Tripwire: A Synchronisation Primitive for Virtual Memory Mapped Communication

David Riddoch[1,2], Steve Pope[1], Derek Roberts[1], Glenford Mapp[1], David Clarke[1], David Ingram[1], Kieran Mansley[1], Andy Hopper[1,2]

{djr, slp, der, gem, djc, dmi, kjm, ah}@uk.research.att.com

[1] AT&T Laboratories Cambridge,
24a Trumpington Street, Cambridge, England

[2] Laboratory for Communications Engineering,
Department of Engineering, University of Cambridge, England

Abstract

Existing user-level network interfaces deliver high bandwidth, low latency performance to applications, but are typically unable to support diverse styles of communication and are unsuitable for use in multiprogrammed environments. Often this is because the network abstraction is presented at too high a level, and support for synchronisation is inflexible.

In this paper we present a new primitive for in-band synchronisation: the *Tripwire*. Tripwires provide a flexible, efficient and scalable means for synchronisation that is orthogonal to data transfer.

We describe the implementation of a non-coherent distributed shared memory network interface, with Tripwires for synchronisation. This interface provides a low-level communications model with gigabit class bandwidth and very low overhead and latency. We show how it supports a variety of communication styles, including remote procedure call, message passing and streaming.

1 Introduction

It is well known that traditional network architectures, such as ethernet and ATM, deliver relatively poor performance to real applications, despite impressive increases in raw bandwidth. This has been shown to be largely due to high software overhead. To address this problem a variety of user-level network interfaces have been proposed[1, 2, 3, 4, 5, 6]. These have achieved dramatically reduced overhead and latency, and deliver substantial performance benefits to applications.

However, these new interfaces are typically unsuitable for use in general purpose local area networks. The interface is often designed to support a particular class of problem, and is thus a poor solution for others. In some cases the interface does not scale well with the number of applications and endpoints on a single system. What is needed is a network interface that will deliver high performance for a wide variety of communication styles, and do so in a multiprogrammed environment.

For data transfer, non-coherent distributed shared memory offers very low overhead and latency, and high bandwidth. Small messages can be transferred with a few processor reads or writes, and larger messages using DMA. Messages are delivered directly into the address space of the receiving application with no CPU involvement, and because the interface presented is low-level, it is also very flexible.

A key problem for distributed shared memory based communications, however, is that of *synchronisation*. For example, the DEC Memory Channel has a communication latency of only $2.9\mu s$, yet acquiring an uncontended spin-lock takes $120\mu s$[7]. In order to realise high performance for applications in a multiprogrammed environment it is critical that applications be able to synchronise in a timely, efficient and scalable manner.

Mechanisms used for synchronisation in existing solutions include polling known locations in memory corresponding to endpoints, raising an interrupt when certain pages are accessed, or sending out-of-band synchronisation messages. These solutions are inflexible, and typically give high performance only for restricted styles of communication and classes of problem.

The main contribution of this paper is a novel in-band synchronisation primitive for distributed shared memory systems: the *Tripwire* . Tripwires provide a means to synchronise with reads and writes across the network to arbitrary memory locations. This enables very flexible, efficient and fine-grained synchronisation. Using Tripwires, synchronisation becomes orthogonal to data transfer, decoupled from the transmitter, and a number of optimisations which reduce latency become possible.

In the rest of this paper we present a new gigabit class network interface which provides non-coherent distributed shared memory with Tripwires for synchronisation. We describe the low-level software interface to the network, support for efficient and scalable synchronisation, and go on to give some performance results. We also show how the network interface can be used to implement a variety of higher-level protocols, including remote procedure call, message passing and streaming data. Initial results suggest that excellent performance and scalability is delivered to a wide variety of applications.

This project derives from earlier work remoting peripherals[8] in order to support the Collapsed LAN[9].

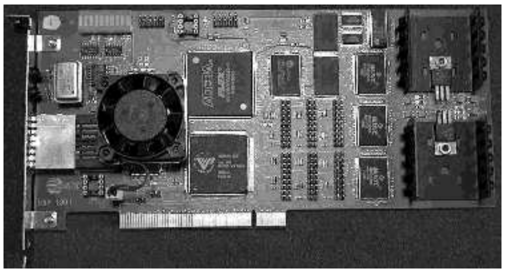

Figure 1: The CLAN NIC.

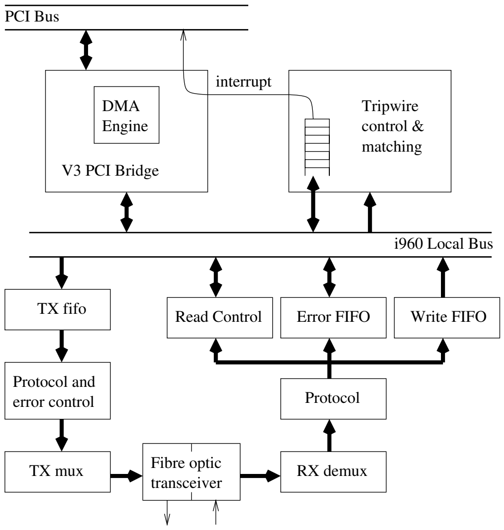

Figure 2: Logical structure of the CLAN NIC.

2 The Interconnect Hardware

We have implemented a user-level Network Interface Controller (NIC) based on a non-coherent shared memory model. The current implementation of the NIC is a 33 MHz, 32 bit PCI card, built from off the shelf components including a V3 PCI bus bridge, an Altera FPGA, and HP's G-Link chip set and optical transceivers. The latter serialise the data stream at 1.5 Gbit/s, and were chosen to map neatly onto the speed of the target PCI bus.

This platform has enabled us to perform synchronisation experiments at gigabit speeds whilst retaining the flexibility of FPGA timescales. Figure 2 shows a schematic for the NIC.

2.1 Non-coherent shared memory

The basic communications model supported by the NIC is that of non-coherent shared memory. Using this model, a portion of the virtual address space of one application is mapped over the network onto physical memory in another node, which is also mapped into the address space of an application on that node. Data transfer is achieved by an application issuing processor read or write instructions to the mapped addresses — known as Programmed I/O or PIO. The NIC also has a programmable DMA[1] engine, enabling concurrent processing and communication.

Read and write requests arriving at the NIC are processed in FIFO order, and likewise at the receiving NIC; thus PRAM[10] consistency is preserved. Applications must use memory barrier instructions at appropriate places to ensure that writes are not re-ordered by the processor or memory management hardware when order is significant.

It should be noted that this interface is not intended to provide an SMP-like shared memory programming

1. Direct Memory Access

model. Although some algorithms designed to run on an SMP might work correctly on a non-coherent memory system, the performance is likely to be very poor due to the non-uniform memory access times. Rather it is intended that higher-level communications abstractions be implemented on top of this interface, hiding the details of the hardware and peculiarities of the platform, which include:

- The latency for network reads is nearly twice that for writes.
- On some platforms there are size and alignment restrictions when accessing remote memory.

2.2 Wire protocol

Write requests for remote memory arrive at the NIC over the PCI bus, and are serialised immediately into packets, each of which represents a write burst. A write packet header simply contains a destination address, and is followed by the burst data. The length of the burst is not encoded in the header, and so a packet can start to be emitted as soon as the first word of data arrives at the NIC, and finishes when no more data within the burst is available. Thus the minimum packet size is just a header and a single word of data, and the maximum is bounded only by the largest write burst that is generated. This burst based protocol is similar to that used in the Cray T3D supercomputer network[11].

The address of any data word within a packet can be calculated from the address in the packet header and the offset of the data word within the packet. A consequence of this is that packets may be split at any point, as a new header for the second part of the packet can be calculated trivially. Similarly consecutive packets that happen to represent a single burst can easily be identified, and coalesced. This technique is well known within the memory system of single systems, but here is being extended into the LAN.

The utility of this can be seen when implementing a switch, an important function of which is to schedule the network resource between endpoints. It is important that a large packet does not hog an output port of the switch unfairly, and so it should be possible to split packets. Using the protocol presented it is possible to do this in a worm-hole switch with no buffering, whilst preserving the efficiency of large packets when there is no contention. Even if a packet is split, it may potentially be coalesced at another switch further on its route.

Hardware flow control is based on the Xon/Xoff scheme, but we expect to move to a new scheme combining Xon/Xoff with credits in the next implementation of the NIC. Error detection is performed using parity.

It should be noted that other physical layers and link layer protocols are possible and desirable, and by using an existing standard we would make use of commodity high performance switches.

2.3 True zero copy

The network interface and on-the-wire protocol described above are able to support true zero copy through the whole network. Read and write requests (whether from programmed I/O or DMA) are serialised by the NIC immediately, without buffering. This is shown in Figure 3.

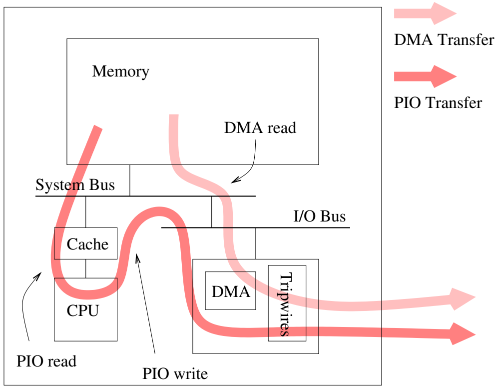

Figure 3: The path of data through the system for PIO and DMA network writes.

At the receiving NIC incoming data is transfered directly into host memory, again without buffering. The delivery of the data does not involve any processing on the CPU in the receiving node, and so has very low overhead. As shown in Figure 4, the cache on the receiving node snoops the system bus, and if data in the cache is written by a remote node, it will be updated or invalidated.

2.4 Synchronisation

In traditional network architectures all messages pass through the operating system kernel, which implicitly provides a handle to provide synchronisation with message transmission and arrival. Shared memory interfaces, however, do not of themselves offer any means for efficient synchronisation between communicating processes.

Within a single system, shared memory systems typically use some lock primitive such as the mutex and

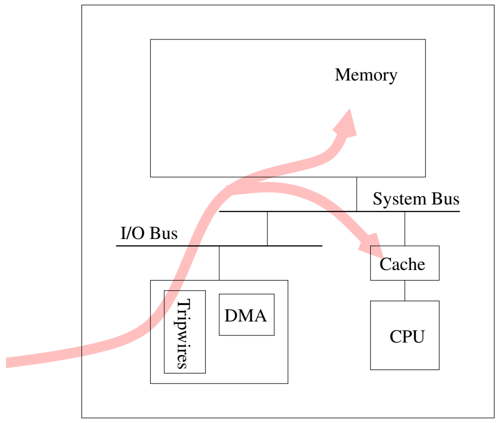

Figure 4: The path of data through a receiving node.

condition variable of pthreads, or semaphores. Mechanisms used in existing distributed shared memory systems include:

- Polling memory locations, otherwise known as *spinning*. This technique provides the best latency when servicing a single endpoint, but does not scale well with the number of endpoints and the processor resource is wasted whilst polling.

- Generate an interrupt on page access. This has been used on a number of systems, and allows the receiving process to synchronise with remote accesses to it's pages. However, the granularity is at the level of the page, so it is not possible to selectively synchronise with specific events within a data transfer.

- Explicit out-of-band synchronisation messages (as used by the Scalable Coherent Interface[12]). This solution complicates the network interface and is inflexible.

If the performance advantages of memory mapped networking are to be fully realised, then an efficient and flexible user-level synchronisation mechanism must be provided.

2.5 Tripwires for synchronisation

Here we present a novel primitive for in-band synchronisation: the *Tripwire* . Tripwires provide a means to synchronise with cross network accesses to arbitrary memory locations. Each Tripwire is associated with some memory location (which may be local or remote), and *fires* when that memory location is read or written via the network.

For example, an application can setup a Tripwire to detect writes to a particular memory location in its address space, and will be notified when that location is written to by a remote node. It is also possible to setup Tripwires to detect reads, or to specify a range of memory locations.

This is achieved by snooping the addresses in the stream of network traffic passing through the NIC, and comparing each of them with the addresses of the memory locations associated with each of the Tripwires,[2] without hindrance to the passing stream. The comparison is performed by a content-addressable memory, and a synchronising action is performed when a match is detected.

The synchronising action performed is to place an integer identifier for the Tripwire into a FIFO on the NIC and raise an interrupt. The device driver's interrupt service routine retrieves these identifiers, and does whatever is necessary to synchronise with the application owning the Tripwire. This is not the only possible type of action; others include setting a bit in a bitmap in host memory (in the address space of the application) or incrementing an event counter.

We believe that the combination of non-coherent shared memory with Tripwires for synchronisation provides a flexible and efficient platform for communications:

- The choice of *non-coherent* shared memory allows a relatively simple and highly efficient implementation with very low latency. For example, although the SCI specification includes hardware based coherency, to date very few implementations have included it.

- Communications overhead is very low; small transfers can be implemented using just a few processor writes, and larger transfers using DMA.

- Multicast can be implemented efficiently with suitable support in the switch.

- Tripwires allow highly flexible, efficient and fine-grained synchronisation; it is possible to detect reads or writes to arbitrary locations in memory.

- Synchronisation is completely orthogonal to data transfer. It is possible to synchronise with the arrival of a message, an arbitrary position in a data stream, or on control signals such as flags or counters.

2. The current incarnation of the NIC supports 4096 Tripwires.

- Synchronisation is decoupled from the transmitter. The recipient of a message decides where in the data stream to synchronise, and can optionally tune this for optimal performance.
- It is possible to reduce latency by overlapping scheduling and message processing with data transfer. See section 4.3.
- As well as synchronising with accesses to local memory by remote nodes, Tripwires can be used to synchronise with accesses to remote memory by the local node. This can be used to detect completion of an outgoing asynchronous DMA transfer for example.

Together these points mean that it is possible to implement a wide variety of higher level protocols efficiently, including remote procedure call (RPC), streaming, message passing and shared memory based distribution of state.

The current incarnation of the NIC requires that applications interact with the device driver to setup Tripwires and synchronise when a Tripwire fires. A new version, under active development, will enable applications to program Tripwires at user-level through a virtual memory mapping onto the NIC. When a Tripwire fires, notification will be delivered directly into the application's host memory, and so synchronisation may also take place at user-level. Only when the application blocks waiting for a Tripwire or other event need the driver request that the NIC generate an interrupt as the synchronising action.

3 Low Level Software

The device driver for the NIC targets the Linux 2.2 kernel on x86 and Alpha platforms, and WinDriver portable driver software (Linux and Windows NT), and requires no modifications to the operating system core. The NIC can thus be used in the vast majority of recent commodity workstations.

The interface presented by the device driver and hardware is wrapped in a thin library. The main purpose of this layer is to insulate higher levels of code from changes to the division of functionality between user-level, device driver and hardware implementation. For example, a future incarnation of the hardware may provide facilities to perform connection setup entirely at user-level.

This layer also makes it easy to write code which can be used both at user-level and for in-kernel services,[3] since it abstracts above the device driver and hardware interface.

3.1 Endpoints

An *endpoint* is identified by host address and a port number. Fixed size[4] out-of-band messages are demultiplexed by the device driver into per-endpoint queues, and are used for connection setup, tear down and exceptional conditions. The message queue resides in a segment of memory shared by the application and device driver, and so messages can be dequeued without a system call. This technique is used throughout the device driver interface to reduce the overhead of passing data between the device driver and application. This technique is described in detail elsewhere[13].

The out-of-band message transfer is itself implemented in the device driver using the NIC's shared memory interface and Tripwires, using a technique similar to that shown in Section 4.1.

An API is provided to create and destroy endpoints, and manage connections. To amortise the relatively high cost of creating endpoints and their associated resources, they may be cached and reused across connection sessions.

3.2 Apertures

An *aperture* represents a region of shared memory, either in the local host memory or mapped across the network. An aperture is identified by an opaque descriptor, which may be passed across the network and used by an application to map a region of its virtual address space onto the remote memory region, or used as the target of a DMA transfer.[5]

The NIC uses a very simple direct mapping between PCI bus addresses and network addresses, and provides no explicit protection on the receive path. Physical pages of memory in an aperture must be contiguous, and must be locked down to prevent them from being swapped out by the operating system. Basic protection is provided by the virtual memory system at the granularity of the page — a process may only create mappings to remote apertures for which it holds a descriptor — but a faulty or malicious node could circumvent this protection. We will be addressing these problems in future revisions of the NIC.

It should be noted that although facilities are provided for managing connections, this is not the only

3. Such as IP and NFS, see Section 6.
4. 40 byte payload.
5. In this context the descriptor is known as an RDMA cookie.

model supported. Any number of apertures may be associated with an endpoint, and mappings can be made to any aperture in any other host independently of connections.

3.3 Tripwires

An API is provided to allocate Tripwires, and to setup and enable Tripwires to detect local and remote reads and writes. The application may test the state of a Tripwire, or block until a Tripwire fires.

When a Tripwire fires the NIC places an integer identifier for the Tripwire into a FIFO and raises an interrupt. The interrupt service routine wakes any process(es) waiting for that Tripwire, and sets a bit in a bitmap. This bitmap resides in memory shared by the application and device driver, and so it is possible to test whether a Tripwire has fired at user-level. Thus polling Tripwires is very efficient.

A Tripwire may be marked *once only*, so that it is disabled when it fires. This improves efficiency for some applications, by saving a system call to disable the Tripwire, but is unlikely to be necessary when Tripwire programming is available at user-level.

A *Tripset* groups a number of Tripwires from a single endpoint together so that the application can wait for any of a number of events to occur.

3.4 DMA

The NIC has a single DMA engine, which reduces communications overhead by allowing concurrent data transfer and processing on the CPU. There is currently no direct user-level interface to DMA hardware, so DMA requests must be multiplexed by the device driver.

All DMA interfaces that we are aware of require that the application invoke the device driver or access the hardware directly for each DMA request. We have implemented a novel interface which reduces the number of system calls required and schedules the DMA engine fairly among endpoints. The application maintains a queue of DMA requests in a region of memory shared with the device driver, and the device driver reads requests from that queue.

When the first request is placed in the queue, the application must invoke the device driver. After that the device driver will dequeue entries asynchronously, driven by the DMA completion interrupt, until the queue is empty. The application may continue to add entries to the queue, and only need invoke the device driver if the queue empties completely. The device driver maintains a circularly linked list of DMA request queues which are not yet empty, and services them in a round-robin fashion. Note that this asynchronous interface would work just as well if the DMA engine were managed on the NIC.

3.5 Multiple endpoints

The primitives discussed above are sufficient to manage communication through a single endpoint efficiently, bit it is also necessary to have non-blocking management of multiple endpoints to support common event driven programming models typically used in server applications. Traditionally some variant on the BSD *select* system call is used. However, the deficiencies of these are well known, and even with sophisticated optimisations[14] do not scale well with the number of endpoints.

We have developed a solution to this problem in the form of an *asynchronous event queue*. Tripwire, DMA completion and out-of-band message events are delivered by the device driver into a circular queue. The API allows the application to register and unregister interest in individual events, poll for events and block until the queue is non-empty.

As with out-of-band messages above, this queue is maintained in memory shared by the device driver and application. The cost of event delivery is $O(1)$ and events are dequeued without a system call — so this interface is very efficient. Indeed event delivery becomes more efficient when a server is heavily loaded with network traffic, since the server is less likely to have to make a system call to block before retrieving events. More information is given elsewhere[13].

Ideally such a mechanism should be a standard part of an operating system, but while that is not the case it is important to be able to block waiting for I/O activity on other devices in the system as well as the network. To support this the asynchronous event queue itself is integrated with the system's choice of 'select' variant, and becomes 'readable' when the queue is not empty.

A number of other solutions to this problem with $O(1)$ cost have been proposed, including POSIX realtime signals, the `/dev/poll` interface in the Solaris operating system and others[15]. The main contribution of our approach is the delivery of events directly to the application without a system call.

4 Programming the NIC

4.1 Simple message transfer

Figure 5 shows how a simple, one-way, point-to-point message transfer protocol is implemented.

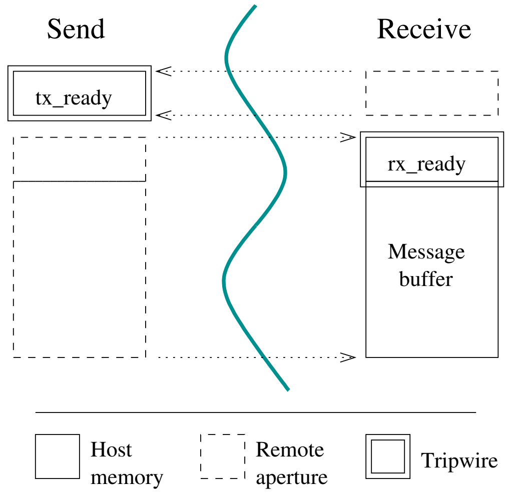

Figure 5: Message transfer using semaphores.

On the transmitting side `tx_ready` is a boolean flag which is *true* when the buffer is free for sending. On the receive side `rx_ready` is true when a valid message is ready in the buffer. Initially `tx_ready` is set to true and `rx_ready` is set to false.

To send a message the sender must wait until `tx_ready` is true, set it to false (to indicate that the buffer is busy), copy the message into the remote buffer and set `rx_ready` to true. The receiver waits for `rx_ready` to become true, clears it, copies the message out of the buffer and resets `tx_ready` to true, so that the buffer can be used again.

This protocol will work as described on any PRAM-consistent shared memory system — the difficulty being synchronising with updates to the `tx/rx_ready` flags. This is achieved using Tripwires.

A C program implementing this protocol is shown in Example 1. For simplicity the example shows that the message data is copied twice — into and out of the message buffer — but this is not necessary. The sender could write the message data directly into the remote buffer, and on the receiving side the application could read the message contents directly from the message buffer, giving true zero copy.

We have used this simple protocol to support a remote procedure call (RPC) style of interaction. The client writes a request into the server's message buffer, and when the RPC completes the result is written back into the client's message buffer. This arrangement only requires two control flags and two Tripwires (one in the client and one in the server), since the order in which messages can be sent is restricted.

Example 1 Simple message transfer.

```
void send_msg(const void* msg)
{
    while( !tx_ready )
        tripwire_wait(tx_ready_trip);
    tx_ready = 0;
    memcpy(remote->buffer, msg, msg_length);
    remote->rx_ready = 1;
}

void recv_msg(void* msg)
{
    while( !rx_ready )
        tripwire_wait(rx_ready_trip);
    rx_ready = 0;
    memcpy(msg, buffer, msg_length);
    remote->tx_ready = 1;
}
```

4.2 A distributed queue

A disadvantage of the above protocol is that the sender and receiver are tightly coupled — they must synchronise at each message transfer. This prevents streaming and hence decreases bandwidth. This can be alleviated by using a queue as shown in Figure 6. The corresponding C program is shown in Example 2.

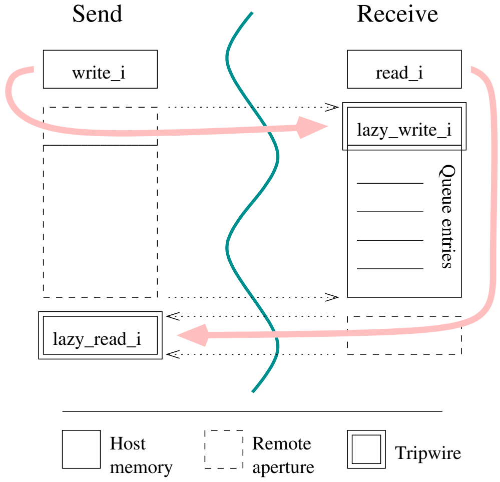

Figure 6: A distributed queue.

This implementation works similarly to a type of queue commonly used within applications. Fixed size messages are stored in a circular buffer. Two counters, `read_i` and `write_i`, give the positions in the buffer at which the next message should be dequeued and enqueued respectively.

The traditional implementation uses an additional flag to distinguish between the full and empty cases when the counters are equal. However, this requires a lock to ensure atomic update, which would significantly degrade performance in a distributed implementation. For this reason we do not use the additional flag; the queue is considered to be empty when the counters are equal, and full when there is only one empty slot in the receive buffer. This effectively reduces the size of the queue by one.

Example 2 A distributed queue.

```
void q_put(const q_elm* elm)
{
    while( (write_i + 1) % q_size == lazy_read_i )
        tripwire_wait(lazy_read_i_trip);
    remote->q[write_i] = *elm;
    write_i = (write_i + 1) % q_size;
    remote->lazy_write_i = write_i;
}

void q_get(q_elm* elm)
{
    while( read_i == lazy_write_i )
        tripwire_wait(lazy_write_i_trip);
    *elm = q[read_i];
    read_i = (read_i + 1) % q_size;
    remote->lazy_read_i = read_i;
}
```

The distribution of the control state between the sender and receiver is motivated by the desire to avoid cross-network reads. The sender 'owns' the `write_i` counter, which it updates each time a message is written into the remote buffer. A copy of this counter, `lazy_write_i`, is held in the receiver, the sender copying the value of `write_i` to `lazy_write_i` each time the former is updated. The `read_i` counter is maintained similarly, with the receiver owning the definitive value.

Thus the sender's and receiver's lazy copies of `read_i` and `write_i` respectively may briefly be out-of-date. This is *safe* though, since the effect is that the receiver may see fewer messages in the buffer than there really are, and the sender may believe that the buffer contains more messages than it does. The inconsistency is quickly resolved.

4.3 Discussion

When designing the two protocols given above, care was taken to ensure that all cross-network access were writes. This is important because reads have a relatively high latency, and generate more network traffic. In each case the control variables that must be read by an endpoint are stored in local memory. This memory is cacheable, so polling the endpoints is very efficient.

It is interesting to note that both of the protocols given above will work correctly without Tripwires, by polling the values of the control variables, and this choice can be made independently in the transmitter and receiver. Tripwires merely provide a flexible way to synchronise efficiently, and a number of enhancements are possible:

- If a message is expected soon, the application may choose to poll the control variables for a few microseconds before blocking. This reduces latency and overhead by avoiding a system call[16].
- It is possible to overlap data transfer with scheduling of the receiving application. This is done by setting a Tripwire in the receiving application for some memory location that will be written early in the message transfer, and going to sleep. The application will be rescheduled as the rest of the message is being transfered, thus reducing latency.
- It is even possible to begin processing the header of a message in parallel with receiving the body, by setting separate Tripwires for the header and body of the message.
- DMA or PIO can be used for the data transfer. The choice will depend on the characteristics of the application and the size of the messages.
- The distributed queue can be adapted in a number of ways, including support for streaming unstructured data, sending multi-part messages, and passing meta-data.

5 Raw Performance

In this section we present a number of microbenchmarks which demonstrate the raw performance of the hardware, and how this translates into high performance for practical applications. All tests are performed at user-level.

5.1 System characterisation

The test nodes are 530 MHz Alpha systems running the Linux 2.2 operating system. Fine grain timing was performed using the free running cycle counter. A number of system parameters were first measured:

- The system call overhead (measured using an *ioctl*) is about $.75\mu s$. The true overhead of system calls is made significantly worse, however, by the effect they have on the cache.

- The scheduler overhead when a single process is un-blocked is $8\mu s$.

- The interrupt latency is $3\mu s$.

5.2 Shared memory

These benchmarks exercise the distributed shared memory interface, and use 'spinning' for synchronisation.

1. The round-trip time for a single DWORD was measured by repeatedly writing a word to a known location in a remote application, which then copied that word into a known location in the sending application. The round-trip time is $3.7\mu s$, giving an application to application one-way latency of $1.9\mu s$.

2. The raw bandwidth was found by measuring the interval between the arrival of the first DWORD of a message at the processor, and arrival of the last DWORD. Figure 7 shows the results for DMA and PIO. Peak bandwidth is 910 Mbit/s, and is limited by a stall for one cycle on the receiving V3 PCI bridge at the start of each burst.

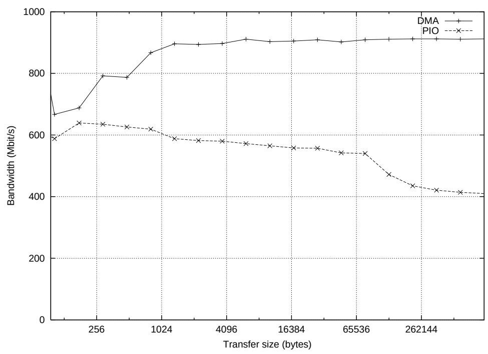

Figure 7: Raw bandwidth vs. size

The plots given in Figure 7 need some further explanation. Firstly, it is not possible to meaningfully measure the bandwidth for messages smaller than about 256 bytes, since blocking effects of the cache and various bus bridges dominate. The latency across the NIC hardware (measured using a logic analyser) is about $.5\mu s$, so $.9\mu s$ are spent traversing buses and bridges — which are likely holding data back in order to generate large bursts — and cache loading.

The surprising results for programmed I/O are due to a bug in the V3. Under some circumstances, the V3's internal FIFOs lock, causing the V3 to emit single word bursts until the FIFOs drain. A workaround in our FPGA logic detects this condition, and backs off for a time to allow the V3 to recover. This effect causes the negative gradient seen in the figure.

The receiving V3 generates 16 DWORD bursts on the PCI bus for PIO traffic, and 256 DWORDS for DMA traffic. Together with the additional V3 stall at the start of each burst, this severely limits the PIO bandwidth. To address this we have some initial work implementing coalescing of bursts on the receiving NIC. Initial tests suggest that with this logic in place the performance of PIO will be very similar to that of DMA.

5.3 An event-based server

The benchmarks presented in this section are designed to exercise the Tripwire synchronisation primitive, and show how this interface scales and performs under load.

The server application is built using a single-threaded event processing model, the main event loop being based on the standard BSD *select* system call. The file descriptor corresponding to an asynchronous event queue (section 3.5) is a member of the *select set*, and becomes readable when there are entries in the queue. In this case a handler extracts events from the queue and dispatches them to call-back functions associated with each endpoint. The event queue is serviced until it is empty, and then polled for up to $15\mu s$ before returning to the main select loop.

When a connection is established a Tripwire is initialised to detect client requests. This Tripwire is registered with the asynchronous event queue, and the request management code receives a call-back whenever a request arrives.

The test client makes a connection to the server, and issues requests to perform the various benchmarks. When waiting for an acknowledgement or reply the client spins, which is reasonable since the reply will usually arrive in less time that it takes for a process to go to sleep and be rescheduled.

5.3.1 Round-trip time

In the first set of tests we measure the round-trip time for a small message under differing conditions of server load. Each test is repeated a large number of times, and the result reported is the mean:

- When a large ($> 15\mu s$) gap is left between each request the server goes to sleep between requests, and the round-trip time consists of the transmit overhead, the hardware latency, the interrupt overhead, the rescheduling overhead, the message processing overhead and the reply. The total is $16.8\mu s$.

- As explained in Section 3.5, a server using the asynchronous event queue becomes more efficient when kept busy, since it avoids both system calls and going to sleep. In this benchmark messages are sent as quickly as possible, one after another, but with only one message in flight at a time. In this case the round-trip time consists of the same steps as above, but without the reschedule, and improved cache performance. The round-trip time is reduced to just $7.9\mu s$.

- Figure 8 shows how the server performs as the number of connections increases. The two traces correspond to the server being busy and idle, as above. The client chooses a connection at random for each request. As expected, the *response time is independent of the number of connections*.

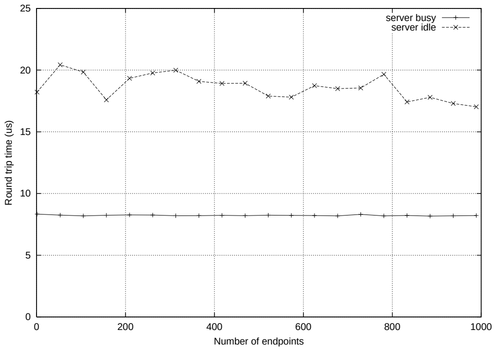

Figure 8: Round-trip time vs. number of endpoints

- In the final test multiple requests are sent by the client on different connections before collecting each of the replies. This simulates the effect of multiple clients in the network, and is shown in Figure 9. Note that even when the server is overloaded the performance of the network interface is not degraded.

  The curve is tending towards 340000 requests per second, and given that the server is effectively performing a no-op, we can assume that all the processor time is spent processing messages. Thus the total per-message overhead on the server when fully loaded is $1/340000s$, or $2.9\mu s$.

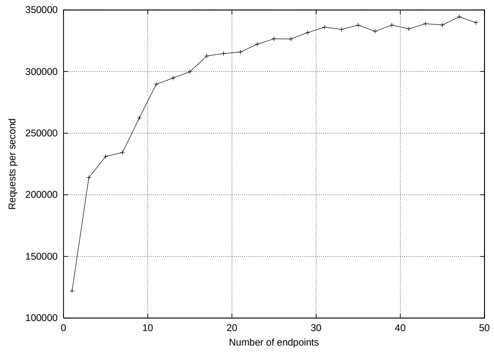

Figure 9: Requests per second vs. number of endpoints

5.3.2 Bandwidth

Figure 10 shows the bandwidth achieved sending various sizes of message one way to the server, the server acknowledging each message.

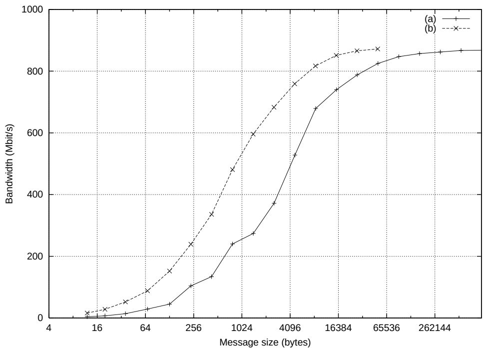

Figure 10: Unidirectional bandwidth vs. message size for (a) single messages in flight and (b) streaming.

- For curve (a) each message is sent as a single DMA, and acknowledged by the server. The client waits for the acknowledgement before sending the next message.

- Curve (b) shows the bandwidth when streaming messages. Each message is sent using DMA, and the next is started without waiting for acknowledgement of the first.

Curve (b) corresponds closely to the theoretical throughput for a link with a bandwidth of 900 Mbps and a setup time of $6\mu s$. This consists of the $3\mu s$ interrupt latency and the time taken to setup the next transfer. The interrupt overhead will be eliminated when user-level programmable DMA and chaining are supported in the hardware. Note that half the maximum available bandwidth is achieved with messages of about 750 bytes in size.

Another study[17] has compared the performance of a number of user-level communication interfaces implemented on the Myrinet interconnect, which has raw performance comparable with the CLAN NIC. Their 'Unidirectional Bandwidth' experiment is equivalent to this one, and out of AM, BIP, FM, PM and VMMC-2, only PM gives better performance than the CLAN NIC. PM[18] is a specialised network interface not suitable for general purpose networking.

6 Supporting Applications

It is important that innovations in network interface design translate into real performance improvements at the application level, for existing as well as new applications. Support for existing applications can be provided at a number of levels:

- Implement an in-kernel device driver.
- Recompile applications against a source-compatible API.
- Replace dynamically loaded libraries.

Of these 2 is likely to give the best performance, but is at best inconvenient, and often not possible. An in kernel solution will provide binary compatibility with existing applications, but performance is likely to be relatively poor due to the high overheads associated with interrupts, the protocol stack, cache thrashing and context switches. The last solution works well when the network abstraction is provided at a high level, as is the case with middleware such as CORBA and MPI.

We are actively porting a number of representative protocols and applications to our network interface, and this section presents some initial results.

6.1 IP

The Internet Protocol is the transport of choice for the majority of communication applications, and has been implemented over almost all network technologies. Because the protocol is presented to applications at a very low level, and must be fully integrated with the other I/O subsystems,[6] it is difficult to provide a fully functional implementation at user-level. We have thus chosen initially to provide support by presenting the network interface to the kernel using a standard network device driver[19].

The implementation is illustrated in Figure 11. A single connection is made on demand between pairs of nodes which need to communicate using IP. The receiving node pre-allocates a small number of buffers, and passes RDMA cookies for these buffers to the sending node through a distributed queue (as described in Section 4.2). Completion messages are passed through another queue in the opposite direction.

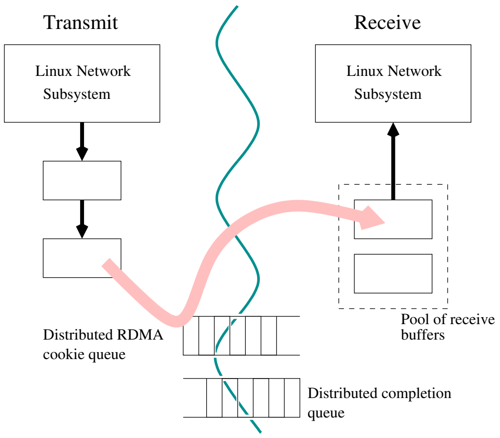

Figure 11: The implementation of CLAN IP

Data transfer consists of the following steps:

1. The Linux networking subsystem passes in a buffer (`struct sk_buff`) containing the packet data to send.
2. The packet data is transfered across the network by DMA, using an RDMA cookie from the distributed queue as the target.
3. A completion message is written into the completion queue.
4. On the receive side, a Tripwire is used to synchronise with writes into the completion queue. The buffer containing the packet data is passed on to the networking subsystem.
5. New buffers are allocated if necessary, and their RDMA cookies are passed back to the transmitting

6. Using *select* or *poll* on UNIX systems

node. The distributed RDMA cookie queue effectively provides flow control, so we aim to prevent it from emptying, since this would cause the sender to stall.

Although the development of this implementation is still work in progress, we have some preliminary results from an early version. In Table 1 we compare the round trip time and small message bandwidth for CLAN IP with fast- and gigabit ethernet technology. The test uses TCP/IP, repeatedly sending and returning a message.

	RTT	B/W at 1KByte
CLAN IP	$100\mu s$	225 Mbit/s
100 Mbit ethernet	$160\mu s$	23 Mbit/s
Gigabit ethernet	$260\mu s$	23-28 Mbit/s

Table 1: Round trip time and bandwidth for CLAN and ethernet.

Figure 12 shows the bandwidth achieved against message size for the same test. The gigabit ethernet was configured to use jumbo frames and large socket buffers (256 kilobytes) to improve its performance, whereas CLAN IP is using just 32 kilobytes per socket buffer.

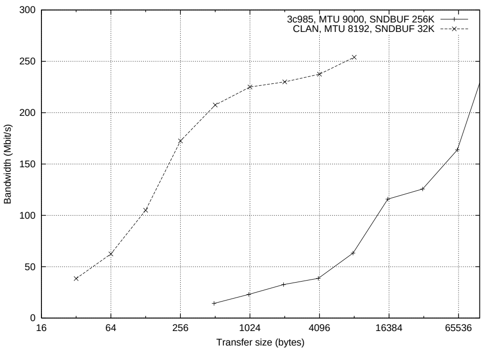

Figure 12: TCP/IP bandwidth for CLAN and gigabit ethernet, as a function of message size.

Due to a bug in the CLAN NIC which caused deadlocks[7] at the time of this experiment, the bandwidth is limited to 250 Mbit/s, so no conclusions can be drawn as to the maximum bandwidth that can be achieved over CLAN IP. Further, this implementation was only able to support a single packet in flight at any one time. We expect the small message bandwidth to be significantly improved with the new implementation, which allows streaming. Despite these problems CLAN IP is able to deliver nearly 10 times the performance of gigabit ethernet for 1 kilobyte messages.

6.2 NFS

Another service which we have implemented in the Linux kernel is the Sun Network File System Protocol[20] (NFS). The implementation consists of two sets of modifications:

- The SunRPC layer runs over a custom socket implementation over the CLAN NIC.
- A number of DMA optimisations are added at higher levels in the NFS subsystem to increase performance for file data transfer.

A CLAN connection is setup when a filesystem is mounted. NFS messages are passed through the socket layer via the SunRPC layer, whilst file data is transfered using DMA directly from the Linux buffer cache into the receiving node. As a further optimisation it is possible to transfer file data directly from the disk to the receiving node, bypassing local host memory altogether. Whether this results in a performance gain depends on the pattern of file access, and the performance of the disk controller cache.

We have not been able to perform any standard NFS benchmarks, since these mandate the use of the UDP protocol, and often implement the client side NFS. It would be necessary to implement an ethernet UDP to CLAN bridge to make use of these tests.

As an initial empirical guide we have timed the task of compiling the Linux kernel on a local file system, and one imported over NFS. In both cases the *ext2* file system is used. For the local filesystem the compile time is 6 minutes 42 seconds, and over NFS it is 6 minutes 45 seconds.

Although compilation is typically a CPU bound process, it was not parallelised in this test, so file system overhead makes a significant contribution to the compile time. This result suggests that importing a file system over CLAN NFS has very low overhead.

6.3 MPI

In order to test the performance of existing applications designed to use clusters of workstations, we have ported the Message Passing Interface standard to the CLAN network. The LAM[21] implementation was chosen for its efficient interface to the transport layer, requiring the implementation of only 9 functions. The entire MPI standard is fully supported.

7. The deadlocks are due to our implementation using a shared bus for transmit, receive and Tripwire traffic. The shared bus architecture is a consequence of using off-the-shelf components.

Synchronisation is achieved by spinning, so this implementation is most suitable for use in a system dedicated to a single problem. The round-trip time is $16\mu s$, which compares favourably with other systems, such as MPI-BIP[22], which has a round-trip time of $19\mu s$. It is interesting to note that MPI-BIP is implemented at a higher level than our version, so reducing the library overheads.

We are developing a second version using Tripwires for synchronisation, which will be suitable for use in a general purpose workstation, and coexist well with other applications. This implementation will spin for a few micro-seconds before going to sleep if no events occur — so we expect the performance to be comparable with the first version when heavily loaded.

We have tested our first implementation using the *N-body* problem, and will present these results and a detailed discussion of the implementation at a later date.

7 Related Work

Although designed as a multiprocessor system, rather than an interconnect for heterogeneous networks, the FLASH[2] project shares many of our goals. These include efficient support for a wide range of protocols in a general purpose multiprogrammed environment. The FLASH multicomputer provides cache-coherent distributed shared memory and message passing. Protocol specific message handlers run on a dedicated protocol processor in each node, and can be used to provide flexible and efficient synchronisation.

The Myrinet network interface has been used to implement a large number of user-level networks. Like the CLAN NIC, it is implemented as a PCI card, and provides gigabit class raw performance. A programmable RISC processor is used to implement a particular communications model, making this platform ideal for research purposes. A disadvantage of this approach is that at any one time, the network is programmed to support just one model, and so it is difficult to provide simultaneous support for diverse protocols. Another problem is that data is staged in memory on the NIC, which leads to a latency/bandwidth tradeoff, as explored in the Trapeze[23] project.

Other important distributed shared memory based systems include DEC Memory Channel[3], the Scalable Coherent Interface[12], SHRIMP[6] and BIP[22]. All of these have addressed the requirements for high bandwidth, low latency data transfer, providing excellent performance for a particular class of problem, but support for synchronisation is often inflexible.

The U-Net[1] project was the first to present a user-level interface to local area networks, using off-the-shelf communications hardware. U-Net heavily influenced the Virtual Interface Architecture[24], which provides a communication model based on asynchronous message passing through work queues. Completion queues multiplex transmit and receive events, so performance scales well with the number of endpoints. Disadvantages include relatively high overhead, which leads to poor performance for small messages, and high per-endpoint resource requirements[25]. End to end flow control must be handled explicitly by the application, as buffer overrun on the receive side leads to the connection being closed.

8 Conclusions

The CLAN project has addressed the problem of synchronisation in distributed shared memory systems. Our solution, the *Tripwire*, provides an efficient means for flexible and scalable synchronisation.

Tripwires have a number of advantages over existing solutions to the synchronisation problem. Synchronisation is orthogonal to data transfer, since applications may synchronise with arbitrary in-band data. A consequence of this is that synchronisation is decoupled from the transmitter, allowing great flexibility in the receiver.

In this paper we have presented a high performance network interconnect based on distributed shared memory with Tripwires for synchronisation, and shown that it delivers gigabit class bandwidth and very low overhead and latency for data transfer. The low-level software interface provides a flexible means for synchronisation that is highly scalable. These characteristics translate to high throughput and short response times for high-level protocols and practical applications.

For applications such as parallel number-crunching and multicomputer servers the performance is comparable with, and often exceeds that achieved by other specialised network interfaces. At the same time superior performance is delivered to multiprogrammed systems, making the CLAN NIC suitable for general purpose local area networks, and expanding the space of problems that can be tackled on them.

We believe that significant benefits are drawn from tailoring the network abstraction to the application, and by presenting the network interface at a low level it is possible to implement a wide range of abstractions efficiently.

9 Future Work

The authors are currently working on a number of enhancements to the CLAN network. We have just built the first prototype of a high performance, worm-hole routed switch, and will soon begin testing. Like the NIC, the switch is based on FPGA technology, and will allow us to experiment with wire protocols and routing.

In the next revision of the NIC we will implement user-level programmable Tripwires. This should further the reduce the overhead of synchronisation, and will be achieved by providing *Virtual Memory Mapped Commands* as used in the SHRIMP multicomputer[6]. We also intend to provide DMA chaining, which should significantly reduce message passing overhead and increase small message bandwidth. Another enhancement is hardware delivery of Tripwire notification to user-level. This will reduce the number of interrupts taken and reduce Tripwire notification latency.

At a higher-level, we are investigating how this network interface can be used to improve the performance of middleware. We are developing a software implementation of the Virtual Interface Architecture, and intend to implement a high-performance transport for a CORBA ORB.

Other areas for future research include:

- User-level reprogramming of aperture mappings. A number of other projects have enhanced existing network interfaces with flexible memory management[26][27]. It should be possible to extend this with re-mapping of local and remote apertures at user-level, using a pool of virtual memory mappings onto the NIC.
- Hardware delivery of out-of-band messages directly into an user-level queue. Together with reprogramming of aperture mappings, this would make it possible to perform connection setup entirely at user-level.

Acknowledgements

The authors would like to thank all of the members of AT&T Laboratories Cambridge and the Laboratory for Communications Engineering. David Riddoch is also funded by the Royal Commission for the Exhibition of 1851.

The work on coalescing write bursts to improve PIO bandwidth was performed by Chris Richardson whilst on an internship at AT&T.

References

[1] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *15th ACM Symposium on Operating Systems Principles*, December 1995.

[2] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[3] R. Gillett and R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, 17(1), 1997.

[4] Maximilian Ibel, Klaus Schauser, Chris Scheiman, and Manfred Weis. High Performance Cluster Computing using SCI. In *Hot Interconnects V*, August 1997.

[5] Nanette Boden, Danny Cohen, Robert Felderman, Alan Kulawik, Charles Seitz, Javoc Seizovic, and Wen-King Su. Myrinet — A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1), 1995.

[6] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[7] David Culler and Jaswinder Pal Singh. *Parallel Computer Architecture, A Hardware/Software Approach*, chapter 7, page 520. Morgan Kaufmann.

[8] Steve Hodges, Steve Pope, Derek Roberts, Glenford Mapp, and Andy Hopper. Remoting Peripherals using Memory-Mapped Networks. Technical Report 98.7, AT&T Laboratories Cambridge, 1998.

[9] Maurice Wilkes and Andrew Hopper. The Collapsed LAN: a Solution to a Bandwidth Problem? *Computer Architecture News*, 25(3), July 1997.

[10] Richard Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical report, Princeton University, 1988.

[11] David Culler and Jaswinder Pal Singh. *Parallel Computer Architecture, A Hardware/Software Approach*, chapter 10, page 818. Morgan Kaufmann.

[12] IEEE. Standard for Scalable Coherent Interface, March 1992. IEEE Std 1596-1992.

[13] David Riddoch and Steve Pope. A Low Overhead Application/Device Driver Interface for User-Level Networking. Paper in preparation.

[14] Gaurav Banga and Jeffrey Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX Technical Conference*, June 1998.

[15] Gaurav Banga, Jeffrey Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Technical Conference*, June 1999.

[16] Stefanos Damianakis, Yuqun Chen, and Edward Felten. Reducing Waiting Costs in User-Level Communication. In *11th International Parallel Processing Symposium*, April 1997.

[17] Soichiro Araki, Angelos Bilas, Cezary Dubnicki, Jan Edler, Koichi Konishi, and James Philbin. User-Space Communications: A Quantitative Study. In *SuperComputing*, November 1998.

[18] Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. PM: A High-Performance Communication Library for Multi-user Environments. Technical Report TR-96015, Tsukuba Research Center RWCP, 1996.

[19] Alessandro Rubini. *Linux Device Drivers*, chapter 14 Network Drivers. O'Reilly, 1998.

[20] Sun Microsystems. NFS: Network File System Protocol Specification, 1989. RFC 1050.

[21] LAM / MPI Parallel Computing. `http://www.mpi.nd.edu/lam/`.

[22] Loic Prylli, Bernard Tourancheau, and Roland Westrelin. The design for a high performance MPI implementation on the Myrinet network. In *EuroPVM/MPI*, pages 223–230, 1999.

[23] Kenneth Yocum, Darrell Anderson, Jeffrey Chase, Syam Gadde, Andrew Gallatin, and Alvin Lebeck. Balancing DMA Latency and Bandwidth in a High-Speed Network Adapter. Technical Report TR-1997-20, Duke University, 1997.

[24] *The Virtual Interface Architecture*. `http://www.viarch.org/`.

[25] Philip Buonadonna, Andrew Geweke, and David Culler. An implementation and analysis of the virtual interface architecture. In *Supercomputing*, Nov 1998.

[26] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects V*, August 1997.

[27] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects V*, August 1997.