1. The SCI Standard and Applications of SCI

Hermann Hellwagner

Institute of Information Technology, University of Klagenfurt
A–9020 Klagenfurt, Austria
email: hermann.hellwagner@uni-klu.ac.at
http://www.itec.uni-klu.ac.at/

1.1 Introduction

Scalable Coherent Interface (SCI) is an innovative interconnect standard (ANSI/IEEE Std 1596-1992 [26]) that addresses the high-performance computing and networking domain. This book describes in depth one specific application of SCI, namely its use as a high-speed interconnection network (often called a system area network) for compute clusters built from commodity workstation nodes. Yet, SCI's original design does not specifically address this segment and several other applications of SCI have been devised and realized.

This chapter therefore provides the context for the rest of the book by (1) introducing the basic concepts of SCI and related standards, and (2) presenting the most important other uses and implementations of SCI not covered by the contributions in this book.

In doing so, we will not delve deeply into the technology of SCI and its various implementations, but rather concentrate on the essential concepts so that the reader may acquire the basic knowledge required for the subsequent chapters and perceive the flexibility and wide applicability of SCI.

For technical details, the interested reader is referred to the literature pertaining to this chapter, most importantly the SCI standard document [26] and several survey articles [16][18]. When needed, later chapters will provide more background information on SCI, e.g., Chapter **??** on low-level protocols.

This introductory chapter is organized as follows. Section 1.2 summarizes the principal objectives and features of SCI and attempts to assess the achievements and current status of SCI. Section 1.3 presents essential concepts of the base SCI standard in some more detail and refers to some of the extensions developed so far or currently under discussion. Section 1.4 outlines four application areas of SCI, namely its use as:

- a system area network for compute clusters, which will be treated in depth in the remainder of this book;
- a memory interconnect for large-scale cache-coherent multiprocessors;
- an I/O subsystem interconnect; and
- an interconnect for high-performance data acquisition systems.

Examples of existing systems are given there. Related interconnection networks and concepts are dealt with in Section 1.5. Concluding remarks are given in Section 1.6. Other uses and future developments of SCI are also addressed in Chapter ??.

1.2 SCI Overview

1.2.1 Background

SCI has its origins in an effort of bus experts in the late 1980s to define a very high performance computer bus ("Superbus") that would support a significant degree of multiprocessing (i.e., number of processors). However, the group soon realized that backplane bus technology would not be able to meet this requirement, despite advanced concepts like split transactions and sophisticated, expensive implementations of the latest bus standards and products.

The reasons are that (1) a bus is a centralized resource and, thus, an inherent serial bottleneck that would be exacerbated by ever faster microprocessors, and (2) bus signaling is already approaching its fundamental limits (speed of light), resulting in electrically complex and expensive solutions and in short bus lengths. Both factors limit scalability to a few state-of-the-art, high-speed microprocessors that can be well served by a common bus.

As a result, the committee abandoned the bus-oriented view and developed novel, distributed solutions to overcome the shared-resource and signaling problems, while retaining the overall goal of defining an interconnect that offers the convenient services well known from centralized buses.

The resulting specification, SCI, finally approved in 1992, thus describes hardware and protocols that provide processors with the shared-memory view of buses. SCI also specifies related transactions to read, write, and lock memory locations without software protocol involvement, as well as to transmit messages and interrupts. Hardware protocols to keep processor caches coherent are defined as an implementation option. In contrast to most previous solutions, the SCI interconnect, the memory system, and the associated protocols are fully distributed and scalable: an SCI network is based on point-to-point links only, implements a distributed shared memory (DSM) in hardware, and avoids serialization in almost any respect.

1.2.2 Goals

Several ambitious goals guided the specification process of SCI and partly determined its name.

High performance. The primary objective of SCI, as of any interconnection network and associated protocols, is to deliver high communication performance to parallel or distributed applications. This comprises three aspects [14]:

- high sustained *throughput*;
- low *latency*;
- low CPU *overhead* for communication operations.

The performance goals set forth were in the range of GBit/s link speeds and latencies in the low microseconds range in loosely-coupled systems, even less in tightly-coupled multiprocessors.

Scalability. SCI was devised to address scalability in many respects, among them [17]:

- scalability of *performance* (aggregate bandwidth) as the number of nodes attached to the system grows;
- scalability of interconnect *distance*, from centimeters to tens or even hundreds of meters, depending on the media and physical layer implementation, yet based on the same logical layer protocols;
- scalability of the *memory system*, in particular of the *cache coherence protocols*, which must not have a built-in limit on the number of processors or memory modules that could be handled;
- *technological* scalability, i.e., (1) use of the same mechanisms in large-scale and small-scale as well as tightly-coupled and loosely-coupled systems, and (2) the ability to readily make use of advances in technology, e.g., high-speed links;
- *economic* scalability, i.e., use of the same mechanisms and components in low-end, high-volume and high-end, low-volume systems, opening the possibility to leverage the economies of scale of mass production of SCI hardware;
- no short-term practical limits to the *addressing capability*, i.e., an addressing scheme for the DSM wide enough to support a large number of nodes and a large memory on each node.

Coherent memory system. Caches are becoming ever more important for modern microprocessors to reduce average access time to data. This specifically holds for a DSM system with NUMA characteristics (non-uniform memory access) where remote accesses can be roughly an order of magnitude more expensive than local ones. To support a convenient programming model as known, e.g., from symmetric multiprocessors (SMPs), the caches should be kept coherent in hardware.

Interface characteristics. The SCI specification was intended to describe a standard *interface* to an interconnect that would enable multiple devices from multiple vendors to be attached and to interoperate. In other words,

SCI should serve as an *open distributed bus* connecting components like microprocessors, memory modules, and intelligent I/O devices in a high-speed system.

1.2.3 Concepts

Many of the goals have been met, but some have not. In the following, the major concepts and features of SCI will be summarized and an attempt will be made to assess the achievements and the current status of SCI, as represented by several implementations of SCI networks.

Point-to-point links. An SCI interconnect is defined to be built only from unidirectional, point-to-point links between participating nodes. These links can be used for concurrent data transfers, in contrast to the one-at-a-time communication characteristics of buses. The number of the links grows as nodes are added to the system, increasing the aggregate bandwidth of the network. The links can be made fast and their performance can scale with improvements in the underlying technology. They can be implemented in a bit parallel manner (for small distances) or in a bit serial fashion (for larger distances), with the same logical-layer protocols.

Most implementations today use parallel links over distances of a few centimeters or meters.

Sophisticated signaling technology. The data transfer rates and lengths of shared buses are inherently limited due to signal propagation delays and signaling problems on the transmission lines, such as capacitive loads that have to be driven by the sender, impedance mismatches, and noise and signal reflections on the lines. The unidirectional, point-to-point SCI links avoid these signaling problems. Since there is only a single transmitter and a single receiver rather than multiple devices (capacitive loads), the signaling speed can be increased significantly. High speeds are also fostered by low-voltage differential signals; see Section 1.3.3.

Furthermore, SCI strictly avoids back-propagating signals, even reverse flow control on the links, in favor of high signaling speeds and scalability. (A reverse flow control signal would make timing of, and buffer space required for, a link dependent on the link's distance [18].) Thus, flow control information becomes part of the normal data stream in the reverse direction, leading to the requirement that an SCI node (as described below) must at least have one outgoing link and one incoming link. Many of these link-level issues and low-level protocols are discussed in Chapter **??**.

SCI link speeds today reach 500 MByte/s in system area networks (distances of a few meters, 16-bit parallel links, clocked at 125 MHz, differential signals, CMOS technology) [9] and 1 GByte/s in closely-coupled, cache-coherent shared-memory multiprocessors (GaAs link controller) [41]; transfer rates of 1 GByte/s have also been demonstrated over a distance of about 100 meters, using parallel fiber-ribbon cables and BiCMOS link-level devices [13].

Nodes. SCI was designed to connect a large number of *nodes* (up to 64k). A node can be a complete workstation or server machine, a processor and its associated cache only, a memory module, I/O controllers and devices, or bridges to other buses or interconnects, as illustrated exemplarily in Figure 1.1. Each node is required to have a standard interface to attach to the SCI network, as described in Section 1.3. In most SCI systems implemented so far, nodes are complete machines, often even multiprocessors.

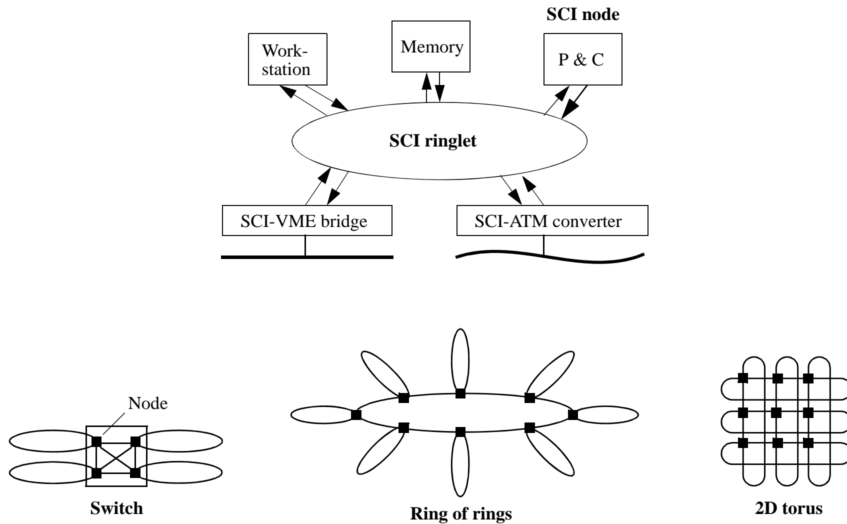

Fig. 1.1. Simple SCI network topologies

Topology independence. In principle, SCI networks with complex topologies could be built; investigations into this area are described, e.g., in Chapters **??** and **??**. However, the standard anticipates simple topologies to be used. For small systems, for instance, the preferred topology is a small ring (a so-called *ringlet*); for larger systems, topologies like a single switch connecting multiple ringlets, rings of rings, or multidimensional tori are feasible; see Figure 1.1.

Most SCI systems to date use single rings, a switch, multiple rings, or two-dimensional tori. Special applications with well-known communication patterns or very high bandwidth requirements may require specific multistage topologies to be devised, as shown exemplarily for a data acquisition system in Chapter **??**.

Fixed addressing scheme. SCI uses the 64-bit fixed addressing scheme defined by the Control and Status Register (CSR) Architecture standard

(IEEE Std 1212-1991) [25]. The 64-bit SCI address is divided into two fixed parts: the most significant 16 address bits specify the node ID (node address) so that an SCI network can comprise up to 64k nodes; the remaining 48 bits are used for addressing *within* the nodes, in compliance with the CSR Architecture standard.

While this addressing model meets the requirement of ample addressing capability, conversion from the 32-bit address space that most nodes employ today, to the 64-bit global SCI address space becomes necessary. An example of this address transformation is given in Chapter **??**.

Hardware-based distributed shared memory (DSM). The SCI addressing scheme spans a global, 64-bit address space; in other words, a physically addressed, distributed shared memory system. The distribution of the memory is transparent to software and even to processors, i.e., the memory is logically shared, just as in a system with a centralized bus and shared memory. A memory access by a processor is mediated to the target memory module by the SCI hardware.

The major advantage of this feature is that inter-node communication can be effected by simple load and store operations by the processor, without invocation of a software protocol stack. The instructions accessing remote memory can be issued at *user level*; the operating system need not be involved in communication. This results in very low latencies for SCI communication, typically in the low microseconds range.

A major implementation challenge, however, is how to integrate the SCI network (and, thus, access to the system-wide SCI DSM) with the memory architecture of a standard single-processor workstation or a multiprocessor node. Sections 1.4.1 and 1.4.2 will describe the common solutions – attaching SCI to the I/O bus or to the memory bus – in more detail.

Bus-like services. To complete the hardware DSM, SCI defines transactions to read, write, and lock memory locations, functionality well-known from computer buses. In addition, message passing and global time synchronization are supported, both as defined by the CSR Architecture; interrupts can be delivered remotely as well. Broadcast functionality is also defined.

Transactions can be tagged with four different priorities. In order to avoid starvation of low-priority nodes, fair protocols for bandwidth allocation and queue allocation have been developed. Bandwidth allocation is similar in effect to bus arbitration in that it assigns transfer bandwidth (if scarce) to nodes willing to send. Queue allocation apportions space in the input queues of heavily loaded, shared nodes, e.g., memory modules or switch ports which are targeted by many nodes simultaneously. Since the services have to be implemented in a fully distributed fashion, the underlying protocols are rather complex.

Split transactions. Like modern multiprocessor buses, SCI strictly splits its transactions into *request* and *response* phases. This is a vital feature to avoid scalability impediments; that is, it makes signaling speed independent

of the distance a transaction has to travel and avoids monopolizing network links. Transactions therefore have to be self-contained and are sent as packets, containing a transaction ID, addresses, commands, status, and data as needed. A consequence is that multiple transactions can be outstanding per node. Transactions can thus be pumped into the network at a high rate, using the interconnect in a pipeline fashion.

Most SCI systems today do make use of this feature in order to achieve high throughput rates, yet not to the full extent of 64 outstanding transactions as defined in the standard. The decoupled nature of transactions again adds complexity due to the need of keeping track of outstanding packets and more complicated flow control.

Optional cache coherence. SCI defines distributed cache coherence protocols, based on a distributed-directory approach (doubly-linked sharing list per shared, cacheable memory block), a multiple readers–single writer sharing regime, and write invalidation [24]. The memory coherence model is purposely left open to the implementor, allowing sequential consistency or more relaxed memory models to be realized in SCI systems. The standard provides optimizations for common situations such as pairwise sharing that improve performance of frequent coherence operations.

The cache coherence protocols are designed to be implemented in hardware; however, they are highly sophisticated and complex. The complexity stems from a large number of states of coherent memory and cache blocks, correspondingly complex state transitions, and the advanced algorithms that ensure atomic modifications of the distributed sharing lists (e.g., insertions, deletions, invalidations). The greatest complication arises from the integration of the SCI coherence protocols with the snooping protocols typically employed on the nodes' memory buses. Although the standard specifies three sets of coherence implementation options (the minimal set, a typical set, and the full set), the implementation is highly challenging and incurs some risks and potentially high costs. Not surprisingly, only a few companies have done implementations so far; see Section 1.4.2.

The cache coherence protocols are provided as *options* only. A compliant SCI implementation need not cover coherence; an SCI network even *cannot* participate in coherence actions when it is attached to the I/O bus as is the case in the compute clusters addressed in this book; see Section 1.4.1. Yet, a common misconception has emerged over the years that cache coherence is required functionality at the core of SCI. This misunderstanding has clearly hindered SCI's fast proliferation for non-coherent uses, e.g., as a system area network or a high-speed LAN.

Reliability in hardware. In order to enable high-speed transmission, error detection is done in hardware, based on a 16-bit CRC code which protects each SCI packet. Transactions and hardware protocols are provided that allow a sender to detect failure due to packet corruption, and allow a receiver to notify the sender of its inability to accept packets (due to a full input

queue) or to ask the sender to re-send the packet. Since this happens on a per-packet basis, SCI does not automatically guarantee in-order delivery of packets. This may have a considerable impact on software which would rely on a guaranteed packet sequence. An example is a message passing library delivering data into a remote buffer using a series of remote write transactions and finally updating the tail pointer of the buffer; see Chapter **??**, for instance. SCI hardware like Dolphin's SCI cluster adapter provides functionality to enforce a certain memory access order, e.g., via memory barrier operations; see Chapter **??**.

Various time-outs are provided to detect lost packets or transmission errors. Hardware retry mechanisms or software recovery protocols may be implemented based on the standard transmission-error detection and isolation mechanisms; these are however not part of the standard. As a consequence, SCI implementations today differ widely in the way errors are dealt with.

The protocols are designed to be robust, i.e., they should, for instance, survive the failure of a node with outstanding transactions. For this purpose, error containment and logging procedures, ringlet maintenance functions and a packet time-out scheme are specified, among other mechanisms. Robustness is particularly important for the cache coherence protocols which are designed to behave correctly even if a node fails amidst the modification of a distributed sharing list.

Layered specification. The SCI specification is structured into three layers: the physical layer, the logical layer, and the optional cache coherence layer. The latter two layers will be dealt with in more detail in Section 1.3.

At the physical layer, three initial physical link models are defined: a parallel electrical link operating at 1 GByte/s over short distances (meters); a serial electrical link that operates at 1 GBit/s over intermediate distances (tens of meters); and a serial optical link that operates at 1 GBit/s over long distances (kilometers).

Although the definitions include the electrical, mechanical, and thermal characteristics of SCI modules, connectors, and cables, the specifications were not adhered to in the early implementations. SCI systems today typically use specific physical layer implementations, incompatible to others. It is fair to say, therefore, that SCI has not become the open, distributed interconnnect system that the designers had envisaged to create.

C code. One of the most remarkable features of the SCI standard is that major portions are provided in terms of a "formal" specification, namely C code. (Exceptions are packet formats and the physical layer specifications.) Text and figures are considered explanatory only, the definitive specification are the C listings. The major reasons for this approach are that C code is (largely) unambiguous and not easily misunderstood and that the specification becomes executable, as a simulation. In fact, much of the specification was validated by extensive simulations before release. This was deemed necessary because SCI introduces new approaches and many novel, sophisticated

distributed protocols. The designers could not always be sure that their solutions were correct and feasible.

1.2.4 Discussion

SCI addresses difficult interconnect problems and specifies innovative distributed structures and protocols for a scalable distributed shared memory architecture. In doing so, the designers have come to cover a wide spectrum of bus, network, and memory architecture problems, ranging from signaling considerations up to distributed directory-based cache coherence mechanisms.

In fact, this wide scope of SCI has raised criticism that the standard is actually "several standards in one" and difficult to understand and work with. This lack of a clear profile and the wide applicability of SCI have probably contributed to its relatively modest acceptance in industry.

Furthermore, as pointed out above, the SCI protocols are complex (albeit well-devised) and not easily implemented in silicon. Thus, implementations are quite complex and therefore expensive.

Moreover, in an attempt to reduce complexity, many of the implementors adopted the concepts and protocols which they regarded as appropriate for their application, and left out or changed other features according to their needs. This use of SCI led to a number of proprietary, incompatible implementations.

As a consequence, the goal of economic scalability has not been satisfactorily achieved in general. (However, this does not necessarily hold for individual companies which may well leverage the economies of scale for their own implementation to achieve reasonable prices for their SCI products.)

As a further consequence, SCI has clearly also missed the goal of evolving into an open distributed "bus" that multiple devices from different vendors could attach to and interoperate.

However, in terms of the technical objectives, predominantly high performance and scalability, SCI has well achieved its ambitious goals. The vendors that have adopted and implemented SCI (in various flavors), offer innovative high-throughput, low-latency interconnect products (see Sections 1.4.1 and 1.4.3) or full-scale shared-memory multiprocessing systems (see Section 1.4.2).

1.3 The SCI Standard and Some Extensions

This section describes the most important concepts of SCI in further detail, with a focus on the logical and coherence layers of the standard. For other features and a comprehensive treatment, the interested reader is referred to the standard document [26]. Some of the standards extending or supporting the base SCI standard are referred to here as well.

1.3.1 Logical Layer

The logical layer specifies transaction types and protocols, packet types and formats, packet encodings, the standard node interface structure, bandwidth and queue allocation protocols, error processing, addressing and initialization issues, and SCI-specific CSRs.

Transactions. Transactions are split, consisting of a *request* and (for most transactions) of a *response subaction*. Correspondingly, the nodes involved are called the *requester* and the *responder*. A transaction is performed by sending a request packet from the requester to the responder and a response packet (if any) back to the requester. Packets carry addresses, command and status information, and data (depending on the type of transaction). Up to 64 transactions can be outstanding per node.

Each subaction consists of a *send* packet generated by the sender and an *echo* (acknowledgment) packet returned by the receiver; see Figure 1.2, packets (1) and (2), for an illustration of this *handshake* on a request subaction. The echo tells the sender whether the packet has been accepted (stored in the receiver's input queue for further processing) or rejected (e.g., due to a full input queue). In the former case, the sender can discard the send packet from its output queue (where it is required to be held); in the latter case, the sender retransmits the packet.

A consequence of this rejection/retry mechanism is that in-order delivery of packets cannot be guaranteed by the SCI hardware: a packet rejected by a busy queue can be overtaken by a later packet which happens to find space in the same queue. In addition, the retransmissions consume bandwidth and aggravate congestion on an already busy network or on the input queue of the "hot" receiver. Recent investigations have shown that this can have a significant impact on the sustained throughput of a heavily loaded SCI network; Chapter **??** reports on some of the results.

Figure 1.2 depicts the flow of packets for the more complicated case of a *remote* transaction which involves nodes sitting on different rings and one or more intermediate *agents*, e.g., bridges or switches coupling two or more SCI rings. An agent takes over responsibility for a subaction when it crosses rings, echoing the send packet on its source ring and forwarding it on to the target ring. An echo therefore is only a ring-local acknowledgement, i.e., a flow control action, not an end-to-end confirmation of the subaction. End-to-end confirmation of a transaction is only provided by a response.

Transaction types. Transactions fall into three categories: transactions with responses (*read*, *write*, and *lock* transactions), *move* transactions and *event* transactions. The transactions further differ in the amount of data they carry. An overview is given in Figure 1.3.

Transactions with responses are read, write, and lock accesses to DSM, in various flavors. Read transactions copy data from the responder to the requester, with 16, 64, or 256 bytes being transferred. In a 16-byte transaction,

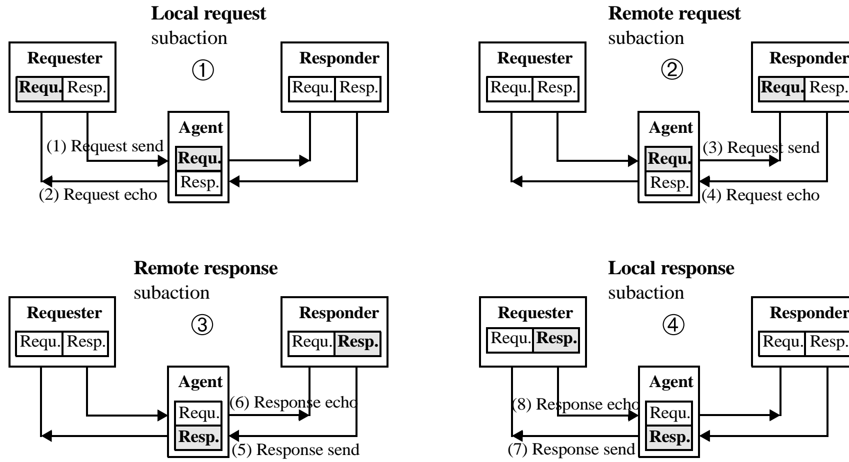

Fig. 1.2. Remote transaction phases

	Request		Response	
read*nn*	Header (16)		Header (16)	Data (*nn* = 0, 16, 64, or 256)
write*nn*	Header (16)	Data (*nn* = 16, 64, or 256)	Header (16)	
lock	Header (16)	Data (16)	Header (16)	Data (16)
move*nn*	Header (16)	Data (*nn* = 0, 16, 64, or 256)		
event*nn*	Header (16)	Data (*nn* = 0, 16, 64, or 256)		

Fig. 1.3. Transaction types (The numbers in parentheses denote the number of bytes of the packets. For space-accounting purposes, the length of the packet header includes the length of the trailer, the 2-byte CRC code.)

the size of the valid data may be between 1 and 16 bytes (selected-byte reads, *readsb*). 64 bytes, the size of a cache line in SCI, may be read in a coherent and a non-coherent manner, which distinguishes whether or not the data block is subject to coherence maintenance. Optionally, 256-byte transactions may be provided to transfer large data blocks in a non-coherent way. 0-byte reads are available as well, essentially only updating the coherence state of the addressed cache line. Besides data, responses to read transactions carry the status of the read operation.

Write transactions transfer data from the requester to the responder. The data sizes and the variants are defined as for read operations, with some minor exceptions; 0-byte writes are not supported. The responses to write requests acknowledge the success or signal the failure of the write. The 16-byte write transactions (selected-byte writes, *writesb*) are useful for manipulating control registers. The non-coherent 64-byte write operations can be employed for message passing according to the CSR standard.

The lock transaction specifies an indivisible operation (a *read–modify–write*) to be performed on the target memory location and the old memory contents to be delivered back to the requester. Such an atomic operation is required to build efficient mutual exclusion locks or semaphores in a shared memory system [24]. Early bus-based multiprocessor systems, for instance, supported the *test&set* operation by an indivisible bus and memory cycle. In a distributed system like an SCI network, the target node (the responder) must be able to perform the atomic read–modify–write operation. SCI defines a number of variants of this update operation to be provided by SCI memory controllers, among them the well-known *compare&swap* and *fetch&add* operations. The lock transaction carries data in both directions, the new (or update) value in the request packet and the old memory value in the response packet, plus status information.

Move transactions carry data from the source node to the destination node in a non-coherent way, like non-coherent writes. In contrast to write transactions, however, move operations do not have a response subaction. Consequently, moves are more efficient than writes, but correct data delivery is unconfirmed at the logical protocol layer. (There is still flow control due to the existence of the echo packet in the request subaction.) Moves are therefore expected to be used when reliable transfer is less important than timeliness, e.g., when data is written into a video frame buffer. Move transactions may address a single node (directed move, *dmove*) or multiple nodes (broadcast move). Broadcast moves and broadcast capability of nodes are specified as options only. Special protocols are defined that ensure reliable delivery of the broadcast even to nodes that are unable to accept it at first (because their receive queues are full).

Event transactions are even more special than moves in that they lack both confirmation and flow control, i.e., they have no response and do not generate an echo. Event transactions are to be accepted and delivered without

delays. Their intended use was to distribute a time stamp for global time synchronization in an SCI system; yet, they may also be used for fast data transfers under certain provisions [18].

Packets. Corresponding to the transaction phases, there are four basic types of SCI packets: *request send*, *request echo*, *response send*, and *response echo* packets. In addition, SCI specifies special packets like *init* and *sync* packets that are used during the system initialization process and for data stream re-synchronization purposes, respectively.

As an example, the format of the *request send* packet is shown in Figure 1.4. As illustrated, a packet consists of a contiguous sequence of 16-bit *symbols*. The packet header normally comprises seven symbols (14 bytes) and the trailer (CRC code) one symbol (2 bytes).

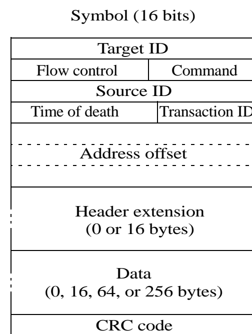

Fig. 1.4. Request send packet format (simplified)

The first symbol of the header contains the address of the destination node, *target ID*. Nodes receiving a packet on the incoming link inspect the target ID symbol to quickly determine whether to accept the packet (take it off the link) or to pass it on to the outgoing link.

The second symbol carries *flow control* information and the *transaction command*. The flow control field contains information for the ring bandwidth allocation and the queue allocation protocols, e.g., the packet priority and a field identifying whether and why the packet has been retransmitted. The command field specifies the type of the request transaction, e.g., readsb.

The third symbol provides the *source ID*, i.e., the address of the originator of the packet.

The fourth symbol specifies the *time of death* of the packet, i.e., the global time when it is to be discarded, and a 6-bit *transaction ID*. In conjunction

with the source ID, the latter allows to distinguish between up to 64 outstanding transactions per node.

The following three symbols specify the *address offset* to be used by the responder, e.g., a memory address to fetch data from.

The optional *header extension* (16 bytes) is employed by certain cache coherency transactions only.

Depending on the request type, the packet carries 0, 16, 64, or optionally, 256 bytes of *data*.

The 16-bit CRC code consumes the final symbol of the packet. The CRC polynomial and a parallel hardware implementation model are specified in the SCI standard so that CRC codes can be computed and checked at full link speeds. The flow control information of the packet is excluded from the CRC calculation since it changes during the passage of the packet through the SCI network.

As compared to the request send packet depicted in Figure 1.4, response send packets carry status information in place of the address offset. Echo packets are considerably smaller than send packets, comprising four symbols (8 bytes) only; see, e.g., Chapter **??** for a description. The special packets required for network initialization and control are eight symbols (16 bytes) long. All packets therefore consist of an integer multiple of four symbols, which simplifies the design of systems that use wider data paths, whether as internal paths or external cabled links.

Packet encodings and idle symbols. The 16-bit symbols are the basic units for packet encoding. To encode and transmit a symbol, two extra signals are needed in addition to the 16 data signals: a *clock* signal that determines symbol boundaries and a *flag* signal that delineates the start and end of packets. The well-devised use of this flag ensures that there need not be special start and stop symbols locating packet boundaries.

A parallel SCI implementation, as exemplified by Dolphin's Link Controller chip [9], therefore typically comprises 18 parallel signal lines: the 16 data signals, the clock, and the flag. For serial implementation, the SCI standard encodes these 18 signals together with additional synchronization information in a 20-bit unit.

In order to enable SCI links to run continuously and at high speeds, the space between packets is filled up with so-called *idle symbols*. Idle symbols serve two purposes: (1) they allow SCI nodes to permanently synchronize the incoming data stream to the local clock, and (2) they transfer allocation and other network control information; see, e.g., Chapter **??** for details. At least one idle symbol must be present after a "long" packet, i.e., a request or response send; echoes and special packets may be transmitted back to back. Coarsely speaking, idle symbols are created whenever a node takes a packet off the incoming link, and are replaced when a node has to send a packet. A node stripping off an idle symbol is responsible for saving and later

re-inserting some of its control information such that, e.g., the allocation protocols can work properly.

Allocation protocols. The *queue allocation* protocols ensure that space will be reserved in the input queue of the receiver so that retransmitted packets will eventually be accepted. On the other hand, the *bandwidth allocation* protocols guarantee that the sender will eventually obtain bandwidth to be able to send its packets. For a detailed discussion of these protocols and the information that is transmitted through an SCI network to facilitate their fair and efficient operation, refer to Chapter **??**.

SCI node interface structure. An SCI node, more precisely its interface to the SCI network, must perform a number of challenging tasks. For instance, it must be able to insert packets onto the outgoing link, while concurrently stripping off packets addressed to itself from the incoming link or buffering (and later re-injecting) packets destined for other nodes, all at full link speeds. Moreover, it must participate in the allocation protocols as well as network maintenance and error processing, to that end observing, manipulating, and creating the control information in packets and idle symbols. It must keep track of outstanding transactions until confirmation or until a time-out, re-transmit rejected packets, signal errors, and buffer transactions from or to the node's main components, e.g., the processor or a memory controller.

To address the complexity involved with these tasks, the SCI designers have proposed a standard interface of an SCI node, to be implemented in hardware. Figure 1.5 illustrates the interface model. Dolphin's Link Controller chip, for example, has been designed according to this model.

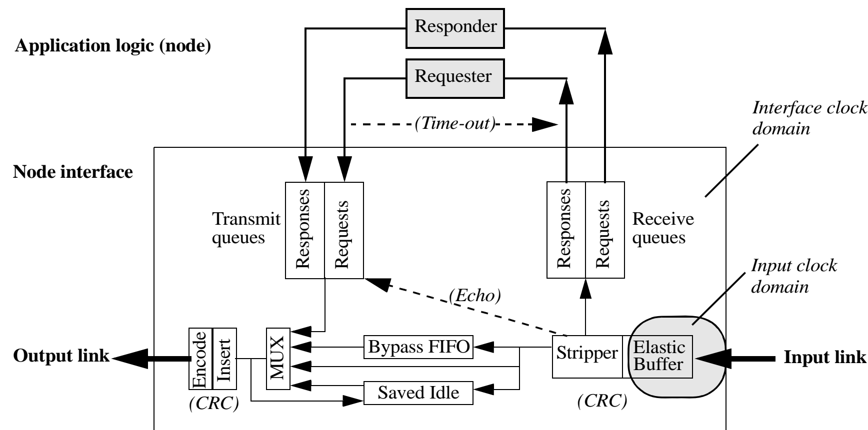

Fig. 1.5. Standard SCI node interface structure

The node interface has an incoming SCI link and an outgoing SCI link. On the input link, symbols arrive asynchronously w.r.t. the interface clock and carry their own clock signal. The first stage of the interface is therefore to synchronize the incoming data stream to the local clock domain in an asynchronous circuitry called the *elastic buffers*. Idle symbols are inserted or deleted in case the two clocks drift too far apart, subject to the rule that packets remain intact. The rest of the node interface is strictly synchronous, greatly simplifying its design.

Packets addressed to the local node are taken off the link by the *stripper* circuit and appended to one of the receive queues for further processing by the node's main components (application logic). These are symbolized by a requester block (e.g., a processor) and a responder block (e.g., main memory). In case the packet is destined for another node, it is forwarded towards the outgoing link.

If, during the arrival of a packet, the local node is inserting a packet from a transmit queue onto the output link, the arriving packet must be buffered in order not to be lost. Storage for this purpose is called the *bypass FIFO*. A node is only allowed to inject one of its own packets onto the outgoing link whenever it has detected the bypass FIFO to be empty. Thus, the bypass FIFO must be designed to hold one packet of maximum size. As soon as the output link becomes available again, the bypass FIFO is emptied.

Incoming idle symbols are consumed whenever the node sends packets of its own. The buffer *saved idle* serves to store the arriving control and allocation information. Successive idles are merged into this buffer according to the fairness requirements of the allocation protocols. Idles are produced from this buffer whenever the receiver strips off a packet for the local node.

In general, the SCI node interface maintains two pairs of FIFO *queues*, serving as buffers until transmission bandwidth becomes available for outbound packets or until inbound packets can be processed by the node, respectively. Separate queues are required in order to avoid deadlocks in case of heavy input and output traffic and the danger of queue overflow.

A request or respond send packet needs to be held in its transmit queue until the corresponding echo arrives. When the echo arrives, the stripper signals this to the transmit queue. Depending on the type of the echo, the send packet is either discarded from the output queue (done echo) or needs to be re-sent (retry echo). Time-outs are used to deal with send packets whose echoes are overdue.

Error handling. SCI provides several features on the hardware and protocol levels to detect and isolate errors. Among them are: the CRC code and its calculation in hardware; hardware time-outs to detect damaged or lost packets; error status code fields in some packet types; the time of death that may be associated with a packet; distributed error logging according to the CSR Architecture; and the concept of CRC "stomping". The latter feature means that the CRC code of a packet is modified in a recognizable way if a

problem during packet creation is encountered (and part of the packet is in transit already); an example of such a packet is a request echo that, for latency reasons, is generated during the receipt of a request send packet, which eventually is detected to have an invalid CRC code. An error is to be logged whenever a packet is stomped which allows to isolate the source of the error.

Ringlets are required to have a so-called "scrubber" node which monitors ringlet activity and is responsible for network-maintenance functions, e.g., deleting corrupted or stale packets and idle symbols, handling packets with addressing errors, and circulating ring-maintenance information. A single scrubber node is identified and activated automatically during system initialization.

These features mainly contribute to detection, containment, and logging of errors. It is beyond the scope of the standard to specify how to handle and recover from errors. Software is expected to deal with these issues; see, e.g., Chapters **??** and **??**.

1.3.2 Cache Coherence Layer

The cache coherence layer provides concepts and hardware protocols that allow processors to cache remote memory blocks while still maintaining coherence among multiple copies of the memory contents. Since an SCI network no longer has a central resource (i.e., a memory bus) that can be snooped by all attached processors to effect coherence actions, distributed directory-based solutions to the cache coherence problem had to be devised.

The resulting coherence protocols are quite complex. However, their implementation is *optional* only, not impairing system performance when they are not implemented or not being used. Further, *within* the cache coherence protocols, there are different sets of interoperable protocols: the *minimal* set, the *typical* set, and the *full* set. The implementor will choose one according to a cost-versus-performance trade-off.

The coherence protocols basically operate according to the multiple readers–single writer sharing model and use the write-invalidation scheme; no coherence model is prescribed by the standard.

At the core of the cache coherence protocols are the *distributed sharing lists* shown in Figure 1.6. Each shared block of memory has an associated distributed, doubly-linked sharing list of processors that hold a copy of the block in their local caches. The memory controller and the participating processors cooperatively and concurrently create and update a block's sharing list.

The coherence tags and related information required for coherence maintenance are depicted in Figure 1.6 as well. With the standard size of an SCI memory/cache block (64 bytes), this results in only a few percent storage overhead in the main memory and caches. The overhead is *fixed*, i.e., independent of the system size, which is an important building block of the scalability of the coherence protocols. Note that cache coherence can be enabled

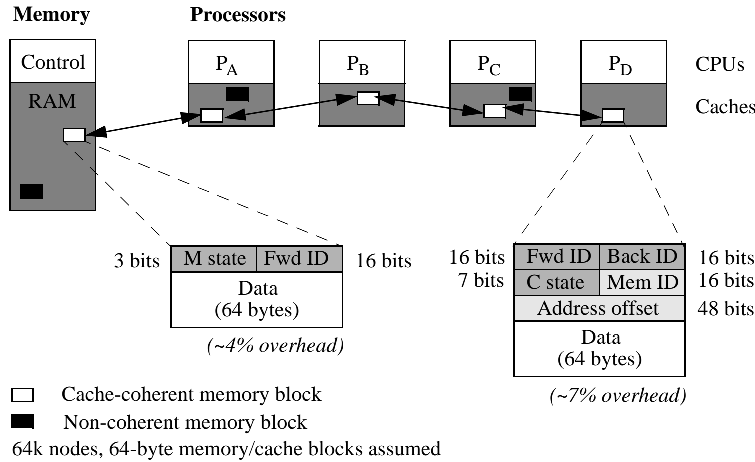

Fig. 1.6. Sharing list and coherence tags of SCI cache coherence protocols

or disabled on portions of the memory, typically on a per-page basis. Memory blocks can therefore be cached in a coherent or non-coherent manner, or be uncached at all.

Each shared, coherent memory block has a 3-bit *memory state* field and a 16-bit *forward pointer* that holds the SCI node ID of the node at the head of the sharing list. The cache entries contain a 7-bit *cache state* field, a *backward pointer* and a *forward pointer* (node IDs) for list maintenance, and the full address of the cached data block, consisting of the SCI node address of the memory module, *memory ID*, and the *address offset* within that module.

It is beyond the scope of this introduction to describe in detail the states and state transitions of the memory and cache blocks, the coherent transactions and the actions and issues involved with updating the sharing lists. Only the basic principles are given in the sequel.

When a processor accesses a cacheable, coherent memory block, e.g., by a coherent-read transaction, the memory controller updates the state of the block and saves the address of the requesting processor in its forward pointer field.

If no cached copies of the block exist at that time (i.e., the sharing list is empty), the requester becomes the head of a new list; the memory ships the data block and the requesting node initializes its backward pointer, cache state, and address fields.

If a sharing list for the block already exists at the time of the request, the requesting node becomes the new head of the sharing list as follows. The memory controller updates its memory state and the forward pointer to point to the requester (the new head); it returns the pointer to the old head back

to the requester. Note that the memory does not ship the data block to the requester since it may have a stale copy of the data only. The requester uses this information to update its coherence tags and pointers to become the new head (provisionally); it further sends a transaction to the old head to identify itself as the new head and, simultaneously, request the data block. The old head updates its backward pointer, thereby degrading itself to a regular list entry, modifies its cache state, and returns the (up-to-date) data block from its cache to the requester. The requester finally loads the data into its cache and definitively establishes itself as the new list head.

When a processor requires write access to a shared data block, it first makes itself the new head of the block's sharing list, since only the head of the list has write permission. It then deletes all other entries down the list in order to obtain an exclusive copy of the block; in other words, all other processors have to discard their copies of the block from their caches. Depending on when the node at the head of the list is actually permitted to modify the data, different coherence models can be realized; under a sequential consistency regime [24], the node would have to wait until purging the rest of the list is completed, while a more relaxed consistency model would allow the head to proceed before completion of the deletion process.

Notice that the list insertion sequence described above involves three nodes, two transactions with responses (requester $\rightleftharpoons$ memory and requester $\rightleftharpoons$ old head), and transitory states to prepend the requesting node to the sharing list. Furthermore, other nodes may concurrently request access to the same memory block and must be added to the block's sharing list as well; the order of the list insertions is defined by the arrival times of the requests at the memory controller.

Despite the distributed nature of the protocol, both in terms of space and time, and the concurrency issues involved, list insertions (and deletions as well) must appear atomically. (SCI protocols consistently use *compare&swap* to achieve atomicity.) This requirement is further exacerbated by the fact that transactions (or subactions thereof) may fail, potentially leaving nodes or sharing lists in inconsistent states unless provisions for recovery are defined.

While the above discussion disclosed only a small part of coherence maintenance actions, it should have become clear that the designers of SCI had to deal with many difficult and subtle correctness and robustness problems pertaining to the cache coherence protocols. This explains the complexity of the coherence specification. Simulations of the specification (given by the C code) have greatly helped in designing and debugging the protocols. Despite the well-devised protocols and the C code, implementation of SCI cache coherence, and especially interfacing it to typical processor coherence schemes, remains a major challenge, though; see, e.g., Section 1.4.2.

The more advanced coherence protocol sets introduce optional optimizations for important sharing or access conditions. A good example is *pairwise sharing* where, e.g., one producer and one consumer share a data block; in this

case, data may be transferred directly from the producer's cache to the consumer's cache, without manipulating the sharing list on each modification and without involving memory. A further example is *queued-on-lock-bit (QOLB) synchronization* which provides FIFO access to an exclusive resource; the sharing list structure is exploited to hand over the resource without needless communication.

1.3.3 Extensions

A number of projects have been started to develop standards that extend or support SCI in a *compatible* way. The rationale for these projects was mainly to specify issues and mechanisms that were regarded to be important, but could not be covered by the base standard anymore, or that could be postponed until the basic concepts were finished.

Some of these projects have successfully produced approved standards, among them:

- *Low Voltage Differential Signals (LVDS) for SCI* (IEEE Std 1596.3-1996) [27] that defines low-voltage differential signals (250 mV swing) and signal encodings for data paths that are 4, 8, 32, 64, or 128 bits wide; and
- *Shared-Data Formats Optimized for SCI* (IEEE Std 1596.5-1993) [28] which defines formats for data transfers among heterogeneous computing systems in a distributed environment based on SCI.

A nearly completed standards project (IEEE P1596.8) specifies the cables and connectors that have become favored by SCI users. Another successful standardization effort seems to be the *SCI Physical Layer API* project (IEEE P1596.9) which specifies a general-purpose shared-memory hardware abstraction layer designed for SCI (but usable on other shared-memory systems as well). This upcoming standard describes functionality required to set up and run DSM systems with little overhead; an in-depth discussion is given in Chapter **??**.

A standardization project that has been active for a long time is the attempt to specify *Cache Optimizations for Large Numbers of Processors using SCI*, most often called *KiloProcessor Extensions* (IEEE P1596.2). This project has been motivated by the observation that, due to the linear structure of the sharing lists, SCI cache coherence actions will exhibit poor performance when a large number (perhaps thousands) of nodes share a single memory/cache block. Further, congestion can arise in the network or at the memory controller when many nodes concurrently access a given data block. The KiloProcessor Extensions address this deficiency by providing tree-like sharing structures compatible with the linear sharing lists of SCI, and by supporting more efficient mechanisms for distribution of widely shared data and for purging copies, e.g., by using write-update mechanisms and combinable operations when possible. The aim is to reduce data-access latencies to

an order logarithmic in the number of participating processors rather than linear. A description of the concepts under discussion can be accessed at [35].

A highly promising extension of the SCI standard was the *High Speed Memory Interface (SyncLink)* standardization effort (IEEE P1596.7-199X), until recently driven by SLDRAM, Inc., a non-profit corporation sponsored by most of the DRAM manufacturers. The SyncLink project specified a high-bandwidth, synchronous-link interface optimized for interchanging data between a memory controller and DRAM chips. The mechanisms are described in a draft standard [29]. Although the technology did work, the companies supporting SLDRAM have withdrawn recently. The reasons and the future prospects of the technology are briefly addressed in Chapter **??**.

1.4 Applications of SCI

SCI was originally conceived as a shared-memory interconnect and a first implementation of such a network emerged in 1994 (in the HP/Convex Exemplar SPP multiprocessor). However, SCI's flexibility and performance potential also for other applications, e.g., as a system area network, was soon realized and leveraged by industry. This section introduces several of these "classical" applications of SCI and provides examples of commercial systems exploiting SCI technology. Upcoming applications and more recent developments, e.g., SyncLink [29] and SerialPlus [30], are covered in Chapter **??**.

1.4.1 System Area Network for Clusters

Compute clusters, i.e., networks of commodity workstations or PCs, are becoming ever more important as cost-effective parallel and distributed computing facilities. An SCI system area network can provide high-performance communication capabilities for such a cluster. In this application, the SCI interconnect is attached to the I/O bus of the nodes (e.g., PCI) by a peripheral adapter card, very similar to a LAN; see Figure 1.7. In contrast to a LAN though (and most other system area networks as well), the SCI cluster network, by virtue of the common SCI address space and associated transactions, provides hardware-based, *physical* distributed shared memory. Figure 1.7 shows a high-level view of the DSM. An SCI cluster thus is more tightly coupled than a LAN-based cluster, exhibiting the characteristics of a NUMA (non-uniform memory access) parallel machine.

The SCI adapter cards, together with the SCI driver software, establish the DSM as depicted in Figure 1.8. (This description pertains to the solutions taken by the Dolphin SBus-SCI and PCI-SCI adapter cards; see Chapter **??**.) A node that is willing to share memory with other nodes (e.g., A), creates shared memory segments in its physical memory and exports them to the SCI network (i.e., SCI address space). Other nodes (e.g., B) import these DSM

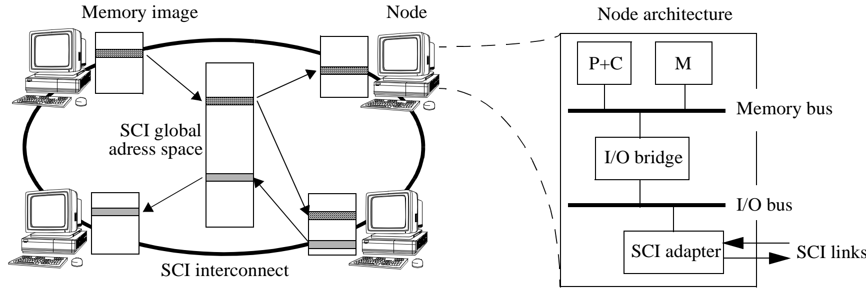

Fig. 1.7. SCI cluster model

segments into their I/O address space. Using on-board address translation tables (ATTs), the SCI adapters maintain the mappings between their local I/O addresses and the global SCI addresses. Processes on the nodes (e.g., i and j) may further map DSM segments into their virtual address spaces. The latter mappings are conventionally being maintained by the processors' MMUs.

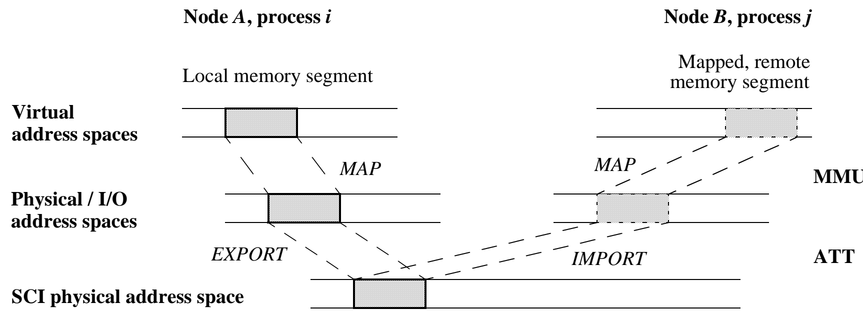

Fig. 1.8. Address spaces and address translations in SCI clusters

Once the mappings have been set up, internode communication may be performed by the participating processes at *user level*, by simple CPU load and store operations into DSM segments mapped from remote memories. The SCI adapters translate I/O bus transactions that result from such memory accesses into SCI transactions, and vice versa, and perform them on behalf of the requesting processor. Thus, remote memory accesses are both transparent to the requesting processes and do not need intervention by the operating system. In other words, no protocol stack is involved in remote memory accesses, resulting in low communication latencies even for user-level software.

Currently there is only one commercial implementation of such an SCI cluster network, the SBus-SCI and PCI-SCI adapter cards (and associated switches) offered by Dolphin Interconnect Solutions; Chapter **??** describes the adapter cards in detail. These cluster products are used by other companies to build key-turn cluster platforms; examples are Sun Microsystems, offering high-performance, high-availability server clusters [10], and Scali Computers and Siemens providing clusters for parallel computing based on MPI (see Chapters **??** and **??** as well as [37]). The research and development projects and products described in this book are to a large extent based on Dolphin's SCI cluster technology.

In addition, there are two research implementations of PCI-SCI adapter cards, one developed at Technische Universität München and described in Chapter **??**, the other one developed at CERN as a prototype to investigate SCI's feasibility and performance for demanding data acquisition systems in high-energy physics [2].

The description of an SCI cluster interconnect given above does not address many of the low-level problems and functionality that the implementation has to cover. For instance, issues like the translation of 32-bit node addresses to 64-bit SCI addresses and vice versa, the choice of the shared segment size, error detection and handling, high-performance data transfers between node hardware and SCI adapter, and the design and implementation of low-level software (SCI drivers) represent a spectrum of research and development problems. Parts II and IV of this book describe these problems and corresponding solutions in some detail. In addition, there is a wide design space for SCI interconnection networks which is explored by some of the chapters in Part III.

As an example of specific functionality and characteristics of SCI cluster hardware, consider Dolphin's recent PCI-SCI adapter cards. There are two ways to transfer data to and from the SCI network. The first method works as described above: a node's CPU actively reads data from (writes data to) a remote memory using load (store) operations into a DSM address window mapped from the remote node; this can be fully done on user level, resulting in round-trip latencies as low as 4–5 μs, but occupying the CPU for moving the data. The second method involves a direct memory access (DMA) engine on the SCI adapter that copies the data into and out of the node's memory; while this approach relieves the CPU, it has higher startup costs since the SCI driver software has to be involved to set up the DMA transfer. The SCI adapters provide several features optimizing the performance of the data transfers into and out of the node, among them using PCI burst operations, combining consecutive small SCI transactions addressing a contiguous memory region into a larger transaction, and prefetching. Under ideal circumstances (see Chapter **??**), throughput of more than 80 MByte/s into and out of a node can be achieved, limited by the memory and I/O architecture

of the node. The SCI interface chips and links can provide a bandwidth of 500 MByte/s.

An important property of such SCI cluster interconnect adapters is worth pointing out here. Since an SCI cluster adapter attaches to the I/O bus of a node, it cannot directly observe, and participate in, the traffic on the memory bus of the node. This therefore precludes caching and coherence maintenance of memory regions mapped to the SCI address space. In other words, remote memory contents are basically treated as non-cacheable and are always accessed remotely. Current SCI cluster interconnect hardware does not implement cache coherence capabilities therefore. Note that this property raises a performance concern: remote accesses (round-trip operations such as reads) must be used judiciously since they are still an order of magnitude more expensive than local memory accesses.

The basic approach to deal with the latter problem is to avoid remote operations that are inherently round-trip, i.e., reads, as far as possible. Rather, remote writes are used which are typically buffered by the SCI adapter and therefore, from the point of view of the processor issuing the write, experience latencies in the range of local accesses, several times faster than remote read operations. In Part V of the book, several chapters describe how considerations like this influence the design and implementation of efficient message-passing libraries on top of SCI.

Several chapters in Part VI deal with how to overcome the limitations of non-coherent SCI cluster hardware, in particular for the implementation of shared-memory and shared-object programming models. Techniques from software DSM systems, e.g., replication and software coherence maintenance techniques, are applied to provide a more convenient abstraction, e.g., a common *virtual* address space spanning all the nodes in the SCI cluster.

1.4.2 Memory Interconnect for Cache-Coherent Multiprocessors

The use of SCI as a cache-coherent memory interconnect allows nodes to be even more tightly coupled than in a non-coherent cluster. This application requires SCI to be attached to the memory bus of a node, as shown in Figure 1.9. At this attachment point, SCI can participate in and "export", if necessary, the memory and cache coherence traffic on the bus and make the node's memory visible and accessible to other nodes. The nodes' memory address ranges (and the address mappings of processes) can be laid out to span a global (virtual) address space, giving processes transparent and coherent access to memory anywhere in the system. Typically, this approach is adopted to connect multiple bus-based commodity SMPs to form a large-scale, cache-coherent (CC) shared-memory system, often termed a CC-NUMA machine.

There are three notable examples of SCI-based CC-NUMA machines: the HP/Convex Exemplar series, now in its second generation [6][40], the Sequent NUMA-Q multiprocessor [32][7], and the Data General AViiON scalable servers [5]. The latter two systems comprise bus-based SMP nodes with

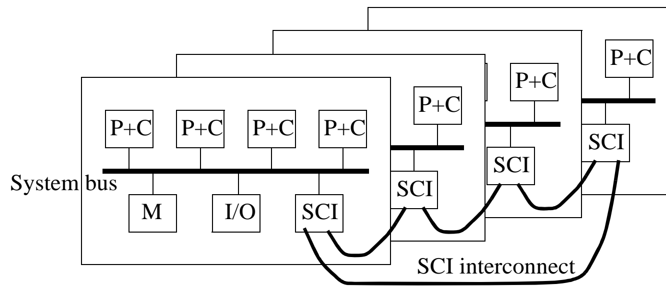

Fig. 1.9. SCI-based CC-NUMA multiprocessor model

Intel processors, while the Exemplar uses HP PA-RISC processors and a non-blocking crossbar switch within the nodes. The inter-node memory interconnects are proprietary implementations of the SCI standard, with specific adaptations and optimizations incorporated to ease implementation and integration with the node architecture and to foster overall performance.

The major challenge in building a CC-NUMA machine is to bridge the cache coherence mechanisms on the intra-node interconnect (e.g., the SMP bus running a snooping protocol) and the inter-node network (SCI). Since the *Sequent NUMA-Q* machine is well documented, it is used as a case study to illustrate the essential issues in building such a bridge and making the protocols interact correctly. A block diagram of the adapter board coupling the node bus and the SCI network is given in Figure 1.10; Sequent's SCI implementation is called *IQ-Link* [33].

The NUMA-Q machine interconnects commodity bus-based SMP nodes by a ring structure, with typically four Intel Pentium Pro or Xeon processors per node. Every processor in the system has a common view of the system-wide memory and I/O address space. The IQ-Link board plugs into the node bus and participates in the standard MESI bus snooping protocol on behalf of remote nodes.

The essential building block of the interface is the *remote cache*. This is a large (32 MByte in first-generation systems [32]), expandable, 4-way set-associative cache, maintaining copies of blocks that are fetched from remote memories and representing the node to the rest of the system. The remote cache is kept coherent with local processor caches by means of the bus-snooping protocol, and with processor caches on remote nodes and remote memories by SCI's distributed directory protocol. The SCI directory protocol only operates on the remote cache; it is unaware of the local processor caches. Vice versa, the local caches are not aware of other nodes; they are supplied with remote data from the remote cache by virtue of the snooping protocol.

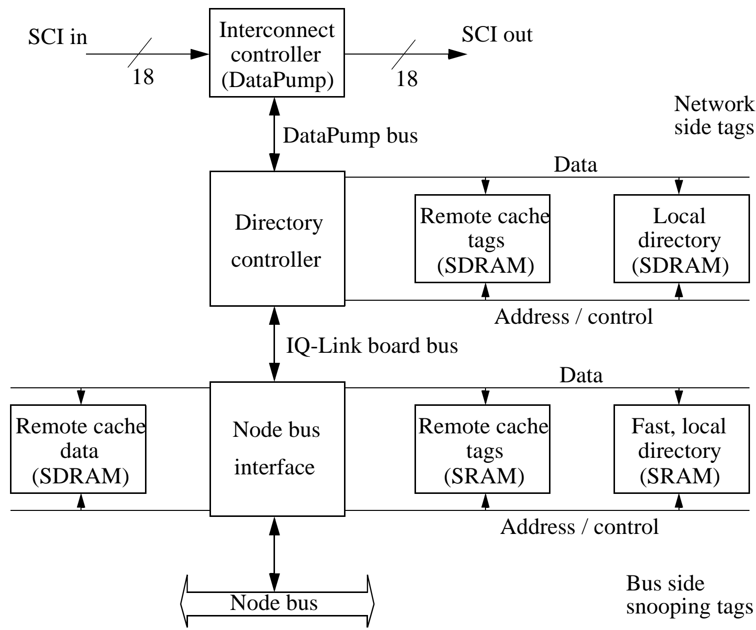

Fig. 1.10. Block diagram of Sequent's IQ-Link board

The block size of the remote cache is 64 bytes. The remote cache tags carry information as shown in principle in Figure 1.6. The implementation deviates from the standard, though. Forward and backward pointers are only 6 bits wide (allowing 64 nodes in a first-generation system); only an address tag of 13 bits is held, rather than a full 48-bit address offset. Notice that the remote cache tags are duplicated so that the two protocols may operate concurrently on the remote cache. Since only the SCI protocol is aware of other nodes, only the network-side copy of the coherence tags need to have pointers to other nodes; the bus-side tags only contain address and state information. The bus-side tags must be accessible at bus speed and are therefore implemented in SRAMs.

The *local directory* maintains information about the state of the blocks that have their home in the local memory; see the memory tags in Figure 1.6. Again, the implementation differs from the standard, providing a 2-bit memory state and a 6-bit list head pointer only. The local directory tags are duplicated as well and again need not contain pointers to other nodes on the bus side.

The *node bus interface* is an agent that snoops the node bus and manages the bus-side tags as well as the remote cache data arrays. When the bus interface controller detects that a bus request to a remote memory can be satisfied by the remote cache on the IQ-Link board, it supplies the requested block on the bus, with a latency close to that of a local memory access. Accesses that cannot be satisfied locally are forwarded to the directory controller. Further, the bus interface forwards incoming requests that access data, change state, or invalidate blocks, to the node bus on behalf of the directory controller. The bus interface can handle multiple outstanding requests and operate on multiple incoming requests.

The *directory controller* manages the network-side tags and executes the SCI coherence protocol. In first-generation systems, this component is implemented as a programmable protocol processor. The reason behind this decision is to avoid the risks and costs associated with a hardware implementation of the complex SCI cache coherence protocols; a firmware implementation is more easily debugged and optimized than hardware. Clearly, this does have performance impacts, as indicated below. The protocol processor can handle up to 12 protocol transactions and one interrupt transaction simultaneously by invoking a firmware "task" for each operation and supporting fast hardware task switch. Special logic for bit-field processing is available.

The *interface controller* is an SCI interface as depicted in Figure 1.5, implementing the physical and logical layer SCI protocols. The Vitesse DataPump chip is employed for NUMA-Q machines, driving 1 GByte/s, 18-bit wide SCI links.

Several problems arise due to two cache coherence protocols interacting in the machine [7]. An obvious problem is the different cache line size on the node bus (32 bytes) and on SCI (64 bytes). Some limitations of the Pentium

Pro bus (in first-generation systems) become difficult to work around when long-latency operations appear. For example, since the bus expects in-order responses to requests, a memory access that must be satisfied remotely must be signaled to the bus as delayed by the IQ-Link bus interface controller. When the reply comes back eventually, the bus interface must place it on the bus and complete the deferred transaction. A delayed reply does not automatically update main memory, requiring special actions by the IQ-Link interface in case a locally allocated block is concerned. Furthermore, when two read-exclusive requests appear on the bus back-to-back, the second one is aborted by the bus and must be retried later on, rather than being buffered to implement a reply more efficiently. Subtle issues also arise with serialization of concurrent remote and local accesses to a memory block. This is done in SCI at the home memory of the block. When the home is an SMP node, the bus protocol must be involved on the home node to update the states of local copies of the block on remote accesses. In fact, the bus becomes the serialization point rather than the IQ-Link agent.

Performance characteristics of the NUMA-Q machine are given in [7]. Initial systems can sustain 30 MByte/s data transfer rate in each direction through the IQ-Link board. Remote memory access latencies are about 3 μs minimum and up to 9 μs under heavy load; more than 60% of these remote latencies are typically spent traversing the IQ-Link boards. Optimized microcode and hardware assist in the directory controller were identified as promising techniques for reducing these latencies. Nevertheless, the NUMA-Q machine delivers good performance on the commercial workloads it is designed to handle [32].

1.4.3 I/O Subsystem Interconnect

SCI can be used to connect one or more I/O subsystems to a computing system in novel ways. The shared SCI address space can include the I/O nodes which then are enabled to directly transfer data between the peripheral devices (in most cases, disks) and the compute nodes' memories using DMA; software needs not be involved in the actual transfer. Remote peripheral devices in a cluster, for instance, thus can become accessible like local devices, resulting in an I/O model similar to SMPs; remote interrupt capability can also be provided via SCI. High bandwidth and low latency, in addition to the direct remote memory access capability, make SCI an interesting candidate for an I/O network.

There are currently two commercial implementations of SCI-based I/O system networks. One is the GigaRing channel from SGI/Cray [36], the other one the external I/O subsystem interconnect of the Siemens RM600 Enterprise Servers, based on Dolphin's cluster technology [38].

The *SGI/Cray GigaRing channel* is a high-speed I/O network using point-to-point links organized as two counter-rotating rings. It is used to interconnect peripheral controllers, network bridges, and SGI/Cray supercomputers

in a variety of configurations. For example, a number of I/O nodes attached to a single GigaRing can form the I/O subsystem of an SGI/Cray computing system. One or multiple GigaRing channels with peripheral devices can be shared by several, even heterogeneous SGI/Cray systems, e.g., a T90 vector machine and a T3E massively parallel processor. The GigaRing therefore represents a common I/O system interconnect for the various types of SGI/Cray computers.

The GigaRing channel heavily borrows from the SCI concepts and protocols, but deviates from the standard in many respects. Unused features have been discarded and new ones added to meet the requirements of a high-speed I/O channel.

The physical layer of the GigaRing is implemented by the proprietary GigaRing node chip. This chip is basically a simplified SCI node interface, but with 32-bit links in both directions and a 64-bit interface to the client (compute or I/O node). The chip and the wide links provide more than 1 GByte/s bandwidth per direction.

The logical layer uses SCI's ring access, bandwidth allocation, and queue reservation protocols with minor modifications. However, the packet types are tailored specifically to the requirements of I/O, providing transactions for small-message transfers, DMA operations, and maintenance and control. Packet formats differ significantly from the standard, with 32-bit symbols, 16-byte headers, up to 256 bytes of payload, and a 4-byte trailer. GigaRing also provides mechanisms for end-to-end congestion control, an area which is not covered by the SCI standard. Obviously, the cache coherence protocol is not implemented on GigaRing.

The two counter-rotating rings provide for increased availability: a broken ring can be disabled (ring masking), and the two rings can be appropriately cross-connected to isolate a broken node, resulting in one intact ring (ring folding). Disabling and reconfiguration can be effected by maintenance packets sent on the GigaRing channel.

Siemens provides an *external I/O expansion* for the RM600 Enterprise Servers, using Dolphin's PCI-SCI cluster components. SCI rings can be used to couple multiple PCI controllers such that they appear as one large PCI bus with more than 100 PCI slots. SCI can also be used to directly couple nodes in a cluster configuration.

1.4.4 Large-Scale Data Acquisition System

A special form of an I/O system is a data acquisition system. The most challenging data acquisition tasks exist in high-energy physics applications, e.g., particle detectors or nuclear fusion experiments. As an example, consider the ATLAS experiment at the Large Hadron Collider (LHC), now under construction at CERN. It is expected that merely the second-level data acquisition and real-time selection system will comprise about 2000 data sources and 1000 processing nodes. These will have to be connected by a high-performance

network that must be able to sustain a throughput of several GByte/s. See Chapter **??** for a detailed explanation of the structure and requirements of this data acquisition system.

It is not surprising, therefore, that researcher have investigated SCI as the basis for these interconnects for many years. CERN, together with associated institutions, have designed and demonstrated SCI components, network structures, and software in various projects. Early research is documented in [2], and a current project to build an SCI prototype system is described in Chapter **??**. Another contribution in this book, Chapter **??**, studies topologies and performance characteristics of SCI networks for plasma fusion devices, currently using simulation as the main tool.

1.5 Related Communication Networks and Concepts

Besides SCI, a number of related communication networks and concepts have been proposed for building clusters or tightly-coupled multiprocessors (NUMA or CC-NUMA systems, respectively). For the sake of brevity, only the most significant differences between the related interconnects and the SCI standard and SCI implementations are pointed out in the sequel.

Myricom's *Myrinet* [1] is a high-speed system area and local area network that has its origins in the interconnect technology of a massively parallel machine. Network interface cards attaching to workstations' I/O buses, high-speed links, switches, and a wealth of software, predominantly optimized message-passing libraries, are available to facilitate the construction of high-performance compute clusters. In contrast to SCI, a shared address space across the nodes in a cluster is not provided by the technology. However, the adapter card hosts a programmable processor, which allows specific communication mechanisms to be implemented, among them abstractions akin to DSM [11]. Chapter **??** describes Myrinet in more detail and investigates its performance by several communication benchmarks.

Other cluster interconnects supporting high-bandwidth, low-latency message-passing communication include *ParaStation* [43] and *ATOLL* [4]. *PAPERS* is an interconnect technology focussing on fast aggregate communication and computation, e.g., barrier synchronizations and global reduction operations [21]. The *Gigabyte System Network* (GSN) is an implementation of the ANSI standard developed under the name HIPPI-6400; it is currently being offered by SGI for highest-throughput (close to 800 MByte/s) cluster computing and as a storage area network [39].

The Compaq/Digital *Memory Channel* (MC) [14][15] network is conceptually similar to SCI in that it provides a hardware-based, non-coherent physical DSM in a cluster of workstations or SMPs; low communication latencies and overheads are of primary concern as well. However, the MC uses the reflective memory concept which mirrors write operations to a memory in other, connected memories. The nodes' memories can be connected to each

other by address mappings similar to SCI, with page-level granularity. The network adapters attach to the I/O bus and can be accessed from user level, as in SCI. Only writes to remote memories are facilitated, though, no read accesses. The most important difference to SCI is that the shared address space of MC is provided by a separate device, the MC Hub, which imposes a strict limit on the number of nodes that can be attached. MC provides in-order delivery of writes, flow control in hardware, and more sophisticated error-detection and reliability features than SCI.

The *SHRIMP* multicomputer [3] introduced the notion of Virtual Memory Mapped Communication (VMMC). The concept is similar to SCI DSM in that it allows applications to transfer data directly between two virtual memory address spaces over the network. The basis are virtual memory-mapped network interfaces and import-export mappings similar to SCI. Two transfer strategies are supported: deliberate update, which is an explicit transfer (send) of data, and automatic update, which reflects operations on exported local memory segments in the remote memory (by hardware means). A number of communication libraries have been implemented exploiting the advantages of VMMC, most importantly user-level access to the network and the opportunity to perform zero-copy data transfers [8]. In contrast to SCI, SHRIMP does not provide a cluster-wide shared address space. The first implementation was based on a proprietary Intel routing backplane as used in the Paragon multiprocessor; the second implementation adapts Myrinet to support the VMMC concept [11].

ServerNet, developed by Compaq/Tandem [22][23], is a flexible system area network that can support communication among processors (by memory-to-memory data transfers), I/O traffic between processors and peripheral devices, and even between I/O controllers. The ServerNet interconnect has interfaces to processors and associated memories, and to peripheral buses, with the goal to unify inter-processor and I/O connectivity. Processors and memories can be connected to two redundant, multi-stage interconnect fabrics. Similar to SCI, ServerNet supports both read and write transactions directed towards remote memories or I/O nodes in order to pull or push data across the network without software intervention at the remote node; remote interrupts are supported as well. The network hardware provides delivery and ordering guarantees as well as fault detection and isolation features at various architectural levels. ServerNet is a proprietary implementation, originally designed to build high-throughput, reliable I/O systems [23] and fault-tolerant clusters. More recently, however, ServerNet-II was proposed as a native implementation of the Virtual Interface (VI) Architecture standard [12], targeted also towards high-performance compute clusters for scientific applications [19].

The *Virtual Interface (VI) Architecture* industry standard [12] summarizes many of the concepts and approaches developed in earlier projects, like U-Net [42] or SHRIMP [3], for high-throughput and low-latency communi-

cation in clusters. VI Architecture specifies cluster communication concepts and a network interface architecture in an open, implementation-independent fashion. Notable concepts are: virtualization of the network interface to user processes; organization of send and receive queues and descriptors; protected, user-level access to the network; registration and use of portions of the virtual memory for communication; virtual-to-physical address translation for these memory regions, bypassing the operating system; avoidance of intermediate data copies as well as of interrupts and context switches, in order to minimize communication latency and CPU overheads; and remote memory accesses without involving software on the remote node. It is expected that these concepts will have a significant influence on future products designed for fast cluster communication.

As to scalable cache-coherent multiprocessors, the SGI *Origin 2000* [31] seems to be the only commercial alternative to the SCI-based CC-NUMA machines mentioned in Section 1.4.2. The Origin 2000 is an advanced, optimized server machine with origins in the Stanford DASH project. The nodes in the Origin are equipped with two MIPS processor which are connected to each other, to local memory, and to I/O and network interfaces by a single hub chip. Up to 512 nodes can be interconnected in a hypercube-like topology. Both the memory in the system and the I/O address space are globally addressable. Cache coherence is maintained by a bit-vector directory scheme where up to 64 bits per memory block keep track of where copies of the block exist. The bit vector can be adapted to identify a single node per bit (full bit vector) or, for systems larger than 128 processors, groups of processors (coarse bit vector). The directory memory of a node is associated with its local memory and grows proportionally as the latter is extended. The hardware and the operating system have mechanisms to count the references to memory pages and migrate or replicate the page to requestors, if this promises to reduce the average memory access latency.

S3.mp was an experimental research project by Sun [34] that aimed at connecting off-the-shelf workstations (SparcStations) into a large-scale, cache-coherent multiprocessor. The project developed a memory controller that attaches to the Mbus of a SparcStation, handles accesses to local and remote memories, and executes the cache coherence protocol by two microcoded protocol engines. S3.mp uses an invalidation-based protocol which keeps track of nodes sharing a memory block using singly-linked lists. The second major contribution of the project was the interconnection network, comprising an interconnect controller with an integrated arbitrary topology router and high-speed serial fiber-optic links. Unlike the SCI-based CC-NUMA multiprocessors which use special packaging, S3.mp nodes were anticipated to be spatially distributed (up to 200 m) and to be connected in arbitrary topologies.

Finally, the *Wisconsin Wind Tunnel Project* [20] pursues in-depth research on the design trade-offs for cost-effective DSM parallel machines, taking fur-

ther many of the issues that SCI addresses. Research areas include network interface design, cooperation of hardware and software to manage the DSM, combining shared memory and message passing for parallel computation, and the use of system-wide prediction and speculation techniques, e.g., to accelerate coherence actions.

1.6 Concluding Remarks

SCI is an interconnect standard that specifies leading-edge high-speed networking and distributed shared memory (DSM) technology. Although the concepts and protocols are well devised and were published as an open specification in the early 1990s already, SCI has not lived up to its promise of becoming an "open distributed bus". Chapter**??** presents an insider's view on the obstacles to rapid and wide-spread acceptance by industry.

SCI was, however, rather quietly adopted by several companies that recognized its superior concepts and protocols as well as its potential of high performance, but had no interest in implementing and providing SCI as an open interconnect. A number of proprietary implementations and products therefore appeared over the years, ranging from high-performance cluster interconnects, to shared-memory multiprocessor networks with cache coherence implemented in hardware, and high-speed I/O subsystem interconnects. In particular, the CC-NUMA machines based on SCI technology from HP/Convex, Sequent, and Data General turned out to be quite successful.

Adoption of workstation clusters using an SCI interconnect (and its DSM), is however slower than expected, despite the superior performance characteristics of SCI cluster networks available today. The reasons may well be that for many years there has been only one serious vendor of SCI adapters and switches, Dolphin Interconnect Solutions, and that progress on developing software for the efficient use of these clusters has been slow. This even holds for message-passing libraries and more so for shared-memory programming paradigms that may take advantage of SCI's DSM. This book is intended to contribute to faster and wider acceptance of SCI clusters in academia and industry by summarizing the state of the art of SCI cluster computing, pointing out achievements and remaining obstacles as an aid for potential users.

SCI's development and influence is not finished yet. For example, recent interconnect standard projects like SyncLink, a high-speed memory interface, and Serial Express (now tentatively named SerialPlus), an extension to the SerialBus (IEEE 1394) interconnect, directly emerged from SCI or are heavily influenced by SCI concepts. SCI also plays a role in the debate on future I/O systems (NGIO versus FutureIO). These current developments and future directions are explored in more detail in Chapter**??**.

References

1. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, Feb. 1995.
2. A. Bogaerts et al. *RD 24 Status Report: Application of the Scalable Coherent Interface to Data Acquisition at LHC.* Oct. 1996. http://nicewww.cern.ch/~hmuller/~HMULLER/docs/report96.pdf.
3. M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, pages 21–28, Feb. 1995.
4. U. Bruening, L. Schaelicke. ATOLL: A High-Performance Communication Device for Parallel Systems. *Proc. Advances in Parallel and Distributed Computing.* Shanghai, 1997.
5. R. Clark. *SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium II Xeon SHV Nodes.* White Paper. Data General Corp. 1999.
   http://www.dg.com/about/html/sci_interconnect_chipset_and_a.html.
6. Convex Computer Corp. *Convex Exemplar Architecture.* Technical Document DHW-014. Convex Computer Corp., Nov. 1994.
7. D. E. Culler, J P. Singh, with A. Gupta. *Parallel Computer Architecture: A Hardware-Software Approach.* Morgan Kaufmann 1998.
8. S. N. Damianakis, A. Bilas, C. Dubnicki, E. W. Felten. Client-Server Computing on SHRIMP. *IEEE Micro*, pages 8–18, Jan./Feb. 1997.
9. Dolphin Interconnect Solutions. *Link Controller LC-2 Specification.* Data Sheet, Dolphin 1997.
10. Dolphin Interconnect Solutions and Sun Microsystems. *Sun Enterprise Cluster Architecture.* Application Note, Dolphin 1998.
    http://www.dolphinics.com/dolphin2/interconnect/applications/Sun_Cluster_Arc.htm.
11. C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, K. Li. SHRIMP Project Update: Myrinet Communication. *IEEE Micro*, pages 50–51, Jan./Feb. 1998.
12. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.
13. D. R. Engebretsen, D. M. Kuchta, R. C. Booth, J. D. Crow, W. G, Nation. Parallel Fiber-Optic SCI Links. *IEEE Micro*, pages 20–26, Feb. 1996.
14. R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12–18, Feb. 1996.
15. R. B. Gillett, R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, pages 19–25, Jan./Feb. 1997.
16. D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–22, Feb. 1992.
17. D. B. Gustavson. The Many Dimensions of Scalability. *Proc. COMPCON Spring'94*, 1994.

18. D. B. Gustavson, Q. Li. The Scalable Coherent Interface (SCI). *IEEE Communications Magazine*, pages 52–63, Aug. 1996.
19. A. Heirich, D. Garcia, M. Knowles, R. Horst. *ServerNet-II: a Reliable Interconnect for Scalable High Performance Cluster Computing.* Technical Report. Compaq Computer Corporation, Tandem Division. Sept. 1998. http://www.servernet.com/flat/public/brfs_wps/snetii/snetii.pdf.
20. M. D. Hill, J. R. Larus, D. A. Wood. *The Wisconsin Wind Tunnel Project: An Annotated Bibliography.* Technical Report. Computer Sciences Dept., Univ. of Wisconsin-Madison. Aug. 1999. http://www.cs.wisc.edu/~wwt.
21. R. R. Hoare, H. G. Dietz. A Case for Aggregate Networks. *Proc. IPPS/SPDP'98.* IEEE CS Press 1998.
22. R. W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, pages 1–9, Feb. 1995.
23. R. W. Horst, D. Garcia. ServerNet SAN I/O Architecture. *Proc. Hot Interconnects V*, Aug. 1997.
24. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill 1993.
25. IEEE Std 1212-1991. *IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses.* The Institute of Electrical and Electronics Engineers, Inc., 1991.
26. IEEE Std 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1993.
27. IEEE Std 1596.3-1996. *IEEE Standard for Low Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1996.
28. IEEE Std 1596.5-1993. *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors.* The Institute of Electrical and Electronics Engineers, Inc., 1993.
29. IEEE Std 1596.7-199X Draft 0.99. *Draft Standard for A High-Speed Memory Interface (SyncLink).* 1999. http://www.SLDRAM.com/FAQ/SyncLinkD0.99.pdf.
30. D. V. James, D. B. Gustavson, B. Fleischer. Serial Express: A High-Performance Workstation Interconnect. *IEEE Micro*, pages 54–65, May–June 1998.
31. J. Laudon, D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. 24th Int'l. Symp. on Computer Architecture.* ACM Press 1997.
32. T. Lovett, R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *Proc. 23rd Int'l. Symp. on Computer Architecture.* ACM Press 1996.
33. T. D. Lovett, R. M. Clapp, R. J. Safranek. *NUMA-Q: An SCI-based Enterprise Server.* White Paper. Sequent Computer Systems, Inc., 1996. http://www.sequent.com/products/highend_srv/numa_sci.pdf.
34. A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. Int'l. Conf. on Parallel Processing.* 1995.
35. *SCIzzL: the Scalable Coherent Interface and Serial Express Users, Developers, and Manufacturers Association.* http://www.SCIzzL.com.
36. S. Scott. The GigaRing Channel. *IEEE Micro*, pages 27–34, Feb. 1996.
37. Siemens AG. *High Performance Computing – HPCLINE.* http://www.siemens.de/computer/hpc/en/hpcline/index.htm.
38. Siemens AG. *RM600 E. Model E30, E70.* Data Sheet. Siemens, Sept. 1998. http://manuals.mchp.siemens.de/servers/rm/rm_us/rm_pdf/rm600e37.pdf

39. Silicon Graphics Computer Systems. *Gigabyte System Network.* Data Sheet. SGI, Nov. 1998. http://www.sgi.com/Products/PDF/2287.pdf.
40. R. Thekkath, A. P. Singh, J. P. Singh, S. John, and J. L. Hennessy. An Application-driven Evaluation of the Convex SPP-1200. *Proc. 11th Int'l. Parallel Processing Symposium.* IEEE Computer Society Press 1997.
41. Vitesse Semiconductor Corp. *Compliant Link Controller 1 GByte/sec SCI VSC7201a.* Data Sheet, Vitesse 1996.
42. T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proc. 15th ACM Symposium on Operating System Principles.* ACM Press 1995.
43. T. M. Warschko, J. M. Blum, W. F. Tichy. ParaStation: Efficient Parallel Computing by Clustering Workstations: Design and Evaluation. *Journal of Systems Architecture.* Vol. 44 (3–4) (Special Issue on Cluster Computing), pages 241–260, Dec. 1997.