# Eliminating the Protocol Stack for Socket based Communication in Shared Memory Interconnects

Stein Jørgen Ryan and Haakon Bryhni

Department of Informatics, University of Oslo
PO Box 1080, Blindern, N-0316 OSLO, Norway

{steinrya,bryhni}@ifi.uio.no

**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 [6], ServerNet [7], SCI [11], Myrinet [1] or ATM [3] 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 [2], 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.
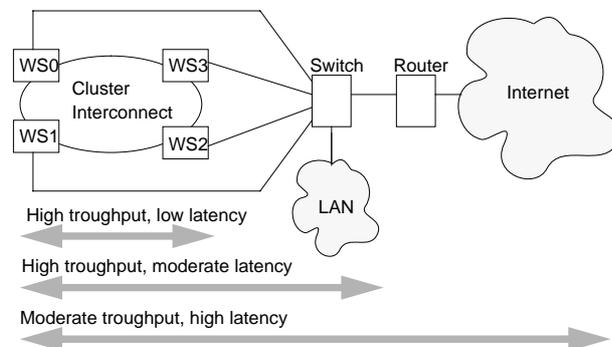


**Fig. 1.** SCILAN cluster architecture

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.

## 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.

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 [5]. 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.

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 | MPI/PVM etc | Variable | Low | No | Few |
| SCILAN | Berkeley Sockets | Proprietary | Low | No | Many |

**Table 1.** Comparison of interconnect access techniques

The best obtainable performance is achieved using a *Native API*, tuning the application to the underlying interconnect technology and its native paradigm for communication - in this case shared memory. This is shown in figure 2a. Note that the thick dotted line in the figures indicates direct access to shared memory. An example of this type of API is the Dolphin SCI API [4]. 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

---

[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.

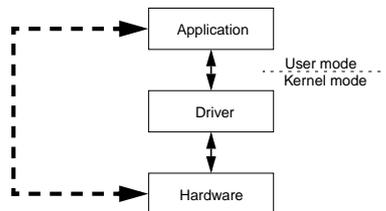technology takes the lead in price/performance, the application may have to be largely rewritten.



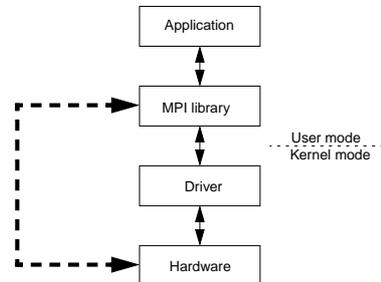**Fig. 2a.** Native technology API.

**Fig. 2b.** MPI approach for shared memory.

Another approach is to interface the new technology to a standard *High Performance Computing (HPC) API*, as shown in figure 2b. MPI [9] 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.

A third approach commonly used in local area networks, is to use the new technology at the lowest layer in a *traditional 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 [5]. 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.

### 2.1 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 3.2.

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 [12]. 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 3 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 [15].

### 3.2  Socket Based 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.
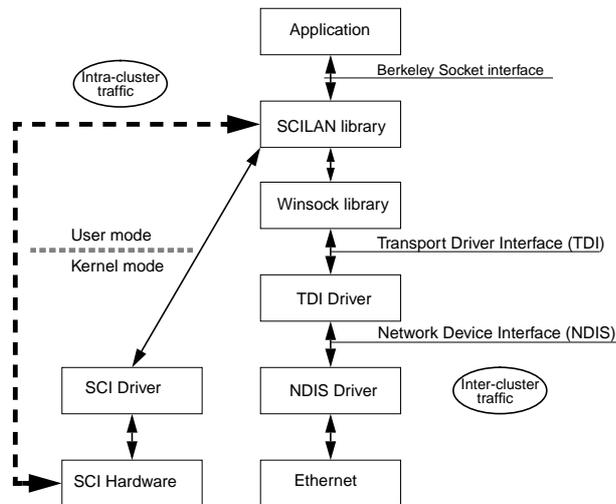
**Fig. 3.** Functional blocks in the SCILAN architecture.

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.

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). 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.

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(s)`.
2. $A$ can now invoke `accept(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. 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. Each receiver has a receiver buffer and an interrupt flag. 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.

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 [14].

## 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. The reference configuration has been two 200 MHz Pentium Intel motherboards running the Windows NT 4.0 operating system. See [14] for a detailed description of the reference configuration.

### 4.1 Factors Limiting Performance

Figure 4 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 4). 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 5a shows that on our test machines, the system bus is indeed a serious bottleneck as explained below.
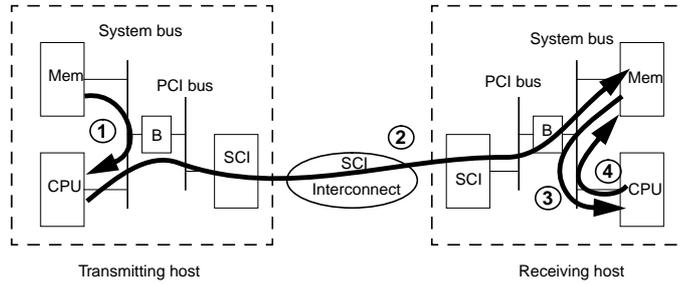
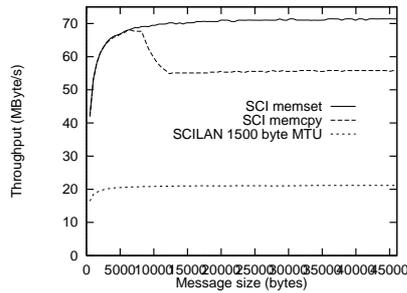**Fig. 4.** Flow of SCILAN data transfer. B denotes the PCI bus bridge chip.
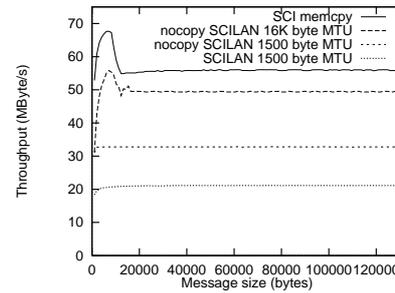


**Fig. 5a.** Copy performance



**Fig. 5b.** Effect of Flow Control and Receive copy operation

– The `SCI memset` graph shows the throughput obtained when writing a constant value into remote memory over SCI along flow number 2 in figure 4. 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 5a 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 5a 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.

## 4.2 Memory Bandwith

The bus saturation problem has also been observed on the SparcStation 20 platform [2], but is not an issue on the Ultra platform due to the improved memory bandwith [10]. As we can see from the results in table 2a, 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. [2]. This result clearly demonstrates the important effect of memory bandwith.

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

**Table 2a.** Measured throughput of Rx copy

| Technology | Latency [$\mu$s] |
|---|---|
| NDIS SCI | 394.1 |
| ATM | 304.5 |
| 100baseT | 209.3 |
| SCILAN interrupt | 179.8 |
| SCILAN polling | 16.1 |

**Table 2b.** Measured latency (on a 200 MHz system)

## 4.3 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 5b compares the throughput of the transmit copy operation with `nocopy SCI LAN` where flow control is added, but the

---

[2] Measured when transmitting from one UltraSparc-II/248 MHz to an UltraSparc-II/296 MHz with the `ctp` program used in [10]. For this speed, both rate control of the sender and flow control is required to avoid loss of data in the receiver.

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 5b), 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 5b. Data is now moved using transfer 1,2,3 and 4 in figure 4.

### 4.4 Comparing Network Throughput and Latency

Figure 6a 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. 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.
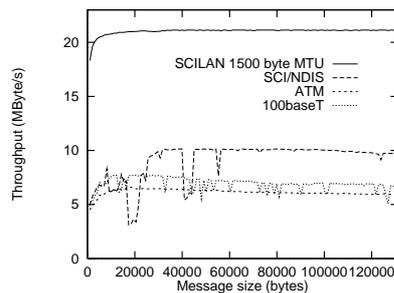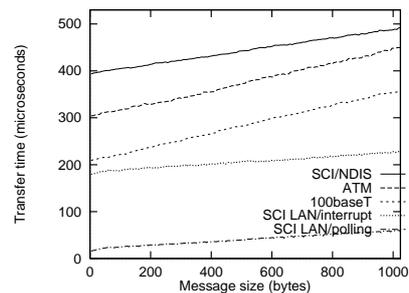


**Fig. 6a.** Comparing Throughput



**Fig. 6b.** Comparing Transfer Time

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.

A significant advantage of the SCILAN technology is the good throughput for small messages. 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 6b 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. 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.

The novel interrupt flag mechanism presented in section 3.2 facilitates a simple polling approach which can back off to interrupts after prolonged unsucessful polling. The SCILAN polling graph in figure 6b shows 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.

## 5   Related Work

We discuss the Berkeley Fast Sockets and the U-net virtual endpoints as examples of similar technologies. A more elaborate discussion of related work can be found in [14].

The Medusa FDDI interface card described in [8] has been used to implement *Berkeley Fast Sockets*, a user space socket library very similar to ours [13]. 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. 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 [13] 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.

*U-Net Virtual Endpoints* [17] 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.

## 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 current version of SCILAN only supports WinSock 1.1. The *Service Provider Interface* (SPI) introduced in WinSock 2.0 allows us to implement a new version of SCILAN in a standardized way. This guarantees that SCILAN will work with future versions of WinSock and Windows NT, further discussed in [14]. 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.

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[16] 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).

## Acknowledgements

## References

1. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. E. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second LAN. *IEEE Micro*, pages 29–36, Feb. 1996.

2. H. Bryhni and K. Omang. A Comparison of Network Adapter based Technologies for Workstation Clustering. In *Proceedings of 11th International Conference on Systems Engineering* , July 1996. Available at http://www.ifi.uio.no/~sci/papers/icse96.ps.

3. M. DePrycker. *ATM: Solution for Broadband ISDN*. Ellis Horwood Ltd., 1993.

4. Dolphin Interconnect Solutions. *SBus-to-SCI Adapter User's Guide*, ver 1.8 edition, 1995.

5. Dolphin Interconnect Solutions. *PCI-1 PCI-SCI Cluster Adapter User's Guide for Windows NT 4.0*, Jan. 1997.

6. R. B. Gillett and R. Kaufmann. Experience Using the First-Generation Memory Channel for PCI Network. In *Proceedings of Hot Interconnects IV*, pages 205–214, Aug. 1996.

7. R. W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, Feb. 1995.

8. R. P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects II*, 1994.

9. 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.

10. K. Omang and B. 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, Apr. 1997.

11. I. P1596. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Std 1596-1992, IEEE Computer Society, Aug. 1993.

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

13. S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *Proceedings of Usenix Annual Technical Conference*, 1997.

14. S. J. Ryan and H. Bryhni. SCI for Local Area Networks. Research Report 256, Department of Informatics, University of Oslo, Norway, Jan. 1998. Available at http://www.ifi.uio.no/~sci/papers/tech-rep-256.ps.

15. S. J. Ryan, A. Maus, and S. 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 Computing, Santa Clara, California, USA*, Mar. 1997.

16. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

17. M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proceedings of 3rd International Symposium on High-Performance Computer Architecture*, Feb. 1997.