Debugging a Parallel Program: Capturing Inter-Processor Communication in an iWarp Torus

Thomas Gross and Susan Hinrichs
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

To understand or improve the execution behavior of a program on a parallel system, it is often necessary to consider the interaction between the processors in the system. Since communication is important for all parallel programs, obtaining information about the inter-process communication of the program is an important aspect of understanding program execution. If the user's concern is the performance of the parallel program, then the user must be able to capture the dynamic aspects of inter-processor communication. Unfortunately, timing information about inter-processor communication is often not easy to obtain. Building a special-purpose hardware performance monitor is too costly in most scenarios, and the use of VLSI to integrate communication and computation on a single component often provides only a few externally visible measurement points. A software monitor on the other hand is often too slow to allow execution of the monitored program without serious perturbation.

Although the integration of communication and computation engines on a single component provides some challenges to monitoring, it also opens the opportunity to program the processors in a parallel system so that they can monitor communication traffic. That is, during monitoring some processors execute the user program (to be monitored) while other processors execute a special monitor program (which captures information on the inter-processor communication). With adequate software tools, this information then can be analyzed to present a picture of the communication between the user processes. This paper discusses the benefits and difficulties of an implementation based on this idea for the iWarp system. We conclude that a programmable processor that integrates communication and computation is also suited to serve as a hardware monitor at a fraction of the cost of a special-purpose design.

1 Introduction

Debugging a program on a parallel system is difficult when the objective is to get a correct program. Getting a program to run fast (while maintaining its correctness) is even harder and requires a wide range of information about the execution behavior of the program. A performance debugger must provide information on how computation cycles, memory bandwidth, communication bandwidth, or other system

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

resources are spent by the program. With this knowledge the programmer knows where to concentrate her efforts in tuning the program to remove bottlenecks.

Obtaining, analyzing, and presenting performance information for a sequential system is sometimes difficult and still an active area of research[SK90]. For parallel systems, the situation is even worse; in addition to the information that must be obtained for each processor, we must understand the interaction between the processors in the parallel system.

We have been exploring issues in performance debugging while working on the iWarp system, a private-memory, MIMD, parallel computer[BCC+88]. A single-node `gdb`-like debugger exists to debug the code on individual nodes of the iWarp, but this provides only a pigeon-hole view of the execution of a program. The single-node debugger treats inter-processor communication by a node the same way input/output is treated by a uniprocessor debugger: after a message has been sent, it is invisible and cannot be tracked until it is received by another node (when the single-node debugger for that node is able to inspect the message). To understand or improve the execution behavior of a program on a parallel system, it is often necessary to monitor this inter-processor communication. If we can observe how messages travel through the parallel system, we can combine this information with the information about the execution on each node to provide the user with a global picture of system performance. Consider the example where processor A must receive messages from both processors B and C before it can begin some computation. Suppose the message from C will arrive long after the message from B has arrived. If the compiled program on processor A has committed to receive C's message first, processor A will block, waiting to receive the message from C, and processor B will block, waiting for its message to be received. A single-node debugger is of little help in identifying such situations, since it cannot capture the reason for the delay in processor B by solely inspecting this processor's state. Furthermore, a single-node debugger may perturb the execution of the inspected process. Some pertinent communication issues to measure are frequency of messages, the number of words transmitted, and specific patterns in the messages.

There exist two approaches to obtaining such communication information: using a hardware monitor or software instrumentation. There is a well-known cost versus accuracy tradeoff between these two options. Using additional hardware to monitor performance is more accurate, because the monitoring does not steal any resources from the monitored program. However, using additional hardware takes much more effort. The monitoring hardware must be designed and built, and since it is special purpose, it cannot be used when the programmer is not interested in monitoring. For this reason, the buyers of systems are usually unwilling to pay for additional monitoring hardware (and the component designers are unwilling to sacrifice significant area or design effort to provide it). Furthermore, as advances in VLSI allow a tighter coupling of communication and computation, the monitoring of inter-process communication becomes more and more difficult. On an iWarp system, a hardware monitor that attempts to observe the communication between two adjacent nodes must to understand an inter-processor bus protocol that includes resource allocation, routing, and flow-control. Once two or more processor nodes are integrated onto the same component, obtaining access to any communication between two such nodes will be next to impossible.

Software instrumentation costs less, because only the software is changed. The program or the system software can be instrumented to gather information. This option is more flexible than an approach based on a hardware monitor, but the additional software steals cycles and hardware resources from the original program, so the execution of the monitored program is perturbed. As communication overhead in parallel systems has decreased over the years enabling finer-grained communication, this software monitoring intrusion becomes less acceptable. For example, an iWarp node can send and receive four 32-bit words every 100 ns (the time it takes to perform either one single-precision floating point addition or multiplication or two integer operations), so “bracketing” each communication operation with monitoring code may slow down a program on the iWarp by up to a factor of 40.

However, a parallel system like iWarp offers a hybrid solution. Each iWarp processor contains a computation and communication engine, which are tightly coupled, providing the computation engine with a detailed view of the communication system. At the same time, the computation engine can be programmed like a general-purpose processor, providing the opportunity to use such a processor as a monitor processor. The absence of a global memory makes all communication explicit at the hardware level, so if we monitor the communication between processors, we can obtain a complete picture of the communication in the system.

For the iWarp array, we have created a hybrid performance monitoring system that has many of the benefits of hardware monitoring while incurring costs close to those associated with software instrumentation. Section 2 gives some background about the monitored parallel system. In Section 3 we describe this idea for performance monitoring on iWarp in detail. In Section 4 we evaluate the current implementation of this performance monitoring system. Section 5 describes tools to display the inter-processor information captured by the monitor and gives an example of its use.

2 Background

In this section, we describe some key aspects of the parallel system.

2.1 Communication

We distinguish between two communication styles: memory communication and systolic communication[BCC+90]. In memory communication, all communication between two processors is directed through the local memory of the sender and receiver processors. The sender assembles a message in memory and then passes it to the communication system, which transfers the message to the memory of the receiver processor. Message passing is a well-known paradigm that is based on memory communication.

For systolic communication, the processor is directly connected to the communication subsystem. On the sender processor, the words of a message are passed to the communication system as they are generated without any buffering in memory. Similarly, the receiving processor consumes the words directly from the communication system without first buffering them in memory.

With memory communication, the unit of transfer from the sender processor to the receiver is a block of data; each block contains a number of words. Memory communication is easier to monitor than systolic communication. If a sender program uses memory communication, the data of a message are first assembled in memory, then the communication system is invoked to transfer the data. The entry point into the communication system is well-defined, and the communication system can record information like the starting time of the message transfer, the message size, and (upon completion) the transfer bandwidth. Any overhead associated with such bookkeeping operation is paid for only once per message. However, recording information at the start and end of a message transfer is not sufficient if the programmer needs information about the transfer of the individual words of a message, e.g., if data are transmitted in bursts.

With systolic communication, the data of a message are produced (or consumed) on the fly. That is, data are sent as they are produced by a computation unit (e.g., the floating point adder). Since the data are generated by processor on the fly, the processor cannot be used to record any information like the time the item was sent. If we wanted to use the processor to capture such information, then the processor would have to stop generating data. So if we want to obtain timing information about systolic communication or if we need information for memory communication at a finer grain than complete messages, we cannot use the sending or receiving processor to monitor communication without serious perturbation of the program execution.

2.2 The iWarp system

The design of the iWarp component has been described earlier[BCC+88, BCC+90]; here we summarize only those aspects that are essential for the understanding of this paper.

The iWarp processor supports multiple *logical channels* between adjacent nodes, so multiple high speed connections can be set up using the same physical busses. On each node there is a finite number of queues, which buffer the data sent over the logical channels. A logical channel is assigned a queue on the source node and another queue on the destination node. These logical channels can be chained together to form a *pathway* by using the destination queue of one logical channel as the source queue of another logical channel. The communication hardware takes care of forwarding data through the intermediate nodes of chained logical channels, so the pathways provide direct, high-speed connections between nodes that are not physically adjacent. Pathways can be used to reserve resources for memory communication (message passing), or to set up connections for systolic communication.

An iWarp system is built out of iWarp processors and is arranged in an $n \times m$ torus. Each iWarp system is connected to a workstation that server as its front end, e.g., the output of a `printf` statement executing on a node appears on the file system of the front end. To date (August 1992), a number of systems have been built ranging in size from 4 to 256 nodes.

2.3 User model

The user seldom programs the iWarp system at the level of logical channels. Instead, she either uses a parallel program generator or describes the computation as a set of processes with connections between these processes. A parallel program generator like Assign [O'H91] or Apply [HWW89, BG91] takes a high-level description of a computation (that is, without any explicit statements for communication or data placement) and translates that description into a program for each node of the iWarp system with connections between nodes as appropriate. If the programmer describes the computation as a set of processes, a tool maps this set of processes onto the target iWarp torus, provided the number of processes is less than or equal the number of nodes in the target iWarp torus. After the processes are mapped onto the iWarp torus (either by the parallel program generator or the tool mentioned above), another tool sets up logical channels to implement a connection between any two nodes if the programs mapped onto these nodes are connected. The number of connections that can pass through a node is limited by the number of logical channels, which is taken into account when mapping the processes onto nodes. Figure 1 shows the mapping of a set of processes and connections onto an iWarp torus.

3 Monitoring inter-process communication

To get a handle on inter-process communication in a parallel system, it is necessary to start with the inter-processor communication. For example, if we want to observe the flow of data from Node 2 to

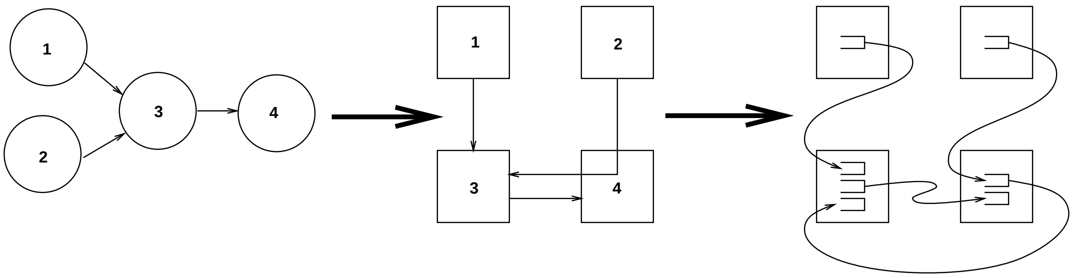

Figure 1: An example of mapping processes onto the iWarp array and mapping connections onto logical channels.

Node 3 in Figure 1, we must consider Node 4 as well, since the data travel through this node. Since the current runtime system allows only a single user process per node (and this is not likely to change in the future), information about the flow of data between processors can be directly mapped onto the inter-process communication of the user program.

If we know how the length of a logical channel queue changes over time, we have an idea of the run time communication patterns. For instance, while a queue is empty, we know either there is no communication going on over this logical channel or the receiving node is waiting for data from the sending node. While the queue is full, the sending node is sending faster than the receiving node is receiving. By watching how queues fill and empty during the course of a program's execution, the programmer can visualize the communication flow of her program and identify bottlenecks or dependences between messages. The actual number of words in a queue is less important at this level of analysis than the state (full, empty, partially_full) of the queue. If the queue is full, the sender cannot proceed. If the queue is empty, the receiver cannot proceed; otherwise both can go on.

To get an accurate picture from monitoring inter-process communication we must try to maximize the accuracy of the gathered data and minimize the perturbation caused by monitoring. Section 1 explained how pure hardware or pure software monitoring solutions are not practical for the iWarp system. A pure hardware solution could not gather sufficiently detailed data about the communication, and instrumenting all communication functions could add too much perturbation for programs with fine-grained systolic communication requirements. However, we notice that not all nodes of an iWarp array are used all the time, and these unused nodes have the same capabilities as the nodes used to execute the user program, so we can use these nodes to gather information about the execution of the user program. That is, some nodes are program nodes (executing the user program), and others are monitor nodes, capturing the inter-processor communication between program nodes. Since the monitoring programs are running on a different set of nodes, they are not stealing cycles from the user program, one of the advantages of using additional hardware to monitor performance. Since the nodes are programmable, no extra hardware needs to be designed, and when monitoring is not desired, the monitoring nodes can run user programs.

This hybrid approach of using some of the nodes in the parallel system for user program execution and the rest for monitoring is integrated into the standard communication tool chain [Hin91]. Whenever the user selects the monitoring option, the tool chain spreads out the connection definitions and node mappings, so monitoring nodes can be inserted between each adjacent pair of user program nodes. The pathways between any two nodes are lengthened to pass through the monitoring nodes. See Figure 2 for two examples of this transformation. The monitoring nodes (shown dashed) are loaded with the monitoring program.

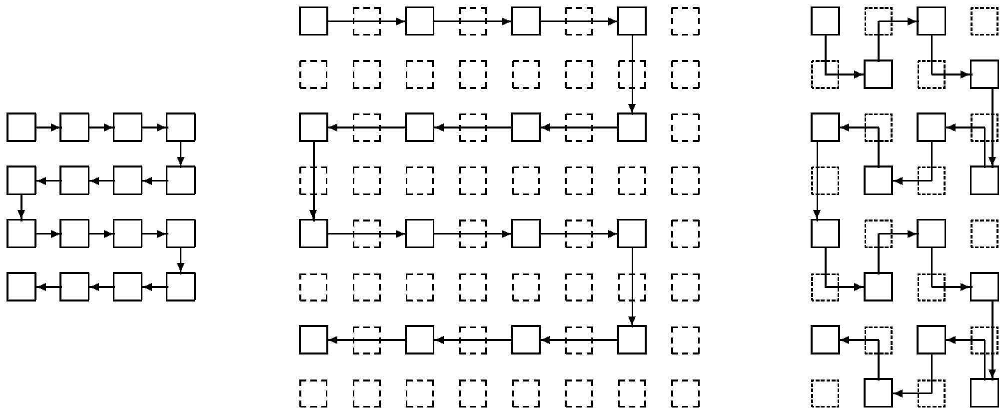

Figure 2: Network layout of original program is on the left. Two different network layouts of the monitored program are to the right. The current monitoring tool chain uses the center layout.

During the execution of the user program, the monitoring nodes gather data about the size of the logical channel queues of the pathways that pass through them. At the end of execution, this information is downloaded to a general-purpose workstation that is connected to the iWarp torus. There the programmer can use other tools to process, analyze and view the queue length data.

3.1 Gathering queue state data

To gather accurate queue length data while adding little perturbation, the monitoring program takes advantage of several features of the iWarp architecture. Each iWarp node consists of a computation agent and a communication agent that coexist on the same chip. These agents can execute asynchronously, or the computation agent can control the communication agent. The monitoring program runs the two agents asynchronously, so data passing on pathways through the monitoring node can continue without software interference. Since the computation and communication agents exist on the same chip, the computation agent can sample the state of the communication agent with little overhead. To determine the length of all network queues, the computation agent must read 4 control space registers, which takes 4 cycles.

The monitoring program also takes advantage of the hardware support for configuring multiple logical channels between nodes. The program's connections are mapped onto logical channels, and different connections use different logical channels with distinct physical queues, so the monitor program can easily differentiate between different program connections. If all the communication traffic was collected in a single queue, this division would be difficult to discern without additional protocol information and bookkeeping.

The monitoring program stores timestamps with the queue length data, so the programmer can see how network queue length changes over time. It is also interesting to compare how queue lengths change over time between different nodes. To make this comparison accurately, there must be some sense of global system time. Each iWarp processor contains two clock timers (with a resolution of 8 clocks, i.e. 400 ns on a 20 MHz system), and one of these timers is reserved for the user program. Our implementation sets the user timers of all nodes to a common global time in two steps using a synchronization package developed locally[FGOS92].

3.2 Processing queue state data

The monitoring program stores the gathered data in a buffer. At the end of execution or when the buffer fills up, the monitoring program sends this data to the host processor. If the data is transferred after the end of the monitored program's execution, the time and bandwidth spent sending the data is not critical. However, if the buffered data must be sent during execution of the user program, this transfer steals bandwidth from the monitored program, and the queue length cannot be monitored by this node while it is sending the data. This problem could be avoided by halting execution of all nodes in the iWarp torus. However, since the monitoring nodes pre-process the samples on the fly (see below), there has been no need so far to download the monitoring data before the end of the user program's execution.

Forms of data compression can be used to reduce the amount of sampled data stored on the monitoring node. We have observed that network queues are often in a steady state, so the queue length from the last sample is often the same as the queue length of the current sample. Data from these two samples can be merged into one entry. This sort of data compression should be performed some time before the data is analyzed by the programmer to merge large blocks of redundant data. Other standard data compression techniques can also be used to reduce the amount of space needed to store individual entries. These data compression techniques can be performed during the sampling loop to reduce the amount of space needed in the monitoring buffers and so increase the amount of time before the monitor data buffer is filled. However, adding this additional computation to the critical sampling loop may reduce the queue length sampling rate.

To determine the cost of the data compression, we created two versions of the time critical sampling loop in assembly code. The first version did no data compression and stored all samples in the monitoring buffer. The second version compared the current sample with the last stored sample, and only stored the current sample if it was different. On a set of example programs, the first version on average took 31 cycles per iteration. The second version on average took 30 cycles per iteration. It may seem surprising that the loop speeds are so similar, even though the first loop is simpler and performs fewer conditionals. However, often the queue length does not change between between samples, so the second loop performs fewer stores, and the comparisons are slightly faster than the stores. We decided to use the version of the loop that performed data compression in the loop, because the times of the two loops were very close and the second loop consumes far less buffer space.

4 Evaluation of the current implementation

We implemented a prototype version of the monitoring system described in the previous section. This section discusses how close we came to our goals of minimizing perturbation of the monitored program and maximizing accuracy of the gathered network data.

4.1 Monitored program perturbation

From timing several monitored programs it appears that the monitoring code does not significantly affect the execution time of the monitored program. To get a better idea of how monitoring affects the execution of the monitored program, we looked at individual factors that might skew the execution.

Spreading networks over monitoring nodes can cause two types of skew. First, each network must go over twice as many links, so the time to transmit a word doubles. The other possible source of skew is the addition of network queues. Each monitoring node adds one queue to the network capable of buffering 8 additional words. If a sender and receiver are just barely blocking, the additional buffering may keep the sending node from blocking.

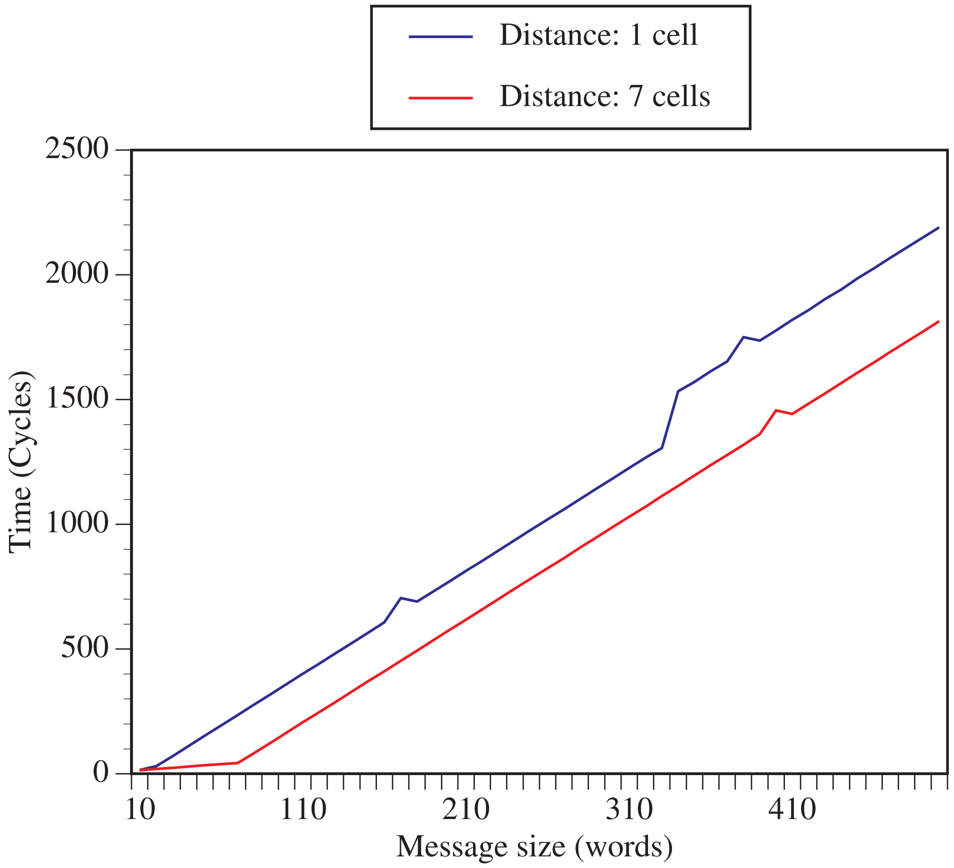

Figure 3: Graph of message size versus message sending time when the sender is faster than the receiver.

To determine the effect of longer networks, we ran and timed several versions of a simple send and receive program that varied the distance between the sending node and the receiving node. For one set of these programs, the sender sent messages faster than the receiver could receive them. For another set of programs, the sender and receiver operated at the same rate. See Figures 3 and 4 for graphs of the results. From these graphs you can see when the sender was faster than the receiver, the distance between the sending and receiving nodes made more difference. The additional buffer space on the intermediate nodes could store more data so the sender that was further away finished faster, but the slope of the graph is the same regardless of the node distance. Once the queues had initially filled, the sender sent at the same rate regardless of the distance between the sender and the receiver. When the sender and receiver were working at the same speed, the distance between the nodes made a much smaller difference. When the sender and receiver are communicating at the same speed, the messages is not really taking advantage of the buffers added by the longer network. In this case the only change dependent on the network length would be additional time it takes to travel over twice as many links.

The node programs address other nodes to specify the destination when sending data over networks. Nodes are addressed by the row and column numbers of their location in the processor array. After the

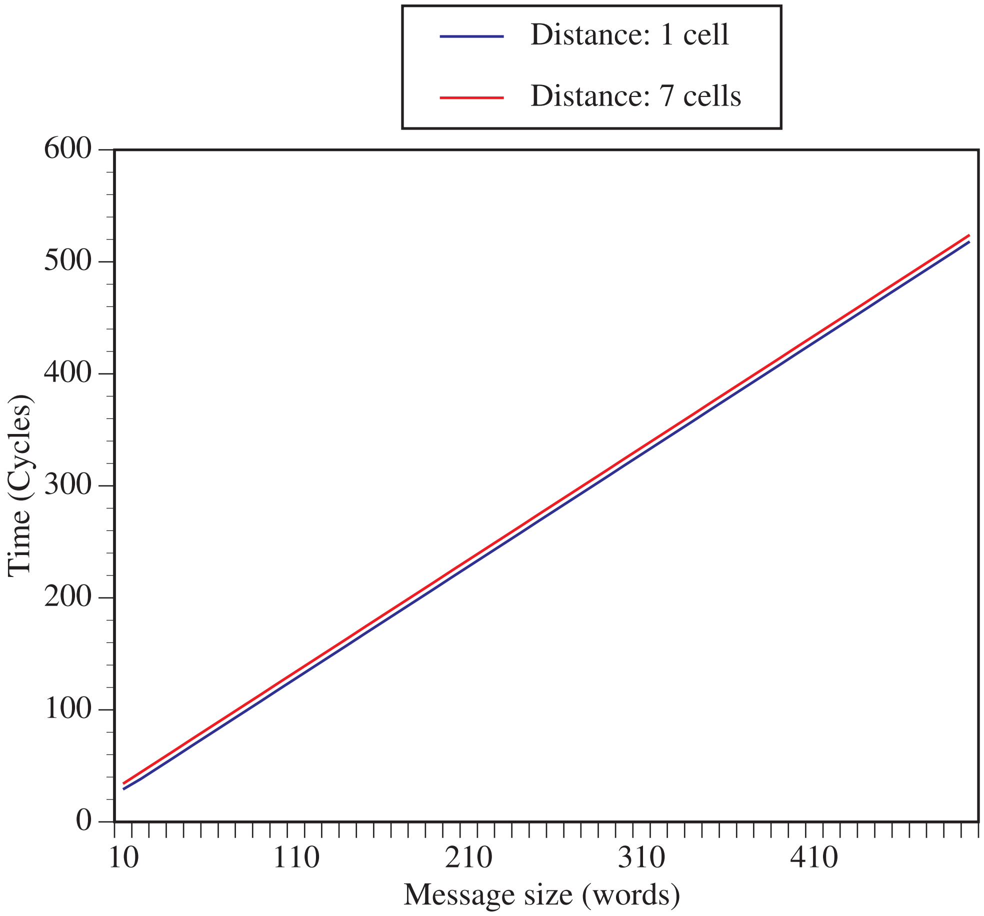

Figure 4: Roundtrip message time when the sender and receiver are operating at the same speed.

user's program has been spread out to insert monitoring nodes, the addresses of destination nodes in the node programs must be adjusted. This address transformation is performed at run time by multiplying the row and column numbers of the destination by 2 (or equivalently shifting these numbers 1 bit to the left), which adds a small overhead to the setup of a connection.

4.2 Sample data accuracy

The monitor program reads the current network queue sizes from 4 control space registers. The length of each queue is stored in four bits, and these bits are striped over the 4 control space registers. Since the queue lengths can change with each cycle, it is the possible for a queue length to change while the control space registers are being sampled. The queue lengths are constrained to change at most by one with each clock tick for single word transfers (queue lengths can change by two per each clock tick for double word transfers), so the percentage of errors is lower than it would be if the queue lengths could change to any value. We have also observed that the queue lengths tend not to change continually with each clock tick, so the odds of sampling during a queue length transition are low.

Over the period the program is reading the control space registers, the length of a queue may vary by 3 for single word transfers. Consider the four number transitions that are allowed on the 0 to 8 counter, e.g. $8 \rightarrow 7 \rightarrow 6 \rightarrow 5$ or $4 \rightarrow 4 \rightarrow 5 \rightarrow 4$. There are 203 valid four number transitions over the range of 0 to 8. The monitor program reads the queue length registers one bit at a time while the actual queue length goes through one of these four number transitions. The queue size read is *inaccurate* if it was not one of the queue length values during the 4 cycles it took to read the queue length regiesters.

The monitor program reads the queue lengths starting with the most significant bit, so the worst inaccuracies occur when the top bit changes in transitions such as $8 \rightarrow 7 \rightarrow * \rightarrow *$ or $7 \rightarrow 8 \rightarrow * \rightarrow *$. Downward transitions from 8 will result in numbers greater than 8. The monitor program truncates such numbers to 8. In this case the queue size is accurate, because 8 one of the queue length values during the time it took to read the queue length register. However, transitions that start from 7 to 8 are bad because the two most significant bits read will be 0 and the resulting value read will be at most 3, at least 4 off from either 8 or 7. The next worst inaccuracies occur in transition such as $* \rightarrow 4 \rightarrow 3 \rightarrow *$ or $* \rightarrow 3 \rightarrow 4 \rightarrow *$, because the the middle two bits will be inaccurately set to 1 or to 0. For transitions of the form $* \rightarrow 4 \rightarrow 3 \rightarrow *$ the resulting number read will be 6 or 7 at least two off. For transitions of the form $* \rightarrow 3 \rightarrow 4 \rightarrow *$ the resulting number will be 0 or 1 at least 2 numbers off.

Without more accurate information about the distribution of queue length changes, we cannot determine the sampling error introduced by misreading queue lengths. [1] However, as stated earlier, for an understanding of inter-processor communication, the state of the queue is more important than the actual number of words in the queue, so the current sampling technique is adequate. Designers of future system may want to chose a different design for the status registers that allows an atomic read of the queue length registers.

5 Using queue length data

After the monitored program has been executed, the programmer is left with large files of numbers. To get any useful information from this data, the programmer needs tools to effectively manipulate and display the queue size data. Much work has been done on displaying monitored data (e.g. Tapestry [Mal90], BEE [Bru90], and others). We have created two relatively simple programs to aid in the post-mortem understanding of the collected data.

The first program **xmon** is an X window program that examines the data files and displays the user's array and input logical channel queues. It steps through the execution of the monitored program and displays the queue lengths at each point in the original execution. Figure 5 shows a screen image of **xmon** in action. In that figure, **xmon** is working on a program that ran on a 4×4 array of nodes. There are two distribution networks snaking through the array, and there are networks between each pair of physically adjacent nodes. Figure 6 shows the logical arrangement of the networks used in that array program. The queues in the **xmon** display are shaded to reflect their length. The display reflects the state of the queues at the point in the execution of the original program shown by the time bar at the bottom of the screen. The time bar is labeled from 0 to the end of the original execution and is filled to the currently displayed execution time (Figure 5 shows the state of queues after the program has executed for 74,080 cycles). The user can step to the next or previous state of the queues. The user can also run in *animate* mode, where **xmon** automatically displays each consecutive state.

By looking at the changes in queue length, the programmer can visualize the flow of data in her

[1] The reason for this striped arrangement of the queue length registers was that it simplified the design of the communication agent (which contains a number of speed-critical paths). In the original processor design, the queue length registers were just testing registers and so did not need to be read quickly.

program. For example, she can see periods where one set of networks is more active than another set or where a set of queues is consistently full. While **xmon** is helpful in visualizing the global data flow, it is not so good for presenting the fine details of the communication patterns. **xmon** only shows *changes* in queue length, so when running in animate mode, the time bar does not move forward smoothly. If the queue lengths did not change for a long time, the time bar will jump ahead to the time of the next queue length change. If there are short alternations of communication and computation, the short jumps in the time bar are easy to miss.

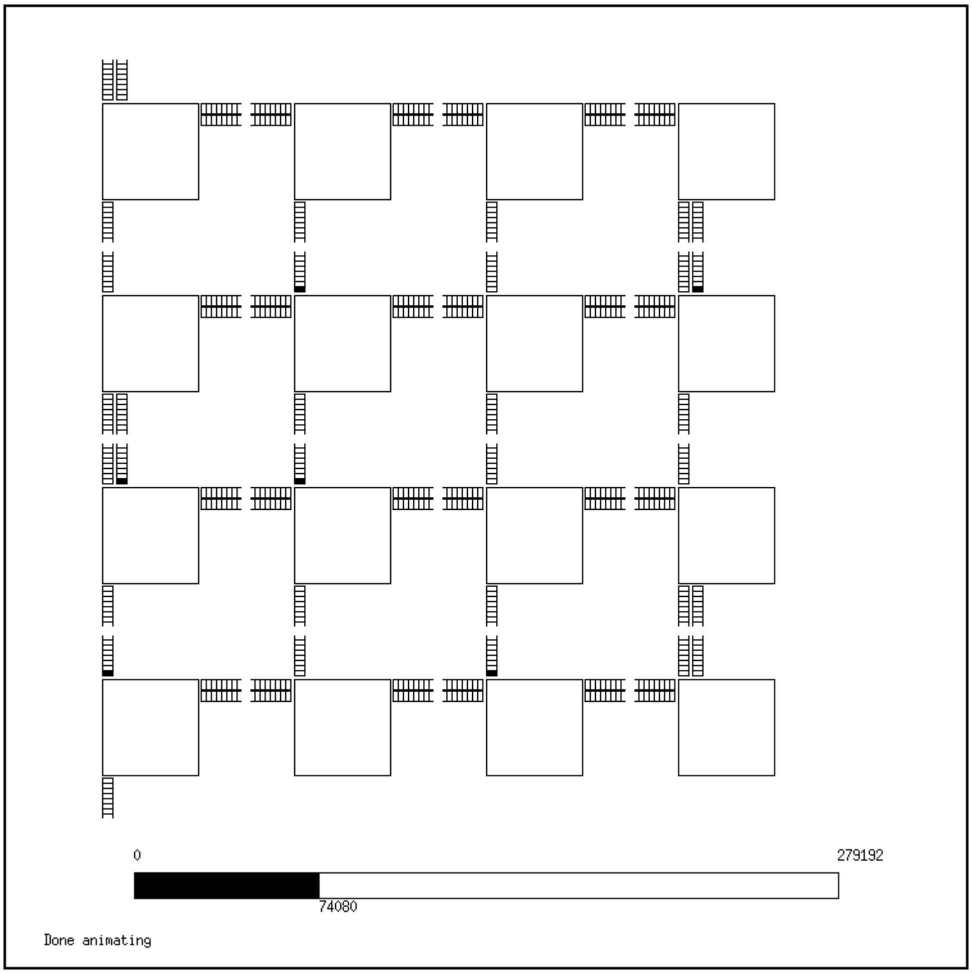

Figure 5: This is a picture of **xmon** displaying the data obtained from running the AL matrix multiply program.

By graphing queue length versus monitored execution time for each node, the programmer can enlarge interesting areas to see the fine details of the communication pattern. We created a tool called **mon-graph** that transforms the monitored data into load and data files for **gnuplot**. By adjusting the X range in the load files, the programmer can create graphs that zoom in on the interesting subsequences. Of course this creates many graphs, but by using a postscript previewer like **ghostscript**, the programmer can quickly examine many different queue length graphs.

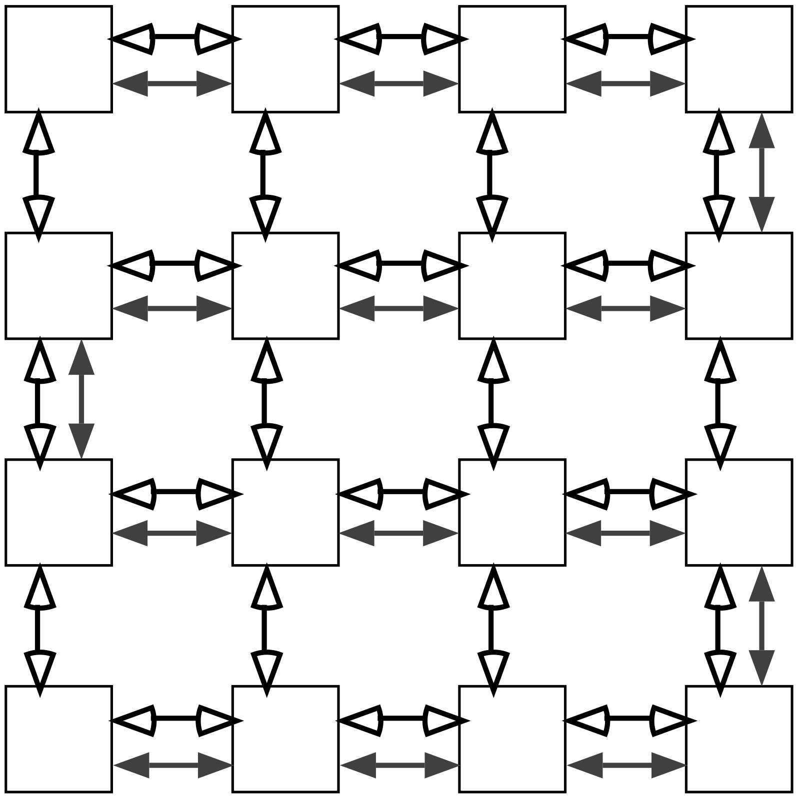

Figure 6: A picture of the logical connections used in the in the AL matrix multiply program.

5.1 Examples

Here we show an example of using **xmon** and **mon-graph** to evaluate the monitored data from two matrix multiply programs. One was an AL program which uses memory communication (see [Tse89] for details). The other program uses systolic communication and was generated by another mapping tool [Rib90]. Both programs run on a 4×4 array of iWarp nodes (using a preliminary version of the runtime system and version 2.3 of the C compiler). Figure 5 shows the queue states of the AL program displayed by **xmon**. By running **xmon** for each program, we can distinguish the main phases of the programs: loading, computing, and closing down.

From **xmon** the communication patterns of the two programs looked similar. In both programs the lengths of the active queues quickly changed between empty and partially full. By using **mon-graph**, we were able to zoom in on the computation phases of the two programs. Figures 7 and 8 show the resulting graphs. In the AL program queues regularly alternated between being empty and oscillating between being empty and having one entry. From knowing the block oriented nature of AL, this is expected. The program alternates between exchanging blocks of data and computing on those blocks. While computing, the program exchanges no data, so the queues stay empty. In the systolic program, communication appears to be interspersed with the computation. The first part of the graph shows the queue has length one most of the time, dropping to zero at regular intervals. The second part shows a queue length of zero most of the time, jumping to one at regular intervals. Since the systolic program sends and receives data in one word intervals as it is computing, this pattern is reasonable. In the first half, the program is consuming

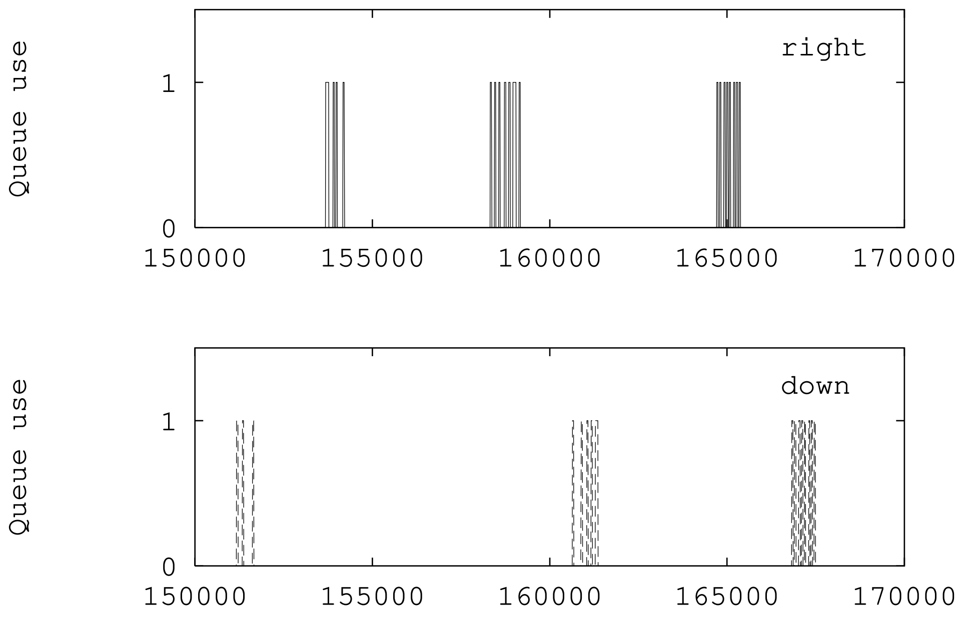

Figure 7: A graph of the queue lengths on node (1,1) over time during the execution of the AL matrix multiply program.

data slower than it is being sent, so the data stays on the queue. In the second half, the program is trying to consume data faster than it is being sent, so data is quickly read out of the input queue.

5.2 The next step

In the previous example, we gathered data on inter-processor communication and saw that there no message was blocked in the communication system. If we had discovered a blockage, the display program would have identified the node(s) that were involved in the communication, but if the nodes exchange multiple messages, the user would have to identify which message was affected. Without data from the user nodes, there is no way to relate the queue sizes and times directly with the user node activity. For some programs, the information from the monitoring nodes alone is sufficient to understand the communication patterns, but for complicated or unfamiliar programs, relating the queue sizes back to instructions in the node program would be very useful. Specifically, we would like to link queue lengths with the execution of user instructions and with the user node state (either active or spinning). One of the challenges that such a tool must overcome is that any more detailed recording of information may perturb the program execution significantly.

6 Conclusions

Our experimental monitor shows that a processor that provides a close coupling between communication system and the computation units can be programmed to serve as a performance monitor. Although this idea is simple, it is quite useful. Using idle nodes to capture information about the inter-processor

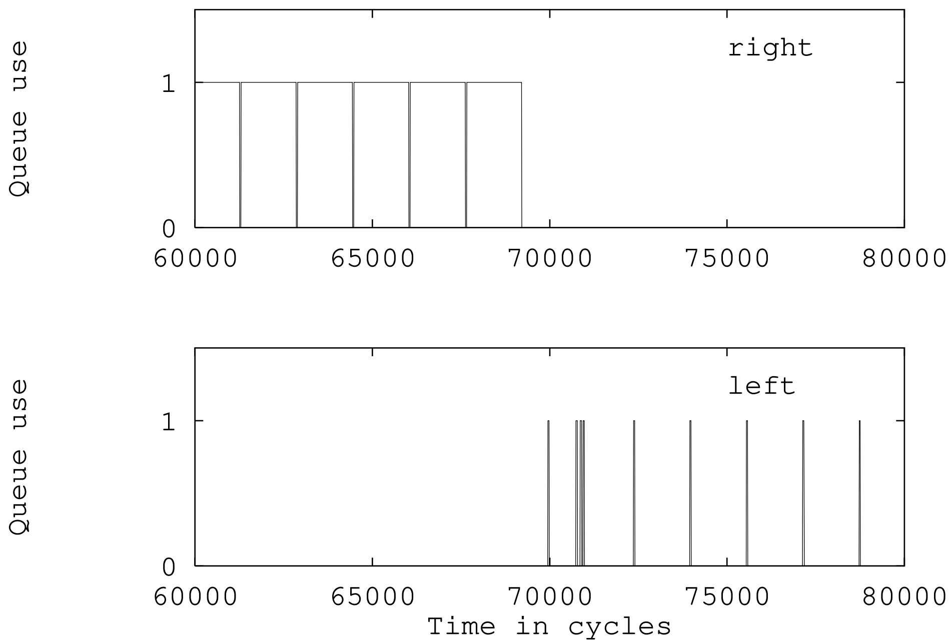

Figure 8: A graph of the queue lengths on node (1,1) over time during the execution of the systolic matrix multiply program.

communication does not require any additional hardware, and when the programmer is not interested in monitoring, all resources in the system are available to execute user programs. The information gathered by such a monitor can be used to identify communication bottlenecks, and building a visualization system is not difficult.

Future designers of integrated components that are to serve as building blocks for parallel systems may want to consider this usage of a processor during the design phase. Although we did not encounter any insurmountable problem in implementing the monitor, there are several rough edges (like the need to read the queue length registers sequentially).

The close coupling between the communication agent and the computation agent of the iWarp component is crucial for the successful operation of the monitoring program. It is not clear if other systems (where communication and computation are not integrated as closely as in iWarp) can build a similar tool as easily. However, the advantages of tightly integrated communication and computation are recognized by other parallel systems, so future systems may provide the foundation for similar hybrid monitoring.

References

[BCC+88] S. Bokar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urban, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, C. Peterson, J. Sussman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp, CMU-CS-90-197. Technical report, Carnegie Mellon University, School of Computer Science, 1990. Revision of a paper that appeared in the 17th Annual Intl. Symposium on Computer Architecture, Seattle, 1990, pp. 70-81.

[BG91] B. Baxter and B. Greer. Apply on iWarp. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.

[Bru90] B. Bruegge. BEE: a Basis for Distributed Event Environments: Reference Manual, CMU-CS-90-180. Technical report, Carnegie Mellon University, School of Computer Science, 1990.

[FGOS92] A. Feldmann, T. Gross, D. O'Hallaron, and T. Stricker. Subset Barrier Synchronization on a Private Memory Parallel System. In *Proc. Symposium on Parallel Algorithms and Architectures*, San Diego, June 1992. ACM.

[Hin91] S. Hinrichs. *Programmed Communcation Service Tool Chain User's Guide*, 1991.

[HWW89] L. G. C. Hamey, J. A. Webb, and I. C. Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*, 48:246–264, 1989.

[Mal90] A. Maloney. *Performance Observability*. PhD thesis, University of Illinois, Urbana-Champaign, 1990.

[O'H91] D. O'Hallaron. The Assign Parallel Program Generator. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.

[Rib90] H. Ribas. *Automatic Generation of Systolic Programs from Nested Loops*. PhD thesis, Carnegie Mellon University, 1990.

[SK90] M. Simmons and R. Koskela, editors. *Performance Instrumentation and Visualization*. ACM Press, New York, 1990.

[Tse89] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989.