

SCI SOCKET - A Fast Socket Implementation over SCI

Version 2

Friedrich Seifert
sfr@foobar-cpa.de
foobar GmbH
Bernsdorfer Str. 210-212
D-09126 Chemnitz, Germany

Hugo Kohmann
hugo@dolphinics.no
Dolphin Interconnect Solutions AS
Olaf Helsets vei 6
NO-0619 Oslo, Norway

Abstract

In this paper we introduce a very low latency implementation of the Berkeley Sockets interface on top of the Dolphin SCI interconnect. Average one byte latencies as low as 2.31 microseconds are achieved. To our knowledge, this is the fastest sockets implementation in terms of latency. SCI SOCKET combines outstanding performance and seamless integration with legacy applications. This enables networking applications to transparently exploit the capabilities of SCI without being modified or even recompiled. We describe major design decisions regarding transparent integration with legacy sockets and data transfer protocols.

Keywords— **High performance networking, BSD Sockets, Low latency, SCI, User level communication.**

1 Introduction

With link level speed of modern interconnect hardware being increased, the communication bottleneck is moved more and more to protocol software. For this reason all high speed networks provide their own efficient interfaces and protocols. The low level user space interface for SCI is called SISI [1] which provides basic mechanisms to share memory segments between nodes and to transfer data between them. There is also a kernel space interface called GENIF.

On the other hand there is a large number of applications that make use of legacy protocols such as the TCP/IP protocol suite to maintain the highest possible degree of portability. The de-facto standard interface is the Berkeley Sockets API [2]. Rewriting a mature application to use a hardware specific API may be unprofitable in many cases. Also, the same effort is necessary for every new network that is to be supported.

SCI SOCKET aims to bring together legacy applications and the low latency SCI communication hardware by providing a sockets API abstraction layer on top of the low level SISI and GENIF interfaces. Special emphasis has been put on transparent integration with existing software as well as on achieving highest possible performance.

In this paper we discuss the principal design alternatives and present the decisions made for the implementation of SCI SOCKET.

Section 2 presents related achievements that have motivated and influenced this work. After that we present a brief introduction to the Dolphin interconnect technology. After discussing general design questions in section 4 we will look at implementation details in section 5 and present performance numbers in section 6. A short summary and outlook to future enhancements round off the paper.

2 Related Work

SCI SOCKET builds upon results of a number of research projects in this area. The first of which was FastSockets [3], a stream socket implementation on top of ActiveMessages [4], a light weight communication protocol for Myrinet [5]. Later, a stream socket interface for the *SHRIMP* multicomputer was introduced [6]. SOVIA [7] is user-level socket layer on top of the Virtual Interface Architecture [8]. A similar approach has been shown for Gigabit Ethernet in [9]. GMSOCKS [10] describes how standard Windows sockets can be mapped onto Myrinet while the TCP/IP stack is replaced by a special buffer management.

Another recent initiative worth mentioning is the Offload Sockets Framework (OSF) [11] for Linux which enables communication over system area networks (SAN) while bypassing the kernel internal TCP/IP protocol stack. An example offload protocol is the Socket Direct Protocol (SDP) for Infiniband [12].

3 Dolphin Interconnect Technology

Dolphin's interconnect adapter provides a transparent, reliable high bandwidth and very low latency connection between PCI buses based on the ANSI/IEEE 1592-1992 Scalable Coherent Interface (SCI) standard [13].

The PCI-SCI adapter is designed to meet the requirements for high availability clustering and remote I/O applications. The programmed, remote memory access (RMA) feature of the PCI-SCI bridge enables ultra-low latency

messaging and low overhead and transparent I/O transfers. PCI bus memory transactions are converted into corresponding SCI transactions allowing physically separate PCI buses to appear as one. This feature allows applications to send data between system memories without the use of operating system services, greatly reducing latency and overhead. A full remote memory write made up of a request/response pair typically takes 1.4 microseconds. Pipelined 4 byte write posts account for only 0.21 microseconds each.

By use of the DMA controller, blocks of memory can be copied directly between PCI buses in a single copy operation with no need for intermediate buffering in adapter cards or buffer memories. This feature greatly reduces latency and lowers overhead of data transfers. The DMA controller supports both read and write operations and is fully interleaved with RMA operations.

The PCI-SCI bridge has built-in address translation, error detection and protection mechanisms to support highly reliable connections.

4 General Design Questions

4.1 Implementation Level

The most important design question for SCI SOCKET was where to implement it: at kernel level or at user level. Both approaches have advantages and drawbacks which we discuss in the following:

Kernel level: The simplest way is to provide an IP-to-SAN layer that can be used seamlessly in conjunction with all kernel resident protocols such as TCP or UDP. While this is the least complex method it allows for little optimization only, since all higher level protocols are still part of the communication path. Also, the underlying hardware might already provide reliable transfers.

In contrast to this a sockets-to-SAN layer inside the kernel can exploit the hardware capabilities much better to build an efficient but compliant sockets interface. Advantages are that applications can use the fast transport transparently, and all kernel-provided functionality such as `fork()` and `select()` are supported natively. This approach is for example used by the Offload Socket Framework.

The major drawback is that all communication goes through the kernel and hence involves a user-kernel-transition (system call). The times needed for system calls on Linux have been measured to be below one microsecond [14, Section 8.3.3]. We considered this delay relevant in relation to the extremely low hardware latencies of SCI when we started the SCI SOCKET development. However, measurements on recent machines have yielded kernel call times to be as low as 0.18 μsec in the best case.

User level: an implementation at user level enables direct

application to application transfers without kernel intervention. The disadvantage, however, is that some functionality normally provided by the kernel must be reimplemented or simulated at user level. An example is the socket port space. Advanced features such as `fork()` require special handling, and transparent integration with existing applications is more difficult. With this approach it is not possible to create accelerated network file systems residing in the kernel, like it is with the kernel level implementation.

Conclusion For the sake of achieving the highest performance we decided to implement SCI SOCKET at user level initially. Actually, we apply a hybrid method as used by FastSockets and *SHRIMP* Stream Sockets, but all communication takes place at user level. A more detailed description will be given in section 5.2.2.

The need to support kernel space consumers such as iSCSI and network file systems led to the development of a kernel implementation of SCI SOCKET. Surprisingly, it performs nearly as well as the user level version, on some platforms even better.

An IP driver for SCI has also been developed [15].

4.2 Transparent integration

For both implementations the question arose how to enable existing applications to use such sockets. Since relinking or even recompiling may not be an option for a number of applications, completely transparent integration has been an objective from the beginning. In the following we describe how this has been realized for UNIX-like operating systems.

In order to divert socket communication, without touching the application, the sockets API functions must be intercepted. Although it is possible to modify the C library (at least on Linux) this would be a non-portable and fragile approach, since it would have to be kept in sync with C library development.

The solution is dynamic linking. Usually all applications are linked dynamically to the C library, where the sockets interface is defined. By using the preload mechanisms we can force the dynamic linker/loader to load a special SCI SOCKET library before all the other libraries. That library contains wrapper functions that override the sockets API functions of the C library. Whenever the application invokes one of those calls the SCI SOCKET library decides if it will handle the call by itself, pass it on to the C library, or both.

The SCI Kernel Sockets differ from regular sockets only in the address family. For SCI User Sockets the complete functionality must be provided by a library at user level. Therefore the preload library for SCI Kernel Sockets only intercepts the `socket` call to replace `AF_INET` by `AF_SCI`.

In both cases the preload mechanism can be activated by specifying the SCI SOCKET library in one of two places:

System wide in the file `/etc/ld.so.preload`

Selective in the environment variable `LD_PRELOAD`

In the cases where the application can be relinked, the wrapper functions can be skipped by linking it statically to the `SCI SOCKET` library.

4.3 Configuration

`SCI SOCKET` needs to be configured for the cluster it runs on, in order to properly map IP addresses to SCI hardware addresses, called **nodeids**. The mapping is defined by a simple configuration file containing a host name or IP address and the associated SCI nodeid per line.

This is a simple example configuration file:

```
#host          #nodeid
192.168.10.1   8
192.168.10.2   12
```

Especially if the system wide preload mechanism is used, it might be desirable to exclude specific applications from using `SCI SOCKET`. This can be done by specifying allowed and forbidden port numbers in a separate configuration file. For instance, telnet will hardly benefit from microsecond latencies, even for users typing really quickly.

Configuration of SCI Kernel Sockets is done through a utility program that reads the configuration files and passes the information to the `AF_SCI` driver. This must be once after loading the driver. The user space `SCI SOCKET` library evaluates the file at application startup.

5 Implementation Details

This section gives an insight into the implementation of the SCI User Sockets and SCI Kernel Sockets. Currently, the user space version only supports stream sockets.

5.1 SCI Kernel Sockets

SCI Kernel Sockets have been implemented as a new address family. Currently, Linux is able to handle 32 address families where numbers from 27 to 30 are unused. We have chosen 27 for the new `AF_SCI` address family. Since `AF_SCI` is compatible to `AF_INET`, only the address family parameter of the `socket` has to be changed to allow an application to use the SCI network.

In the following sections we present the data structures used by SCI Kernel Sockets and how they plug into the Linux networking subsystem, and describe in brief how the socket function calls have been implemented.

`SCI SOCKET` provides two major types of kernel sockets, explicit and transparent. Explicit SCI sockets live in their own port space and are unrelated to system TCP or UDP sockets. However, in this paper we concentrate on transparent sockets since they are designed to allow seamless integration with existing applications.

5.1.1 Data structures

Linux uses a number of data structures, some of which are protocol dependent, to implement the socket abstraction. Every socket is represented by a `struct socket`. It is created by the generic socket driver and contains among others a reference to the associated inode, flags, state and socket type. `proto_ops` is an essential member that points to a set of protocol specific functions, e.g. `bind`, `connect` etc. The member `sock` is a pointer to a `struct sock`. `protinfo` is a union comprising various protocol specific information. SCI Kernel Sockets use the generic union member `destruct_hook` to store a reference to a SCI specific `sci_socket_t` structure.

As one objective of `AF_SCI` is to act completely transparently with respect to `AF_INET`, every transparent `AF_SCI` socket has an `AF_INET` socket associated with it, called *native socket*.

For `AF_SCI` sockets of type `SOCK_STREAM` `sci_socket_t` points to a single `sci_conn_t` structure, which holds all connection related information, e.g. the message queue handles. The relations of all the data structures mentioned so far are illustrated for stream sockets in figure 1.

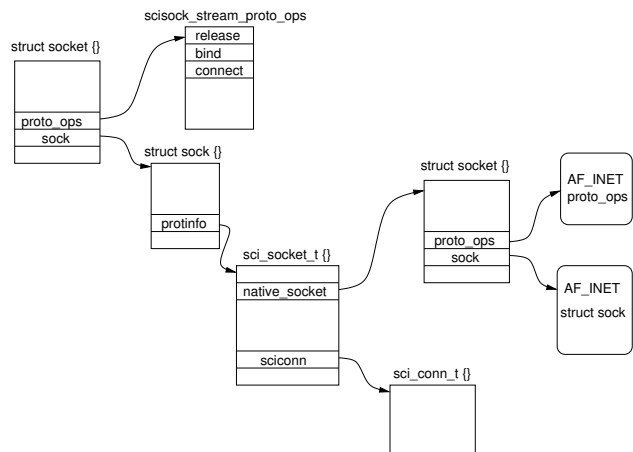


Figure 1: Stream socket data structures

Datagram sockets are connectionless by definition. However, SCI communication requires that a connection is set up between the nodes. That is why datagram type `AF_SCI` sockets use implicit connections. Since a single datagram socket can communicate with multiple destinations, a list of implicit connections must be maintained. The data structures for a datagram socket are shown in figure 2.

5.1.2 Socket Create

The socket creation function `scisock_create` belongs to the `AF_SCI` family. Whenever a socket of this family is created, the kernel calls that function passing a pointer to a `struct socket`. The function creates a native `AF_INET` socket, allocates a `sci_socket_t` and

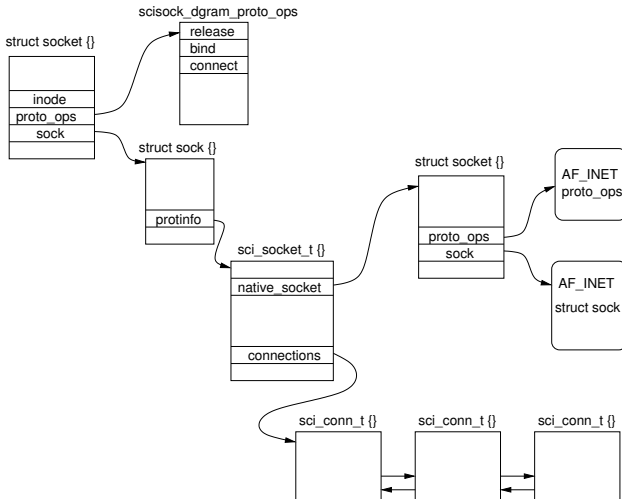


Figure 2: Datagram socket data structures

`struct sock` structure and links them together as described in section 5.1.1. Finally, a set of protocol specific operations is installed for the `struct socket`. The kernel will forward all further operations on the socket, e.g. `bind` to one of those functions.

5.1.3 Socket Bind

Transparent `AF_SCI` sockets share the port space with all system sockets. Therefore the `bind` operation is first performed on the native sockets. The address assigned by the system is then retrieved and used for `SCI` specific addressing.

5.1.4 Socket Listen

In transparent mode `AF_SCI` stream sockets can be used to communicate with other `AF_SCI` sockets as well as with regular `AF_INET` sockets. The `listen` function ensures that both kind of connections can be established by calling `listen` on the native socket and setting up the mechanism to receive `SCI` connection requests. A dedicated kernel thread is responsible for handling incoming `SCI` connection requests in the background.

5.1.5 Socket Connect

Both stream and datagram sockets can be “connected”. However, the semantics are quite different. While an actual connection is established for stream sockets, only a remote address is recorded for datagram sockets, so that a remote address does not have to be specified for subsequent `send` and `receive` operations.

The behaviour of the `connect` operation on an `AF_SCI` socket depends on the configuration of `SCI SOCKET`. If the remote node is listed in the node map file and if the destination port is enabled an `SCI` connection is attempted. In order to allow a non-blocking `connect` a

`sci_conn_t` structure is allocated, initialized with the remote address and passed to another dedicated kernel thread that is responsible for establishing the connection. Once the connection is set up, `sci_conn_t` is linked to the `sci_socket_t` structure.

If the remote address is not configured for `SCI` or if the `SCI` connection fails for some reason, the `AF_SCI` driver falls back to a native connection by calling `connect` on the native socket. When this happens, the `SCI` related resources are not needed anymore, so they are all freed. In this case the original `struct socket` and the native socket are merged. That means that all relevant information of the native socket is transferred to the original socket, including the protocol operations vector. From that time on the original socket appears as a regular `AF_INET` socket.

5.1.6 Socket Accept

As mentioned in section 5.1.4 a transparent `AF_SCI` socket is able to accept both `SCI` and `AF_INET` connections. In order to achieve this the `accept` function makes use of the poll mechanism of the native socket to wait for native connections and `SCI` connections at the same time. If a native connection is available, `accept` is called on the native socket. If there is an `SCI` connection in the form of a `sci_conn_t` structure a new `struct sock` and a `sci_socket_t` are allocated and linked with the `struct socket` that was passed to the `accept` method by the kernel.

5.1.7 Implicit Connections for Datagram Sockets

Datagram support was not completed at the time of writing, but we present the basic ideas.

Since datagram sockets are connectionless, necessary `SCI` connections must be set up implicitly. When `sendmsg` is called the driver checks if there is already a connection to the destination. If not, a connection request is passed to the connector kernel thread, similarly to the stream socket `connect` function. At the receiver `recvmsg` also checks if there is already a matching connection established. If not, the mechanism to receive `SCI` connection requests is set up. The further handling resembles the `accept` function in that `recvmsg` must wait for incoming `SCI` connections and for data on the native socket at the same time. Again, the poll mechanism will be used.

Implicit connections have a limited life time. If a connection has not been used for a certain amount of time, it is disconnected automatically to save resources.

5.2 SCI User Sockets

5.2.1 Supporting `fork()`

While `FastSockets` and `SHRIMP` Stream Sockets do not support the `fork()` system call, and `SOVIA` presents only a partial solution requiring modification of the application, `SCI SOCKET` aims at providing fully transparent `fork()` support. The difficulty is that we have to emulate some functionality normally provided by the kernel, at user level.

Normally, every open file is represented by a `file` structure within the kernel. A socket is treated as a special file that is linked to a unique `socket` structure. There is exactly one `file` object per open file in the system. The association of files to processes is done using a per-process array of pointers to `file` structures which are indexed by the descriptor. We refer to this array as `fd_array` in the following description. Every time an application passes a descriptor to a system call, the kernel looks up `fd_array` to find the corresponding `file` structure. Upon `fork()` the parent process's `fd_array` is duplicated for the child process. Hence, the `fd_array`'s of parent and child point to the same `file` structures, so they share all files and sockets.

In order to share SCI SOCKET connections between related processes, the kernel data structures need to be imitated at user level. This is achieved by a combination of process-private and process-shared data structures as illustrated in figure 3.

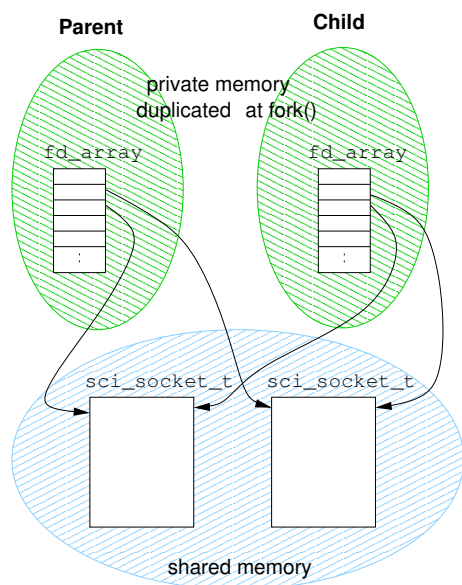


Figure 3: Sharing SCI sockets

Access to those shared structures must be synchronized by process-shared locks. Since such primitives are not provided by LinuxThreads, a combination of spinlock and pipe is used. Upcoming Pthread implementations for Linux promise to support process-shared mutexes natively [16].

The `sci_socket_t` structure represents the combination of the kernel's `file` and `socket` structures and are allocated in shared memory area. After `fork()` both processes refer to the same physical memory. In contrast to this, `fd_array` is allocated in private memory, so that it is copied during `fork()`. Initially both processes own the same sockets. However, they can alter their `fd_array` independently.

5.2.2 Connection Management

As mentioned above SCI SOCKET uses a hybrid user/kernel level approach in that SCI SOCKET connections are built upon regular socket connections. This enables SCI SOCKET among other things to share port namespace with the system. However, efficient usage of system wide preloading of the SCI SOCKET library requires that performance impact on applications be kept as small as possible. In particular, applications that do not need accelerated sockets should not suffer performance drops. SCI SOCKET takes this into account by not intercepting sockets API calls until a connection attempt is made. That means that there is no difference between a regular socket and an SCI socket, so far. `socket()`, `bind()` and `listen()` calls are forwarded to the C library immediately, only a single function is added to the call path. Binding the system socket ensures that there cannot be a regular socket and an SCI socket having the same port number at any time.

Shadow Connection Setup According to the Sockets API the server side next invokes `accept()`. SCI SOCKET intercepts this call as described above, and first of all calls the original C library `accept()` function to establish a connection. So, basic connection setup including address matching is left to the system and results in a new socket descriptor, referred to as shadow socket.

In the same way the client's `connect()` call is intercepted and the C library is called to establish the shadow connection.

SCI Specific Setup Once the shadow connection has been established, still within the SCI SOCKET library in `accept()` or `connect()`, respectively, the server and the client check if the socket family is `AF_INET` and if the type is `SOCK_STREAM`. Furthermore they examine their respective peer addresses to determine if they are contained in the configuration file and if the port is enabled. If not, the calls return to the application and the connection is treated as a regular one. Otherwise, the corresponding SCI nodeids are read from the configuration file. If an SCI connection is made for the first time, the internal SCI SOCKET structures and threads are initialized. This is what happens: A so called watcher thread is responsible for detecting remote connection shutdown by `select()`-ing on all shadow connections. Further, an array of `sci_socket_t` structures, called socket table, is allocated in (locally) shared memory.

After the (one time) initialization is done, an entry is allocated from the socket table at both the server and the client and added to the `fd_array` using the shadow socket descriptor as index.

Now, the SCI specific communication channels are set up. Two antiparallel, unidirectional message queues are created between the server and the client, and a remote notification mechanism is initiated. The message queues and applied transfer protocols are covered in section 5.3.

Finally, the sockets on both sides are marked as SCI-enabled socket and control is returned to the application.

5.3 SCILib - Efficient Message Queues

The SISCI API provides all the functionality needed to communicate over the SCI network. It is a stable and robust interface, but quite a low-level interface. Understanding that there is a need for an interface that is less low-level Dolphin started developing SCILib, a library on top of SISCI that offers unidirectional message queues.

SCILib has been chosen as the base of the SCI SOCKET implementation. The original version of SCILib has been extended to meet the requirements of high performance socket communication. This section provides an insight into the design and implementation of SCILib.

SCILib has also been ported to kernel space using the GENIF interface to support SCI Kernel Sockets.

5.3.1 Message Queue Realization

An SCILib message queue provides an unidirectional, non-blocking stream-oriented communication channel using SCI-shared memory segments. For both ends of the queue a local SCI segment is created, which the remote process connects to and maps into its address space. Figure 4 illustrates the layout of such a segment.

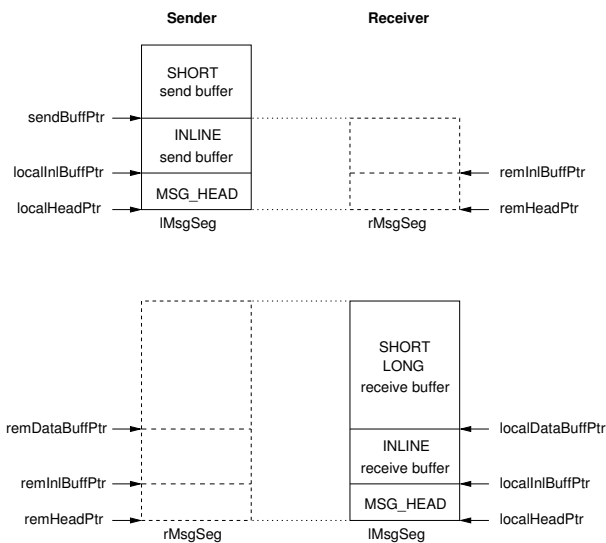


Figure 4: SCI segment layout of a message queue

MSG_HEAD is structured as shown in figure 5. Most entries are aligned to be just below a 128 byte boundary, in order to enable implicit stream buffer¹ flushing, when an entry is written to. The most important elements are tx and rx, the transmit (tail) and receive (head) pointers of the ring-like SHORT/LONG receive buffer. The sender and the receiver each have a copy of MSG_HEAD that are synchronized at certain points in time. Since tx is modified only by the sender, and rx only by the receiver, no

¹Stream buffers are used by the PCI-SCI bridge to perform write-combining before an SCI packet is sent over the link. Stream buffer and SCI packet payload size is 128 byte.

locking is required. For performance reasons the remote instance of MSG_HEAD is never read.

INLINE send and receive buffers are used for a separate very low latency protocol, while the SHORT send buffer is for gathering small messages into larger chunks.

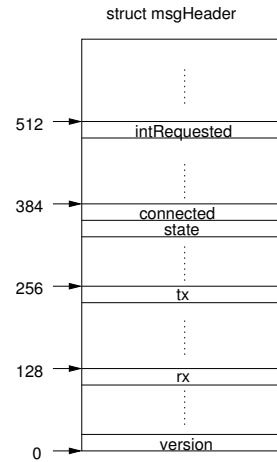


Figure 5: Structure of MSG_HEAD

SCILib offers three different protocols to transfer data that are described in the following sections.

5.3.2 INLINE Protocol

The INLINE protocol is designed for very low latency transfers of small messages. The term “INLINE” is derived from the fact that the synchronization information is sent along with the data in a single SCI transaction. Figure 6 shows the format of an INLINE message. Its total size is 128 byte, corresponding to the stream buffer size of the current SCI hardware. Byte 127 is used for synchronization. Since the SCI adapter always writes from low to high addresses the synchronization element is written only after data has landed in memory. Hence, the receiver can poll at seq to wait for a message.

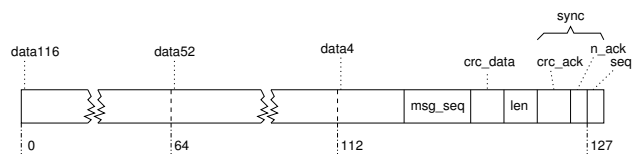


Figure 6: Structure of an INLINE message

The INLINE messages are kept in ring buffers of equal size at both the sender and the receiver. The send buffers are used to assemble the messages, which are then transferred to the corresponding ring buffer element at the receiver. seq is also used to manage buffer allocation at the sender. A buffer is available if its seq equals a special value SEQ8_FREE. Every time a new message is sent the sequence number in seq is incremented and eventu-

ally wrapped around. This number is necessary to detect duplicates caused by retransmissions.

Error checking is optimized for low latency operations by including a CRC in the message. CRC calculation and sending the message to the PCI-SCI bridge is a low cost operation since after having been touched by the application the data resides in the CPU cache. Actual sending is performed by a CPU posted store operation to the local I/O system. That means that the CPU does not have to wait for any kind of acknowledge and is immediately free to do other work. Reliable transfer is ensured by means of CRC and retransmission. `crc_data` is calculated over the data portion, `msg_seq`, `len` and `seq`. `crc_ack` is calculated only over `seq` and `n_ack`. Every time a buffer is transferred, a timestamp is recorded. If a subsequent send operation does not find an empty send buffer it looks at all buffers and retransmits them if their resend timeout has expired.

Once the receiver has seen the next expected sequence number in the ring buffer it checks the CRC in `crc_data`. If it is okay the `seq` field of the corresponding send buffer is set to `SEQ8_FREE` along with `crc_ack` via a CPU posted store operation. Since no immediate error checking is done, for performance reasons, this update may fail completely or data may have been lost without notification of the sender. In this case the sender will not see the buffer being freed again and will retransmit it after the timeout expires. The receiver detects this duplicate by means of the sequence number. All messages with numbers older than the next expected number are considered duplicates. Duplicates are just discarded by setting the sender's `seq` field to `SEQ8_FREE` once more. The "age" of sequence numbers is determined using the following expression:

$$a < b \Leftrightarrow ((char)((a) - (b)) < 0) \quad (1)$$

where a and b are of type unsigned char, and $<$ means "is older than".

`data116`, `data52` and `data4` are used to optimize the copy size by aligning data to the end of the stream buffer. Table 1 lists the possible message sizes and the corresponding offsets and copy sizes.

Message size	Offset	Member	Copy size
1 - 4	112	data4	16
5 - 52	64	data52	64
53 - 116	0	data116	128

Table 1: INLINE Message sizes and offsets

5.3.3 SHORT Protocol

The SHORT protocol has been designed for medium sized messages and is used for gathering small chunks of data to be sent as one large message to improve the streaming throughput. SHORT messages start with a message descriptor containing type (SHORT), length of data and the

message sequence number. The latter is needed to ensure ordering between SHORT or LONG and INLINE messages. Data starts immediately after the descriptor. Figure 7 illustrates the layout.

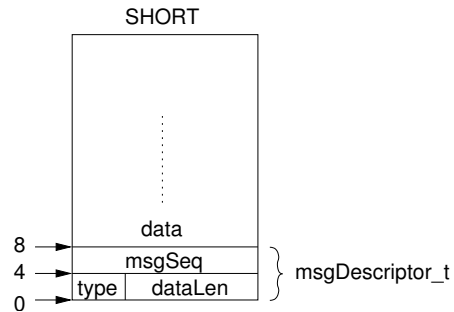


Figure 7: Structure of a SHORT message

To send a SHORT message data is copied into the local send buffer, the descriptor is set up and the complete message is transferred to the remote receive buffer by remote write operations. One or two copy operations are needed depending on if the message wraps around the end of the receive buffer. Send gathering is achieved using the three internal functions:

`startShortMsg()` Sets the message type in the send buffer to `MSG_TYPE_SHORT` and copies the data to the beginning of the send buffer data area.

`appendShortMsg()` Appends the new data to the send buffer.

`flushShortMsg()` Assigns a message number to the message, sets the data length in the descriptor and performs the SCI copy operation(s).

The SCI copy operations are checked for errors if data error checking is requested.

After transferring the data the local and remote `tx` pointers in `MSG_HEAD` are updated. The remote write operation is checked using a so called sequence mechanism if protocol error checking is enabled. However, explicit protocol checking can be saved to reduce latency, instead the caller must regularly kick `SCILib` to retransmit `tx` from local to remote `MSG_HEAD`. The overhead of this is about $0.2 \mu sec$ and thus negligible.

5.3.4 LONG Protocol

The LONG message protocol is intended to be used for large messages. The difference from the SHORT protocol is that data is copied directly to the receive buffer without an intermediate local copy. As a requirement the source buffer must be aligned to a 4-byte boundary in order to satisfy `SCIMemCpy()`. LONG messages are always sent in two steps, first the message descriptor is copied to the remote buffer, then the data is transferred out of the caller's send buffer to the remote receive buffer. Data always starts

at the next 128 byte boundary beyond the message descriptor in order to achieve maximum SCI performance. Figure 8 shows how the LONG messages are structured.

Error checking and protocol information handling is identical to the SHORT protocol.

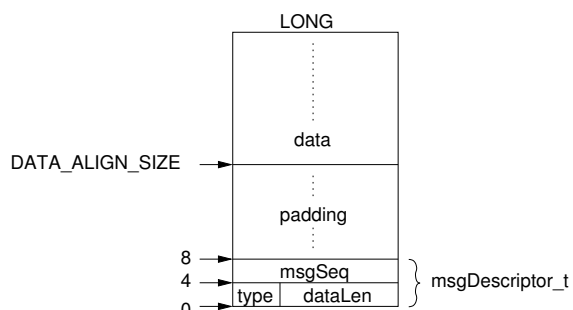


Figure 8: Structure of a LONG message

5.3.5 Receiving

The SCILib message queues provide streaming semantics for data transfers, i.e. the sender and the receiver do not have to use matching buffer sizes. A receive operation picks up where the previous one left off. This section explains how this has been implemented on top of the message transfer protocols described above.

`SCILReceiveMsgPartial()` must be used to get data from an SCILib message queue. It tries to receive as much data as possible to the given buffer. The actual number of bytes received is returned. This operation is non-blocking. If there is no data available, error code `SCI_ERR_EWOULD_BLOCK` is returned.

Everytime `SCILReceiveMsgPartial()` is called it checks if there is a pending message that has not been completely read yet by using the structure `pendingMessage_t`, which is a member of `sci.msqueue_t`:

```
typedef struct pendingMessage {
    unsigned int    type;
    unsigned int    length;
    genericMessage_t *msgBuff;
    unsigned int    rxPos;
} pendingMessage_t;
```

`type` is one of the following:

MSG_TYPE_INVALID No pending message

MSG_TYPE_INLINE Incompletely received `INLINE` message

MSG_TYPE_MSQ Incompletely received `SHORT` or `LONG` message

If there is no pending message the receiver checks if a new message is available. There are two sources of messages, the `INLINE` receive buffer and the `SHORT/LONG`

receive buffer. In order to maintain data ordering between them each message is assigned a unique sequence number by the sender. The receiver looks for a message with the next expected sequence number at the head of both buffers. The type of the message is recorded in `type`. `rxPos` keeps the current receive position within this message and is initialized to zero.

Once a pending message is available, as much data as requested, but at most as much as is contained in the current message, is copied to the caller's buffer, starting at `rxPos`. `rxPos` is updated according to the amount of data read. If the current message has been completed, `type` is set to `MSG_TYPE_INVALID`, and the message is acknowledged. That means that `rx` in `MSG_HEAD` is advanced for `SHORT` and `LONG` messages. `INLINE` send buffers are freed as described in section 5.3.2.

5.4 Stream Sockets based on SCILib

This section describes how the stream socket communication using the functions `sendmsg()` and `recvmsg()` has been built on top of SCILib. Other func such as `write[v]()`, `read[v]()`, `send()`, `receive()` etc. are mapped onto these two.

In the following we assume that a connection has already been established, i.e. there is an SCILib message queue in each direction.

5.4.1 Sending

The function `sendmsg()` is passed a pointer to a `struct msghdr` structure which describes the data source as a scatter-gather array using an `struct iovec` structure.

Depending on the message size and the buffer alignment different transfer protocols are used.

1. If the complete message fits into a single `INLINE` message, all `iovec` buffers are copied into a contiguous gather buffer and then sent through the message queue. Gathering is skipped if only a single `iovec` is given. Otherwise the `iovecs` are sent one after another.
2. If the current `iovec` buffer is well aligned (at 8 byte boundary), as much of it as possible is sent. Depending on that size and on its configuration SCILib will use the `SHORT` or the `LONG` protocol. This step is repeated until the whole `iovec` has been sent.
3. If the buffer is not well aligned the largest possible part of it, up to an aligned address, is sent as a `SHORT` message since it imposes no alignment restrictions. The remainder is processed as in step 2.

If the `NO_DELAY` option is set for the socket, data is transferred immediately. Otherwise the gathering mechanism of SCILib is used to collect small chunks of data into one large message in order to improve throughput, in analogy to TCP's Nagle algorithm [17].

5.4.2 Receiving

The function `recvmsg()` is passed a pointer to a `struct msghdr` structure which describes the data destination as a scatter-gather array using an `struct iovec` structure.

The `iovs` are filled one after another by retrieving data from the receive message queue using `SCILReceiveMsgPartial()`. If there is not enough data available, the calling process is put to sleep until new data arrives.

5.4.3 Blocking communication

Blocking sockets communication has been implemented using a combination of polling and sleep/wake-up based on SCI remote interrupts. The `SCILib` functions `SCILRequestInt()` and `SCILIsRequestedInt()` help to reduce the number of trigger operations. The remote interrupt mechanism is encapsulated in a separate `MBox` module. It provides independent notification channels whereby the target side can wait for an event explicitly or register a callback routine, and the initiator side triggers an event. The implementation details of `MBox` are beyond the scope of this document.

If `sendmsg()` or `recvmsg()` cannot be completed since the message queue is full or empty, respectively, and `MSG_DONTWAIT` was not specified, the calling process is delayed until the message queue condition changes. That delay method is a variable combination of polling and sleeping, where the time to poll before going to sleep is configurable. A poll time of zero means immediate sleeping. Once `sendmsg()` or `recvmsg()` have decided to go to sleep the remote end is informed that it must send a notification when the buffer status changes. This is done by calling `SCILRequestInt()` for the send or receive message queue, respectively. `SCILRequestInt()` performs a remote write operation. The process is blocked until it is woken up by the `MBox` callback. When woken up, the message queue condition is checked again, and if positive, the transfer operation is continued after cancelling the notification request.

A notification is sent through `MBox` during send and receive operations every time the message queue status has been changed and the remote side has asked for notification, which is checked by `SCILIsRequestedInt()`. That function performs a test on local memory, thus it is very fast.

6 Performance Evaluation

The hardware platforms used for latency benchmarks are two dual AMD Opteron 244 servers running at 1800 MHz and two dual AMD 760MPX based machines with Athlon MP2600+. Each machine was equipped with a Dolphin D331 PCI-SCI adapter card running in a 64Bit/66MHz PCI slot. Gigabit Ethernet results were obtained on

dual Xeon 2.4 GHz machines in conjunction with Intel 82544GC network controllers.

Figure 9 presents the small message one way latency of SCI SOCKET on both AMD platforms. The Opteron systems perform best, achieving an average one byte latency of $2.31 \mu\text{sec}$. Minimal user latency was found to be $2.26 \mu\text{sec}$. Average time for a one byte transfer on the Athlon machines is $3.19 \mu\text{sec}$, best case is $2.90 \mu\text{sec}$.

For comparison, an implementation of the Sockets Direct Protocol over Infiniband is reported to achieve $28 \mu\text{sec}$ latency for 2 byte messages [18].

The SCI Kernel Sockets results are remarkable as they show the different behaviour of Opteron and Athlons systems. On Opterons the kernel socket latency is $0.37 \mu\text{sec}$ higher than the user socket latency, while on Athlons kernel sockets beat user sockets by $0.32 \mu\text{sec}$. We have measured system call times to be in the range from 0.18 to $0.32 \mu\text{sec}$ on Opterons and from 0.27 to $0.47 \mu\text{sec}$ on Athlons.

A simple ping-pong test program based on `send()` and `recv()` was used in these tests.

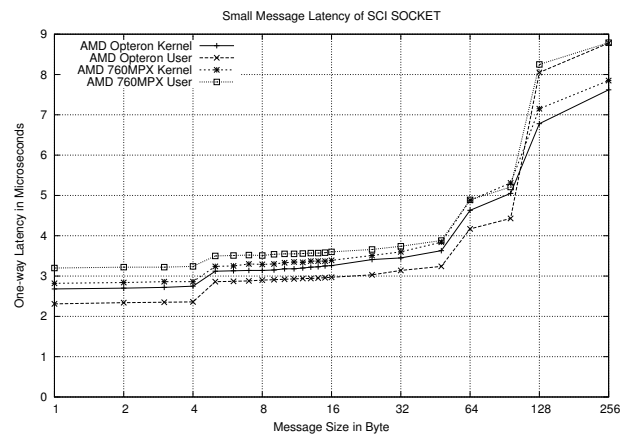


Figure 9: Small Message Latency of SCI SOCKET

The graph shows steps at 5, 64 and 128 byte. While the first two steps result from the copy size optimization for the `INLINE` protocol, explained in section 5.3.2, the third step is caused by switching from `INLINE` to `SHORT` protocol at 117 bytes. The latency for an `INLINE` message is dominated by the two necessary remote write operations: the message from sender to receiver and the acknowledge back. CRC calculation has only minor influence since there is nearly no increase to be seen within one copy size step, i.e. between 1 and 4, 5 and 52, and between 53 and 116.

In addition to our self-made ping-pong test we have used the `Netperf` microbenchmark to evaluate `SCI SOCKET`. We also ran it on Gigabit Ethernet.

First we present the results for `Netperf`'s "request-response" test which is essentially a ping-pong communication, but performance is reported in transactions per second. A transaction is defined as the exchange of a single request and a single response. Round trip time and one way latency can be inferred from the transaction rate. The

results for SCI SOCKET and Gigabit Ethernet on Opteron machines are shown in figure 10. SCI User Sockets reach a sustained transaction rate of about 203800, corresponding to a one way latency of $2.45 \mu\text{sec}$. Again, the steps are to be seen. Gigabit Ethernet results are one order of magnitude worse at about 21400 or $23.2 \mu\text{sec}$. This is where the Dolphin PCI-SCI bridge in combination with a very low overhead protocol can bring its potential to bear. SCI Kernel Sockets score about 184800 transactions per second. Next, we will look at bulk transfers.

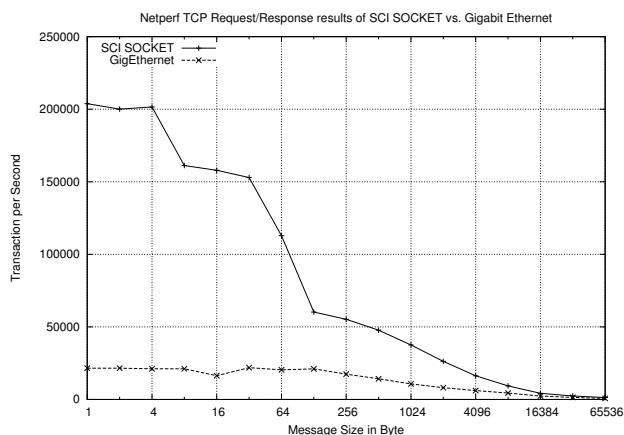


Figure 10: Netperf Request/Response Results

Netperf also provides a benchmark to measure unidirectional stream performance where the sender continuously pushes data over the socket connection and the other side receives it as fast as possible. The results for the Opteron and Athlon MPX systems are shown in figure 11. The maximum stream bandwidth of SCI User and Kernel Sockets is 186 MB/s, while user sockets are faster for small message sizes. In the same tests AMD 760MPX Athlon systems reached transfer rates of about 257 MB/s (user) and 256 MB/s (kernel).

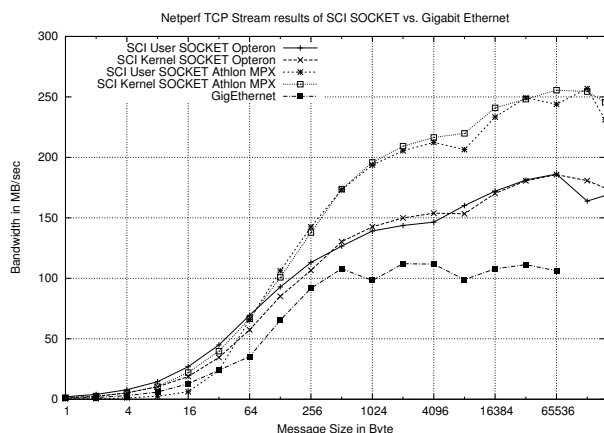


Figure 11: Netperf Stream Results

7 Summary and Outlook

Our implementation of the BSD sockets API achieves record breaking low latency by exploiting the capabilities of Dolphin's PCI-SCI bridge. Its transparent access semantics to remote memory in combination with CRC checksums greatly reduce overhead for very small messages.

Special emphasis has been put on allowing fully transparent integration with existing applications. By using the preload mechanism of the dynamic loader it is possible to make applications use the Dolphin SCI interconnect for socket connections without any change to the code, even without recompilation or relinking.

SCI supports channel bonding where 2 or more adapter cards can be used to achieve 2x, 3x or 4x the throughput of one SCI card. This can be done without sacrificing the low latency of the memory mapped transfer. SCI SOCKET will be enhanced to take advantage of this, as well as automatic fault isolation, if one or more of the multiple connections fail.

While the current release of SCI SOCKET runs on Linux only, we are considering porting it to Windows using Winsock Direct and also to Solaris.

References

- [1] "ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI)," 1998.
- [2] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.
- [3] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler, "High-Performance Local-Area Communication With Fast Sockets," in *Proc. of Winter 1997 USENIX Symposium*, January 1997, pp. 257–274.
- [4] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, pp. 256–266.
- [5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.
- [6] Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten, "Stream Sockets on SHRIMP," in *Communication, Architecture, and Applications for Network-Based Parallel Computing*, 1997, pp. 16–30.

- [7] Jin-Soo Kim, Kangho Kim, and Sung-In Jung, "SO-VIA: A User-level Sockets Layer Over Virtual Interface Architecture," in *Proc. of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)*, Los Angeles, CA, USA, October 2001.
- [8] Compaq, Intel and Microsoft Corporations, *The Virtual Interface Architecture Specification Version 1.0*, December 1997, Available at <http://www.vidf.org>.
- [9] P. Balaji, P. Shivan, P. Wyckoff, and Dhabaleswar Panda, "High Performance User Level Sockets over Gigabit Ethernet," in *Proc. 4th IEEE International Conference on Cluster Computing (Cluster 2002)*, Chicago, Illinois, USA, September 23–36 2002.
- [10] Markus Fischer, "GMSOCKS - A Direct Sockets Implementation on Myrinet," in *Proceedings of the IEEE Conference on Cluster Computing CLUSTER'01*, Newport Beach, CA, October 2001.
- [11] Intel Corporation, "Offload Sockets Framework and Sockets Direct Protocol High Level Design, Draft 2," June 2002.
- [12] InfiniBand Trade Association, *InfiniBand Architecture Specification, Release 1.0*, October 2000.
- [13] Institute of Electrical and Electronics Engineers, New York, NY, *IEEE Standard for the Scalable Coherent Interface (SCI)*, ANSI/IEEE Std. 1596-1992 edition, August 1993.
- [14] Intel Corporation, "Linux System Software for the InfiniBand Architecture," August 2002.
- [15] Dolphin Interconnect Solutions, "IP over SCI," http://www.dolphinics.no/products/software/sci_ip.html.
- [16] Ingo Molnar Ulrich Drepper, "The Native POSIX Thread Library for Linux," Tech. Rep., RedHat, Inc., January 2003, <http://people.redhat.com/drepper/nptl-design.pdf>.
- [17] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, January 1984.
- [18] Pavan Balaji, Sundeep Narravula, Karthikeyan Vaidyanathan, Savitha Krishnamoorthy, Jiesheng Wu, and D. K. Panda, "Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2004)*, Austin, Texas, 2004.