Contention and Queueing in an Experimental Multicomputer: Analytical and Simulation-based Results

Wenjia Fang*, Edward W. Felten*, and Margaret Martonosi†

* Department of Computer Science, Princeton University
† Department of Electrical Engineering, Princeton University

Computer Science Technical Report TR-508-96

Abstract

We use two methods, analytic modeling and event-driven simulation, to evaluate the performance of communication in the XYZ[1] multicomputer under a variety of workloads. This work has two purposes: to learn about the behavior of the XYZ machine, and to explore the tradeoffs between analytic modeling and simulation as performance prediction techniques.

We find that the system's performance depends mostly on the behavior of the endpoints; most of the message latency is spent arbitrating for the sender's and receiver's memory buses, and doing DMA transfers of data to and from memory. The raw speed of the network is relatively unimportant. Performance is mainly determined by details of DMA speed, network connectivity, and policies for arbitrating for buses and network links.

Even for very simple synthetic benchmark programs, the analytic model and the simulator differ by roughly 10% in their performance predictions. Although we believe the simulator to be more accurate than the model, each method has difficulty accurately representing some feature of the real system. Finally, we observe that each of the two tools was valuable in helping to improve the accuracy of the other.

Keywords: Message passing, Interconnection networks, Arbitration policies, Performance modelling techniques.

1 Introduction

Modern parallel machines have elaborate communication mechanisms, and the performance of these mechanisms is often very difficult to predict, even under simple workloads. Complex interactions between network interface buffering, network congestion, bus arbitration policies, and cache coherence hardware make performance prediction challenging. The cost and engineering effort required to build a parallel machine make performance prediction important.

This paper presents and compares the results of two performance prediction methods, analytic modeling and event-driven simulation, for predicting the performance of the XYZ multicomputer [1]. Using both methods, we study the behavior of the system under different communication patterns and report on system performance parameters such as message latency, occupancy of system buffers, and network congestion.

The contributions of this work are two-fold. First, we present evaluations and analysis of how the real XYZ system behaves. Our models allow us to predict message latencies for the current

[1] Machine name changed for anonymous reviewing.

design under varying usages; we can also examine how program behavior will change as various buffer sizes and bus speeds are modified. Second, we examine tradeoffs between analytic and simulation-based modeling techniques. We point out when analytic modeling and simulation make similar predictions and when they predict different behavior. We also explain why these differences arise and how they can be corrected.

The remainder of this paper is organized as follows. In Section 2 we give background information on the architecture of the XYZ Multicomputer. Following this, Section 3 describes the modelling and simulation methodology we use for evaluating its performance. Section 4 presents initial results for the system's steady-state behavior when the communication destinations are randomly spread over all nodes. We follow up on this in Section 5 by presenting results on many-to-one communication patterns and on system behavior during bursts of messages. Section 7 discusses related work as well as some of the insights gained from the study, and Section 8 offers conclusions.

2 Background: The XYZ Multicomputer

The XYZ project studies how to provide high-performance communication mechanisms in order to integrate commodity desktop computers such as PCs and workstations into inexpensive, high-performance multicomputers [2, 1]. Our goal is to provide a low-latency, high-bandwidth communication mechanism whose performance is competitive with or better than those used in specially designed multicomputers.

Figure 1 shows a single XYZ node, with a block diagram of its main components. The XYZ prototype system uses commodity PC systems as nodes and a multicomputer routing backplane as its interconnection network. The current XYZ nodes are unmodified 66MHz Pentium PC systems, each configured with disk and standard commodity devices such as tape drives, monitors, keyboards and LAN adaptors. The current network is the Intel Paragon mesh routing backplane [9]. The connection between a network interface and the routing backplane is via a simple signal-conditioning card and a cable.

A main focus of the design work has been on the network interface. The XYZ network interface implements virtual memory mapped communication to support protected, user-level message passing, and fine-grained remote updates and synchronization for shared virtual memory systems. XYZ supports a variety of programming styles by supporting communication through either *deliberate update* or *automatic update*. Under deliberate update, values written by one processor are not propagated until that processor issues an explicit send command. Deliberate update allows user-to-user block transfers with no operating system intervention. Under automatic update, a sending process can map memory within its address space as a send buffer; any time the sending process writes to one of these mapped (outgoing) memory regions, the writes are propagated automatically to the virtual memory of the destination process to which it is mapped.

The software interface to this functionality is defined by the XYZ Basic Library (XBL) [3]. For example, the library provides calls to allow for the memory imports and exports that are key to XYZ's virtual memory-mapped communication. One XBL daemon runs on each compute node, doing the bookkeeping and verification related to setting up communication paths between nodes. Thus parallel programs written using XBL message passing calls can be run directly. In addition, we also have a version of Intel's NX message passing library [7] that translates down into XBL calls. Using this library, Intel NX-based message passing programs may also be run through the simulator.

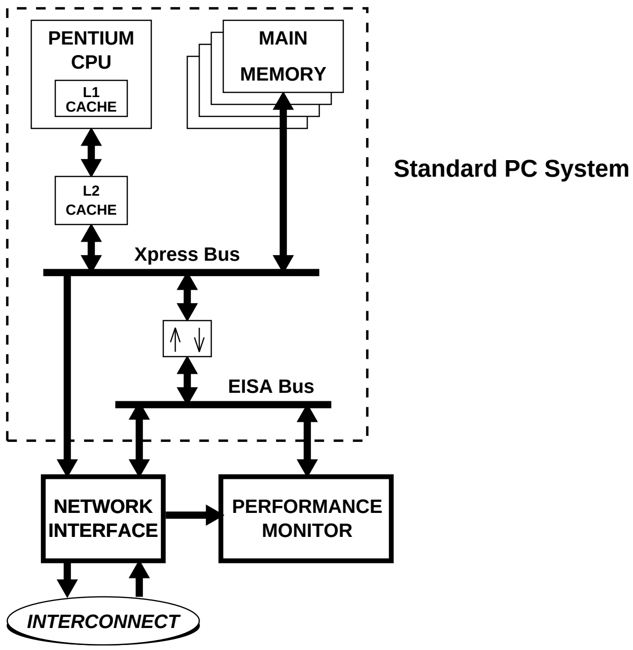


Figure 1: Block diagram of a compute node in the XYZ prototype.

Figure 2 shows the major components in XYZ's communication system, and the path taken by outgoing and incoming message data. In order to send a message via deliberate update, the user process makes a user-level call to the XYZ Basic Library. This call specifies the size and starting position of the data to be transferred. If the Outgoing Direct Memory Access (DMA, i.e. message sending) engine is already busy, the message send must retry later. When the Outgoing DMA is no longer busy, the message send will successfully initiate a DMA. This will transfer data from the node's memory, over the Xpress and EISA bus, and into the FIFO on the Network Interface board (referred to as the Staging FIFO). Since the EISA bus can't be held for more than 64 bus cycles, messages longer than 256 bytes (64 words) will be broken into 256-byte chunks as they cross the EISA bus into the Staging FIFO. Each chunk will traverse the network as a separate packet.

From the Staging FIFO, the outgoing data passes into the Intel Network Interface Chip (NIC). It travels over the 320 MBytes/s NIC bus to the outgoing NIC FIFO. It then contends for the port to the mesh-connected network, travels over the network, and then contends for the port back into the destination node.

On the incoming side, the data is collected in the incoming NIC FIFO. When the amount of data in the incoming NIC FIFO reaches a specified threshold (or when the packet trailer is received), the incoming DMA engine arbitrates for both the EISA bus and the NIC bus. When it has both, it DMAs the data into memory. As on the outgoing side, the EISA bus specifications limit bus ownership to 64 cycles, so in longer messages, the incoming FIFO will periodically give up all the buses and then rearbitrate for them. The EISA bus is shared fairly by the CPU (for reads), the CPU's write buffer, the outgoing DMA engine, and the incoming DMA engine. The NIC bus is used by both incoming and outgoing message data, and the incoming DMA engine is given priority on it.

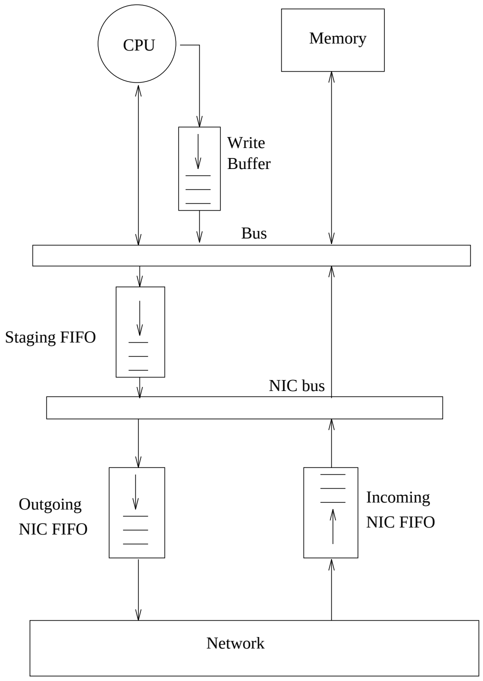


Figure 2: Contention and queueing points in XYZ communication.

3 Evaluation Methodology

Our goal in this study was to develop both an analytic model and a simulation-based model in order to facilitate a broad range of experiments. The analytic model is useful for quick insights into the trends of SHRIMP performance, and allows one to investigate a wide variety of implementation tradeoffs (such as FIFO sizes and bus speeds) quickly. The simulation based model is useful for understanding SHRIMP behavior for specific applications and for aspects of performance that are difficult to model analytically. In this section, we will describe the two approaches, and then briefly compare the main differences.

3.1 Analytic Models

Our analytic model makes two standard assumptions:

- packet interarrival times and service times at every component are exponentially distributed, and
- the states of any pair of components are independent random variables.

While these assumptions do not match the way the system really operates, we choose to make them because without them we are unable to solve the analytic model. Some advocates of analytic performance modeling have claimed that these assumptions are generally acceptable because the resulting models, although approximations, tend to accurately predict qualitative behavior of the system being modeled. Evaluating these claims is one of the purposes of this paper.

We will describe here a model that assumes that all processors execute the same program, which alternates between computing and sending, that all messages are the same size, and that every message is sent to a randomly (and uniformly) chosen destination. Later in the paper, we will present results from this model and from some simple variants of it. For brevity, we will describe only the basic model, and will omit the details of the variant models.

We treat each packet as an indivisible unit which moves as a single entity through the system. This is an approximation, since the real hardware breaks messages into packets of 256 bytes, and allows a packet to move through multiple stages of the system in a pipelined fashion.

Our basic approach is to model each component of the system separately. Each sub-model is relatively simple; all but one of the sub-models can be solved analytically. Each sub-model is subject to a set of boundary parameters, which describe the rates at which the environment offers packets to the sub-model and accepts packets from the sub-model. The boundary parameters for a sub-model can be calculated from the states of all the other sub-models.

Solving the entire model thus corresponds to solving a coupled set of non-linear equations. Given a set of boundary parameters, we can solve each sub-model, which allows us to calculate a new set of boundary parameters. A fixed point of this cycle is a solution to the overall model. We find such a fixed point by using a simple iterative scheme.

3.1.1 Submodels

We use five submodels: the application program, the SHRIMP FIFO, the Outgoing NIC FIFO, the network, and the Incoming NIC FIFO. The three FIFO sub-models are variants on the same basic FIFO model, with different parameters. We now describe the individual sub-models.

The FIFO Model We model a FIFO as a Markov process with states numbered 0 through C, where C is the capacity of the FIFO (the maximum number of packets it can hold). The FIFO model has two boundary parameters: r_{in}, the rate at which the environment offers packets to the FIFO's input end, and r_{out}, the rate at which the environment accepts packets from the FIFO's output end. If p_i is the probability of being in state i, the FIFO obeys these equations:

$$p_0 r_{in} = p_1 r_{out} \tag{1}$$

$$p_{i-1} r_{in} + p_{i+1} r_{out} = p_i (r_{in} + r_{out}), \text{ for all } \ 0 < i < C \tag{2}$$

$$p_{C-1} r_{in} = p_C r_{out} \tag{3}$$

Solving these equations, we get

$$p_i = \frac{r^i(1-r)}{1-r^{C+1}}, \tag{4}$$

where $r = \frac{r_{in}}{r_{out}}$.

The Application Program Model We assume an application program that alternates between local computation and sending a message. We model this as a state machine with two states: LocalCompute and Send. The rates of transition between the two states are computed from a set of application parameters, which describe things like how much computation is done between send operations, and from the boundary parameters of the application program model.

The boundary parameters of this model include: the success of the application at arbitrating for the bus, the rate at which outgoing DMA can transfer data, and the fraction of time during which the SHRIMP flow control mechanism prevents the application from computing.

The Network Model The network model is by far the most complex of the sub-models. It has two boundary parameters: the probability that an out-fifo is offering a packet, and the probability that an in-fifo is willing to accept a packet. Since all nodes behave according to the same probability distribution, the same probability applies to all out-fifos and all in-fifos.

The network model's prediction is computed in two stages. First, we determine the expected aggregate throughput $\tau(i, j)$ of the network, assuming that exactly i out-fifos are offering packets and exactly j in-fifos are willing to accept packets, for all values of i and j. This stage's computation depends on the specific network topology we are assuming, but it does not depend on the boundary parameters, so its result can be precomputed once for each network topology.

Our system uses a wormhole network, which is difficult to model analytically. We therefore compute the throughput $\tau(i, j)$ by a simple Monte Carlo method. The method generates a random set of i enabled senders (out-fifos), with a random message destination for each, and a random set of j enabled receivers (in-fifos). It then considers the senders in a random order, and allows each in turn to reserve as many network links as it can along its route, counting how many messages successfully reach their destinations. This random "experiment" is repeated many times and the mean number of successful messages is computed. The experiment is repeated enough times so that the computed mean is accurate to within 0.1% with 99% confidence. This procedure takes a long time, but it is simple and accurate and it need only be done once.

The second stage depends on the result of the first stage, and on the values of the boundary parameters. In this stage, we compute the apparent bandwidth of the network as seen by each out-fifo, and by each in-fifo. The apparent bandwidth of, for example, an out-fifo is the rate at which the network accepts packets from that out-fifo, given that that out-fifo is offering a packet.

The apparent network bandwidth seen by an out-fifo is calculated by considering the possible states of the other $N - 1$ out-fifos and the $N - 1$ in-fifos, and summing the product of the probability of a state and the throughput seen by our out-fifo in that state. This is calculated as

$$b_{out}(p_{out}, p_{in}) = \sum_{n_o=0}^{N-1} \sum_{n_i=0}^{N} \binom{N-1}{n_o} p_{out}^{n_o} (1 - p_{out})^{N-1-n_o} \binom{N}{n_i} p_{in}^{n_i} (1 - p_{in})^{N-n_i} \frac{\tau(n_o + 1, n_i)}{n_o + 1}, \tag{5}$$

where p_{out} is the probability that an out-fifo is offering a packet, and p_{in} is the probability that an in-fifo is willing to accept a packet. The apparent bandwidth of an in-fifo is calculated by a similar formula:

$$b_{in}(p_{out}, p_{in}) = \sum_{n_o=0}^{N} \sum_{n_i=0}^{N-1} \binom{N}{n_o} p_{out}^{n_o} (1 - p_{out})^{N-n_o} \binom{N-1}{n_i} p_{in}^{n_i} (1 - p_{in})^{N-1-n_i} \frac{\tau(n_o, n_i + 1)}{n_i + 1}, \tag{6}$$

3.1.2 Arbitration

The final component of our analytic model deals with arbitration for the system bus and the NIC bus. Our arbitration model calculates, based on the arbitration policy of the modeled system, and the states of the sub-models which contend for each resource, what fraction of the time each component of the system succeeds in arbitrating for its output resource. These results are used to calculate the amount of data flowing across each bus in each direction; these values are used in turn to calculate the boundary parameters for various sub-models.

3.1.3 Latency Calculation

In addition to the states of the buffers, network and buses, we also want to predict the communication latency: how long it takes a packet to traverse the system from sender's memory to receiver's memory. Each submodel computes its own contribution to the latency. For example, the FIFO submodel's latency contribution is equal to the FIFO's average occupancy after a packet arrives, divided by the effective rate at which the FIFO drains. Devices which do not queue data, such as the network, do not compute a latency contribution themselves. Network congestion makes itself felt because it affects the effective drain rate of the out-fifo.

The method described above would be adequate if packets passed through a FIFO in a store-and-forward fashion, that is, if the complete packet had to arrive in a FIFO before it could start leaving the FIFO. In reality, however, the out-fifo and the in-fifo allow the head of a packet to move through to the next stage without waiting for the tail of the packet to arrive. Our latency calculation corrects for this effect by subtracting from the latency the time for one packet to drain out of each such component.

3.2 Simulation Framework

Having presented the basic analytic model used in these studies, we will now describe the SHRIMP simulator. The SHRIMP simulator is built using the Mint simulation package [10]. Mint supports program-driven execution. In it, interesting events in a parallel program (such as memory references and synchronization) are sent to a simulator "back-end", that simulates the timing behavior of the machine being studied. In the SHRIMP simulator, independent asynchronous units in the system, such as FIFO's and buses, are simulated by independently scheduling events such as FIFO dequeue operations and bus arbitrations. The Mint package manages the fine-grain interleaving of these events, to produce an event ordering that would be legal and realistic on the machine being simulated.

We model the system as shown in Figure 3. The main simulator components are shown in the box on the right hand side. They can be divided into two parts. One part, labelled Timing Simulation, forms the core of the SHRIMP machine model. It takes machine events as inputs and simulates their timing and contention effects on the machine's memory hierarchy, buses, and FIFO's.

The input events simulated come from two sources. One set of events are the instructions coming directly from the application program itself. This allows for detailed simulation of instruction times. Using the stream of reads and writes, we also simulate the memory hierarchy. The second set of input events are produced by message passing calls in the application program, and come via the module labelled 'Daemon Simulation'. This Daemon Simulation module simulates the functionality of the Shrimp Basic Library (SBL) Daemon, described in Section 2.

3.3 Summary and Comparisons

It is important to note several key ways in which the simulator and the analytic model differ. First, the analytic model does not take into account the 64-cycle limit on EISA bus transactions. Essentially, it models a fresh arbitration on each transferred word. The simulator, however, accounts for long transfers using up to 64 cycles on the bus before rearbitration is needed. In Section 4 we will discuss the impact this has on message latency estimates. Second, the analytic model assumes that the states of individual entities are independent random variables.

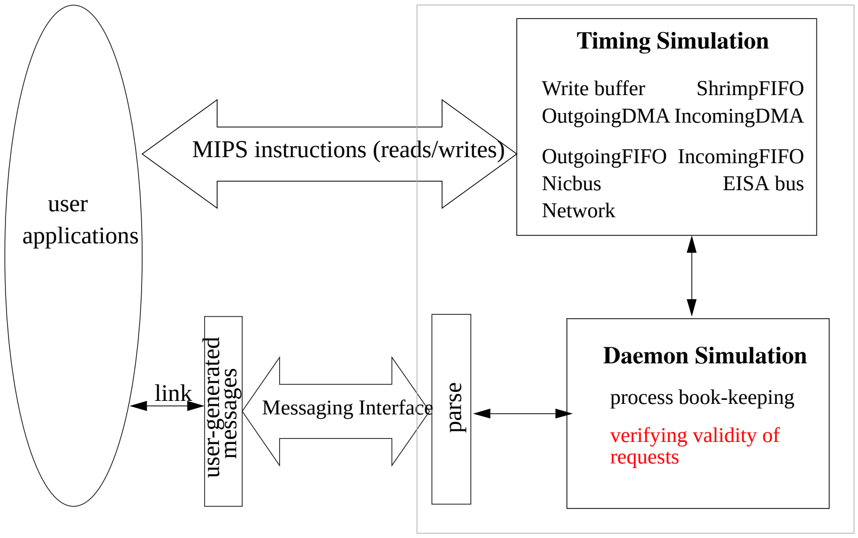


Figure 3: An overview of the components of the SHRIMP simulator.

Third, the analytic model assumes exponentially-distributed service times for components such as bus latencies. The simulator, on the other hand, assumes fixed latencies, or latencies per unit transferred.

Parameter	Value
EISA Bus Speed	29.3 MBytes/s
Staging FIFO Size	32768 bytes
Outgoing NIC FIFO Size	2048 bytes
NIC Bus Speed	320 MBytes/s
Network Speed	36 MBytes/s
Incoming NIC FIFO Size	2048 bytes

Table 1: FIFO Sizes and Bus Speeds in SHRIMP.

Finally, note that unless otherwise specified, our experiments all use default system parameters that reflect the actual SHRIMP implementation. These values are given in Table 1.

4 Steady-State Behavior with Random Message Destinations

We begin by evaluating XYZ's queueing and contention with both simulator and analytic model. For all of the results presented in this section, we use the default XYZ FIFO sizes and bus speeds as given in Table 1. All of our calculations and measurements in this section are for the system

```
for (i = 0; i < NUMPROCS;  i++) {
   Export a receive buffer for each other node to send to.
   Import a receive buffer to send to at each other node.
}

Build a message body to send out repeatedly during run.

for (j =0; j < NUMMESGS; j++) {
    recvId = RandomReceiver();
    errc = SendMessage(sourceBuf, size, recvId, recvBuf);

    /* Delay between messages by doing some computation */
    for (i=0; i < RandomDelay[myPid][j]; i++) {
        x = reg;
        x++;
    }
}
```

Figure 4: Pseudo-code executed per-node in the synthetic benchmark.

at steady-state.

In these baseline experiments, each processor sends out an equal number of messages, and the message destinations are uniformly distributed over all nodes. A send rate of 10MBytes/s means that the program sends 10 Megabytes of data for every second it spend in local (non-communicating) computation. The time spent in computation (between messages) follows an exponential distribution. We vary the mean of this distribution to achieve send rates that range from 1 Mbytes/s up to 20 Mbytes/s. Thus, the results presented here range from light to fairly heavy message passing traffic.

The analytic model is parameterized to support such a setup. To gather comparable results from our simulator model, we drive the model with a synthetic benchmark program. The pseudo-code executed by each node in the synthetic benchmark is shown in Figure 4.

Essentially, each node loops repeatedly, first sending a message to a random receiver, and then delaying for a exponentially distributed amount of time. (Note that the program never explicitly receives a message, but messages in XYZ are automatically DMA'd to the user's memory. Thus, the node still sees the full queueing and contention resulting from a complete message delivery.) For the experiments presented here, each node sends a total of 1000 messages.

4.1 Communication Performance with Varying Message Send Bandwidth

In this initial data, we are using 256 byte messages, which should just fit into the 64 EISA cycles granted in a single EISA arbitration. We examine several figures of merit, including the the message latencies, and the FIFO occupancies.

4.1.1 Message Latency

The left hand side of Figure 5 shows a plot of latency versus send rate. As the message send bandwidth is increased from 1 MBytes/s up to 20 MBytes/s, the message latency increases significantly. The solid line shows average message latencies as predicted by the analytic model. The dashed line represents the message latencies predicted by the XYZ simulator over the same range.

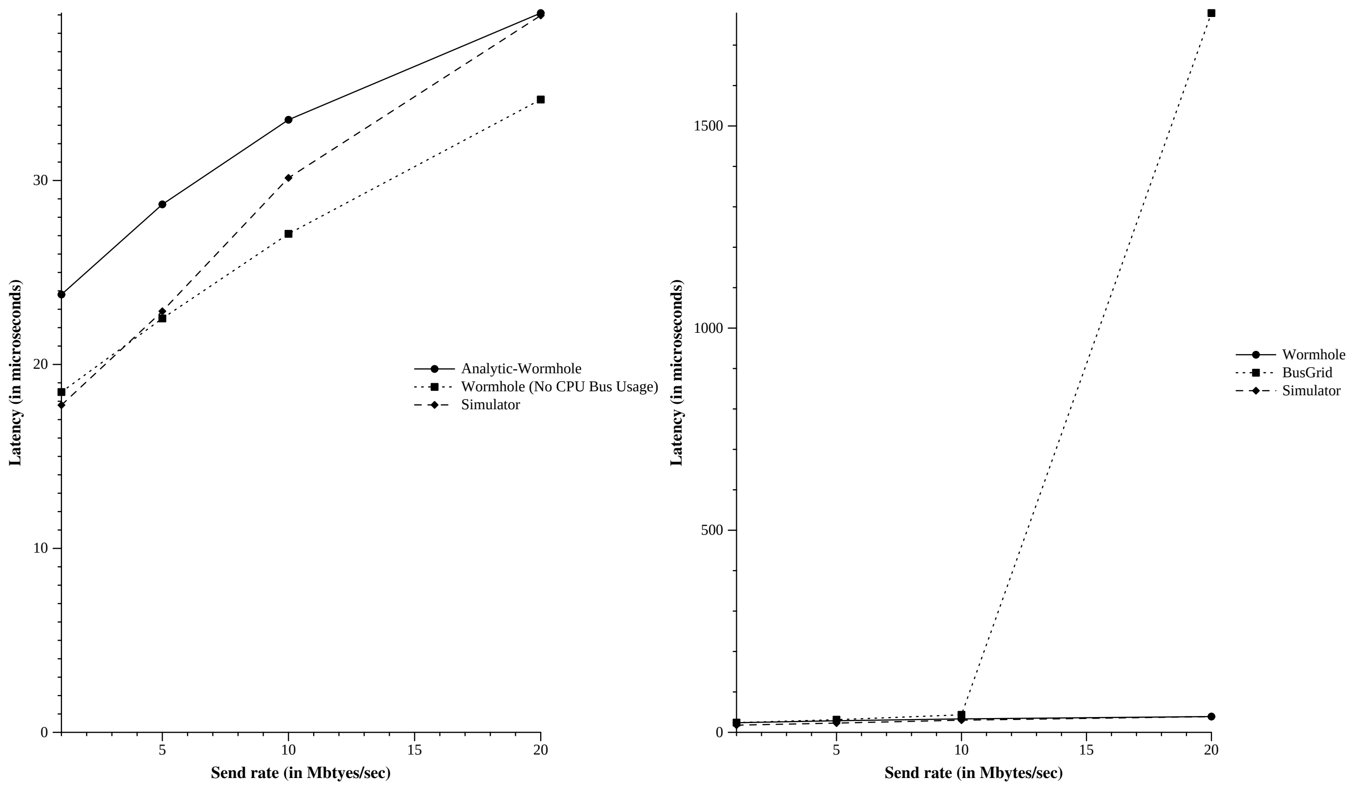


Figure 5: Latency vs. Send Rate for 256 byte messages.

Qualitatively the figures predicted by both methods are within the same range. Furthermore for the minimum send rate, the latency values approach the expected value of roughly 18 μs. When the system is uncongested, the EISA bus transfer time accounts for virtually all of the message passing overhead; for 256-byte packets, the delay for an EISA transfer is roughly 8.7 seconds, and a packet incurs such delays at both the sender and the receiver.

Across the values measured, there is roughly a 20% discrepancy between the simulator and analytic model. This difference can be attributed largely to the differences in bus arbitration assumed by the two methods. In these experiments, bus contention occurs between outgoing message data, incoming message data, and CPU writes. The analytic model treats bus operations as if rearbitration occurred for every word transferred. In the simulator, however, once a transfer acquires the EISA bus, it can hold it for multiple cycles, up to the 64-cycle limit. This essentially favors longer transfers across the bus, since a long transfer can keep a CPU data transfer, which nominally has higher priority, from using the bus.

Interestingly, although the traffic in this scenario is never overwhelmingly large, this baseline uniform-senders experiment was useful in pointing out the limitations of a previous, simpler analytic model for the network. The right hand side of Figure 5 plots the average message latency calculated by this simpler "bus grid" model (prominent dotted line) and compares it to the analytic-wormhole model and the simulator latencies. The bus grid model assumes that on the two-dimensional, mesh-connected network, an entire row or column of the mesh is arbitrated for (and held) as a unit. This means that relatively low amounts of traffic can lead to large network contention. This causes messages to back up into the upstream FIFOs, leading to large latency estimates for the outgoing Staging and NIC FIFOs. For the 20 MBytes/s data point with the bus-grid model, this effect is especially dramatic; the model predicts an average latency of over 1770 μs. In contrast, the lower, dashed and solid lines (overlapping) show the the average message latencies as predicted by the XYZ simulator and analytic wormhole model. (This is the same data as on the left hand side, just plotted to a different scale.)

4.1.2 Buffer Occupancy

One of the primary goals in developing both the analytic model and the simulator was to be able to evaluate design tradeoffs and their impact on required buffer sizes. Here, we present data on buffer occupancy, once again for this default scenario of 256 byte messages, sent out at an average rates 20 MBytes/s.

Figure 6 shows occupancy data collected for the three system FIFO buffers under this configuration. In each graph, the curve represents the occupancy predicted by the analytic model. The histogrammed data represents the occupancy of each FIFO *as observed on the first attempt of each transaction to place data in the FIFO* from the simulator.

The model and simulator results are remarkably similar. The main difference seems to be that the model's prediction is a smooth exponential curve, which the simulator results reflect the quantization of data into packets.

4.2 Communication Performance with Varying Packet Size

Figure 7 shows latency figures as we vary the size of the packet. In the plot, the send rate is held constant at 10 MBytes/s, but the data is sent out in messages of varying size. As in previous graphs, the data from the simulator is plotted with a dotted line, and the data from the analytic-wormhole model is plotted with a solid line. Overall, the analytic model and simulator agree to within 20% at small latencies, and are relatively closer at higher latencies.

The most prominent feature in the plot is the sharp kink in the line for the simulator data between 256 byte messages and 260 byte messages. This discontinuity occurs because a 256 byte message will just fit into a single EISA arbitration cycle, and will be sent through the network as a single packet. The 260 byte message will require two separate EISA arbitrations, and so it will travel across the network as two independent packets.

4.3 Behavior at High Send Rate

Figure 8 shows the message latency as the send rate becomes very large, as predicted by the analytic model. (The synthetic program we use to drive the simulator cannot achieve very high send rates.) The graph shows that latency increases with send rate, but the latency appears to flatten out as the send rate gets above about 100 MB/sec.

This effect occurs because of a natural "self-throttling" effect. As the send rate increases, packets are injected more rapidly into the network, so incoming FIFOs begin to fill. Because the system's arbitration policy gives incoming DMA higher priority than outgoing DMA, the presence of incoming data slows down the sending of packets. The system regulates itself so that

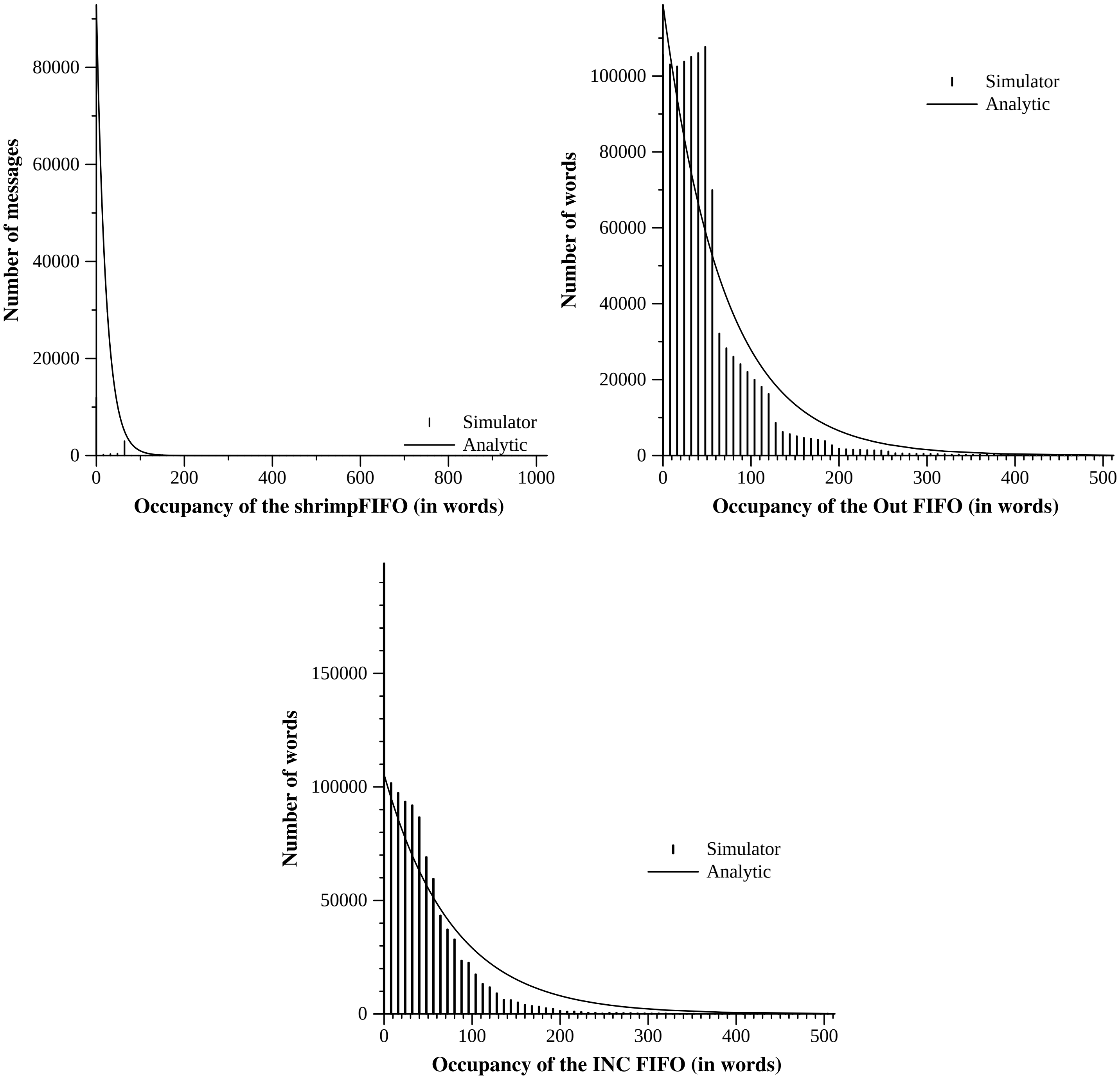


Figure 6: Buffer Occupancy for 256 byte messages at 20MBytes/s send rate.

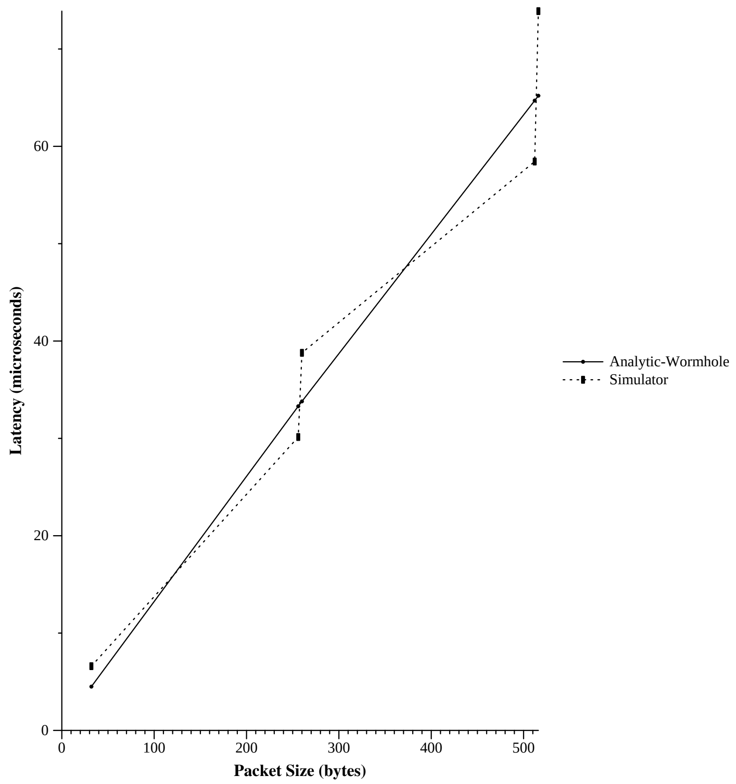


Figure 7: Latency vs. Packet Size, for 10 MBytes/s send rate and default XYZ hardware parameters.

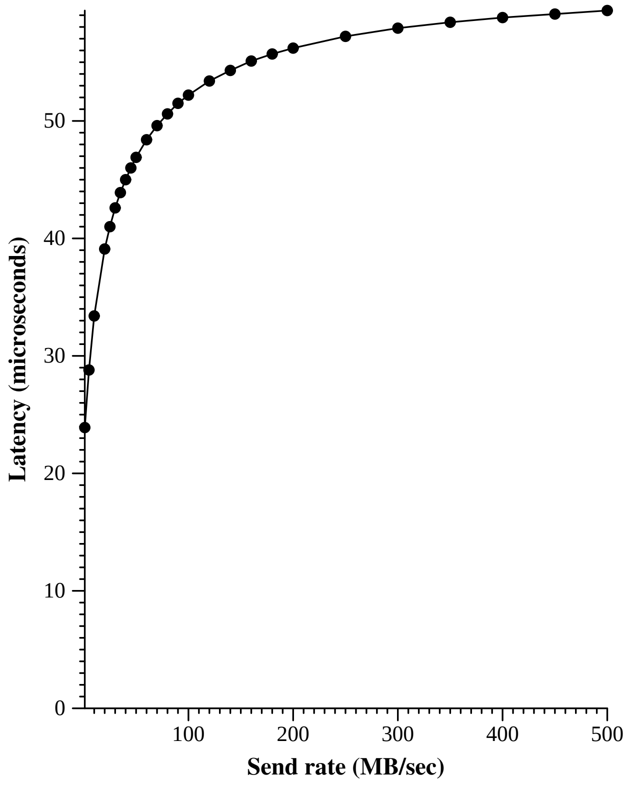


Figure 8: Latency vs. Send Rate, for 256-byte messages and default XYZ hardware parameters, as predicted by the analytic model.

a limited number of packets can be in flight at any time; in effect, congestion control happens automatically.

4.4 Effect of Arbitration Policy

The self-throttling effect demonstrates that resource arbitration policies can have an important effect on performance. To further explore this issue, we now consider the effect on performance of changing the arbitration policy. The experiments in this section use the analytic model only, because it is much more difficult to change the arbitration policy in the simulator.

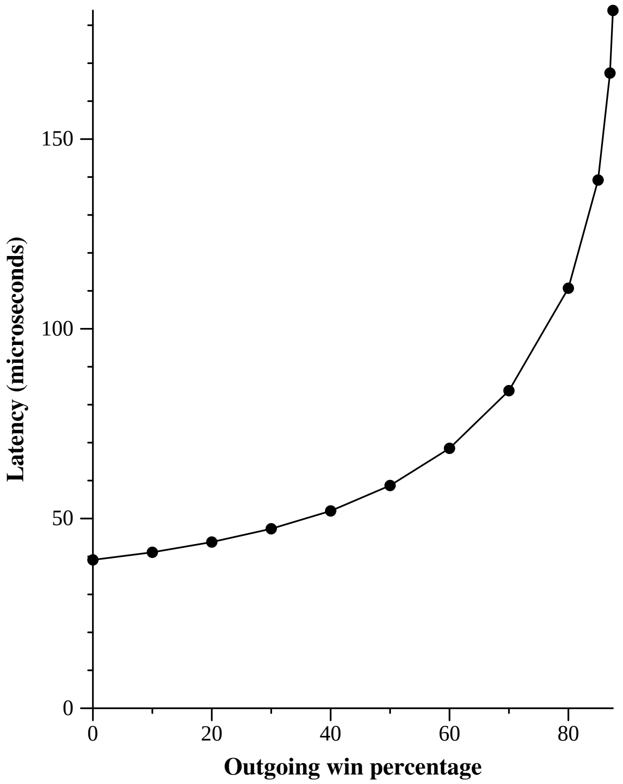


Figure 9: Latency vs. Send Rate, for 256-byte messages and default XYZ hardware parameters, as the bus arbitration policy is varied. The X axis is the percentage of the time, given that both incoming DMA and outgoing DMA are contending, that outgoing DMA gets the bus. The real system operates at the left end of the curve.

Incoming DMA vs. Outgoing DMA Figure 9 shows what happens as we vary the arbitration policy between incoming DMA and outgoing DMA. The real system operates at the left end of the curve, where incoming DMA has absolute priority of outgoing DMA. As we allow outgoing DMA a higher fraction of the bus cycles, latency increases, because the self-throttling effect is defeated. Packets are injected into the network faster, and are drained from the network more slowly, so the average number of packets in flight increases, thus increasing latency. When outgoing DMA wins more than 85% of the arbitrations, the latency becomes very large (several milliseconds).

Another advantage of giving priority to incoming traffic is that there is much less buffering on the incoming side, so the incoming path is more prone to blocking than the outgoing side. Thus it is more important to service the incoming side promptly.

We have also experimented with adaptive arbitration policies, though the results are not shown here. Briefly, we found that the existing policy of giving incoming absolute priority over outgoing has the best performance.

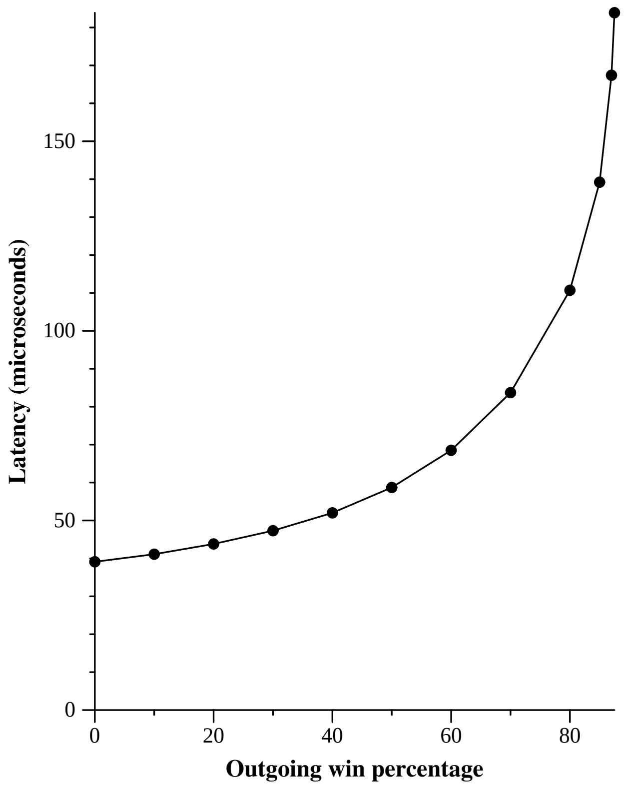


Figure 10: Latency vs. Send Rate, for 256-byte messages and default XYZ hardware parameters, as the bus arbitration policy is varied. The X axis is the percentage of the time, given that both incoming DMA and a CPU memory transfer are contending, that incoming DMA gets the bus. The real system operates at the left end of the curve.

Incoming DMA vs. CPU Memory Traffic The other arbitration choice is between incoming DMA transfers and CPU memory traffic. (The CPU generally spins on a cached value during outgoing DMA transfer, so outgoing DMA vs. CPU memory traffic arbitations do not happen.) In the real system, CPU traffic has higher priority. This is not within our control, as it is a property of the PC systems we are using as computing nodes. However, it is still interesting to consider what would happen if we could change it.

Figure 10 shows the message latency as this component of arbitration policy is changed. As the success rate of incoming DMA is increased, latency decreases, since arriving messages are transferred into memory more quickly, and so spend less time in flight. The effect of incoming DMA vs. CPU arbitration is much less important than the effect of incoming DMA vs. outgoing DMA arbitration.

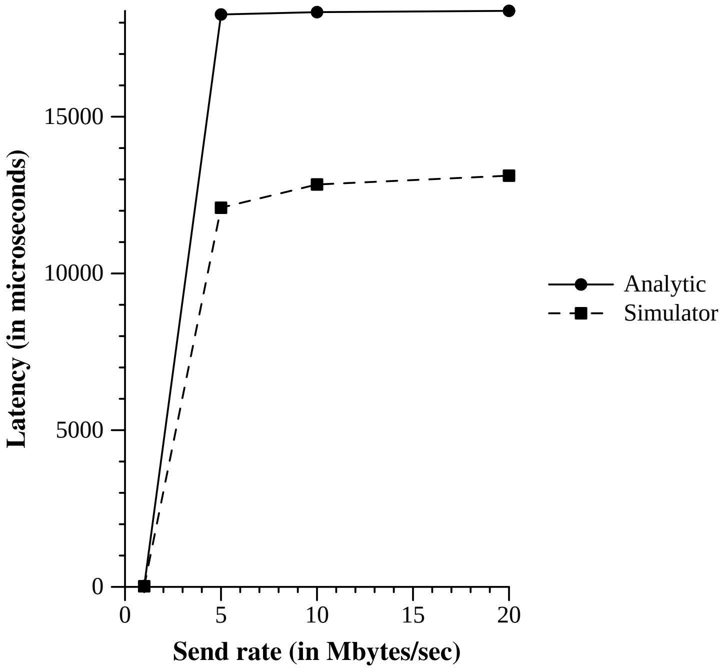
Traditionally, DMA activity has been viewed as peripheral to the system's main responsibility to do computing with the CPU. Under this view, it makes sense to give the CPU higher priority than DMA transfers. In modern, networked machines, communication can be at least as important as computation. It may be wise in future machines to consider giving DMA higher priority.

5 Steady-State Behavior with Many-to-one Communication

This section differs from the previous section in that all messages are sent to the same destination. The destination processor does not send messages but simply absorbs incoming messages. These experiments study network behavior when hot-spotting occurs. As before, we start with 256 byte messages, which should just fit into the 64 EISA cycles granted by a single EISA arbitration.

5.1 Message Latency

Figure 11 shows a plot of latency versus send rate as computed by the analytic model.



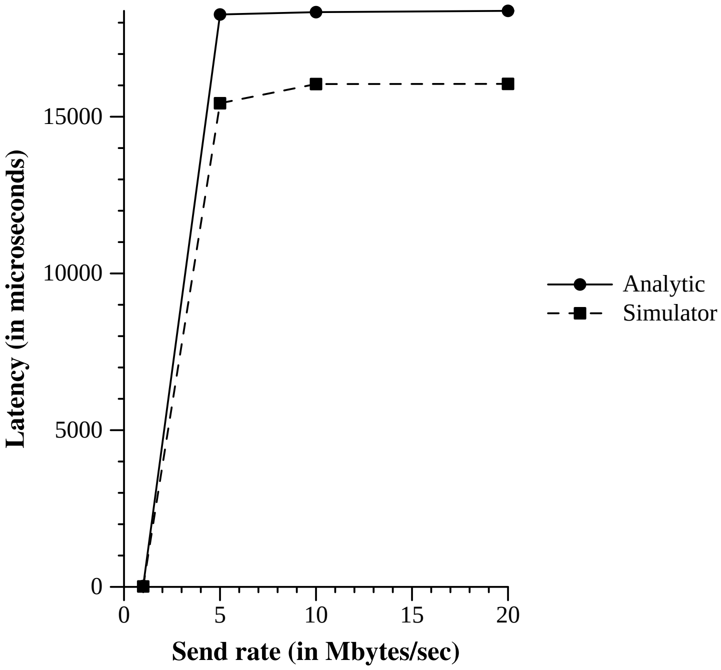


Figure 11: Latency vs. Send Rate for 256 byte messages, with many-to-one communication. The graph on the left is the full data; on the right the simulator numbers ignore data from the initial "warm-up" and final "cool-down" of the simulated system.

As the message send bandwidth is increased from 1 MBytes/s up to 20 MBytes/s, the message latency increases dramatically and then levels off. Already at 5 MBytes per second, the message latency has reached essentially its maximum value of 18300 μs. The increase in latency occurs because the destination processor is overwhelmed with messages. The destination's EISA bus cannot tranfer incoming messages to memory fast enough, so its incoming FIFO fills up and blocks. This blockage propagates backward through the network to the outgoing FIFOs and staging FIFOs of the senders. The large latency is caused by queueing in the staging FIFOs and outgoing FIFOs of the senders.

Note that with many-to-one communication, the network behaves like a fan-in tree. This behavior is independent of the network model used, so all three network models (wormhole, bus grid, and crossbar) behave the same.

Our explanation of the behavior in the high-latency cases can be tested by doing a simple calculation based on Little's Law. The predicted latency is equal to the total size of the FIFOs that we predict are full, divided by the rate at which packets are delivered to the destination.

This predicts a latency of 17352 microseconds. The actual latency should be somewhat smaller, since the FIFOs are not always completely full.

The analytic model predicts slightly higher latency, because it predicts that the receiver cannot quite deliver data to memory at the full EISA bus rate of 29.3 MB/sec. The receiver's in-fifo drains at 29.3 MB/sec when it is non-empty, and is filled by the network at 36 MB/sec when it is non-full. Because the time between drain-events and fill-events is modeled as random and exponentially distributed, the receiver's in-fifo is empty a small fraction of the time. As a result, the incoming FIFO drains at an average rate of 27.45 MB/sec. If we redo our Little's Law calculation with this effective delivery bandwidth, it predicts a latency of 18520 microseconds, which is, as we would like, slightly higher than the observed latency.

As figure 11 shows, the simulator predicts a much lower average latency. This occurs because the simulator runs send only 1000 messages; a nontrivial fraction of the simulation time is spent "warming up" the system at the beginning of the run and "cooling down" the system at the end of the run. If we discard the first 30% of the simulation time and the last 10%, we get the curve shown on the right side of figure 11.

5.2 Further Explorations

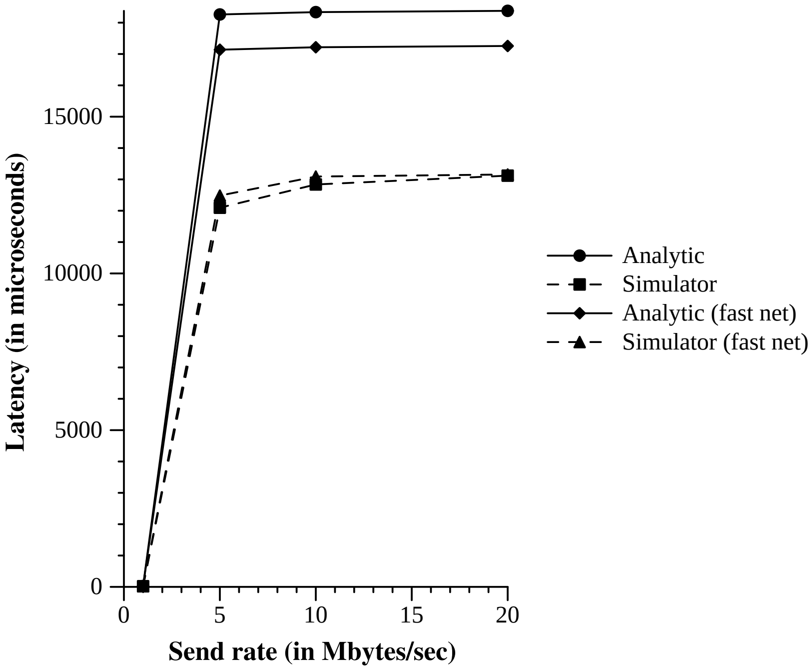


Figure 12: Latency vs. Send Rate, using the analytic model with many-to-one communication and 256-byte messages, for default network and fast network.

We now explore the effect of changing system parameters on many-to-one communication performance, using the analytic model.

Speeding up the network does little to improve the situation. This is not surprising, since the network is not the bottleneck. Figure 12 shows the new data. In this mode, message latency levels off at only a slightly lower value (17150 μs). The latency is lower because the faster network makes it much less likely that the receiver's in-fifo is empty, and hence packets can be delivered to the destination memory almost all of the time. Changes to buffer sizes lead to the predictable, linear change in latency; to save space, these graphs are not shown.

Increasing the I/O bus speed, however, increases the rate at which the receiving node can retire messages. Figure 13 shows the average message latency in the system when all parameters remain at the defaults, but the I/O bus speed is increased to 132 MBytes/s – the bandwidth of the PCI bus [6].

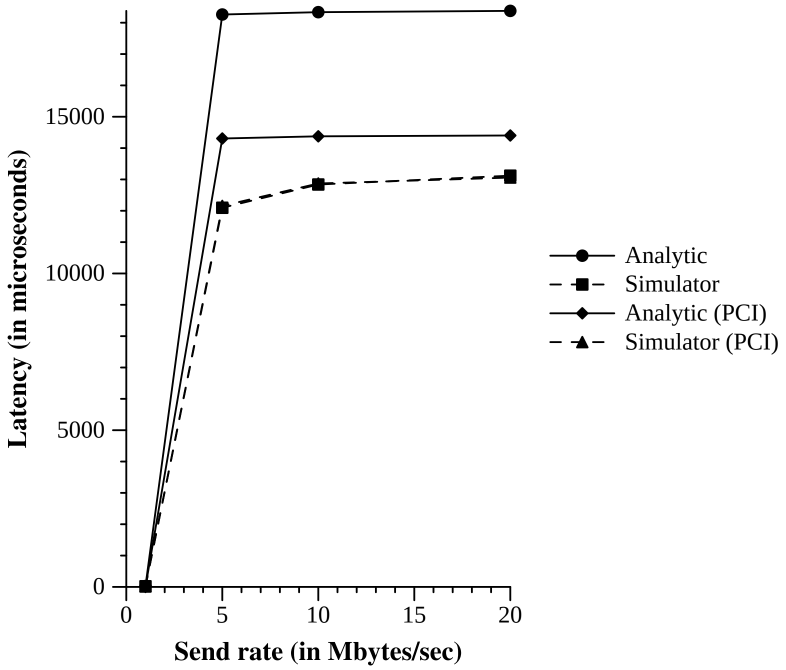


Figure 13: Latency vs. Send Rate, using the analytic model with many-to-one communication, for the default network speed, but with bus operations occurring at PCI speed (132 MBytes/s).

Throughput is no longer limited by the bus bandwidth of the destination node. It is now limited by the network speed, since all messages must pass through the single network link connecting the destination node to the network backplane. The incoming FIFO of the destination node is now mostly empty, but the network is congested, and this congestion causes the outgoing FIFOs and the staging FIFOs of the senders to be full. The latency levels off at 14340 microseconds.

Once again, we can use Little's Law to predict the latency. If we assume that the staging fifos and out fifos of all fifteen senders are full, and that the receiver can retire messages as quickly as they can pass across a 36 MB/sec network link, the predicted latency is 14507 microseconds. This is consistent with the model's prediction.

5.3 Transition to Saturation

We are also interested in how and when the transition to the high-latency, high-congestion case occurs as the send rate is increased. Figure 14 shows the latency as a function of send rate as the "critical" send rate is approached. As send rate approaches the transition point, the latency increases faster and faster. If we increased the send rate by 0.001 MB/sec beyond the right end of the graph, the latency would increase by a factor of 30. (We do not show this, because rescaling the graph would obscure the behavior as saturation is approached.)

The model predicts that the transition to saturation takes place at a send rate of 1.953

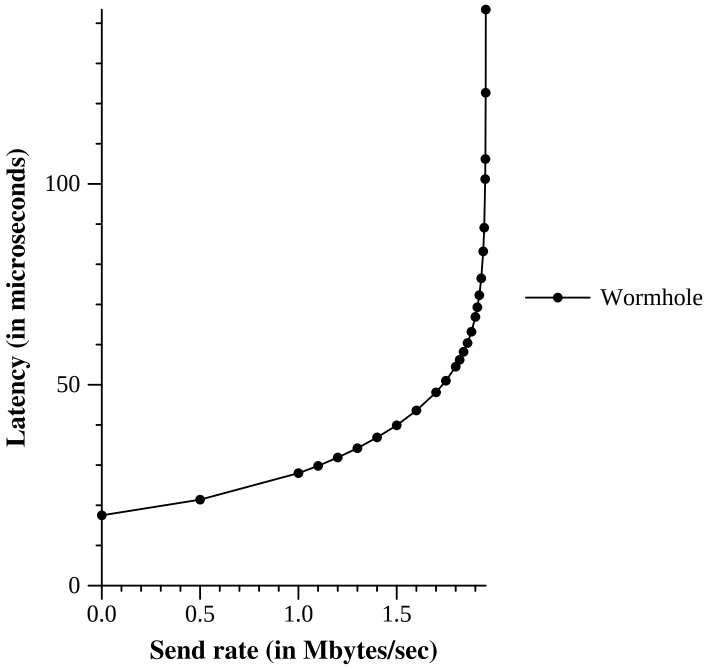


Figure 14: Latency vs. Send Rate, with many-to-one communication, as the system nears saturation.

MB/sec. This is exactly 1/15 of the speed of the receiver's EISA bus (29.3 MB/sec). As expected, the transition occurs when the aggregate send rate of the fifteen senders is higher than the capacity of the single receiver's slowest component.

If we run the simulator with the send rate just above the transition point, we see that it takes a long time for the system to reach its fully saturated state. The average message injection rate is only a little higher than the average message delivery rate, so messages build up in the system very slowly. It may take several thousand messages to reach steady state.

6 Bursty Traffic

We turn next to the behavior of the system under bursty message traffic. Most application programs tend to communicate in bursts, so we believe that the behavior of the system under bursts is important. We use the simulator to evaluate performance under bursty traffic, because it is very difficult to handle bursty traffic in the analytic model.

Our baseline bursty-communication runs use 16 processors sending 256-byte messages. Sending processes emit bursts of 150 messages. Within a burst, the average send rate is 20 MB/sec; the overall average send rate is 1 MB/sec. The time between bursts and the intermessage interval within a burst are both chosen randomly from exponential distributions.

In the all-to-all case, the average message latency with bursty communication is 18.70 microseconds. This is about 5% higher than the 17.79 microseconds we measured for non-bursty traffic with the same average send rate.

The difference appears to be that it takes longer on average for outgoing DMA to acquire the system bus in the bursty case. When a process is in a send-burst, one sixteenth of the messages it is sending are to itself. The result is that the arrival rate at a node increases when it is in a send-burst, so outgoing messages must wait longer on average to acquire the bus.

In the many-to-one case, the average message latency with bursty communication is 119.32 microseconds. This is considerably more than the 20.48 microseconds we measured for non-bursty traffic with the same average send rate. The difference is that the receiver is occasionally overwhelmed, which causes a backlog of packets. The backlogged packets experience very high latency, increasing the average.

If only one sender is in a burst, the receiver can handle the additional load, since the receiver's EISA bus can transfer 29.3 MB/sec, which is more than the 20 MB/sec that the sender is emitting. However, if more than one sender happens to be in a burst at the same time, then the receiver is temporarily overwhelmed. Coincidental bursts are rare, but they do occur. Since each of the 15 senders spends about 5% of it time bursting, the probability that more than one burst is occurring at a randomly-chosen time is about 17%.

With larger amounts of receiver-side buffering, we could smooth out bursts of many-to-one communication to avoid congestion backing into the network. The difficulty is that the incoming link from network to node is also a performance limiting link, since data from multiple senders converges as it passes through the network and into the node. Raising the bandwidth of this link well above expected application send rates would allow receivers to handle bursts gracefully, without causing congestion to propagate upstream.

7 Discussion and Future Work

The research presented in this paper has offered us several insights on the performance of the XYZ system. First, the analytic and simulation-based results allowed us to estimate performance and study the regimes in which system saturation occurs. Under conditions of communication uniformly distributed amongst nodes, we found that the system is fairly well-balanced. Although improving I/O bus bandwidth reduces latency somewhat, the bandwidth of the incoming network link also soon becomes a bottleneck. By using Intel's Network Interface Chip in "streaming mode", the incoming network bandwidth would once again almost match the I/O bus speed. We are currently investigating the performance implications of this configuration in both models.

Our study focused largely on system behavior under fairly controlled communication patterns. For the simulation-based numbers these patterns were generated using a synthetic benchmark. These stylized communication patterns allow us to pinpoint aspects of XYZ performance that would be difficult to isolate amongst aggregated statistics from whole program runs. For example, we noted that the all-to-all communication pattern avoids saturation because of a self-throttling effect. That is, since incoming and outgoing data share both the I/O bus and the NIC bus, heavy incoming traffic to a node prevents outgoing traffic from proceeding. This slows the overall message send rate, which eventually leads to reduced incoming bandwidth, and pulls the system away from the saturation point.

In many-to-one communication, it is quite easy to push the system into saturation.

We are currently extending this research by using the simulator to measure the behavior of message passing benchmarks as well. (The most promising pool of 'real' message passing applications are those written in NX, and we are currently working to implement fully the path from NX message passing calls to XYZ library calls and the simulator.)

8 Conclusions

References

[1] Anonymous. Omitted for blind review.

[2] Anonymous. Omitted for blind review.

[3] Anonymous. Omitted for blind review.

[4] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proc. 20th Annual Intl. Symp. on Computer Architecture*, pages 2–13, May 1993.

[5] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley and Sons, New York, NY, 1975.

[6] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.0*, Apr. 1993.

[7] P. Pierce. The NX/2 operating system. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, Jan. 1988.

[8] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On Characterizing Bandwidth Requirements of Parallel Applications. In *Proc. Sigmetrics '95 Intl. Conference on Measurement and Modeling of Computer Systems*, pages 198–207, May 1995.

[9] R. Traylor and D. Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of the Hot Chips '92 Symposium*, Aug. 1992.

[10] J. E. Veenstra and R. J. Fowler. MINT Tutorial and User Manual. Technical Report Technical Report 452, Univ. of Rochester Computer Science Department, June 1993. Revised August 1994.