Directed Point: An Efficient Communication Subsystem for Cluster Computing

Chun-Ming Lee, Anthony Tam, Cho-Li Wang
The University of Hong Kong
{cmlee+clwang+atctam}@cs.hku.hk

Abstract

In this paper, we present a new communication subsystem, *Directed Point* (DP) for parallel computing in a low-cost cluster of PCs. The DP model emphasizes high abstraction level of interprocess communication in a cluster. It provides simple application programming interface with syntax and semantics similar to UNIX I/O function call, to shorten the learning period. The DP achieves low latency and high bandwidth based on (1) a efficient communication protocol *Directed Messaging*, that fully utilizes the data portion of packet frame; (2) efficient buffer management based on a new data structure *Token Buffer Pool* that reduces the memory copy overhead and exploits the cache locality; and (3) light-weight messaging calls, based on INTEL x86 *call gate*, that minimizes the cross-domain overhead. The implementation of DP on a Pentium (133 MHz) cluster running Linux 2.0 achieves 46.6 microseconds round-trip latency and 12.2 MB/s bandwidth under 100 Mbit/s Fast Ethernet network. The DP is a loadable kernel module. It can be easily installed or uninstalled.

1. Introduction

Commodity workstation/PC clusters represents a cost-effective platform for scalable parallel computation. The message passing mechanism is usually employed for interprocess communication in a cluster for data sharing and process synchronization. To let users can easily develop their parallel program, the design of communication subsystem should not only consider high performance but also emphasize on programmability.

The most widely used network programming interfaces are Berkeley Socket and UNIX System V TLI (Transport Layer Interface) [12]. However, both are designed based on client/server communication abstraction. The client/server model is not suitable for cluster computing since it a peer-to-peer relationship between the processes. In addition, they cannot deliver underlying network communication performance due to the large software overhead incurred in various protocol stacks. In recent years, various communication subsystems, such as Active Message [6], Fast Messages[10], BIP[11], U-Net[1], GAMMA[3] and etc., have been implemented to provide high-performance communication. While they achieves better performance in data communication, legacy network applications must be re-coded since the

semantics and syntax of the new application programming interface (API) is different from classic UNIX I/O model like Socket and TLI. For example, the file descriptors in Socket is used to abstract the communicating process at the remote side and a sequence of read and write operations are performed to deliver the data. The API of those proposed mechanisms is lack of programability, thus not attractive for users to write parallel applications.

In this paper, we propose a new communication subsystem called *Directed Point* (DP) for commodity clusters system. To provide higher level of abstraction, we view the interprocess communication in a cluster as a *directed graph*. In the directed graph, each *vertice* represents a communicating process. Each process may create multiple *endpoints* for the identification of communication channels. A *directed edge* connecting two endpoints at different processes represent a uni-directional communication. The DP model can be used to describe any interprocess communication pattern in the cluster. The programmer or the parallel programming tool can easily translate the *directed graph* to the SPMD code. DP also provides a small set of new API which conform the semantics and syntax of UNIX I/O calls. This further makes the translation very easy.

In contrast to the advocacy of user-level communication mechanism, DP is implemented in kernel level. This design makes DP a highly protected communication software. To achieve low latency and high bandwidth, we propose a simple communication protocol *Directed Message* (DM), that fully utilizes data field of packet frame. The DP can send message directly from user space to network interface without copying to intermediate buffer in kernel. On the receiver side, we design an efficient buffer management mechanism based on a new data structure *Token Buffer Pool* that shortens the memory copy time and reduces the multiplexing and signaling overhead. The interrupt handler can efficiently multiplex the incoming packets directly to the user space without delay. In DP, a set of light-weight messaging procedures are implemented based on INTEL x86 *Call Gate*. The *call gate* command provides a leeway to enter kernel without causing extra overhead in rescheduling or additional context switch delay.

Based on the above techniques, the implementation of DP on a Pentium (133 MHz) PC cluster, running Linux 2.0 operating system, achieves 46.6 microseconds round-trip latency and 12.2 MB/s bandwidth under 100 Mbit/s Fast Ethernet network, which exploits the maximum power of the underlying network. We have also implemented the MPI point-to-point communication by modifying the MPICH package using DP. We achieve an 8.7 MB/s point-to-point communication bandwidth on the same platform.

The rest of the paper is organized as follows. In the next section, we present *the Directed Point* model. Section 3 shows the system architecture of DP. In section 4, we discuss the techniques used in DP to achieve high performance. Section 5 shows the API provided by DP. Performance evaluation of DP and comparison with other research work are conducted in

section 6. Section 7 discusses the related work. Finally, we present our conclusion in section 8.

2. The Directed Point Model

Our communication subsystem is designed based on a new communication endpoint abstraction, called *Directed Point* (DP). The DP abstraction provides programmers with a virtual network topology among a group of communicating processes. Using DP model, all interprocess communication patterns can be described by a *directed graph*, with the vertex representing the communicating processes and the *directed edge* as a uni-directional communication channel between a pair of processes. In Figure 1, we show a typical interprocess communication pattern based on the DP abstraction. In DP model, each node in the cluster is assigned a logical identity called *Node ID* (NID). Multiple processes can run on each node. Each endpoints of the directed edge is labeled with a unique *Directed Point ID* (DPID). Thus using an association of 3 tuples: { *local DPID, peer Node ID, peer DPID*} we can uniquely identify a communication channel in the DP model.

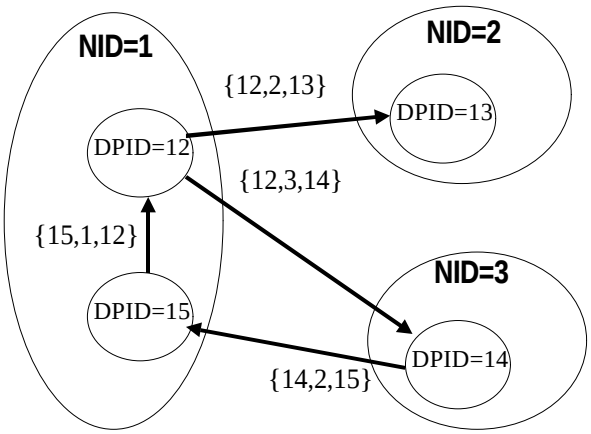

Figure 1. An abstract view of the DP communication model.

DP model provides flexible abstraction for depicting the virtual topology for data communication in a cluster. This feature can facilitate the design of higher level parallel languages or runtime support for cluster computing.

3. DP System Architecture

In this section, we describe the DP architecture. Figure 2 shows the system overview of our DP design.

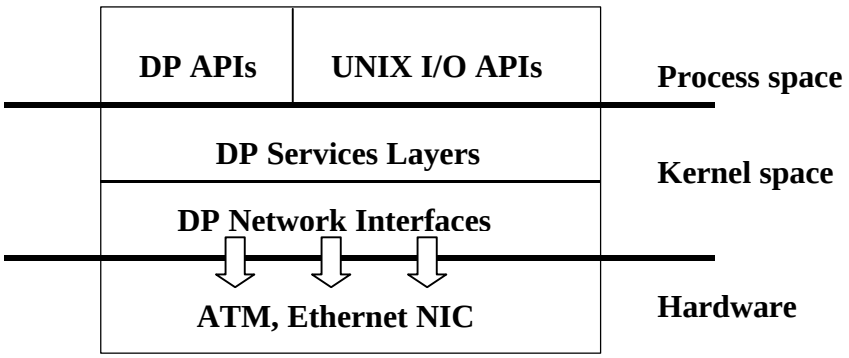

Figure 2. A design overview of DP communication subsystem.

3.1 DP Programming Interfaces

DP APIs consists of one new system call, user level function calls, lightweight message calls, and traditional UNIX I/O calls. Details of the DP API and the functions of each command will be discussed in Section 5.

All DP I/O calls follow the syntax and semantics of the UNIX I/O calls. Similar to the the *file descriptor* used in UNIX I/O calls to identify the target device or a file, we also use the it to abstract the communication channel. Once the connection is established, a sequence of *read* and *write* operations are performed to receive and send the data. The DP API provides handy and friendly interface for network programming. This unloads the burden of learning new API and makes it easier to be used, since the UNIX I/O calls have been widely used for developing code in the past.

3.2 DP Services Layer

DP Services Layer is the core of the DP subsystem. It is built by different components to provide services for passing message from user space to network hardware and deliver incoming packets to the buffer of the receiving processes. These components are:

1. DP Communication Acquirement Table (DP CAT): At user process level, DP endpoint is represented by the DPID. In kernel it is an entry of CAT. Each entry keeps the information about which process acquired the communication resource.
2. Message Dispatch Routine (MDR): MDR is responsible to distribute or multiplex the incoming message to the destination process. It is called by the interrupt handler.
3. Network Address Resolution Table (NART): Each node's network address is maintained in this table. Each node will keep an identical copy of it. It is accessed by message transmission operation to translate a NID to network address in order to build the packet frame.

4. DP I/O Operation Routines: These routines handle the actual send and receive requests from user processes.

3.3 DP Network Interfaces

This layer provides an interface for DP Services Layer to interact with the network hardware. The interface is primarily to signal the hardware to receive message or inject it to the network. It consists of the data structures required by a particular network controller. In the current implementation, we use Digital 21140 fast Ethernet controller. It requires one circular message transmission (TX) descriptor chain and one circular reception (RX) descriptor chain to be setup. Each descriptor contains a pointer to physical memory address of the buffer space in host memory that storing incoming and outgoing messages.

4. Techniques to Achieve High Performance

To achieve efficient communication, we employ various techniques to reduce software overhead in handling the messages. In this section, we describe each of them.

4.1 Directed Message

The transmission unit of DP is *Directed Message* (DM), which is variable-sized packet with a header and a data portion called *container*. The header is constructed at DP Services Layer. It consists of three fields: *target NID, target DPID*, and the length of the container.

On the Fast Ethernet implementation, the DM header is put in the source address portion of the IEEE-802.3 frame. Thus, the container has maximum size of 1500 bytes. Comparing with other implementations, U-Net has 4 bytes header put in data port, GAMMA has 20 bytes, while DP consumes no data space.

4.2.Efficient Buffer Management

Buffer management affects the communication performance[5]. In DP, we adopt an approach which avoids intermediate buffering of outgoing message. One the receiving side, we maintain a dedicated buffer, called *Token Buffer Pool* (TBP) for receive buffer. *Token Buffer Pool* is a fixed size physical memory area dedicated to a single DP endpoint. TBP is shared by kernel and the receiving process through the page remapping. Thus, the interrupt handler can directly copy incoming messages to the dedicated buffer space without delay. To provide protection in a multitasking environment, the TBP is defined as *read-only* memory space. In Figure 3, we show the *chain* structure of TBPs associated with a process.

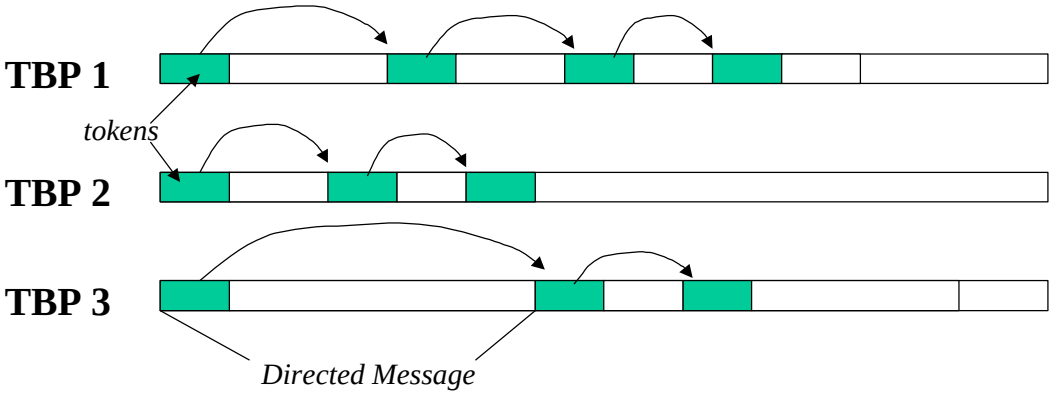

Figure 3. The data structure of the Token Buffer Pool (TBP) associated with a typical process

The unit storage in TBP is *Token buffer*. It is a variable length storage unit for storing the incoming DP message. Each *token buffer* has a control header, called *token*. It is a control structure containing the length of message and the pointer to next token buffer. Token buffers are chained as a singular FIFO queue in the TBP. New message is always appended to the tail. Old message is removed from the head by the receiving process.

4.3 Lightweight Messaging Calls

In contract to advocacy of *user-space* high-speed communication, all messaging procedures in Directed Point are implemented in the kernel level. DP API consists of a small set of new API to provide well-protected, low latency and high bandwidth communication for data communication. The kernel level approach provides maximum protection to the network resources.

In DP, *light weight messaging* call triggers the kernel level transmission routine by using Intel x86 c*all gate*. The *call gate* does not cause CPU exception unlike software interrupt that is used to implement, for example, the Linux system calls. The *call gate* command provides a leeway to enter kernel without causing extra overhead. In general, traditional system call will incur process rescheduling and necessary context switching after returns from a system call. Moreover, some *bottom-half* operations in interrupt handling will be performed after returning from system call, which leads to unexpected delay in data communication.

Using *call gate*, the *wrapper* routine is very short which only consists of a x86 *CALL* instruction and a few instructions to pass parameters. Thus, it is desired to be put inline at the user source code so as to reduce overhead in traditional function calls.

4.4. Sending and Receiving

Figure 4 shows a simplified diagram for illustrating the interaction between the user processes and the network component. The network component is the Digital 21140 controller.

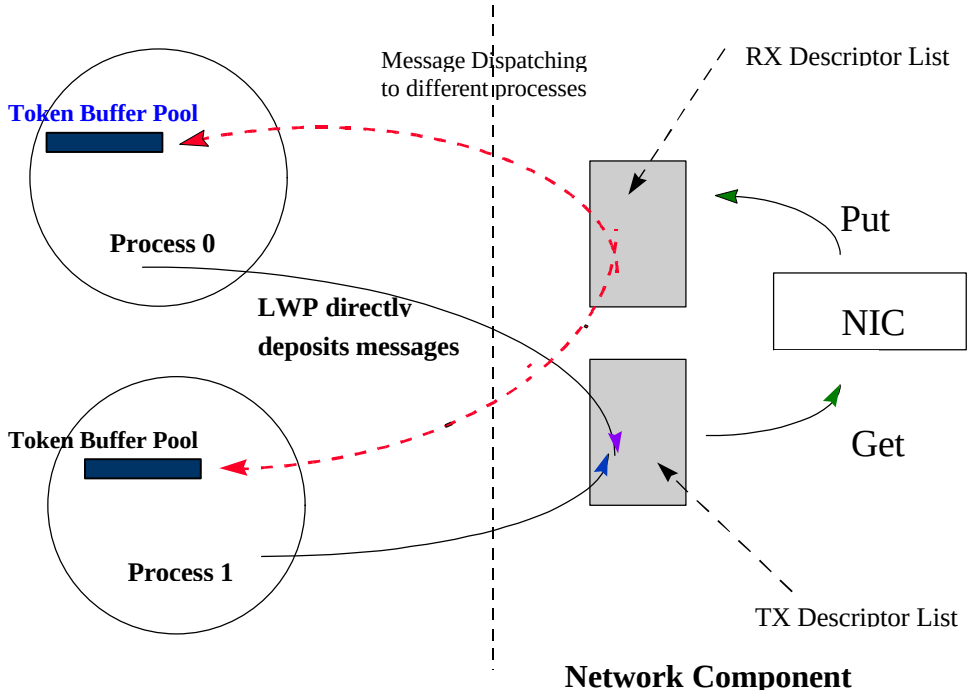

Figure 4. The Interaction between the communicating processes and the network component.

In this example, we show two communicating processes at the user space. When a process wants to transmit the message, it simply issues either a traditional *write* system call or a DP lightweight messaging call dp_write( ). The process then switches from user space to kernel space. The corresponding DP I/O operation for transmission is triggered to parse the NART to find the network address based on the given NID for making the network packet frame. The network packet frame is directly deposited to the buffer pointed by the TX descriptor without intermediate buffering. Afterward, it signals the network hardware to indicate that there is message to be injected on the network. At this stage, it is on the context of calling user process or thread which is consuming its host CPU time. After the messages stored in TX descriptor, the network component takes the message and then injects it to the network without CPU interference.

When a message arrives, the interrupt signal is triggered and the interrupt handler is activated. The interrupt handler calls MDR to look at the *Directed Message* header to identify the destination process. If it finds the process, the process's TBP is used to save the incoming message. In our implementation, the TBP is pre-allocated and locked in the physical memory. Thus, it is guarantee that the destination process will obtain the message without any delay.

5. The DP API

DP APIs consists of one new system call, user level function calls, lightweight

message calls, and traditional UNIX I/O calls. Table 1 shows all the DP function calls in our current implementation.

Table 1: The Directed Point Application Programming Interface

NEW SYSTEM CALL	
dp_open (int dpid)	create a DP endpoint
USER LEVEL FUNCTION CALLS	
int dp_read (int fd, char **address)	read an arrival DP message
int dp_mmap (int fd, dpmmap_t *tbp)	associate a TBP with an DP endpoint
void dp_close (int fd)	close the connection
LIGHTWEIGHT MESSAGING CALLS	
int dp_write (int fd, void *msg, int len)	send a DP message
int dp_fsync (int fd, int n)	flush *n* DP messages in TBP
void dp_connect (int fd, int nid, int dpid)	establish the connection with the remote endpoint.
UNIX I/O CALLS	
int read(fd, char *buf, int len)	blocking receive
int write(fd, char *buf, int len)	blocking send
int fsync(fd)	flush 1 DP message in TBP

Among the DP function calls, the dp_open( ) is the only new system call added. It is used to acquire a DP endpoint with a user specified DPID. It returns a *file descriptor* for further access on the created commuication channel. The dp_read( ) is a non-blocking receive operation. It takes the file descriptor and a pointer to pointer as the argument. It returns length of message and the virtual address of message stored in *Token Buffer Pool* (TBP). The dp_mmap maps the *Token Buffer Pool* to user virtual address. The dp_close( ) releases the acquired DP communication object that is represented by the file descriptor to the system.

There are some lightweight messaging procedure calls implemented in DP. These lightweight message procedure calls are compiled as *inline* functions. The dp_write( ) is a non-blocking transmission operation. The dp_connect( ) constructions a uni-directional communication channel with another process by given two DP endpoints. This command can be used during the runtime to reconfigure the virtual topology. The dp_fsync( ) command free the token buffer stored in TBP. Thus, the TBP can store more incoming DP messages.

Some UNIX I/O calls such as read( ), write( ), and fsync( ) can be used in DP. They follow the same syntax and semantics as before but can achieve better performance while link

with DP layer.

The DP program can be developed and executed in an SPMD model. The DP communication subsystem consists of a single API library and a header file. Compiling a DP program does not require to construct any complex *Make* file. Writing DP program is much less sophisticated and has better program structure.

6. Performance Measurement

In this section, we report the performance of DP and the comparison with other research results. Benchmark tests are designed to explore the round-trip latency and bandwidth benchmarks. We also conduct micro-benchmark tests which provide in-depth timing analysis to justify the performance of DP.

6.1 The Testing Environment

Our experimental setup consists of two AST Bravo MS5133 PCs running Linux 2.0.3x kernel. These AST machines are configured with Pentium-133 processors, 256K L2 cache, and 32MB of main memory. They are linked back-to-back by a cross-over twisted pair cable. All the benchmark measurements are obtained by using software approach, and these programs are compiled by using gcc with optimization option -O2. These tests are running in multi-user mode. To have more accurate timing measurement, we construct our timing function with Pentium TSC counter that is incremented by 1 for every CPU clock cycle. Therefore, we are able to measure the execution time in resolution of less than 16 clock cycles[1] with 133MHz CPU clock.

6.2 Bandwidth and Round-trip Latency

To measure the round-trip latency, we implemented an SPMD version of Ping-Pong program using DP API. The Ping-Pong program measures the round trip time for message size from 1 byte to 1500 bytes. Timing results are obtained by measuring the average execution time of this message exchange operation in 1000 iterations.

```
myrank == 0:
    start = rdtsc();
    for i = 1..1000
        dp_write(fd, msg, n);
        while (dp_read(fd, buf) <=0);
        dp_fsync(fd, 1);
    endfor
    stop = rdtsc();
    calculate latency;
```

```
myrank == 1:
    for i = 1..1000
        while (dp_read(fd, buf) <=0);
        dp_write(fd, buf, n);
        dp_fsync(fd, 1);
    endfor
```

[1] Due to the overhead of executing the RDTSC instruction, the resolution of a pair of rdtsc instructions is bounded by this overhead.

In Figure 5, we see that DP achieves 46.6us round trip latency for a 1-bytes message. For 1500 bytes which is the maximum data size of IEEE-802.3 frame, the round trip latency is about 386us. The latency increases linearly respect to the message size at the rate about 0.225us/byte. Further, we observe that for small message size (m < 64 bytes), the measured latencies are more or less constant.

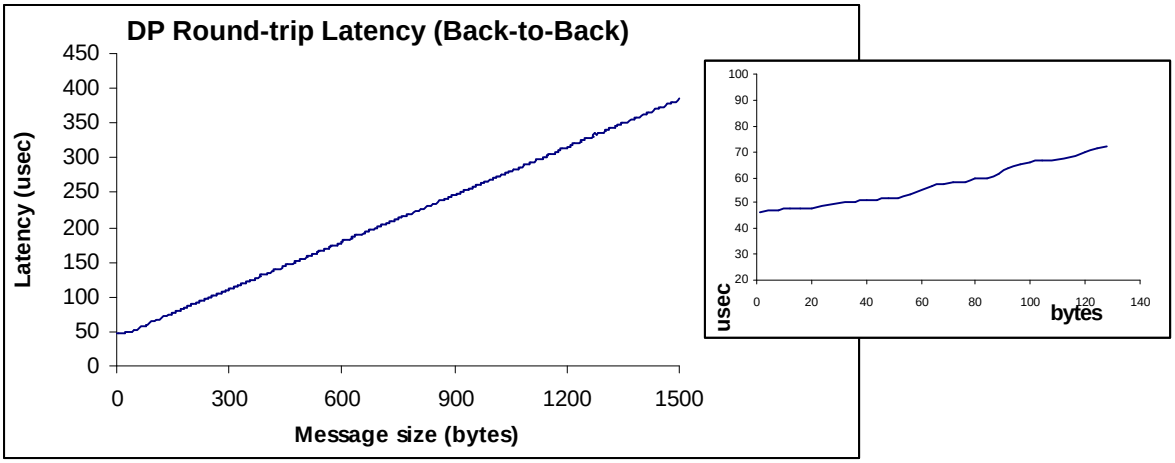

Figure 5. Round-trip latency vs transmitted message size for Fast Ethernet DP implementation on a pair of back-to-back Pentium-133 machines. The graph on the right is a magnified version for message size smaller than 128 bytes.

The bandwidth is measured by transferring 2 Megabytes of data with DP message size from 1 byte to 1500 bytes. From Figure 6, one can see that the measured bandwidth is about 12.2MB/s, which is closed to the Fast Ethernet raw bandwidth – 12.5MB/s.

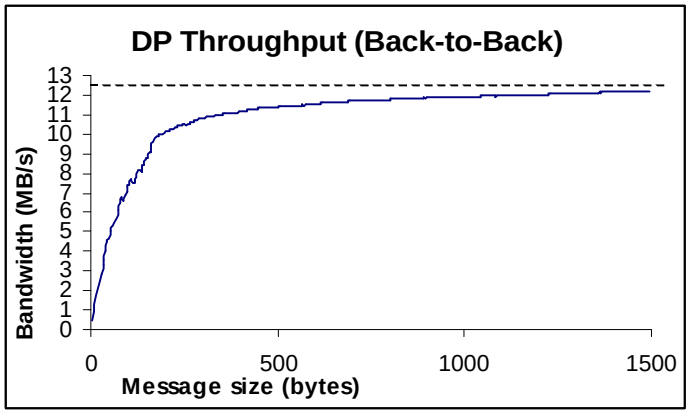

Figure 6. Throughput vs message size for Fast Ethernet DP implementation.

6.3 Micro-benchmark Analysis

To evaluate the performance of the DP communication system, a set of benchmark routines are designed. These tests are derived from a simple data transfer model of DP. This

data transfer model is based on an analytical view of the data flow through a peer-to-peer connection during a communication event. Message exchanges are going through three phases in general, from sender address space to receiver address space. First, the data on the sender side traverse through the *send* phase which is under the control of the host processor. Then they are handled by the network hardware (network interface cards - NICs, switches, cables, etc.) during the *transfer* phase of the transmission. The *receive* phase happens in the destination node when the processor of the receiving node is signaled by the NIC upon reception of incoming data.

The send phase includes all those events happen before the network adapter takes over the control. In traditional communication data path, this includes system call handling, cross-domain data copying, checksum computing, protocol stacks traversing, and, other processing overheads. Using the techniques described in Section 4, all those unnecessary overheads have been removed to shorten the gap between the user process and network hardware. This phase includes only a light-weighted system call, cross-domain copying, and of minimal processing overhead.

Typical system calls are expensive, it is usually involved many instructions before it can reach the actual routine. At the return of each system call, some events may have to be handled such as handling *bottom half* of interrupt routines (non-critical section code), system housekeeping, and context switching etc. Moreover, system calls are triggered by the wrapper routines in the C library that cause additional overheads such as stack frame construction and destruction, pushing parameter to stack due to function call transition and runtime linking of library routine. The light-weighted system call is devised to alleviate this problem. By directly jump to the handling routine and removal of all housekeeping tasks, it significantly reduces the call overhead. Some micro-benchmarking programs are designed to explore this situation. We have found that for those DP light-weight message calls, the calling overhead for the x86 *call gate* is within the range of 1.0~1.2 usec. While for traditional system call, an artificial null system call and the simple *getpid* system call cost us 1.65 and 1.68 usecs respectively when measured on the same P-133 machine. We can see the improvement makes by the use of light-weighted system call. Although it seems that the improvement is insignificant as compared to the “dummy” system call, the strength of this approach is the impact on removing those housekeeping tasks on the system entry.

Similarly, the receive phase includes all events happen in the remote node when arrived messages are ready to be handled by the remote processor, till it is being dispatched to the corresponding receiver’s process space. This is an asynchronous event in a sense that the receiving process does not involve in the whole reception. To dispatch the data to the right place, one needs to know where the data should go. This has to be managed by a privilege process which could get hold of those protected information. Thus in our DP

implementations, we are using the interrupt signals to notify the Operating System on the arrival of data. In general, this phase includes an interrupt event, destination process and buffer selection, data copying and of minimal processing overhead.

In order to estimate the overhead of using interrupt handling, a benchmark routine is designed to measure the software overhead involved, such as interrupt table lookup, interrupt handler dispatch, entry, and return, etc. We find that the cost of handling interrupts on the P-133 machine is approximately 4 usec, of which 2.5 usec is the dispatching and entry cost, and the rest is the cost of interrupt return.

The model also identifies an interesting phenomenon about data movement. As in the case of the send phase, we are involving a data copy from the cache to the main memory. This is because the data message is most likely found in the cache due to the temporal locality, and the output buffer of this phase is a memory block that has not been accessed recently. While in the case of the receive phase, we are performing a data copy from the main memory to another memory block. It is known that the incoming data is placed in a uncached memory block, and it is being copied to another memory block which is also not being accessed recently. From this observation, we can make a conjecture that the data movement overheads involved in the send and receive phases have different memory efficiencies. Thus, another set of benchmark tests is designed to evaluate this phenomenon. Figure 6c provides a diagrammatic summarization of the results find by this set of benchmark tests. We can see that the memory copy efficiency is greatly affected by the locations of the source and destination. Further, for the case of the Ethernet packet, which is of size 1 ~ 1500 byte, the difference in data copy performance could affect the overall performance of the receive phase.

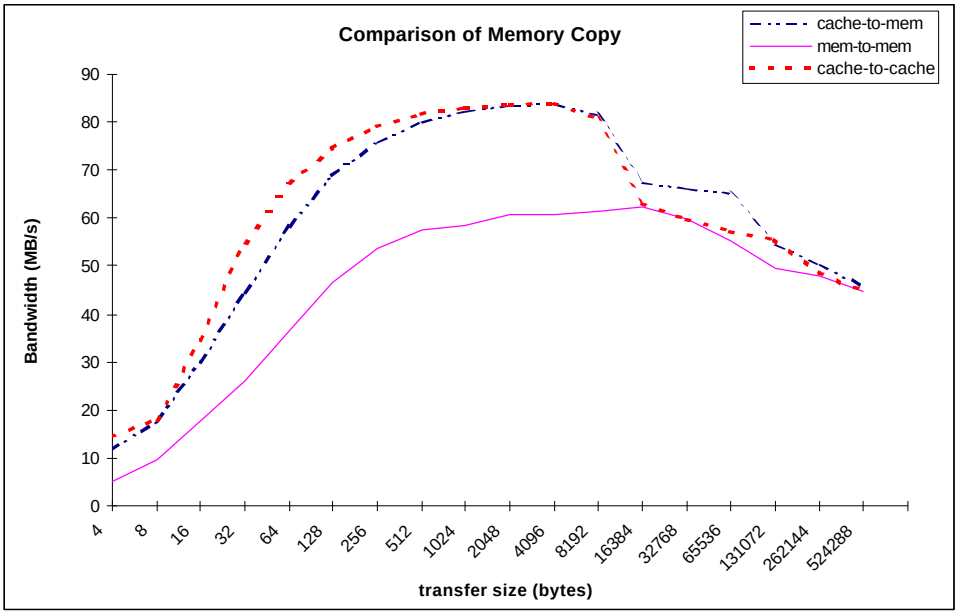

Figure 7. Memory copy bandwidth vs the transfer size on different source and destination combination - cache-to-cache, cache-to-memory, & memory-to-memory.

6.4 Comparison

In order to have more objective comparison, here we only list the performance of the approaches those implementation using same OS, similar network hardware and host machine in the Table 2. From the table, we see that DP is very competitive to other new proposed system. In fact, the bandwidth we achieve is better than others.

Approach	Platform	Round Trip Latency (us)	Bandwidth (MB/s)
Unet/FE	Linux 1.3.x, Digital 21140 Fast Ethernet, Pentium 133MHz	57	12
PARMA	Linux 2.0, 3com 3C595-TX Fast Ethernet, Pentium 100MHz	74	6.6
GAMMA	Linux 2.0, 3com 3C595-TX Fast Ethernet, Pentium 133Mhz	61.4 (cold ache) 36.8 (hot cache)	9.6 9.8
Linux TCP/IP socket	Linux 2.0, 3com 3C595-TX Fast Ethernet, Pentium 133MHz	151	5.3
DP	Linux 2.0, Digital 21140 Fast Ethernet, Pentium 133MHz	46.6	12.2

Table 2. Performance comparison with U-Net, PARMA, GAMMA, Linux BSD Socket, and DP on similar platforms

We also implemented the MPI ADI layer using the DP. We observed an 8.7 MB/s bandwidth based on the MPI point-to-point communication between two Pentium PC connected by Fast Ethernet.

7. Related Work

There are many research projects on the design of communication architecture for high-speed cluster networking. Active Message [6] and U-Net [1] have implemented new protocols and new programming interfaces to improve the network performance. Their protocols and interfaces are light-weight and provide programming abstractions that are similar to the underlying hardware. Both systems realize latency and throughput close to the physical limits of the network. However, none of them offer API compatible with existing applications.

GAMMA [3] is one of Active Message implementation on Linux platform using Fast Ethernet. The GAMMA achieved light-weight system call using *trap gate*. Modification of kernel is necessary. However, the DP is a *loadable kernel module* which makes it very easy for installation and maintenance. Similar to GAMMA, Fast Message [10] is a variant of Active Message with additional features like flow control and simple buffer management.

Also motivated by the idea of Active Message, BIP (Basic Interface for Parallelism) [11] can achieve very low latency. However, the goal of BIP is to achieve the maximum performance of the network device. The level of abstraction is not sufficient. In addition, it only allows one dedicated process to run in a node in the current implementation. This contradicts to the fact that most of cluster system should run on a time-sharing and multi-programming environment.

Other research work on the aspect of client-server based communication are Fast Socket [12], SHRIMP Stream Socket [4] and PARMA project. Fast Socket is implemented on top of Active Message. And it does not support multi-processes. SHRIMP Stream Socket is built on top of SHRIMP network interface which is a customized hardware to provide a communication mechanism called *virtual memory mapped communication*. PARMA project [8] provides a new protocol called PRP (PaRma Protocol) to Linux socket implementation. Comparing to other proposed mechanisms (including DP), PRP has higher latency and lower bandwidth.

8. Conclusion

In this paper, we introduce Directed Point architecture which is a flexible mechanism for constructing virtual network topology for cluster networking. The DP API is consistent with the syntax and semantics of classic UNIX I/O calls, which makes DP a simple interface for network programming. It facilities the development of higher level communication libraries, such as MPI and PVM, or communication protocols (e.g., TCP, UDP, IP, etc). The all-in-kernel design makes DP a highly protected communication subsystem, which can deliver low latency and high bandwidth network communication for clusters computing. We believe with proper design and implementation using low-level optimization techniques in reducing memory copy, efficient buffer management, interrupt handling, avoid the system call overhead, the kernel-level approach is capable of achieving high performance using commodity network component. Currently, we are porting the DP to an ATM platform. A Java-based programming interface for DP is under construction. Performance evaluation of DP on various commodity platforms will be reported in the final version of the paper.

.

References

[1] A. Basu, V. Buch, W. Vogels, T. von Eicken, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado, December 3-6, 1995.

[2] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, "A Virtual

Memory Mapped Network Interface for the SHRIMP Multicomputer", In Proceedings of 21th International Symposium on Computer Architecture, pp. 142-153, April 1994

[3] G. Chiola and G. Ciaccio,"GAMMA: a Low-cost Network of Workstations Based on Active Messages", Proceedings of PDP'97 (5th EUROMICRO workshop on Parallel and Distributed Processing), London, UK, January 1997.

[4] S. Damianakis, C. Dubnicki, and E. W. Felten, "Stream Sockets on SHRIMP", Technical Report TR-513-96, Princeton University, Princeton, NJ, October 1996.

[5] P. Druschel, L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", In Proceedings of the 14th Symposium on Operating Systems Principles, pp.189-202, Asheville, NC, December 1993.

[6] T. von Eicken,D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation.", Proceedings of the 19th Int'l Symp. on Computer Architecture, May 1992, Gold Coast, Australia.

[7] K. Langendoen, R. Bhoedjang, and H. Bal, "Models for Asynchronous Message Handling",IEEE Concurrency, pp. 28-38, June 1997.

[8] P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte, and P. Rossi, "An Operating System Support to Low-Overhead Communications in NOW Clusters", Proceedings of Communication and Architectural Support for Network-Based Parallel Computing CANPC97, San Antonio, Texas, February 1997.

[9] N. Nupairoj, and L. M. Ni, "Benchmarking of Multicast Communication Services", Tech. Rep. MSU-CPS-ACS-103, Department of Computer Science, Michigan State University , Apr, 1995.

[10] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," Proc. Supercomputing '95, CS Press, Los Alamitos, Calif., 1995.

[11] L. Prylli, R. Westrelin, and B. Tourancheau, "Modeling of a High Speed Network to Maximize Throughput Performance: the Experience of BIP Over Myrinet," http://www-bip.univ-lyon1.fr/bip.html, 1997.

[12] H. Steven, Rodrigues, Thomas E. Anderson, and David E. Culler, "High Performance Local-Area Communication With Fast Socket," Proceedings of Winter 1997 USENIX Symposium, January 1997.

[13] W. R. Stevens, "Unix Network Programming", Prentice Hall, 1994

[14] M. Welsh, A. Basu, and T. von Eicken, "ATM and Fast Ethernet Network Interfaces for User-level Communication", Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA), San Antonio, Texas, February 1-5, 1997.