Performance Monitoring in a Myrinet-Connected Shrimp Cluster

Cheng Liao
cliao@cs.princeton.edu

Margaret Martonosi
mrm@ee.princeton.edu

Douglas W. Clark
doug@cs.princeton.edu

Depts. of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544

Abstract

Performance monitoring is a crucial aspect of parallel programming. Extracting the best possible performance from the system is the main goal of parallel programming, and monitoring tools are often essential to achieving that goal. A common tradeoff arises in determining at which system level to monitor performance information and present results. High-level monitoring approaches can often gather data directly tied to the software programming model, but may abstract away crucial low-level hardware details. Low-level monitoring approaches can gather fairly complete performance information about the underlying system, but often at the expense of portability and flexibility.

In this paper we discuss a compromise approach between the portability and flexibility of high-level monitoring and the detailed data awareness of low-level monitoring. We present a firmware-based performance monitor we designed for a Myrinet-connected Shrimp cluster. This monitor combines the portability and flexibility typically found in software-based monitors, with detailed, low-level information traditionally available only to hardware monitors. As with hardware approaches, ours results in little monitoring perturbation. Since it includes a software-based global clock, the monitor can track inter-node latencies accurately. Our tool is flexible and can monitor applications with a wide range of communication abstractions, though we focus here on its usage on shared virtual memory applications. The portability and flexibility of this firmware-based monitoring strategy make it a very promising approach for gathering low-level statistics about parallel program performance.

1 Introduction

Performance is the crucial driving factor in parallel programming, and thus detailed performance monitoring plays a key role. Initial performance studies on parallel programming are often performed using simulations. However, many deeper questions about the detailed behavior of the final system cannot be answered through simulation, because it is too slow to use on significant applications. Furthermore, it is often impossible to collect simulation-based results that fully reflect the effects of multiple processes and the operating system. Therefore, performance monitoring tools built on the actual system are often essential to achieving the best performance for the parallel programs.

A common tradeoff in parallel performance tools arises in determining at which system level to monitor performance information and to present results. High-level monitoring approaches often measure quantities directly tied to the software programming model, but unfortunately may abstract away lower-level hardware details that can be important to the programmer. Low-level approaches such as hardware monitors, on the other hand, can gather fairly complete performance information about the underlying system, but often at the expense of portability or flexibility.

Thus, some compromise must be struck among the portability, flexibility, and "software awareness" of high-level monitoring, and the monitoring detail of lower-level approaches. This paper describes a monitoring approach that achieves such a compromise. We have embedded extra performance monitoring code into the message sending/receiving firmware running on the LANai processor in the Myrinet Network Interface [4]. Our system can make accurate measurements of network latencies and I/O bus transfer times that would be difficult and in some cases impossible to measure directly in software. The monitoring system is also flexible: the same monitoring firmware can work with any software system running on the compute nodes. Finally, the monitoring code is more portable than hardware monitors offering similarly detailed statistics; it can be easily embedded into other parallel systems using the Myrinet interface, regardless of operating system or CPU node architecture.

The contributions of this work are two-fold. First, we discuss a monitoring strategy that is directly applicable to a large number of current research and commercial machines based on Myrinet networks, and that is conceptually applicable to an even broader set of machines as well. Second, we discuss concrete results of using the monitor on applications with different communication abstractions.

The remainder of this paper is structured as follows. In Section 2 we give an overview of the monitoring strategy and design and implementation issues, as well as the background of the Myrinet-based Shrimp system. We next discuss the accuracy and perturbation of the performance monitor in Section 3. Section 4 presents results from running applications on top of the performance monitor. Section 5 discusses related work. Finally, Section 6 describes some possible future work following on this research and Section 7 offers our conclusions.

2 Monitor Design and Implementation

There are two main ideas behind this work. First, we explore the utility of embedding performance monitoring code into network interface firmware, as a way of compromising between the flexibility and portability of high-level tools and the detail afforded by lower-level tools. The second main idea lies in using that detailed information, particularly latency performance signatures, to understand application behavior.

Programmable network interfaces have become increasingly common. They allow flexibility in low-level communication implementation and they offload work from the main processor. Monitoring at the firmware level on these network interfaces gives us access to crucial determinants of application performance not available in higher-level software. It also allows one to build monitoring infrastructure that applies to a range of different types of higher-level software programming models. For us, these have included:

- Science and engineering applications written using a shared memory model and running on top of a shared virtual memory (SVM) system [17].
- Message passing applications written using NX [1].
- A distributed filesystem written using sockets [8].

The monitor we describe here is primarily focused on identifying *communication* bottlenecks in programs, and does not include more comprehensive profiling analysis for other computation or memory bottlenecks that may be in the code. These could clearly be integrated to form a complete tuning environment. Our focus, instead, has been on the challenge of monitoring communication behavior difficult to observe at the software level. For this reason, Section 4 focuses on the first application category: shared virtual memory applications. We will also briefly discuss experiences with other applications in Section 4.3.

Another issue on programmable network interface is where to put which functionality, in the host or in the network interface. Currently the data communication between the host and the network interface, especially from the network interface to the host, is more expensive compared to the cost of data management inside the network interface. For this reason, we put almost all functionality in the network interface in order to reduce the perturbation. The host retrieves the performance data off-line. We will talk more about function-placement in Section 2.2.

2.1 Myrinet-based Shrimp and Monitor Overview

The Myrinet-based Shrimp cluster being monitored implements the virtual memory-mapped communication model (VMMC) [3] on a Myrinet network of PCI-based PCs. Myrinet is a high-speed local/system area network for computer systems produced by Myricom [21]. A Myrinet network is composed of point-to-point links that connect hosts and switches. Figure 1 depicts a block diagram of the system. At each node, an ordinary PC contains a network interface card. A blow-up of the network interface is also shown. The primary component of the network interface is a CPU (called the "LANai" processor) that processes queued message send/receive requests and manages data being sent to (or from) the compute node's memory. The functionality of the network interface is determined by the firmware it runs; the network interface is shipped with a default version of this firmware called the Myrinet Control Program (MCP). The VMMC implementation [9] we use has modified this MCP. The tool discussed in this paper is built specifically for Myrinet, but the general idea can also be applied to other programmable network interfaces.

The Myrinet-based cluster used in this study consists of 8 PCI-PCs connected to a Myrinet switch though Myrinet PCI network interfaces. Each PC is a Gateway P5-166 running Linux 2.0.24 with a 166 MHz Pentium CPU, a 512 KB L2 cache and 64 MB main memory. Each network interface is controlled by a 33 MHz LANai processor and has 1 MB SRAM.

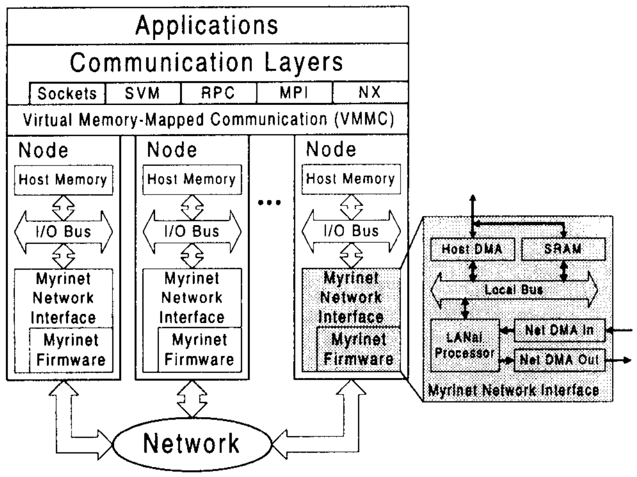


Figure 1: Myrinet-based Shrimp Cluster block diagram.

The core of our monitoring tool is implemented as additional MCP code run by the LANai processor. The network interface adds timestamps to outgoing messages, measures packet characteristics (and removes timestamps) as incoming messages are received, records statistics, and maintains a global clock. Our system maintains an array of statistics that include packet size, sender and receiver node IDs, and latencies in different parts of the system. These statistics are maintained in several multi-dimensional histograms so that the data can be post-processed to be presented in a number of ways. Currently these arrays occupy about 180 KB memory on the network interface. This number can be reduced if the histogram bins are widened. We also provide an interface to application to start and stop recording performance information on-the-fly. This makes it easier to pin down the problematic part in the application.

Figure 2 is an example of some of the data collected by the monitor. In this case, we have plotted the data as a multidimensional histogram showing the frequency of messages of different sizes being sent versus SourceLatency. Briefly, SourceLatency is the time to DMA the message from the host to the network interface. These histograms have proven useful in understanding performance issues in several types of programs. We will discuss the monitor's use further in Section 4.

2.2 Clock Synchronization

Accurate latency measurements are key to understanding program performance, and globally-synchronized clocks are the main requirement for getting them. However, globally-synchronized clocks are rarely available in loosely-coupled distributed machines. Although clocks are available on each

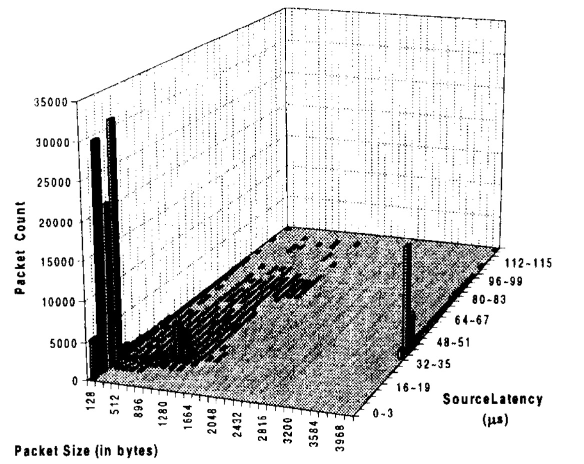
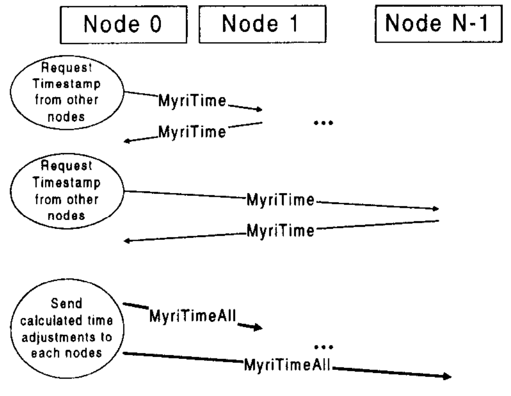


Figure 2: Example of data generated by the monitoring tool.

of the Myrinet network interface boards, the clock on one interface is not synchronized with (or even aware of) clocks on other boards. Thus, we have implemented a mechanism that synchronizes these board clocks periodically, in order to have a basis for accurately measuring communication latencies among nodes.



Node 0
Node 1
Node N-1
Request Timestamp from other nodes
MyriTime
MyriTime
Request Timestamp from other nodes
MyriTime
MyriTime
Send calculated time adjustments to each nodes
MyriTimeAll
MyriTimeAll

Figure 3: Basics of our clock synchronization algorithm.

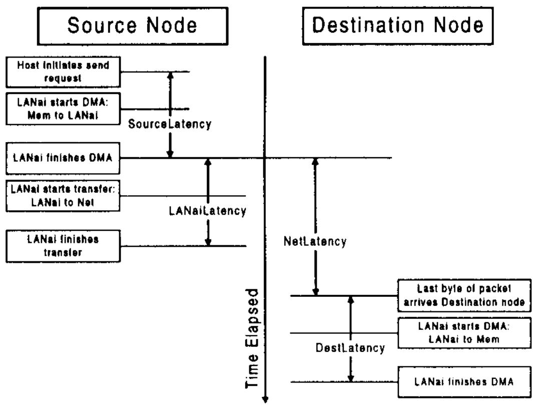
Our clock synchronization algorithm extends Christian's algorithm [6]. In our system, Node 0 is always responsible for collecting and distributing global clock information. Periodically, it queries other nodes for their current time. When the answer is returned, Node 0 computes the time difference between the pair. When Node 0 finishes the collection, it broadcasts the time difference table to all the other nodes, as shown in Figure 3. Other nodes then update their clocks based on the table received from Node 0. The centralized role of Node 0 reduces clock synchronization messages dramatically, but if this were our sole means of aligning clocks, we would need to perform a global re-synchronization roughly once per second. This frequent global re-synchronization, unfortunately, would impose a rather heavy overhead on the system.

Instead, to decrease global clock synchronization communication, we re-synchronize globally much less frequently—every 5 minutes in these experiments, rather than once per second. To maintain acceptable accuracy, we note that clock drifts are roughly constant within short time periods. With this assumption, we allow each node to "re-synchronize" *without* global communication by interpolating current time differences based on measured drift rates and the time difference table from the most recent global re-synchronization.

Such global clock synchronization algorithms have previously appeared at the software level, but we put the globally-synchronized clock code in the network interface for accuracy reasons. If we had put the synchronization code on the host, we would have to poll for the arrival of the timing packet or let the network interface to interrupt the host. The polling approach would apparently be infeasible because it wastes too many processing cycles on the host. And the interrupt approach would not be suitable either because the interrupt cost on our system is about tens of microseconds. This overhead would be too high to keep the global clock accurate at the microsecond level.

2.3 Latency Measurement

Obviously, an important issue in performance monitoring is choosing what to monitor. From our previous experience with the hardware performance monitor built on Shrimp-II and from speaking with application developers, we know that host-to-host latency of packets is a very important factor for understanding the performance of the system. As we already have a globally-synchronized clock, getting this latency is quite easy.



Source Node
Destination Node
Host initiates send request
LANai starts DMA: Mem to LANai
SourceLatency
LANai finishes DMA
LANai starts transfer: LANai to Net
LANaiLatency
LANai finishes transfer
NetLatency
Last byte of packet arrives Destination node
LANai starts DMA: LANai to Mem
DestLatency
LANai finishes DMA
Time Elapsed

Figure 4: Latencies measured by the monitor.

In order to better understand bottlenecks in communication, we break the latency of sending a message into several parts, as depicted in Figure 4. Each node's monitor tracks four different latencies for each packet. The first, *SourceLatency*, is the time between the first appearance of the sending request of this packet in the LANai's request queue, and the completion of the DMA into the LANai's memory. The second latency, *LANaiLatency*, measures the time between the end of SourceLatency and the end of the LANai's insertion of the packet into the network. These two latencies are measured entirely at the sending node. The third latency, *NetLatency*, measures the the total transport latency across the network: it begins as the sending LANai starts the transport, and ends when the receiving LANai gets the last word of the packet. NetLatency involves two nodes, and thus requires the clock synchronization techniques discussed in Section 2.2. Finally the fourth latency, *DestLa-*

tency, measures the time between the last word's arrival at the destination node, and the completion of the destination LANai DMA into its host's memory. DestLatency is local to the receiving node. Breaking latency into these pieces lets us isolate performance effects. For example, NetLatency would be hurt by network contention, while SourceLatency would be hurt by I/O contention.

We have found that the latency information is more useful if we combine it with packet length data. For example, if a one-word packet has the same latency as a one-page packet, there is clearly something wrong, but with latency information alone, we cannot notice it. That is the reason we keep several multi-dimensional histograms featuring both packet latencies and packet lengths.

Currently the histograms are kept in the memory of the network interface, so we have to sacrifice some precision due to the memory limitation. Currently, we count packet lengths in bins of 8-word granularity. SourceLatency, LANaiLatency and DestLatency are binned in multiples of $1\mu s$, while NetLatency uses multiples of $2\mu s$. We chose these parameters because we found that we could get fairly fine-grained data while still allowing the histograms to fit in the memory of the network interface. Memory limitations are also one reason why we did not include tracing functionality in our performance monitor.

An alternative that solves the memory limitation problem would be to keep the multi-dimensional histograms in host memory. If we chose to save the performance data in host memory, either the host would have to read the performance data in the network interface from time to time, or the network interface would have to DMA the data into host memory. Either way would involve extra overhead to move the performance data from network interface to host memory; thus it would certainly be contradictory to the goal of minimizing monitoring intrusion. As the data in Section 3 show, the performance monitor we discuss in this paper has a very small perturbation on the system.

2.4 Performance Signatures

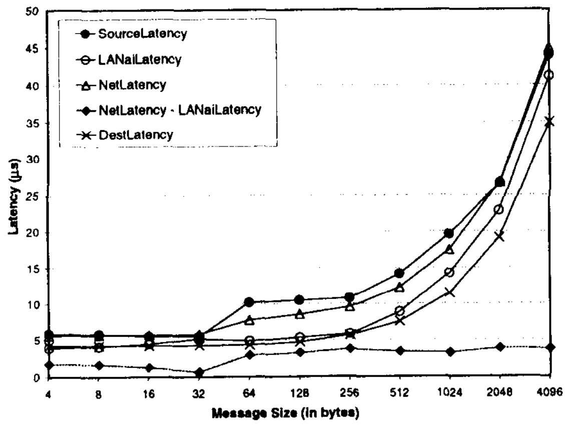


Figure 5: Baseline latencies for different message sizes. (Note that the X-axis is logarithmic.)

Each of the latencies discussed in the preceding section has a certain baseline value it can be expected to achieve, as shown in Figure 5. A number of factors can degrade performance from that ideal value. By examining the deviations of these latencies from their baseline values, summarized in a *performance signature*, users gain insight into which parts of the system are bottlenecks.

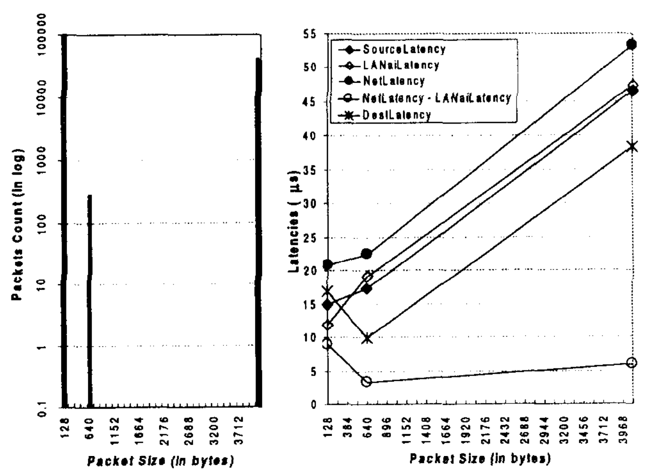


Figure 6: Performance signature output for FFT.

Figure 6 shows an example performance signature output. The output has two parts. On the right hand side is a graph of the different latency categories versus packet size. Each data point represents an average latency for all the packets of a particular size. Because of the data structure used for storing on-the-fly statistics, we can also create performance signatures versus sending node ID, receiving node ID, or packet type. Overall, this graph allows users to see where their program deviates from expected latencies.

Since the data points are averages, one also would like to know whether the average value was seen frequently, or whether it was an anomaly. For this purpose, the graph on the left hand side is included. This histogram plots the frequency of different packet sizes. (Note the semi-log scale.) We see here that the average latencies for a packet size of 128 are computed based on tens of thousands of packets. In contrast, only several hundred packets of size 640 bytes were sent out; that average latency may be less representative.

2.5 Monitoring Multiprogrammed Workloads

In many detailed performance studies, applications are studied in isolation, without considering the impact of other running programs, process scheduling, and other real-world effects. Real parallel programs rarely run in stand-alone mode, however; performance monitors for real systems must deal appropriately with multiprogramming effects. In Myrinet systems, once the packet is sent over the network, information about which program it is associated with would typically be lost. To avoid this problem, our monitor embeds a program tag in each packet, indicating which process produced the current message. This allows communication statistics on both the sending and receiving ends to be separated according to the program initiating the messages, so the performance data from different applications will be kept in different histograms.

3 Monitor Accuracy and Intrusiveness

Interesting challenges arise in checking the accuracy of a monitor of this type. For example, since the firmware-based monitor provides statistics that are otherwise impossible to get without a custom hardware design, it can be difficult to completely verify the accuracy of the statistics it produces without building the custom hardware monitor our scheme is intended to avoid. Program-level validations are insufficient because information on network latencies for individual packets is not available to software (which is why we used a firmware-based implementation in the first place). For this reason this section first shows that our monitor only has a small impact on the final performance of the applications. We then validate our monitor using a series of carefully-structured microbenchmarks for which the expected message latencies and bandwidths can be reasoned about or measured via other methods.

3.1 Execution Time Overhead

First, we give a feel for our performance monitoring's impact on the overall behavior of real programs. To address this, we have run several full applications with and without monitoring. These programs include SVM applications, NX applications and some applications running on a socket-based distributed file system. The overhead of monitoring, as measured at the program level, is quite low—less than 3% for all the applications. The microbenchmark characterizations of the following sections also validate the low-level measurements being made. Overall these results are extremely favorable compared to the previous alternative: simulations or custom-designed hardware-based monitors.

3.2 Clock Accuracy

An important part of our performance monitor is the globally-synchronized clock. As the LANai process does not use an interrupt mechanism, every event is handled in the MCP's main polling loop. This polling feature of the LANai may cause inaccuracies. For example, when a DMA operation finishes, the LANai cannot know this until it returns to main event loop, and then calculates the latency of this DMA operation. If the LANai is busy executing a long dedicated operation, it will not notice the end of the DMA right away, and will thus measure a latency larger than it should.

This problem is serious enough as it is, but when clock synchronization is involved, it becomes even worse. The global synchronization algorithm we discussed in Section 2.2 assumes that the latencies of the packet from Node 0 to other node and the packet from the same node back to Node 0 are the same. It is a reasonable assumption in the Myrinet-based network. However, when the scenario described in the previous paragraph happens, the two latencies we measure may differ significantly, even though the two actual latencies are still the same. Thus the time difference we calculate will not be correct, nor will NetLatency. Even if the two measured latencies are the same, there is still slight chance that they are measured incorrectly by the same amount, so the time difference calculated by them will still be incorrect.

To solve this problem we use several methods. First we try to keep each dedicated operation as short as possible. Second, for long, dedicated operations, we insert code into them to check if there are any events of interest, and to mark the timestamp of these events. These two methods cannot eliminate the polling delay completely, but they can ensure that the error will be negligible. In addition, we also boost the priority of clock synchronization packet arrival events in order to keep the global clock as precise as possible.

To measure the accuracy of our global clock, we use a program that alternatively sends same-length messages between all node pairs. The NetLatency of the messages on each pair should be the same. Our results show that the error of the global synchronized clock is less than $3\mu s$.

3.3 Latency and Bandwidth Accuracy

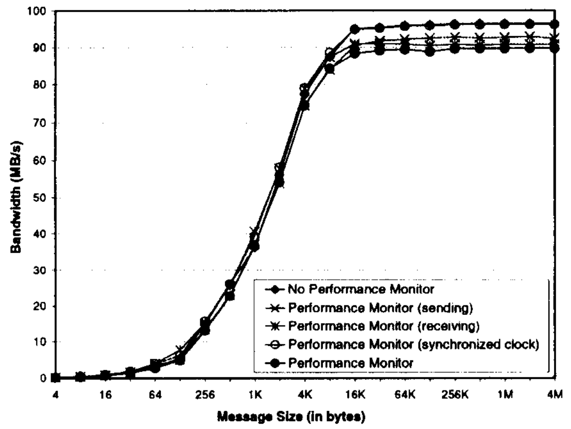


Figure 7: Unidirectional microbenchmark: Bandwidth measurements with varying degrees of monitoring.

To verify latency and bandwidth accuracy, we begin with a program that simply sends a stream of messages from one node to the other. We execute several runs of this program, each with a different message size. At the software level, the message sizes vary from just 4 bytes up to 4 MB, but for messages over 4 KB, the networking interface packetizes the messages into several 4 KB packets. Figure 7 gives the resulting average bandwidths for communication of this type, with varying levels of monitoring. The curve marked "No Performance Monitor" gives the bandwidth that can be achieved when the MCP does no monitoring at all. "Performance Monitor" gives the bandwidth that can be achieved when performance monitor is running. "Performance Monitor (sending)/(receiving)/(synchronized clock)" shows bandwidth when only the sending/receiving/ synchronized-clock part of the performance monitor is running, respectively. These numbers are computed simply by taking the total bytes sent by the program and dividing it by the total execution time. Since the microbenchmark does nothing but send messages, this is a reasonably accurate measurement of bandwidth. Overall, bandwidth measures are within 4% of the unmonitored time for message sizes over 128 bytes. For smaller messages, the effects on bandwidth will be higher, because the monitoring overhead is quite constant for messages of all sizes. Even so, the bandwidth measured when performance monitor is running is still within 10-15% of the unmonitored case.

A second microbenchmark we considered was a "bidirectional ping pong" program, which performs a series of alternating message sends and receives between two nodes. Figure 8 shows it has similar accuracy as in the unidirectional case.

From the bidirectional ping pong program we can also compute one-way host-to-host latency, which is parameter $RTT/2$ in the Berkeley LogP model [7]. This measures the time from when the sending node issues the send request until all the data arrives in the receiving node's memory. From Figure 9 we deduce that the overhead due to adding monitoring into MCP is constant at about $5\mu s$. We measure the host overhead of sending a message: O_s, which is

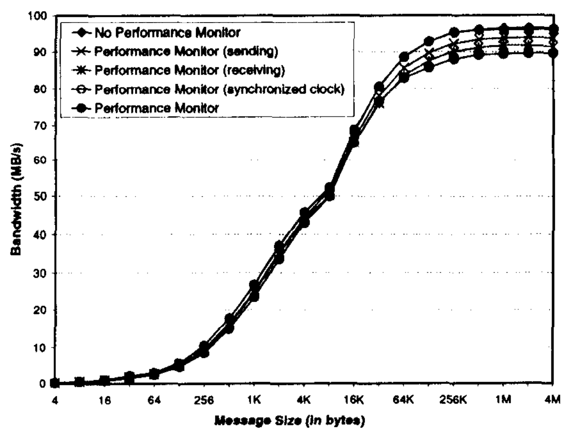


Figure 8: Bidirectional Ping Pong microbenchmark: Bandwidth measurements with varying degrees of monitoring.

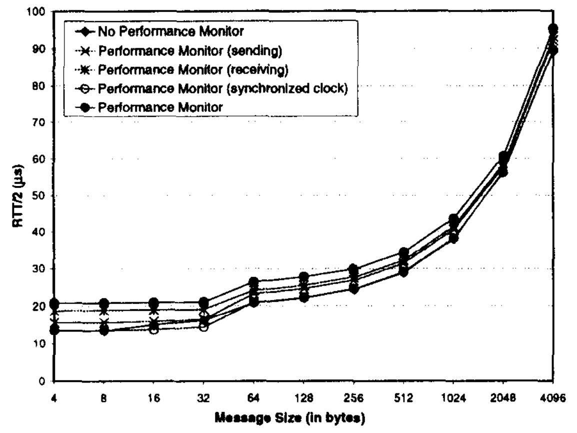


Figure 9: LogP microbenchmark RTT/2: One-way host-to-host latencies with varying degrees of monitoring.

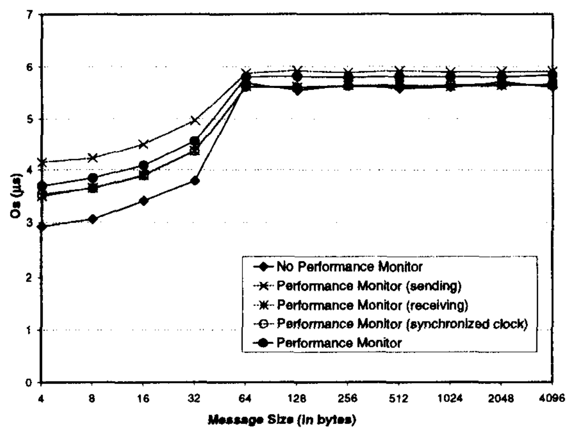


Figure 10: LogP microbenchmark O_s: Host overhead of sending a message with varying degrees of monitoring.

also a LogP parameter, by running the program to get unidirectional bandwidth. As shown in Figure 10, the extra overhead introduced by performance monitoring is less than $1.5\mu s$.

We also performed other microbenchmark studies, such as measuring the bandwidth of both nodes sending messages simultaneously, and others. All of these show little perturbation due to performance monitoring, and these small degradations are quite tolerable since they give access to measurements otherwise difficult to get.

4 Tool Usage on Applications

This section briefly describes our experiences using our tool on parallel applications. Although we have used our tool on applications of several different communication styles, we focus here to shared virtual memory programs.

4.1 SVM Applications: Overview and Methodology

Shared virtual memory (SVM) systems provide a software layer that presents programs with a global shared memory abstraction even if the underlying hardware does not provide support for global shared memory or cache coherence. In Shrimp, the SVM system runs on top of a basic message passing API (application programmer interface). This interacts with the Myrinet network cards to actually send and receive the messages that enforce coherence of the shared data.

The SVM system we use implements a home-based lazy release consistency (HLRC) protocol [29]. The HLRC protocol assigns each page to a *home* node. To alleviate the false-sharing problem at page granularity, HLRC implements a multiple-writer protocol based on using "twins" and "diffs" [15]. After an acquire operation, each writer of a page is allowed to write into its local copy once a clean version of the page (twin) has been created. Changes are detected by comparing the current (dirty) copy with the clean version (twin) and recorded in a structure called a *diff*. At a release operation, diffs are propagated to the designated home of the page and not to the other sharers. The home copy is thus kept up to date. Upon a page fault following a causally-related acquire operation, the entire page is fetched from the home. We have examined several SPLASH-2 [27] applications, but focus on two here: FFT and RADIX.

4.2 SVM Applications: Results

For FFT we use a 256K-point data size. The performance signature in Figure 6 shows that there is significant network contention. It is shown by "NetLatency - LANaiLatency", which increases from baseline values of 3-4μs to about 9μs. The large NetLatency indicates that there is contention inside the network interface as well. SourceLatency and DestLatency indicate I/O bus contention also. We can see SourceLatency of packets less than 128 bytes increases from less than 11μs in the baseline cases shown in Figure 5 to over 20μs here, while DestLatency increases from less than 5μs to about 16μs. This is because FFT has a bursty, all-to-all communication pattern that sometimes swamps the bus and network.

We also find some serious I/O bus contention effects in RADIX, as shown in Figure 11. The problem size we run is 2M keys. SourceLatency in this case is very high. This is because RADIX's access pattern is quite random, and many pages are modified at each node. Thus, diff computation puts a heavy load on the memory bus, because two whole pages have to be read for each page diff'ed.

Finally, Figures 12 and 13 show performance signatures

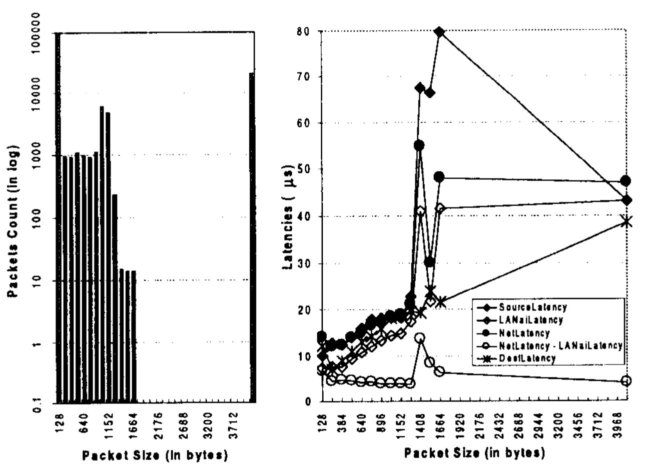


Figure 11: Performance signature output of RADIX.

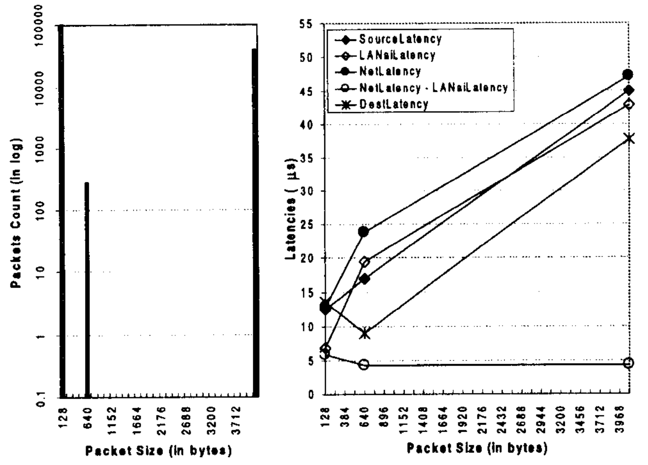


Figure 12: Performance signature output of FFT when run with RADIX.

obtained when running FFT and RADIX simultaneously. These indicate that network and network interface contention are alleviated in the multiprogrammed runs because of scheduling effects and the time dilation of the multiprogrammed execution. I/O bus contention for RADIX is also slightly reduced. On the other hand, we do not have gang scheduling or any coscheduling techniques implemented on the system, so we also have the problem that one program cannot be guaranteed to run at the same time on every node in the system. This leads to significant increases in data fetch and synchronization time. The slight improvement on packet latencies cannot compensate for this increase. Thus, the application execution time is much greater here than when they are run alone.

4.3 Monitoring Other Applications

Besides SVM applications, we have also used the monitor on a range of other applications, from message-passing programs using NX, to a full implementation of a distributed filesystem running on top of Shrimp sockets [8, 25]. Figure 14 shows the performance signature of an NX application: Gaussian Elimination on a 1024x1024 matrix. It is clear that SourceLatencies and LANaiLatencies are very high here. The reason is that in Gaussian Elimination, every node needs to broadcast the pivot value and the rest of the line it handles to other nodes. In the NX implementation we use, the broadcast function is implemented by separate VMMC sends to every other node in the system. In our 8-node system, each broadcast call will actually result in seven message sends observed by the network interface. Worse, even though the contents of these messages are the same, they have to be DMAed from the host to the network interface and from the network interface to the network seven times. Thus the send call in the end of the queue will wait a long time to be served.

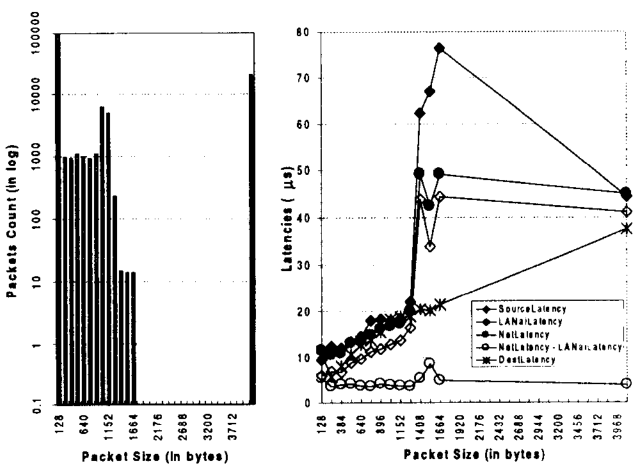


Figure 13: Performance signature output of RADIX when run with FFT.

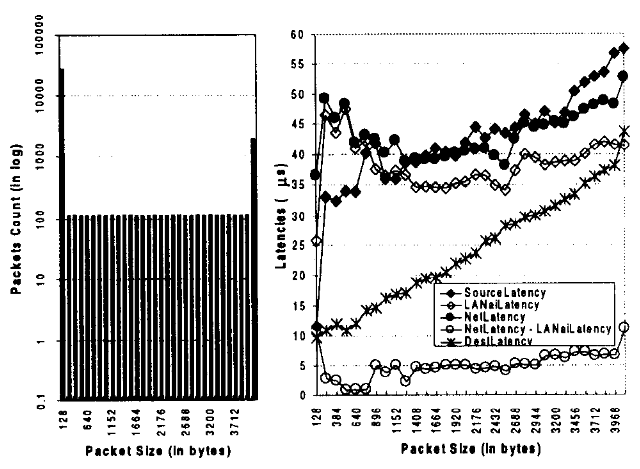


Figure 14: Performance signature output of Gaussian Elimination on NX.

The most pleasing aspect of our experience with this tool has been the inherent breadth of the monitoring approach. Because it is implemented *below* the programming model layers, it is useful across a wide range of programming models. Even in cases (such as SVM) where several levels of software lie between the application program and the

LANai, the low-level communication statistics have aided in performance-debugging the program. In many cases, the base monitor implementation is sufficient on its own, but in other cases, application-specific hooks can help gather additional information needed for a particular program. One of the advantages of the firmware-based implementation is the ease with which the monitor can be customized to include such hooks. With a hardware monitor, such incremental customizations would be infeasible. With a software monitor, the flexibility is there, but one cannot gather low-level latency statistics.

5 Related Work

This research builds on extensive prior work in parallel performance monitoring and tools. Unlike DOSMOS [5], our approach focuses on low-level data collection rather than user interface and protocol issues. Because of our desire to be portable across programming models, we do little automated analysis of the performance information as in other recent work more focused on particular software models [14, 22].

Our performance monitoring approach is somewhat similar to techniques proposed in the FlashPoint system [20]. That performance monitoring system embedded extra monitoring code into the cache coherence protocol handlers running on the FLASH multiprocessor's MAGIC chip. Our approach differs from FlashPoint in that we are using it on a machine without hardware cache coherence. Our Myrinet-based Shrimp cluster is used with a wider variety of programs than FLASH. For this reason, a low-level performance monitor that can be used for both shared-memory and message-passing programs is essential.

This performance monitoring tool also derives some ideas from our previous hardware-based performance monitor on Shrimp-II [19]. They both implement the functionality of keeping packet-based length and latency data in multiple-dimensional histograms. However, the hardware-based monitor is a separate board from the network interface. It does not consume the processing power of the network interface, so it has an even smaller perturbation on the system than the firmware-based approach. It also has additional memory so that tracing functionality can be implemented. On the other hand, implementing the firmware-based monitor takes far less time and effort than designing and building the hardware-based monitor. The firmware-based approach is also more portable.

Currently, PC or workstation clusters connected by programmable network interfaces are becoming the subject of considerable research. Several of them [10, 21, 23] use Myrinet network interface. While some of these projects have studied application performance on their clusters, we know of no publications specifically describing performance tools for them, nor any using our firmware-based approach. Other work has used only microbenchmark and statistical methods [7] or high-level software measurements [28].

There are some other research projects on other programmable network interfaces [11, 24, 26]. They also study the placement of functionality between the host and the network interface. However, the primary focus of these projects is on reducing the software overhead of communication to achieve maximum performance from the raw hardware, instead of on collecting the performance data of the system.

6 Future Work

There is much further work to be done with this tool. For example, in dealing with the performance data gathered, we could use more statistical parameters, such as variance.

One very important design goal of this firmware-based performance monitor approach is portability. Actually, we are now porting the performance monitor into the Windows NT platform. The port is going quite smoothly, and requires us only to rewrite part of the device driver. This demonstrates the portability of this style of monitoring.

Also, as we stated earlier in this paper, memory in the Myrinet network interface is quite precious, so we have had to sacrifice some precision in the multi-dimensional histogram and omit the functionality of tracing to reduce the memory requirements. As previous experience shows, the tracing ability can sometimes be quite important for understanding application behavior and for debugging. We are now studying an approach in which we move the main performance data into host memory and use memory in the Myrinet network interface as a cache. Certainly this approach will have more perturbation due to the performance monitor, but we hope the added finer granularity in the multi-dimensional histogram, and the extra trace data will compensate for the extra overhead. In terms of how to move the data from the network interface to host memory, we prefer DMA transfers by the LANai rather than reading from the network interface by the host. This is because DMA outperforms reading one word at a time whenever the block size is over 8 words. We have measured that to complete a DMA transfer of less than 128 bytes from network interface to host memory takes about $4\mu s$. If we can do this transfer when the LANai is idle, we may reduce this extra overhead further.

In many cases, the base performance monitor is sufficient on its own, but in some cases interaction with the communication layer above it can be helpful. For example, in some communication systems, a primitive is composed of several messages. In this case, coordination between the performance monitor and the upper-layer communication system is necessary to better understand the primitives. We have implemented an interface between the performance monitor and SVM system to study the page fetch, lock acquire and barrier operations in the SVM system [18]. We hope to build a general interface of this kind so it can coordinate with other communication systems easily and productively.

Moreover, in the future, we would like to explore the possibility of integrating dynamically adaptive functionalities into the performance monitor. Currently it can only record the performance data, and relies on off-line tools to analyze them. The ability to use data on-line to identify the bottlenecks and then fix them is very appealing. For example, migrating pages or even jobs based on the performance data collected can improve performance. This kind of ability may be even more rewarding in multiprogrammed cases because the performance monitor can get the whole picture, while individual applications cannot.

7 Conclusions

We believe that our firmware-based monitoring strategy is very attractive for gathering detailed statistics about parallel program performance in a way that is somewhat portable across both platforms and programming models. As also discussed in [20], parallel hardware is converging towards architectures in which compute nodes interconnected via network interfaces like Myrinet are common. Our monitoring code would work on any system currently using the Myrinet interface [2, 13, 23]; we also expect that porting to similar systems with other programmable interface processors would be fairly straightforward. To gather low-level statistics, our approach represents a significant improvement over previous reliance on simulation techniques or custom-designed hardware monitor boards [12, 16, 19].

Acknowledgments

This work was supported via DARPA contract N000014-95-1-1144, NSF grant MIP-9420653, and Intel Corporation. In addition, Margaret Martonosi is supported in part by an NSF Career Award CCR-95-02516. We thank the referees for their helpful suggestions.

References

[1] R. D. Alpert, C. Dubnicki, E. Felten, and K. Li. The Design and Implemenation of NX Message Passing Using SHRIMP Virtual Memory Mapped Communication. In *Proc. of 25th Intl. Conference on Parallel Processing*, pages 111–119, Aug. 1996.

[2] T. E. Anderson, D. E. Culler, D. A. Patterson, et al. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb. 1995.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proc. of the 21st Annual Symposium on Computer Architecture*, Apr. 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, Feb. 1995.

[5] L. Brunie et al. Execution analysis of DSM applications: A distributed and scalable approach. In *Proc. of SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 51–60, May 1996.

[6] F. Christian. Probabilistic Clock Synchronization. *Distributed Computing*, vol 3:146–158, 1989.

[7] D. Culler, L. Liu, R. P. Martin, and C. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, Feb. 1996.

[8] S. N. Damianakis, C. Dubnicki, and E. W. Felten. Stream Sockets on SHRIMP. In *Proc. of 1st Intl. Workshop on Communication and Architectural Support for Network-Based Parallel Computing*, Feb. 1997.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[10] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*, pages 256–266, May 1992.

[11] M. E. Fiuczynski and B. N. Bershad. A safe programmable and integrated network environment. In *WOP session of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[12] M. A. Heinrich. DASH Performance Monitor Hardware Documentation. Stanford University, Unpublished Memo, 1993.

[13] M. Hill, J. R. Larus, and D. A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *Proc. COMPCON*, Mar 1995.

[14] J. K. Hollingsworth. An online computation of critical path profiling. In *Proc. of SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 11–20, May 1996.

[15] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, Jan. 1994.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Protocol for the DASH Multiprocessor. In *Proc. 17th Annual Int'l Symp. on Computer Architecture*, May 1990.

[17] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, Aug. 1986.

[18] C. Liao, D. Jiang, L. Iftode, M. Martonosi, and D. W. Clark. Monitoring shared virtual memory performance on a Myrinet-based PC cluster. In *Proc. of the 12th ACM Intl. Conf. on Supercomputing*, 1998.

[19] M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP Performance Monitor: Design and Applications. In *ACM Sigmetrics Symposium on Parallel and Distributed Tools*, May 1996.

[20] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proc. ACM SIGMETRICS Conf. on Meas. and Modeling of Computer Systems*, May 1996.

[21] Myricom, Inc. Myrinet on-line documentation. http://www.myri.com:80/scs/documentation, 1996.

[22] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proc. of SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 31–40, May 1996.

[23] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, 1995.

[24] M.-C. Rosu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. In *Proc. of the 6th IEEE International Symposium on High Performance Distributed Computing*, Aug. 1997.

[25] R. A. Shillner and E. W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Princeton University Computer Science Department, Princeton NJ, 1996.

[26] P. Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer*, Mar. 1994.

[27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. of the 22st Int'l Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[28] K. Yocum, J. Chase, et al. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proc. 6th IEEE Intl. Symposium on High Performance Distributed Computing*, Aug. 1997.

[29] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proc. of the Operating Systems Design and Implementation Symposium*, Oct. 1996.