Department of Informatics

SCI for Local Area Networks

Stein Jørgen Ryan
and Haakon Bryhni

Research Report 256

ISBN 82-7368-180-7
ISSN 0806-3036

January 1998

SCI for Local Area Networks

Stein Jørgen Ryan and Haakon Bryhni
University of Oslo
PO Box 1080 Blindern, 0316 Oslo, Norway
Email: steinrya,bryhni}@ifi.uio.no

January 1998

Abstract

We show how the traditional protocol stack, such as TCP/IP, can be eliminated for socket based high speed communication within a cluster. The SCI shared memory interconnect is used as an example, and we demonstrate how existing applications can utilize the new technology without relinking. This is done by dynamically remapping the TCP/IP socket implementation to our high performance SCILAN sockets. We describe a novel mechanism for synchronization of communication through shared memory, aimed at minimizing the interrupt load on the receiving system. We discuss the implementation and present an evaluation with comparison to alternative technologies, such as 100baseT and ATM. Significant improvement over current solutions are shown both in terms of throughput and latency.

1 Introduction

Recent network technologies give us the ability to transfer multi-gigabytes per second over interconnects with low latency *at the physical layer*. When fast network technologies such as MemoryChannel [GK96], ServerNet [Hor95], SCI [P1593], Myrinet [BCF$^+$96] or ATM [DeP93] are used at the physical layer, traditional protocol stacks may not fully utilize the capabilities at the lowest layer. For example, sending short messages with TCP/IP over 155 Mbit/s ATM has a latency from sending to receiving process of about 20000 CPU cycles [BO96], comparable to latencies using 10BaseT Ethernet! Thus, applications cannot utilize the low latency made possible by the new technology. When traditional network protocols are used, only bandwidth is the distinguishing factor between the various interconnects.

In this paper we show how new high speed network technologies can be utilized with no changes in existing application software. Our approach relies on remote memory access, such as offered by ServerNet or SCI. We show how a user level transport protocol can be used to replace TCP for high speed communication within a cluster, while retaining the socket semantics widely used in networking applications. The design called "SCILAN" is an implementation of a Berkeley sockets layer for Windows NT that succeeds in keeping the software overhead at an absolute minimum while running existing binaries (i.e. no relinking or source code modifications required). We choose SCI as our example technology since this interconnect provides very high throughput combined with low latency for communication within a cluster. SCI is a typical "System Area Network" suited for machine-room interconnects. Normally, copper cabling is used, but both paralell and serial fiber cables can be used to extend the physical range. An example of an SCILAN cluster is shown in figure 1. The cluster is engaged in two types of communication:

- For intracluster communication, the SCILAN library bypasses the protocol stack, giving high throughput and very low latency.
- For local area communication external to the cluster, socket operations transparently fall through SCILAN into the standard protocol stack. In this way, IP level connectivity is obtained over a switched IP LAN, giving high throughput and moderate latency.

In both cases, the same socket Application Programming Interface (API) is used to access communication services, and all socket based applications are used without modification.

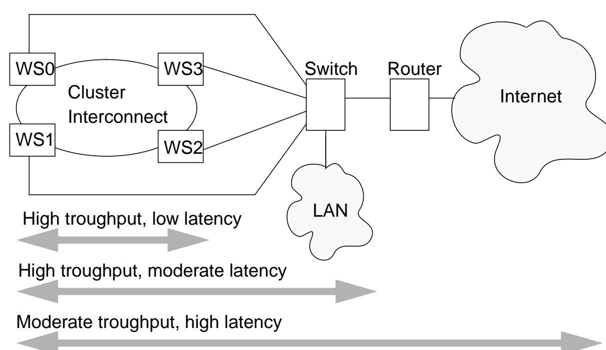

Figure 1: SCILAN cluster architecture

The paper is organized as follows. An overview of common methods for access to shared memory interconnects and requirements for the new SCILAN approach are given in section 2. The architecture and design of SCILAN is discussed in detail in section 3. An evaluation of the method and comparisons with other technologies are given in section 4. Related work is discussed in section 5. Finally, we present opportunities for further work in section 6 and conclude the paper.

2 Background

Our interest is how a shared memory interconnect can be used to obtain the highest possible performance for a computer system in a cluster environment, while running existing socket based applications. We describe current methods for accessing such interconnects, and present our solution requirements.

2.1 Shared memory interconnects

A shared memory interconnect allows the interconnected machines to access each others memory. Data can be moved across the interconnect with a single CPU `load` or `store` instruction. We refer to this as Programmed IO (PIO). PIO has very low software overhead for communication. We want to interconnect Commercial-Off-The-Shelf (COTS) systems, so the cluster interconnect adapters must be hosted by a standard IO bus. In our performance analysis of SCILAN we use PCI based PCs interconnected with PCI/SCI cluster adapters from Dolphin [Dol97]. Remote shared memory can not be consistently cached by hardware, since the IO bus does not support a full cache coherence protocol. Modern processors rely heavily on caching, so access to remote memory will be orders of magnitude higher than access to local memory. [1] For efficiency considerations, complex shared data structures should not be placed in shared memory. The rest of this paper assumes that we use shared memory for efficient message passing.

The performance of a traditional message passing system is governed by several parameters. Information is transferred in packets, and the *Maximum Transmission Unit* (MTU) dictates the maximum size of a packet. MTU size is very important for throughput since each packet may carry a considerable processing overhead. Error control is done for each packet by checksum control and acknowledgement packets. The *window size* dictates the maximum number of bytes transferred before acknowledgement is required. The required window size increases with increasing link speed and physical distance. For a shared memory interconnect, data transfers are not packetized by software. On such systems, the MTU size represents the maximum amount of data transmitted before checking for transfer errors. The window size is simply the size of the receiver buffer.

In this section we take a closer look at different ways to utilize the high performance of a shared memory interconnect. These methods provide different trade-offs between efficiency, portability and ease of application development. Table 1 lists different alternatives, discussed below. Both regular Sockets and SCILAN work in a message passing paradigm, while the others operate using message passing or shared memory. The SCILAN architecture is described separately in section 3.

Method	API	Protocol	Latency	Porting necessary	Applications
Sockets	Berkeley Sockets	TCP/IP	High	No	Many
Native API	Proprietary	Proprietary	Low	Yes	Very few
HPC API[2]	MPI/PVM etc	Variable	Low	No[3]	Few
SCILAN	Berkeley Sockets	Proprietary	Low	No	Many

Table 1: Comparison of interconnect access techniques

[1] Accessing remote memory over the PCI/SCI interconnect is in the order of $2\mu s$, while local memory has worst case latency of roughly 70ns.

2.1.1 Native API

The best obtainable performance is achieved by tuning the application to the underlying interconnect technology and its native paradigm for communication - in this case shared memory. This means direct use of the “native” interconnect API as shown in figure 2. An example of this type of API is the Dolphin SCI API [Dol95]. Access to the memory of other machines is controlled by a driver. Once connected, data transfers bypass the driver using PIO into remote memory. With knowledge of the application behaviour, the application programmer can take advantage of hardware-specific transfer mechanisms. For example, PIO can be used efficiently to transfer short and medium sized messages, while DMA is used for transferring large messages in parallel with computations. In this way, the application can get the most out of the clustering hardware. The down side is that the application depends heavily on mechanisms in the current hardware (such as a DMA engine), which may be unavailable on other hardware. As soon as another technology takes the lead in price/performance, the application may have to be largely rewritten. To summarize, this approach is only possible for specialized applications where the cost of programming can be defended by the need for ultimate performance such as in the High Performance Computing (HPC) domain.

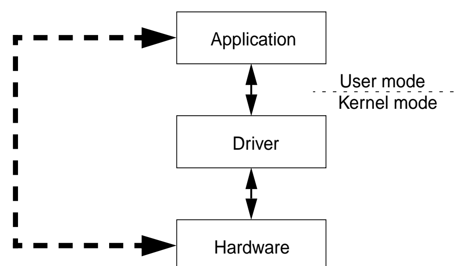

Figure 2: Native technology API. Thick dotted line indicates direct application access to shared memory.

2.1.2 High Performance Computing APIs

Another approach is to interface the new technology to a standard HPC API, as shown in figure 3. MPI [Mes94] is an example of such an API, designed for portable parallel programming. Portability is improved by using a standardized programming interface, rather than tailoring the application directly to one particular technology. Performance is slightly reduced compared to the native API approach, since one (thin) software layer is added, but the higher level of abstraction ensures portability.

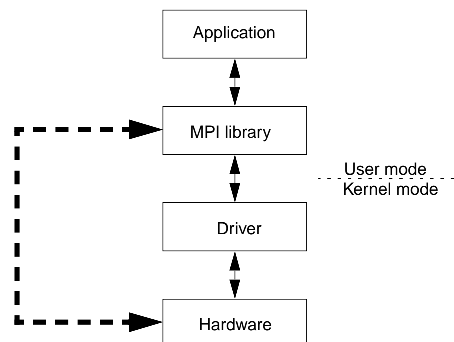

Figure 3: MPI approach for shared memory. Thick dotted line indicates direct application access to shared memory.

[2] High Performance Computing API like MPI or a similar parallel programming environment such as PVM

[3] Applications must be written for the specific parallel programming API. However, many APIs exist and there are limited number of applications available.

2.1.3 Traditional protocol stack

A third approach commonly used in local area networks, is to use the new technology at the lowest layer in an existing protocol stack. With this approach, all applications will work without modification. To interface at the lowest layer, a standard OS-specific driver interface is required. For Windows NT this interface is the Network Device Interface Specification (NDIS). Examples are Gigabit Ethernet and clustering technologies such as Dolphin's SCI/NDIS solution [Dol97]. The SCI/NDIS solution is especially interesting since it uses the same hardware layer as SCILAN. By comparing SCI/NDIS performance with SCILAN, we can quantify the protocol stack overhead.

Given high CPU processing power, sufficient Maximum Transmission Unit (MTU) size and zero-copy [Hsi96] methods in the protocol stack, this approach is capable of driving many interconnects to link speed. However, latency is high due to software overhead. Moving data through a protocol stack generally involves a number of data touching operations, which are expensive in terms of latency. Data are for instance touched by the CPU when checksum is calculated (as required by e.g. TCP).

2.2 Solution requirements for SCILAN

A socket interface to a shared memory interconnect is not trivial when the goal is hardware level performance. A number of problems must be solved to fulfill this requirement:

1. In general, socket implementations are located in the operating system kernel. Thus, every socket operation involves a transition into the kernel and back, which is very expensive in terms of latency. Invoking a driver-supported kernel entry point costs roughly 6000 CPU cycles on Windows NT. Thus, kernel calls should be avoided in the predominant code path of the socket implementation. A kernel component is required to set up shared memory mappings, but during the bulk of the data transfer, the kernel should not be involved.

2. Interrupt processing is expensive on most operating systems (several thousand CPU cycles) and should be reduced to a minimum. Per packet interrupts should be avoided. This is especially important with high packet arrival rates, as can be expected on a low latency, high speed interconnect.

3. The most interesting class of applications requires reliable sockets (TCP style). We must support the exact same semantics as the socket API, including error control and flow control. Flow control at the buffer level is usually not part of a shared memory interconnect such as SCI. Transactions are secured at the physical level by hardware retries, but flow control at the buffer level is left to software. The socket API semantics requires flow control, which is usually implemented by the transport provider below the socket level (typically by the TCP protocol). Since we want to bypass the protocol stack, we have to implement our own flow control. This is described in section 3.4.

To summarize, in order to integrate the socket interface with a memory mapped technology, we need a solution that avoids kernel calls and interrupts while implementing flow control at the buffer level and utilizes the SCI hardware error control.

3 Architecture

The SCILAN architecture is focused on moving the operating system kernel out of the predominant code paths executed by the socket library. We also attempt to minimize the number of interrupts during communication by using a novel mechanism in the cluster adapter hardware.

3.1 Design overview

Applications using sockets for communication in the Windows NT environment will use runtime linking to link to the socket library "WinSock". We modified the runtime linking step so that all socket calls from the application can be redirected to our replacement SCILAN socket library on a process by process basis. When

a process starts, the SCILAN library gains control and modifies tables in the executable image which control run-time linked calls. A detailed description of this method is outside the scope of this paper, but the technique is described in [Pie95]. SCILAN sockets bypass the protocol stack and communicate directly through SCI shared memory in user space. This dramatically reduces the software overhead of communication. Figure 4 shows the new architecture. It consist of a device driver and the SCILAN socket library alongside the existing protocol stack. The device driver is known as the Interconnect Manager (ICM) and is described in more detail in [RMG97].

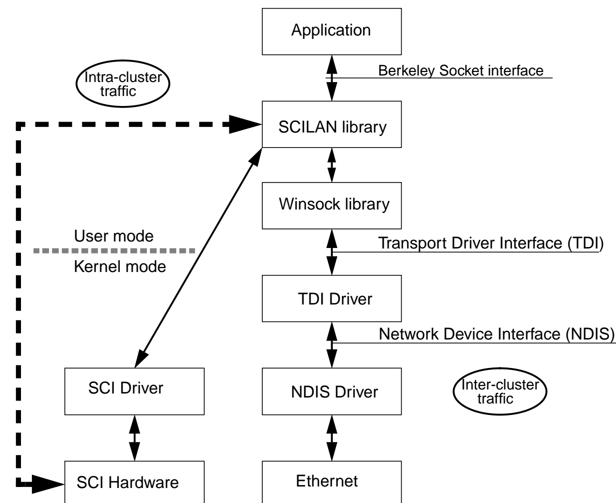

Figure 4: Functional blocks in the SCILAN architecture. Thick dotted line indicates direct application access to shared memory.

3.2 Shared memory message passing

We can view communication between machines as a two step process: First, data must be moved from the physical address space of the sending party into the physical address space of the receiving party. Second, the receiver and transmitter must synchronize. The receiver needs to know when data becomes available, and the sender needs to observe flow control restrictions.

Traditional communication hardware (Ethernet etc) will interrupt the host when a new packet has arrived in the receiver buffer, thus combining these two basic steps. When communicating through shared memory, we must implement the two steps explicitly. SCILAN uses the following approach:

- A communication endpoint in the receiver consists of a pagelocked receiver buffer and a special *interrupt flag*. An interrupt flag is simply a word in memory. If a remote host writes to that word (*stimulates* the flag) over the interconnect through a special mapping, the target cluster adapter will atomically increment the word. This is a specific feature of the PCI/SCI adapter. If the most significant bit of the word was zero prior to the increment, an interrupt is triggered. Any write operations to the word through ordinary mappings are carried out with usual write semantics so we can toggle the most significant bit, thereby switching interrupts on and off.

- Using CPU `store` instructions, the sender writes data across SCI into the buffer of the receiver endpoint. Due to the protocol used in the SCI hardware, the sender is able to verify immediately whether this transfer succeeds or not. Thus there is no need for the sender to explicitly wait for acknowledge packets from the receiver. This makes throughput less sensitive to interrupt latency and Maximum Transfer Unit (MTU) size than if using SCI at the bottom of a traditional protocol stack. A full protocol stack would typically mandate the use of acknowledge packets because it does not know that the lowest level is capable of detecting communication errors on its own.

- If the transfer succeeds, the sender stimulates the interrupt flag of the receiver endpoint. The flag has an initial value of `0x7FFFFFFF` so that multiple stimuli from a sender will result in just a single interrupt.

The first stimulus will trigger an interrupt but also increment the flag thereby setting the most significant bit and so inhibiting further interrupts. Further stimuli remain visible to the receiver (they will keep incrementing the flag which can be inspected in memory), but will not interrupt.

Note that this approach does not require concurrent processes to serialize their use of the communication hardware. This is because the different communication endpoints all use different hardware resources (a different buffer and interrupt flag). In order to send data to a remote endpoint as described above, it is not necessary to prevent others from writing to a different remote endpoint buffer through the same local cluster adapter hardware. The situation would be entirely different if the data transfer took place using a DMA engine or some other single hardware resource which can only be used by one thread at a time. The lack of competition for hardware resources means that we do not have to serialize access to the communication hardware through a device driver. This is an important optimization because entering a kernel device driver represents high software overhead. Instead we can perform the transfers directly from user mode once the remote endpoint buffer and interrupt flag have been mapped into virtual memory.

3.3 Addressing

Each cluster adapter has a node ID unique to the cluster interconnect. This node ID is the first 16 bits of the cluster-wide 64 bit SCI shared memory address space. We use regular DNS to register cluster-specific IP addresses, obtained from the class C private network between 192.168.0.0 - 192.168.255.0 as defined in [RMK+96]. This address space gives us 255 networks with 255 hosts, which are mapped into hardware specific node IDs. Thus, DNS is used to administer SCI node IDs in a simple way. These IP addresses will never be in use on a regular IP network, and traffic within the SCILAN will never be visible on the IP network either since no routing is supported.

3.4 Socket implementation

In order to describe how SCILAN implements sockets, we show how two machines A and B use the SCILAN socket library to establish a bidirectional connection between them.

1. Machine A must first create a socket s and prepare the socket for receiving connection requests from machine B. This is done by invoking `listen()` on s.

2. A can now invoke `accept()` on s to block waiting for a connection request.

3. B creates a socket b and issues a connection request to machine A by invoking `connect`.

4. The connection request from B unblocks the `accept()` in A and creates a new socket a in machine A.

A bidirectional connection now exists between the sockets a and b allowing the machines A and B to communicate using the socket `send` and `recv` routines.

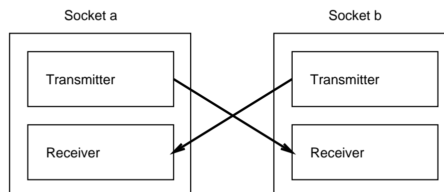

Figure 5: The internal structure of a connected pair of SCILAN sockets

Internally, the sockets a and b both consist of a *transmitter* and a *receiver*. The transmitter of socket a will transmit to the receiver of socket b and vice versa as shown in figure 5. Each receiver has a receiver buffer and an interrupt flag which are used for data transfer as explained in section 3.2. Each transmitter has an interrupt flag which is used for flow control. The receiver will stimulate the transmitter interrupt flag when

room becomes available in the receiver buffer. This allows a transmitter to pause transmission and go to sleep on the transmitter interrupt flag when the target receiver buffer runs full. The transmitter detects a full target buffer by inspecting the read pointer in the receiver buffer.

For connection oriented traffic there is only one transmitter per receiver. This is not true for connectionless sockets. For example, the receiver of s must be able to receive connection requests from any number of transmitters, so transmitters sending connection requests to s must synchronize access to the receiver buffer of s. Synchronization is done using a disabled interrupt flag as a test-and-set mechanism. This is not required in the connection oriented case where there is only one transmitter per receiver. Accordingly we have implemented both connectionless and connection oriented senders and receivers. The connection oriented versions are more efficient.

3.5 The interrupt flags

The interrupt flags play an important role in the design because they let us control the use of interrupts independently for each receiver and transmitter pair (i.e. for each virtual channel). Disabled interrupt flags also acts as a general test-and-set synchronization primitive. A transmitter always stimulates the interrupt flag in the receiver whenever it has written data into the corresponding receiver buffer. This does not necessarily result in an interrupt at the receiving end. An interrupt can be generated only when the receiver has emptied its receive buffers and decided to sleep, waiting for more data. Prior to sleeping, the interrupt flag will be enabled so that the sleeping thread is woken up by the kernel as more data becomes available. Stimuli received while the receiving thread is runable will not trigger interrupts.

The interrupt flag mechanism used by SCILAN attempts to minimize the cost of per packet interrupts by selectively disabling interrupts per virtual channel. On SCILAN, packet reception triggers an interrupt only when a thread is known to sleep waiting for data to arrive on the targeted virtual channel. The interrupt will transfer control to the kernel which wakes up the thread.

Minimizing interrupt processing becomes very important as the speed of the interconnect increases because the number of interrupts per second during a multipacket data transfer is (at least partially) a function of the raw bandwidth of the interconnect. In traditional systems, interrupts must occur on *every* packet to invoke a device driver which pulls the packet out of a limited receiver buffer in the communication hardware to make room for new packets. On SCILAN, each receiver has its own buffer, so receiving a packet does not have to trigger immediate action through an interrupt. This allows us to keep the interrupt flag of a particular receiver disabled until a thread blocks, waiting for data to arrive.

3.6 Performance considerations

The suggested approach for communication by PIO through shared memory works best with smaller messages. For larger messages, it is better to do the data transfer by DMA rather than PIO. This frees the CPU to do other things while the transfer takes place. The overhead of invoking a device driver is also more readily justified because the cost can be amortized over a large message. The focus of the SCILAN design is efficient communication for short messages, so we currently do not use DMA. However, expanding the system to DMA is straightforward because the Dolphin PCI/SCI adapter card comes with a DMA engine controlled by the SCI driver. The investigations of DMA software overhead in [RMG97] indicate that, for our test machines, we should not start using DMA until the packet size reaches roughly 5K. This parameter may vary depending on the hardware of the host machine, and needs to be tuned from machine to machine.

3.7 Semantic constraints

In order to bypass the protocol stack, we had to make certain assumptions. In the traditional UNIX socket implementations, socket descriptors are valid kernel file descriptors which can be passed in to the `read` and `write` system calls. This implies that sockets are implemented in the kernel, conflicting with our design philosophy. However, for historical reasons, the WinSock 1.1 specification explicitly states that socket descriptors are not valid file descriptors. This limitation allows us to implement the WinSock 1.1 specification entirely in user mode. A large majority of Windows networking applications use Winsock 1.1, and can

therefore be run unmodified on SCILAN. With the introduction of WinSock 2.0, socket descriptors *may* be valid file descriptors (depending on the protocol stack implementation), but WinSock 2.0 applications can not generally assume this. We are currently extending SCILAN to support WinSock 2.0 as described in section 6.

4 Evaluation

In order to evaluate our approach, we have studied throughput and latency characteristics of SCILAN and alternative solutions like the SCI/NDIS solution from Dolphin ICS, 155 Mbit/s ATM using LAN Emulation and standard 100baseT (Fast Ethernet). We executed the same binary copy of the test program for all the network technologies, relying on runtime remapping of socket calls in the SCILAN case. All measurements report latency and throughput at the application level.

4.1 Reference configuration

The reference configuration has been two 200 MHz Pentium Intel motherboards with Intel 430HX host bridge chipset and 512K cache running the Windows NT 4.0 operating system (build 1381, service pack 3). Windows NT was chosen mainly due to our experience with injecting new code into existing program binaries by on-the-fly modification of runtime linker tables. The following network technologies were used:

- PCI/SCI adapters, revision B, from Dolphin ICS, using the ICM device driver [RMG97].
- SCI/NDIS software drivers version 1.0 from Dolphin ICS.
- Fore ATM 200E ATM cards connected to a Fore 200WG ATM switch, using ForeThought drivers version 4.0.
- 3Com Ethernet Xpress 100baseT Fast Ethernet cards connected back-to-back.

All technologies were tested at 1500 byte MTU size. SCILAN can be configured to use any MTU size, but 1500 byte was chosen for fair comparison. ATM LAN Emulation can support larger MTU sizes, but the current NT drivers only support MTU of 15000 bytes. It should be noted that the MTU size is highly critical for throughput. The SCI/NDIS solution use SCI at the MAC level of a standard Windows NT protocol stack. The Dolphin drivers implement the Network Device Interface Specification (NDIS) on top of SCI.

4.2 Factors limiting performance

Figure 6 tracks the path of data through hardware as the SCILAN socket implementation transfers data from one host to another. Data transfer is accomplished by copy operations on the receiver buffer in shared memory as described below.

1. The *transmitter copy operation* is executed by the sending host when the sending application invokes the `send` socket call. The SCILAN implementation of `send` will use the CPU `load` instruction to read a word at a time from the local application buffer that was specified in the `send` call (flow 1 in figure 6). Using the `store` instruction, each word is written across SCI (flow 2) into the receiver buffer of the targeted endpoint. The transmitting SCI adapter will gather multiple writes into a single 64 byte SCI write request which is then carried out and acknowledged in hardware by the receiving SCI adapter.
2. The *receiver copy operation* occurs as the receiver invokes `recv` to consume data from the receiver buffer. Data is read from the receiver buffer (flow 3) and written to the application buffer given to `recv` (flow 4). This is done using PIO.

We see that the transmitter copy operation moves data twice across the system bus of the sending host and once across the system bus of the receiving host. The receiver copy operation moves data twice across the system bus of the receiving host. In total, the system bus of the receiving host must move the received data no less than three times! It seems that the system bus can easily become a bottleneck. Figure 7 shows that on our test machines, the system bus is indeed a serious bottleneck as explained below.

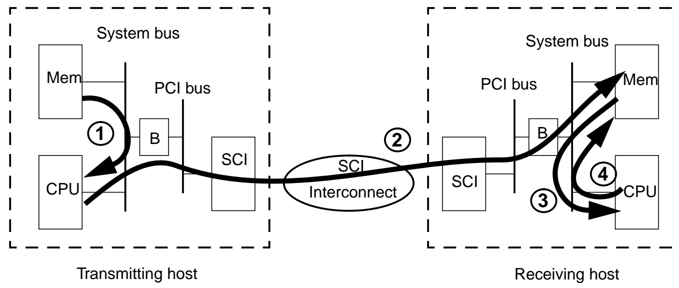

Figure 6: Flow of SCILAN data transfer. The system bus is connected to the PCI bus via bridge chip B.

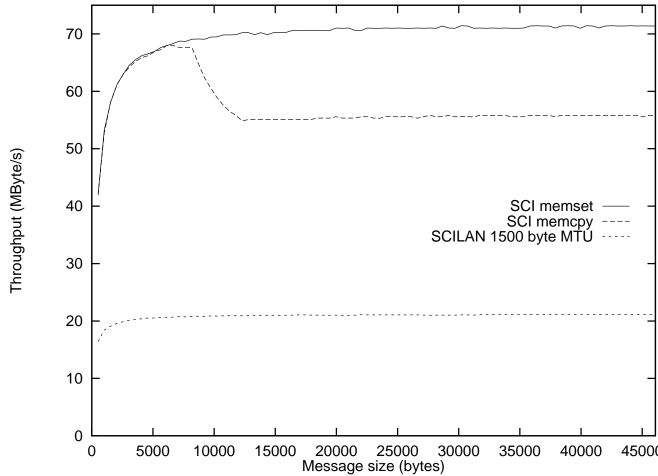

Figure 7: Copy performance

- The `SCI memset` graph shows the throughput obtained when writing a constant value into remote memory over SCI along flow number 2 in figure 6. Data flows through the sender system bus, then across the SCI interconnect and finally through the receiver's system bus into the receiver buffer.
- The `SCI memcpy` graph shows the throughput of the *transmitter copy operation* which reads data from the local memory of the sender (flow 1) and writes it into the receiver buffer over SCI (flow 2). Data flows through the sender system bus twice, then across SCI and finally through the receiver's system bus once before ending up in the receiver buffer. The difference between the `memset` and `memcpy` graphs in figure 7 indicates the cost of the extra system bus transfer on the sender side caused by reading data from the buffer that was passed to `send`.
- The `SCILAN` graph shows the dramatic drop in performance caused by the receiver copy operation. This operation effectively triples the load on the receiver system bus compared to executing only the transmitter copy operation. On our test system, this saturates the receiver system bus.

The `SCI memcpy` graph shows that performing only the transmitter copy operation achieves a throughput of roughly $X_t = 55$ MB/s. We have measured the receiver copy operation to $X_r = 36$ MB/s. Because both operations involve the receiver system bus, we can model the transmitter copy operation as occuring in its entirety before the receiver copy operation. This allows us to determine an upper limit X_{max} on SCILAN throughput.

$$X_{max} = \frac{X_t * X_r}{X_t + X_r}$$

$X_t = 55$ MB/s and $X_r = 36$ MB/s gives $X_{max} = 21.8$ MB/s. Measured SCILAN throughput shows very good performance relative to this hardware dictated upper limit. The SCILAN graph of figure 7 shows that measured SCILAN throughput reaches the hardware limit X_{max}. 90% of maximum throughput is reached already at 1500 bytes. These results indicate that the SCILAN socket library carries very little software overhead.

4.3 Memory bandwith

The bus saturation problem has also been observed on the SparcStation 20 platform [BO96], but is not an issue on the Ultra platform due to the improved memory bandwith [OP97]. As we can see from the results in table 2, the aggressive memory subsystem in the Ultra family of computers gives superior performance during the receiver copy operation. Flow-controlled transfers comparable to those of SCILAN have been carried out on the Ultra platform using the same PCI/SCI cards. On the Ultra we reach 67 MByte/s, significantly higher than the 21 MByte/s measured with SCILAN on the test PCs. [4]. This result clearly demonstrates the important effect of memory bandwith.

Processor	Throughput [Mbyte/s]
Pentium 200 Mhz, HX chipset	36
UltraSPARC-I 167 MHz (E3000)	178
UltraSPARC-II 296 MHz (Quark)	214

Table 2: Measured throughput of the receiver copy operation.

4.4 Link and bus bandwith

For SCI, the link speed is much higher than the bandwith of the memory bus and IO bus since the link is intended to support many workstations in a ring topology optionally augmented by switches. The current PCI/SCI adapters use the LC-1 chip with link speed of 200 MByte/s. The recently announced LC-2 chip handles 500 MByte/s. The other components of the PCI/SCI adapter saturate at 88 MByte/s, which constitutes a hard upper limit for application performance. For comparison, the link speed of our ATM adapters is 155 Mbit/s, which corresponds to 17.5 Mbyte/s when overhead of ATM headers is taken into consideration. Link speed for 100baseT is 100 Mbit/s, i.e. 12.5 MByte/s.

4.5 Overhead of flow control

The socket API requires flow control so that a slow receiver can block further transmissions until data has been consumed. However, flow control at this level is not supported by the underlying SCI technology. Figure 8 compares the throughput of the transmit copy operation with `nocopy SCI LAN` where flow control is added, but the receive copy operation is not performed on `recv`. Lowering the MTU size increases overhead on the sending side, because checking for transmission errors is done more frequently. On SCILAN, the throughput is limited by the system bus on the receiver side, so the MTU size does not affect performance. When the receiver copy is dropped (see the `nocopy SCILAN` graphs of figure 8), we see a considereable difference between 1500 byte and 16Kbyte MTU size. As we can see from the `nocopy SCILAN` graph, the flow control logic adds very little overhead. If we include the receiver copy operation, we get the true SCILAN performance at about 20 MByte/s as shown in the `SCILAN` graph in figure 8. Data is now moved using transfer 1,2,3 and 4 in figure 6.

Note that the throughput of the transmitter copy operation (flow 1+2) is often (misleadingly) compared to TCP throughput. A fair comparison must take into consideration both the transmitter and receiver copy operations as well as buffer-level flow control. Many papers that discuss a new interconnect technology measure the raw write performance of the interconnect, thus ignoring flow control (see for instance [BCF$^+$96], [BO96]). This approach gives artificially high throughput numbers since receive buffers are continously overwritten. [5]

[4]Measured when transmitting from one UltraSparc-II/248 MHz to an UltraSparc-II/296 MHz with the `ctp` program used in [OP97]. For this speed, both rate control of the sender and flow control is required to avoid loss of data in the receiver.

[5]The write performance is of limited interest, only in applications where "unlimited" receive buffers are available, such as in a Data Aquisition system tracking a short event or in specialized parallel applications where flow control is implemented in the application for instance by means of semaphores.

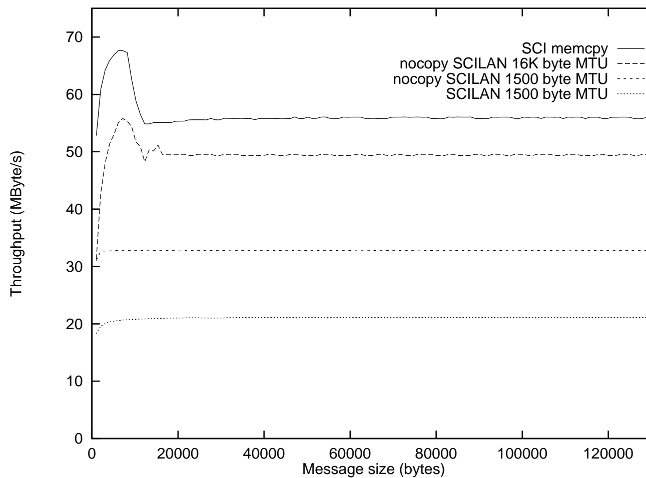

Figure 8: Throughput effect of SCILAN flow control and receiver copy

4.6 Comparing network throughput for large messages

Figure 9 shows that SCILAN throughput greatly outperforms the competing technologies. The closest competitor is the SCI/NDIS implementation where a throughput of 10 MByte/s is obtained for large packets. However, 90% of maximum throughput for this solution is achieved first at 25 Kbytes message size. The next technology is ATM Lan Emulation and finally Fast Ethernet. [6] For smaller packets, for instance 4 to 8 KBytes, 100baseT gives the best performance of the technologies that use a traditional protocol stack. In all tests, however, SCILAN achieves more than twice the throughput of any of the other technologies for all message sizes.

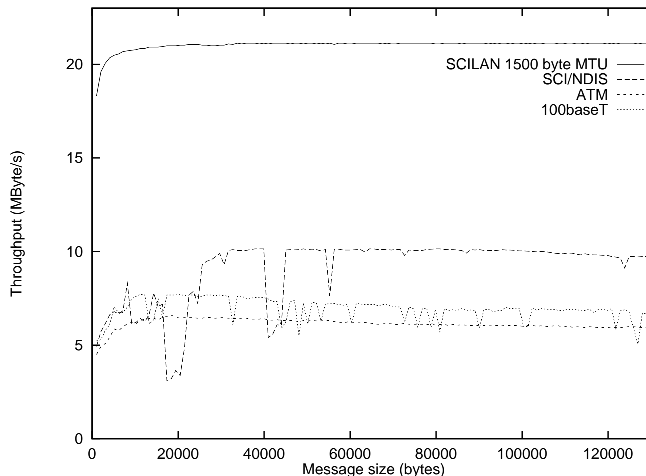

Figure 9: Throughput of SCILAN compared to SCI NDIS, ATM and 100baseT using TCP

To make sure the comparison is fair with regard to transmission window size, we ran a series of TCP measurements with different `so_sndbuf` and `so_rcvbuf` parameters. We used 8, 32 and 64K send and receive buffers, as well as the default values. The effect of increasing the transmission window size does not give more than 10% increase in measured throughput for all the evaluated technologies using TCP.

4.7 Latency and throughput for small messages

A significant advantage of the SCILAN technology is the good throughput for small messages. Figure 10 shows the performance of messages from 64 to 4096 bytes for SCILAN and 100baseT technologies. 100baseT

[6] It should be noted that we use ATM LAN Emulation with 1500 byte MTU. Thus, ATM cannot leverage its potential of large packets (9180 or 18K) possible in an all-ATM environment. Larger MTU values give higher throughput. Measurements in Unix and Macintosh environment show that we can drive ATM close to theoretical maximum. However, we have not included measurements with larger MTUs for ATM due to the lack of driver support for Windows NT.

is chosen for comparison, since this technology has best latency characteristics compared to SCI/NDIS and ATM.

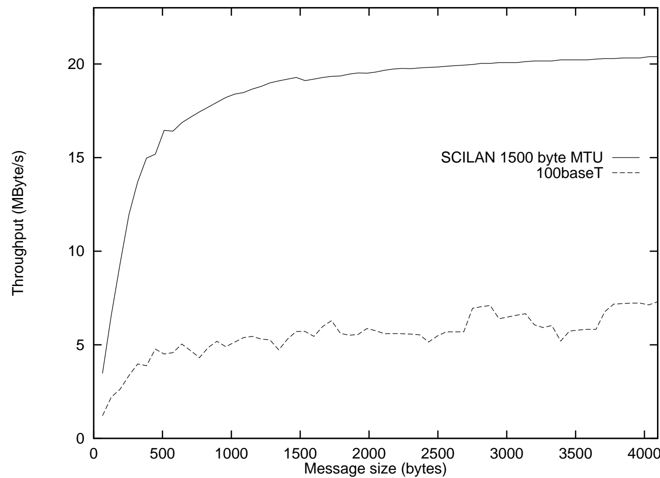

Figure 10: Throughput for small packets

As the message size decreases, the transfer overhead (setup and interrupt time) consitutes more and more of the total transfer time. Thus, the interconnect latency and interrupt processing time determines the obtainable throughput for small messages. For many cluster applications, the transfer overhead is the single most important performance metric. Figure 11 compares transfer times for short messages on the different technologies. The graphs were obtained by timing a ping pong test over a large number of iterations for each message size. Transfer time is defined from a `send` call is invoked to the corresponding `recv` call returns. Latency is defined as the transfer time for a message of 0 bytes.

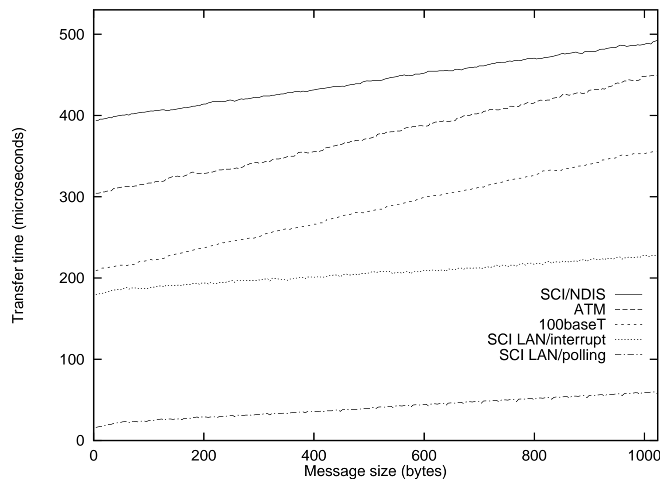

Figure 11: Transfer time of SCILAN compared to SCI NDIS, ATM and 100baseT using TCP

As we can observe, the SCI/NDIS implementation has the highest latency, followed by ATM, 100baseT and SCILAN. Note also the steeper inclination of the ATM and 100baseT graphs compared to SCI/NDIS and SCILAN. This is due to the lower throughput of these technologies. Table 3 summarizes the latency measurements.

We note that the SCI/NDIS solution has a fairly high latency compared to the other solutions that use the standard protocol stack. This is due to a known weakness in the interrupt handling in the Dolphin driver software for Windows NT. The latency for ATM and 100baseT is primarily due to the DMA setup overhead and interrupt handling. Latency of the SCILAN interrupt approach is primarily interrupt handling in the receiving system.

The novel interrupt flag mechanism presented in section 3.5 facilitates a simple polling approach which can back off to interrupts after prolonged unsucessful polling. The SCILAN polling graph in figure 11 shows

Technology	Latency [μs]
NDIS SCI	394.1
ATM	304.5
100baseT	209.3
SCILAN interrupt	179.8
SCILAN polling	16.1

Table 3: Measured latency (on a 200 MHz system)

Technology	Adapter [USD]	Switch port [USD]	Total 8 [USD]	Distance [m]
100baseT	100	360	3680	200 (UTP5)
SCI ring	870	0	6960	10
ATM 155Mbit/s	390	940	10640	100 (UTP5)
SCI switched	870	600	11760	10
Myrinet	1700	300	16000	12
Gigabit Ethernet	1700	N/A	N/A	300 (fiber)

Table 4: Suggested retail price in November 1997 for adapters and switch ports for competing technologies.

the effect of polling. Polling gives artificially good results in a ping-pong test, and SCILAN latency in a real application will fall somewhere between the polling and interrupt approach. Thus, by disabling the polling feature of SCILAN (as in the SCILAN/interrupt graph) we get a conservative comparison of the technologies.

4.8 Price and performance

Our evaluation indicates that SCILAN offers the best throughput and latency characteristics of the evaluated technologies. However, other attributes such as price and physical constraints of the interconnect is of importance. Table 4 compares the manufacturer's suggested retail price of the interconnect part of a reference system with eight processing nodes, as well as a rough estimate of the maximum node-to-switch distance. Note that the high link speed and point-to-point topology of SCI allows fairly large configurations without switches. Thus, we have included the price of a simple SCI ring and a switched SCI configuration. All adapter cards are listed with 3 meter cable included. The table shows that SCI delivers the best performance at a highly competitive price.

4.9 Adapter optimizations

As pointed out in section 4.2, the receiver copy operation saturates the system bus. This can be relieved by improving the system bus bandwith, but this does not scale with the number of local cluster adapters and increased link speed. Figure 6 shows that data moves across the receiving system bus no less than three times. This can be reduced by placing a medium to large memory on the cluster adapter itself as shown in figure 12. The Myrinet [BCF+96], Medusa [Mar94] and Afterburner [DWB+93] network adapters use such onboard memory. This approach is supported by research in zero-copy methods [Hsi96].

The transmitter copy operation moves data into the memory of the receiving adapter board, while the receiver copy operation moves data from this memory directly to the application buffers in main memory. A disadvantage of this approach is that the number of virtual channels becomes limited by the size of the onboard memory (each channel must have a separate buffer).

Another optimization is to perform latency critical protocol processing on the network adapter using a separate onboard CPU. This approach is demonstrated in the VMP Network Adapter Board [KC88], and can ease the load on the system bus in the receiver.

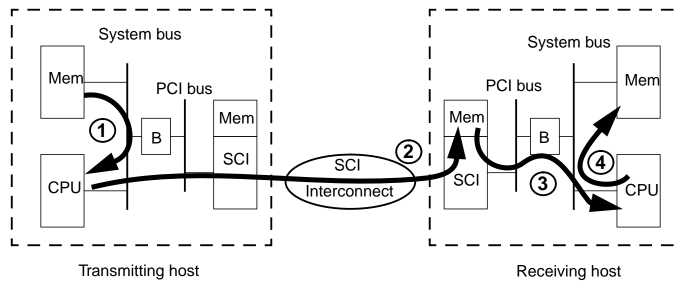

Figure 12: Architecture of a communications system optimized for message passing. The system bus is connected to the PCI bus through a bridge chip B.

5 Related work

5.1 Active Messages

Active Messages [vECGS92] address the problem of copying in the protocol stack, and enables decentralized message demultiplexing. Moving received data to the correct receiver endpoint is no longer done by the protocol stack, but by a handler routine indicated in the message itself. This allows seamless injection of data into the receiving process.

The SCILAN solution solves this problem in another way. Since each communication endpoint has its own receiver buffer, the transmitter writes data directly into the correct receiver endpoint. Thus, moving received packets to the local endpoint is no longer an issue and copying is at the minimum for socket semantics. Different virtual channels do not share hardware resources, allowing direct access to hardware without any security concerns. Active Messages adopts an event-driven programming model, and requires extensive rewriting of a large class of applications.

5.2 The GAMMA project

The GAMMA project [CC97] implements active messages on a low-cost Network of Workstations using Fast Ethernet adapters. The GAMMA software resides in the OS kernel (Linux) and must cope with overheads of kernel calls and interrupt handling. However, the GAMMA team has optimized entry points to the Linux kernel and use a particularly fast interrupt mechanism. This allows the reported 12.7 μs latency for active messages — impressive when standard PC components are used. According to our measurements, these results could not have been obtained on Windows NT. The second stage interrupt handler is invoked only after 6000 cycles (corresponding to 45 μs on a 133 MHz machine). Optimization is not possible unless the first stage interrupt handler is used, but this is explicitly discouraged by Microsoft. The GAMMA approach relies on the basic assumption that transmission errors are extremely rare in a homogenous cluster environment. 100baseT network adapters are used, providing error detection by means of the Ethernet CRC mechanism. Thus, GAMMA guarantees that garbled packets do not get through to the application. There is no support for flow control or data recovery, and lost packets are not detected. In the SCILAN approach, the SCI hardware detects all transmission errors by means of the SCI protocol [P1593] which includes hardware retries. It is important to note that GAMMA is intended for communication in a HPC setting where lost packets would never lead to incorrect computations, only deadlocks. Thus, lost packets could never generate incorrect answers, justifying the GAMMA approach.

5.3 Thread libraries

In a High Performance Computing scenario, each machine in the cluster will typically be used by a single number crunching process. This has a number of advantages which can be readily exploited on a shared memory interconnect: All local communication endpoints belong to the same single process, thus the process itself can route received packets to the correct local endpoint. Polling is a good alternative to interrupts because there is no competition for the CPU by multiple processes. By using polling we can minimize latency down to the limit dictated by the communication hardware.

To allow computation to overlap with polling based communication, a user level thread library can be used [LRBB96]. These threads are unknown to the kernel, so they are not scheduled preemptively. Instead a thread switch typically occurs when the running thread blocks waiting for some external condition (e.g. the arrival of more data). The status of the local communication endpoints are then polled, potentially making one or more threads runnable. A special polling thread provided by the library executes if no other threads are runnable. This thread is always runnable, and repeatedly polls the local communication endpoints. Synchronization of communication is done entirely through polling, thus avoiding the overhead of interrupt processing and ensuring minimum latency. This is done at the cost of practically monopolizing the CPU for use by one process. The interrupt mechanism used by SCILAN provides a solution which lies somewhere between per packet interrupts and the radical polling integrated in a user level thread library.

5.4 Berkeley Fast Sockets

The Medusa FDDI interface card described in [Mar94] has been used to implement a user space socket library very similar to ours [RAC97]. The Medusa card has 1 MB of onboard dual port VRAM which is divided into 8K blocks. These blocks are used as transmitter and receiver buffers. A system of FIFOs implement buffer management in hardware. Note that the Medusa VRAM implements the improvement discussed in section 4.9.

The Medusa card supports sending and receiving messages directly from user space by accessing the VRAM blocks and Medusa FIFO registers. This avoids costly kernel calls. However, packets to different virtual channel endpoints will be serialized through the FIFO buffer queue of the receiver. The receiver remains responsible for routing each packet to the correct local endpoint. Thus, there must be a centralized software component in each receiving host performing this routing. Such local packet routing is not required on SCILAN.

SCILAN uses the dynamic runtime linking of Windows NT to inject itself into the binary socket application code. In contrast, the socket implementation described in [RAC97] requires relinking of the application. Because object code is not normally distributed, this is in practice the same as requiring access to the source code of the application. The SCILAN implementation should be usable even with commercial software where neither object code nor source code is available.

Fast Sockets are used in the Unix environment where programs assume that socket descriptors are valid operating system file descriptors. This assumption is violated when using Fast Sockets, since socket descriptors are created by a user-level library. SCILAN does not have this problem because the Winsock API explicitly states that socket descriptors are not necessarily file descriptors.

5.5 U-Net virtual endpoints

U-Net [WBvE97] is similar to SCILAN in that virtual endpoint buffers are directly accessible in user mode. However, data is moved into these virtual endpoint buffers by software in the receiving host. This is in sharp contrast to SCILAN, where the transmitter copy operation moves data directly into the user mode virtual endpoint buffers without any action by the receiver CPU. U-Net virtual endpoint buffers are filled by the per-packet interrupt handler which copies data from buffers filled by the Network Interface Card (NIC). Such copying in the interrupt handler must be strictly limited since it may block other processing. In particular, copying large messages by PIO (such as the MTU of 1500 byte) should be avoided at interrupt time.

To achieve socket semantics with U-Net, the receiving memory bus may be traversed no less than five times in a worst case scenario: DMA from NIC to buffers in host memory (1 bus traversal), PIO by the interrupt handler from these buffers to virtual endpoint buffers (2 bus traversals), moving data by PIO on `recv` (2 bus traversals). In comparison, SCILAN requires only three bus traversals on the receiver side for socket semantics. It is important to note that U-Net is intended for active messages and not sockets. With active messages the copy on `recv` is avoided. Another advantage of U-Net is that it works on inexpensive hardware.

6 Further work

In order to support new versions of the WinSock interface, we want to adhere to standard Windows NT programming interfaces. The current version of SCILAN only supports WinSock 1.1. Our investigations into supporting WinSock 2.0 revealed a straightforward migration path through the *Service Provider Interface* (SPI) introduced by WinSock 2.0.

To insert itself between existing applications and the WinSock 1.1 library, SCILAN currently performs on-the-fly modifications to the WinSock function table built by the runtime linker for the starting process. Under WinSock 2.0 this practice is no longer required because WinSock 2.0 allows several protocol stacks to coexist as WinSock 2.0 Service Providers. For WinSock 2.0, SCILAN is simply a light weight service provider which exists alongside any service providers used for intercluster communication. This is illustrated in figure 13.

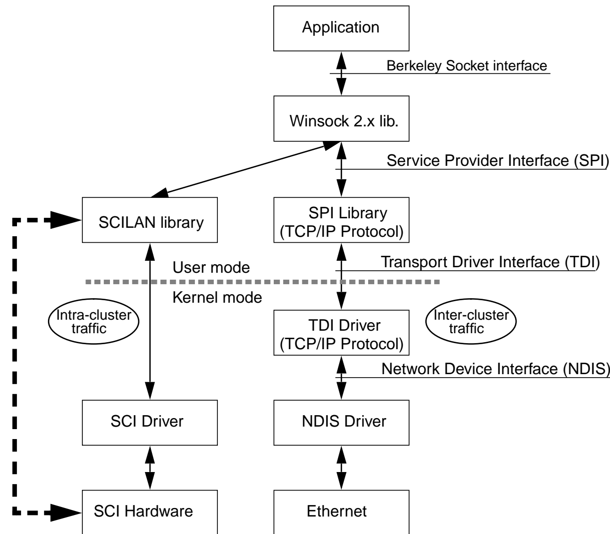

Figure 13: Generic SPI support in SCILAN

SPI is a user mode interface, allowing the SCILAN service provider to run in user mode. Since protocols may be implemented entirely in user mode under SPI, WinSock 2.0 does not mandate that socket descriptors must be valid kernel file descriptors. SPI is strikingly similar to the Berkeley Sockets interface offered by the existing SCILAN. Thus, it is straightforward to adapt SCILAN to offer an SPI interface so that it can run with any future WinSock versions.

Conclusion

We have shown how the SCILAN approach can eliminate the protocol stack for socket based communication in shared memory interconnects while maintaining IP level connectivity outside the cluster. We have demonstrated significant improvement over current practice in high speed networks for cluster area communication. The main contributions of SCILAN is very low latency communication through the widely used Berkeley socket interface. The low latency and high throughput of SCILAN is available to a large base of unmodified binary applications since we hook into the socket API through dynamic linking. The SPI interface introduced in WinSock 2.0 allows us to implement SCILAN in a standardized way. This guarantees that SCILAN will work with future versions of WinSock and Windows NT.

Using off-the-shelf hardware and operating system, SCILAN achieves a latency in the range from 16 microseconds (polling approach) to 180 microseconds (interrupt approach). The combination of low latency and high throughput gives superior performance for small packets, important for a large class of applications. Lower latency can be obtained if we switch away from the socket semantics and employ an active message strategy. Using SCI cluster adapters, we can achieve a single bus traversal in the receiver at the cost of rewriting applications. Since the receiver bus is identified as a primary bottleneck, this would be an interesting topic to pursue.

We have shown that SCILAN has very low software overhead. Application performance is limited primarily by hardware performance. Local memory copy performance is a very critical parameter for throughput. On the PC platform we have demonstrated socket throughput of more than 20 MByte/s. With more aggressive memory subsystems, such as in the UltraSPARC family of computers, we can get more than 67 Mbyte/s with the same PCI/SCI hardware. Processing in the receiver is minimized since we do not need to demultiplex received packets. This is possible since there is no single hardware resource that must be administered by a central software component to move received data into local virtual channel endpoints. SCILAN directly bridges the application interface (sockets) to the lowest layer of the protocol (SCI shared memory).

A high performance solution must avoid kernel calls and interrupts while implementing flow control at the buffer level. In SCILAN this is realized by a novel interrupt mechanism which can minimize the interrupt load on the receiving system. Communication is performed through memory mapped user level buffers without invoking the kernel, so the number of copy operations is reduced by at least one compared to a traditional approach. Only the receive copy operation is needed in addition to the inevitable transmitter copy operation. In a traditional protocol stack many copy operations are needed (from user buffers to socket buffers, between socket buffers and internal protocol processing buffers etc.). The SCILAN implementation relies on hardware supported secure delivery of SCI transactions, such that the acknowledge packets used by the TCP based solutions can be avoided.

It should be noted that the current SCILAN implementation is a research prototype, and does not implement the entire Winsock interface API. Supporting the full API requires substantial implementation effort accompanied by careful validation. We regard this work to be outside the scope of our research contribution, but note that full Winsock support is straightforward.

Acknowledgements

We are grateful for comments to this paper by the anonymous referees who spurred our investigations into the WinSock 2.0 architecture, paving the way for seamless integration of SCILAN into Windows NT. Dolphin Interconnect Solutions gave us access to SCI hardware and early versions of the SCI/NDIS solution. We are also grateful for comments from our colleagues Tarik Čičić, Stein Gjessing, Espen Klovning, Øivind Kure, Arne Maus, Frode Nilsen and Pål Spilling. We credit Knut Omang for reference measurements of UltraSparc memory copy performance. Both authors are supported by the Norwegian Research Council and the ESPRIT/OMI PROJECT: 20.761 ASCISSA.

References

[BCF+96] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, February 1996.

[BO96] Haakon Bryhni and Knut Omang. A Comparison of Network Adapter based Technologies for Workstation Clustering. In *Proceedings of 11th International Conference on Systems Engineering* , July 1996.

[CC97] Giovanni Chiola and Giuseppe Ciaccio. Architectural issues and preliminary benchmarking of a low-cot network of workstations based on active messages. In *Proceedings of (14th ITG/GI conference on Architecture of Computer Systems), Rostock, Germany*, pages 13–22, September 1997.

[DeP93] Martin DePrycker. *ATM Transfer Mode: Solution for Broadband ISDN.* Ellis Horwood Ltd., 1993.

[Dol95] Dolphin Interconnect Solutions. *SBus-to-SCI Adapter User's Guide*, version 1.8 edition, 1995.

[Dol97] Dolphin Interconnect Solutions. *PCI-1 PCI-SCI Cluster Adapter User's Guide for Windows NT 4.0*, January 1997.

[DWB+93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[GK96] Richard B. Gillett and Richard Kaufmann. Experience Using the First-Generation Memory Channel for PCI Network. In *Proceedings of Hot Interconnects IV*, pages 205–214, August 1996.

[Hor95] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, February 1995.

[Hsi96] Hsiao-keng Jerry Chu. Zero-copy TCP in solaris. In *Proceedings of USENIX Technical Conference, San Diego*, pages 253–264, 1996.

[KC88] Hemant Kanakia and David R. Cheriton. The VMP network adapter board (NAB): High performance network communication for multiprocessors. In *Proceedings of IEEE SIGCOMM '88*, pages 175–187, 1988.

[LRBB96] K.G. Langendoen, J.W. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating polling, interrupts, and thread management. In *Proceedings of Frontiers'96, Annapolis, Maryland, V.S.*, pages 13–22, 1996.

[Mar94] Richard P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects II*, 1994.

[Mes94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. (draft obtainable from `ftp://info.mcs.anl.gov/pub/mpi`), May 1994. Version 1.0.

[OP97] Knut Omang and Bodo Parady. Scalability of SCI Workstation Clusters, a Preliminary Study. In *Proceedings of 11th International Parallel Processing Symposium (IPPS'97)*, pages 750–755. IEEE Computer Society Press, April 1997.

[P1593] IEEE P1596. *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE Std 1596-1992, IEEE Computer Society, August 1993.

[Pie95] Matt Pietrek. *Windows 95 System Programming Secrets*. IDG Books Worldwide, Inc., Foster City, CA 94404, USA, ISBN 1-56884-318-6, 1995.

[RAC97] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *Proceedings of Usenix Annual Technical Conference*, 1997.

[RMG97] Stein J. Ryan, Arne Maus, and Stein Gjessing. The Design of an Efficient Portable Driver for Shared Memory Cluster Adapters. In *Proceedings of Seventh International Workshop on SCI-based High-Performance Low-Cost ComputingSanta Clara, California, USA*, March 1997.

[RMK+96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. *RFC 1577: Address Allocation for Private Internets*. Internet Engineering Task Force, 1996.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proceedings of 3rd International Symposium on High-Performance Computer Architecture*, February 1997.