Future Generation Computer Systems 18 (2002) 421–433

FGCS
Future Generation Computer Systems

www.elsevier.com/locate/future

VI architecture communication features and performance on the Giganet cluster LAN

Hermann Hellwagner*, Matthias Ohlenroth

Department of Information Technology, University Klagenfurt, Universitätsstraße 65-67, A-9020 Klagenfurt, Austria

Abstract

The virtual interface (VI) architecture standard was developed to satisfy the need for a high throughput, low latency communication system required for cluster computing. VI architecture aims to close the performance gap between the bandwidths and latencies provided by the communication hardware and visible to the application, respectively, by minimizing the software overhead on the critical path of the communication. This paper presents the results of a performance study of one VI architecture hardware implementation, the Giganet cLAN (cluster LAN). The focus of the study is to assess and compare the performance of different VI architecture data transfer modes and specific features that are available to higher-level communication software like MPI in order to aid the implementor to decide which VI architecture options to employ for various communication scenarios. Examples of such options include the use of send/receive vs. RDMA data transfers, polling vs. blocking to check completion of communication operations, multiple VIs, completion queues and scatter capabilities of VI architecture. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: VI architecture; Giganet cLAN; Performance evaluation; System area network

1. Introduction

The performance of parallel applications running on clusters depends on the implementation of the nodes and the LAN or SAN (system area network) that acts as the communication system. Conventional communication systems like Fast Ethernet and legacy TCP/IP protocol stacks are widely used, especially for systems with limited budgets. But due to high software processing overhead (e.g., [13]), their communication performance is not sufficient for all application types. Applications with high communication frequency and small messages may suffer from the high latency of such solutions.

For a modern SAN with several GBit/s hardware throughput, it is crucial to eliminate or minimize this software overhead along the critical path of send/receive communication operations such that application-level latency be improved roughly proportional to the throughput.

Appropriate techniques to reduce software overhead have been pioneered by fast communication systems like Active Messages [5], Fast Messages [12] and U-Net [6].

Such techniques, as briefly reviewed in Section 2, form the basis on which the VI architecture [2,4] has been developed. VI architecture is not intended for application programmers; instead, it provides a set of communication concepts and operations suitable to implement efficient higher level

* Corresponding author.
E-mail addresses: hermann.hellwagner@itec.uni-klu.ac.at (H. Hellwagner), matthias.ohlenroth@itec.uni-klu.ac.at (M. Ohlenroth).

communication libraries, for instance MPI or a socket interface.

There are a number of variants and specific features of the VI architecture operations that the programmer of communication libraries has to judiciously choose from in order to achieve high communication performance in various scenarios. These choices include:

- use of send/receive vs. RDMA (remote direct memory access) data transfer modes;
- use of polling vs. blocking mechanisms to test the completion status of communication operations;
- the number of virtual interfaces (VIs) to use for a given connection;
- use of completion queues (CQs) to simplify testing completion of communication operations;
- use of advanced features of the VI architecture like scatter facilities.

Often, the performance implications of design choices like these are not obvious. Therefore, we have performed a series of experiments in order to gain some insight into the performance behavior of these features. The platforms used were PC clusters configured with the Giganet cLAN which implements VI architecture in hardware. Both low-level microbenchmarks and a higher-level memory system and communication performance benchmark were employed.

The results of these experiments as well as our conclusions are presented in this paper. The paper is organized as follows: Section 2 briefly reviews the important features of the VI architecture standard and covers in more detail the communication variants and features under investigation. Section 3 introduces the cluster platforms for the experiments and the communication benchmarks used and presents the performance results. Section 4 addresses related research. Our conclusions are given in Section 5.

2. The VI architecture

2.1. Rationale

In conventional cluster communication systems based on the TCP/IP protocol stack, the network is accessed by the application (on a send operation) via a message passing library which calls the socket interface of the operating system (OS). During the system call, the kernel-level TCP/IP protocol processing code copies message data and adds headers at different protocol layers and forwards the final packet to the network interface controller (NIC) driver software. This driver arranges the packet according to the requirements of the NIC's DMA engine and programs the NIC to inject the packet into the network. On the receiving side, the NIC writes the packet into a kernel-level buffer and informs the device driver (via an interrupt) that a new packet has arrived. This packet is then processed by the kernel-level protocol stack. The receiving application's message passing library calls the OS (via a socket) to copy the message into an application-level buffer.

Hence, expensive protocol processing and data copying may occur at different software layers inside the application and inside the OS. In addition, context switches and interrupt processing have to be performed on each communication request. These factors contribute to significant software processing overhead, leading to a so-called *cluster communication gap*, i.e., a discrepancy between the communication performance provided by the network hardware and visible to the application, respectively.

Previous research, e.g., within the Active Messages [5], Fast Messages [12] and U-Net [6] projects, has devised and implemented techniques to minimize the software overhead on the critical communication path and has thus been able to reduce the application-level communication latency by several factors.

Key techniques to minimize software processing overhead that have been developed by this research, can be briefly summarized as follows:

- Connection management (most importantly, connection setup and teardown) and communication operations proper are separated such that the OS needs to be only involved in the former activities.
- User processes are given direct (user-level) access to the NIC such that the OS may be bypassed on the critical communication path. This requires a well-defined interface to the NIC, usually in the form of communication endpoints and functions to access them or descriptors which specify communication operations to be performed.
- User memory regions are registered with the NIC as message buffers such that data can be directly read from or written to them during communication

operations. Memory registration usually includes negotiation of buffer identifiers (memory handles) between the user process and the NIC, setting up address translation and protection tables, and pinning of the message buffers in physical memory.

- By virtue of memory registration, address translation and protection checking can be cheaply done by the NIC on communication requests without the need to involve the OS.
- Moreover, data transfers involved in communication operations can also be effected directly to and from registered user message buffers by the NIC, without the need to do extra data copies, which leads to the so-called zero-copy protocols (in case receive buffers are available).

2.2. Overview

Building on these principles, the VI architecture standard [2,4] defines a set of concepts and primitives that allow an application to perform protected communication operations directly from the user level without involving the OS. VI architecture provides point-to-point connections, the endpoints of which are implemented as VIs. Each user process is provided with the illusion of one or more private interfaces to the NIC. A VI consists of a send queue, a receive queue and a notification mechanism called doorbell.

The VI architecture model comprises a VI consumer part and a VI provider part (Fig. 1). The *VI consumer* consists of an application and a communication library like MPI that uses VI services via a lower level library (VI user agent in the figure). The *VI provider* consists of the NIC hardware and a kernel-level driver component (VI kernel agent). This component is responsible for handling protection related functions like opening and closing connections, for registering the application's memory regions (e.g., message buffers) with the NIC for communication purposes, and for address mapping of these message buffers. Communication requests like send, receive, and RDMA bypass the OS interface and interact directly with the NIC via a VI.

Initiating a communication operation requires the VI consumer to prepare a *descriptor* of the work to be done, post it on the respective queue of a VI, and inform the NIC about the new communication request using the doorbell mechanism. The NIC will execute the communication request according to the information provided in the descriptor. After communication, the NIC will record completion status information in the status field of the descriptor.

Communication occurs to and from user memory regions which the VI consumer has to register with the NIC and which are subsequently identified by a pair (virtual address, memory handle). It is important that, at the beginning of a communication activity, the

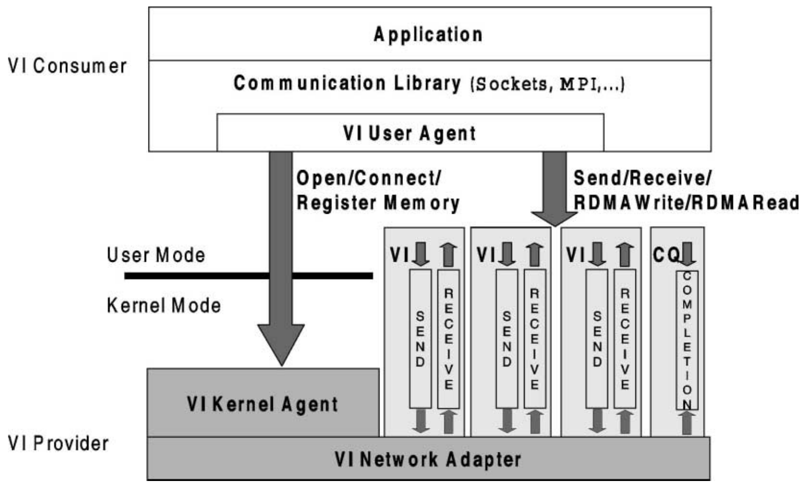

Fig. 1. VI architecture model.

Table 1
Ping-pong benchmark codes (simplified)

	Send/receive polling	Send/receive blocking	RDMA write polling	RDMA write blocking
Requester	while (loop--) {	while (loop--) {	while (loop--) {	while (loop--) {
	VipPostSend();	VipPostSend();	*flag = 1;	*flag = 1;
	VipPostRecv();	VipPostRecv();	VipPostSend();	VipPostSend();
	while(!VipSendDone());	VipSendWait();	while(!VipSendDone());	VipSendWait();
	while(!VipRecvDone());	VipRecvWait();	while(*flag);	while(*flag);
	}	}	}	}
Responder	while (loop--) {	while (loop--) {	while (loop--) {	while (loop--) {
	while(!VipRecvDone());	VipRecvWait();	while (!*flag);	while (!*flag);
	VipPostSend();	VipPostSend();	*flag = 0;	*flag = 0;
	while(!VipSendDone());	VipSendWait();	VipPostSend();	VipPostSend();
	VipPostRecv();	VipPostRecv();	while(!VipSendDone());	VipSendWait();
	}	}	}	}

receive descriptor be posted prior to the send descriptor such that a zero-copy protocol can be executed.

2.3. Communication options

The VI architecture standard offers a number of options to execute communication operations. First, an asynchronous *send/receive* model is provided. Initiating a communication and testing its completion status are separate actions as described above. The second model is called remote direct memory access (RDMA) which denotes one-sided communication and is asynchronous as well. For example, an RDMA write operation transfers the message directly into the receiver's message buffer without notifying the receiver about the message transfer. The memory region (virtual address and memory handle) to be written must have been disclosed to the writer prior to this operation.

A VI consumer (application) has two options to check the completion status of communication operations, namely via non-blocking and blocking calls. These calls are typically used by an application to *poll* on the completion of a communication (on user level), or to *wait and be interrupted* after its completion (which involves the OS), respectively. These modes are illustrated by the benchmark pseudo-codes in Table 1. Clearly, the blocking mode is more expensive, but allows the CPU to perform useful activity in lieu of busy waiting.

An application may own multiple VIs and many communication operations may be outstanding on each VI. In such a case, the application may need to spend considerable time on checking its VIs for completed communication activities. To avoid this, VI architecture specifies the concept of completion queues (CQs). CQs collect completion notifications for multiple VIs. They enable the VI consumer to check completion status in a single location and then directly access the completed descriptor in a VI.

Another feature that appears to be attractive to be used by communication software on top of VI architecture is its *scatter capability*. A scatter operation can be effected in send/receive mode by specifying, for instance: (1) a single data segment in the send descriptor describing the length, memory handle, and virtual address of a single large send buffer; and (2) multiple data segments in the receive descriptor holding the lengths, memory handles, and virtual addresses of multiple smaller receive buffers. The NIC at the receiving end of the connection then autonomously scatters the data to the receive buffers, given that they provide sufficient aggregate space. This behavior may be more convenient for the programmer than to explicitly have a receiver process store the data into multiple destination locations. Furthermore, it is a zero-copy protocol, whereas scattering data under program control represents an extra copy step.

Vice versa, a zero-copy gather operation can be effected using VI architecture features.

3. Performance evaluation

For implementation of higher-level communication libraries, it may be important to be aware of the performance implications of communication options like those described in the previous subsection. With this knowledge, he/she can select the communication modes and features that promise to provide the best performance, or he/she can design the library to do so depending on the actual communication scenario.

We therefore performed a study to explore the performance behavior of the communication options outlined above.

3.1. Platforms and benchmarks

The performance experiments were run on two cluster platforms. One platform runs the Windows NT 4.0 (SP 6) OS, the second cluster runs Linux (kernel 2.2.16). Both clusters consist of two machines each equipped with an Intel Pentium® III 450 MHz processor and the Intel 440BX chipset. The machines are interconnected by the Giganet cLAN (cluster LAN) network, more precisely by directly connected GNN1000 cluster adapters which implement VI architecture in hardware [7]. Peak performance of this cluster interconnect is 1.25 GBit/s. We used the Windows NT driver revision 4.0.0 and the Linux driver revision 1.1.1.

Latency and bandwidth figures for the different platforms and communication options were obtained using well-known microbenchmarks. In addition, in order to test more realistic communication patterns, we adopted one of the memory system and communication benchmarks proposed by Stricker and co-workers [9,16]. Specifically, we report the results for copy and communication operations with strided stores of the data into the target buffer, i.e., a scatter operation which occurs as a part of a matrix transposition, for instance.

3.2. Latency

Latency was measured using ping-pong tests (based on `viptest`) with four-byte packets. We evaluated the send/receive model with polling and blocking synchronization and the RDMA write model without immediate data. Synchronization in this context denotes how a VI consumer detects, or is notified of, the completion of a communication operation (send or receive). The ping-pong test in the RDMA write model is realized by having the receiving party check a flag in its local memory that is set by the sending party upon completion of the RDMA write. Subsequently, the parties change their roles. Receive descriptor processing is not involved, therefore.

Table 2
Round-trip latencies for send/receive and RDMA write

Communication and synchronization model	Windows NT (μs)	Linux (μs)
Send/receive, polling	14.5	14.8
Send/receive, blocking	45.0	33.0
RDMA write, polling	14.8	15.0
RDMA write, blocking	19.2	15.6

Table 1 depicts the ping-pong benchmark codes for the communication modes under investigation. Only the cases with identical synchronization styles at both the sending and receiving sides were considered.

Table 2 summarizes the round-trip latency results. The polling mechanism is significantly faster than the blocking mechanism on both platforms. This is due to the software overhead for system calls, context switching and interrupt processing associated with the blocking synchronization style. The numbers indicate that this software overhead is about 15 μs for the Windows NT and less than 10 μs for the Linux machine. The RDMA write with blocking synchronization is surprisingly fast. An analysis (under Linux) of the system calls performed by the VI library revealed that not all of the wait calls (`ioctl`s) do become effective, on average yielding a low overhead value per wait operation.

We instrumented all eight benchmark loops shown in Table 1 using the RDTSC (read time stamp counter) assembler instruction. Its overhead is approx. 41 clock cycles (about 90 ns). The tests ran under the Linux OS. Figs. 2 and 3 exemplarily show detailed timing results for the RDMA write communication loops. Posting descriptors (`VipPostSend()` and `VipPostRecv()`) are low-overhead operations (770–900 ns). Testing send descriptor completion status (`VipSendDone()`) takes 3.0–4.1 μs. It is interesting to see that the blocking counterpart `VipSendWait()` is very expensive for RDMA write–style communication (11.7 μs), compared to 4.9 μs for send/receive style communication (Figs. 2 and 3). The diagrams show the steady-state unidirectional latency (4.8 μs) that we measured for the hardware transfer time.

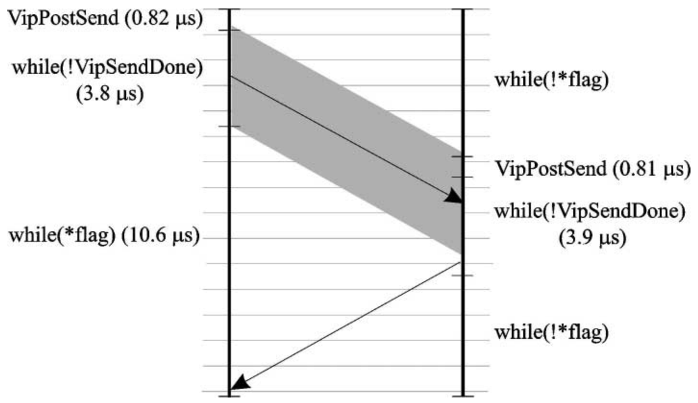

Fig. 2. Timing of RDMA write–style communication using polling synchronization.

3.3. Throughput

We measured one-way throughput by having the sending process send out data at the maximum possible rate and having the receiver finally respond by a single confirmation message to complete the test. Furthermore, we measured two-way throughput using the ping-pong code. Again, results were obtained for both send/receive and RDMA write communication models.

Figs. 4 and 5 summarize the results obtained for the clusters running the Windows NT and Linux OSs, respectively. The performance under both OSs is comparable. The polling synchronization method is faster than the blocking method on both clusters. Traditional send/receive style communication can achieve approx. 830 MBit/s. Depending on the synchronization method, the RDMA write communication method is 45–100 MBit/s faster than send/receive style communication. RDMA write can achieve approx. 869 MBit/s throughput.

For comparison purposes, one-way throughput results for the TCP/IP protocol over the Giganet cLAN (using the socket interface) are shown in Fig. 4. The performance does not exceed approx. 133 MBit/s on the Windows NT cluster.

3.4. Strided copy performance

Strided copy tests are based on the memory system performance benchmarks proposed by Stricker and Gross [16]; these benchmarks have also been adapted

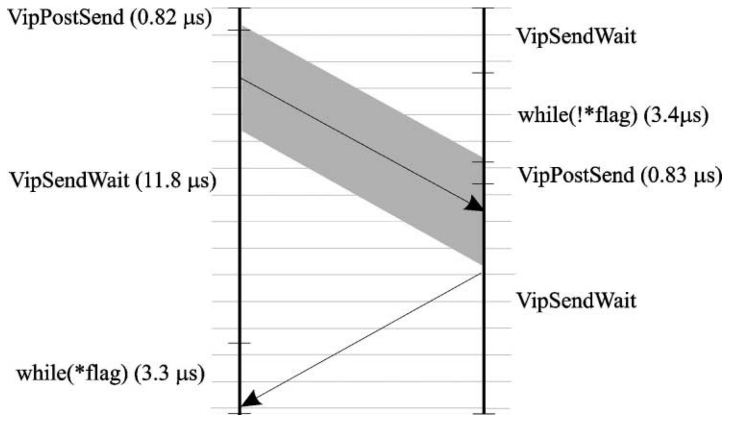

Fig. 3. Timing of RDMA write–style communication using blocking synchronization.

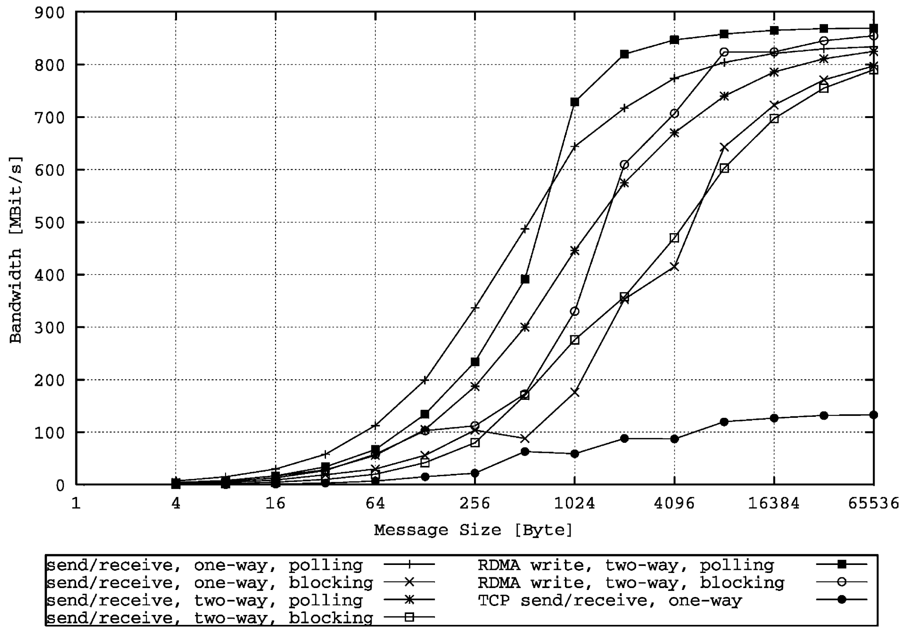

Fig. 4. Throughput on the Windows NT cluster.

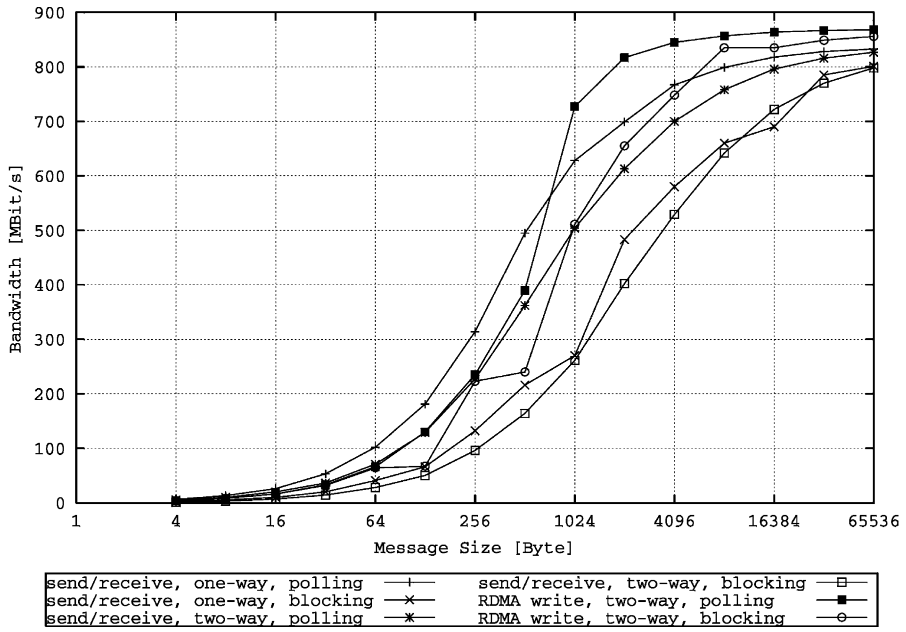

Fig. 5. Throughput on the Linux cluster.

to test distributed shared memory systems and, thus, their underlying interconnects, e.g., scalable coherent interface (SCI) and Cray T3D [9]. In the original strided copy benchmark, contiguously stored blocks of double-precision floating point (FP) numbers are spread over the target buffer, which may be located on a different node. The distance between two buffer entries depends on the organization of the buffer and is called the *stride*. Stride 1 simulates copying memory regions contiguously. Performance values for strides greater than 1 are influenced by the memory and cache organization and the properties of the communication system.

The strided copy algorithm was implemented in three different ways. First, the performance of the memory system was evaluated using a local (single-machine) version of the algorithm. A second form combines this test with a message passing step: the memory region is transferred over the network as a contiguous block and then stored away (scattered) locally on the receiving node with the required stride. A third version combines both communication and strided copy into one communication request using the capability of the VI architecture to distribute received data over a list of memory blocks (i.e., the scatter capability): for each double-precision FP number to be received (as one element in a block that has been sent contiguously), a separate target address is specified in the receive descriptor; the NIC performs the strided store as part of the receive operation, avoiding the extra copy step required in the second implementation.

Fig. 6 compares the throughput achieved for different strides on both clusters. All tests handle blocks of approx. 8 KB. (Buffer usage depends on the stride size.) Only the local strided copy version is influenced by the stride value. This version achieves a peak throughput in excess of 3 GBit/s, indicating that the local memory system is not a bottleneck. The second form of strided store achieves a peak throughput of 585 MBit/s on the Windows NT and 653 MBit/s on the Linux platform. In contrast, bandwidth for communication integrating the strided store (version 3) is nearly constant at the low rate of 22–26 MBit/s and is determined by the latency of the communication system; in this case, data segment processing by the NIC appears to be the bottleneck because each store of a double-precision FP value is encoded into one data segment of a receive descriptor. Clearly, an eight-byte value associated with a single data segment is too fine-grained to be processed efficiently by the NIC.

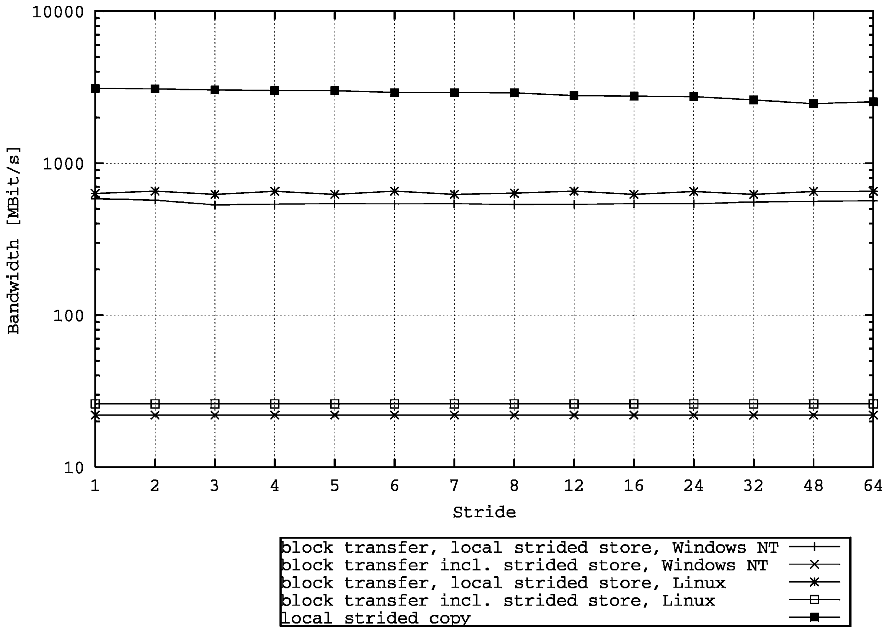

Fig. 6. Block transfer with strided stores.

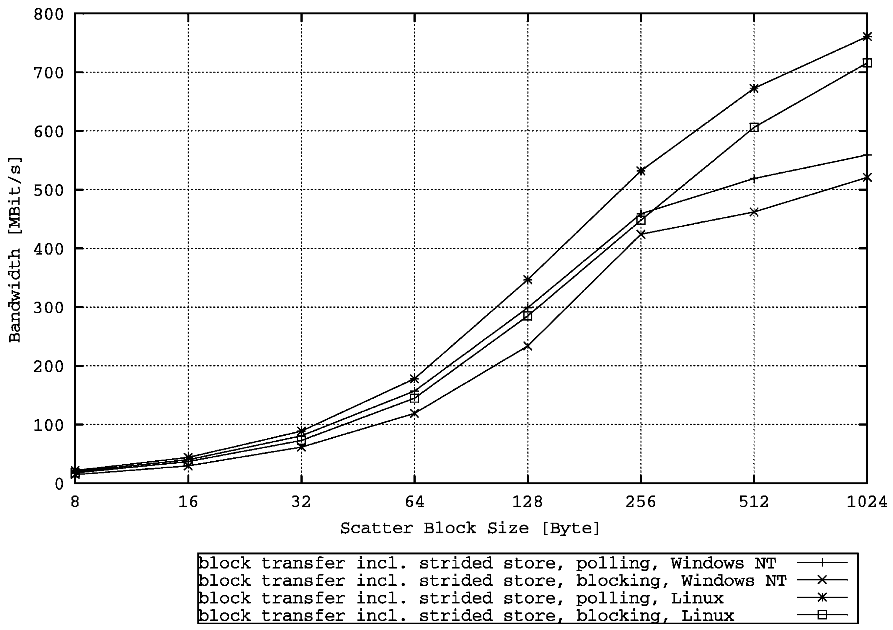

Fig. 7. Automatic scattering with varying block sizes.

It is therefore interesting to see if and how performance can improve when scattering applies to coarser-grained blocks of data. Fig. 7 depicts the results of such an experiment, where the size of the individual blocks to be scattered by the NIC increases from 8 to 1024 bytes and 32 data segments are used in the receive descriptor; i.e., the overall amount of data transferred increases from 256 bytes to 32 KB.

Increasing the data block size for scattering rapidly increases throughput, up to a value of about 560 MBit/s (Windows NT) and 731 MBit/s (Linux) under a polling synchronization scheme; the results for the blocking mode are slightly lower. Varying the stride size did not significantly impact performance. A comparison with Fig. 4 shows that data scattering at the receiving end costs up to one-third of the maximum throughput even for large scatter blocks. Moreover, the scatter support provided by the VI NIC (third algorithm) is inferior to explicit scattering under program control (second algorithm); cf. Fig. 6. Therefore, from a performance point of view, the use of the scattering facilities of the Giganet VI architecture implementation cannot be recommended.

3.5. Completion queues

Completion queues record completed descriptors in a single location and allow to directly branch to VI work queues associated with completed communication requests. This simplifies communication software and should lower the overhead of checking the completion status of communication operations when multiple VIs are in use. To analyze the efficacy of this concept, we compared the use of CQs to a hand-crafted round robin (RR) descriptor processing strategy. To ensure fairness, the RR code skips empty VI send queues. The test program creates N connections (on N VIs) between two communicating processes and distributes communication requests randomly among them.

Figs. 8–10 present the results obtained using a total of one, two and five outstanding messages, respectively, on Linux machines. We do not present results for the Windows NT platform because creating multiple connections was unstable.

The CQ processing overhead is independent of the number of VIs. This overhead is approx. 3% compared to the RR strategy with one VI and one

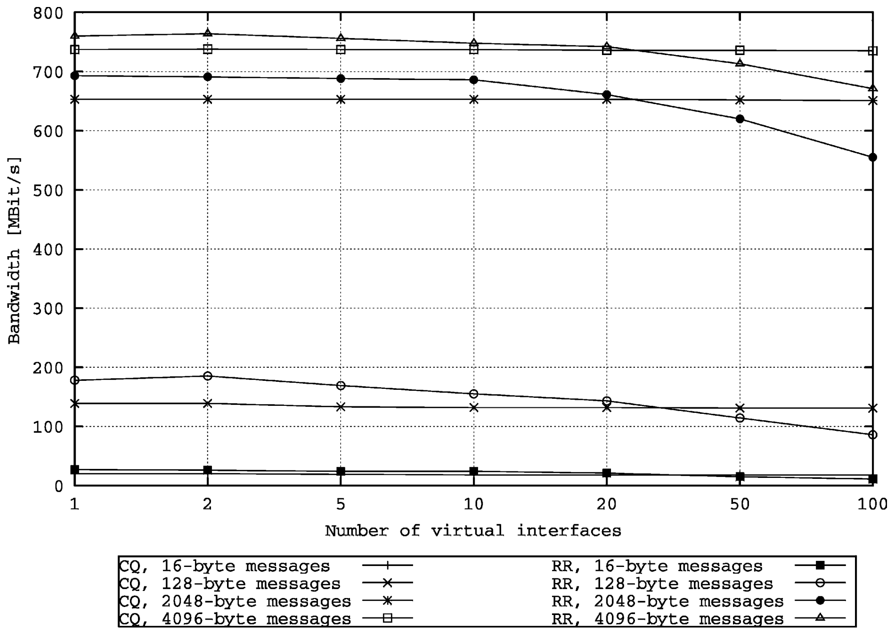

Fig. 8. CQ vs. RR descriptor processing strategy (one outstanding message).

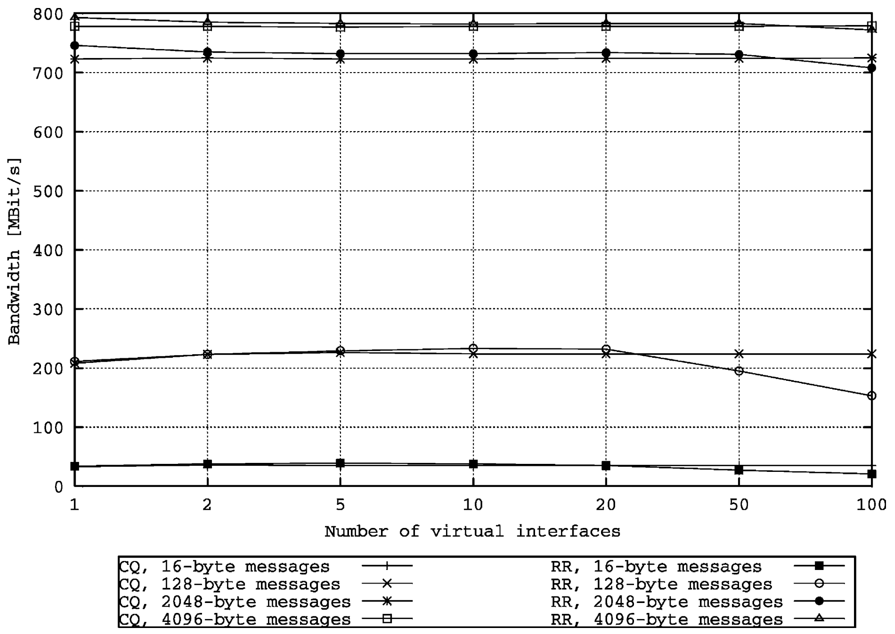

Fig. 9. CQ vs. RR descriptor processing strategy (two outstanding messages).

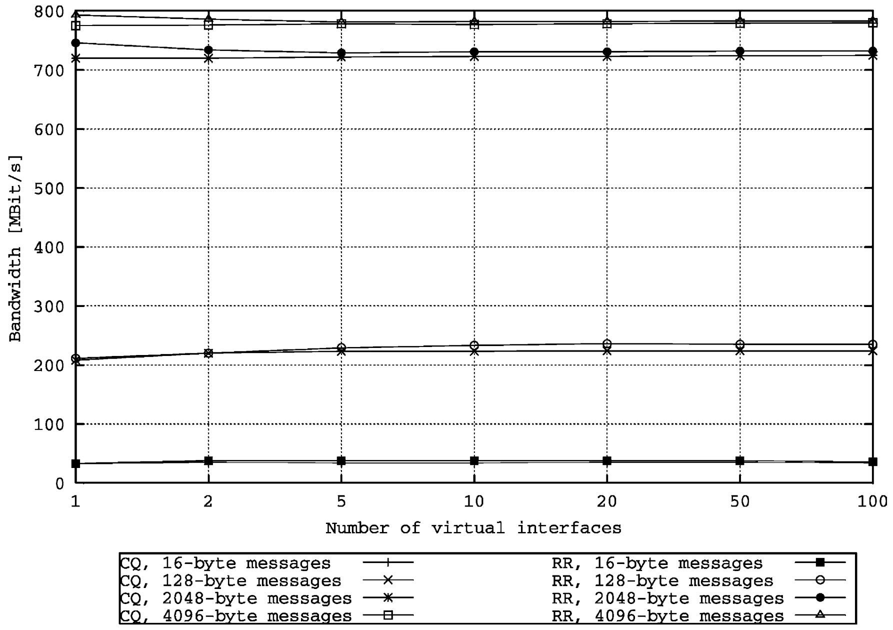

Fig. 10. CQ vs. RR descriptor processing strategy (five outstanding messages).

outstanding communication request. Fig. 9 shows that Giganet cLAN achieves its maximum throughput (793 MBit/s) when two or more communication requests are outstanding. In this case, the CQ processing overhead is approx. 1.8%. The advantage of CQs vanishes when five messages are outstanding. Fig. 10 reveals that the overhead is approx. 0.5% using 4096-byte messages. We obtained equivalent results with 10 outstanding messages. The overhead caused by the RR strategy increases with the number of connections (Fig. 8). Fig. 10 shows that the overhead is hidden when the number of outstanding messages is high.

4. Related work

A number of investigations of VI architecture implementations and, more specifically, of the Giganet cLAN implementation have been reported in the literature. Prototype implementations (e.g., over Myrinet) and their performance results are described in [1,4,11], for instance. Higher-level communication layers (e.g., TCP, RPC and MPI) over Giganet's cLAN and their performance behavior are introduced in [3,15], among others. Performance results for various communication layers and applications are also available on the Giganet web site [8].

In contrast to these analyses, our work goes into more details of various VI Architecture features and their performance implications. Similar work for the Compaq/Tandem ServerNet II VI architecture implementation is reported in [14]. Our work also extends the performance results of [9] by providing data for the strided copy benchmarks (direct deposit data transfers to remote memory in the terminology of [9]) for the Giganet cLAN interconnect.

5. Conclusions

In this paper, we investigated the performance implications of using various communication models and specific features of VI architecture, more precisely its implementation incorporated in the Giganet cLAN (GNN1000 adapter cards). The results can aid an implementor of higher-level communication software in deciding which features to use or avoid.

RDMA write provides better performance than send/receive communication methods in terms of throughput, given that the same synchronization method for testing completion of communication operations is used (polling vs. blocking, i.e., interrupt-based notification). In terms of latency our results indicate that the choice of the synchronization method is very important: polling-based completion checking avoids the overheads of interrupt processing and context switching of the blocking synchronization (less than 10 μs and about 15 μs, on the Linux and Windows NT platforms, respectively), yielding round-trip latency figures several times lower than with blocking. It must be noted, though, that the choice of polling vs. interrupt-based notification involves several other aspects and trade-offs (e.g., [10]).

The use of multiple VIs and a completion queue (CQ) appears to be comparable to a hand-crafted descriptor processing strategy. Our results indicate that multiple VIs do not increase performance (throughput); the use of a CQ generates an overhead of 1.8–3% for the case of a single VI and pays off when more than 20 VIs are active. Given that a CQ can simplify communication software, its use can be recommended. Multiple outstanding messages always increase throughput.

The scatter capability of the send/receive model that we investigated using a more realistic and complex strided copy benchmark turned out to yield disappointing throughput, more than an order of magnitude worse than the simple method to transfer a contiguous block and scatter it locally on the receiving node. Increasing the size of the data blocks to be scattered, rapidly increases the performance of automatic scattering yet does not reach the maximum throughput that can be achieved when scattering is performed explicitly under program control. The superiority of the simple, explicit method is consistent with the results reported in [9].

References

[1] P. Buonadonna, A. Geweke, D. Culler, An implementation and analysis of the virtual interface architecture, in: Proceedings of the High Performance Networking and Computing Conference 1998 (SC'98), Orlando, FL, November 7–13, 1998. http://www.supercomp.org/sc98/papers/.

[2] Virtual Interface Architecture Specification, Version 1.0, Compaq Computer Corp., Intel Corporation, Microsoft Corporation, December 1997. http://www.viarch.org.

[3] R. Dimitrov, A. Skjellum, An efficient MPI implementation for virtual interface (VI) architecture-enabled cluster computing, in: Proceedings of the Third MPI Developer's Conference, March 1999.

[4] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, C. Dodd, The Virtual Interface Architecture, IEEE Micro, March/April 1998.

[5] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, Active messages: a mechanism for integrated communication and computation, in: Proceedings of the 19th International Symposium on Computer Architecture, ACM, New York, CA, 1992.

[6] T. von Eicken, A. Basu, V. Buch, W. Vogels, U-Net: a user-level network interface for parallel and distributed computing, in: Proceedings of the 15th ACM Symposium on Operating System Principles, ACM, New York, CA, 1995.

[7] GNN1000 High Performance Host Adapter User Guide, Giganet, Inc., September 1998.

[8] cLAN
Performance, Giganet, Inc. http://www.giganet.com/products/performance.htm.

[9] C. Kurmann, T. Stricker, A comparison of three gigabit technologies: SCI, Myrinet and SGI/Cray T3D, in: H. Hellwagner, A. Reinefeld (Eds.), SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters, LNCS 1734, Springer, Berlin, 1999.

[10] K. Langendoen, R. Bhoedjang, H. Bal, Models for asynchronous message handling, IEEE Concurr., April–June 1997.

[11] M-VIA: A High Performance Modular VIA for Linux, National Energy Research Scientific Computing Center (NERSC). http://www.nersc.gov/research/FTG/via.

[12] S. Pakin, V. Karamcheti, A. Chien, Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs, IEEE Concurr., April–June 1997.

[13] C. Papadopoulos, G.M. Parulkar, Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation, IEEE/ACM Trans. Network. 1 (2) 1993.

[14] N. Patel, H. Sivaraman, A model of completion queue mechanisms using the virtual interface API, in: Proceedings of the IEEE International Conference on Cluster Computing (Cluster'2000), Chemnitz, Germany, November/December 2000.

[15] H.V. Shah, C. Pu, R.S. Madukkarumukumana, High performance sockets and RPC over virtual interface (VI) architecture, in: Proceedings of the Third International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'99), LNCS 1602, Springer, Berlin, 1999.

[16] T. Stricker, T. Gross, Global address space, non-uniform bandwidth: a memory system performance characterization of parallel systems, in: Proceedings of the ACM Conference on High Performance Computer Architecture (HPCA-3), San Antonio, TX, February 1997.

Hermann Hellwagner received his Dipl.-Ing. degree (in Informatics) and PhD degree (Dr. Techn.) in 1983 and 1988, respectively, both from the University Linz, Austria. From 1989 to 1994, he was senior researcher and team/project manager at Siemens AG, Corporate Research and Development, München, Germany, working on distributed shared memory, parallel file systems, and performance analysis of computer and communication systems. From 1995 to 1998, he was Associate Professor of parallel computer architecture at Technische Universität München (TUM) where he led a group of researchers working on compute clusters and lightweight protocols and communication libraries based on the high-performance scalable coherent interface (SCI) SAN. Since late 1998, Dr. Hellwagner has been a full professor of computer science in the Department of Information Technology at the University Klagenfurt, Austria. His current areas of interest are multimedia communication, cluster computing, embedded systems and hardware/software interaction.

Matthias Ohlenroth received a Electrical Engineering degree from the University of Technology, Chemnitz in 1994. Currently he is working on his PhD at the Department of Information Technology, University Klagenfurt. His research interests include active networks, multimedia and cluster systems.