A Comparative Study of Some Network Subsystem Organizations[*]

Dmitry V. Ponomarev and Kanad Ghose
Department of Computer Science
State University of New York, Binghamton, NY 13902–6000
{dima, ghose}@cs.binghamton.edu

Abstract

The impact of alternative network subsystem design for realizing low end–to–end latencies and high network throughput in a switched LAN are studied in detail through simulation. These alternatives include choices in the disposition of the network interface card (NIC), DMA priorities and OS services. Our simulation model captures the delays of OS services/software layers, message copying DMAs and, in addition, models non–network related traffic on the I/O and memory buses introduced by paging and on–chip cache misses. In a conventional setup, with the NIC placed on the I/O bus, we show that changing traffic priorities on the memory bus to speed up the transfers between the NIC and the DRAM has little impact on overall latency and network throughput as the offered network traffic increases. Improving the speed of the I/O bus produces some performance gains. These performance gains are shown to be quite limited until message demultiplexing capabilities are added to the NIC. The best performance comes from the use of dual–ported DRAMs, with a dedicated connection between the NIC and the added port.

1. Introduction

Significant advances have been made in the world of networking hardware over the recent years – these include high speed transmitters and receiver chips, cheaper, high speed media and connectors and high speed switches and routers. It is a well–known fact that the high speeds of these networking hardware do not translate into commensurate performance at the level of applications. These disparities have been well–documented in [1] and [18]. The reason for this disparity between the seemingly available performance and the actual or delivered performance can be traced to the complex interplay between the hardware and software related to networking, the operating system, the nature and functionality of the network interface card (NIC, network adapter), the hardware of the host system and, of course, on the characteristics of the application. The goal of this paper is to undertake a simulation based study of exactly some of these tradeoffs.

* This work is supported in part by the NSF through award No. CDA 9700828 and award No. STI–9413854

Let us first quickly review the message flow from an application running on one node to an application on another to illustrate where the various hardware and software components play a role in message passing. We assume that the NIC has very simple capabilities, in particular, the NIC is not capable of message demultiplexing – a fact that is true of virtually all NICs in use today. In conventional systems, the NIC is a device on the I/O bus, as shown in Figure 1. Sending and receiving messages thus involves data transfers on both the memory bus and the I/O bus to move data between the main memory and the NIC.

The NIC, as well as message buffers in the system have to be protected from the individual applications. Consequently, the data movement in message passing involve the crossing of protection domains. These cross domain transfers have to be co–ordinated by the operating system (OS) after performing the required authentication checks. These checks and the data movement are generally initiated through system calls. In the process of moving the message from the source application to the destination application, additional software delays are incurred due to buffer management and the protocol stack. Delays are also introduced in the physical transfer paths, such as the memory bus, the I/O bus and the hardware artifacts on the way (the NICs at both ends, propagation delays in the links and switches/routers etc.). Optimizing the performance of the network thus requires careful consideration of the hardware and software tradeoffs, the functionalities of the NICs as well as the consideration of the physical interface of the NIC, including the physical disposition of the NIC within the system. The various delays in the way indicate that improving the raw speed of the network is of little use unless changes are made in the design of the rest of the hardware and software components in the network subsystem.

The goal of this paper is to study the impact of various software and hardware innovations made to promote efficient networking on a comparative basis. We employ a simulation model, driven by synthetic traces to study the effect of the following changes within the network subsystem:

- The impact of enhancing the DMA capabilities within the NIC
- The impact of changing priorities of DMA traffic on the I/O bus.

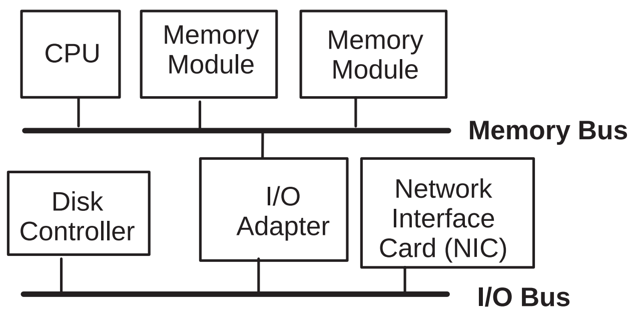

Figure 1. The organization of a conventional network interface

- The impact of adding more functionality to the NIC itself ("smart NICs")
- The impact of changes made to the network subsystem software, including modifications to the OS kernel.
- The impact of alternative placements of the NIC within a system that promotes efficient transfers between the NIC and the memory. These include a dedicated bus (or connection) between the NIC and the memory and the use of separate buses for the NIC and other I/O devices.

Although our main focus is to study the impact of these changes on the end–to–end message passing latencies, we also analyze their influence on the overall performance of the system, in particular network throughput, CPU utilization and the utilization of the buses.
A recent study was undertaken to analyze the impact of network subsystem design alternatives in fine–grain message transport on the overall performance of parallel message passing scientific applications [15]. Our study focuses on the impact of the network subsystem design choices on the end–to–end latency and other performance metrics for general network applications that use larger messages and also rely on the use of higher–level APIs (such as TCP/IP). More importantly, we consider delays in the software message passing layers and traffic introduced on the memory bus due to cache misses, as well as paging traffic on the memory and I/O buses. *Our goal is not to propose new solutions for speeding up network interfaces, but rather to do a comparative study of existing techniques under a common framework, taking into account both hardware and software latencies.*
The rest of the paper is organized as follows. In Section 2, we describe the reasons for the various delay components in traditional message passing. In Section 3, we review hardware and software techniques for reducing end–to–end delays followed by the description of the alternative placements of the NIC within the host to cut down the latency of message passing. Section 4 describes our simulation strategy. The results are presented and discussed in Section 5, followed by our conclusions in Section 6.

2. Message passing in traditional systems

We now examine the sources of delays in message passing from one application to another on a different node. We assume a very traditional NIC and a traditional software layers.

2.1 Message sending

The sending application typically calls a library routine (such as a write to a socket or a "send"), specifying a channel or a socket id and the variable containing the message. This call eventually makes a system call that copies the message from the variable into a kernel level buffer after performing an authentication check. The authentication check verifies that the application has the appropriate rights to send the message on the specified channel or socket, using the process id of the application maintained within kernel–level data structures. Control returns to the application after the message has been copied into kernel level buffers. The protocol layers add appropriate headers to the message, form a message descriptor for the message in a kernel level queue of descriptors for outgoing messages. From the kernel level buffer the protocol layers, through the OS (or some appropriate daemon process) initiate the DMA of the message directly (or indirectly) into raw buffers within the NIC, from where it is eventually transmitted on its way to the destination.
The delays in sending a message out from the source node thus consist of the following components:

a) System call overhead for performing the authentication check and copying the message from the variable in the user's address space to the system level buffers.
b) Protocol stack delays.
c) Delays in DMA–ing the message from the system–level buffers to the buffer within the NIC.
d) Potential waiting in the buffers within the kernel and the NIC and delays introduced due to buffer management overhead. This is especially acute for the uniprocessor systems with single–threaded kernel.

We will see later that techniques exist for cutting down most of these delay components.

2.2 Message transport

After the outgoing message has been transmitted, it may go across several links and switches on the way. The potential delays in this part are waiting delays within the switches, propagation delays within the switch and propagation delays in the media.

2.3 Receiving a message

On receiving a message, the NIC DMAs this message into a global system level buffer, since it cannot perform any message demultiplexing and since it is generally not possible for the systems software to read the header of the message on the NIC buffer. If the NIC is not a bus master, this generally involves the generation of an interrupt from the NIC on message arrival. The service of this interrupt results in the initiation of a DMA transfer to the global memory buffer from the raw buffer within the NIC. From the global buffer, system–level routines (or daemons) perform the necessary message demultiplexing, copying messages to appropriate per–channel or socket buffers. An application executes a primitive (such as "receive" or a socket read) to have the message copied from the per–channel or socket buffers into appropriate variables in the application's address space. This library call again results in a system call that performs authentication checks and the atomic copying of the message into the variable.

The delays in receiving a message thus consist of the following components:

a) Time needed for message demultiplexing
b) Delay in DMA–ing the message from the NIC to the global system–level buffer.
c) Protocol stack delays
d) Copying overhead between various buffers.
e) Delays in waiting in various buffers, including the delays incurred in buffer management.
f) The system call overhead for a "receive" or socket read primitive.

Again, we will look at some solutions that have been proposed and implemented for cutting down on one or more of these delay components.

3. Reducing end–to–end delays

A fair number of solutions have been proposed over the recent years to close the difference between the available network hardware performance and the application level performance. These include the following:

3. 1. Innovations within the NIC

These solutions enhance the functional capabilities of the NIC in an effort to implement functions that are traditionally implemented within the OS and the communication libraries. Examples of solutions in this category are:

- Bus master NIC, multiple DMA channels: here the NIC is the master of the I/O bus, allowing it to DMA in an incoming packet into the global system level buffer when a message comes in. Another relatively simple innovation is to support multiple, multiplexed DMA transfers between the NIC and the memory. Yet another improvement to reduce buffer management overhead is to build in gather–scatter support within the NIC to allow DMA transfers between the NIC and a series of non–contiguous memory locations. Most modern high–end NICs have these capabilities.
- Parallelism within the NIC: Concurrent engines for DMA transfers and/or concurrent transmit/receive engines within the NIC, as implemented in the Afterburner NIC [4] and [7], can also improve the raw performance of the NIC and help in cutting down on the end–to–end latency.
- Partial hardware support for message demultiplexing: Here, the NIC allows the network–related software components to read the header of the incoming message and let software set up DMA to an appropriate buffer in memory, thus resulting in a hardware–assisted, but software controlled message demultiplexing, as suggested for the x–Chip [3].
- Message demultiplexing by the NIC: Here, facilities are incorporated within the NIC to perform message demultiplexing, as in the U–Net project [18] or the Illinois Fast message implementation on a Lanai Myrinet card [16], the MAGIC chip for FLASH [9] and several other systems along similar lines [7].
- Snooping logic within the NIC to pick up data being written to a memory area designated as a send buffer and eventually transmitting that data, as implemented in the SHRIMP prototype [2]. The DEC Memory channel prototype [8] uses a similar mechanism that implements reflective memory to propagate the writes to a memory area designated as the send buffer to a buffer in a memory area on a remote node that serves as the receive buffer.
- Enhanced flow control support within the NIC, as in the Telegraphos prototype [11].
- Hardware support for barrier synchronization within the NIC, as implemented in the ParaStation prototype [19] and [7].
- Logic, circuit design and other implementation techniques for speeding up the NIC, including the use of micropipelining, highly integrated transmitter and receiver chips and fast implementation technologies, such as GaAs.

3.2. OS modifications and innovations

In a traditional system, the role of the OS in the context of networking is to provide protected access to the shared NIC and associated networking software, to provide buffering facilities and implement message multiplexing and demultiplexing.

Examples of OS/Protocol layer improvements and modifications made in this respect are:

- Reducing the number of copying steps in the message flow path between the raw buffer within the NIC and the application level buffer: Several solutions under this category have been used to reduce the end–to–end message passing latency, namely sharing virtual pages, page remapping [6], copy–on–write semantics and fast buffers (fbufs) mechanism [5].

- Reducing the time needed for invoking OS services: Protected message passing requires authentication checks, in the least, when a message send is requested. The usual overhead of making an explicit system call can be avoided by streamlining the system call interface. One approach, as used in U–Net [18], is to use trap instructions that cause handcrafted system code to be executed. GLUnix, developed for the NOW project at Berkeley [1] provides similar facilities. Another approach, as used and implemented in [7], for fast message passing in a Solaris environment, is to force an exception and modify the first–level interrupt handler in the OS to perform an authentication check based on the known *physical address* of the system–supplied trusted code implementing the send. This, in effect, implements a very low overhead system call for performing the authentication check.
- Reducing delays in the protocol stack: Any end–to–end protocol operating over a high–speed network must be able to rapidly process and update state information in order to keep up with the higher network speeds. This requirement has resulted in light–weight protocols, state–exchange protocols, rate–controlled protocols, NACK based protocols and source / receiver controlled protocols into our proposed courses.

3.3. Application–level techniques

A variety of techniques have been used to reduce the network latency at the application level. The concept of active messages [14] is an example. Each incoming message specifies the address of an appropriate handler for the message, thus saving critical time in locating such a handler. Other application level techniques to reduce message passing latencies include application–level buffer management and alternative semantics for message passing.

3.4. Alternative placements of the NIC

All of the techniques discussed above are generally oblivious to the physical disposition of the NIC within the system. In conventional systems, the NIC is located on the I/O bus, as shown in Figure 1. For this particular placement, data transfers between the NIC and the main memory have to go through both the I/O bus and the memory bus. Since the I/O bus generally operates at a lower frequency than the memory bus, this placement can have an adverse impact on the message passing latency.
An alternative will be to place the NIC on the memory bus. This, however, may not be an acceptable solution, since the network traffic introduced on the memory bus can adversely impact the miss handling time for the on–chip caches, resulting in poor CPU (and overall) performance. A solution that avoids this problem is depicted in Figure 2. Here, the NIC is placed on a dedicated bus (called the "NIC bus"), operating at (or in excess) of the speed of the I/O bus. Dual–ported memory modules are connected to both the memory bus and the NIC bus, removing the need for explicit transfers between these two buses. The dual ported memory modules, of course, have to be large enough to implement all of the network buffers. Clearly, for this arrangement to make sense, a page remapping scheme (or the fbufs) scheme have to be used to avoid copying across the boundary of the domains for the kernel and the applications. Note that this scheme still requires the message to be DMAed between the dual–ported buffer RAMs and the raw buffer on the NIC. The resulting scheme is thus not a true zero–copy interface.

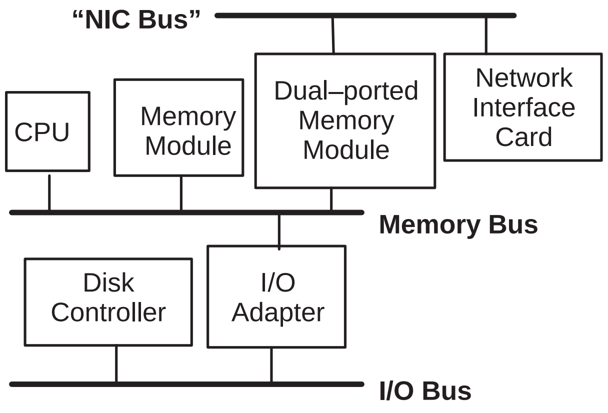

Figure 2. The organization of a network interface with the NIC on a dedicated bus

A true zero copy interface requires that the raw buffers on the NIC be visible directly to the application. Such an interface has been implemented as part of the MINI project at Sarnoff labs [13]. Here dual–ported RAMs are used on the NIC and SIMM extenders are used to connect one port on these RAMs to the normal RAM (SIMM) sockets on the memory bus. The existing MMU protection mechanism and page remapping is used to realize end–to–end latencies of a few microseconds. In this scheme, a send amounts to a write into the buffer space for the sending channel on the RAM on the NIC, while receives amount to a read from the same RAM.

4. Simulation methodology

To model the various delays associated with message passing, including hardware delays, queueing delays and delays in the software layers (specifically, delays for OS services and copying delays), we constructed a simulator, that was primarily driven with two synthetic inputs. One was the offered network load, which generated traffic on the links, the I/O bus and the memory bus. The other input to the simulator was traffic on the memory bus generated due to cache misses, which in turn could result in page faults and trigger paging traffic on the memory and I/O buses.
To generate the interarrival times of the network packets and cache miss requests, we used the exponentially distributed functions with the average determined by the

average packet size, network throughput and the cache hit ratio under consideration. For simulations using variable message sizes, the packet sizes were also considered to be exponentially distributed with upper and lower bounds of 1500 and 64 bytes respectively and the given average.

A limited form of flow control was incorporated into the simulator to prevent a node from sending further packets until the appropriate number of acknowledgments has been received from the destination. This roughly mimics the flow controls put into place in a real networking system.

The end–to–end latencies were calculated by keeping track of the latency of each individual packet within the system. Latencies were averaged based on the total number of packets received. On the I/O bus, paging traffic was always given lower priority to favor the reduction of the end–to–end latency while the priorities of the incoming and outgoing network traffic were rotated for the use of the memory and I/O bus to avoid any starvation at high data rates. Different combinations of priority assignments for the packets on the memory bus have been studied as explained further. We also assumed that all necessary checksums required by the protocol layers were computed during the message transfer over the memory bus and did not thus require additional copying.

We confined our studies to a 8–node system that is connected using switched interconnection and point–to–point links as shown in Figure 3. The 8x8 switch has a total capacity of 8Gbps that allows to maintain four 2Gbits/sec simultaneous point–to–point connections with the switching latency of 12 microsec. per packet.

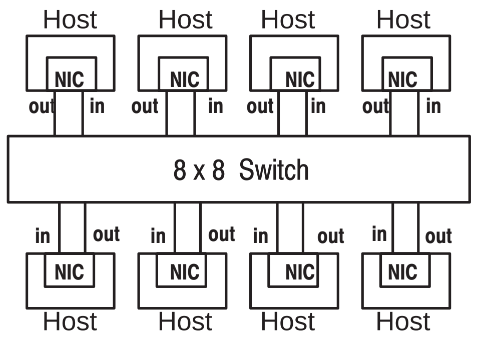

Figure 3. The Configuration of the Simulated System

The processing nodes were assumed to be 200 MHz superscalar CPUs capable of sustaining 2.5 issues per cycle on the average. We considered 128 bits wide memory bus, operating at a clock rate of 100 MHz. The I/O bus was assumed to be 64 bits wide and two I/O bus clock rates were studied: one was 33 MHz. and the other was 66 MHz. The DMA burst size for both the memory and I/O buses was assumed as 32 bytes. We assumed a cache line size of 32 bytes, allowing a cache miss to be handled by a single burst on the memory bus. An optimistic cache hit rate of 99% was considered, but we also studied the impact of lower cache hit rates, as reported later. The page fault rate was assumed to be one per 500,000 memory references, which is on the higher side based on what is reported in [10], to deliberately increase the paging traffic on the memory and I/O buses. The page size was assumed as 8 Kbytes. We did not explicitly model the impact of caching message headers and performance losses due to misaligned message headers. (A refined model that does this is under development.)

We now describe the software delays assumed in the simulations. The software latencies for the OS and protocol layer delays, when the traditional NIC (which is incapable of message demultiplexing) was in use were assumed as follows, based on the fast authentication check technique described in [7], and on the use of appropriate OS changes to allow fast handler invocations, as described in [17]:

- Software delays during a ‘send’, *excluding* the copy overhead (which is dependent on the data size and characteristics of the DMA hardware, which are described separately) is 6 μsecs. (for the 200 MHz. superscalar CPUs assumed in the simulations.) This is only the time to perform authentication checks, buffer management, system call and initiation of the DMAs for the copies. The actual time needed to copy the message was dependent on the message size and any waiting delays – these were accounted for explicitly in the simulation as DMA times (memory–to–memory copy, DMA between NIC and memory).
- Software delays during a ‘receive’ is 9 μsecs., which again excludes the data size dependent copy overhead. This delay is essentially what is needed to serve the interrupt from the NIC when a message is received, delays for buffer management and initiating any copy. Actual copying delays were accounted for separately.

We implicitly assumed that address translations required by the NIC were cached in the TLB (and translations performed automatically, as in the SBUS protocols [Mason 94]. We also assumed that all buffer pages are pinned in memory, so that they are not swapped out. The paging traffic on the memory bus and I/O bus that was simulated thus do not involve any message buffer pages.

We also studied the impact of using a modified handler for authentication check with a smaller delay, in conjunction with a NIC capable of performing message demultiplexing. In this case, we assumed that the NIC could DMA in or out of buffers remapped to the user’s address space, resulting in a single–copy interface. With this configuration, no additional software delays are incurred in depositing an incoming message into the receiver’s buffer. The only thing that the ‘receive’ primitive does is to initiate the copying of the message from the buffer into a variable – this time was assumed as 0.1 μ secs., based on the timings for our SNOW prototype, scaled to a 200 MHz. clock rate. (No access checks are required on the receiving side, since the NIC has done the demultiplexing on its own.) On the sending side, the software delay comes only from the software components required for authentication checking and initiating the DMA into the NIC on a ‘send’. Based on our experience with the SNOW prototype, we assumed this delay as 1.6 μ secs. (using the fast authentication checking technique described in [7], with appropriate prescaling for the CPU clock rates).

The simulation studies reported in this paper were confined to the four different designs for the networking subsystem:

- **Base system (B):** We assume that a traditional NIC incapable of performing message demultiplexing is in use. We also assume that this NIC has scatter–gather hardware but can support only one active DMA channel at any time (given that only one set of DMA registers is incorporated into the NIC).
- **Base system augmented with multiple DMA channels (BMDC):** This is identical to the base system, but multiple DMA channels are supported by the NIC in either direction. We assume that the NIC supports up to 8 DMA channels in either direction. The software associated with the NIC uses two copies in moving the message between the NIC and the user's address space.
- **Smart NIC on I/O bus (SNIO):** This is a design that uses a "smart" NIC capable of performing message demultiplexing in addition to the capabilities of the BMDC configuration. We assume that the hardware/firmware/software delay on the NIC for performing message demultiplexing is a flat 2 μsecs. The hardware latency for message demultiplexing within the NIC in our SNOW prototype [7] is within this limit. The associated software allows the NIC to DMA messages between user–space buffers.
- **Smart NIC on dedicated bus (SNDB):** This is a system that uses a smart NIC as described above but uses a dedicated bus to connect the NIC to dual–ported RAMs, which are also connected on the memory bus (Figure 2).

For the configurations above, we varied the average message size, the network offered load (that is equivalent to network throughput in the absence of packet losses), the priority of the network traffic vs. the traffic generated by the cache misses, cache hit rate, and the speed of the I/O bus to study their effect on the end–to–end latency, also noting the effects on the CPU utilization and the utilizations of the memory and I/O buses.

Most of the time, we compare SNIO and BMDC designs. This is because most NICs support multiple DMA channels, while placing the NIC on a dedicated bus, although being one of the best and radical solutions, is quite expensive by all means. Unless otherwise indicated, the frequency of the I/O bus was assumed to be 33 MHz and the D–cache hit ratio – 99%.

5. Results

Figure 4 shows that not only the end–to–end latency improves significantly when SNIO design is in use, but the maximum sustainable throughput is substantially higher than in both B and BMDC systems. One can also see that for the SNDB design, the offered load practically does not effect the end–to–end latency almost in the absence of queueing delays.

Figure 5 depicts different behavior of the BMDC and SNIO designs for medium and large packet sizes of 320, 640 and 960 bytes. In the case of a SNIO, the smaller packets have lower latency (under the same offered load), because the I/O bus transfer time (that is accounted twice in a message–passing) is comparable with the delays within the NIC and delays in the software layers. For the BMDC design, however, per–packet OS service time is a dominating factor. The same trend can be seen for the smaller packets for the SNIO system. Figure 6 demonstrates that the faster I/O bus does not help to reduce the end–to–end latency in the case of a BMDC configuration (slight improvement comes only because of a faster transfers over the I/O bus itself), since the major bottleneck really lies in the presence of the OS calls, but it can significantly cut down the delays in the case of a SNIO design. In fact, slow I/O bus may reach the saturation point under high loads in the SNIO design, as can be seen in the Figure 7.

As Figure 8 shows, the network load may influence the utilization of the CPU since CPU–memory traffic experiences more contention for the use of the memory bus that leads to the longer miss handling times. A drop of several percentage points in the CPU utilization is observed as the network traffic increases, and the drop rate is more than doubles when cache hit ratio decreases from 99 to 90 percent.

Figure 9 illustrates the effect of changing the priorities of network traffic and CPU–memory traffic. There is a difference of several percentage points in the CPU utilization and this difference is bigger for the BMDC system since every network packet has to traverse the memory bus twice and it thus creates more delays for the cache miss packets when the higher priority is statically assigned to the network traffic.

The end–to–end latencies for the two priority allocation schemes have also been measured but the difference of the observed results never exceeded 0.1 microsec. Indeed, for the long packets, even several collisions on the memory bus with the higher–priority cache miss packets, that are short and are served very quickly, add just several more cycles to the latency that is largely dominated by the OS service time, NIC delays, I/O bus transfer time, wire transfer time and switching delay.

Thus, especially for the long packets, statically assigning the higher priority to the CPU–memory packets gains several percents of the CPU utilization and practically does not change the end–to–end latency.

6. Conclusions

This paper presents the comprehensive review of hardware and software modifications and innovations (introduced elsewhere) made to support a high bandwidth and low latency message passing over the network of workstations. We also present and discuss the results of the simulation modeling employed to study the impact of some of these

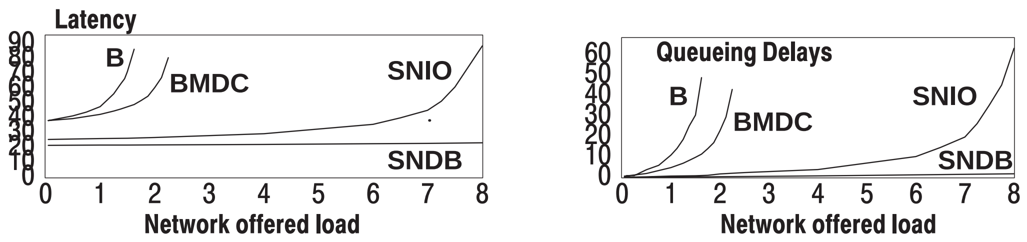

Figure 4. Total end–to–end latency and queueing delays (μsec) vs. network offered load (Gbits/sec) . Average packet size – 640 bytes

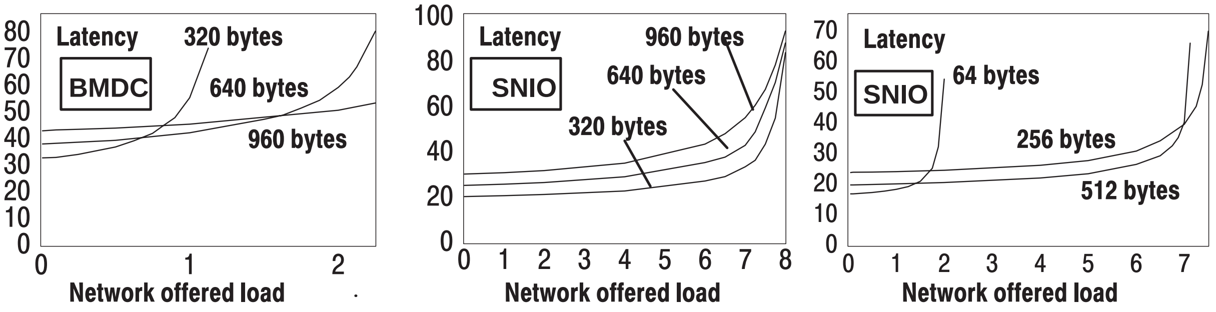

Figure 5. Latency (μsec) vs. network offered load (Gbits/sec) for different packet sizes.

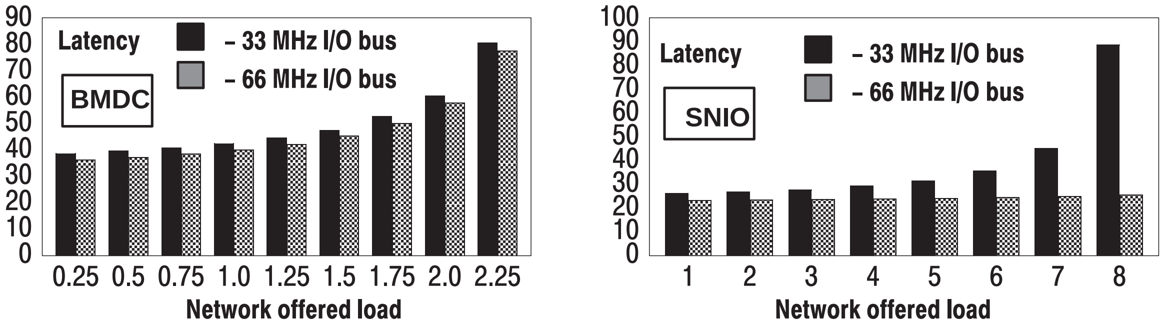

Figure 6. Latency (microsec) vs. network offered load (Gbits/sec) for different I/O bus speeds. Average packet size is 640 bytes

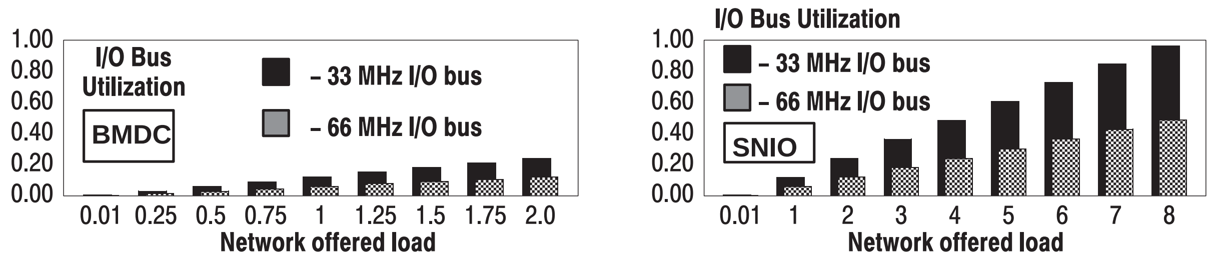

Figure 7. I/O bus utilization vs. network offered load . Average packet size is 640 bytes

techniques. The various delays associated with message passing including hardware delays, queueing delays and delays in the software layers have been accounted for in the simulator for various networking subsystem organizations. In particular, we analyzed the impact of the average packet sizes, traffic priorities, cache miss rate and the speed of the I/O bus on the message passing latency, CPU utilization and the utilization of the buses. Our results show that for a conventional placement of the NIC on the I/O bus, changing traffic priorities on the memory bus to speed up the transfers between the NIC and the DRAM has little impact on overall latency and network throughput as the offered network traffic increases. Improving the speed of the I/O bus produces some performance gains. These performance gains are shown to be quite limited until message demultiplexing capabilities are added to the NIC. Significant performance gains are realized with the use of dual ported DRAM banks, and the use of a dedicated connection between the NIC and these banks.

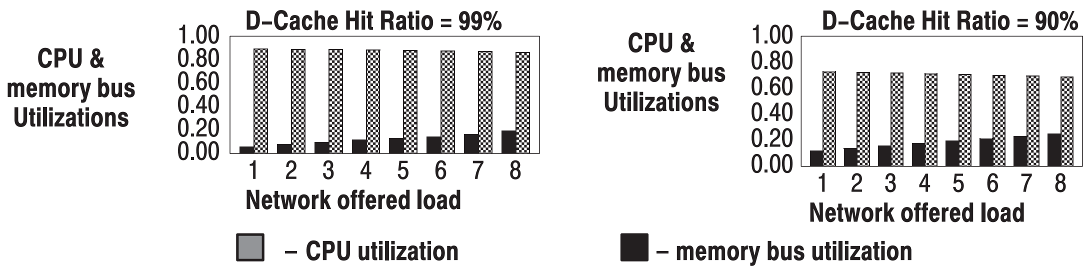

Figure 8. CPU and memory bus utilization vs. offered load for the SNIO system for D–cache hit ratios of 99% and 90%. Average packet length is 640 bytes.

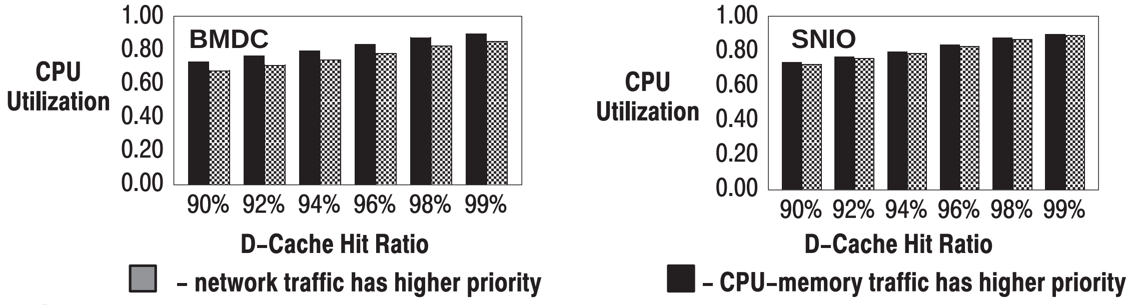

Figure 9. CPU utilization vs. D–cache hit ratio for the BMDC and SNIO configurations. Average packet length is 640 bytes. The network throughput is 2 Gbits/sec

References:

[1] Anderson, T.E. et al., "A Case for NOW (Network of Workstations)" *IEEE Micro*, Vol. 15, No. 1, 1995, pp. 54–64.

[2] Blumrich, M. A., "Virtual Memory–Mapped Network Interfaces", *IEEE Micro*, Vol. 15, No. 1, 1995, pp. 21–28.

[3] Balley, M.L., Pagels, M.A. and Peterson L.L., "The x–Chip: An Experiment In Hardware Multiplexing", IEEE Workshop on the Arch. and Impl. of High Perf. Comm. Subsystems, Feb., 1992.

[4] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A. and Lumley, J., "Afterburner", IEEE Network Magazine, Vol 7, No. 4., July 1993, pp. 36–43.

[5] Druschel, P. and Peterson, L., "Fbufs: A High–Bandwidth Cross–Domain Transfer Facility", in Proc. of the 14th Sym. on Operating Systems Principle, December 1993, pp. 189–202.

[6] Druschel, P., Abbott, M.B., Pagels, M.A. and Petersen, L.L., "Network Subsystem Design", *IEEE Network*, July 1993, pp. 8–17.

[7] Ghose, K., Melnick, S., Gaska, T., Goldberg, S., Jayendran, A. and Stien, B., "The Implementation of Low Latency Communication Primitives in the SNOW Prototype", in Proc. of the 26–th. Int'l. Conference on Parallel Processing (ICPP), 1997, pp.462–469.

[8] Gillet, R. and Kaufmann, R.,"Using the Memory Channel Network", IEEE Micro, Vol.17, No.1, 1997, pp.19–25.

[9] Heinlein John J. et al., "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor", *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, October 1994.

[10] Hennessy, J. and Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1995.

[11] Katevenis, M., Markatos E., Kalokairinos G. and Dollas A., "Telegraphos: A Substrate for High Performance Computing on Workstation Clusters", Journal of Parallel and Distributed Computing, 43(2), June 1997, pp. 94–108, Academic Press.

[12] Mason, Susan A., *SBus Handbook*, SunSoft Press, 1994.

[13] Minnich, R. et al., "The Memory–Integrated Network Interface", *IEEE Micro*, Vol. 15, No. 1, pp. 11–20.

[14] Mainwaring A. and Culler D. "Active Message Application Programming Interface and Communication Subsystem Organization, Tech.Report CSD–96–918, Univ. of California, Berkeley, October, 1996.

[15] Mukherjee S.S. and Hill, M.D.,"The Impact of Data Transfer and Buffering Alternatives on Network Interface Design", *Proc. of the 4th Int'l Symposium on High–Performance Computer Architecture*, February, 1998.

[16] Pakin, S., Karamcheti, V. and Chien, A.,"Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs", IEEE Concurrency, Vol.5, No.2, 1997, pp.60–73.

[17] Thekkath, C.A. and Levy, H.M.,"Hardware and Software Support for Efficient Exception Handling", *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, October 1994, pp.110–119.

[18] vonEicken, T., Basu, A., Buch, V. and Vogels, W., "U–net: A User–Level Network Interface for Parallel and Distributed Computing". in Proc. of 15th Operating System Principles, 1995.

[19] Blum, J. M., Warschko, T.M. and Tichy, W. F., "PULC: Parastation User Level Communication. Design and Evaluation", in PC–NOW workshop, 12th Int'l Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing, Orlando, 1998.