High-Performance Cluster Computing Using SCI *

Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman, and Manfred Weis

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{ibel,schauser,chriss,weis}@cs.ucsb.edu
http://www.cs.ucsb.edu/research/sci

Abstract

The Scalable Coherent Interface (SCI) is a recent communication standard for cluster interconnects. We study the use of SCI in a high-performance parallel computing setting, using a cluster of UltraSparcs connected via Dolphin SCI SBus-2 adapters. We chose SCI as network fabric since it offers very low latencies and high bandwidth.

In this paper, we study how to map a variety of programming models efficiently onto the SCI hardware. We focus on message passing and global address space support, implementing Active Messages and Split-C. We present implementation trade-offs and present performance measurements. We found that the user-level load/store programming interface of SCI is very convenient to use, achieves low latencies, and is fully virtualized, simultaneously supporting multiple parallel programs and communication channels. On the other hand, neither of the programming models studied maps directly to SCI. Issues such as notification, atomic operations, and virtual address space limitations represent major implementation challenges, which we address with a combination of compiler and run-time support. Overall, we found the SCI network to form a good substrate for high-performance cluster computing.

1 Introduction

Networks of workstations (NOWs) have been tremendously successful in supporting high-performance parallel computing, offering a cost-effective alternative to custom MPP systems [ACP95]. The recent explosive growth of the Internet, as well as advances and interest in multimedia, data mining and data warehousing have created additional communication-intensive parallel applications that require cluster networks with low latency and high bandwidth. Several such networks have recently emerged into the commodity market, making it feasible to create a high-end NOW by connecting standard workstations and PCs.

In this paper, we focus on one of the new networking technologies, the Scalable Coherent Interface (SCI), and analyze its use for high-performance parallel cluster computing. We examine two important programming models: message passing and global memory.

Our testbed consists of a cluster of 8 UltraSparcs connected by Dolphin's SCI SBus-2 adapters [Dol95b], as shown in Figure 1. The SCI adapter cards provide a simple user-level load/store programming interface for accessing remote memory. Processes can export memory segments which can then be imported by other processes on remote nodes. Accesses to the imported memory via plain load/store instructions are handled by the SCI adapter cards, which send messages to access the remote data. Although the SCI standard contains a coherency protocol to allow for cache-coherent shared memory, the Dolphin adapters reside on the I/O bus and therefore do not implement cache coherency.

This user-level load/store interface is a result of more than a decade of development of network interfaces in parallel machines and workstation clusters. Traditional network interfaces rely on the operating system for protection among multiple users and for virtualizing the network. As a result, each communication event involves a system call which incurs a high overhead. However, the demand for low-latency, high-performance clusters has led to the development of more efficient approaches in the commodity market, as witnessed by the popularity of Tandem ServerNet [Hor95], Myrinet [BCF$^+$95], and the recently defined VIA interface [DR97]. A common approach to achieving low latency is to bypass the operating system for normal communication, but still ensure protection by relying on the virtual memory mechanism and using the operating system

*This work was supported by the National Science Foundation NSF CAREER Award CCR-9502661, NSF Postdoctoral Award ASC-9504291, Sun Microsystems, and the California Micro Program. Computational resources were provided by the NSF Instrumentation Grant CDA-9529418 and NSF Infrastructure Grant CDA-9216202.

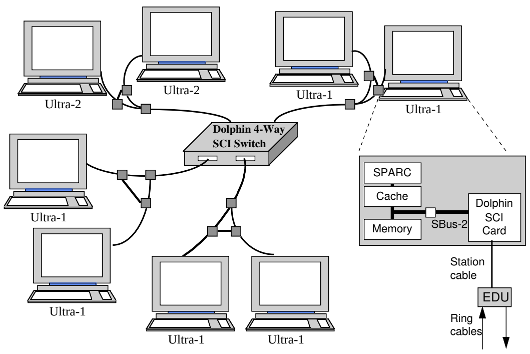

Figure 1: *The SCI cluster at UCSB. Two nodes are dual processor Ultra-2's, all others are uniprocessor Ultra-1 workstations.*

only to set up communication segments. Essentially, part of the network interface is mapped into the application's address space, such that messages can be composed and received at user-level. DEC Memory Channel [Gil96] and SCI take this approach a step further by having individual load or store instructions directly turn into remote memory accesses.

In this paper, we study how to map parallel programming models efficiently onto the SCI hardware. We focus on two common programming models: message passing and global address space. For message passing, we implement Active Messages [vECGS92]; but our results apply to any message passing scheme, since Active Messages is a low-overhead messaging layer which can, and often is, used to implement other protocols. To study global address space support, we implement Split-C, a simple parallel extension of C, which provides a global address space, global pointers, and split-phase accesses. Again, our work generalizes to other global address space models.

We discuss implementation trade-offs and present performance measurements collected on our UltraSparc cluster. We found that the user-level load/store programming interface of SCI is very convenient, achieves low latencies, and is fully virtualized, supporting multiple parallel tasks or communication channels. On the other hand, neither of the programming models studied maps directly to SCI. For example, message passing requires a remote queue abstraction to address notification and synchronization problems. Unfortunately, support for enqueuing and atomic operations is limited, therefore we must implement such a remote queue out of SCI's load/store primitives. Currently, we solve the notification problem by polling since interrupts are too expensive. Similarly, supporting a global address space often requires access to a large fraction of a remote process' virtual address space. On our platform, this is limited by the fact the SCI adapter card sits on the I/O bus, does not support cache coherence, has a limited-size address translation table, and requires the pinning of remote memory. We solve this by implementing a software TLB of global pointers and managing remote memory segments ourselves. We observe that some of these issues could be solved very efficiently with minimal hardware support, for example, a hardware remote enqueue transaction.

1.1 Outline

In Section 2 we describe the SCI hardware, the programming API, and analyze the raw performance of Dolphin's adapter cards. In Section 3 we discuss how to implement efficient message passing on top of SCI. Section 4 discusses issues arising when supporting a global address space. Section 5 presents related work. In Section 6 we conclude with an outlook on future work.

2 SCI Hardware and Raw Performance Data

SCI grew out of the desire to develop a scalable bus architecture which could support a large number of processors. SCI is an IEEE standard [IEE93] which describes a scalable coherent interface that allows a large number of nodes to be connected by point-to-point links, arranged in rings or attached to switches. While the full version of SCI also describes cache coherence and provides true shared memory support, we consider only a limited version that is currently available on workstation clusters. The Dolphin adapter cards reside on the I/O bus (SBus-2 or PCI), and therefore do not provide cache-coherence.

In this section, we give an overview of this SCI derivation. We first describe the SCI functionality and API as seen by the programmer, then discuss architectural details, and conclude with measurements and raw performance data collected on our experimental platform.

2.1 SCI from a Programmer's Perspective

The Dolphin SCI device driver provides two basic mechanisms for communication between workstation nodes: A traditional read/write system call interface that uses kernel buffering, and a more efficient load/store interface based on shared memory segments.

Dolphin's device driver supports a traditional socket-like interface: there are ioctl system calls to create, connect, disconnect and destroy channel endpoints. Simple read and write system calls can then be used to transfer data. The hardware will use a built-in DMA engine to transfer the data. Because of the high overhead of setting up the DMA, this interface is only appropriate for very large message sizes.

2.1.1 Load/Store Interface

The Dolphin SCI adapter cards also provide a simple user-level load/store programming interface for accessing remote memory. In order to use this interface, the user must first export a memory segment and import it on another node. Then accesses to the imported memory result in accesses to remote data.

The operating system is only called for initializing the memory segments. The device driver provides ioctl() system calls to create and map memory. Similarly, there are system calls for un-mapping memory segments and removing (deallocating) them. Each segment is between 8 KB and 256 KB, and exists in contiguous physical memory on the node that exports it. The memory segment is pinned down, that is, marked as non-swappable. Each memory segment has a unique identifier which is used by other nodes to import the segment. Importing a segment is a two step process. First, the node that wants to import a segment maps it into its I/O space and sets up an entry in the address translation table in the adapter card. Then, a process on that node can map the imported segment into its virtual memory space using standard mmap calls. Any loads or stores into the virtual address range of this segment are forwarded to the I/O bus, where they are handled by the SCI hardware.

Figure 2 shows a simple example with three processes on three different nodes. Process A wants to share memory with two other processes, B and C. Process A creates memory segments M1 and M2, and imports them into its own memory space. Process B only needs memory segment M1, so it just imports and maps M1 into its own address space, whereas process C maps both M1 and M2. Figure 3 shows the code segments necessary to set up the memory mapping.

The number of importable memory segments is limited due to the size of the address translation table supported on the interface card, as described in Section 2.2. Likewise, the amount of memory to export is limited by the amount of memory available for pinning down. The operating system is only involved in setting up the memory segments and mapping them to local or remote address spaces. Once pages are set up, all memory operations (e.g. load and store) are directly handled by the hardware.

2.1.2 Interrupts

Another feature of our SCI hardware is the ability to generate hardware interrupts on remote nodes, by writing a value into a special status register address (mailbox) on the target node. The current version of Dolphin device drivers does not support interrupts for user processes. In order to experiment with interrupts, we modified the device driver to trigger an interrupt on a remote user process. Once an interrupt is triggered on the target node, the SCI driver interrupt handler dispatches it and posts a signal to the target process or just schedules it immediately.

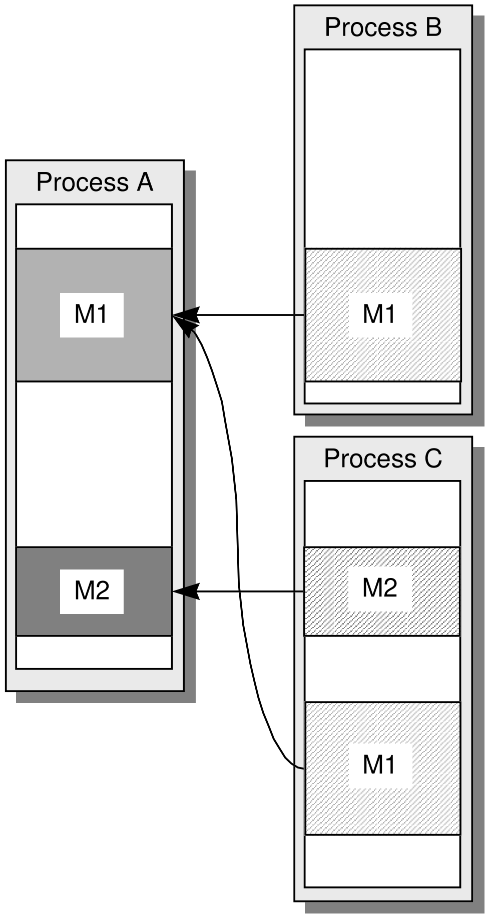

Figure 2: *The exporting and importing of shared memory segments.*

2.2 The Dolphin SCI Architecture

SCI networks are built using point to point links that form a ring on which all nodes are attached. Data passes around the ring, from node to node, until it is absorbed by the receiver node. SCI switches allow for multiply connected rings.

The SCI standard specifies four basic types of transactions: read, write, move, and lock. All transactions are split-phase, separating requests from responses. Read and write transactions send replies back, whereas move transactions are one-way. Several vendors have added or removed transactions [LCS96, Dol95a]. Our Dolphin network adapters do not support lock transactions. Also, the mapping of bus operations to SCI transactions is implementation specific. Our hardware maps all SBus write instructions (byte, 2-byte, word and double-word) to SCI write transactions — the move instructions are only performed

Process A

```
char *p1,*p2;
int size1=8192, size2=4096;

...

s1=sci_create_shm(M1,size1);
s2=sci_create_shm(M2,size2);

r1=sci_map_shm(M1,0,&p1,&size1);
r2=sci_map_shm(M2,0,&p2,&size2);

while(*p1==0)
    ;
*p1=0;

...
```

Process B

```
char *p1;
int size1=8192;

...

r1=sci_map_shm(M1,0,&p1,&size1);

*p1=1;
while(*p1==1)
    ;

sci_unmap_shm(r1);
sci_renove_shm(s1);

...
```

Process C

```
char *p1,*p2;
int size1=8192,size2=4096;

...

r1=sci_map_shm(M1,0,&p1,&size1);
r2=sci_map_shm(M2,0,&p2,&size2);

...
```

Figure 3: *Export and import of shared memory segments.*

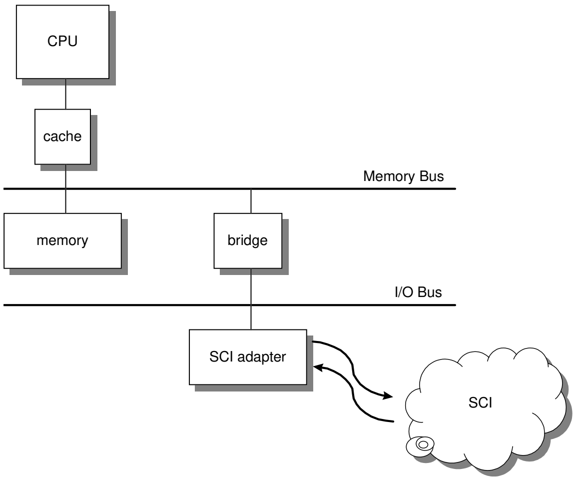

Figure 4: *Details of the SCI architecture.*

for 64-byte block stores and for DMA transfers.

SCI transactions by themselves are atomic and delivery is guaranteed, but they could be delivered out of order. Therefore, we have to be careful when sending a sequence of SCI transactions that have to be processed as a whole. They have to be separated by a write barrier. On the Dolphin cards, a simple read instruction acts as a write barrier, ensuring that all previous write or move transactions have completed.

Figure 5 shows the mapping process and the different steps a remote memory access goes through on the Dolphin SCI adapters. First, the virtual address is mapped by the MMU to a physical address in the I/O space. There it is handled by the SCI SBus adapter card. The adapter card consults the address translation table (ATT) to obtain the SCI address corresponding to the accessed memory segment. In our Dolphin SBus-2 implementation, the maximum segment size is 256 KB, so 18 bits of the 28 bits of the I/O space are used as segment offset, leaving 10 bits for identifying the segment. The hardware then looks up the target node number (16 bits as defined by the standard), and a 30 bit segment-ID from the ATT. This partial address, together with the segment offset, forms a 64 bit SCI address. When the SCI request reaches the receiving adapter card, the remote physical address is obtained by just taking the lowest 32 bits of the 64 byte SCI address.

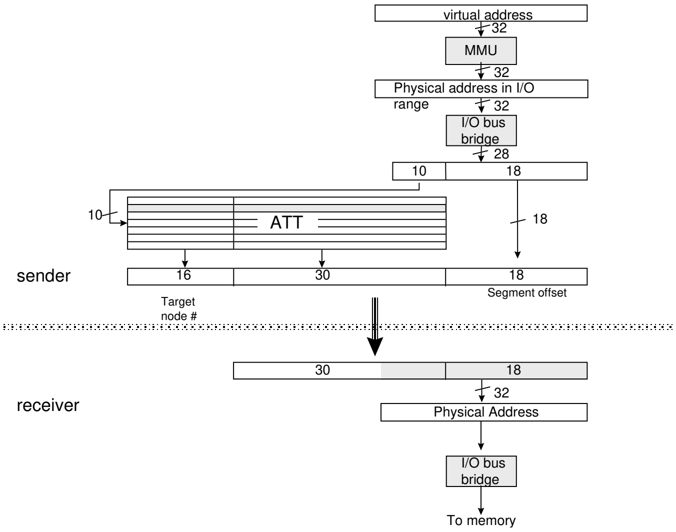

Figure 5: *Details of address translation for commodity SCI adapters on I/O buses.*

2.3 SCI Performance Measurements

In this section, we measure the raw performance of our SCI platform.

2.3.1 Our SCI Cluster Setup

The Parallel Systems Lab at UCSB has six 143 MHz Ultra-1 with 64 MB of RAM, and two dual-processor 167 MHz Ultra-2 with 128 MB of RAM. We have connected these workstations with the SCI network in various topologies, including a ring of 8 nodes, 2 rings of four workstations each, and 4 rings of two workstations each, where the rings are joined by a 4x4 SCI switch. The latter setup is shown in Figure 1. We have also used 4 Ultra-1 workstations from another lab, resulting in a 12 node cluster.

In addition to the SBus-based cluster, we have several high-end PC servers connected by Dolphin's PCI-SCI adapters. We will also augment our hardware with the new PCI-based Sun systems.

2.3.2 Latency

We measured the roundtrip latencies for read and write SCI transactions between two nodes using a ping pong program. We use 1, 2, 4, and 8-byte stores and loads, as well as 64-byte block stores supported by the UltraSparc VIS instructions set. We ran the experiments in three different settings: two nodes exchanging messages directly connected by a station cable, eight nodes connected in a ring, and the eight nodes in two rings of four, attached to two different ports of the SCI switch.

The results are summarized in Table 1 and, for two workstations directly connected by an SCI cable, show a one-way latency (calculated as half of the roundtrip latency) of $4\mu s$ for 4-byte stores and a roundtrip latency of $6\mu s$ for 4-byte reads. Because of hardware support, 64-byte block writes cost only 60% more than 4-byte writes. Note that the roundtrip write times are for two writes, ping ponged between two nodes. The read time are for a single read, since it is inherently a roundtrip communication. These results are better than the fastest measurements reported for comparable commodity networks, for example fast Ethernet ($30\mu s$) [WBvE96] or Myrinet ($10\mu s$ for 4-byte transfers as reported by Myricom).

Write RT Lat.	1 byte	2 byte	4 byte	8 byte	64 byte
SCI (2 nodes)	$7.9\mu s$	$7.9\mu s$	$7.9\mu s$	$10.5\ \mu s$	$13.3\ \mu s$
SCI (ring of 8)	$9.1\mu s$	$9.1\mu s$	$9.1\mu s$	$11.7\ \mu s$	$14.3\ \mu s$
SCI (switched)	$10.9\mu s$	$10.9\mu s$	$10.9\mu s$	$13.5\ \mu s$	$16.4\ \mu s$
Read Lat.	1 byte	2 byte	4 byte	8 byte	
SCI (2 nodes)	$6.0\mu s$	$6.0\mu s$	$6.0\mu s$	$6.3\ \mu s$	
SCI (switched)	$7.7\mu s$	$7.7\mu s$	$7.7\mu s$	$7.7\ \mu s$	

Table 1: *Roundtrip latencies for the raw communication primitives on the SCI cluster. (Two writes comprise the roundtrip; read is inherently a roundtrip communication. Our SCI hardware does not support 64-byte reads.)*

As expected, the times are fastest for two directly connected workstations. We repeated the same experiment with 8 nodes on a single ringlet, with only two of the nodes using the SCI network. We observed that the latency increases by approximately $1.2\mu s$ for 8 nodes, or about $0.1\mu s$ per SCI adapter a message passes on its routing path. This extra overhead occurs since every SCI transaction needs to be examined by every SCI adapter between the source and destination nodes. Finally, we also measured the roundtrip latency between two nodes attached to 4-way ringlets on two different ports of our 4-way switch. We measured an additional increase of $1.8\mu s$ per roundtrip, showing the switch delay to be $0.9\mu s$ in each direction.

2.3.3 Bandwidth

We measured the bandwidth of a node writing a large memory region repeatedly over the SCI network to an idle destination node. As with the latency measurements, we used varying store sizes, including 64-byte block stores. We also measured the total bandwidth when nodes are concurrently sending and receiving. Finally, we determined the throughput achievable using the embedded DMA engine and the SCI driver provided by Dolphin. Table 2 shows the achieved peak bandwidths.

Machine	Raw Bandwidth
SCI (4 byte)	2.4 MB/s
SCI (8 byte)	4.98 MB/s
SCI (64 byte)	30.4 MB/s
SCI (DMA)	25.5 MB/s
SCI (DMA, bidirectional)	36 MB/s
SCI (DMA, 2 adapters, bidirectional)	57 MB/s

Table 2: *Bandwidth for the raw communication primitives for SCI cluster the Meiko network and Myrinet. The data for 2 adapters is from [OP96].*

The bandwidth one can obtain with 4 or 8-byte stores does not exceed 5 MB/s. Using 64-byte block stores the peak bandwidth is 30.4 MB/s. With DMA, up to 25.5 MB/s are feasible. Since DMA startup time is relatively high ($200\mu s$), small messages should be transferred using 64-byte block stores. For larger messages, there is a tradeoff between the higher bandwidth of the 64-byte block stores and CPU overhead required; the DMA engine offloads the main processor.

We also measured the total bandwidth (send + receive) for a pair of nodes that bidirectionally exchange data with each other. Using DMA, we observed a peak bandwidth of 36 MB/s. Omang and Parady report experience with a multi-threaded DMA driver that can hide some of the DMA setup cost by pipelining DMA setup and DMA transfer [OP96]. They also report results for multiple SCI adapters in one node. In particular, the bandwidth of 2 adapters per node sending and receiving bidirectionally reaches 57 MB/s.

So far, we have explored and analyzed the native SCI performance in terms of latency and bandwidth. We now wish to use this raw SCI platform to build higher level programming models. In the next two sections, we explore efficient means to implement two popular higher-level programming models: message passing and a global address space.

3 Supporting Message Passing on SCI

Message passing is a frequently used programming model with many implementations: RPC, Active Messages, and libraries with traditional send&receive interface such as MPI and PVM. Many other programming models are based on message passing in one form or another (e.g. distributed shared memory systems). In this section we describe strategies for implementing Active Messages on our SCI cluster.

3.1 Active Messages and Split-C

Active Messages is a simple, light-weight protocol, yet flexible enough to efficiently implement higher level message passing abstractions like MPI. An Active Message contains the address of a handler function, which, upon arrival, is executed on the receiving processor. Message handlers must be short and execute quickly. To eliminate complicated buffer management and simplify deadlock considerations, Active Messages are divided into requests and replies. Request handlers may answer by sending a single reply message, while reply handlers are prohibited from additional communication. Active Messages achieve an order of magnitude performance improvement over more traditional communication mechanisms.

Split-C is a simple SPMD parallel extension of the C programming language [CDG+93]. Both the parallelism and data layout are explicit and are specified by the programmer. Split-C provides a global address space in the form of distributed arrays and global pointers, and supports several efficient split-phase operations to access remote data. Split-phase operations, such as get and put, separate the initiation of a memory operation (the request) from the response. Split-C implementations have been built on top of Active Messages on many computational platforms, including the TMC CM-5, Intel Paragon, Meiko CS-2, IBM SP-2, and networks of workstations. On a few machines like the T3D, Split-C has directly been built on top of the hardware. Versions that use Active Messages require implementing a remote queue abstraction.

3.2 The Remote Queue Abstraction

Under message passing, multiple messages may arrive asynchronously at any receiver. In addition, messages often require processing on the remote side. This is especially true for Active Messages, where each message handler is executed on the receiver. The receiving processor must be notified of new message arrival and synchronization requirements must be obeyed. *Remote queues* is an abstraction addressing the notification and synchronization problems associated with message delivery [SS95, BCL+95]. Sending processors just enqueue their messages on the remote queue. The receiver is notified of the message either using interrupts or polling. Since interrupts are very expensive, our implementation relies on polling. In polling, the receiving processor checks whether a message has been enqueued, and if so, removes it from the queue and processes it.

We have to emulate a remote queue under SCI. We now briefly discuss some implementation alternatives and sketch possible optimizations. More details can be found in [ISSW97].

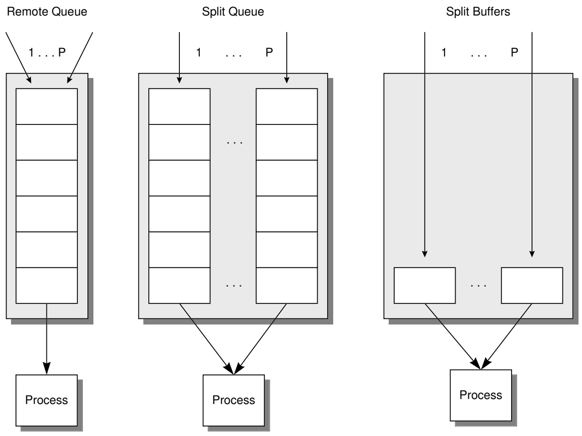

Figure 6: *Three different implementations of a remote queue.*

3.2.1 Remote Queue on Shared Memory

Figure 6 shows three possible implementations of a remote queue. On shared-memory architectures, the straightforward way to implement the queue-like functionality is to build a queue in user memory using hardware synchronization primitives. Since many processes can concurrently enqueue data, we need some sort of atomic read/write operation, such as test-and-set, to prevent write conflicts [Oma97]. The SCI standard provides support for lock transactions [Jam96].[1] Given such a lock transaction, our scheme for sending a message could be: lock the remote queue, read the destination write pointer, increment the pointer by the message size, check for overflow, write the

[1]Unfortunately, lock transactions are currently not supported by our SBus-2 network adapters.

pointer back to the receiver, put the message in the queue, then release the lock. Unfortunately, such an implementation requires many roundtrip transmissions, and might result in lock contention when many nodes send messages to the same target.

3.2.2 Split Queues

The problem with a single remote queue is that it needs to be locked to prevent multiple senders from simultaneously enqueuing messages. We can avoid this problem by maintaining a separate circular message queue for each sender on every processor, as shown in Figure 7. For each of these queues, only a single writer enqueues messages, while only the receiving node dequeues them. This approach simplifies synchronization substantially, by maintaining only a single write and a single read pointer. The simplest implementation uses queues of length one (the Split-Buffers shown in Figure 6). Using deeper queues which can hold more messages requires slightly more resources, but allows us to overlap several communication operations to the same processor. Empirically, we determined that increasing the queue length beyond 8 messages does not improve performance for most applications.

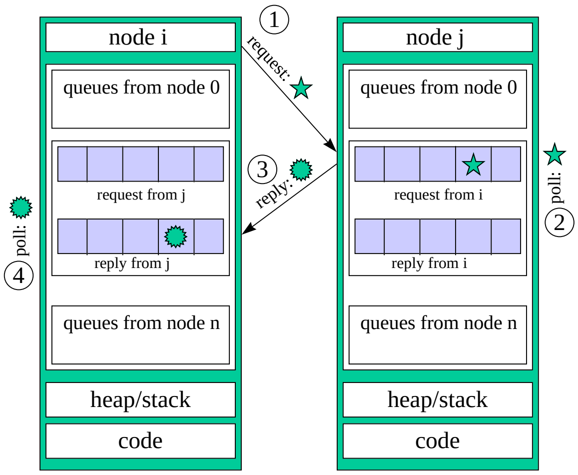

Figure 7: *Memory layout for the split queue implementation on the SCI cluster.*

3.2.3 Polling Strategies

Another important consideration is the time spent polling. In our split-queue implementation a poll has to check P incoming request buffers, as well as the reply buffers to determine whether messages have arrived or not. These checks are made in (physically local) memory locations, which are exported to other processors. Nevertheless, the time for a poll under this simple scheme grows linearly with the number of processors, which clearly is unacceptable for a large number of nodes but still very fast for our 8 processor cluster. We are currently examining several ways of improving polling, including setting a common flag, using adaptive schemes to examine "hot" incoming buffers first, or combining both polls and interrupts.

Our SCI implementation provides for remote interrupts through the use of mailboxes. By writing to a particular mailbox location, which exists in the SCI adapter card's memory, the card sends a signal to the operating system, which in turn notifies the user process. Unfortunately, using interrupts is very expensive due to context switches and OS involvement. The roundtrip time for ping-ponging an Active Message on our UltraSparc cluster using an interrupt scheme is $282\mu s$. Therefore, interrupts should only be used in exceptional cases. For example, when a node has to wait too long for an outstanding response it may want to send an interrupt to the servicing process.

3.3 Support for Long Messages

Our SCI adapter cards have a DMA engine for supporting long message transfer. Since using the DMA engine incurs a high overhead, we have also implemented alternative ways for transferring long messages.

DMA Transfers: In order to use the DMA engine we use the read/write system calls interface provided by the device driver. A node that wants to initiate a bulk transfer sends a regular Active Message to the receiver and notifies it of the upcoming data. The sender then invokes the write system call, and the receiver invokes the read call to transfer the message.

Buffered Block Moves: This scheme uses the 64-byte SCI move transactions to transfer the data from user-space into a preallocated buffer on the receiver side. The receiver, notified by a standard Active Message, copies the data into its final location and sends a confirmation back to the sender. Very long messages are transferred in buffer-sized chunks (32 KB in our implementation). Since the 64-byte block move transfers have to be aligned, we have to check if the source and destination addresses are 64 byte aligned and if not, copy the leading/trailing bytes or words separately.

Short Bulk Stores: The previous two schemes require extra messages for handshaking between the sender and receiver. This overhead is substantial for small message sizes. We have found many applications use bulk transfers with message sizes less than 128 bytes. We therefore send the data as a sequence of 64-byte Active Messages, which directly deposit the data to its final location.

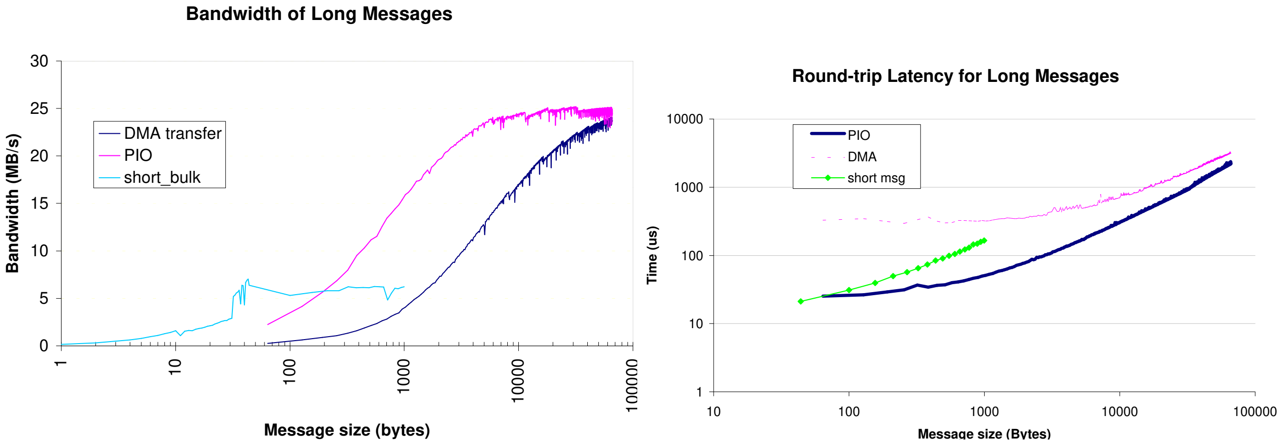

Figure 8: *Bandwidth and latency of the adaptive bulk transfer scheme. It uses short bulk transfers for message sizes up to 256 bytes, then switches over to buffered block transfers with a buffer size of 32 KB.*

Feature	Operation	SCI NOW (Split queue)	(Split buffer)	(Mapped)	CM-5	Meiko	Paragon	T3D	NOW (Myrinet)
RT Latency	AM	13.5	14.4	13.5	14.0	23	20.1	9.6	28.9
	Read	18.9	15.9	6.0	16.5	24.3	20.5	0.85	29.5
	Write	18.9	15.3	4.9	14.6	24.3	20.4	0.98	29.3
Overhead	Get	5.6	3.2	6.0	6.0	2.6	5.9	0.40	7.1
	Put	5.7	3.3	0.7	6.1	2.5	5.0	0.31	7.1
	Store	4.4	2.5	4.4	3.4	1.7	5.2	2.9	2.6
Gap	Get	5.6	12.3	6.7	6.2	15.5	8.3	0.40	7.9
	Put	5.7	12.0	1.2	6.0	15.8	8.5	0.31	7.9
	Store	5.3	11.1	5.3	3.3	13.9	8.5	2.9	5.8

Table 3: *Timing in μs for Active Messages and basic Split-C operations, for the SCI cluster and several other architectures. The numbers for the CM-5, Meiko CS-2, Paragon, T3D and Myrinet NOW are from [KSS$^+$96].*

Figure 8 shows the bandwidth and latency for all three alternatives. For messages of size up to 200 bytes, short bulk transfers are the best. For longer messages, the buffered block transfer outperforms all other alternatives. The DMA transfers, while not achieving the highest bandwidth, have the advantage of offloading the CPU.

3.4 Microbenchmarks

In order to evaluate our different implementation alternatives of Active Messages, and also to compare our implementation on SCI against other parallel platforms, we ran a set of microbenchmarks. Table 3 shows basic latency and overhead measurements for Active Messages and various Split-C primitives on top of Active Messages. We compare two remote queue implementations, of depth 1 and 8. In addition, we present an implementation of Split-C which directly uses the SCI load/store interface, avoiding Active Messages whenever possible, as discussed in Section 4.

We measured roundtrip latency, overhead (the time the sender/receiver is busy), and gap (the time between successive operations) by running a ping-pong protocol in a tight loop on two processors. The Active Messages are implemented using 64-byte block operations. The Split-C primitives used (read, write, put, get, and store) access integers, and are implemented with Active Messages. The get and put are split-phase versions of read and write, while the store is a one-way operation. The message handlers not only access data, but also modify counters to keep track of the number of outstanding requests.

We see that the roundtrip latency is slightly larger than the 13.3μs raw roundtrip latency for 64-byte block writes, as previously shown in Table 1. Looking at the measurements for the gap we see a clear distinction between the split-buffer and split-queue implementations. For the split-buffer implementation, we allow each node only one outstanding request at a time. Thus, the time for the gap is approximately the time for one roundtrip. The split-queue implementation can overlap the sending of multiple messages, and as a result the gap is very close to the overhead.

Table 3 also shows results for a variety of supercomputers, as well as for a Myrinet cluster [KSS$^+$96]. We see that

Program	Description	Problem Size	SCI Split Queue 1 proc	2 proc	4 proc	8 proc
bitonic	Bitonic sort	4M keys	14.73	12.83	11.3	8.29
Radix	Radix sort	4M keys	38.52	27.52	26.80	33.29
sample	Sample sort (bulk transfers)	4M keys	23.98	15.52	8.33	4.36
fftb	FFT (bulk transfers)	2M pts	552.8	14.70	8.76	5.81
fft	FFT (small transfers)	2M pts	123	33.08	11.85	6.09
cannon	Cannon Matrix Multiply	1024x1024 matrix	71.61	44.13	26.71	22.52

Table 4: *Run times in seconds for various Split-C programs on the SCI cluster.*

the performance of the SCI network is comparable to that found in other parallel platforms. We observe that the SCI split-buffer implementation is similar to the Meiko, while the split-queue resembles the CM-5, Paragon, and NOW implementations. The mapped implementation, which is discussed in more detail in Section 4, is very similar to the T3D.

3.5 Split-C Applications

We now briefly discuss our experience with using our cluster for a suite of Split-C applications. The benchmarks include fine grained sorting algorithms (bitonic and radix sort), a bulk-store sample sort algorithm, FFT algorithms (both fine-grained and bulk transfer versions), and cannon's matrix multiplication algorithm. The benchmark results are summarized in Table 4, and show the run-time in seconds for each program on configurations of 1, 2, 4, and 8 processors. We can see that for relatively coarse grained applications, like cannon or sample sort, we get good speedups. Very fine grained applications like bitonic and radix sort, on the other hand, do much worse. For example, bitonic sort achieves a speedup of less than two for 8 nodes. The superlinear speedups for FFT result from the high memory requirements (they need about 80 MB). An analysis of the communication pattern of the fine grained programs shows that most of the exchanged messages are very short, typically passing around objects of 4 to 32 bytes, such as integer, doubles or small structures. Since SCI already provides the hardware to directly store those objects into remote memory, it only natural to ask whether we can improve the performance for fine grained applications by using direct remote memory access instead of Active Messages whenever appropriate. The next section discusses this approach.

4 Global Address Space on SCI

A variety of parallel languages, for instance Split-C, HPF, CC++, and AC use a global address space as programming model. We focus on Split-C. While the mapping of Split-C primitives onto Active Messages is the natural thing to do on a message passing machine, shared memory machines directly support a global address space in hardware and thus further optimizations are possible. For example, the get could fetch the value from the remote memory using a plain shared memory load operation; as we saw in Table 3, this would lead to a big performance improvement. In the section, we discuss how to implement such a global address space given the limitations of our SCI hardware.

4.1 Implementation Issues

The problem with directly supporting a global address space is that parallel processes often require access to a large fraction of a remote process' virtual address space. Unfortunately, there are limitations as to how much memory is globally accessible at any given time. These limitations, which are not tied to the SCI standard itself, result from the network interface design. For example, the Dolphin SCI adapter card is attached to the SBus-2 IO bus and therefore cannot access an arbitrary virtual address in the remote process (as may be required by the Split-C semantics). Remote accesses are limited to previously allocated remote segments, each of a maximum size of 256 KB. The device driver pins these segments down in physical memory to ensure that the virtual memory manager does not swap them out.[2] This limits the amount of memory a node can export. Similarly, the network interface has a limited size address translation table, which limits the number of memory segments that can be imported. The Dolphin network interface can keep track of 1024 imported segments, each up to 256KB large. This limits the imported address space of any node at any time to 256MB. Thus, it is not feasible to allocate all of the globally addressable data structures (which may include both the heap and stack) into the remote segments.

To overcome these restrictions, we need to dynamically (re)map portions of memory. If the amount of memory we want to share is small enough, then the SCI hardware can

[2]Otherwise, the SCI adapter card would have to do a general virtual to physical memory translation and would have to be able to handle page faults on an incoming message.

easily support global addressing. However, for larger memory requirements, we need to manage the memory, keeping track of what segments should be globally accessible at any given time.

4.2 Dynamic Memory Mapping with SCI

Our approach is to maintain a software TLB on each node that translates a remote address to a corresponding imported segment in the local memory. If the translation succeeds, we can use the more efficient SCI load/store interface. In the case of a TLB miss, we send an Active Message to the remote side, as in the pure Active Message implementation of Split-C. The receiving node delivers the requested value and decides whether to export the corresponding memory segment.

A straightforward implementation would be to create and map new memory segments as they are needed. However, exporting an SCI memory segment is expensive, since all nodes in the cluster must be informed. A better way is to keep a pool of segments that have already been exported by SCI. We need to manage this pool, depending on access pattern of the global memory.

Now, instead of using the SCI device driver to export a memory segment M, the following steps are performed:

1. Find a free SCI segment M^* from the pool of previously allocated SCI segments.

2. Copy M to M^*.

3. Use mmap() to map M^* over M, so that all local accesses to M really access to M^*.

4. Send a message to the remote node and tell it to use the preestablished M' segment address corresponding to M^*.

When evicting segments, we undo the mmap, and copy the data back in its original location. Thereafter, the SCI segment can be reused. Using this scheme, the high cost of globally mapping the segment in the SCI network is only incurred once, at the start of the program when the pool of segments is allocated. The cost of our new map operation consists of the cost of `mmap()` plus the cost of copying the segment into the SCI space.

4.3 Experimental Results

We now analyze the performance of the TLB scheme outlined in the previous paragraphs.

4.3.1 Cost of Mapping Primitives

For our SCI cluster, the breakdown of times associated with all mapping primitives is shown in Table 5.

Measurement	Time
SCI allocate memory: 8KB segment	100 μs
SCI allocate memory: 100 8KB segments	200 μs
SCI allocate memory: 128KB segment	1400 μs
SCI allocate memory: 100 128KB segments	2200 μs
SCI map memory segment	188,000 μs
SCI unmap memory segment	86 μs
OS map/unmap (pin/unpin) segment	60 μs
copying 8KB of data locally	50 μs
copying 128KB of data locally	700 μs

Table 5: *Relevant times when remapping a shared segment into globally accessible memory.*

While the OS map/unmap and the SCI unmap are relatively inexpensive, mapping memory to the globally accessible memory costs 188 ms. The SCI map memory primitive is the most expensive operation, since it involves a global synchronization between all nodes. The times are taken on two directly connected nodes. For larger clusters, the cost of these operations will even be higher. Our approach solves this problem by mapping and exporting each segment only once. Thereafter, we use our software TLB to manage the segment mapping ourselves.

4.3.2 Memory Access Microbenchmarks

We run a microbenchmark consisting of a simple loop that stores values into a remote array using strided access. We measure the time per store instruction as a function of the array size and the stride. Figure 9 shows the results of three tests, corresponding to three different segment sizes (8 KB, 32 KB, and 128 KB) managed by the software TLB. The results reveal a number of interesting design details. First, we see that, when the array size is less than the total amount of exported memory, each write operation takes about 1μs. This corresponds to the minimum gap between successive writes in our SCI network, which is also 1μs since writes have a latency of 4μs and our SCI adapters can have up to four outstanding requests. Second, as expected, when the array size exceeds the total size of the exported memory, we are forced to dynamically remap segments. The cost for mapping grows with the segment size, partly due to the two memory copies that are necessary to reuse the pool of SCI exported segments.

If a segment is not mapped, we use Active Messages to request the data item. A good strategy for the receiver would be to immediately service the request and begin mapping the segment in the background. That way, the time per access will not degrade much beyond the time to process an Active Message request. As the measurements show, we currently do not implement this. For large arrays and

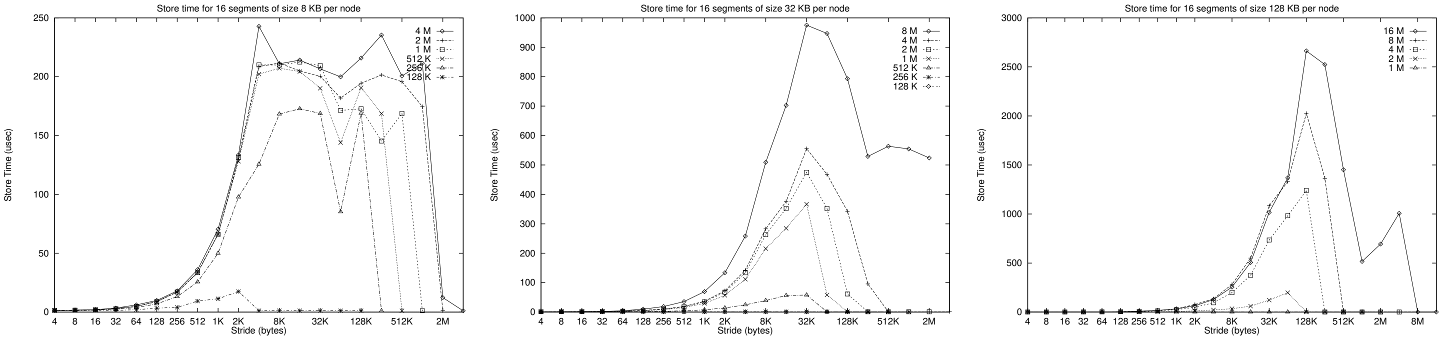

Figure 9: *Store times for different segment sizes.*

strides, the time per access is proportional to the time it takes to copy a whole segment. The reason is that the receiver currently copies the segment using a plain memcopy, during which it does not service messages because it does not poll.

5 Related Work

Several other groups are also studying SCI as interconnection network for NOWs. Gustavson and Li were the first to promote SCI in the LAMP project [GL95], and point out that SCI is well suited for networks of workstations.

Probably the most comprehensive experimental performance study of SBus-2 based SCI adapters is from Omang and Parady [Oma95, OP96]. In this study, important hardware parameters like bandwidth and latency in various configurations of two nodes, connected with one or two SCI rings, are explored, and bottlenecks like the DMA engine are highlighted. Omang also studied the support of atomic read/write operations, such as test-and-set on the IO adapter cards [Oma97]. Omang's work is part of the SCILLA project at the University of Oslo, which studies the use of SCI in workstation clusters. Another SCILLA project is to create a portable device driver for Dolphin PCI adapters on a wide range of platforms [RGL96, RMG97]. Their device driver work focuses on the support of DMA transfers and interrupts.

The SCALE cluster project [GPT$^+$96, GPTR97] strives to investigate techniques for efficient programming of workstation clusters for high performance computing. George et al. report their experiments with a multithreaded runtime system and their version of Active Messages. Their implementation has a 68μs latency, but is thread safe.

Besides Dolphin Interconnects [Dol95a], Interconnect Systems Solution [Kib97] and Vitesse are also developing commodity SCI products to connect clusters of workstations. Several other groups even build their own SCI network cards or bridges, for example the SMILE group at the University of Munich [AHKL96], or the RD24 project at CERN. SCI has been used in several parallel machines (e.g. HP Convex) and forms the basis for existing and future large scale enterprise servers (e.g. Sun Starfire [Mic96], Data General [CA96], and Sequent NUMA-Q [LCS96]). In the last two products, the nodes are connected via cache-coherent SCI on the memory bus.

There are other network architectures that are very similar to SCI, in that they provide a load/store interface to remote memory. We discuss the SHRIMP multicomputer [BLA$^+$94] and the DEC Memory Channel [GK96]. The DEC Memory Channel is very similar to SCI. It too provides a limited global memory to a cluster of workstations via network interfaces on the I/O bus. Unlike in SCI, pages are replicated if several nodes import the same memory for reading. A write to an exported page updates the replicas of this page. The size of the global memory is limited by hardware design (e.g., 128MB total). The network has a raw latency of 2.9μs for 32 byte messages, enabling higher level protocols such as MPI, PVM, and HPF to have communication latency of less than 10μs [GK96].

SHRIMP is based on a similar shared memory principle as SCI, but only supports remote writes. Like our SCI interface, SHRIMP uses an import/export mapping to keep track of global memory. The SHRIMP hardware is connected to the memory bus, not the I/O bus. In this way, the network interface can snoop all writes, and automatically forward them to a remote node if they access a page that has been exported. The work presented in [DBDF96] shows how a number of message passing libraries, in particular RPC and socket interfaces, can be implemented efficiently on SHRIMP. The fastest RPC implementation on SHRIMP yields a roundtrip time of 9.5μs.

6 Conclusions and Future Work

In this paper we described our experience with a cluster of UltraSparc workstations connected by Dolphin SCI hardware. We are excited about the capabilities of SCI, which is similar to other modern "HotInterconnects", such as Memory Channel or SHRIMP. The load/store programming interface is very convenient and easy to use, and results in very low latencies ($4\mu s$ for a one word store) and low overhead ($0.7\mu s$), which compares favorably to other cluster technologies. Although the observed bandwidth of 30 MB/s on the SBus-based adapter cards is lower than that of most dedicated parallel machines, SCI has the potential to achieve a very high bandwidth. The next generation of adapter cards based on PCI achieves a much higher bandwidth.

SCI is flexible enough to support a variety of programming models. We focused on message passing and global address space as two important programming models. We found that neither model maps directly to the SCI hardware. We addressed this by a combination of compiler techniques and run-time support in our implementation of Active Messages and the Split-C programming language. In order to support message passing, we needed to implement a remote queue. Our implementation resulted in a one-way latency of $6.7\mu s$ for Active Messages. We present two implementations of a global address space under Split-C, one which uses Active Messages for communication, and a better one which directly utilizes the SCI load/store interface to access global memory. For the latter one, we implemented a software controlled TLB, that loads or stores directly from remote memory whenever possible, and uses Active Messages only as a fallback mechanism. This scheme results in a much lower latency and overhead, approaching the raw SCI performance.

Although we found that SCI performs well, there are a number of simple extensions to the hardware that would further improve support of higher level programming models, and thus increase performance. For example, a remote enqueue transaction that takes a message and enqueues it in the destination memory, atomically updating a queue pointer, could help solving the notification problem and reducing the polling overhead [ISSW96]. A larger address translation table would facilitate the use of SCI for global address space applications and make them more efficient. Furthermore, the current SBus implementation limits the most efficient 64 byte stores to aligned data. Applications which access unaligned data have to use less efficient store primitives. The next generation of PCI SCI cards already addresses this problem by coalescing consecutive writes.

In the future, we plan to expand our current 8 node cluster to at least 32 nodes, including PCI based systems. We will study the scaling behavior of our hardware and software, and intend to examine adaptive polling techniques and hybrid schemes combining interrupts and polling. We will also explore further issues pertaining to global address space and distributed shared memory implementations. In particular, we will study how to use direct memory mapping to compensate for the lack of co-scheduling on non-dedicated environments.

Acknowledgment

We thank Albert Alexandrov and Arvind Krishnamurthy for providing us with helpful comments on this paper.

References

[ACP95] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), February 1995.

[AHKL96] G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit Per Second Local Area Network. *IEEE Micro*, February 1995.

[BCL+95] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *7th Annual Symposium on Parallel Algorithms and Architectures*, July 1995.

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21st Int'l Symposium on Computer Architecture*, April 1994.

[CA96] R. Clark and K. Alnes. An SCI Interconnect Chipset and Adapter. In *Proc. of Hot Interconnects IV*, August 1996.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[DBDF96] S. N. Damianakis, A. Bilas, C. Dubnicky, and E. W. Felten. Client-Server Computing on the SHRIMP Multicomputer. In *Hot Interconnects IV — Symposium on High Performance Interconnects*, August, 96.

[Dol95a] Dolphin. *Multiprocessor Systems Design With SCI and Dolphin Technology*. Dolphin Interconnect Solutions, 1995.

[Dol95b] Dolphin. *SBus-to-SCI Adapter User's Guide, DIS303 SBus-2*. Dolphin Interconnect Solutions A.S., 1995.

[DR97] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Hot Interconnects V — Symposium on High Performance Interconnects*, August 97.

[Gil96] R. Gillett. Memory Channel Network for PCI. *IEEE Micro*, February 1996.

[GK96] R. Gillet and R. Kaufmann. Experience using the First-Generation Memory Channel for PCI Network. In *Hot Interconnects IV — Symposium on High Performance Interconnects*, August, 96.

[GL95] D. B. Gustavson and Q. Li. Local-Area MultiProcessor: the Scalable Coherent Interface. Technical report, SCIzzl, Santa Clara University, Department of Computer Engineering, Santa Clara, California, 1995.

[GPT+96] A. George, W. Phipps, R. Todd, D. Zirpoli, K. Justice, M. Giacoboni, and M. Sarwar. SCI and the Scalable Cluster Architecture Latency-Hiding (SCALE) Project. In *Proceedings of the Sixth International Workshop on SCI-based High-Performance Low-Cost Computing*, September 1996.

[GPTR97] A. George, W. Phipps, R. Todd, and W. Rossen. Multithreading and Lightweight Communication Protocol Enhancements for SCI-based SCALE Systems. In *Proceedings of the Seventh International Workshop on SCI-based High-Perforance Low-Cost Computing*, March 1997.

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

[IEE93] IEEE, 345 East 47th Street, New York. *IEEE Standard for Scalable Coherent Interface (SCI)*, 1993.

[ISSW96] M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[ISSW97] M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-Performance Cluster Computing Using Scalable Coherent Interface. In *Seventh International Workshop on SCI-based Low-cost/High-performance Computing*, March 1997.

[Jam96] D. V. James. Combinable locks: NullSwap efficiently delays cache-line transfers. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[Kib97] K. Kibria. Prospects for new SCI interface chips. In *Proceedings of the Seventh International Workshop on SCI-based High-Perforance Low-Cost Computing*, March 1997.

[KSS+96] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[LCS96] T. D. Lovett, R. M. Clapp, and R. J. Safranek. NUMA-Q: An SCI based Enterprise Server. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[Mic96] Sun Microsystem. Sun Cluster Channel. http://www.sun.com/products-n-solutions/hw/servers/hpc/tech/interconnect.html, 1996.

[Oma95] K. Omang. Performance results from SALMON, a cluster of Workstations Connected by SCI. Technical report, Department of Informatics, University of Oslo, Norway, 1995.

[Oma97] K. Omang. Synchronization Support in I/O Adapter Based SCI Clusters. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, CANPC'97*, April 1997.

[OP96] K. Omang and B. Parady. Performance of Low-Cost UltraSparc Multiprocessors connected by SCI. Technical report, Department of Informatics, University of Oslo, Norway and Sun Microsystems Inc., 1996.

[RGL96] S. J. Ryan, S. Gjessing, and M. Liaaen. Cluster Communication Using a PCI to SCI Interface. In *IASTED 8th Int. Conf. on Parallel and Distributed Computing and Systems*, October 1996.

[RMG97] S. J. Ryan, A. Maus, and S. Gjessing. An operating system independent driver for an I/O based SCI interface. In *Proceedings of the Seventh International Workshop on SCI-based High-Perforance Low-Cost Computing*, March 1997.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[WBvE96] M. Welsh, A. Basu, and T. von Eicken. Low-Latency Communication over Fast Ethernet. In *Proceedings of Euro-Par '96*, August 1996.