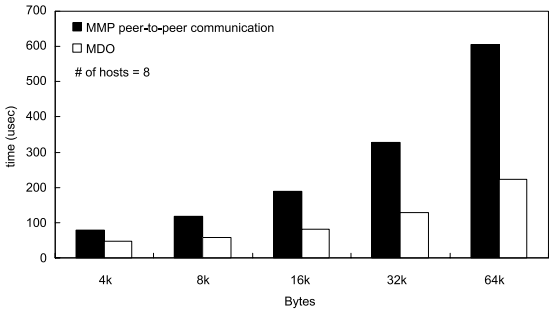

Fig. 8 Time of broadcast using MDO and MMP peer-to-peer function with varying data size.

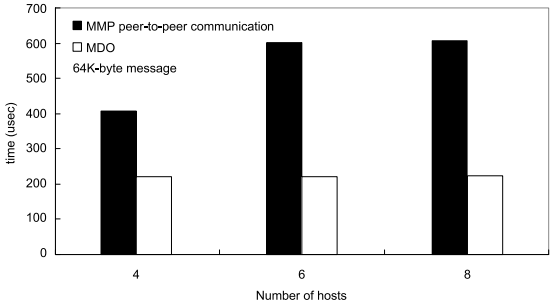

Fig. 9 Time of broadcast using MDO and MMP peer-to-peer function with varying the number of hosts.

steps in these experiments is 2, 4, and 3, when the number of hosts is 4, 6, and 8, respectively. Therefore, the time of MMP with 6 hosts is the longest.

On the other hand, the time of MDO are almost constant because communication steps are constant independently of the number of host processors. The results show that the synchronization program using MDO is three times faster than the one using peer-to-peer communication at the maximum.

4.3.3 Performance of Broadcast

We measured the performance of broadcast by using MDO. We measured the time to complete broadcasting with 8 host machines with varying data size and the time to broadcast 64 K-byte message with varying the number of host machines. We also performed the same communication with MMP's peer-to-peer communication and compared the result with the one of MDO.

The results are shown in **Figs. 8** and **9**. All results show that the broadcast using MDO is always faster than the one using peer-to-peer communication between hosts in all cases. Figure 8 shows the time of MDO is only 38% of the one of peer-to-peer at minimum. From Fig. 9, the time of MDO is almost constant even if the number of host increases. On the other, the time of MMP increases as the number of hosts increases. Especially additional communication time is required when the number of hosts is not a power of two as is the case with synchronization. And the time of MDO is up to 63% smaller than the one of peer-to-peer. From these results, we confirmed that the time for broadcast can be reduced effectively by using MDO.

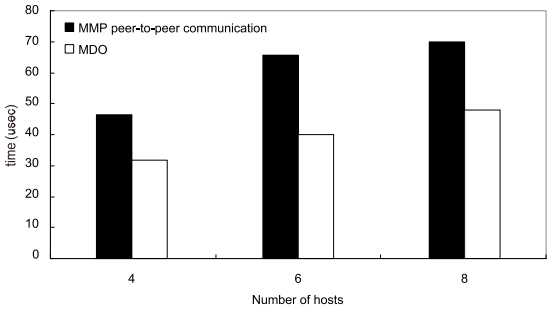

Fig. 10 Time of allreduce using MDO and MMP peer-to-peer function.

4.3.4 Performance of Allreduce

We measured the performance of allreduce with varying the number of hosts. In this experiment, we developed the module which calculates the product of values received from hosts and returns the result to all hosts. We also developed an equivalent function by using MMP's peer-to-peer communication as is the case in the experiment of broadcast, and we compared the result of MDO with the result of peer-to-peer communication.

Figure 10 shows the results of this experiment. As this figure shows, MDO was able to reduce the time for communication up to 39%.

4.3.5 Comparison of Traffic over PCI Bus

We compared the amounts of data transferred via host machines' PCI bus when they perform broadcast and allreduce with MDO and MMP peer-to-peer communication. In **Fig. 11**, we can see the traffic over a PCI bus when four and eight hosts perform broadcast (Fig. 11 (a)) and allreduce (Fig. 11 (b)). The traffic consists of data and control commands for the network interface. All the traffic on the PCI bus of all the hosts are accumulated and shown in Fig. 11.

In Fig. 11 (a), the initiator denotes the node that invoked the broadcast. Broadcast based on the peer-to-peer communication requires intermediate hosts, which re-send the message soon

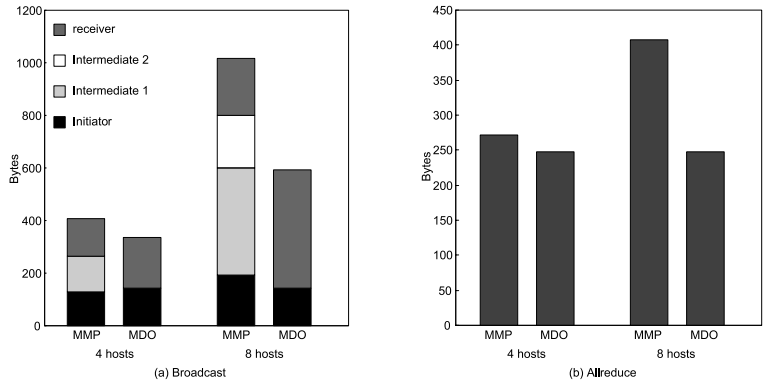

Fig. 11 Comparison of traffic over PCI.

after receiving it, for relaying. This relaying yields traffic on the PCI bus. The intermediate 1 and 2 in Fig. 11 (a) indicate the first and the second intermediate hosts that relay the message. These show the traffic on the PCI bus in receiving and re-sending the message. In broadcast with four hosts, for example, the initiator issues the message twice, intermediate 1 receives and re-sends the message, and two hosts receive; i.e., the one from the initiator and the other from the intermediate 1. In broadcast with eight hosts, three hosts work as intermediate 1 and one host works as intermediate 2. On the other hand, broadcast by MDO requires no intermediate hosts because the switch box generates and sends the copies of the message for broadcast. As a result, traffic on the PCI bus is reduced by 18% and 42% of peer-to-peer communication in the case of four and eight hosts, respectively.

Meanwhile, all hosts have to send and receive data several times in allreduce by peer-to-peer communication. Figure 11 (b) shows the accumulated PCI traffic of all hosts in sending and receiving data. The breakdown of the traffic is not shown in this figure, since in allreduce, each host behaves as an initiator, an intermediate node (only in MMP), and a receiver. The number of send and receive operations is in the order of $\log n$, where n is the number of hosts. In contrast, allreduce by MDO requires only one send and one receive operations at each host as described in the Section 4.3. Figure 11 (b) shows that the reduction ratios of the traffic on PCI are 9% and 39% for four hosts and eight hosts, respectively. It has been confirmed that MDO reduces more traffic on the PCI bus as the number of hosts involved in allreduce increases.

From the results shown in Fig. 11, MDO can reduce the traffic on the PCI bus for both broadcast and allreduce operations. The reduction of the traffic improves the efficiency of the PCI bus and contributes to reducing the execution time of collective communication.

5. Related Work

Several researches or products that are capable of offloading communication protocol to network devices have been proposed or released so far; e.g., TOE [8)~10)], RNIC [18)], or EMP [5)]. In addition, we can also find several researches [19)~21)] that try to increase performance by offloading the protocol processing or a part of the communication library to intelligent network devices such as Myrinet or QsNet. MDO aims to offload not only communication processing, but also larger units of software such as a whole communication library or a part of an application program.

The processors of the other intelligent networks (e.g., LANai on Myrinet or ELAN4 on QsNetII) have no floating-point unit. MDO and the general purpose processor on Maestro2 network provide an environment of higher potential for executing various kinds of applications.

We can find researches [22),23)] that try to improve communication performance by employing one of the SMP nodes for protocol processing. However, it is impossible for these techniques to reduce bus traffic as shown in this paper because the processor has to control their network hardware via a bus.

The switch of QsNetII has a dedicated hardware for broadcast to increase the performance in collective communication. The switch of Maestro2 includes a general purpose processor tightly coupled with routing hardware. This can make the switch to be more flexible in en-

hancing functionality, and MDO provides the hands-on method to the programmers for offloading software module to the switch.

Meanwhile, in the system that allows the users to offload user-defined software, protection can be an important issue, since the offloaded software may affect the stability of the system. Fiuzynski, et al. [19] addresses this issue by forcing programmers to use a type-safe language in developing software to be offloaded. The offloading mechanism proposed by Wagner, et al. [20] introduces a virtual machine, on which offloaded software is executed, to guarantee the stability. Unfortunately, the current MDO does not have any protection mechanism. However, we are developing a new firmware that has the capability of page-based protection by using MMU of embedded PowerPC.

6. Conclusion

In this paper, we presented a software environment that allows an application program to offload user-defined modules to the Maestro2 cluster network, named Maestro dynamic offloading mechanism (MDO). MDO includes 1) user-defined modules, 2) the MDO library to offload user-defined modules to the network interface and/or the switch box dynamically, and 3) the firmware to invoke the loaded modules.

To evaluate the performance impacts of MDO, we have developed three offload modules using MDO: synchronization, broadcast, and allreduce. The evaluation results showed that MDO is effective in reducing the time of those collective communications, even though MDO requires extra time for offloading and invoking modules. Additionally, we measured the traffic over the PCI bus when the hosts perform broadcast and allreduce with and without MDO. From the results, we have confirmed that MDO can reduce the traffic on the PCI bus by approximately 40% compared to peer-to-peer communication.

For the future work, we will evaluate the performance of MDO by offloading a part of an application. We are also planning to develop a distributed shared memory system using the MDO mechanism. Furthermore, we are currently developing the next generation of the Maestro2 cluster network which has more powerful processor and a higher-capacity memory on board. We will evaluate the performance impacts of offloading a user-defined module by using this next-generation network.

Acknowledgments This research was partially supported by Japan Society for the Promotion of Science, a Grant-in-Aid for Scientific Research(B), No.16300012.

References

1) Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W.-K.: Myrinet — A Gigabit-per-Second Local-Area Network, *IEEE Micro*, Vol.15, No.1, pp.29–35 (1995).
2) Beecroft, J., Addison, D., Petrini, F. and McLaren, M.: QsNetII: An Interconnect for Supercomputing Applications, Technical report, Quadrics Ltd. (2003).
3) Myricom, Inc.: *The GM Message Passing System* (1999).
4) Takahashi, T., Sumimoto, S., Hori, A., Harada, H. and Ishikawa, Y.: PM2: High Performance Communication Middleware for Heterogeneous Network Environment, *Proc. IEEE/ACM SC2000 Conference*, pp.52–53 (2000).
5) Shivam, P., Wyckoff, P. and Panda, D.: EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing, *Proc. ACM/IEEE SC 2001 Conference* (*SC'01*), pp.49–57 (2001).
6) Pfister, G.F.: An Introduction to the InfiniBand Architecture, *High Performance Mass Storage and Parallel I/O*, (Hai, J., Toni, C. and Buyya, R. (Eds.), chapter 42, pp.617–632, John Wiley & Sons Inc (2001).
7) Bierbaum, N.: MPI and Embedded TCP/IP Gigabit Ethernet Cluster Computing, *27th Annual IEEE Conference on Local Computer Networks* (*LCN'02*), pp.733–734 (2002).
8) Mogul, J.C.: TCP offload is a dumb idea whose time has come, *Proc. 9th Workshop on Hot Topics in Operating Systems* (*HotOS IX*), pp.25–30 (2003).
9) Cline, L., Foong, A., Huggahalli, R., Illikkal, I., Iyer, R., Makineni, S., Minturn, D., Newell, D. and Regnier, G.: TCP onloading for data center servers, *Computer*, Vol.37, pp.48–58 (2004).
10) Feng, W., Balaji, P., Baron, C., Bhuyan, L.N. and Panda, D.K.: Performance Characterization of a 10-Gigabit Ethernet TOE, *Proc. 13th Symposium on High Performance Interconnects* (*HOTI' 05*), pp.58–63 (2005).
11) Brightwell, R. and Underwood, K.D.: An Analysis of NIC Resource Usage for Offloading MPI, *Proc. 18th International Parallel and Distributed Processing Symposium* (*IPDPS'04*) (2004).
12) Aoki, K., Yamagiwa, S., Ferreira, K., Campos, L.M., Ono, M., Wada, K. and Sousa, L.:

Maestro2: High Speed Network Technology for High Performance Computing, *Proc. 2004 IEEE International Conference on Communication* (*ICC2004*), No.HS01-8 (2004).
13) IEEE Standard Department: *IEEE Standard for Low-Voltage Differential Signals* (*LVDS*) *for Scalable Coherent Interface* (*SCI*) (1996).
14) Freescale Semiconductor, Inc.: *MPC603e RISC Microprocessor User's Manual* (2002).
15) Xilinx Inc.: *Virtex-II Platform FPGA Data Sheet* (2002). http://www.xilinx.com
16) Aoki, K., Yamagiwa, S., Wada, K. and Ono, M.: Development and Evaluation of Message Passing Library for Maestro2 Cluster Network, *IEICE TRANSACTIONS on Information and Systems* (*Japanese edition*), Vol.J89-D, No.5, pp.919–931 (2006).
17) Gropp, W., Lusk, E., Doss, N. and Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard, *Parallel Computing*, Vol.22, No.6, pp.789–828 (1996).
18) Hilland, J., Culley, P., Pinkerton, J. and Recio, R.: *RDMA Protocol Verbs Specification* (*Version 1.0*), RDMA Consortium (2003).
19) Fiuczynski, M.E., Martin, R.P., Owa, T. and Bershad, B.N.: SPINE: A Safe Programmable and Integrated Network Environment, *8th ACM SIGOPS European workshop on Support for composing distributed applications*, pp.7–12 (1998).
20) Wagner, A., Jin, H.-W., Panda, D.K. and Riesen, R.: NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters, *CLUSTER 2004*, pp.205–214 (2004).
21) Petrini, F., Coll, S., Frachtenberg, E. and Hoisie, A.: Hardware- and Software-Based Collective Communication on the Quadrics Network, *Proc. IEEE International Symposium on Network Computing and Applications* (*NCA'01*), pp.24–36 (2002).
22) Falsafi, B. and Wood, D.A.: Scheduling communication on an SMP node parallel machine, *Proc. 3rd International Symposium on High-Performance Computer Architecture*, pp.128–138 (1997).
23) Almasi, G., Archer, C., Castanos, J.G., Erway, C.C., Heidelberger, P., Martorell, X., Moreira, J.E., Pinnow, K., Ratterman, J., Smeds, N., Steinmacher-Burow, B., Gropp, W. and Toonen, B.: Implementing MPI on the BlueGene/L Supercomputer, *Euro-Par 2004 Parallel Processing*, Springer Berlin/Heidelberg, pp.833–845 (2004).

(Received September 5, 2006)
(Accepted July 3, 2007)
(Released October 10, 2007)

Keiichi Aoki received his M.E. in 2004 and his Dr. Eng. in 2007, both from the University of Tsukuba, Japan. His main research interests are high performance parallel and distributed computing, especially high performance network hardware and communication software for cluster computing. He is a member of IPSJ.

Koichi Wada received his M.E. degree in computer science in 1981, his Ph.D. degree in computer science in 1984, both from the Kobe University. He has been a Visiting Professor at the University of Victoria, Canada, in 2000. He is currently a Professor in the Department of Computer Science at the University of Tsukuba in Japan. His research interests include parallel and distributed computing, high-performance network architecture, parallel programming environment, parallel simulation, and high-performance media processing. He is a member of IPSJ, IEICE, IEEE, and ACM.

Hiroki Maruoka received his B.E. degree in computer science from University of Tsukuba in 2006. He is currently a graduate student of master's program at the same university. His research interests include high performance computing, especially general purpose GPU.

Masaaki Ono received his B.E. in communication engineering in 1981 from Shibaura Institute of Technology. He is a Technical specialist at the Academic Service Office for Systems and Information Engineering, University of Tsukuba in Japan. His research interests include digital hardware systems and FPGA.