COMB: A Portable Benchmark Suite for Assessing MPI Overlap

William Lawry, Christopher Wilson, Arthur. B. Maccabe[*] and Ron Brightwell[†]

April 2002

Abstract

This paper describes a portable benchmark suite that assesses the ability of cluster networking hardware and software to overlap MPI communication and computation. The Communication Offload MPI-based Benchmark, or COMB, uses two different methods to characterize the ability of messages to make progress concurrently with computational processing on the host processor(s). COMB measures the relationship between overall MPI communication bandwidth and host CPU availability. In this paper, we describe the two different approaches used by the benchmark suite, and we present results from several systems. We demonstrate the utility of the suite by examining the results and comparing and contrasting different systems.

1 Introduction

Recent advances in networking technology for cluster computing have led to significant improvements in achievable latency and bandwidth performance. Many of these improvements are based on an implementation strategy called Operating System Bypass, or OS-bypass, which attempts to increase network performance and reduce host CPU overhead by offloading communication operations to intelligent network interfaces. These interfaces, such as Myrinet [1], are capable of "user-level" networking, that is, moving data directly from an application's address space without any involvement of the operating system in the data transfer.

Unfortunately, the reduction in host CPU overhead, which has been shown to be the most significant factor in effecting application performance [7], has not been realized in most implementations of MPI [8] for user-level networking technology. While most MPI microbenchmarks can measure latency, bandwidth, and host CPU overhead, they fail to accurately characterize the actual performance that applications can expect. Communication microbenchmarks typically focus on message passing performance relative to achieving peak performance of the network and do not characterize the performance impact of message passing relative to both the peak performance of the network and the peak performance available to the application.

We have designed and implemented a portable benchmark suite called COMB, the Communication Offload MPI-based Benchmark, that measures the ability of an MPI implementation to overlap computation and MPI communication. The ability to overlap is influenced by several system characteristics, such as the quality of the MPI implementation and the capabilities of the underlying network transport layer. For example, some message passing systems interrupt the host CPU to obtain resources from the operating system in order to receive packets from the network. This strategy is likely to adversely impact the utilization of the host CPU, but may allow for an increase in MPI bandwidth. We believe our benchmark suite can provide insight into the relationship

[*]W. Lawry, C. Wilson, and A. B. Maccabe are with the Computer Science Department, The University of New Mexico, FEC 313, Albuquerque, NM, 87131-1386, {bill,riley,maccabe}@cs.unm.edu. This work was supported in part through the Computer Science Research Institute (CSRI) at Sandia National Laboratories under contract number SF-6432-CR.

[†]R. Brightwell is with the Scalable Computing Systems Department, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM, 87111-1110, bright@cs.sandia.gov. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

between network performance and host CPU performance in order to better understand the actual performance delivered to applications.

One particular characteristic that we are interested in determining, is whether an MPI implementation is able to make progress on outstanding communications independently of calls to the MPI library. MPI provides an immediate, or non-blocking, version of its standard send and receive calls that provide an opportunity for overlapping data movement with computation. When messages make progress independently of the host CPU(s), we refer to this semantic as *application offload*, since part of the application's activity or protocol is offloaded to the operating system or the network interface.

The rest of this paper is organized as follows. Section 2 describes the approach that our benchmark suite employs. Section 3 outlines the hardware and software components of the platform used for gathering results. We present these results along with an analysis and discussion of important findings in in Section 4. Section 5 describes related work. We conclude in Section 6 with a summary of the contributions of this research, and describe our intentions for future work related to this benchmark suite in Section 7.

2 Approach

Our main goal in developing this benchmark suite was to be able to measure overlap as accurately as possible while still being as portable as possible. We have chosen to develop COMB with the following characteristics:

- One process per node
- Two processes perform communication
- Either process may track bandwidth
- One process performs simulated computation
- Both processes perform message passing
- Primary variable is the simulated computation time

The COMB benchmark suite consists of two different methods of measuring the performance of a system, each with a different perspective on characterizing the ability to overlap computation and MPI communication. This multi-method approach captures performance data on a wider range of the systems and allows for results from each benchmark to be validated and/or reinforced by the other. The first method, the *Polling Method*, allows for the maximum possible overlap of computation and MPI communication. The second method, the *Post-Work-Wait Method* tests for overlap under practical restrictions on MPI calls. The following sections describe each of these methods in more detail.

```
read current time
for( i = 0 ; i < work/poll_factor ; i++ ){
   for( j = 0 ; j < poll_factor ; j++){
      /* nothing */
   }
   if(asynchronous receive is complete){
      start asynchronous reply(s)
      post asynchronous receive(s)
   }
}
read current time
```

Figure 1: Polling Method Psuedocode For Worker Process

2.1 Polling Method

The *polling method* uses two processes, one process, the *worker process*, counts cycles and performs message passing. A second, *support process*, runs on the second node and only performs message passing. Figure 1 presents pseudo code for the worker process. All receives are posted before sends. Initial setup of message passing as well as conclusion of same are omitted from the figure. Additionally, Figure 2 provides a pictorial representation of the method.

This method uses a ping-pong communication strategy with messages flowing in both directions between sender node and receiver. Each process polls for message arrivals and propagates replacement messages upon completion of earlier messages. After a predetermined amount of computation, bandwidth and CPU availability are computed. The polling interval can be adjusted to demonstrate the trade-off between bandwidth and CPU availability. Because this method never blocks waiting for message com-

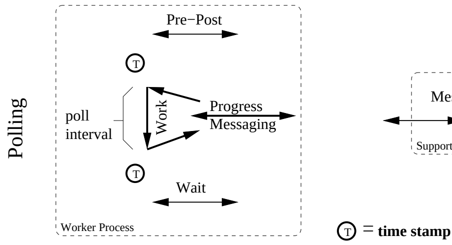

Figure 2: Overview of Polling Method

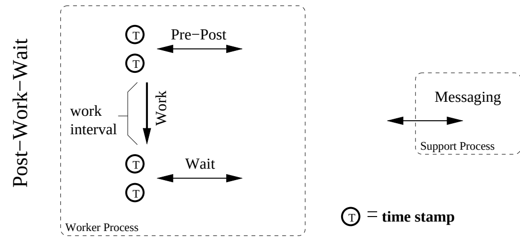

Figure 3: Post-Work-Wait Method

pletion it provides an accurate report of CPU availability.

As can be seen in Figure 1, after a fixed number of iterations in the inner loop the worker process polls for receipt of the next message. The number of iterations of the inner loop determines the time between polls and, hence, determines the polling interval. If a test for completion is negative, the worker process will iterate through another polling interval before testing again. If a test for completion is positive, the process will post related messaging calls and will similarly address any other received messages before entering another polling interval. The support process sends messages as fast as they are consumed by the receiver.

We vary the polling interval to elicit changes in CPU availability and bandwidth. When the polling interval becomes sufficiently large all possible message transfers may complete during the polling interval and communication then must wait, resulting in decreased bandwidth.

The polling method uses a queue of messages at each node in order to maximize achievable bandwidth. When either process detects that a message has arrived, it iterates through the queue of all messages that have arrived, sending replies to each of these messages. When we set the queue size to one, a single message passed between the two nodes then the polling method acts as a standard ping-pong test and maximum sustained bandwidth will be sacrificed.

The benchmark actually runs in two phases. During the first, *dry run*, phase the amount of time to accomplish a predetermined amount of work in the absence of communication is recorded. The second phase records the time for the same amount of work while the two processes are exchanging messages. The CPU availability is reported as:

$$\text{availability} = \frac{\text{time(work without messaging)}}{\text{time(work plus MPI calls while messaging)}}$$

The polling method reports message passing bandwidth and CPU availability, both as functions of the polling interval.

2.2 Post-Work-Wait Method

The second method, the *post-work-wait method* or PWW, also uses bi-directional communication. However, this method serializes MPI communication and computation. The worker process posts a collection non-blocking MPI messages (sends and receives), performs computation (the work phase), and waits for the messages to complete. This strict order introduces a significant (and reasonable) restriction at the application level. Because the application does not make any MPI calls during its work phase, the underlying communication system can only overlap MPI communication with computation if it requires no further intervention by the application in order to progress communication. In this respect, the PWW method detects whether the underlying communication system exhibits application offload. In addition, as we will describe, this benchmark identifies where host cycles are spent on communication.

Figure 3 presents a pictorial representation of the PWW method. This method is similar to the polling method in that each process sends and receives messages, but only the worker process monitors CPU cycles.

With respect to communication, the PWW method performs message handling in a repeated pair of operations: 1) make non-blocking send and receive calls and 2) wait for the messaging to complete. Both processes simultaneously send and receive a single message. The worker process performs work after the non-blocking calls before waiting for message completion. As in the Polling method, the work interval is varied to effect changes in CPU availability and bandwidth.

The PWW method collects wall clock durations for the different phases of the method. Specifically, the method collects individual durations for i) the non-blocking call phase, ii) the work phase, and iii) the wait phase. Of course, the method also records the time necessary to do the work in the absence of messaging. These phase durations are useful in identifying communication bottlenecks or other causes of poor communication.

It is worth emphasizing here that the terms “work interval” and “polling interval” represent the foremost difference between the PWW method and the Polling method. After the polling interval, the Polling method checks whether or not there are arrived messages that require response but *in either case* “computation” then proceeds via the next polling interval. In contrast, after PWW’s “work interval,” the worker process waits for the current batch of messages even if the messages have not begun to arrive, such as in the case of a very short work interval. This is one of the most significant differences between the two methods and is key to correctly interpreting the results.

3 Platform Description

In this section we provide a description of the hardware and software systems from which our data was gathered.

Each node contained a 500 MHz Intel Pentium III processor with 256MB of main memory and a Myrinet [1] LANai 7.2 network interface card (NIC). Nodes were connected using a Myrinet 8-port SAN/LAN switch.

The supported message passing software from Myricom for Myrinet is GM [9], which consists of a user-level library, a Linux driver and Myrinet Control Program (MCP) which runs on the NIC. Myricom also supplies a port of the MPICH [5] implementation of the MPI Standard. Our results were gathered using GM version 1.4, MPICH/GM version 1.2..4, and a Linux 2.2.14 kernel.

Results were also gathered using the Portals 3.0 [2, 3] software designed and developed by Sandia and the University of New Mexico. Portals is an interface for data movement designed to support massively parallel commodity clusters, such as the Computational Plant [4]. In particular, the semantics of Portals 3.0 support *application offload*. We have also ported the MPICH implementation of MPI to Portals 3.0.

The particular implementation of Portals for Myrinet used in our experiments is kernel-based. The user-level Portals library interfaces to a Linux kernel module that processes Portals messages. This kernel module in turn interfaces to another kernel module that provides reliability and flow control for Myrinet packets. This kernel module works with a Sandia-developed MCP that simply acts as a packet engine. This particular implementation of Portals does not employ OS-bypass techniques.

4 Results and Analysis

Figure 4 shows the results of the polling method for Portals using message sizes of 10 KB, 50 KB, 100 KB, and 300 KB. In the CPU availability graph, availability remains low and relatively stable until it rises steeply. Before the steep increase, polling is so frequent that messages are processed as soon as they arrive. This keeps the system active with message handling and availability is kept low due to related interrupts to the OS with this particular version of Portals. CPU availability steeply climbs when the poll interval becomes infrequent enough to cause stops in the flow of messages; lack of message handling equates to lack of interrupts and the application no longer competes for CPU cycles.

Figure 5 shows the bandwidth calculated by the polling method. Initially , the messaging bandwidth graphs exhibit a plateau of maximum sustained bandwidth until a point of steep decline. The point of steep decline occurs when the poll interval becomes large enough that all messages in flight are completed during the poll interval. When this happens, messages are delayed until the occurrence of the next poll.

The PWW availability graph, Figure 6, lacks the initial plateau as seen in the polling availability graph for Portals. This difference is due to the fact that the polling method returns to work (i.e., to another polling interval) if a message has not yet arrived, whereas the PWW method waits regardless of what the cause is for the delay. This *wait while delayed* functionality suppresses apparent CPU

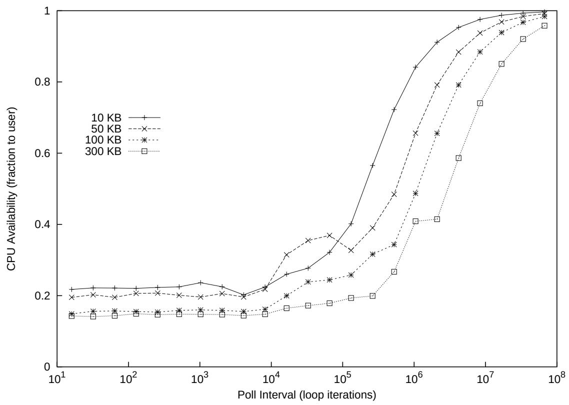

Figure 4: Polling Method: CPU Availabilty

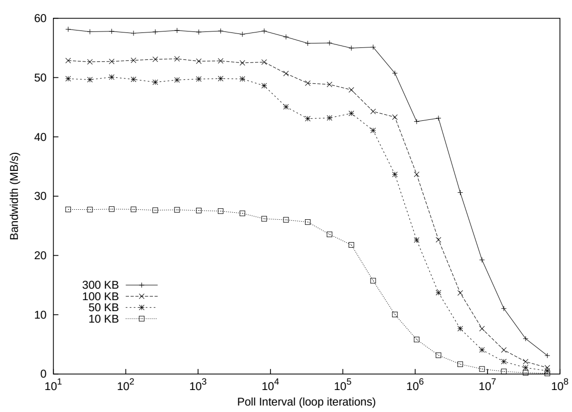

Figure 5: Polling Method: Bandwidth

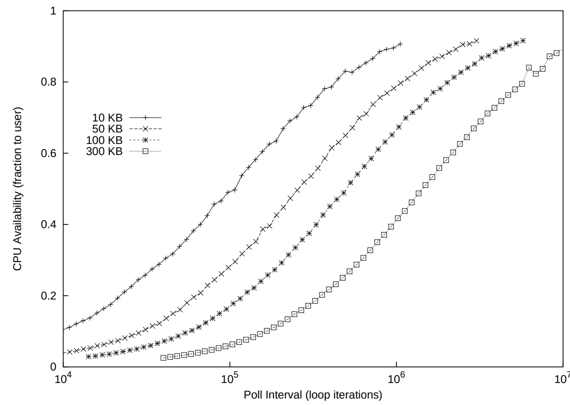

Figure 6: PWW Method: CPU Availability

availability until the work interval becomes sufficiently long to fill the delay period of time.

Figure 7 shows bandwidth as calculated by the PWW method. Compared to the bandwidth graph for the polling method, we see a more gradual decline in bandwidth as the work interval increases. This is due to the ability of the polling method to maintain sustained peak bandwidth for longer polling intervals.

4.1 Testing for Application Offload

An important characteristic of communication systems that we wanted to be able to identify is whether or not the system provides application offload. In this section we describe how results from the two methods can be used to analyze and compare systems.

Figure 8 shows the bandwidth performance of GM and Portals using the polling method. From the graph we can see that the performance of GM is significantly better than Portals on identical hardware. We would expect this to be true given what we know about the implementations of each system. GM is implemented using OS-bypass techniques and is able to deliver messages directly from the NIC to the application without interrupts or moving data through buffers in kernel space. In contrast, Portals is implemented using interrupts and copies data into user-space from kernel buffers. The reliance on interrupts and memory copies each causes a significant performance degradation for Portals.

Figure 9 shows the bandwidth performance of GM and Portals using the PWW method. Again, we see that the performance of GM significantly better than Portals for smaller work intervals.

However, if we look at the different phases of the PWW method more closely, we can gain more insight into these two systems. Figure 10 shows the average time to post a receive in the PWW method. Again, GM significantly outperforms Portals. In contrast, Figure 11 represents the duration of the wait phase or the time expended waiting for message completion. This graph indicates that, given a large enough "work" interval, Portals will virtually complete messaging whereas GM will not. Recall that, in the PWW method, the communication system will not make progress unless it can proceed with messaging based on only the initiating non-blocking posts. Therefore, this graph indicates that GM does not provide application offload while Portals does.

4.2 CPU Overhead

We now examine the work phase of the PWW method. The duration of the work phase is of interest when considering communication overhead. Depending on the system, a separate process or the kernel itself could facilitate communication while competing with the user application for CPU time. In such cases, the time to complete

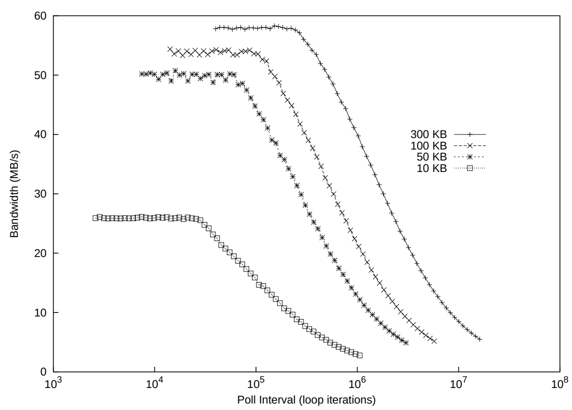

Figure 7: PWW Method: Bandwidth (Portals)

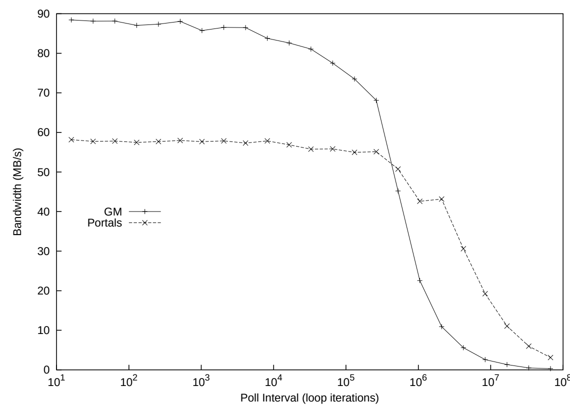

Figure 8: Polling Method: Bandwidth for GM and Portals

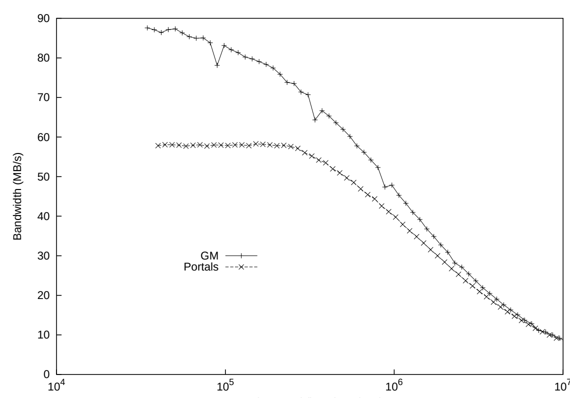

Figure 9: PWW Method: Bandwidth for GM and Portals

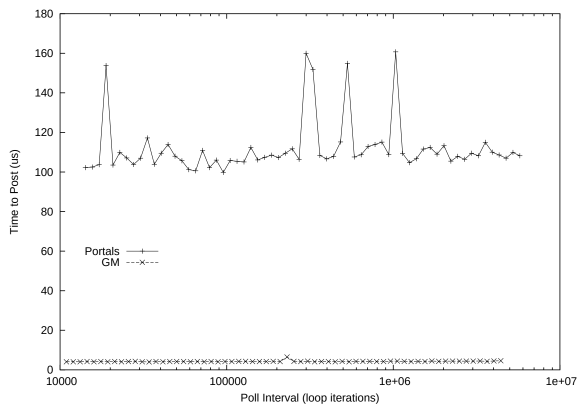

Figure 10: PWW Method: Average Post Time (100 KB)

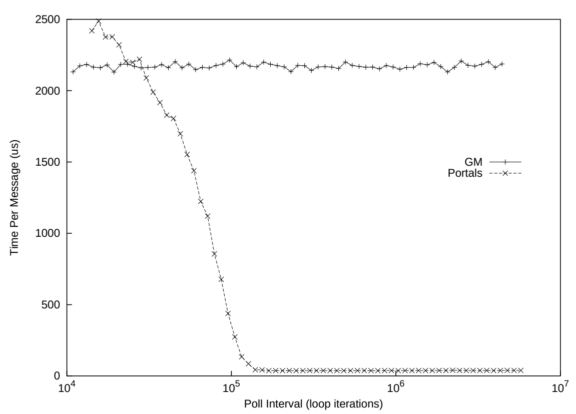

Figure 11: PWW Method: Average Wait Time (100 KB)

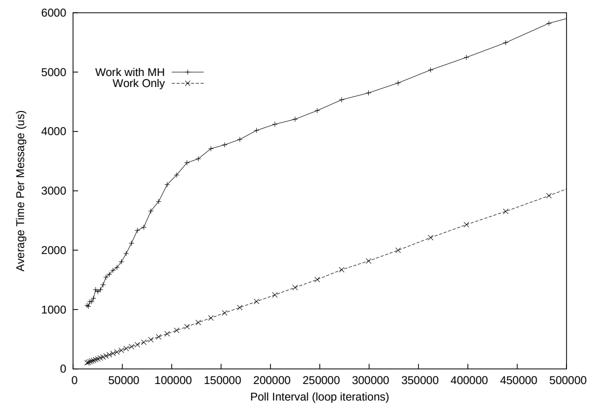

Figure 12: PWW Method: CPU Overhead for Portals

the work phase during messaging will take longer than the time to complete the same work in the absence of a competing process.

Figure 12 depicts a PWW run on Portals. The graph shows time to complete work as a function of work interval. Recall that both methods time the duration needed to complete work with and without communication. In Figure 12, the work with message handling takes a greater amount of time relative to work without message handling; the difference is due to the overhead of interrupts needed to process Portals messages.

In contrast, Figure 13 displays results for GM and shows virtually no communication overhead in that the time to do work is the same regardless of the presence or absence of communication. The lack of a time gap between work with and without message handling is the general indicator of a system that lacks communication overhead. However, one needs to check a little further for systems which lack *application offload* as does GM.

What about systems like GM that lack *application offload*? Message handling is blocked during the work phase of PWW. Because message handling is blocked, there ought to not be any communication overhead during the work phase as reflected in Figure 13.

When a system does not have application bypass, we can look to the results of the polling method to assess whether the system has communication overhead. Consider Figure 14 which shows the relationship between

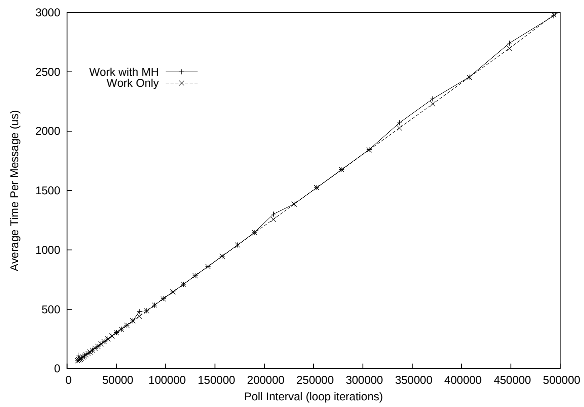

Figure 13: PWW Method: CPU Overhead for GM

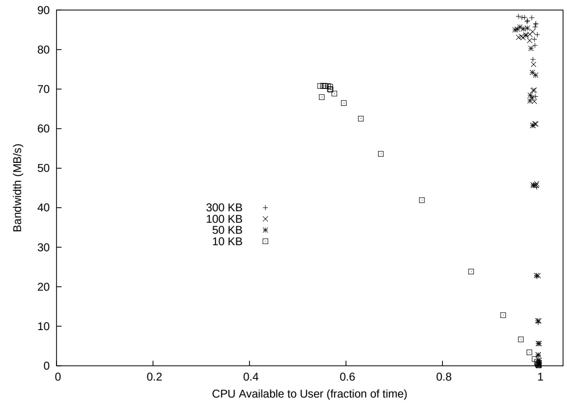

Figure 14: Polling Method: Bandwidth Versus CPU Overhead for GM

bandwidth and availability.

Note that in Figure 14, virtually all of the CPU cycles are given to the application for work while the network concurrently operates at maximum sustainable bandwidth; this testifies to the *OS offload* to the NIC for GM. If GM had communication overhead then the Polling data in Figure 14 would rather have the shape of Figure 15. Figure 15 reflects the Portals communication overhead which restricts maximum sustained bandwidth to the lower ranges of CPU availability.

Finally, compare Figure 15 with Figure 14. As previously discussed, Figure 15 reflects the communication overhead in terms of restricting maximum bandwidth to lower CPU availability. For GM, Figure 14 shows the lack of overhead except for the 10 KB message size. This difference is due to the large versus small message protocols. For small messages, messages less than about 16 KB, GM spends an increased amount of time in the non-blocking send (about 45 microseconds per message versus about 5 microseconds with larger messages on our system). With this extra time, GM completes the application tasks with respect to sending the small message but the result is in decreased CPU availability to the applicaiton as shown in Figure 14.

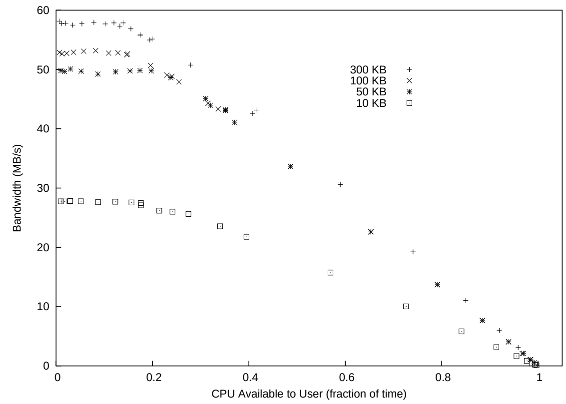

Figure 15: Polling Method: Bandwidth Versus CPU Overhead for Portals

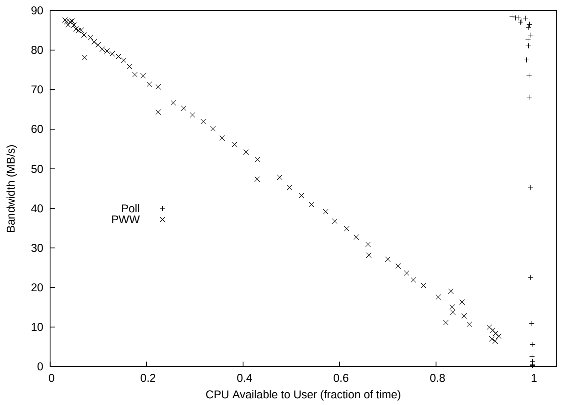

Figure 16: Polling and PWW Method: Bandwidth for GM

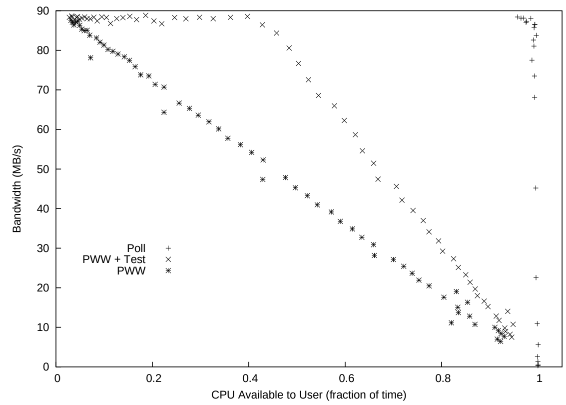

Figure 17: Polling and Modified PWW Method: Bandwidth for GM

4.3 MPI Library Call Effect

We have asserted that the PWW method detects lack of application offload. We considered that, if this is truly the case, then inserting a library call into the work phase should extend the maximum sustained bandwidth into higher CPU availabilities with MPICH/GM. We chose to insert the one MPI library call contained in the polling method that is not used in the PWW method: `MPI_Test()`.

Figure 16 plots bandwidth versus CPU availability for the standard COMB methods. Note that the benchmark methods do not directly control availability. Instead, the methods of the benchmark control the polling/work interval and Figure 16 depicts the elicited relationship between bandwidth and availability.

We inserted one call to `MPI_Test()` early in the work phase of the PWW method. The results are shown in Figure 17. For reference, the data from Figure 16 are replotted in Figure 17. Clearly, the added library call has aided the underlying system in progressing communication.

Previous versions of the PWW method interleaved three and four batches of messages such that after completion of one batch the communication pipeline was still occupied with a following batch. The purpose was to keep a large numbers of messages in flight for full detection of maximum sustained bandwidth. While the results from such interleaving provides useful information, it is redundant with information from the polling method, and can lead to this *MPI call* effect. A high degree of inter-leaving necessitates the interspersing of MPI calls for other message batches inside of the PWW timing cycle of the current batch.

We should point out that this behavior is typical of many MPI implementations for OS-bypass-enabled transport layers. Since the mechanism required to progress communications is embedded in the MPI library, an application must make frequent library calls in order for data to move. This is actually a violation of the Progress Rule in the MPI Standard which states that non-local message passing operations will complete independently of a process making library calls.

5 Related Work

Previous work related to assessing the ability of platforms to overlap computation and MPI communication have simply characterized systems as being able to provide overlap for various message sizes [11]. Our benchmark suite extends this base functionality in an attempt to gather more detailed information about the degree to which overlap can occur and the effect that overlap can have on latency and bandwidth performance. For example, our benchmark suite is able to help assess the overall benefit of increasing the opportunity to overlap computation and MPI communication at the expense of decreasing raw MPI latency performance.

The netperf [6] benchmark is commonly used to measure processor availability during communication. Our benchmarks uses the same general approach as that used in netperf. Both benchmarks measure the time taken to execute a delay loop on quiescent system; then measure the time taken for the same delay loop while the node is involved in communication; and report ratio between the first and second measurement as the availability of the host processor during communication. However, in netperf, the code for the delay loop and the code used to drive the communication are run in two separate processes on the same node.

Netperf was developed to measure the performance of TCP/IP and UDP/IP. It works very well in this environment. However, there are two problems with the netperf approach when applied to MPI programs. First, MPI environments typically assume that there will be a single process running on a node. As such, we should measure processor availability for a single MPI task while communication is progressing in the background (using non-blocking sends and receives). Second, and perhaps more important, the netperf approach assumes that the process driving the communication relinquishes the processor when it waits for an incoming message. In the case of netperf, this is accomplished using a *select* call. Unfortunately, many MPI implementations use OS-bypass. In these implementations, waiting is typically implemented using busy waiting. (This is reasonable, given the previous assumption that there is only one process running on the node.)

6 Summary

In this paper, we have described the COMB benchmark suite that characterizes the ability of a system to overlap computation and MPI communication. We have described the methods and approach of COMB and demonstrated its utility in providing insight into the underlying implementation of communication system. In particular, we have demonstrated the benchmark suite's ability to distinguish between systems that provide application bypass semantics and those that do not.

Of the two methods used in the suite, the polling method is distinguished by providing a basis for viewing a systems performance in an unfettered manner. The polling method makes periodic calls to the MPI library and logs computation whenever the user application does not need to progress messaging. The result is that maximum overlap between communication and computation is allowed regardless of how the system might implement application offload and/or OS offload. As such, the polling method provides a basis for an unqualified or general comparison between different systems.

In contrast, the PWW method identifies actual limitations with respect to application offload. Although there may be some cost in terms of suppressed CPU availability in the low range, this method detects whether a system requires multiple MPI library calls in order to make communication progress. The PWW method also provides timing information which identifies where the hosts spent time on communication – whether it be as overhead during the work phase, as a prolonged time in the non-blocking posts, or potentially as some amount of time in the wait phase. As such, the PWW method provides performance comparisons in the area of application offload as well as provides a means to help identify bottlenecks during the post-work-wait cycle.

We believe COMB is a useful tool for the analysis of cluster communication performance. We have used it extensively to benchmark several systems, both development and production, and it has provided new insights into the effects of different implementation strategies. COMB has also been used by other researchers to assess their NIC-level messaging system for Gigabit Ethernet with programmable Alteon NICs [10].

7 Future Work

Our future efforts will take three paths. Our immediate goal is to make both of these benchmarks available to the community where they can be used to characterize the performance of other systems. Second, we plan to address multi-processor nodes. Our current method for measuring CPU availability will not work on systems with multiple processors per node. Once we have addressed this issue, we plan to benchmark several of the DOE ASCI machines.

Acknowledgements

Jim Otto from Sandia National Labs was invaluable if getting our Cplant setup for testing and development. Pete Wyckoff from the Ohio Supercomputer Center offered lots feedback in the early stages of development and actually used an early version of the benchmark. Wenbin Zhu from the Scalable Systems Lab at UNM and Michael Levenhagen of Sandia National Laboratories ran more recent benchmark versions and helped with increasing cross-platform compatibility. Patricia Gilfeather of the Scalable Systems Lab at UNM offered lots of constructive criticism and helped to improve the general methodology employed in the benchmark.

References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] Ron Brightwell, Tramm Hudson, Rolf Riesen, and Arthur B. Maccabe. The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[3] Ron Brightwell, Bill Lawry, Arthur B. Maccabe, and Rolf Reisen. Portals 3.0: Protocol building blocks for low overhead communication. In *CAC Workshop*, April 2002.

[4] Ron B. Brightwell, , Lee Ann Fisk, David S. Greenberg, Tramm B. Hudson, Michael J. Levenhagen, , Arthur B. Maccabe, and Rolf Riesen. Massively parallel computing using commodity components. *Parallel Computing*, 26:243–266, February 2000.

[5] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[6] Rick Jones. The network performance home page. http://www.netperf.org/netperf/NetperfPage.html.

[7] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 85–97, New YOrk, June 2–4 1997. ACM Press.

[8] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[9] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[10] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit Ethernet message passing. In *Supercomputing*, November 2001.

[11] J. B. White and S. W. Bova. Where's the overlap?: An analysis of popular mpi implementations. In *Proceedings of the Third MPI Developers' and Users' Conference*, March 1999.