Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach*

Marcel-Cătălin Roşu, Karsten Schwan, and Richard Fujimoto
Georgia Institute of Technology, College of Computing
Atlanta, GA 30332-0280
{rosu, schwan, fujimoto}@cc.gatech.edu

Abstract

This paper presents a novel networking architecture designed for communication intensive parallel applications running on clusters of workstations (COWs) connected by high speed networks. This architecture permits (1) the transfer of selected communication-related functionality from the host machine to the network interface coprocessor, and (2) the exposure of this functionality directly to applications as instructions of a Virtual Communication Machine (VCM) *implemented by the coprocessor. The user-level code interacts directly with the network coprocessor as the host kernel only 'connects' the application to the VCM and does not participate in the data transfers. The distinctive feature of our design is its flexibility: the integration of the network with the application can be varied to maximize performance. The resulting communication architecture is characterized by a very low overhead on the host processor, by latency and bandwidth close to the hardware limits, and by an application interface which enables zero-copy messaging and eases the port of some shared-memory parallel applications to COWs. The architecture admits low cost implementations based only on off-the-shelf hardware components. Additionally, its current ATM-based implementation can be used to communicate with any ATM-enabled host.*

We use three applications to demonstrate the high performance of the architecture's current implementation: (1) a synthetic client/server application, (2) a parallel implementation of the Traveling Salesman Problem, and (3) a parallel engine for discrete event simulation (GTW). The distributed- and shared-memory versions of these applications have comparable performance. Furthermore, the VCM-based approach to creating a clustered, parallel machine is shown to scale well in terms of matching required with offered communication bandwidths.

*Partially funded by DARPA grant DABT63-95-C-0125 and by NSF Grants CDA-9501637, CDA-9422033, ECS-9411846, and MIP-94085550

1. Introduction

Clusters of Workstations (COWs) are an attractive option for parallel processing because their cost-performance typically exceeds that of comparable parallel machines, and their CPU and memory systems are often a generation ahead of those of parallel machines. One issue with COWs, however, is the high communication overhead experienced by parallel applications. High speed interconnects, like Myrinet and ATM, address this issue with novel device drivers and with programmable microprocessors (*coprocessors*) on the network interface. The goal is to make available a significant fraction of the hardware performance at the application level.

Our research addresses the fundamental question of how to structure and implement the interface between the network interface and applications. The approach presented in this paper is based on the assumption that the critical step in improving the performance of parallel applications on COWs is the reduction of host communication overheads, as shown both in our past work [18], and in recent work [16]. Such reductions may be attained by transferring selected communication-related processing activities from the host CPU(s) to the network coprocessor. Activities like packet multiplexing/demultiplexing directly from/into the application data structures are transferred to the coprocessor.

In our design, the coprocessor implements a low-level communication abstraction called *Virtual Communication Machine* (VCM). Applications directly interact with the VCM and use it either as a zero-copy messaging layer, or as a basis for application-specific or general purpose protocols. In either case, the VCM enables applications to customize their communication for high performance and better resource usage. The complexity of interacting with the VCM is hidden by a user-level library. A kernel extension cooperates with the coprocessor to enforce the usual protection rules among multiple, local applications.

To communicate, an application builds VCM programs (i.e., instruction sequences) and signals the VCM to execute

To be presented at HPDC-6 '97

them. Data transfers are performed as VCM instructions. A multithreaded application can have several VCM programs running simultaneously. A special type of interaction between application and VCM, called *loop interaction*, can be used to further reduce the application communication overhead for frequent data transfers. In all cases, application and VCM operate asynchronously and use shared memory to communicate instructions, parameters, and results.

The architecture admits implementations based entirely on off-the-shelf hardware. Currently, we use FORE SBA-200E cards in a cluster of Sun UltraSPARCs I. Furthermore, since we use standard ATM interconnect technology with standard AAL5 packets, clusters can be extended across any ATM-based metropolitan area network.

We consider that the intelligent network interface approach distinguishes itself from similar proposals [2, 3, 8, 9, 17, 25] for several reasons:

- *The network interface has an active role.* Rather than providing applications with user-level access to a (slave) network interface (by sharing it time- or memory-wise), our design gives the network interface access to the address spaces of all the applications it is servicing. This feature can be used to replicate the data segments used for interprocessor communication in shared-memory applications across the network, as will be shown in Section 2. This role is remotely similar to that of a coherence controller in a distributed shared-memory parallel machine and it is the basis for future extensions of this architecture.
- *Zero-copy messaging available at both sending and receiving ends.* In addition, the gather/scatter capabilities of most VCM instructions allow to extend the zero-copy feature to higher level protocols. Zero-copy messaging results in substantial reductions in the message latencies and in the system memory bandwidth consumed by the application. This capability enables highly efficient implementations of server-like applications and of cluster versions of certain shared-memory parallel applications.
- *Communication-related processing can be transfered to the network coprocessor.* As a result, a significant fraction of the host overhead can be transferred 'closer' to the physical network where it can be executed more efficiently. Additionally, this favors better performance, as the VCM and the applications operate asynchronously.
- *Buffer pages are managed by the application.* Although the host kernel is actually handling the buffer pages[1], the decision of which pages should be pinned down at any given time is made entirely by the application. The kernel also establishes and enforces the page limit for each application. This design decision removes functionality from the kernel or network coprocessor that they cannot typically perform well, due to insufficient information about application behavior. A server-like application can benefit from this feature by caching frequently requested data in pinned down regions.
- *Multiple network interfaces (i.e., VCMs) per host are supported.* Communication-intensive applications can use multiple VCMs. Additionally, the interconnect technologies of different VCMs do not have to be identical (e.g. a host may use ATM- and Myrinet-based VCMs simultaneously).

The next section presents how parallel applications can benefit from our architecture. Sections 3 and 4 present the architecture and its current implementation. The performance enabled by our architecture is presented in Section 5. Related work and our future plans conclude the paper.

2. Applications

The intelligent network interface approach proposed in this paper not only leverages good performance for parallel applications running on COWs, but it also facilitates the port of some of the shared-memory applications to a cluster environment. The performance of many parallel applications running on *distributed-memory* platforms is limited by high communication costs. Poor communication performance has an especially negative effect on applications with internal structures that result in poorly distributed communication loads, such as client-server applications. Our design improves the performance of such applications through its low overhead, high performance messaging, and its efficient integration of multiple network interfaces per host. Porting existing *shared-memory* applications to COWs is attractive because of the large cost difference between the two platforms. Shared-memory parallel applications that use explicit inter-processor communication can be ported to our networking architecture with modest programming effort and small performance penalty.

Three applications demonstrate the high performance of the current implementation: (1) a synthetic client/server application, (2) an implementation of the Traveling Salesman Problem (TSP), and (3) the Georgia Tech Time Warp (GTW), a parallel engine for discrete event simulation.

In client/server applications, the server is typically overloaded with data requests from its client machines. Using our architecture, the server can lower its communication overhead significantly by caching frequently requested data in zero-copy enabled regions or, equivalently, by enabling zero-copy messaging for the regions containing frequently

[1]Zero-copy messaging assumes that the application memory used in communication is pinned down.

accessed data. Furthermore, the server may use multiple network interfaces, if available.

The TSP application finds the shortest Hamiltonian tour of a set of cities using a client/server paradigm. The server generates partial routes and sends them to clients upon request, for expansion. In our cluster implementation, the server overhead (including the communication processing) is low enough to run an additional client on the same node. The loop interactions have an important contribution to reducing this overhead.

The first large shared-memory application ported on our networking architecture is the Georgia Tech Time Warp (GTW) system [5], a parallel kernel for discrete event simulation used in both research and commercial environments. GTW is designed to support efficient execution of small granularity discrete-event simulation applications that require as little as a few hundred machine instructions for processing an event. Using traditional message passing primitives is inappropriate since they usually require the execution of a significant number of instructions per message. Sending multiple events per message increases event latencies, thus increasing the number of rollbacks. The cluster version of GTW uses an application-specific protocol for passing events between remote processors. This protocol is built on top of VCM's zero-copy messaging and loop interactions. The performance of the distributed- and shared-memory versions of GTW are comparable.

Using VCM abstractions, the event pool shared by two processors is replicated across the network (see Figure 1). Events generated on one site are mirrored at the other end with little host involvement. Events are usually replicated at different addresses within the two regions, which differs from what happens in most distributed shared-memory systems. Similar techniques can be used for many parallel applications using explicit inter-processor communication.

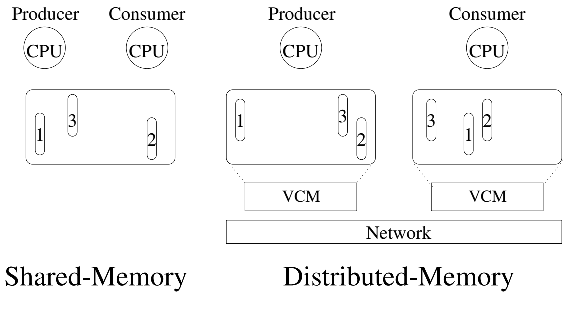

Figure 1. Extending Shared Regions across the Network

3. Architecture description

The high cost of communication on COWs comes from the inherent inability of scaling the performance of communication software along with the system's CPU performance. Our solution to this problem assumes that a microprocessor (*network coprocessor*) and a few megabytes of fast memory are present on the network adapters used in the cluster. Each card must also include one or more DMA engines under the control of the coprocessor and network-specific hardware to help with performance critical processing (e.g. CRC).

We claim that the potential benefits of the additional intelligence in the network card may outweigh its cost. We substantiate this claim by proposing and demonstrating a cluster communication architecture that efficiently supports communication intensive parallel applications. Its performance advantage is due to the fact that it enables applications to set a direct communication path between their address spaces and the 'wire'; data is manipulated by the network coprocessor under the control of the host CPU. To achieve this, selected parts of the communication software are moved from the host CPU(s) to the network interface.

The coprocessor is made responsible for ensuring data integrity, for assembling/disassembling messages directly from/into an application's data structures, for multiplexing/demultiplexing network messages, and for enforcing protection. Data caches on the host CPU are protected against unnecessary copies because messages are DMA-ed directly from/into application-level data structures. Host interrupts are reduced because the communication-related processing, usually done in interrupt context or started by a network interrupt, is now performed by the network coprocessor. As a result, a message may be sent or received with very low overhead.

The main component of the proposed networking architecture is the Virtual Communication Machine (VCM), which is implemented in firmware on the network interface card. The VCM has instructions for sending and receiving data, for managing the application interface, and for enforcing protection. On a host, each network interface that supports our architecture implements a VCM; other types of interfaces may be active at the same time and applications can use any combination of the available interfaces. This paper describes experiments that use only network interfaces implementing VCMs.

To use an interface, an application must register with the corresponding VCM. Upon registration, the application's address space is extended with a few pages, also mapped in the VCM address space (see Figure 2). This shared memory region – called *command area* – is used for the interaction between VCM and application; it is where VCM programs (i.e., sequences of VCM instructions), status words, and control message buffers are placed. The command area

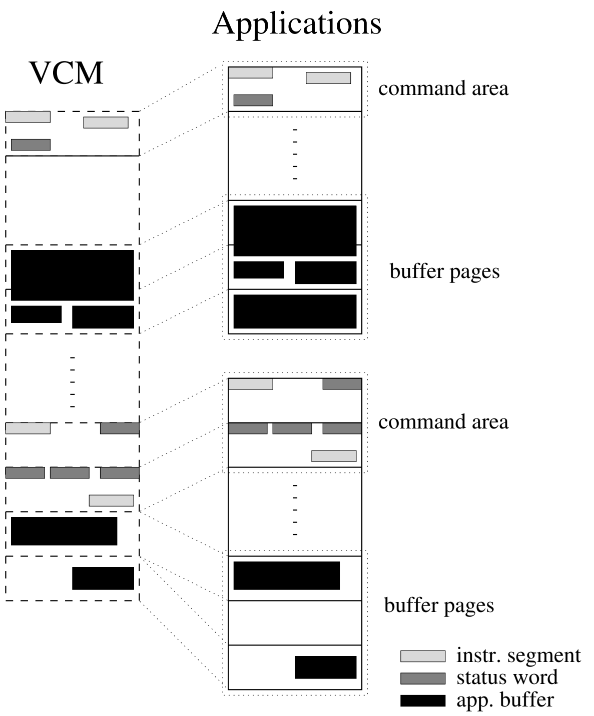

Figure 2. The mapping between applications and VCM address spaces

must remain mapped in the VCM address space for the entire duration of the communication. On the other hand, the application pages referenced by a VCM instruction – called *buffer pages* – must be part of the VCM address space only during the execution of the instruction using them, and they can be mapped/unmapped as needed. In summary, the VCM address space is the union of all the application pages described above. Buffer pages always reside in main memory, whereas the command areas may be placed in the network interface memory or in the host's main memory.

While mapped in the VCM address space, buffer pages must be pinned down in main memory. Command areas must be pinned down only if they are physically located in main memory. The management of the VCM address space, including the pinning/unpinning of the application pages is performed by a loadable *kernel extension* module. This extension notifies the coprocessor when the VCM address space is modified. Whenever the VCM address space is extended, the coprocessor is informed of the new pages' owner. The kernel extension also handles an application termination: it waits until the VCM cancels all of the application's instructions before it releases the application's pages.

Although the kernel manages the VCM address space, the application decides which pages to share with the VCM at any given time. This gives the application flexibility in controlling its communication overheads: it can choose between mapping buffers and using zero-copy data transfers, vs. assembling messages into already mapped buffers.

Application and VCM operate asynchronously with the data transfers being performed without the direct involvement of the host CPU, but under its control. Applications build VCM programs (instruction sequences) in their command areas and pass them to the VCM. Program or instruction completion is signaled using status words also placed in the command area. For each instruction, the VCM checks that an application refers only to the buffers it owns, and uses only its own connections. In contrast to [2, 9], the buffer addresses passed as instruction parameters are references into the VCM address space (i.e., address translation is performed on the host). While less general than a firmware-based TLB, this scheme keeps the overhead on the network coprocessor low. The VCM pulls outgoing data from the host memory and pushes it into the network. Conversely, incoming data is placed directly into memory areas specified by pending receive operations. Zero-copy messaging is possible when the DMA engines on the network interface card can move data directly between the host memory and the 'wire', bypassing the interface memory.

The *loop interactions* between application and VCM are designed to support repetitive operations on the same connection with spatially close buffers. Applications can take advantage of spatial and temporal locality in communication to reduce (1) the host overhead incurred with building VCM programs and (2) the VCM overhead incurred with checking parameters. Asymptotically, the host overhead approaches three instructions when using the loop interaction: a store, a load, and a compare.

A small *user-level library* complements the VCM and the kernel extension. This library hides most of the complexity of programming the VCM. We integrated this library with a user-level thread package and use it for a multithreaded server application. Using the library, applications can build virtually any application specific or general purpose protocols.

When multiple, high performance networks are attached to a single host, I/O bus usage may become a bottleneck and therefore, it may limit the *scalability* of communication performance. To address this issue, the VCM architecture permits trading I/O bus usage for CPU overhead by enabling alternative implementations of some of its components (e.g. the placement of the command areas).

Our network architecture is *open* and easy to *extend* and *port*. Hosts are connected through off-the-shelf routing equipment and can communicate with machines that are not using our networking architecture. Structuring the coprocessor code as a virtual machine makes the architecture easy to extend. Each new VCM instruction translates in a new user-level library call and the corresponding extension to the firmware. We ported our initial implementation from FORE SBA-200 cards and SunOS 4.1.3 to FORE SBA-200E cards and Solaris 2.5. The process was relatively easy because the VCM abstraction makes the firmware, the kernel extension, and the user-level library relatively independent.

4. Implementation

This section briefly describes our current implementation that uses a cluster of Sun UltraSPARCs I Model 170 running Solaris 2.5 equipped with FORE SBA-200E network cards; details can be found in [18, 19]. The FORE SBA-200E cards can transfer data directly between the wire and the host memory, bypassing the card memory.

The VCM is implemented as an interpreter running on the coprocessor (a 25 MHz i960 microprocessor) and the VCM address space is included in the SPARC's DVMA space. The kernel extension is a loadable device driver and it does not require any changes to the Solaris 2.5 kernel. Our implementation also includes a SPANS[2]-based connection server. This server is built as a VCM-level application using pre-assigned ATM connections. The nature of the architecture makes possible implementations for other interconnects that use a network coprocessor and for other operating systems.

The current implementation places the command areas in main memory. Their lengths are application specific but usually no more than a few pages. The initial instruction segment (where the application builds its first VCM program) is placed at a pre-defined offset in the command area and new instruction segments can be registered and released throughout an application's lifetime. The VCM's gather/scatter capabilities are limited to two segments (message header and body) in the current implementation; more segments will be added as the need arises.

4.1. VCM instruction set

The basic VCM instructions are *send* and *accept*; their parameters specify a connection id, the addresses of the corresponding memory segments, and their lengths. The *request* instruction combines a send and an accept. The *accept_multiple* instruction specifies a buffer where all the subsequent messages will be overwritten until a new accept on the same connection is posted. Besides the communication related instructions, the VCM has instructions for protection enforcement, and for registration and release of instruction segments.

Loop interactions are based on the *loop_send*, *loop_accept*, and *loop_request* instructions. A *loop* instruction specifies an access area (address and length) that contains the buffers to be used in all of its subsequent executions, and a status word to be used for activating these executions. After a loop instruction is interpreted as part of a program, the VCM start polling its status word. Whenever the VCM reads a buffer descriptor (i.e., length and offset in the instruction's access area), the corresponding operation is activated. The result of the execution is returned in the status word. Until cancelled, by writing a special value in the status word, a loop instruction can be activated multiple times.

Loop interactions give applications increased flexibility in controlling their communication overhead. However, with each loop instruction, the coprocessor experiences an extra overhead caused by polling the status word. Obviously, it is not a good idea to use loop instructions if they are rarely activated. To address the potential performance problem induced by excessive polling of the status words of loop instructions and of instruction segments, we built two other implementations, which are currently being tested. In the first implementation, all status words are placed in the interface memory. In the second implementation, each loop instruction is associated a status word in the host memory and a command word in the card memory. The coprocessor polls the command word and writes into the status word while the host polls the status word and writes into the command word.

4.2. VCM interpreter

The main loop of the VCM interpreter considers the following requests in the order in which they are listed below (see [18, 19] for details):

- Protection-related instructions: inform the VCM about changes in its address space or in the application's connection (ATM VCIs) set.
- VCM programs: all instruction segments are checked for programs to execute. For each program, all its instructions (limited to several dozens) are executed in order.
- Loop instructions: check all the loop instructions for an activation or cancelation command.
- Incoming data: the coprocessor checks the network for incoming ATM cells at the end of the interpreter loop *and* while waiting for data from the host memory. As a result, very few, if any, incoming cells are lost.

4.3. Protection and buffer page management

The protection mechanism may be described as follows: (1) the VCM only accepts protection management instructions from the kernel or from the connection server, (2) protection instructions are written directly into card memory,

[2]SPANS is a vendor specific protocol for ATM network signaling.

(3) the VCM checks the correctness of all parameters received from the application, and (4) messages longer than expected are truncated to the size of the receiving buffer. The kernel informs the VCM about any changes in coprocessor address space and page ownership. The connection server informs the VCM of the network connections each application may use. Upon notification, the VCM releases the descriptors referring to a connection to be closed or referring to a page to be removed from its address space.

To permit zero-copy messaging, all buffer pages referenced by a VCM instruction must be pinned down and mapped in the VCM address space while the instruction is executed. Mapping/unmapping and pinning/unpinning buffer pages are handled separately, subject to the condition that all mapped pages are also pinned. The host's physical memory and the VCM address space are managed by the kernel extension. Both are shared among multiple applications, and the boundaries are enforced by the kernel and the VCM. The decision to change the status of a page is made by the application, while the action is performed by the kernel. As a result, applications can implement the best policy to manage their buffer pages.

5. Architecture performance and evaluation

The performance of the VCM networking architecture proposed in this paper is evaluated with microbenchmarks, the synthetic client/server application, the Traveling Salesman Problem, and the Georgia Tech Time Warp. Experiments are conducted on a cluster of 11 Sun-Ultra I Model 170 workstations running Solaris 2.5. ForeRunner SBA-200E ATM cards are used as network interfaces and all the workstations are connected to a ForeRunner ASX-200WG switch. The shared-memory machines used for performance comparison are the KSR-2 and SGI Power Challenge running IRIX 6.1.

5.1. Microbenchmarks

The one-way latency (as half of the roundtrip time between two remote applications) is shown in Figure 3. The message size is varied between 4 and 65532 bytes (message size must be a word multiple). The latency is linear with the message size, with $\approx 60\mu secs$ for a 4 bytes message, and increasing by the inverse of the wire bandwidth. More specifically, the one-way latency is expressed by:

$$Latency(M) = ct_{SEND} + \frac{C_{SEND}*M}{B_{CPU}} + \frac{M}{B_{WIRE}} + ct_{RECV} + \frac{C_{RECV}*M}{B_{CPU}},$$

where M is the message size, B_{CPU} is the memory copy bandwidth of the workstation, B_{WIRE} is the wire bandwidth, ct_{SEND} and ct_{RECV} are constants, and C_{SEND} and

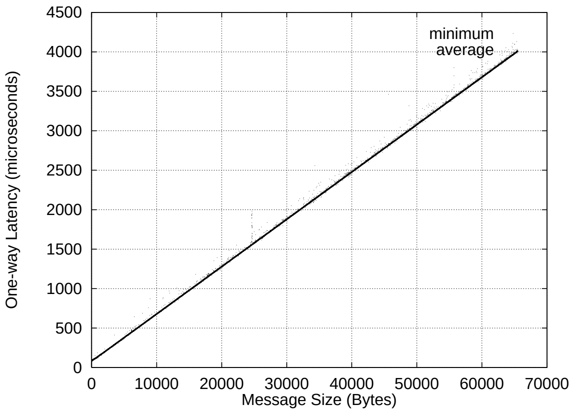

Figure 3. One-way Latency

C_{RECV} are the number of times a message is copied. In our architecture C_{SEND} and C_{RECV} are zero, therefore the latency variation with the message size is minimal. The maximum send rate (see Figure 4) approaches the maximum data capacity of the wire ($\approx 135 Mbits/sec$).

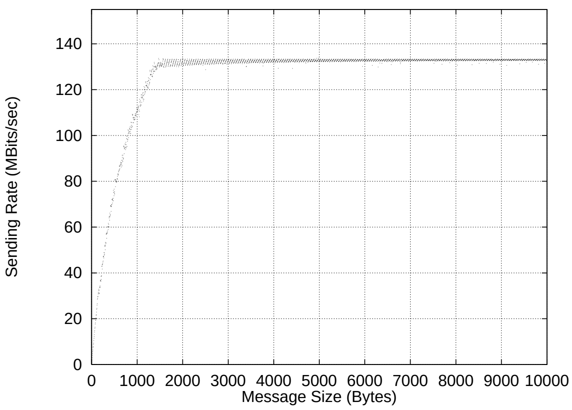

Figure 4. Bandwidth

The performance enabled by our networking architecture in a client-server context is evaluated with a synthetic client/server application. We experiment with a configuration of ten client workstations which issue back-to-back data requests to the server workstation. Figure 5 shows the outgoing bandwidth of the server as a function of the request size, when the server uses one or two interfaces. One can notice that for reasonably large requests, the achieved bandwidth is very close to the hardware limit and that the server almost doubles its performance when using two interfaces.

5.2. Traveling Salesman Problem

In both the shared- and the distributed-memory implementations of the TSP (see Section 2), the processor running the server also runs a client. Initially, the server generates a

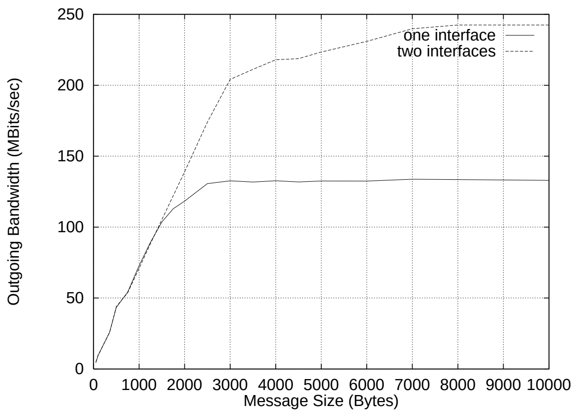

Figure 5. Server performance: one vs. two interfaces

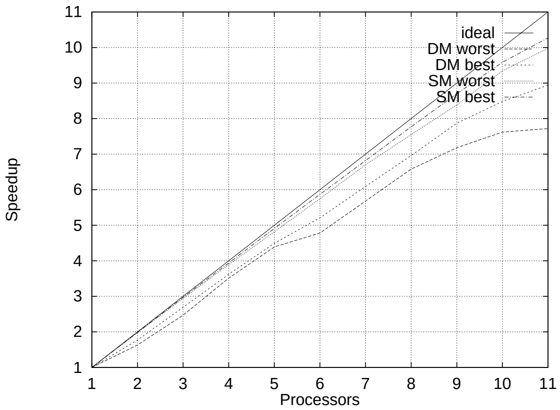

Figure 6. TSP Speedup

small number of partial solutions; after all of them are expanded, the server generates a new batch of partial solutions. The length of the shortest solution found so far is used to prune the search tree.

We experiment with configurations of 18 cities and partial solutions of length 5. A random number generator is used to determine the distances between cities, therefore the problem size varies with the initial seed. Figure 6 shows the results for the problems with the best ('DM best') and the worst ('DM worst') speedup among a dozen of randomly selected configurations. The same two problems were run on the Power Challenge and their speedups are shown as 'SM best' and 'SM worst'. The shape of the speedup curves is explained by the relatively small (when compared to a parallel machine) ratio of the time required for expanding partial solutions to the time spent waiting for them.

Figure 7 shows the communication overhead on the server processor expressed by its percentage in the total

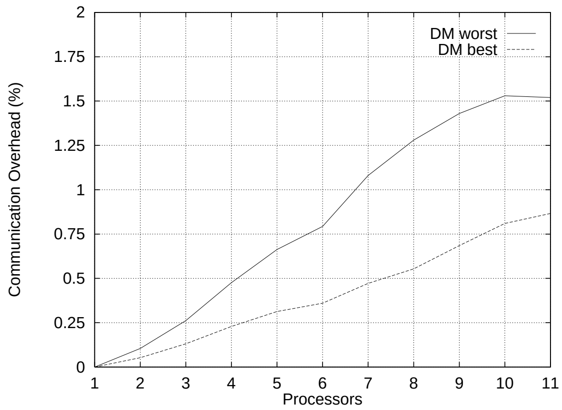

Figure 7. TSP Server Overhead

execution time. Notice that overloading the server is not an issue. However, the low overhead measured is an experimental proof that our communication architecture can be successfully used in high performance server systems.

A similar TSP implementation is used in [26] to characterize the performance of Optimistic Active Messages on an 128-node CM-5. In contrast, we add client activity to the server processor. The difference in processor speed and latency explains the difference between the speedup curves observed in the two implementations.

5.3. Georgia Tech Time Warp

A GTW simulation consists of a collection of logical processes (LPs) that communicate by exchanging timestamped messages or events (in the following, event and message are considered synonymous). The execution of each LP is entirely event driven. As a result of processing an event, an LP may change its state and may generate new events. Events within an individual LP are processed in timestamp order and the execution of a GTW program ends when all the LPs reach the endtime of the simulation. The number of LPs may be larger than the number of physical processors (or processing elements, i.e., PEs) available and several LPs will be mapped to each PE when necessary.

GTW uses an optimistic algorithm for parallel discrete event simulation. This means it automatically detects out of order processing of events within an LP, rolls back the incorrectly processed messages, and re-executes them [15]. Messages sent by rolled back computations must be cancelled, which may in turn cause additional rollbacks. In GTW, the rollbacks are generated only by remote messages (from LPs assigned to different PEs).

The GTW events are data structures (or memory buffers) containing (1) application specific information and (2) a few fields used by the GTW executive. Each PE has two event input queues: one for ordinary messages, and one for canceled messages.

Shared-Memory Specifics Message passing and message cancelation between remote PEs are among the GTW mechanisms that assume a cache-coherent shared memory. The message passing scheme that yields the best performance for large-scale, small-granularity simulation applications is the partitioned buffer pool approach [14]. In this approach, the global event memory is divided among PEs and further, the pool of each processor is subdivided into subpools, one for each destination PE. The buffers in a subpool are always used to carry events between the *same* pair of PEs.

The GTW scheme for cancelling messages sent by rolled back computation is based on keeping a link from every processed event to each of the events it has generated. This enables a PE to find all the messages it needs to cancel.

Cluster implementation In porting GTW to our networking architecture we tried to (1) maximize performance, and (2) make as few changes as possible in the underlying mechanisms. An important reason for (2) is that the cluster version was designed and implemented for clusters of both uni- and multiprocessors. Although the experiments in this paper use only uniprocessors, we expect to add multiprocessors to our clusters in the near future.

The primitives for message passing and message cancelling among remote PEs were changed to work across the network, with each pair of remote PEs being linked by two unidirectional connections. The partitioned buffer pool scheme proved a good match to the VCM capabilities. Each subpool is associated with a set of *loop_send* instructions such that every instruction can send a message located anywhere in the pool[3]. In contrast to the shared-memory version, buffers never leave the pool: they are always added at the end of the list for future use.

The sender's subpool is mirrored at the receiver's end (see Figure 1). Each receiver subpool is associated with several *loop_accept* instructions listening to the corresponding connection. When a message arrives, the buffer is placed in one of the input queues of the destination PE and also enqueued at the end of the subpool. The loop_accept is reactivated immediately with another buffer from the subpool. A sufficiently large number of loop_accept's ensures a low message loss rate.

Overall, the changes to the original message passing mechanism are minimal and no overhead is added to sending a message: the enqueue operation and the associated lock/unlock (input queues are protected by locks) were replaced by an activation of a *loop_send* instruction (one store operation) and an enqueue operation. Receiving a message is more complex and adds a certain overhead. The main source of overhead is the message loss detection and correction mechanism. For this purpose, a sequence number is associated with every message between remote PEs. When a gap is detected, the lost message is requested from the source PE. Message cancelation between remote PEs is performed by sending a small message identifying the message to be cancelled. Upon receipt, the destination PE locates the event to be canceled and adds it to its input queue of cancelled messages.

The optimistic nature of the Time Warp algorithm has both a positive and a negative impact on the performance of the cluster version of GTW. The positive aspect is that messages can be delivered out of order since the rollback mechanism will correct any causality violation. While GTW requires reliable communication between remote PEs, using a general purpose protocol which enforces in-order delivery (e.g., TCP) can hurt the performance. On the negative side, a network bottleneck can make the local simulation advance much faster than the global one, increasing the number of rollbacks. The remainder of this section presents two GTW applications and the performance of their cluster and shared-memory versions.

PHold - a synthetic application Our first experiment with GTW uses a synthetic workload model called PHold [13]. Each event generates one new message with the destination LP selected from a uniform distribution. As a result, the locality of the simulation is poor: in a configuration with n PEs, $\frac{n-1}{n}$-th of the events processed locally generate an event for a remote PE.

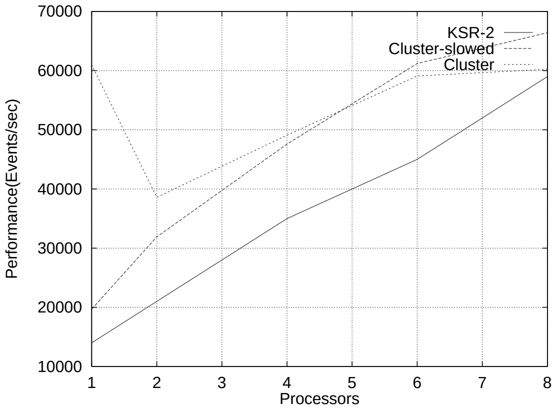

Figure 8. PHOLD Performance

Figure 8 shows the results of the cluster and shared-memory versions. Notice that distributing the simulation results in reduced performance. The mismatch between the processor speed and the number of messages that can be handled by the network interface explains the shape of the curve. Briefly, the card can handle 9,000-10,000 messages per second in both directions while the CPU can process about 60,000 events per second. In the two machine configuration, each processor handles about 19,500 events per

[3]Actually, as the range of a loop instruction is currently limited to 64K bytes, we had to further subdivide the subpools; details are omitted for brevity.

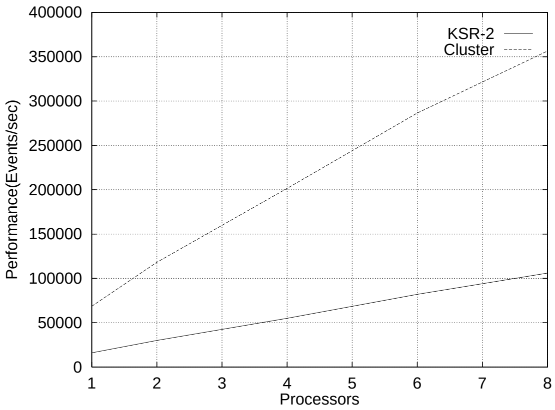

Figure 9. PCS Performance

second, half of them coming from or generating an event for the other PE. Based on this argument, we should expect in the 8 processors configuration about 11,000 events per second for each PE. However, for larger configurations, the effect of rollbacks on performance is significant (21-28% for 8 PEs) which explains the results. To further test our hypothesis, we slowed down GTW about three times. The results are plotted as "Cluster-slowed" and they show that a cluster based on our networking architecture and using processors 50% faster than those in the KSR-2 will have good performance even when running this communication-intensive simulation.

PCS - a wireless network simulation PCS [4] is a simulation of a wireless communication network based on a set of radio transmitters structured as a square grid (one transmitter per grid sector). In contrast to PHold, the event communication of this simulation is highly localized, with typically over 90% of the messages transmitted between LPs on the same processor. Figure 9 shows the performance of the shared-memory and cluster versions: (1) faster CPUs translate in much better performance and (2) our network enables better speedup than the KSR-2 interconnect.

6. Related work

Over the past several years, research efforts in operating systems [7, 8, 23] and network adapter architecture [10, 22, 24] have focused on improving the communication performance over high-speed interconnects like HIPPI, FDDI, ATM, and Myrinet. The main interest is in building faster, lighter, and cheaper communication mechanisms for COWs [3, 9, 11, 17, 25]. Most of these projects use off-the-shelf components while a few use dedicated hardware[11].

Early user-level communication architectures propose some form of integration between application and network interface, either as message areas [27] or ADCs [8]. Similarly, FORE's interface [12] between the host kernel and the firmware of their 200-series ATM cards uses shared message queues placed in the host memory. U-Net [25] maps these message queues into the application's address space. Messages are copied between the application and message queue buffers at both endpoints. U-Net/MM [2] adds zero-copy messaging to U-Net by incorporating a TLB-like data structure into the firmware. A similar software TLB is used to support VMMC on Myrinet [9]. In contrast to U-Net, the Illinois Fast Messages architecture [17] places the shared queues for outgoing data in the adapter memory. The CPU overhead increases, but the coprocessor overhead decreases.

In comparison to [3, 8, 25], in our architecture the message buffers are not allocated by the system; they are part of the application's original address space. This facilitates porting certain shared-memory parallel application to COWs, as shown for GTW. Additionally, different from [2, 9], we do not involve the coprocessor in the buffer management. This characteristic enables our architecture to comply with the operating system abstractions without replicating certain system information, like process priorities, on the adapter. Such replication would be too expensive for both the host and the adapter given the highly dynamic nature of this information in a system with a large number of communication intensive applications (as in [2]). Solutions such as prepinning "a number" of buffer pages per connection [2] may waste host memory and can result in resource allocation contradicting applications' relative priorities. However, in our architecture, the coprocessor performs all the data transfers which is different from the approach in [17, 25] and results in higher latency. We believe this drawback is offset by the decrease in host overhead.

Transferring specialized functionality to the communication coprocessor has been investigated for message-passing parallel machines. For instance, [21] investigates the benefits of running Active Messages on the coprocessor.

[1, 6, 20] propose new hardware architectures for network interfaces. Their primary objective is to eliminate the I/O bus bottleneck by integrating the network interface with the memory system. We believe that VCM-like abstractions will prove useful for such hardware architectures.

7. Conclusions

This paper presents the design and implementation of a low overhead communication architecture for high-speed networks. The distinguishing features of this work derive from the *Virtual Communication Machine*, a novel abstraction of the network interface. The resulting communication architecture is characterized by (1) flexibility of the integration between network and application, (2) very low overhead on the host processor, (3) latency and bandwidth close to the hardware limits, and (4) an application interface which enables zero-copy messaging and facilitates the porting of existing shared-memory parallel applications.

The performance results of our prototype with both synthetic and real-world applications are encouraging. Experiments show that shared-memory parallel applications can easily be ported onto COW platforms with comparable performance.

Two new implementations of the architecture are currently being tested and an ATM Forum UNI connection server is under development. We plan to extend the architecture and improve its performance while exercising it with other real-world parallel applications.

Acknowledgments We thank Chris Carothers for his help in understanding GTW's internals. Robin Kravets and Daniela Roşu provided valuable comments on drafts of the paper.

References

[1] H. Agusleo and N. Soparkar. Employing Logic-Enhanced Memory for High-Performance ATM Network Interfaces. *The 5th IEEE Symposium on High Performance Distributed Computing*, Aug. 1996.

[2] A. Basu, M. Welsh, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. *www2.cs.cornell.edu/U-net/*, Dec. 1996.

[3] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender-managed interface architecture. *2nd Symposium on Operating Systems Design and Implementations*, Oct. 1996.

[4] C. Carothers, R. Fujimoto, Y.-B. Lin, and P. England. Distributed simulations of large-scale pcs networks. *The 1994 MASCOTS Conference*, Jan. 1994.

[5] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. *Winter Simulation Conference*, Dec. 1994.

[6] A. Davis, M. Swanson, and M. Parker. Efficient communication mechanisms for cluster based parallel computing. *www.cs.utah.edu/projects/avalanche/index.html*, Dec. 1996.

[7] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. *2nd Symposium on Operating Systems Design and Implementations*, Oct. 1996.

[8] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. *SIGCOMM '94 Conference*, Aug. 1994.

[9] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. *International Parallel Processing Symposium*, Apr. 1997.

[10] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. *SIGCOMM '94 Conference*, Aug. 1994.

[11] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. *The 23rd International Symposium on Computer Architecture*, May 1996.

[12] FORE Systems Inc. Programmer's Reference Manual for AALI Interface. Jul. 1995.

[13] R. Fujimoto. Performance of Time Warp under synthetic workloads. *The SCS Multiconference on Distributed Simulation*, Jan. 1990.

[14] R. Fujimoto and K. Panesar. Buffer Management in Shared-Memory Time Warp Systems. *The 9th Workshop on Parallel and Distributed Simulation*, Jun. 1995.

[15] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, Jul. 1985.

[16] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead and bandwidth in a cluster architecture. *The 24th Annual International Symposium on Computer Architecture*, Jun. 1997.

[17] S. Pakin, M. Laura, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *Supercomputing*, Dec. 1995.

[18] M.-C. Roşu. Processor Controlled Off-Processor I/O. *Cornell University, CS Department, TR95-1538*, Aug. 1995.

[19] M.-C. Roşu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *Georgia Institute of Technology, College of Computing, TR GIT-CC-97-16*, May 1997.

[20] P. Sarkar and M. Bailey. CNI: A High-Performance Network Interface for Workstation Clusters. *The 5th IEEE Symposium on High Performance Distributed Computing*, Aug. 1996.

[21] K. E. Schauser, C. J. Scheiman, J. M. Ferguson, and P. Z. Kolano. Exploiting the Capabilities of Communication Coprocessors. *International Parallel Processing Symposium*, Apr. 1996.

[22] P. Sttenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer*, 26(3), Mar. 1994.

[23] C. A. Thekkath and H. M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[24] C. B. S. Traw and J. M. Smith. Hardware/Software Organization of a High Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications*, 2(11):240–253, Feb. 1993.

[25] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *The 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[26] D. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. *The 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jul. 1995.

[27] J. Wilkes. Hamlyn – an interface for sender-based communications. *Hewlett-Packard Laboratories, TR HPL-OSR-92-13*, Nov. 1992.