Enhancing Distributed Systems with Low-Latency Networking

S. L. Pope, S. J. Hodges, G. E. Mapp, D. E. Roberts & A. Hopper[†]
Olivetti and Oracle Research Laboratory
24a Trumpington Street,
Cambridge,
England.
[†] Cambridge University Engineering Department
Trumpington Street,
Cambridge,
England.
{spope, sjhodges, gmapp, droberts, ahopper}@orl.co.uk

May 27, 1998

Abstract

Recently several network technologies which support user-level communication between processes using a shared-memory interface have become available [4, 7]. These technologies offer very low latency, high bandwidth communication by eliminating the need for software protocol stacks. Whilst there has been much research on the use of such networks in the context of parallel computing [5, 6, 13], relatively little work has been done on their suitability for distributed applications. This paper describes the work undertaken to integrate the Scalable Coherent Interface (SCI) interconnect with the standard NFS server and a CORBA 2.0 compliant ORB over Linux. It is shown that impressive performance increases can be achieved without modification to either the operating system or the distributed application.

Keywords: SCI, CORBA, NFS.

1 Introduction

The computing environment at the Olivetti & Oracle Research Laboratory (ORL) is highly heterogenous, supporting our work in multimedia and sensor driven systems. We require a high bandwidth and low latency network infra-structure for applications, data and hardware. We were keen to explore opportunities for improving the performance of our existing distributed applications through the use of high speed interconnect technology and believed that an

improvement in latency could have a direct effect on many of our existing applications. One obvious example is NFS where many operations, such as traversing a directory structure are limited by the latency of the communication between client and server. We also identified CORBA as a piece of our infra-structure where there was scope for a performance improvement which would be passed on to many of our distributed applications.

Given this starting point, we first decided to examine the existing interconnects which were available for our system. We were looking for an interconnect which offered a bandwidth which was greater than the usable IO bandwidth of our workstations[1], but which would scale to well over a Gbps and which was sufficiently mature to operate within our heterogeneous environment of workstations. In our lab, this means PCs with Linux or NT and Sun workstations and servers.

We chose SCI from Dolphin Interconnect Solutions as an attractive technology which met the above criteria and which also offered a memory mapped paradigm for access which we thought could lead to very low latencies in our system. Dolphin were generous enough to provide access to the hardware interface and drivers, a prerequisite for any serious development effort.

This paper will describe the work undertaken at ORL to integrate the SCI interconnect with the standard Linux NFS server and our in house CORBA 2.0 compliant ORB, omniORB2. Other interconnect technologies [5, 1] would have been equally valid for our experiments. However it is outside the scope of this paper to provide a comparison between the different technologies.

We show in this paper that through attention to the software overheads and communication paradigms at all levels of the system, it is possible to obtain significant performance improvements for existing distributed applications by running over a low-latency interconnect without heavy modifications to either operating systems or the applications.

2 Dolphin PCI-SCI card

Figure 1 illustrates how SCI cards may be used to interconnect PCI systems. The board allows a processor in one node to directly access the memory of another connected node. The processor achieves this by executing ordinary **load store** instructions addressing the part of its physical address space that is allocated to the SCI interface on the PCI bus. Different parts of the local physical address space may be mapped to different remote memory nodes.

When the PCI-SCI interface receives a read or write request from the PCI bus, it transmits a SCI request packet on the interconnect. The 32 bit PCI address is mapped into a 64 bit SCI address, where the 16 most significant bits contain the target node identity. At its destination, the SCI packet is checked for compliance with address restrictions. If the destination address and source node are legal, the received request is then executed as a PCI transaction, using a mapping from the 48 lower bits of the SCI address to a 32 bit PCI address.

[1] Around 300 Mbps for a Pentium Pro 200Mhz FX

After completing the request, PCI-SCI interface of the target node generates an SCI response packet containing the transfer status and returns it to the requester. At the requesting side, this SCI packet is translated into a response on the PCI bus.

By manipulating the operating system's virtual page tables, it is possible for a user-level process to communicate with a remote machine without calling a kernel-level device driver. Using this hardware, a single user-level dword write to remote memory has been timed at $2.5\mu s$ or $12.8Mbps$. Clearly in order to achieve high throughputs it is necessary to transfer N dwords in a time much less than $2.5N\mu s$. To solve this problem the Dolphin PCI-SCI bridge implements a *streaming* mechanism to achieve high-throughput whilst maintaining low-latency. This is explained in more detail in [11]. However, because a local PCI bus must be locked whilst a remote **load** instruction is executed, the throughput achievable when reading from remote memory is correspondingly less than when writing.

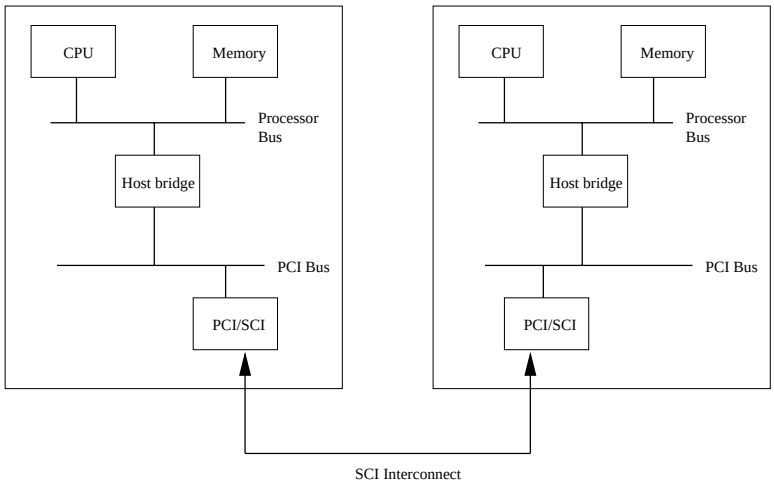

Figure 1: Two PCI systems connected using SCI.

3 Driver software

A Linux driver was written for the SCI cards based on Dolphin's own Windows NT driver. This mainly involved writing routines for manipulating the Linux virtual-memory tables, to allow a user-level process to map the SCI aperture into its address space. Several additional functions were implemented to facilitate experimentation including: an *event channel* to signal and wakeup a blocked

process on a remote node and *write-gathering*[2] on certain portions of the address space for performance on Pentium Pro platforms.

Once a shared memory channel has been set up, there is no software involved in the transportation of raw data, hence the performance of both our Linux driver and Dolphin's Windows NT driver, when write-gathering was disabled, were identical. All of the results which follow were taken on 200MHz Pentium Pro computers with FX440 PCI chipsets and running the Linux 2.0.30 kernel. These were interconnected by a point-to-point SCI network. The raw write performance measured through the interconnect was $220Mbps$ without write-gathering and $520Mbps$ with write-gathering enabled.

During the development of the driver various receiver side blocking strategies were investigated. Damianakis *et al* argue in [3] that a hybrid *spin-then-block* mechanism offers the best overall performance in a wide variety of situations. Our experimentation suggested that when *slow interrupts*[3] are used it is fairer to simply block pending an interrupt. Because of the low overhead of the Linux scheduler this also provides substantial performance benefits in a loaded system: no CPU time is wasted spinning, and in the majority of cases the CPU is immediately returned to the process by the scheduler. The results of these experiments are outside of the scope of this paper and will be reported elsewhere.

4 Communication abstractions

Although the hardware manages the transportation of data through the shared memory channel, it is necessary to ensure that synchronisation is maintained between the applications at the end points of the channel. Effective management of the shared memory is the key to achieving good performance and different communication abstractions suit different applications. This section describes the two abstractions we implemented.

The first is a simple locking mechanism over a shared memory segment. This was found to be sufficient for the omniORB2 application and was implemented as a user level library. The second is a more elaborate shared circular buffer abstraction. This was implemented as both a kernel and user level BSD socket protocol and was able to support the full semantics of the BSD socket.

4.1 Locked shared memory

The locked shared memory abstraction is perhaps the simplest means of managing a shared memory buffer, forming a pair of uni-directional channels. As shown in Figure 2, at the point that a sender wishes to transmit data on a

[2]Where successive discontiguous writes are amalgamated by the host PCI bridge into a contiguous block of given size before bursting the writes to the SCI card.

[3]Linux offers two types of interrupts: fast and slow. The main difference is that after a slow interrupt has been handled the scheduler is invoked, thus allowing a process which was added to the run-queue during the interrupt routine to gain immediate access to the CPU. By contrast, a fast interrupt does not call the scheduler and therefore a newly unblocked process may have to wait a full scheduling quantum before it can progress.

connection, the shared memory library acquires the Tx semaphore (1), copies the data into the shared memory segment (2), and releases the Rx semaphore (3). Assuming the receiver had been blocked on the Rx semaphore, it can then acquire the Rx semaphore (4), copy the data out of the buffer (5) and release the Tx semaphore (6). Note that the buffer abstraction was structured for performance reasons such that reads are always made from local memory.

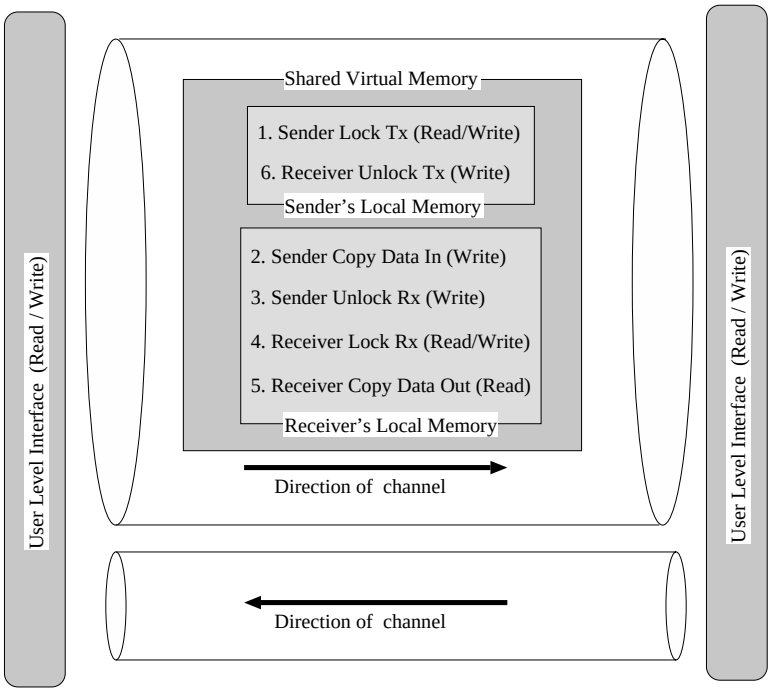

Figure 2: A Bi-Directional Channel Using a Locked Shared Memory Buffer.

This scheme will not effectively support a fully asynchronous bi-directional link. A sender may not proceed until the previous send has been read by the receiver. This is not a problem given the asymmetry of the communication protocol used by an application such as omniORB2, but would result in deadlock for other applications. The circular buffer socket abstraction described next avoids this problem.

4.2 SCI Sockets

Our locked shared memory communication abstraction was extended to a managed circular buffer. This was intended to enable fully asynchronous communication and streaming of data. The circular buffer abstraction was sufficiently powerful to support a BSD socket protocol and was necessary for our support for NFS over SCI.

The SCI socket library maps each socket onto a shared circular buffer, with access controlled using Peterson's mutual exclusion algorithm [15]. A shared buffer has a number of event channels which are used to signal different events associated with the shared buffer. The first channel is used to signal to the receiver that a sender has put data into an empty shared buffer. When this occurs, the receiver is woken up if it was blocked waiting for data to arrive or if the server was blocked on a select call, it is signalled that data was now available on this socket. The second channel is used at connection time to signal that a remote end has mapped in a local buffer, i.e. a connection is being established. This is needed to support the *select* call for listening sockets.

The design of the circular buffer also addressed the asymmetric performance for reading and writing which is characteristic of SCI. Each connection has an associated socket pair, one socket for reading and one for writing. The *read socket* is mapped to the local shared buffer while the *write socket* is mapped onto the remote buffer at the other end of the connection. Thus while socket writes were done remotely, any socket reads were local to the machine avoiding slow remote reads altogether. The *read socket* is created when the user issues a *socket* call. The *write socket* is created when a connection is made. However, the user makes all socket library calls with the *read socket* and is not aware of the *write socket* which is managed entirely by the SCI socket library. Figure 3 shows a connection between two endpoints using SCI sockets.

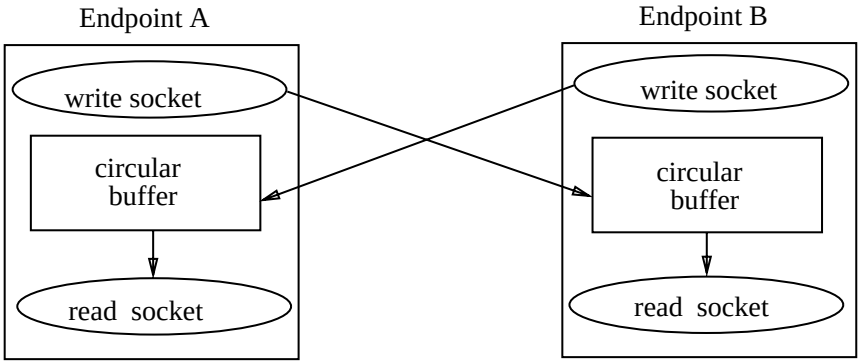

Figure 3: A connection between endpoints A and B using the SCI Socket Library

5 NFS over SCI

The NFS protocol [10] is an example of a mature distributed application which is part of our general infra-structure. Many of the common NFS operations such as those needed when traversing a directory structure are limited by the latency of the communication between the client and server and so we expected a performance improvement of NFS over the low latency SCI interconnect.

The NFS server is single threaded and achieves multiplexing through use of the BSD *select* system call. This represents a markedly different approach in engineering terms to that of omniORB2 (described in section 6) and required a network abstraction which supported the complete BSD socket interface.

To support NFS, we extended the standard SunRPC mechanism to support SCI connections. This involved writing files to support SCI sockets in the client and server stub libraries, thus allowing the NFS Server to support SCI connections as well as traditional TCP and UDP ones.

5.1 Benchmark testing

The benchmarking suite **nfsbench** [8] developed by Olaf Kirch was used to test the SCI NFS server. The suite supports a number of tests including **nfsping** which sends a storm of NULL calls with a 1K payload to gauge network capacity and interface latency. The **nfswrite** test creates a 4 Mbyte file using a maximum 4K byte transfer size. The **nfsread** test reads data back from this file using the same transfer size. The **nfspebbles** benchmark creates several directories and files and runs a mix of operations on them. For our test the **Legato** mix - the one used in **nhfsstone** [14] - was employed. For all tests default values were used for all other parameters.

The tests were run over 10 Mbits/s Ethernet, Classical IP over ATM running at 155 Mbits/s and SCI. Results were also obtained for the case when the client and server are on the same machine. Write-gathering had no significant effect on the results shown because of the large mix of operations.

5.2 Results

The results are shown in Table 1. All of the SCI results are within 30% of the local results. **nfsping** shows the lowest relative improvement due to the overhead of signalling to the receiver that new data is now in the buffer. For a simple NFS call this event mechanism must be used twice, resulting in an additional overhead of $71\mu s$, which is significant for small packet transfers. This has a smaller effect for the **nfswrite** and **nfsread** results where larger transfer sizes are used. The **nfspebbles** results for SCI almost matches the throughput of the local results because a mixture of file and directory operations – some of which take longer than reading or writing – also reduces the relative overhead of signalling. Though better results can be obtained using a faster event mechanism, the results reported for SCI are significantly better than Ethernet or ATM which form the current networking infra-structure of most organisations.

6 CORBA over SCI

OmniORB2 is an implementation of the 2.0 specification of the Common Object Request Broker Architecture (CORBA). It forms the backbone of ORL's distributed object infrastructure and is motivated by our need for a multi-threaded

	Throughput(Mbit/s)	Avg. RTT(ms)
nfsping		
Ethernet	4.036	2.10
IP/ATM	18.12	0.40
SCI	28.65	0.20
Local	40.36	0.10
nfswrite		
Ethernet	6.26	5.30
IP/ATM	26.49	1.20
SCI	45.92	0.60
Local	52.18	0.50
nfsread		
Ethernet	6.38	5.30
IP/ATM	30.96	1.00
SCI	55.44	0.50
Local	70.15	0.40
nfspebbles		
Ethernet	2.12	3.60
IP/ATM	8.48	0.80
SCI	12.71	0.60
Local	12.92	0.50

Table 1: NFS results.

ORB to run our in-house developed hardware and software platforms. It is expected that performance improvements to omniORB2 gained through the use of a low latency interconnect will have an immediate effect on the whole environment. There are also other good reasons for choosing omniORB2 as a test case for our interconnect: omniORB2 is the fastest CORBA 2.0 ORB currently available [9] and is publicly available under a GPL Source License.

OmniORB2 has a structure whereby a thread is dedicated to a connection at both the client and server and multiplexing is achieved through scheduling between a number of blocking threads. To the standard omniORB2 package we added the user level communication abstraction using locked shared memory (described in section 4.1) over SCI in place of TCP. Over this transport abstraction we ran the standard GIOP (General Inter Orb Protocol).

With these modifications we were able to make a direct comparison of our work with omniORB2 over different transport and network types. Our modified ORB is capable of serving objects over SCI without compromising itself for TCP and simultaneous access to an object is possible through all the supported network types.

For comparison, we also implemented the shared memory transport using the standard System V shared memory primitives, portable over all our Unix platforms. The intra-machine shared memory performance was significantly better that both inter and intra machine TCP/ATM performance for all our platforms for both latency and bandwidth tests.

The figures shown in the table below offer a comparison between an omniORB2 Null RPC for a shared memory transport for intra-machine on two platforms and SCI on our Linux 200 Mhz Pentium Pro platform. Also included is the inter-machine time for Linux using TCP over a 155Mbps ATM network. At the IIOP level, the Null RPC required 73 bytes for the request and 29 bytes for the response.

Platform	Transport	Latency(us)
Intra 167 Mhz Solaris UltraSparc	SYS V Shared Memory	510
Inter 200 Mhz Linux Pentium Pro	TCP Socket	470
Intra 200 Mhz Linux Pentium Pro	SYS V Shared Memory	270
Inter 200 Mhz Linux Pentium Pro	SCI Shared Memory	156

These results illustrate a significant improvement in latency when using the SCI transport. It has been estimated that for the Linux platform, 110us is spent in the ORB with protocol and marshalling overhead, the remainder being spent in performing transmission and synchronisation through the shared buffer. It is interesting to note that the inter-machine time using SCI is significantly faster than intra-machine using standard shared memory primitives. This is due to the parallelisation of the GIOP and the avoidance of a context switch between the client and server and should be contrasted with the inter-machine NFS results presented in section 5. These had approached, but never exceeded the intra-machine NFS results. This difference is because omniORB2 takes a more aggressive approach to parallelising its protocol than does NFS.

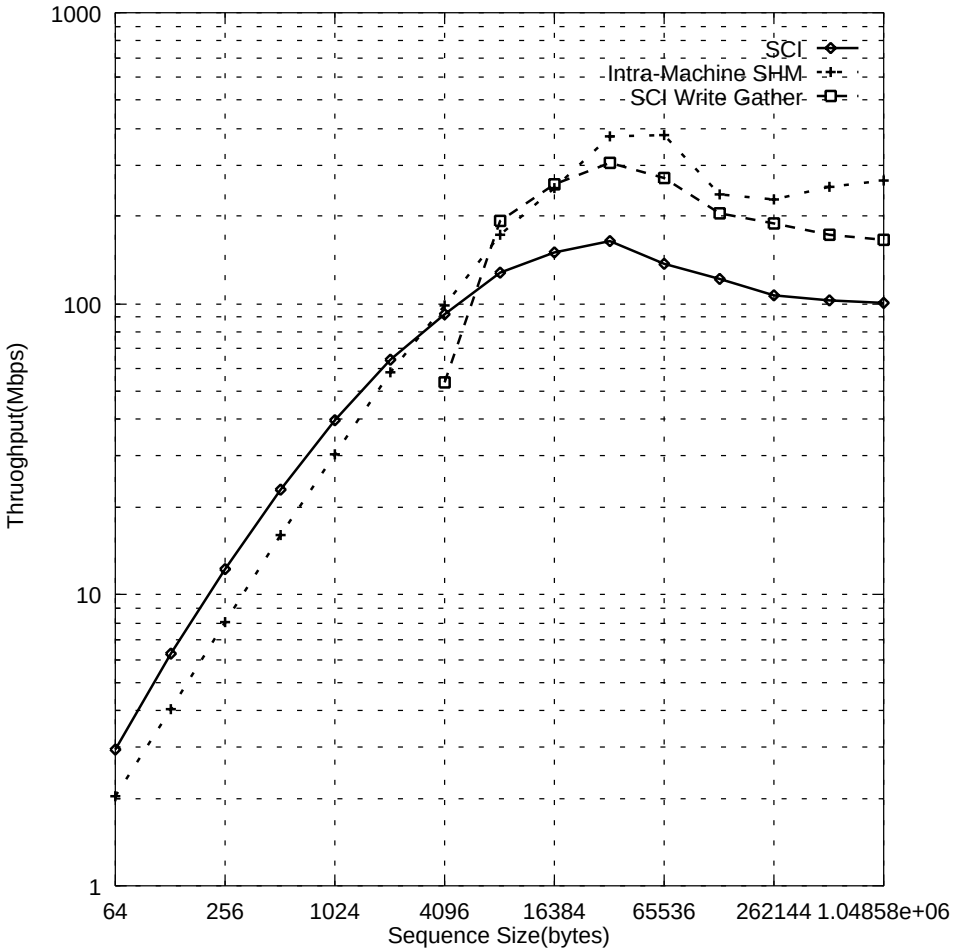

Figure 4: omniORB2 throughput results.

A second experiment to measure throughput on the Linux platform was also made. Blocks of varying sizes were streamed from a client to a server object, with the throughput being measured at the server. All data is transferred through the ORB, with the *oneway* attribute set, hence the client does not expect an acknowledgement from the server for data sent.

From the results shown in Figure 4, the intra-machine shared memory (SHM) trace shows a throughput which dips as the size of the Pentium Pro cache is reached. All the SCI traces show a drop off in performance as the size of the shared memory buffer is reached (32K). This would have been improved had the circular buffering abstraction (described in section 4.2) been used. The inter-machine SCI throughput with *write-gathering* enabled exceeds intra-machine SHM probably because of the avoidance of context switching. However, we were unable to use the write-gathering option for block sizes of less than a page without throughput being significantly reduced. The SCI trace shows a throughput without write-gathering enabled which is substantially less. We believe this in part to be due to problems with the PCI chip set on our FX motherboards.

7 Conclusions

Whilst there has been much research on the use of memory-mapped networks in the context of parallel computing relatively little work has been done on their suitability for distributed applications. Several research groups have examined RPC and socket interfaces using low-latency memory-mapped networks. In particular, the SCILAN project at the University of Oslo [12] has implemented a socket library for Windows NT over SCI which is akin to our socket library implementation for Linux. The SHRIMP project at Princeton University has also examined socket and RPC implementations [2]. Their Sun RPC compatible latency results are comparable with our implementation. However, they have also implemented non-compatible RPC mechanisms with very low round trip times. Neither of these projects have yet reported the performance of real distributed applications running over their socket implementations.

This paper has described our integration of the SCI interconnect with two infra-structural distributed applications, the standard NFS server and our in house CORBA 2.0 compliant ORB, omniORB2. Both these applications were ported, rather than heavily modified to use the interconnect and were run over an unmodified Linux operating system.

With this configuration, we have shown remarkable performance improvements using a strategy whereby multiplexing in the system is achieved through the operating system standard blocking and pre-emption facilities.

References

[1] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, and W. Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(2), 1995.

[2] S. Damianakis, A. Bilas, C. Dubnicki, and E. Felton. Client-Server Computing on Shrimp. *IEEE Micro*, 17(1), 1997.

[3] Stefanos Damianakis, Yuqun Chen, and Edward W. Felten. Reducing waiting costs in user-level communication. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[4] M. Fillo and R. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1), 1997.

[5] R. Gillett and R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, 17(1), 1997.

[6] Lars Paul Huse and Knut Omang. Large Scientific Calculations on Dedicated Clusters of Workstations. In *The proceedings of International Conference on Parallel and Distributed Systems, Euro-PDS'97*, June 1997.

[7] IEEE. Standard for Scalable Coherent Interface, March 1992. IEEE Std 1596-1992.

[8] Olaf Kirch. NFS Test and Performance Measurement Suite. ftp.mathematik.th-darmstadt.de/pub/linux/okir.

[9] S. L. Lo. The Implementation of a Low Call Overhead IIOP-based object request broker. In *ECOOP '97 Workshop CORBA: Implementation, Use and Evaluation*, 1997.

[10] Sun Microsystems. NFS: Network File System Protocol Specification, 1989. RFC 1050.

[11] S. Ryan, S. Gjessing, and M. Liaaen. Cluster communication using a PCI to SCI interface. In *IASTED Eighth International Conference on Parallel and Distributed Computing and Systems*, October 1996.

[12] S. J .Ryan and H. Bryhni. SCI for Local Area Networks. Technical Report 256, Department of Informatics, University of Oslo, 1998.

[13] Jens Simon and Oliver Heinz. Experiences with a SCI Multiprocessor Workstation Cluster. In *ARCS 97*, June 1997.

[14] R. Stine. FYI on a network management tool catalog. RFC-1147, April 1990.

[15] A Tanenbaum. *Operating Systems Design and Implementation*. Prentice-Hall, 1987.