VIA over the CLAN Network

David Riddoch*
Laboratory for Communications Engineering
Department of Engineering, University of Cambridge, England

Steve Pope, Kieran Mansley
AT&T Laboratories-Cambridge,
24a Trumpington Street, Cambridge, England

{djr,slp,kjm}@uk.research.att.com

Abstract

The Virtual Interface Architecture is an industry standard for high performance networking in system-area networks, and the same model is proposed for Infiniband. Existing implementations suffer from high complexity, and scaling to higher bandwidths and large numbers of endpoints is likely to be problematic.

We present a novel implementation of VIA that consists of a thin software layer over the CLAN network. Performance of CLAN VIA is comparable with native solutions. The software implementation is highly flexible: we show that performance optimisations and extensions to the standard are easy to add.

The CLAN network has a very simple network model with very low overhead, that we believe scales well to very high bandwidths and large numbers of endpoints. These desirable properties are thus inherited by CLAN VIA.

1 Introduction

The Virtual Interface Architecture is an industry standard for high performance networking. Its scope is to describe an interface between the network hardware and software on the host, and the expectation is that vendors will implement new devices that conform to the specification.

VIA describes a *user-level* network interface: a network in which applications communicate directly with the network interface controller (NIC) without invoking the operating system. Applications may have a virtual memory mapping onto the NIC hardware, and the NIC is able to access application-level buffers directly, without intervention by the CPU. By avoiding system calls, complex protocol stacks and unnecessary copying of data (and the damaging effect these have on cache-performance), overhead and latency are significantly reduced.

Existing implementations of the Virtual Interface Architecture come in three flavours. Native implementations, such as Emulex's cLAN VIA[1], potentially offer the highest level of performance, but require custom NIC hardware. Cost of development is high and these technologies usually do not inter-operate directly with other networks.

The VIA programming interface can be provided on existing traditional networks by using software emulation. M-VIA[2] consists of a user-level library and loadable kernel module for Linux, and supports VIA over ethernet.[1] Network access is managed by the operating system, and so performance is less good than for native implementations, but this approach is cheap and highly flexible.

The third approach is to use an intelligent NIC. Intel's proof-of-concept[3] implementation and Berkeley VIA[4] both make use of Myricom's Myrinet[5] — a gigabit-class, programmable, user-level accessible NIC. As with custom hardware implementations, scalability is limited by the resources on the NIC.

Implementations that use specialised hardware can certainly achieve significantly higher performance

*David Riddoch is jointly funded by a Royal Commission for the Exhibition of 1851 Industrial Fellowship, and AT&T Laboratories-Cambridge.

[1] M-VIA also supports custom VIA hardware.

than those using software emulation on traditional network architectures, but this comes at a cost: The VIA model is complex, and a hardware implementation correspondingly so. To manage this, part of the work is typically done by a processor on the NIC, but it is not clear whether this approach will scale to network speeds of 10Gbps and above. Further, each application and the endpoints within it require resource in the NIC, and the number of active connections is limited by this resource. Supporting large numbers of endpoints necessarily becomes expensive.

The AT&T CLAN network is a high performance user-level network that exhibits very low per-endpoint resource requirements, and is highly scalable. It presents a low-level network interface that supports a wide range of communication styles efficiently. In this paper we present an implementation of VIA built as a software layer over CLAN, present some initial performance results, and compare it with an existing native hardware implementation.

2 Background

2.1 Virtual Interface Architecture

VIA is the result of an effort to standardise academic research into user-level networks. The design is most strongly influenced by the U-Net[6] project, but a wide variety of other user-level networks have been proposed[7, 8, 9]. The VIA network model has since been adopted for the Infiniband switched fabric interconnect, which has wide and powerful industry support, and hence is likely to be widely adopted.

The VIA Specification[10] defines the network interface's architectural model, including the division of functionality between hardware, application software and system software. The physical layer, on-wire format and application programmer's interface (API) are not defined. We believe this only serves to limit the implementation designer's choices, without even providing interoperability between implementations or application portability. However, a de facto standard API has emerged based on the specification and has been adopted by most existing implementations.

A schematic of the architectural model is given in Figure 1, and the software emulation approach in Figure 2. A Virtual Interface (VI) represents an application endpoint, and connects to another VI in a remote process. The VI has a send queue and a receive

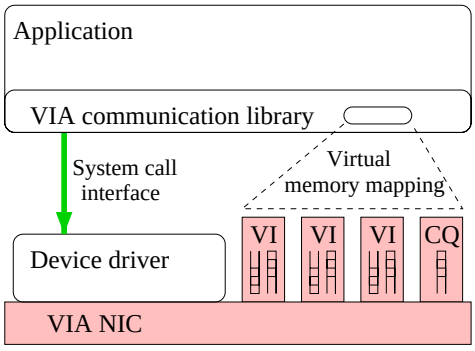

Figure 1: VIA Architectural Model.

queue, onto which the application posts descriptors that describe send and receive buffers. The specification also describes doorbells, which are needed for a native hardware implementation to inform the NIC that a descriptor has been placed on a work queue. These are not exposed in the API, and are not needed for all implementations.

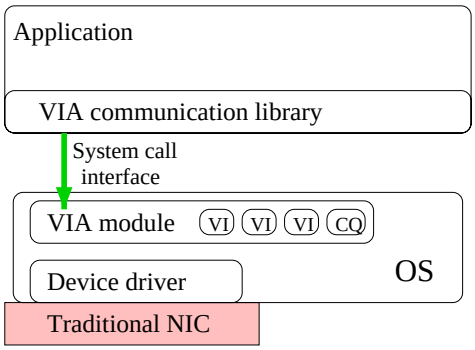

Figure 2: Emulation of VIA on traditional networks.

In the standard send/receive model the sending process specifies the location of the source data, and the receiving process specifies where the data should be delivered. The sending process posts a descriptor to the send queue by calling `VipPostSend()`, which returns immediately. The send operation completes asynchronously, and the application can poll for completion by calling `VipSendDone()`, or block waiting for completion with `VipSendWait()`.

Similarly the receiving process posts descriptors describing receive buffers to the receive queue using `VipPostRecv()`. These descriptors are completed when data is delivered into the buffers, and the application synchronises using `VipRecvDone()` and

`VipRecvWait()`. Applications are responsible for flow control, and if data arrives at a VI and no receive descriptors are available, the data is dropped.

Send and receive buffers must be registered before they can be used. This is necessary for user-level implementations to ensure that the pages of physical memory are pinned in memory, so they can be accessed directly by the NIC.

Three reliability levels are provided. With *unreliable delivery* data is delivered at most once, errors are detected, but packets may be lost. *Reliable delivery* and *reliable reception* guarantee that data is delivered exactly once or not at all. If data is dropped due to errors or buffer overrun, the connection is closed and an error indicated.

To support applications that manage multiple connections, notifications of completed requests from a number of VIs can be directed to a *completion queue*. The application can poll the completion queue (`VipCQDone()`), or block waiting for events (`VipCQWait()`). The returned value indicates which VI the descriptor completed on, and whether it was a send or receive event.

The VIA model also has support for Remote Direct Memory Access (RDMA), where the initiator specifies both the source and destination buffers. This is not discussed further in this paper.

2.2 The CLAN Network

As part of the CLAN[2] project at AT&T Laboratories-Cambridge we have developed a high performance user-level network for the local area. Key aims of the project include support for general purpose multiprogrammed distributed systems, and scalability to large numbers of applications and endpoints. The network is described in detail elsewhere[11], but an overview of the key features follows:

At the lowest level the communications model provided is non-coherent distributed shared memory (DSM). A portion of the virtual address space of an application is logically mapped over the network onto physical memory in another node. Data is transferred between applications by writing to the shared memory region using standard processor write cycles. A buffer in a remote node is represented by an RDMA cookie, the possession of which implies permission to access that buffer.

The NIC provides a number of additional resources to support efficient communication: Small out-of-band messages are used for connection set up and tear down, and a DMA engine reduces the overhead of data transfer for larger messages.

2.2.1 Synchronisation: Tripwires

On the receive path, data is transferred into an application's buffers asynchronously without the intervention of the CPU. This minimises overhead, but makes efficient synchronisation difficult, as the application receives no notification that a message has arrived.

The CLAN NIC provides a novel solution: the *tripwire*. This provides a means to synchronise with accesses to arbitrary locations in the shared memory. A tripwire is associated with a particular memory location, and *fires* when that location is read or written via the network. In response to a tripwire firing, the application may receive a notification of the event, and if blocked may be rescheduled.

Tripwire notifications from any number of tripwires can be directed into an *asynchronous event queue*[12], which can also receive DMA and out-of-band message events. This shared memory data structure allows events to be dequeued at user-level with very low overhead. The cost of event delivery is $O(1)$, and so scales well as the number of endpoints increases.

3 Implementation

3.1 The CLAN Network

Our prototype CLAN NICs are based on off-the-shelf parts, including an Altera FPGA, V3 PCI bridge (32 bit, 33 MHz) and HP's G-Link optical transceivers, with 1.5Gbit/s link speed. The tripwire synchronisation primitive is implemented by a content addressable memory, supporting 4096 tripwires in the current version. We have also built a five port worm-hole-routed switch, again using FPGAs, and a non-blocking crossbar switch fabric.

Application to application latency through the network is $2.6\mu s$, and maximum throughput $960Mbps$ (limited by the PCI bridge).

[2] Unrelated to the Emulex cLAN product range.

3.2 VIA over CLAN

We have implemented the VIA API as a user-space software library over CLAN, as illustrated in Figure 3. The functionality we have so far includes the send/receive data transfer model, polling and blocking modes of synchronisation, completion queues and all three reliability levels.

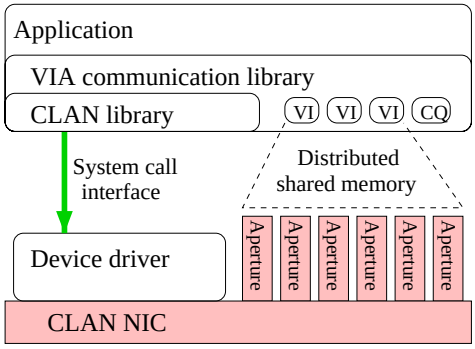

Figure 3: CLAN VIA Architecture.

Two properties of the CLAN network interface are key to the design of the VIA implementation that follows:

- Programmed I/O (PIO) writes have significantly lower overhead and latency than reads.
- The CLAN network is *send-directed*. That is, the sender has to know the location (RDMA cookie) of the receive buffers before it can transmit any data. A major advantage of this is that receive buffer overrun can never happen.

3.3 Data Transfer

The send-directed property of the CLAN network leads to two choices for transferring application data:

- Transfer the data to a known location in the receiving VI's address space, from where it can be copied into the application's receive buffers.
- Pass RDMA cookies for the application's receive buffers to the sender, so the data can be transferred directly.

The latter is clearly superior, since it avoids an unnecessary copy, but means that RDMA cookies have to be passed from the receiving VI to the sender. This is done at the time that a VIA receive descriptor is posted by an application: the receive descriptor is mapped to one or more RDMA cookies, which are transferred to the remote VI via the *cookie queue*.

3.3.1 Distributed Message Queues

The cookie queue is a lightweight fixed-size message queue, based on a distributed circular buffer. This is illustrated in Figure 4. A message is copied through the network into the next free slot in the remote buffer, indicated by the write pointer (`write_i`). The write pointer is then incremented modulo the size of the buffer, and the new value copied to the remote address-space (`lazy_write_i`).

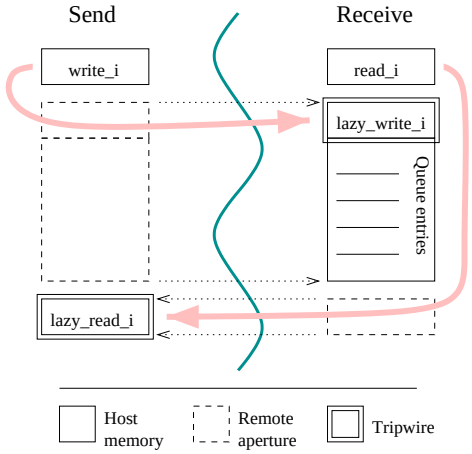

Figure 4: A Distributed Message Queue

The receiver compares its lazy copy of the write pointer with the read pointer to determine whether or not the queue is empty. Messages are dequeued by reading them from the buffer, then incrementing the read pointer, and copying its new value to the sender. Transferring small messages consists of just a few processor write instructions, and hence has very low overhead.

The cookie queue resides in a small buffer in each VI. The RDMAs cookies for these buffers are passed to the other VI as part of the connection set up messages, thus allowing them to create a memory mapping. The cookie queue is used to transfer receive buffer descriptors from the receiver to the sender. Messages are arranged in groups, the first message giving the amount of buffer space given by the descriptor, and the number of segments. The messages that follow

represent segments, each mapped to RDMA cookies.

A similar queue, the *transfer queue*, is used to pass meta-data in the other direction: from sender to receiver. Each message corresponds to a completed VIA send descriptor, and includes the amount of data sent and immediate data.[3] Alternatively a message may indicate an error condition.

3.3.2 Receiving

Basic data transfer is illustrated in Figure 5. The receiving application posts a receive descriptor to a VI using `VipPostRecv()`. The segments within the descriptor are mapped to RDMA cookies, and passed to the remote VI via the cookie queue, and control is returned to the application immediately.

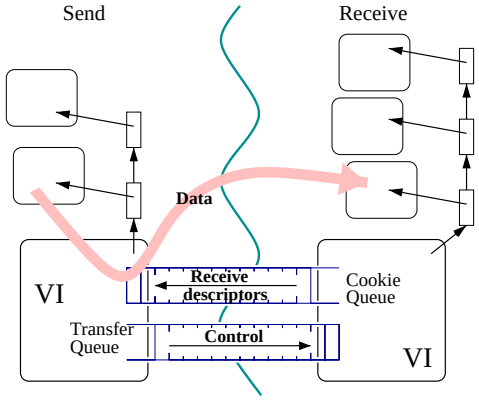

Figure 5: Basic VIA data transfer.

3.3.3 Sending

Some time after the receive descriptor is posted, the sending application posts a send descriptor. The cookie queue is interrogated to find the corresponding receive descriptor, and one or more DMA requests are submitted to transfer the application data directly from the send buffers to the receive buffers. A message is placed in the transfer queue to pass information in the send descriptor to the receiver, and control is returned to the application.

Data transfer itself happens asynchronously when the DMA requests reach the front of the DMA queue. However, the message placed in the transfer queue is used to indicate that the application data transfer has completed, and so must not arrive until that is true. To achieve this efficiently, the body of the transfer queue message is written using PIO at the time the send descriptor is processed, but the transfer queue's write pointer update is done by an asynchronous DMA request after the application data transfer has completed.

[3] Four bytes of application data that are passed from a send descriptor to the receive descriptor.

3.3.4 Buffer overrun

If the cookie queue is found to be empty when a send descriptor is posted, then the receive buffers have been overrun, and VIA specifies that the data should be dropped. In this case the send descriptor is completed without any data being transmitted onto the network. Whether or not an error is indicated on either side depends on the reliability level of the connection. It is a general advantage of send-directed communication that in the case of receive buffer overrun, no data is transferred, and so the network is not loaded unnecessarily with data that cannot be delivered.

3.4 Synchronisation

Send synchronisation is trivial, and merely involves determining whether the DMA requests associated with a descriptor have completed. This information is provided by the CLAN DMA interface.

Receive synchronisation is more difficult. As described so far, this implementation of VIA will work on any PRAM[13] consistent shared memory implementation. However, the completion of an incoming message is signalled by a message arriving in the transfer queue, which is simply an asynchronous change to data in memory (specifically the message queue's write pointer). An application can detect this change by inspecting the relevant memory locations, which is enough to support the non-blocking `VipRecvDone()` method. To support blocking receives (`VipRecvWait()`) a hook into the process scheduling mechanism of the operating system is needed.

Our solution is the CLAN tripwire. As shown in Figure 4, a tripwire is associated with the message queue's write pointer, and fires whenever a message is placed in the queue. When the tripwire fires the application receives a notification, and a thread can be rescheduled if currently blocked in `VipRecvWait()`.

3.5 Programmed I/O

As an optimisation, it is possible to configure a VI to transfer application data using PIO rather than DMA. Although the CPU now has to do work to transfer data, this has lower overhead and latency than DMA for small messages because of the DMA set up cost. Using PIO requires a virtual memory mapping onto the remote memory region, which is relatively expensive to set up. A cache of such mappings is thus maintained, with least-recently-used for eviction. Note that these mappings correspond to regions of registered memory, not individual receive buffers, and a small cache of mappings can cover a much larger number of buffers.

A further benefit of PIO for small messages is that data transfer happens during the application's scheduling time slice, rather than when the NIC chooses to schedule the DMA transfer. Applications that have delay sensitive traffic can use PIO to ensure timely delivery of messages, even when competing with large transfers. Thus the operating system's process scheduling policy also manages network access. QoS support in the network would also be desirable, but is not yet supported.

3.6 Completion Queues

The VIA completion queue is implemented using the CLAN asynchronous event queue. When a VI's receive queue is associated with a completion queue, the tripwire associated with its transfer queue is associated with the event queue. To associate a VI's send queue with a completion queue, DMA completion events are directed to the event queue.

3.7 Extensions to VIA

The VIA API is designed to provide the facilities needed to develop distributed applications. However, it leaves out a number of desirable features, which are thus implemented over and again by developers above the level of VIA. The flexibility of our software design makes it simple to implement extensions to VIA, and two examples are given here:

3.7.1 Flow Control

As discussed in Section 3.3.4, the sending application knows whether there are buffers available in the receiver at the time the send descriptor is posted. If it finds that no receive buffers are currently available, the default behaviour is to drop the data, as prescribed by the VIA standard. As an extension, the application can configure a VI to defer sending until receive buffers are posted. This simple modification adds flow control to VIA, and no change is needed to the standard API.

3.7.2 Request Splitting at Receiver

This extension permits an incoming message that is larger than the space available in the first receive descriptor, to be split over a number of receive descriptors. Applications typically post a number of receive buffers on each endpoint, in order to allow streaming, and each has to be at least as large as the maximum message size. Where message sizes vary substantially, this extension saves buffer space by allowing the receiver to post smaller buffers. A small modification to the standard API is needed to indicate that the message continues in the following descriptor(s).

3.8 Protection

Our prototype NIC hardware currently lacks full protection on the receive path, and having given a remote process access to a buffer it is not possible to revoke access. This means that a faulty or malicious node that goes in below the level of VIA can overwrite data in an application's receive buffers after the receive descriptor has completed.

Proper protection will be available in a future revision of the NIC. Implementing strict protection will have a small performance impact, and so will likely be offered as a configuration option, since it is not always needed in a trusted network environment.

4 Performance

In this section we present a number of synthetic benchmark results in order to evaluate the performance potential of this implementation of VIA, and compare it with an existing native hardware implementation: the Emulex cLAN 1000.

The test nodes were a pair of 650 MHz Pentium III systems running an unmodified Linux 2.2 kernel. Each machine contained an AT&T MkIII CLAN NIC, and an Emulex cLAN 1000. The Emulex cLAN 1000 is based around a single chip design, in a PCI card (64

bit, 33 MHz) with 1.25Gbit/s link speed. The results given below were obtained using identical benchmark software on each system. Fine grained timing was performed using the processors' free running cycle counters. Because we did not have an Emulex switch, both networks were arranged in back-to-back configuration.

Because many distributed applications require reliable communications, the reliability level used in these tests was *reliable delivery*. To prevent receive buffer overrun, the test applications used credit-based flow control. For CLAN VIA we present separate results for PIO and DMA data transfer for clarity (although by default it is able to switch dynamically between the two).

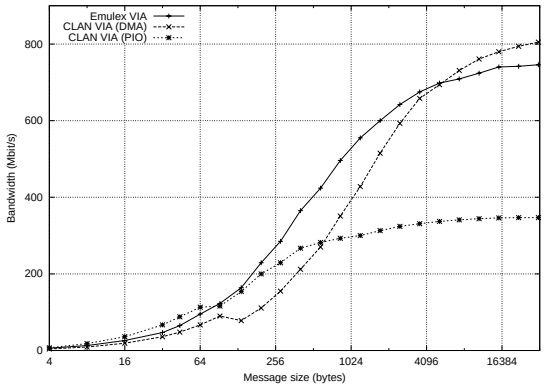

Figure 6: Bandwidth vs. message size.

4.1 Latency

The latency for small messages was measured by timing a large number of round-trips and halving the result. This value includes the time taken to post a send descriptor, process that descriptor, transfer the data and synchronise with completion on the receive side. The results are given in Table 1.

Bytes transferred	CLAN (DMA)	CLAN (PIO)	Emulex cLAN 1000
0	4.4	3.4	6.6
4	6.4	4.6	7.5
40	6.8	4.7	9.5

Table 1: Half round-trip latency (μs) for small messages.

The small message latency for CLAN VIA (PIO) is the lowest by some margin. This reflects the very low overhead of PIO for small messages on the CLAN network. For comparison, M-VIA report latency over gigabit ethernet of $19\mu s$[14], and Berkeley VIA report $23\mu s$[4].

4.2 Bandwidth

To measure bandwidth a large number of messages were sent across the network, with flow control credits passing in the opposite direction. Each measurement was made with a $10MB$ transfer, and using four buffers at the receiver (we found that this was the point at which adding additional receive buffers gave little improvement). The results are given in Figure 6.

CLAN VIA (PIO) gives the best performance for very small messages, due to its very low overhead. However, throughput in this mode is limited by the pattern of traffic generated on the PCI bus to about 370Mbit/s. Comparing Emulex VIA with CLAN VIA (DMA), Emulex does significantly better except for large messages. This is due to the high overhead and turn around time of CLAN's DMA implementation. The dip at 128 bytes occurs when the DMA scheduler switches from PIO to DMA mode.

CLAN's DMA engine uses a single DMA controller which is integrated with the PCI bridge, and only accepts a single request at a time. An interrupt is generated at the end each request, and the interrupt service routine then schedules the next. The interrupt processing overhead and high turn-around time leads to poor performance for small and medium sized messages. We will be improving on this with our own DMA engine in the next revision of the CLAN NIC, which should substantially improve performance.

4.3 Completion Queues

The final benchmark aims to test the performance and scalability of completion queues. A server process accepts new connections, and associates each VI with a completion queue. Incoming message events are taken from the completion queue, and the message echoed to the sender.

Because the network configuration is limited to two nodes, multiple clients are simulated by a single client process on the other node. Small messages are sent to the server on random connections as quickly as possible, but with only one outstanding request on each

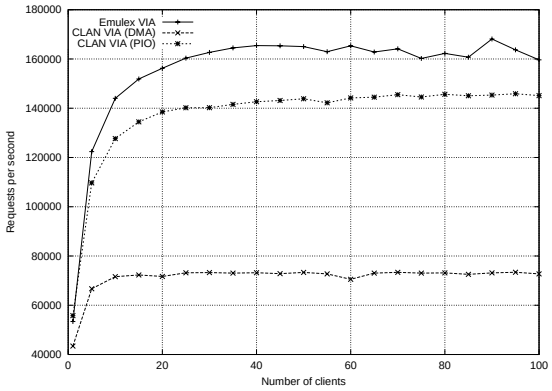

Figure 7: Throughput vs. offered load.

connection. The client application knows which connections it has sent messages out on, and so does less work than the server, ensuring that performance is limited by the server. The results are shown in Figure 7.

Although both implementations show good performance with no degradation under load, CLAN VIA has lower throughput. In the case of CLAN VIA (PIO), this is because the CLAN event queue is driven by an interrupt per message, and hence has high overhead. CLAN VIA (DMA) also suffers from this problem, and the poor performance of the DMA engine for small messages makes the overall result even worse. A revision of the NIC that delivers CLAN events to the application at user-level without interrupts is currently under development, and we project will bring performance to at least the level of Emulex VIA.

We have found that by tailoring the communication abstraction to the application, and implementing it directly over the CLAN network, we get significantly better performance for a range of applications. For example, using a lightweight message-based protocol, a server similar to the one presented above can process over 400,000 requests per second. These results are presented in [11].

4.4 Analysis

The result that our software emulation of VIA over CLAN out-performs a dedicated hardware solution in some tests is surprising. In principle it should always be possible to design a dedicated solution that gives the best possible performance for a particular API. We conclude that there is room for improvement in the implementation of Emulex VIA.

We suspect that the complexity of the protocols has lead to some of the work being done by an embedded processor on the Emulex NIC. In contrast, the CLAN NIC is implemented entirely as hardware state machines, which we believe has better potential to scale to higher bandwidths. Those parts of the protocol we leave to software run on the host CPU, and benefit from performance increases according to Moore's Law.

5 Conclusions

We have shown how it is possible to implement the VIA API as a thin software layer over distributed shared memory with tripwires for synchronisation. Our implementation, using the CLAN network, exhibits performance that is comparable with, and in some cases exceeds, an existing native hardware solution. Where the performance currently lags that of the hardware implementation, we believe it is due to limitations of the CLAN prototype hardware rather than an intrinsic limitation of the CLAN network model.

The software approach is highly flexible, and has allowed us to add optimisations and extensions. The use of programmed I/O for small messages significantly reduces overhead and latency. With the send-directed property of the CLAN network, flow control comes naturally with almost no additional performance penalty.

We have argued elsewhere that the CLAN network presents a very powerful low-level model, on which a wide range of communications paradigms have been implemented efficiently. These include TCP/IP, MPI, CORBA and NFS. Here we have shown that it also supports the VIA interface, and because they are all implemented solely as software on the host, they can all be used together on the same network. This is in contrast with programmable NICs, in which the network is programmed to support a single protocol.

Acknowledgements

The authors would like to thank all of the members of the Laboratory for Communications Engineering, and AT&T Laboratories-Cambridge, particularly members of the CLAN team.

References

[1] Emulex cLAN. http://wwwip.emulex.com/ip/products/clan1000.html.

[2] M-VIA Project. http://www.nersc.gov/research/FTG/via/.

[3] Frank Berry, Ellen Deleganes, and Anne Marie Merritt. The Virtual Interface Architecture Proof-of-Concept Performance Results. Technical report, Intel Corporation.

[4] Philip Buonadonna. An Implementation and Analysis of the Virtual Interface Architecture. Master's thesis, University of California, Berkeley, May 1999.

[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] 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.

[7] 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.

[8] Gary Delp, Adarshpal Sethi, and David Farber. An Analysis of Memnet: An Experiment in High-Speed Shared-Memory Local Networking. In *ACM Symposium on Communications Architectures and Protocols*, 1988.

[9] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *2nd Symposium on Operating Systems Design and Implementation*, pages 245–259, October 1996.

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

[11] David Riddoch, Steve Pope, Derek Roberts, Glenford Mapp, David Clarke, David Ingram, Kieran Mansley, and Andy Hopper. Tripwire: A Synchronisation Primitive for Virtual Memory Mapped Communication. *Journal of Interconnection Networks, JOIN*, 2(3):345–364, September 2001.

[12] David Riddoch and Steve Pope. A Low Overhead Application/Device-driver Interface for User-level Networking. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001.

[13] Richard Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[14] M-VIA Performance. http://www.nersc.gov/research/FTG/via/faq.html#q14.