LogP Performance Assessment of Fast Network Interfaces

David Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa
Computer Science Division
University of California, Berkeley

Abstract

We present a systematic performance assessment of the hardware and software that provides the interface between applications and emerging high-speed networks. Using LogP as a conceptual framework and Active Messages as the communication layer, we devise a set of communication microbenchmarks. These generate a graphical signature from which we extract the LogP performance parameters of latency, overhead, and bandwidth. The method is illustrated on three diverse platforms: Intel Paragon, Meiko CS-2, and a cluster of SparcStations with Myrinet. The study provides a detailed breakdown of the differences in communication performance among the platforms. While the details of our microbenchmark depend on Active Messages, the methodology can be applied to conventional communication layers.

1.Introduction

In recent years, we have seen dramatic advances in scalable, low-latency interconnection networks for parallel machines and workstation clusters. With such "hot interconnects," the performance of user-to-user communication is limited primarily by the *network interface*, rather than the switches and links. The network interface has a hardware component (the NI) and a software component. A diverse set of hardware designs have emerged, differing on key issues such as where the NI connects to the processing node and how much processing power is embedded in the NI, as can be seen in Table 1. We would like to evaluate and compare these design alternatives, but the hardware must be assessed in conjunction with software that provides a specific set of communication operations to the user. In principle, it should be possible use any of the popular communication layers, e.g., TCP/IP, MPI[9], or PVM[10], for this evaluation, however, these layers impose such large software overheads (several hundred to several thousand instructions per message) that all other factors become obscured[12]. There has been substantial work in "lean" communication layers, especially Active Messages[1], which provide simple communication operations with overheads on the scale of tens of instructions per message. Each Active Message layer is carefully optimized for a particular hardware platform, although the user communication operations are the same[4][5][6]. This presents the opportunity to perform a systematic assessment of the combination of fast hardware and fast software in delivering communication performance to applications.

TABLE 1. Two key axes in the network interface hardware design space

Connection	NI Processing Power		
	Controller	Embedded Processor	General Purpose Microprocessor
I/O Bus	Sun SAHI ATM	Fore SBA	IBM SP-2
	SP-1	Myrinet	
Graphics Bus	HP Medusa		
Memory Bus	Cray T3D	Meiko CS-2	Intel Paragon

Our goal is to characterize a range of network interface designs in terms of a few simple performance parameters, rather than as a litany of block diagrams and design specific timings. This approach is inspired by Saavedra's memory system microbenchmarks[2], which attempt to capture the salient features of a complex system through a focused set of small benchmarks that are analyzed in relation to a simple conceptual model. We use LogP[3] as our conceptual model of the communication system. It was developed as a tool for reasoning about parallel algorithm performance, and provides a concise characterization of the processing overhead, latency, and bandwidth of communication operations.

Specifically, we study three important platforms that represent diverse points in the NI design space – the Meiko CS-2[7], a cluster of SparcStation 20s with Myrinet[8], and the Intel Paragon[6]. Each machine is viewed as a "gray box" that supports Active Messages and conforms to the LogP framework. We devise a simple set of communication microbenchmarks and measure the performance on each platform. Our microbenchmark generates a graphical *signature* from which we can extract the LogP communication performance parameters of the hardware/software tandem.

Section 2 provides the background for our study by explaining the LogP model, the Active Message communication layer, and the specific hardware platforms under evaluation. Section 3 characterizes short message performance. It first develops an intuitive understanding of the microbenchmark signature and then analyzes the signatures of the three platforms to extract their performance parameters. We also show that subtle hardware/software variations of the communication interface can be observed using our microbenchmark. Section 4 extends the study to characterize bulk transfer performance. While the details of our microbenchmark depend on Active Messages, the approach applies broadly. In Section 5 we show the communication signature for an asynchronous RPC layer following exactly our methodology and comment on the broader use of the approach.

2.Background

This section provides a brief summary of the LogP model, Active Messages, and our hardware platforms. An in depth treatment of these topics can be found in the references, but the material presented here is sufficient for understanding the design evaluation in the remainder of the paper.

2.1 LogP Model

LogP was developed as realistic model for parallel algorithm design, in which critical performance issues could be addressed without reliance on a myriad of idiosyncratic machine details. The performance of a system is characterized in terms of four parameters, three describing the time to perform an individual point-to-point message event and the last describing the crude computing capability, as follows.

- **L**atency – an upper bound on the time to transmit a message from its source to destination.
- **o**verhead – the time period during which the processor is engaged in sending or receiving a message
- **g**ap – the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.
- **P**rocessor – the number of processors.

In addition, the network has finite capacity. The finite capacity of the network can be reached if a processor is sending messages at a rate faster than the destination processor can receive. If a processor attempts to send a message that would exceed the finite capacity of the network, the processor stalls until the message can be sent without exceeding the finite capacity limit.

These parameters are illustrated for a generic parallel system in Figure 1. The total time for a message to get from the source processor to the destination is $2o + L$. It is useful to distinguish the two components, because the overhead reflects the time that the main processor is busy as part of the communication event, whereas the latency reflects the time during which the processor is able to do other useful work. The gap indicates the time that the slowest stage, the bottleneck, in the communication pipeline is occupied with the message. The reciprocal of the gap gives the effective bandwidth in messages per unit time. Thus, transferring n small messages in rapid succession from one processor to another requires time $o + (n-1)g + L + o$, where each processor expends $n \bullet o$ cycles and the remaining time is available for other work. The same formula holds with many simultaneous transfers, as long as the destinations are distinct. However, if k processors send to the same destination, the effective bandwidth of each sender reduces to $1/(k \cdot g)$. In other words, the aggregate bandwidth of the k senders is limited by the receiver bandwidth, i.e. one message every g time units.

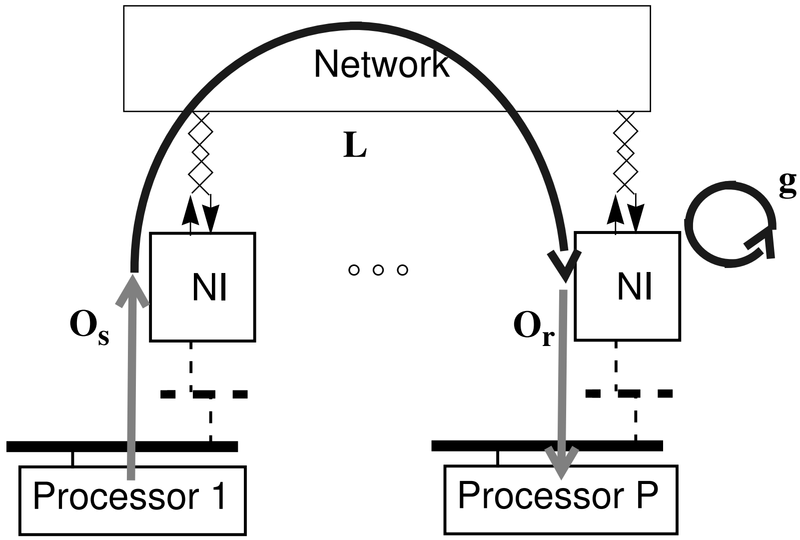

FIGURE 1. **LogP parameters in a generic platform**

The overhead parameter is generally determined by the communication software and is strongly influenced by cost of accessing the NI over the memory or I/O bus on which it is attached. The latency is influenced by the time spent in the NI, the link bandwidth of the network, and the routing delays through the network. The gap can be affected by processor overhead, the time spent by the NI in handling a message, and the network link bandwidth. For a very large system, or for a network with poor scaling, the bottleneck can be the bisection bandwidth of the network. However, in practice the network interface is often the bottleneck for reasonably sized systems, and that is the main focus of our study.

We make some small extensions to the LogP model. First, we distinguish between the send overhead, o_s, and receive overhead, o_r. Second, we recognize that for bulk data transfer, L, o, and g depend on message size. By differentiating the overhead, gap, and latency parameters, the LogP model exposes the overlap between computation and communication.

2.2 Active Messages

The Active Message communication layer provides a collection of simple and versatile communication primitives. It is generally used in libraries and compilers as a means of constructing higher-level communication operations, such as traditional message passing[9] or global shared objects[11]. Active Messages can be thought of as very lightweight asynchronous remote procedure calls, where each operation is a request/reply pair. In LogP terms, an Active Message request/reply operation includes two point-to-point messages, giving an end-to-end roundtrip time of $2(o_s+L+o_r)$. A request message includes the address of a handler function at the destination node and a fixed number of data words, which are passed as arguments to the handler. Active Messages are handled automatically, either as part of the node initiating its own communication, via an interrupt, or as part of waiting for responses.Otherwise, a node can also handle messages via an explicit poll. When the message is received at the destination node it invokes the specified handler, which can perform a small amount of computation and issue a reply, which consists of an analogous reply handler function and its arguments. This basic operation is illustrated in Figure 2 by typical remote read transaction.

Active messages are efficient to implement because messages can be issued directly into the network from the sender and, since the code that consumes the data is explicitly identified in the message, processed directly out of the network without additional buffering and parsing. The handler executes in the context of a prearranged remote process and a fixed set of primitive data types are supported, so the argument marshalling and context switching of a traditional RPC are not required. The sender continues execution as soon as the message is issued; invocation of the reply handler provides notification of completion.

Active Messages has been implemented on the n-Cube/2, CM-5[1], HP workstations with the Medusa FDDI network[4], Sun workstations connected to an ATM network[5], Intel Paragon[6], Meiko CS-2, and Sun workstations with Myrinet. Each Active Message implementation is optimized for the particular hardware. The Generic Active Message (GAM) specification defines a uniform, application programming interface (API) across these platforms.[1] Small messages provide four words of data to the handler and are reliable. (The request/reply protocol permits inexpensive schemes for deadlock avoidance, flow control and error recovery.) In addition to small message transfers, bulk transfers are supported as a memory-to-memory copy in either the request direction or the reply direction; invocation of the handler signifies that the data transfer is complete. In the bulk transfer case we examine, the requester transfers n bytes of data into the remote virtual memory before the request handler fires, and the reply handler indicates completion of the entire transaction.

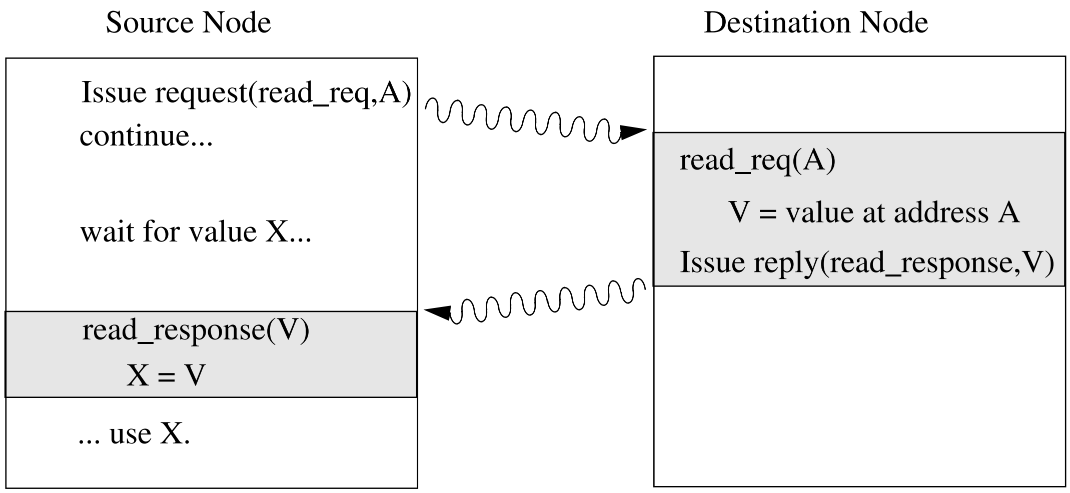

FIGURE 2. The two basic primitives of Active Messages are small request and reply active messages. The grey areas represent control taken as a result of processing an incoming request/response.

2.3 Hardware Platforms

We evaluate the communication performance of three platforms: the Intel Paragon, the Meiko CS-2, and a network of SUN SparcStation 20s connected by Myrinet switches using the LANai S-Bus cards. All of the

1. Generic Active Message Interface Specification v1.1 is available on-line at http://now.cs.berekeley.edu/Papers/gam_spec.ps.

platforms follow the generic hardware model in Figure 1, with a collection of essentially complete computers connected by a scalable communication network. Moreover, they all provide a communication processor and specialized DMA engines as part of the network interface hardware. However, they differ in key design aspects, summarized in Table 2, including where the NI connects to the main processor, the power of the communication processor, and the network link bandwidth. The Paragon uses a general purpose i860XP RISC processor, identical to the main processor, as a dedicated communication processor. The two processors communicate via shared memory over a cache-coherent memory bus. In addition, the communication processor accesses the network link FIFO's across the memory bus. Two DMA engines connect to the memory bus and burst data between memory and the network. In the Meiko CS-2, the communication processor and DMA engine are contained within the Elan network interface chip, which connects directly to the memory bus and to the network. The two processors communicate via shared memory and uncached accesses directly to the Elan. The communication processor has a dedicated connection to the network separate from the memory bus, however, it has only modest processing power and no general-purpose local memory or cache. The Myrinet NI is a I/O card that plugs into the standard S-Bus. It contains a 16-bit CISC-based embedded processor, DMA engines, a modest amount of local memory, and the Myrinet link interface. The Myrinet local memory is used as a staging area for incoming and outgoing DMA operations. For example, a memory copy request will transfer data from the host memory to the Myrinet local memory, then to the network. The Myrinet and Meiko CS-2 offer comparable network link bandwidth at 80 MB/sec and 70 MB/sec respectively. The Myrinet network can be an arbitrary topology connected via crossbar switches with eight bidirectional ports, while the Meiko CS-2 uses two 4-ary fat trees. The Paragon has the highest network bandwidth at 175MB/sec although its 2D mesh network is not as scalable as a fat tree. Moderate sized configurations are used in our measurements since our focus is on the network interface performance, rather than the scaling of the network itself.

TABLE 2. Comparison of the three platforms used in this study

Platform	Main Processor	NI Location	Communication Processor	Network Topology	Peak Network Bandwidth (MB/s)
Intel Paragon	50MHz i860XP	memory bus	50MHz i860XP	2D mesh	175
Meiko CS-2	66 MHz Hyper-SPARC	memory bus	Elan (embedded processor)	4-ary fat tree	70
Myrinet	50MHz Super-SPARC	I/O bus	LANai (embedded processor)	8-port cross-bar	80

There are some semi-technical forces that influence the performance of these classes of machines, such as time-to-market. For example, the Meiko CS-2 used in this study was made available only very recently (a few months before the paper), while the newest Myrinet technology did not make it in time for our study. Thus, cross-machine comparisons do not reflect a snapshot of the 'latest-and-greatest' technology.

3.Small Message Performance

In this section we obtain the LogP parameters for our study platforms using a simple communication microbenchmark. The natural starting point is the round-trip time (RTT) associated with a single Active Message request-reply operation. Table 3 shows the minimum and maximum RTT between pairs of nodes for our platforms in configurations up to sixteen nodes. We can see from this that the NOW communication time is about 1.5 times that of the two MPP platforms. The RTT reflects the sum of the send overhead, latency, and receive overhead. As expected, it varies by a small amount with the distance traveled through the network. Other factors such as the speed of the NI and communication overhead dominate the communication time.

TABLE 3. Round-trip time (in microseconds)

Parameter	Paragon	Meiko CS-2	Myrinet
RoundTrip $2(o_s + o_r + L)$	19.9 - 20.1	20.3 - 21.6	30.6 - 31.5

To extract the individual LogP parameters our microbenchmark will issue a sequence of request messages and measure the average time per issue, which we call *message cost*. The overhead and gap can be deduced from the changes in the message cost as a function of the number of messages issued. We first illustrate the technique in the abstract and then apply it to our study machines.

3.1 Communication Microbenchmark

As a first step towards our communication microbenchmark, consider what should be the time to issue a sequence of M Active Message requests under LogP, as illustrated by the following pseudo-code. The issue-phase is defined as the code between the start/stop timer statements.

```
Start Timer
        Repeat M times
                Issue Request[1]
Stop Timer

... handle remaining replies
```

For small M, the sender will issue all the requests without receiving any replies, as indicated by the top time-line of Figure 3. Thus, the message cost should be simply o_s. (This occurs for M less than RTT/o_s.) For larger M, a fraction of the replies will arrive during the issue phase and the message cost will increase, as indicated by the second time-line in the figure, since the processor will spend o_r for each reply. The replies will be separated by the time for successive messages to pass through the bandwidth bottleneck, *i.e.*, by g. As M increases, the number of messages in the network increases. Eventually, the capacity limit of the network will be reached and a reply must be drained before each new request issued, *i.e.*, when the request function attempts to inject a message into a full network it will stall. Therefore, the message cost will be simply the gap, g.

Thus, the average message cost, as a function of M, should follow a curve as shown by the bottom-most curve in Figure 4. It exhibits three regimes, 'send-only' for small M, 'steady-state' for large M, and a 'transition' in between. The average time per message in the 'send-only' regime reveals o_s. The transition regime will begin at RTT/o_s when the first replies begin to return, or when the capacity limit is reached; whichever comes first. It asymptotically reaches g in the steady-state. The RTT measurement gives us the sum $o_s + L + o_r$r, if we can obtain o_r we can solve for L.

1. The Active Message request function implicitly handles replies, as noted in Section 2.

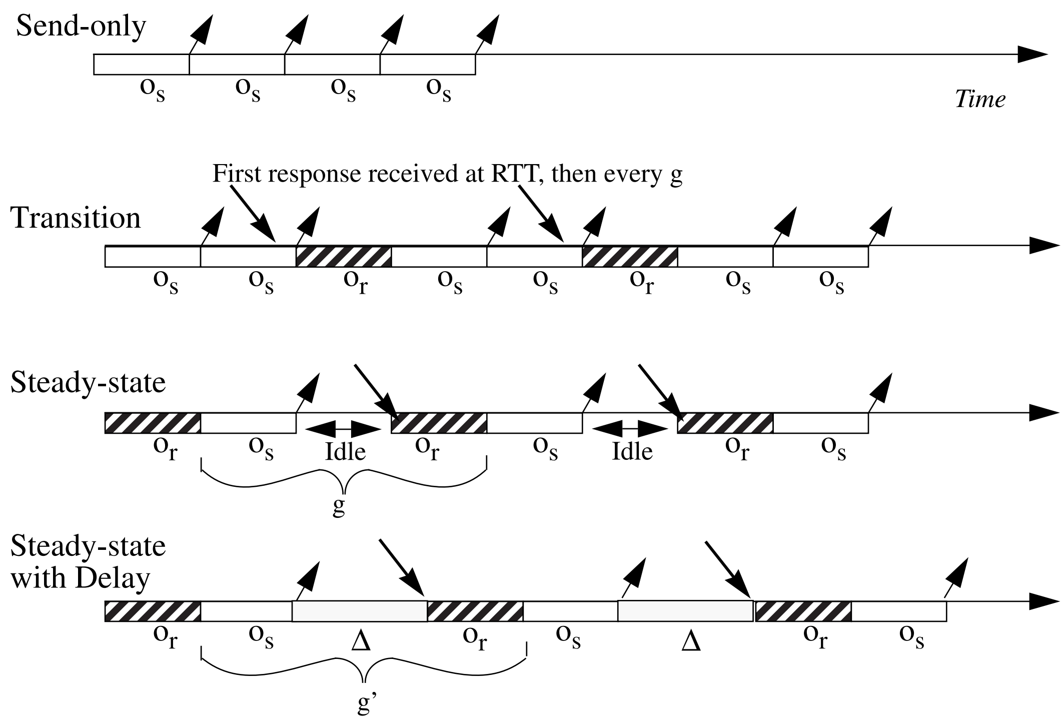

FIGURE 3. Request issue time-line

We can see from the steady-state time-line of Figure 3 that $o_s + o_r + Idle = g$, where Idle is the time the sender spends waiting to drain a reply from the network. However, we cannot directly measure Idle or o_r; since the request function stalls waiting for a reply and then uses o_r time pulling the reply out of the network. Therefore, we attempt to find o_r in a different manner; we add a controlled amount of computation, Δ, between messages. As indicated in the bottom time-line in Figure 3 for Δ>Idle, the sender becomes the bottleneck and the average message cost is $o_s + o_r + \Delta = g'$, where g' is the new bottleneck, e.g., the average time per issue in the steady-state with delay regime. Since we know Δ and can measure o_s and g', we can solve for o_r.

3.2 Microbenchmark Signature

Here is the resulting pseudo-code for our communication microbenchmark:

```
Start Timer
    Repeat M times
        Issue Request
```

Compute for Δ time

Stop Timer

... handle remaining replies

By executing this microbenchmark for a range of M and Δ, we construct a *signature* consisting of several message issue curves, each corresponding to a different value of Δ, as illustrated by Figure 4. In the figure, any value of Δ less than Idle will have a steady state message cost of 'g', while any value of Δ larger than Idle, e.g. Δ', will have a steady state message cost of g' >g. From this graph we can directly read the parameters g, o_s , and o_r. Then, we can compute latency given the round-trip time.

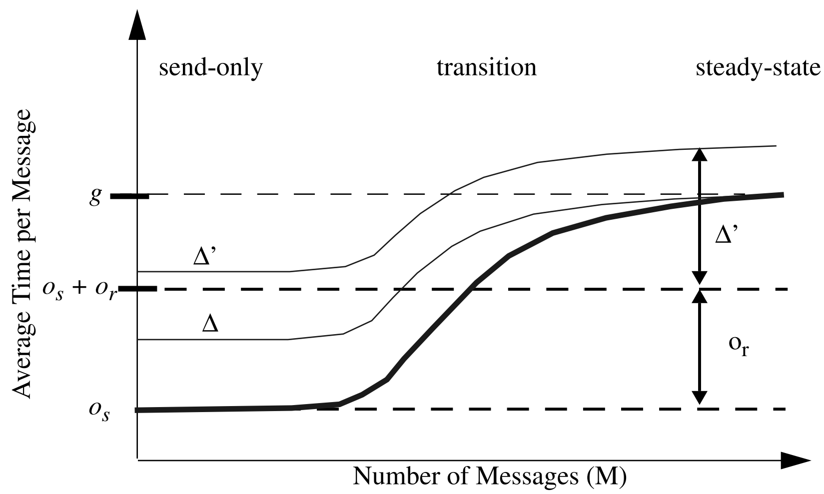

FIGURE 4. Expected microbenchmark signature

3.3 Small Message Empirical Results

Since many of our measurements are in the microsecond range, any undesirable events such as cache misses, context switching, and timer interrupts can lead to significant errors in the measurements. We repeat the microbenchmark for each point in the signature until we obtain an accuracy of ±5% and a confidence level of 95%. We process the measurements in batches of 50 samples to minimize the cache effect of the statistics collection routines.

Figures 5, 6, and 7 give the microbenchmark signatures of our three platforms. The empirical signatures of our platforms closely model the abstract signature of Figure 4. Each graph exhibits the three regimes of operation: Send-only, Transition, and Steady-state. Given the signatures, we can extract the LogP parameters. For the Paragon signature in Figure 7, by averaging the first few points corresponding to small M and Δ = 0, we find that o_s is 1.4 microseconds. Next, by taking the asymptotic value of the Δ = 0 curve we find that g is 7.6 microseconds. To compute o_r, we arbitrarily pick a value of Δ that increases the gap, e.g., the curve Δ = 16. Subtracting Δ + o_s from g' =19.6 gives us o_r to be 2.2 microseconds. Finally, subtracting the overhead from the one-way time (RTT/2) we get L = 7.5 µsec. Similar analysis finds the LogP characterization for Meiko to be o_s =1.7 µsec, o_r = 1.6 µsec, g= 13.6 µsec, and L= 7.5µsec. On the Myrinet, o_s =2.0 µsec, o_r

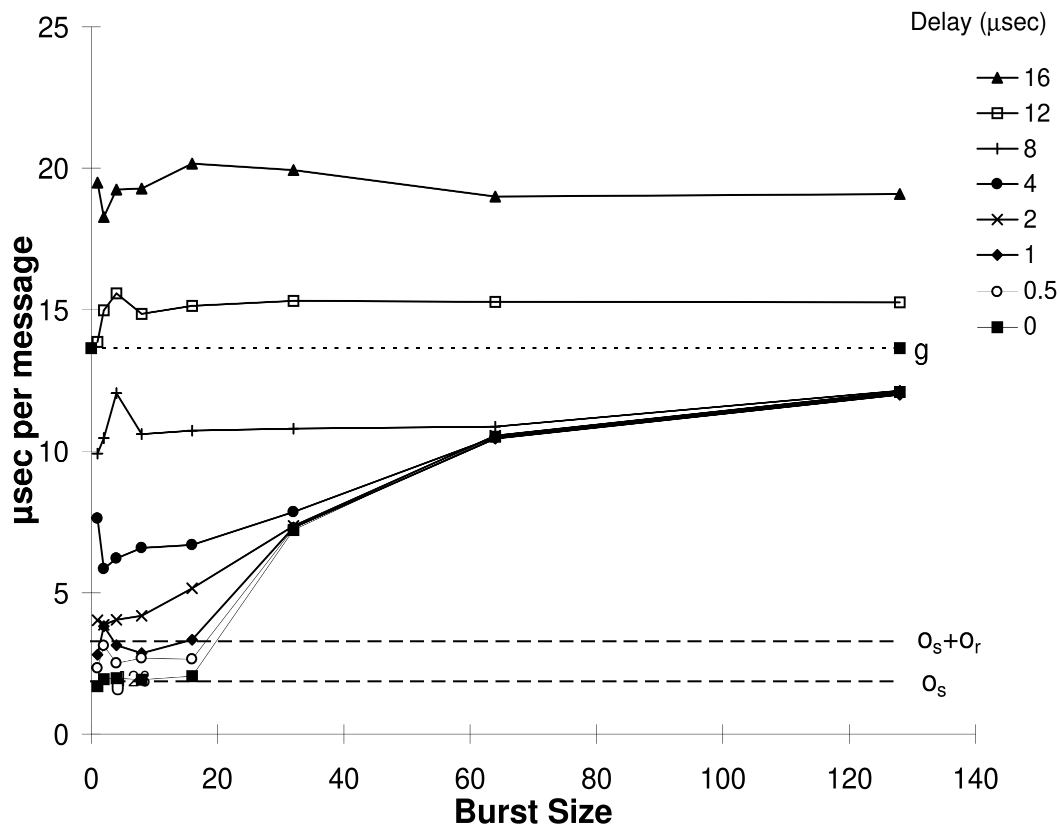

FIGURE 5. Meiko microbenchmark signature

= 2.6 µsec, g= 12.4 µsec, and L= 11.1 µsec. The asymptotic value of the $\Delta = 0$ curve occurs at very large values of M for both the Meiko and Myrinet. For clarity we have only shown the message cost up to M=128.

Figure 8 presents a summary of the LogP parameters for our platforms, along with two variations that we discuss later. The bars on the left show the one-way time divided into overhead and latency, while the ones on the right show the gap. The Myrinet time is roughly 50% larger than the two MPP platforms, although the message bandwidth is comparable. The larger overhead on the Myrinet (4.6 µsec) compared to the MPP platforms (3.3 µsec and 3.7 µsec) reflects the first of the key design choices noted in Table 2. The Meiko and Paragon NIs connect to the cache-coherent memory bus, so the processor need only store the message into the cache before continuing. On the Myrinet platform the NI is on the I/O bus, and the processor must move the message into the NI with uncached stores, resulting in larger overhead.

The latency and gap reflect the second key design issue: the processing power of the NI. The Paragon has the lowest latency, 6.3 µsec, followed by the Meiko and Myrinet, with latencies of 7.5 and 11.1 µsec, respectively. This indicates the advantage of the microprocessor used in the Paragon over the custom embedded processors in the Meiko and Myrinet designs. The Paragon also has the lowest gap, 7.6 µsec, compared to 12.4 µsec for the Myrinet 13.6 µsec for the Meiko. The difference between the Meiko and Myrinet is interesting. Even though the Meiko communication processor resides on the memory bus, it has no caches and so must load messages using uncached reads from the main processor's cache, affecting the rate at which it can send messages. On the Myrinet, the main processor has already deposited the messages into NI memory,

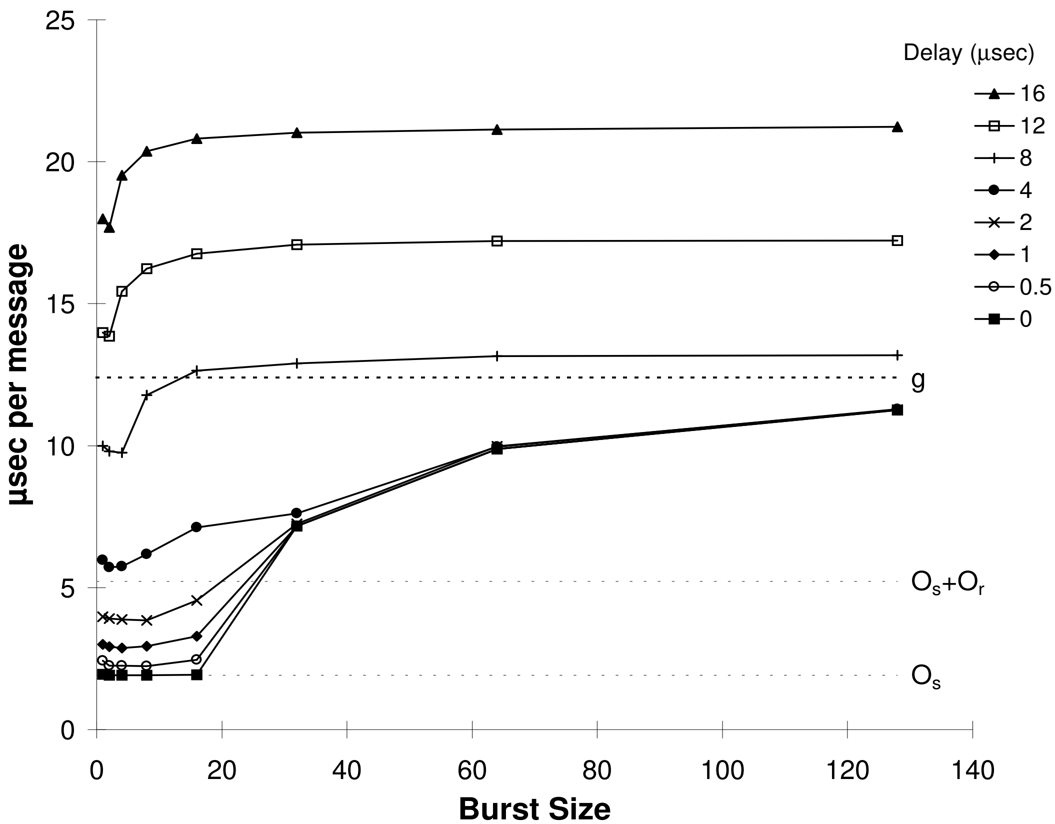

FIGURE 6. Myrinet microbenchmark signature

where the communication processor can access it quickly. Furthermore, since a substantial part of the latency is the processing in the NI, rather than the actual network transfer, trade-offs exist between the overhead and latency components. For example, the Myrinet latency can be reduced with an increase in overhead by performing the routing lookups on the main processor. While this change might reduce the overall communication cost, since the main processor is faster, bus transfers and other factors might mitigate the advantage. The microbenchmark provides a means of quantifying such trade-offs.

3.4 Evaluating Design Trade-offs

The microbenchmark is a valuable tool for evaluating design changes in the communication hardware and software. In many cases, the performance impact is unexpected and subtle, as illustrated by two slightly older variants on our main platforms.

One expects the overhead to track the processor speed, but L and g to be unaffected. When we run the microbenchmark on an older Meiko with 50 Mhz SuperSparc processors using the same Active Message implementation, o_s and o_r increase slightly while L and g increase by about 30%, as illustrated by the Meiko50 bars in Figure 8. Part of the reason is that this machine has a slower memory bus (40 vs. 45 MHz), and the NI processor runs at the memory bus speed. This does not seem to account for the entire difference, but the microbenchmark has served its role in revealing the anomaly.

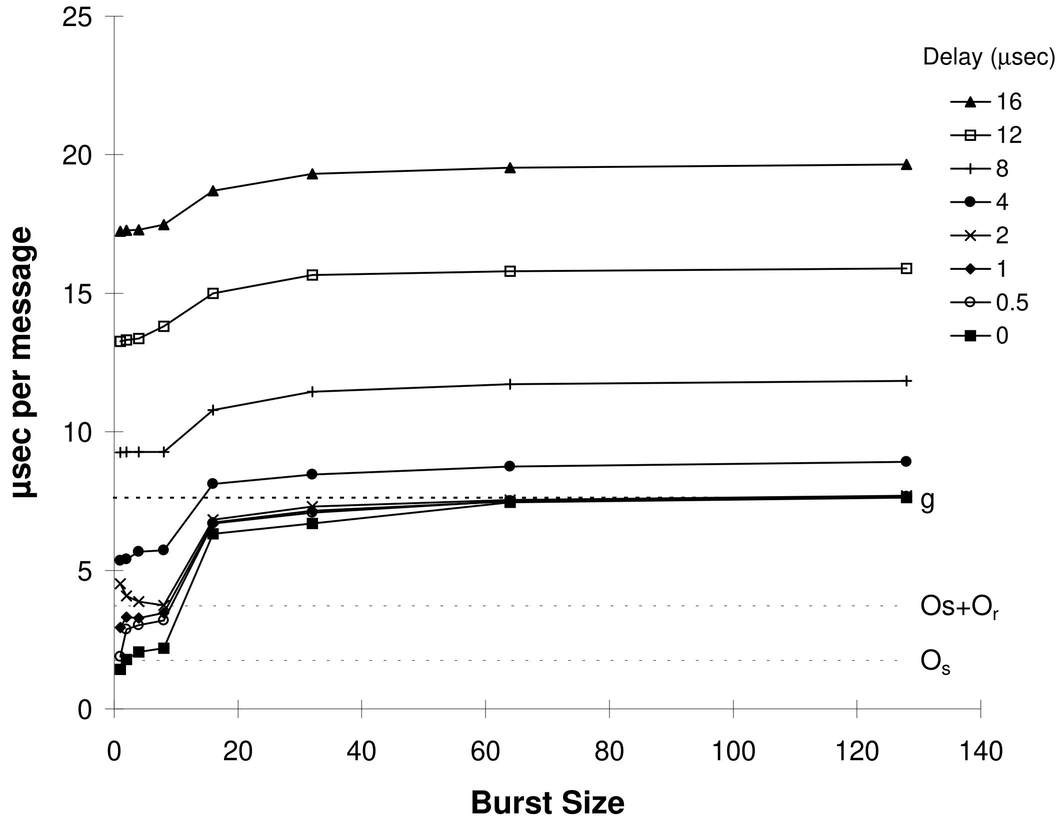

FIGURE 7. Paragon microbenchmark signature

One generally expects program performance to improve with the addition of a large second level cache, however, this may not be the case for communication. The Myrinet10 bars in Figure 8 summarize the performance of an alternative Myrinet platform using SparcStation 10s with the same 50MHz SuperSPARC, and a second-level cache. We see a dramatic increase in overhead, o_s =3.6 µsec and o_r = 4.0 µsec! Processor loads and stores the NI incur an extra delay through the second-level cache controller before reaching the I/O bus. We believe the I/O bus is slower, as well, accounting for the increase in g, since the NI processor is clocked at the I/O bus speed.

3.5 Validation of communication scaling

The measurements conducted thus far calibrate the communication performance of a single pair of nodes in the network. In a scalable network, as assumed by the LogP model, the same performance should be observed for communication between multiple pairs, as long as there is no contention for the individual nodes. To validate that our platforms behave in this manner, we repeated the microbenchmark using all the nodes: half of the nodes as requesters and the other half replying. We see no significant difference in the signatures; the LogP parameters are the same to two significant digits.

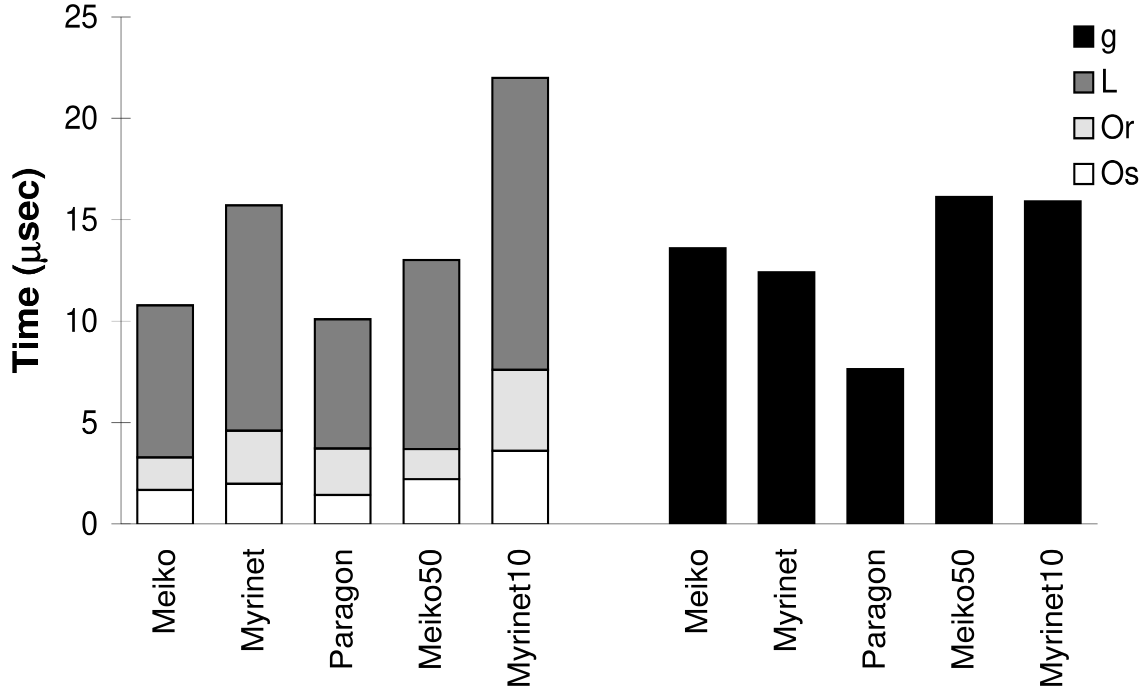

FIGURE 8. LogP summary of the three platforms, including two variations

The LogP model asserts that we should see the same performance if a single node issues requests to many other nodes. Modifying the microbenchmark in this fashion we see that the gap actually decreases by about 30%, and we get roughly 1.4 times the message bandwidth on both the Myrinet and Paragon. This suggests that handling the request turning around reply is the bottleneck. By increasing the number of receivers, we mitigate the bottleneck. On the Meiko, however, the receiver is not the bottleneck.

In the presence of prolonged end-point contention, the LogP model asserts that the aggregate bandwidth will be limited by the bandwidth of the contended receiver. Each sender should see the gap increase in proportion to the number of senders. In this variation on the microbenchmark, there are K nodes making requests to a single node. Table 4 shows the observed increase in the gap in the K-to-1 microbenchmark relative to the 1-to-1 microbenchmark for varying numbers of requesters on our three platforms. The results closely fit our model expectations.

TABLE 4. Contended Validation of g

	Meiko	**Myrinet**	**Paragon**
Number of nodes (K)	15	3	7
$g_{\text{K-to-1}}/g_{\text{1-to-1}}$	16.8	3	6

4.Bulk Transfer Microbenchmark

In this section we extend the methodology used for small messages to evaluate the communication performance of bulk transfers. Active Messages provides two bulk operations, *store* and *get*. A store operation copies a block of memory to a remote node and invokes a request handler on the remote node, which will send a small reply to the requester. Conversely, a get operation copies a block of memory from a remote node and invokes reply handler on the originating node. The store and get operations function like memory copies across the network and the handler invocation signals the completion of the copy. The extension to the microbenchmark is straightforward – each request transfers n bytes of additional data. We can time the issue of a sequence of requests and we can insert a computation delay between the requests in the sequence. The natural generalization of the LogP model is to view the overhead and the gap as a function of n, rather than as fixed constants. The microbenchmark signature must also be extended, since each curve becomes a surface with the additional dimension of transfer length.

4.1 Bulk Transfer Time and Bandwidth.

The equivalent of the "round-trip" measurement for bulk transfers is the time to complete an n byte store operation and receive the reply. Figure 9 shows the time, $T(n)$, for this operation as a function of transfer size on our platforms and the corresponding transfer bandwidth delivered, $BW(n) = \frac{n}{T(n)}$.

Many studies model large message performance using two parameters, the start-up cost, T_0, and a peak rate, R_∞. The time for an n byte transfer is modeled as $T(n) = T_0 + \frac{n}{R_\infty}$. Commonly, these values are derived by fitting a line ($T_0 + nR_\infty$) to a set of measurements, such as those in Figure 9. T_0 is the intercept of the line, and $\frac{1}{R_\infty}$ the slope. While this method is accurate in obtaining the peak bandwidth, the meaning of the T_0 term is not clear. First, non-linearities and small errors in the data often yield a value of T_0 that has little to do with the time for a small message. For example, a least-squares fit of the Myrinet data in Figure 9 yields a negative T_0. Even if the fit is reasonable, there is no indication whether T_0 reflects processing overhead, communication latency, or some sort of protocol between the sender and receiver, such as the round-trip for the acknowledgment of the transfer in the active message bulk store operation. The generalization of LogP provides a framework for articulating these issues.

A more serious shortcoming of the T_0, R_∞ model is that it does not reveal how busy the processor is during the transfer. For example, an algorithm designer may wish the know how much computation can by overlapped with communication. Much as we need to distinguish o and L in the small message case, we need to determine $o(n)$ for bulk transfers, as distinguished from $T(n)$.

4.2 Bulk Transfer Overhead.

To measure the overhead of a bulk transfer, we use a methodology similar to the small message case. A sequence of m bulk transfers is issued with a computation time of Δ between each one. The signature is

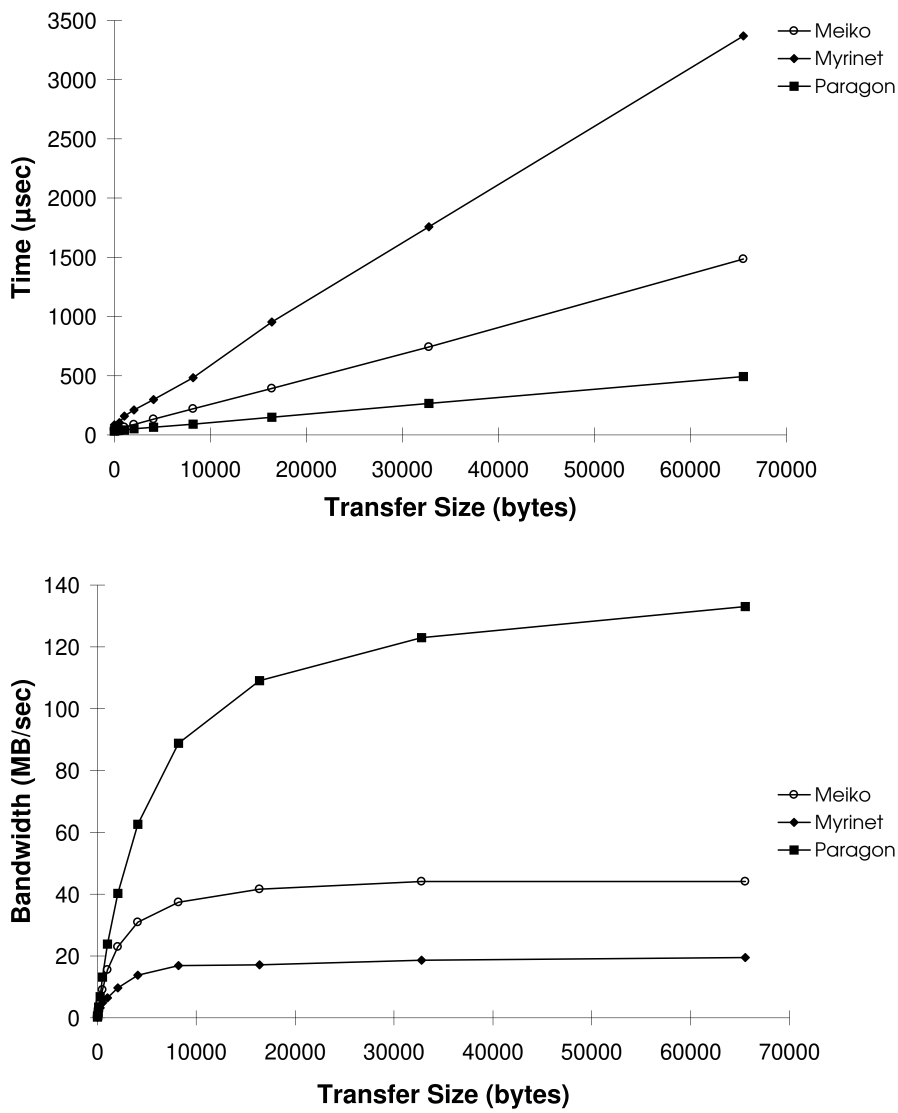

FIGURE 9. **Bulk transfer completion time and resulting bandwidth**

obtained for each given transfer size n. Figure 10 shows the Myrinet signature for a transfer size of 8 bytes. It exhibits the same three regimes as for the small message case. However, g is 18 μsec compared with 12.4 μsec for the small message case. One can conclude that some part of the system is slower at sending bulk messages than small messages even when the additional data transfer is small. Indeed, we can attribute this extra overhead to the cost that the communication processor incurs during the DMA transfer from host memory. For small messages the compute processor stores the data directly into the NI memory and avoids the DMA transfer.

Figure 11 shows the Myrinet signature for a 2K byte transfer. The three regimes are no longer identifiable and the curves for small Δ's do not converge. We can conclude in this case that the bulk operation is overhead limited and the g term is not visible.

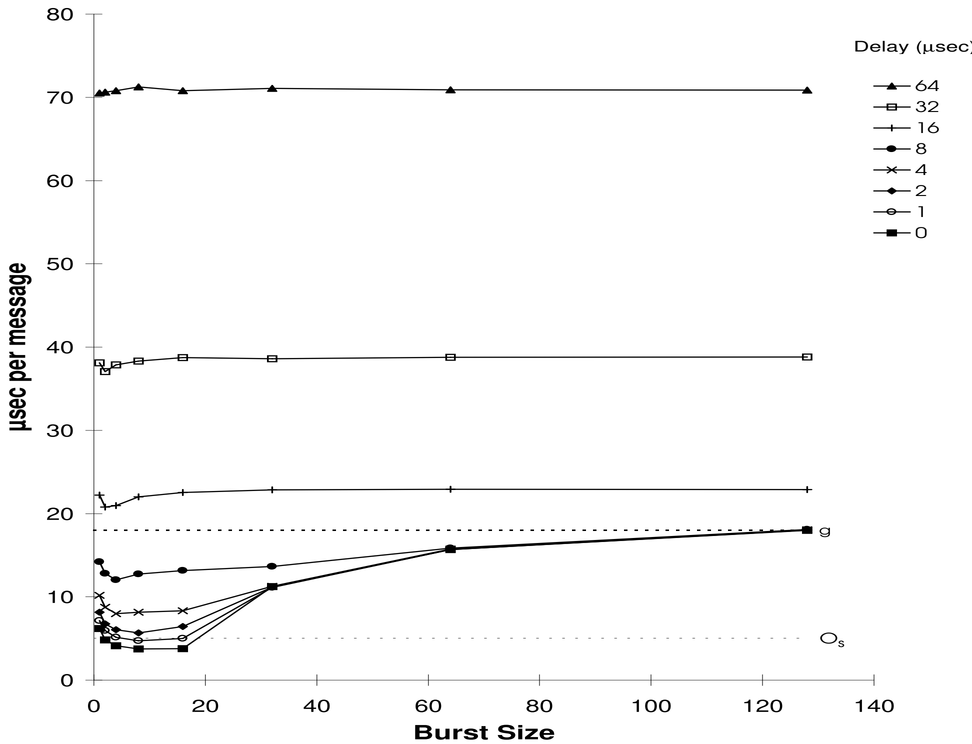

FIGURE 10. Myrinet 8-Byte Bulk Transfer Signature

To compute $o_s(n)$, we take the series of the o_s from the m, Δ signatures for increasing values of n. Figure 12 shows the result for each platform for up to 4K byte messages. The Paragon and Meiko have constant send overhead as a function of the transfer size. These machines allow a program to fully overlap computation with communication. Figure 12 shows that the overhead on the Myrinet increases linearly with the transfer size. Thus, less overlap is possible on that platform than on the other two. Because of I/O addressing limitations and the high cost of locking pages in memory, the processor copies data into pinned I/O addressable regions rather than lock user pages and then map them into I/O space. The microbenchmark signature clearly shows this cost, thus showing how the lack of integration in the Myrinet platform affects performance. However, $o(n)$ is smaller than $T(n)$ showing that a portion can be overlapped. For example, for a 4K byte transfer, $o(n)$ is 143 µsec, and $T(n)$ is 298 µsec.

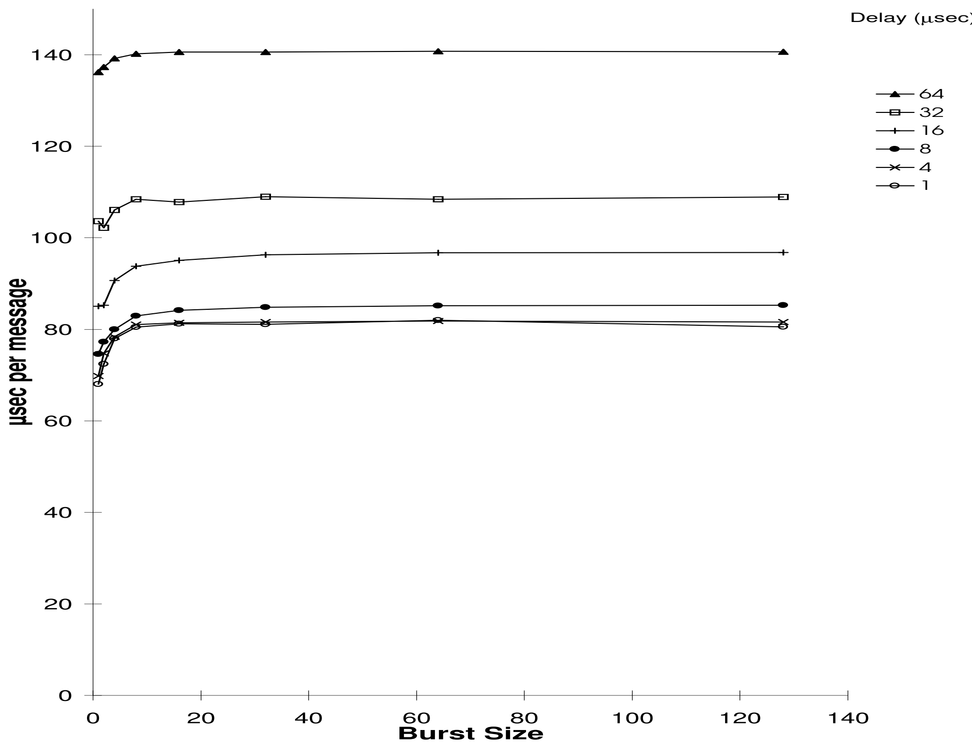

FIGURE 11. Myrinet 2KB bulk transfer signature

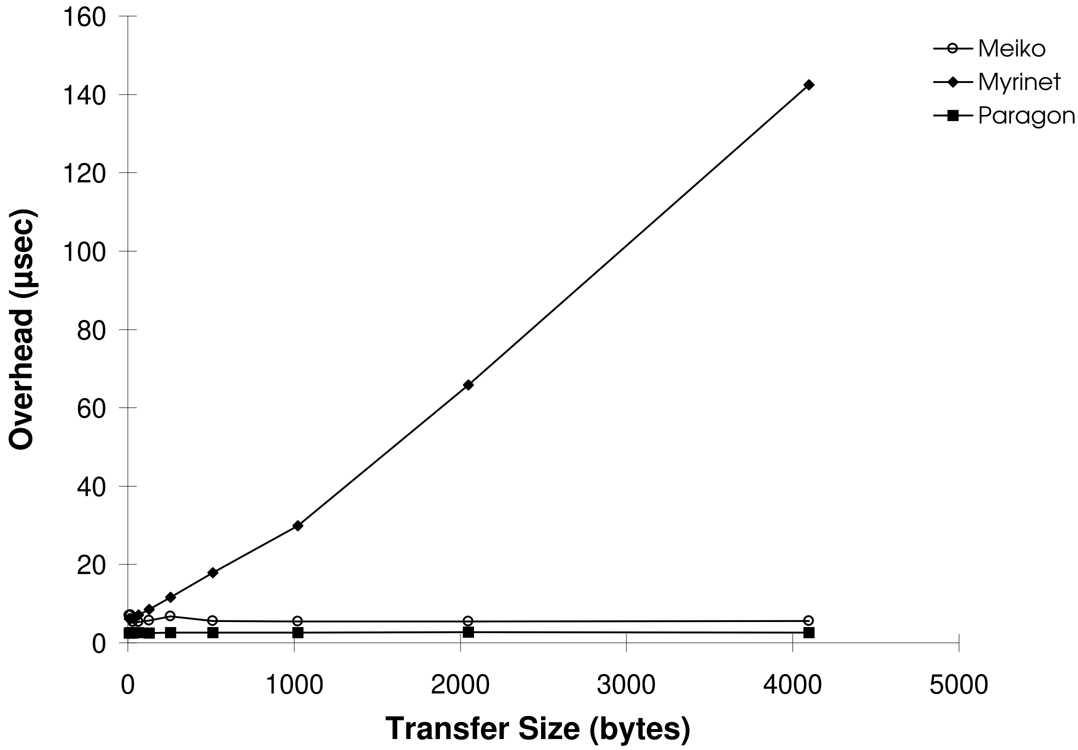

FIGURE 12. Comparison of bulk transfer send overhead vs. transfer size

5. Conclusion

Communication performance at the application level depends on the synergy of all components in the communication system, especially the network interface (NI) hardware and the low-level communication software that bridges the hardware and the application. In this paper, we present a systematic framework to assess the performance trade-offs in NI and communication software designs. We emphasize the use of a simple, low overhead communication layer such as Active Messages to expose the performance characteristics of the communication hardware. We use the LogP model as a conceptual framework to characterize a range of NI designs in terms of a few simple performance parameters that capture the essential features of the communication system. The LogP model addresses the overlap between computation and communication by differentiating the latency, overhead, and bandwidth.

To illustrate our benchmark methodology, we study three interesting platforms that occupy diverse points in the NI design space – a Meiko CS-2, a cluster of SparcStation 20s with Myrinet, and an Intel Paragon. The platforms differ in where the NI connects to the processing node and how much processing power is embedded in the NI. Each machine is viewed as a “gray box” that supports the same Active Message interface. We devise a simple set of communication microbenchmarks. Our microbenchmark generates a graphical *signature* from which we can extract the LogP performance parameters that characterize the performance of the hardware/software in combination. Our benchmark results show that our network of workstations (NOW) connected by Myrinet has competitive communication performance compared to the Meiko CS-2 and the Intel Paragon. The study provides a detailed breakdown of the communication performance and how it differs among the platforms. We are also able to calibrate the effects of changes, such as a faster bus, a new cache level, or new protocol between the main processor and the communication processor.

Microbenchmarks are a valuable tool for assessing the impact of design trade-offs in communication hardware and software. While the specifics of our microbenchmark depend on Active Messages, the methodology applies to any communication layer. For example, Figure 13 shows the microbenchmark signature for an asynchronous RPC layer on SparcStation 10's running Solaris 2.4 connected by Ethernet. (We can see that the high software overhead of RPC is the performance bottleneck, since the first value of Δ increases the gap. Thus, the gap is equal to $o_s + o_r$). Our empirical methodology carries over directly to RPC, because it is a request/reply operation. Traditional message passing and stream communication models require a somewhat different microbenchmark formulation. However, we believe that any standard communication interface should include a performance calibration suite such as the one presented here.

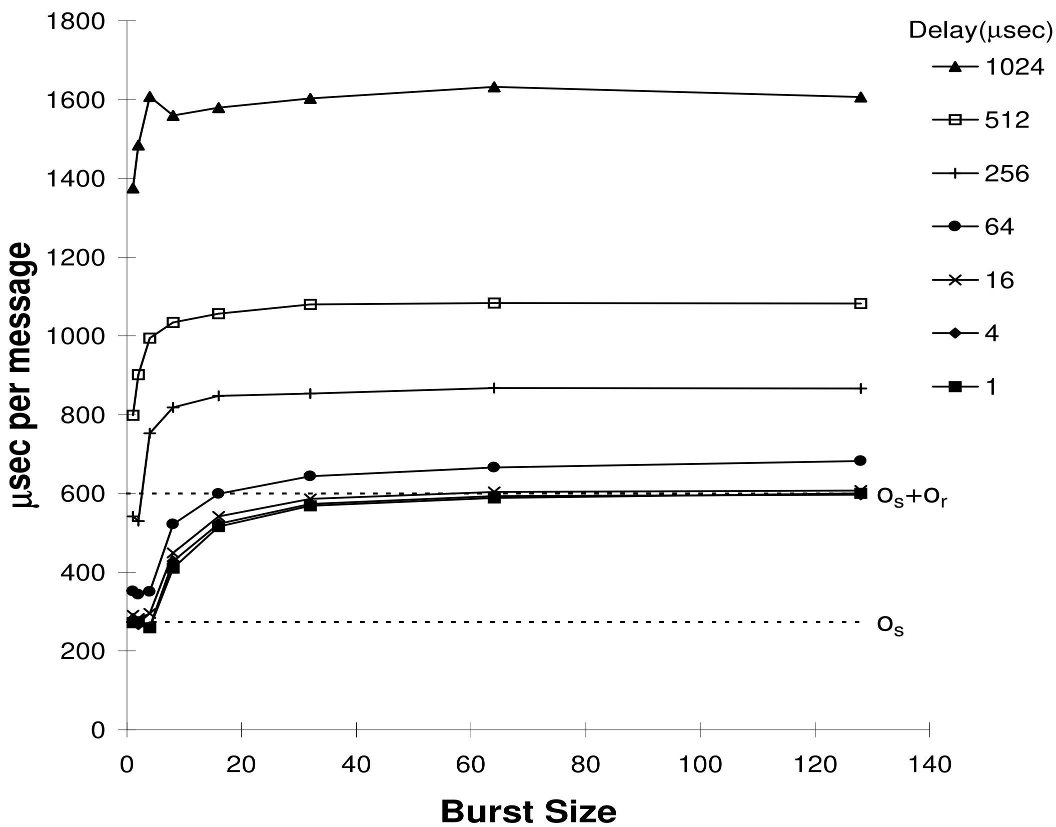

FIGURE 13. Asynchronous RPC signature

6.Acknowledgments

The US Advanced Research Projects Agency (F-30602-95-C-0014), the National Science Foundation (CDA-9401156), and the California Micro program have supported this project. NSF Presidential Faculty Fellowship supports David Culler. We would like to thank Sun Microcomputer, Intel, Lawrence Livermore Labs and UC Santa Barbara for providing equipment and support.

7.References

[1]T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation,” *Proc. of the 19th ISCA*, May 1992, pp 256-266.

[2]R. H. Saavedra-Barrera, *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, Ph.D. Thesis, U.C. Berkeley, Computer Science Division, February 1992.

[3]D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation,” *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[4]R. Martin, “HPAM: An Active Message Layer for a Network of HP Workstations,” *Proc. of Hot Interconnect II*, August, 1994.

[5]T. von Eicken, V. Avula, A. Basu, and Vinnet Buch, “Low-Latency Communication over ATM Networks using Active Messages,” *IEEE Micro*, February 1995, pp. 46-54.

[6]L.T. Liu and D. E. Culler, “An Evaluation of the Intel Paragon on Active Message Communication,” *Proc. of Intel Supercomputer User’s Group Conference*, June 1995. On-line as `http://www.cs.berkeley.edu/~ltliu/papers/isug95/isug_1.html`.

[7]M. Homewood and M. McLaren, “Meiko CS-2 Interconnect Elan-Elite Design,” *Proc. of Hot Interconnects*, August 1993.

[8]N. Boden et al., “Myrinet: A Gigabit-per-second Local Area Network,” *IEEE Micro*, February 1995, pp. 29-36.

[9]Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, June 1995.

[10]A. Beguelin, J. Dongarra, G.A. Geist, R. Manchek, K. Moore, and V. Sunderam, “A user’s guide to PVM: Parallel Virtual machine,” *Technical Report ORNL/TM-11826*, Oak Ridge National Laboratory, July 1991.

[11]D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, “Parallel Programming in Split-C,” *Proceedings of Supercomputing ’93*, November 1993.

[12]D. D. Clark, V. Jacobson, J. Romkey, H. Salwen. “An Analysis of TCP Processing Overhead,” *IEEE Communications Magazine*, June 1989, pp. 23-29.