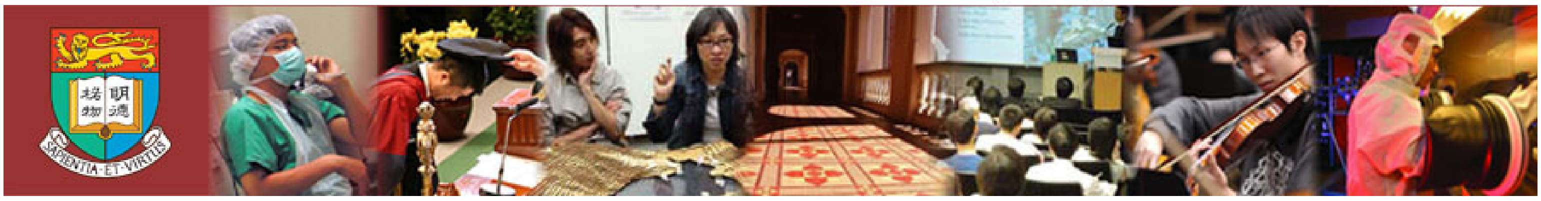

Push-Pull Messaging: A High-Performance Communication Mechanism for Commodity SMP Clusters[*]

Kwan-Po Wong and Cho-Li Wang
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam, Hong Kong
kp.wong@graduate.hku.hk, clwang@csis.hku.hk

Abstract

Push-Pull Messaging is a novel messaging mechanism for high-speed interprocess communication in a cluster of symmetric multi-processors (SMP) machines. This messaging mechanism exploits the parallelism in SMP nodes by allowing the execution of communication stages of a messaging event on different processors to achieve maximum performance. Push-Pull Messaging facilitates further improvement on communication performance by employing three optimizing techniques in our design: (1) Cross-Space Zero Buffer provides a unified buffer management mechanism to achieve a copy-less communication for the data transfer among processes within a SMP node. (2) Address Translation Overhead Masking removes the address translation overhead from the critical path in the internode communication. (3) Push-and-Acknowledge Overlapping overlaps the push and acknowledge phases to hide the acknowledge latency. Overall, Push-Pull Messaging effectively utilizes the system resources and improves the communication speed. It has been implemented to support high-speed communication for connecting quad Pentium Pro SMPs with 100Mbit/s Fast Ethernet.

1. Introduction

A cluster refers to a group of whole computers that works cooperatively as a single system to provide fast and efficient computing services. As the cost of multiprocessor machines decreases, typically those small-scale SMPs with two to four processors, building a low-cost *Cluster Of Multi-Processors* (COMP) is a cost-effective solution to achieve high computing power.

Effective and efficient clustering requires high-speed communication between nodes. However, messaging in a cluster environment is non-trivial since the sender and receiver are usually not synchronized. The asynchronous nature of message passing leads to additional overheads in buffering, queuing/de-queuing, and synchronizing communication threads. Building COMPs brings new challenges in designing a high-performance communication system.

In recent years, several research works were conducted for developing COMPs. COMPaS developed by RWCP [11], Clumps by UC Berkeley [7], and FMP by Tsinghua University [9], are the most successful SMP-type COMPs. All these small-scale COMPs used Myrinet as the connection network. Thus, most implementations can achieve low point-to-point communication latency. The performance, however, mostly bounded by the co-processor performance, which was poorer than the performance of the processors in the SMP machines.

To further improve the communication speed in COMP, we can exploit the unrevealed power of SMP processors to handle messages. In a COMP, all processors in a SMP node can process different messages in parallel. However, they may have to share some common system resources, such as NICs and messaging buffers. Efficient messaging mechanism should minimize the locking effect and reduce the synchronization overhead while multiple user and kernel processes are accessing the shared resources, and intelligently use any idle or less loaded processor in the SMP node to handle the messages.

In this paper, we discuss *Push-Pull Messaging* and its optimizing techniques to achieve low latency and high bandwidth communication between processes in the COMP environment. The idea of *Push-Pull Messaging* is similar to the classical *three-phase protocol*. In three-phase protocol, the communication pattern guarantees buffers along the communication path are not overflowed, thus reducing the amount of retransmission overhead. The protocol, however, introduced a significant amount of overheads during the handshaking phase.

To adopt the good qualities in the protocol while avoiding the penalty in the handshaking phase, we introduce *Push-Pull Messaging*. The messaging process is

[*] The research was supported by Hong Kong Research Grants Council grant HKU 7030/97E, HKU 1998/99 Large Equipment grant, and HKU CRGC grant 10200544.

started by the send party. The send party transmits a message by first directly "pushing" a portion of the message to the receive party. The receive party starts the *pull* phase after the receive operation has been issued and the pushed message has arrived. The rest of the message is sent after an acknowledgement from the receive party is received by the send party.

This communication pattern likes the three-phase protocol guarantees buffers to be properly managed. Unlike the three-phase protocol, Push-Pull Messaging can reduce the handshaking delay for the short message transfers. In addition, the pattern makes it possible to apply various optimizing techniques to remove those unexpected overheads from the critical path.

We have implemented the Push-Pull Messaging on quad Pentium Pro SMPs, connected through 100Mbit/s Fast Ethernet. We have measured the single-trip latency of 34.9 μs**,** and the peak bandwidth of 12.1 MB/s for the internode communication. The single-trip latency between processes within the same SMP node can be as low as 7.5 μs, and the achievable bandwidth is 350.9 MB/s. We also developed an *early* and *late receiver* tests for examining the run-time performance of the proposed messaging mechanism. We found using Fast Ethernet is a low-cost solution to achieve high-speed communication, other than using expensive interfaces like Myrinet, ATM, or future network interface VIA [12].

For the rest of the paper, we first discuss a generic communication model for COMP in Section 2. In Section 3, we present the basic idea of Push-Pull Messaging. In Section 4, we discuss the proposed optimizing techniques. In Section 5, the performance results are shown. Analysis is presented for both internode and intranode cases. Finally, the conclusion is given in Section 6.

2. A generic communication model for SMP

The communication between a pair of COMP nodes can be viewed as a communication pipeline with various processing stages. A generic communication model with four pipelining stages is examined below and the related design issues are discussed.

Stage 1: Transmission thread invocation

User applications initiate the transmission by issuing a send operation in user space. Then, the data transmission thread will be invoked to format outgoing packets. The thread puts the packets to the outgoing first-in-first-out (FIFO) queue in the data dump of the network interface card (NIC). In a COMP, several processors may access the NIC simultaneously. To ensure the correctness of the invocation in the multiprocessor environment, the system has to restrict that only one user or kernel thread invokes the thread at a time. Efficient synchronization between concurrent processes in the COMP node is crucial to the communication performance [5].

Stage 2: Data pumping

After the submission of packets, the NIC pumps packets to the physical network through the hardware on the NIC. The time spent in data pumping mainly depends on the hardware performance. For example, it can be affected by the performance of DMA engines in the host node and the NIC, and the network switch performance [8].

Stage 3: Reception handler invocation

The data arrives at the receive party and stores in a designated buffer in the NIC. *Interrupt* and *polling* are two main mechanisms to invoke the handler to serve the data arrival requests. For COMP nodes, there are two types of interrupt – *asymmetric* and *symmetric* interrupt. With asymmetric interrupt, requests are always delivered to one pre-assigned processor. With symmetric interrupt [4], requests can be delivered to different processors, where the selection of processors is governed by an arbitration scheme. On the other hand, polling is a light-weight approach to handle incoming packets. Polling routine watches the change of state variables and starts the handling routine if necessary. The frequency of polling determines the reliability of communication [3,6].

Stage 4: Reception processing

After invoking the reception handler, the handler processes packets immediately. Reception processing involves re-assembly of packets, copying between buffers, de-queuing buffer entries and pending requests, and synchronization between user and kernel threads. In a COMP node, there are multiple active user-level receiving threads. Without careful coordination between these communication threads and the reception handler in kernel space, high-speed communication is impossible.

3. Push-Pull Messaging

The basic idea of Push-Pull Messaging is based on the communication model discussed in Section 2. Figure 1 illustrates the communication architecture of Push-Pull Messaging.

As shown in the figure, each send or receive process has its application-allocated buffer, *source buffer* and *destination buffer* respectively, resided in the user space. Each process also shares three data structures with the kernel. The *send queue* stores the information of pending send operations. The *receive queue* stores the information of pending receive operations. The *buffer queue* and *pushed buffer* stores pending incoming packets where their destinations in memory are undetermined.

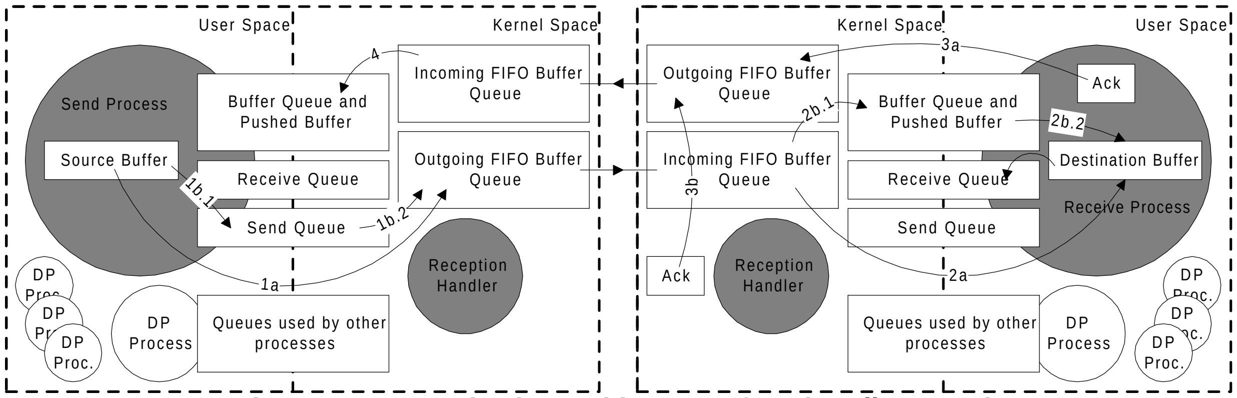

Figure 1. Communication architecture of Push-Pull Messaging

In Push-Pull Messaging, the send process first *pushes* a part of the message to the receive party as shown in arrow 1a. The *pushed message*, which contains BTP (Bytes-To-Push) bytes, is then handled by the reception handler in the receive party. The rest is registered in the send queue through arrow 1b.1. Depending on the timing of the receive operation performed by the receive process, the *pushed message* will be stored in the *pushed buffer* if the receive operation is not yet started as shown in arrow 2b.1. Otherwise, the message will be copied to the *destination buffer* as shown in arrow 2a by the registered information in the receive queue. Once the receive operation started, either the reception handler in the receive party or the receive process itself will *pull* the rest of the message from the send process.

The *pull* phase will be started by sending an *acknowledgement* (or "Ack" in the figure), which implicitly contains request information, through arrow 3a or arrow 3b. The reception handler in the send party processes the *acknowledgement*. If the request is granted, the send handler will *send* the requested part of the message in the send queue through arrow 1b.2 to the receive party. The reception handler in the receive party handles the message and directly copies the message to the destination buffer without buffering in the *pushed buffer* through arrow 2a using the registered information in the receive queue.

The important parameter BTP defines the number of bytes to be pushed by the sender at the beginning. This number is chosen based on the speed of the network and the memory system. The method to obtain this parameter is explained in Section 5.2.

Memory is a valuable resource for improving the communication performance. A pinned memory area is usually used as communication endpoint in either user or kernel spaces to improve the communication performance [1,2,11]. Although this approach could achieve low-latency communication by avoiding the delay in paging overheads, inefficient use of these pinned memory areas will limit the communication bandwidth when multiple communication channels are concurrently connected between SMP nodes. This leads to poor scalability in maintaining high-speed communication in COMP.

In Push-Pull Messaging, only a small buffer of BTP bytes is needed as the *pushed buffer*. Applications can dynamically change the size of the *pushed buffer* to adapt to the runtime environment.

4. Optimizing techniques

In this section, we discuss optimizing techniques to further improve the communication performance based on the Push-Pull Messaging mechanism.

4.1. Exploiting parallelism in COMP nodes

In a COMP node, *push* and *pull* phases can be carried out on different processors to produce maximum performance. After the *push* phase, the rest of the message will be transferred by the *pull* operation. As the *pull* phase is designed to make a direct transfer from the source buffer to the destination buffer without intermediate buffering, this phase can be handled by a lightly loaded processor. It is not necessary to be handled by the same processor as the one used in applications.

The selection of the processor depends on the reception handler invocation method. In all of our tests, we used *symmetric interrupt* mechanism in our optimized Push-Pull Messaging. This mechanism allows the pull phase to be executed on a least-loaded processor. Because of running the *pull* phase on another processor, the phase can be overlapped with the computation or communication events carrying on other processors. This overlapping can hide a portion of the communication latency in the internode test. The latency hiding mechanisms are discussed in Section 4.3 and 4.4.

In the *push* phase, we did not choose the lightly loaded processor to push data. This is because offloading the processing overhead to other processors could not exploit the temporal cache locality in the original processor. Contrarily, it may introduce a large number of cache misses. Instead of offloading, we execute the *push* phase on the processor same as the one serving the send process.

4.2. Cross-space zero buffer

Cross-Space Zero Buffer is a technique to improve the performance of data copying across different protected user and kernel spaces. In common message passing libraries, the syntax of the communication APIs is usually defined as follows.

send(*source_buffer_address, buffer_length*)
receive(*destination_buffer_address, buffer_length*)

The send operation accepts a virtual address of the source buffer and its length. Like the send operation, the receive operation requires the virtual address of the receive buffer and its length. Both buffers are allocated by applications in the process space. As process spaces are protected, direct communication cannot be carried out between two user processes. Typically, the communication is taken place through a *shared memory* facility provided by the kernel. Using shared memory approach, however, introduces an unavoidable memory copy operation. The unavoidable copy operation and implicit synchronization result in extra processing overheads, thus lengthening the communication latency and consuming more memory resources.

We attacked the problem by employing a *cross-space zero buffer* technique which realizes *one-copy* data transfer across process spaces, thus reducing the memory copy overheads. To realize the one-copy transfer across process spaces, physical addresses of source and destination buffers are needed. Although the virtual addresses of buffers are continuous, the corresponding physical addresses may be discontinued across pages. Since buffers may not reside in contiguous memory space, pairs of *physical address* and *length* need to be obtained before the actual data movement. The *physical address* points to the starting address of the multiple buffer pages. The *length* denotes the number of contiguous bytes at the corresponding address. Since this data structure only contains addresses and length values but not the actual messages, we call it *zero buffer*. By knowing the physical addresses of both buffers, data transfer from the source buffer to the destination buffer can be performed by a kernel thread. Therefore, one-copy data transfer across different process spaces could be achieved.

In Push-Pull Messaging, *zero buffers* are not only employed to improve the performance of intranode communication between user processes. The buffers are also implemented to allow direct transfer of data from the NIC designated buffer to the destination buffer in internode communication.

4.3. Address translation overhead masking

Address translation overhead masking is a technique to hide the address translation overhead in the internode communication. With the implemented *zero buffer*, the data transfer from the NIC buffer to the destination buffer on the same machine can be carried out directly by the kernel without the user process's involvement. However, Push-Pull Messaging needs to perform address translation before using *zero buffers*.

The address translation overhead grows linearly as the size of the message increases. Since the communication event requires relatively long latency time to complete than the address translation, we can schedule every network communication event in the *push* and *pull* phases before the address translation to mask the overhead.

However, not all translations can be safely scheduled. The translation of the pushed message needs to be done before initiating the first network transmission. To further hide this translation overhead, the operation of copying the pushed message to the NIC's outgoing buffer has to be performed in user space. This can be done by *direct thread invocation method*, which invokes the transmission thread in the NIC at the user level without using system calls. This method is achieved by mapping NIC control registers and buffers onto the user process space. Thus, the send process can directly trigger the NIC to start the send operation. Similar approaches can be found in DP [8], GAMMA [1] and U-Net [2].

Since all address translations can be safely scheduled, the translation overhead is removed from the critical path in communication. Figure 2 illustrates this masking technique. The address translation, which is shown as "Find out physical addresses", is delayed in the send and receive parties.

4.4. Push-and-acknowledge overlapping

Push-and-Acknowledge is an optimizing technique to hide the acknowledge latency in the internode case. Originally in Push-Pull Messaging, sending the acknowledge message is on the critical path. To further enhance the performance of Push-Pull Messaging, we overlap the push and acknowledge phases in order to hide the long acknowledge latency. This optimization is also shown in Figure 2.

The pushed BTP bytes, originally used in Push-Pull Messaging, are split into two parts. The *first-pushed message* of BTP(1) bytes, is pushed to the destination at the beginning. Transmission of the *second-pushed message* of BTP(2) bytes, is overlapped with the transmission of the acknowledge message. The latency of the acknowledge message is masked. This technique can also minimize the size of the *pushed buffer*, where only the larger values of BTP(1) and BTP(2) is used as the size of the buffer.

5. Performance results and analysis

The proposed Push-Pull Messaging was implemented and evaluated on two ALR Revolution 6X6 Intel MP1.4-

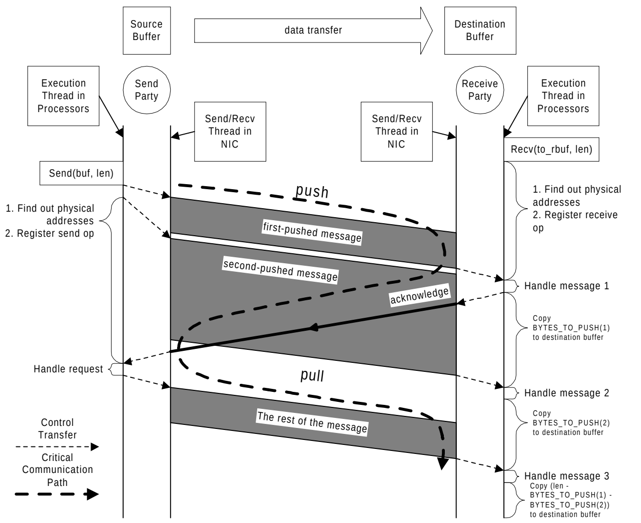

Figure 2. Overhead Masking and Push-and-Acknowledge Overlapping are used in Push-Pull Messaging.

complaint SMP computers. Each computer consisted of four Intel Pentium Pro 200 MHz processors with 256 Mbytes of main memory. Each Intel processor had 8-Kbyte L1 instruction cache and 8-Kbyte data cache. The size of the unified L2 cache is 512 Kbytes. The computers were connected by Fast Ethernet with the peak theoretical bandwidth of 100 Mbit/s. Each computer attached one D-Link Fast Ethernet 500TX card with Digital 21140 controller. Linux 2.1.90 was installed on each machine with symmetric interrupt enabled.

We evaluated the performance of intranode and internode communication. In each case, the single-trip latencies of the communication system with different values of the parameter BTP were measured. In all benchmark routines, source and destination buffers were page-aligned for steady performance. The benchmark routines used hardware time-stamp counters in the Intel processor, with resolution within 100 ns, to time the operations. Each test performed one thousand iterations. Among all timing results, the first and last 10% (in terms of execution time) were neglected. Only the middle 80% of the timings was used to calculate the average.

The round-trip latency test measured the ping-pong time of two communicating processes. The bandwidth test measured the time to send the specified number of bytes from one process to another process, plus the time for the receive process to send back a 4-bytes acknowledgement. The time measured was then subtracted by the single-trip latency time for a 4-byte message. Thus, the bandwidth was calculated as the number of bytes transferred in the test divided by the calculated time.

5.1. Intranode communication

Push-Pull Messaging with different BTP parameters was tested for intranode communication. The parameter varied from zero (Push-Zero) to the whole message length (Push-All). Push-Pull Messaging used 16 bytes as the BTP parameter. The single-trip latency is shown in Figure 3. In the intranode communication, when the size of the message was below 16 bytes, Push-Pull and Push-All Messaging performed equally well and both outperformed Push-Zero Messaging. In this case, both send and receive operations were equally “light”. The receive operation could not complete the registration of the operation before the send operation started the actual data transfer. Therefore, Push-Pull and Push-All needed to utilize the *pushed buffer*. However, copying the message twice between the buffers only costs a small amount of overhead, as the message was so small. Push-Zero Messaging tried to avoid copying twice by synchronizing the send and receive parties. However, the synchronization resulted in a larger amount of overhead.

From 10 bytes to 3000 bytes, the receive operation could register the destination buffer information before the send operation started the actual data transfer. All mechanisms could proceed without using the *pushed buffer*, including Push-All for most of the cases. They all used *zero buffers* to minimize the transfer overhead. For messages shorter than 16 bytes, Push-Pull operated like Push-Zero. For messages larger than 16 bytes, Push-Pull returned to its standard operation. This change in communication pattern allowed Push-Pull to effectively reduce the number of memory copies in the *pull* phase. Push-Zero also synchronized the send and receive parties before transferring the message. This synchronization and the change in pattern allowed both messaging mechanisms utilizing their *zero buffers*. Therefore both messaging mechanisms outperformed Push-All. Around 4000 bytes, the latency of Push-All Messaging was abruptly increased but Push-Pull and Push-Zero kept increasing steadily. The cause of this sudden performance lost was the timing of the send and receive operations. Originally, the receive operation could register the destination buffer information before the actual data transfer. However, the address translation overhead grows with the message size. As the receive operation became

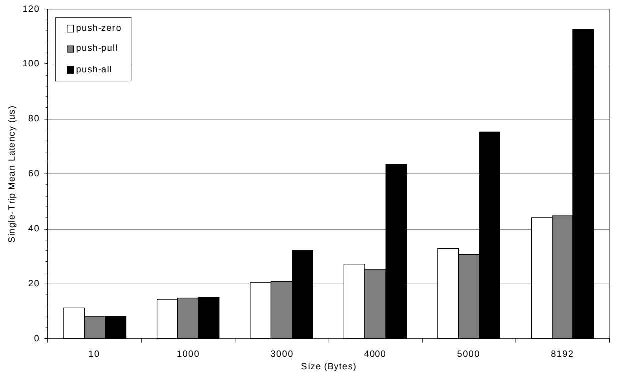

Figure 3. Intranode communication with the pushed buffer of size 12 Kbytes.

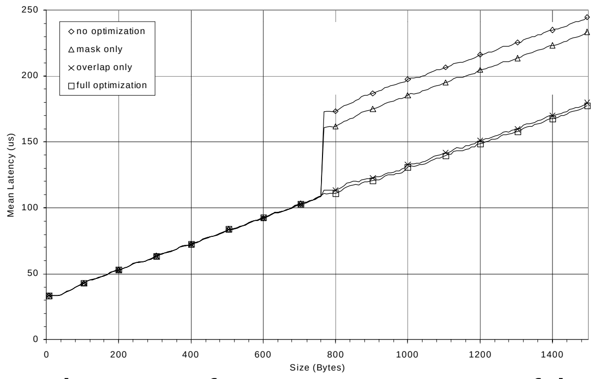

Figure 4. Performance measurement of the internode communication using three optimizing techniques.

"heavier", Push-All could not always proceed without using the *pushed buffer*. The registration could not be completed before the actual transfer in most of the times. Consequently, the transfer process utilized the *pushed buffer* and could not exploit the *zero buffer*. The average performance was further degraded around 3000 to 4000 bytes. Push-All performed poorer than Push-Pull and Push-Zero for most of the message sizes.

Zero buffer played an important role in minimizing the latency in all messaging mechanism. However, to practically exploit the mechanism, a proper communication pattern should be adopted. Since the communication pattern of Push-Pull and Push-All reinforced the execution order of the registration and data transfer phases, the performance of z*ero buffer* could be exploited effectively. The *zero buffer* mechanism not only shortened the latency of the messaging, but it also improved the bandwidth of the communication since only one memory copy is needed. The measured peak bandwidth of Push-Pull is 350.9 Mbytes/s when sending around 4000 bytes, almost 66% of the theoretical 533-Mbyte bus bandwidth. The minimum latency for sending a 10-byte message is only 7.5 μs.

5.2. Internode communication

We carried out three latency tests to evaluate the effectiveness of Push-Pull Messaging in the internode communication. *Symmetric interrupt* was chosen as the reception handler invocation method in all tests.

We used 80 bytes and 680 bytes as the value of BTP(1) and BTP(2) respectively. These parameters were obtained independently by two separate tests.

The first test measured the latency by varying the value of BTP(2) but let BTP(1) be zero. This test only exploited the Push-and-Acknowledge Overlapping technique. As the value of BTP(2) increased, the latency of a longer second-pushed message could be hidden effectively. Thus, the remaining bytes of the message to be pushed could become shorter. Since the pulled message was on the critical path in communication, the overall latency could be shortened as the value of BTP(2) increased. However, there was an upper limit on the BTP(2) value since the latency of the overlapped acknowledge phase was about the single-trip time of a short message. If the value of BTP(2) was too large, the overall latency would increase as the reception handler was unable to serve the second-pushed message and the pulled message in parallel. In this test, we obtained 680 bytes as the value of BTP(2).

In the second test, we fixed 680 bytes as the value of BTP(2) and varied the value of BTP(1). We then measured the overall latency. As the first-pushed message was on the critical path as shown in Figure 2, the latency grew with the value of BTP(1) when the BTP(1) value was larger than a threshold value. However, when the value was smaller than the threshold value, the latency would actually decrease. This reduction is caused by filling the time gap between serving the first and the second pushed message, which is illustrated as "Handle message 1" in Figure 2. As the time to handle the message was a little bit faster than the time to initiate the transmission of the second-pushed message, the receive party would have more time to process the first-pushed message. Therefore sending a longer first-pushed message would save some bandwidth, thus shortening the overall latency. In this test, we obtained 80 bytes as the value of BTP(1).

We compared the raw performance of Push-Pull Messaging with three optimized Push-Pull Messaging – Address Translation Overhead Masking (represented by [Δ]), Push-and-Request Overlapping (represented by [×]) and their combined version (represented by [□]) – as shown in Figure 4.

Before 760 bytes, all four messaging mechanisms behaved the same since the whole message was pushed to the receive party directly. After 760 bytes, the messaging mechanisms with Address Translation Overhead Masking and Push-and-Acknowledge Overlapping efficiently masked the overheads at both send and receive parties. Therefore, both techniques showed significant improvement over the non-optimized messaging mechanism. When we compared these two techniques, Push-and-Acknowledge Overlapping showed larger improvement. It is because the acknowledge latency, which is hidden by Push-and-Acknowledge Overlapping, is larger than the translation overhead saved in Address Translation Overhead Masking. In the figure, the full optimization showed the most promising solution, which integrated both orthogonal techniques.

5.3. Early and late receiver tests

In a cluster environment, the sender and receiver operate in an asynchronous manner. Extra blocking time happens when the receive party starts earlier than the send party; while overheads are always caused by the late start of the receive process. When we measured the latency of

```
ping()                                    pong()
{                                         {
    barrier();                                barrier();
    start = get_timer();                      compute y times;
    compute x times;                          pp_receive(message);
    pp_send(message);                         compute x times;
    compute y times;                          pp_send(message);
    pp_receive(message);                  }
  latency = get_timer() – start;
}
```

Figure 5. The redesigned ping-pong benchmark routine for testing early and late receivers

the internode communication, the ping-pong benchmark routine was redesigned to simulate a typical compute-then-communicate parallel program to examine the runtime performance of Push-Pull Messaging.

As shown in Figure 5, the *ping* and *pong* procedures compute before communicate. Before taking the measurement, we further synchronized both parties with a barrier operation, which was a simple ping-pong operation.

In the test, we varied both computations by inserting different number of NOP (No Operation) instructions. Two variations were tested. In the *early receiver* test (denoted by the word "early" in Figure 6 left), we forced the receive operation started before the send operation. The value of *x* and *y* were chosen to be 500,000 and 100,000 respectively.

The other one is called *late receiver* test (denoted by the word "late" in Figure 6 right). In this test, we forced the receive operation always started after the send operation. The value of *x* and *y* were chosen to be 100,000 and 300,000 in this test. In other words, we forced all messaging mechanisms utilizing the *pushed buffer*. The number of NOPs was pre-computed with the consideration of the barrier synchronization delay since the *ping* process always late about a single-trip latency time spent in waiting the implicit synchronization message from the *pong* process.

We carried out the tests for the three messaging mechanisms, namely Push-Zero, Push-Pull and Push-All, with full optimization. For the *early receiver* test, since the receive operation always finished before the send operation, the address of the destination buffer was known to the reception handler at the receive party before issuing the send operation. Therefore, the reception handlers in all three messaging mechanisms could copy the received data directly to the destination buffer using zero buffers without intermediate buffering. Thus, the size of the pushed buffer did not significantly affect the performance for all message lengths.

However, because of the difference in the communication pattern, Push-Pull and Push-All always outperformed Push-Zero. It is because the *push* phase in Push-Zero was not used to perform any useful data transfer. This phase was originally used to preserve the execution order of the registration of the pending receive operation and the pull communication. This ordering, however, was already reinforced due to the lightly loaded receiver and the heavily loaded sender. Therefore, the push phase in Push-Zero was wasting the communication bandwidth. Push-Zero was constantly slowed down.

Push-Pull outperformed Push-All in most cases in the early receiver test because the address translation overhead was effectively hidden. Push-All could not hide the overhead as the communication pattern did not allow doing so. The improvement of Push-Pull over Push-All, however, was not significant because the translation overhead was not large and the number of memory copies in both mechanisms was the same. During the *push* phase, Push-All could bypass the intermediate buffer as the receive operation was completed like Push-Pull. Therefore, the performance of Push-All was similar to the performance of Push-Pull.

For the late receiver test, as the computation on the receive party was on the critical measurement path, the computation contributed part of the latency. In this test, the transmission of the pushed messages, if any, were always pushed to the *pushed buffer*. Since the receive

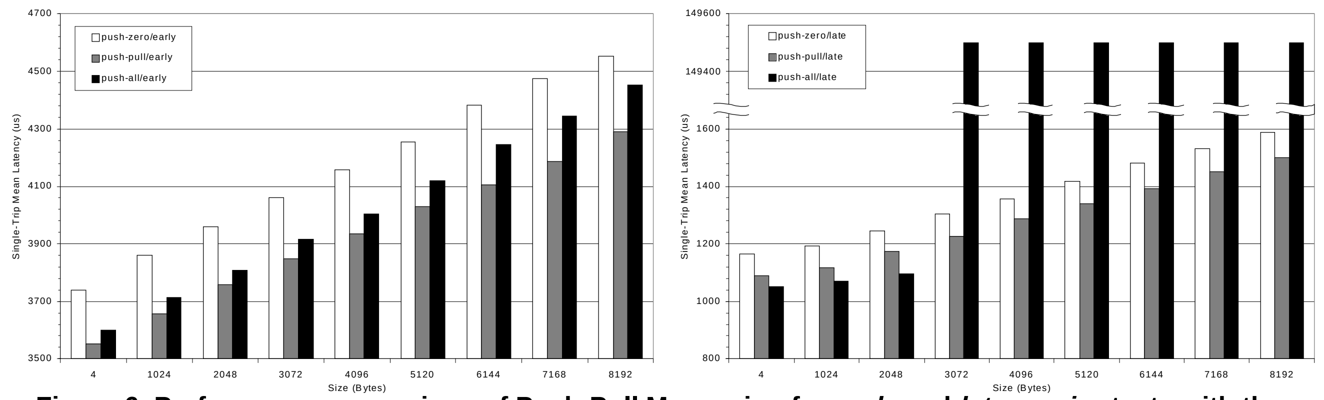

Figure 6. Performance comparison of Push-Pull Messaging for *early* and *late receive* tests with the pushed buffer 4 Kbytes.

operation was initiated so late, the reception handler in the receive party could not process the remaining part of the message without intermediate buffering in the *pushed buffer*. Therefore, the handler had to copy the message one more time before copying to the destination.

Before 3072 bytes, Push-All performed more satisfactory than Push-Pull and Push-Zero because whenever the receive operation was started, the *pushed buffer* contained the whole message. The message could then be copied directly to the destination buffer by the receive process. However in Push-Pull and Push-Zero, the receive operation always needed to initiate the transmission of an acknowledgement. Therefore, Push-Zero performed poorly for all message sizes whereas Push-Pull introduced long network latency time after around 800 bytes.

Although Push-All delivered messages faster than others did, the performance was degraded significantly after around 3000 bytes. This degradation showed that the *pushed buffer* in Push-All was overwhelmed by incoming packets. Most of the packets were lost during the communication. With the implemented go-back-n reliable protocol [10], Push-All could resume the transmission afterwards but it still could not outperform others. It took around 150 ms to transfer a 3072-byte message while Push-Zero took 1303.58 μs and Push-Pull even took only 1227.42 μs.

On the other hand, Push-Pull always outperformed Push-Zero in this late receiver test. The reason is that Push-Pull had sent BTP bytes to the receive party during the *push* phase. Therefore during the *pull* phase, shorter message was delivered.

Overall, the Push-Pull Messaging showed very steady performance in all cases as compared with Push-All and Push-Zero. Push-Pull Messaging could flexibly adapt to the cluster environment with different computation load and maximize the performance. The peak bandwidth could be as high as 12.1 Mbytes/s in fully optimized Push-Pull Messaging.

6. Conclusion

Building COMPs brings new challenges in designing a high-performance communication system. Our communication system is able to achieve very low-latency and high-bandwidth interprocess communication in the COMP environment. *Cross-Space Zero Buffer* mechanism efficiently eliminates all unnecessary memory copy operations in the intranode communication, where a peak bandwidth of 350.9 MB/s is achieved. *Address Translation Overhead Masking* hides the address translation overhead, around 12-13 μs for long messages, from the critical path in the internode communication. The *Push-and-Acknowledge Overlapping* can hide the acknowledge latency from the critical path. Among these optimizing techniques, *Push-and-Acknowledge Overlapping* can reduce most of the overheads in the internode communication, while *Cross-Space Zero Buffer* can significantly improve the communication bandwidth in the intranode communication.

Currently, the bandwidth of Fast Ethernet is still low compared with the peripheral bus bandwidth. We believe the next important step is to design a more general mechanism to work with multiple network interfaces using multiple processors.

References

[1] G. Ciaccio. “Optimal Communication Performance on Fast Ethernet with GAMMA”, *Proc. of International Workshop on Personal Computers based Networks Of Workstations 1998* (PC-NOW '98), Orlando, March 30/April 3, 1998.

[2] T. von Eicken, A. Basu, V. Buch and W. Vogels. “U-Net: A User-Level Network Interface for Parallel and Distributed Computing”, *Proc. of the 15th ACM Symposium on Operating Systems Principles* (SOSP’95), December, 1995.

[3] B. Falsafi and D. A. Wood. “Scheduling Communication on an SMP Node Parallel Machine”, *Proc. of the 3rd International Symposium on High-Performance Computer Architecture* (HPCA-3), 1997, pp. 128-138.

[4] “Intel Architecture Software Developer’s Manual Volume 3: System Programming Guide”, Intel Corporation.

[5] S. S. Lumetta and D. E. Culler. “Managing Concurrent Access for Shared Memory Active Messages”, *Proc. of the 12th International Parallel Processing Symposium* (IPPS’98), 1998, pp. 272-278.

[6] B. H. Lim, P. Heidelberger, P. Pattnaik and M. Snir. “Message Proxies for Efficient, Protected Communication on SMP Clusters”, *Proc. of the 3rd International Symposium on High-Performance Computer Architecture* (HPCA-3), 1997, pp. 116-127.

[7] S. S. Lumetta, A. M. Mainwaring and D. E. Culler. "Multi-Protocol Active Messages on a Cluster of SMP's", *Proc. of Supercomputing '97 High Performance Networking and Computing* (SC97), November, 1997.

[8] C. M. Lee, A. Tam, and C. L. Wang, “Directed Point: An Efficient Communication Subsystem for Cluster Computing”, *Proc. of the 10th International Conference on Parallel and Distributed Computing and Systems* (IASTED ’98), Las Vegas, 1998.

[9] J. Shen, J. Wang and W. Zheng. “A New Fast Message Passing Communication System for Multiprocessor Workstation Clusters”, Tech. Rep., Dept. of Computer Science and Technology, Tsinghua Univ., China, 1998.

[10] A. S. Tanenbaum. “Computer Networks”, 3rd Edition, Prentice-Hall International, Inc., 1996, pp. 207-213.

[11] Y. Tanaka, M. Matsua, M. Ando, K. Kubota and M. Sato. “COMPaS: A Pentium Pro PC-based SMP Cluster and its Experience”, *Proc. of International Workshop on Personal Computers based Networks Of Workstations 1998* (PC-NOW '98), Orlando, March 30/April 3, 1998.

[12] “Virtual Interface Architecture Specification. Ver. 1.0”, Compaq, Intel and Microsoft Corporations, 1997, http://www.giganet.com/.