Fine Grain Parallel Communication on General Purpose LANs

Todd Mummert, Corey Kosak, Peter Steenkiste, Allan Fisher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Commodity workstations connected by commodity networks are increasingly viewed as an economically viable alternative to tightly coupled multiprocessors. In recent years, many scientific computing applications have been able to make effective use of various types of workstation clusters. The main difference between workstation clusters and the more traditional, tightly coupled distributed-memory systems is communication performance.

In this paper we present a host-network interface architecture that supports efficient remote memory writes across standard ATM networks, bringing performance closer to that of special-purpose, tightly coupled systems for a large class of applications. We show that minimal hardware support is required on the adaptor, and that the required features are very similar to those already needed on adaptors for high-speed networks.

We also describe an implementation of this architecture, and present measurements of communication performance indicating its effect on the breadth of applications that can use a general purpose network in an effective way.

1 Introduction

Commodity workstations connected by commodity networks are increasingly viewed as a viable alternative to traditional supercomputers and in recent years, many scientific computing applications have been able to make effective use of various types of workstation clusters [11, 19]. The main difference between workstation clusters and the more traditional tightly coupled distributed-memory systems (Paragon [15], T3D [1]) is communication performance and efficiency. Because of fundamental differences in their architecture (use of standard I/O bus, possibility of network errors), workstation clusters have communication overhead and latencies that can be as much as two orders of magnitude higher than tightly coupled systems. While this gap can be reduced by using optimized software [9], a significant difference remains.

In this paper we present a host interface architecture for workstations and PCs that reduces communication overhead. In particular, we consider a programming model based on remote memory access: hosts can write to predefined memory areas on other hosts without involvement of the receiving host. We show how an efficient implementation is possible over standard networks. Remote memory accesses are protected, i.e. can work in a multi-user environment, but no consistency is provided; it must be provided at a higher level. This communication model is commonly used as the target for parallelizing compilers, but it is more widely applicable: other programming tools, e.g. weakly-coherent shared memory, can be built on top of it.

The network interface architecture presented in this paper not only substantially improves performance, but it is simple and does not require many features beyond those already present in a high-performance network interface. We also discuss an implementation of this interface and present performance numbers.

The remainder of this paper begins by describing the remote memory access model. We then describe a network-host interface, called HARP for Hardware Assisted Remote Put, that supports efficient remote writes (Section 4). In Section 5 we describe a specific realization of the HARP architecture. We analyze HARP's performance in Section 6. We discuss related work in Section 7 and summarize in Section 8.

This research is sponsored by the Advanced Research Projects Agency under contract number F19628-92-C-0116.

2 The remote memory access programming paradigm

In the last few years, a number of distributed-memory systems have implemented a programming model based on "limited shared memory", meaning that nodes can access the private memories of other nodes, but no consistency is provided. Besides this common feature, *remote memory access* models can be quite diverse, both in their implementation (hardware versus software support) and in the details of their semantics. For example, in some cases the model supports both read and write (e.g. Cray T3D [1]), while in other implementations, only remote writes are supported (e.g., Shrimp [6] and deposit model [26]).

The main advantage of the remote memory access model over the more traditional message passing model is that it decouples control and data transfers. Decoupling the control from data transfers potentially improves communication efficiency: in many cases, only a single control transfer is needed for a sequence of data transfers, thus eliminating unnecessary control transfers. An example is a gather operation: a node receives data from a large number of other nodes (data transfers), but only has to be notified (control transfer) after all the data is available. The more traditional communication model based on paired send and receives couples control and data transfers, resulting in a much larger number of control transfers.

The remote memory access model is typically not used directly by programmers, but is a target for programming tools such as parallelizing compilers or runtime object libraries. An example is the Fx parallelizing FORTRAN compiler [14], which uses the deposit

model as a target. The deposit model has been implemented on the iWarp, Paragon and T3D systems [26]. The model can also be used to implement other programming models, such as message passing or coherent shared memory. Note that in the remote memory access model, memory consistency issues have to resolved explicitly by the higher level software. Some examples are discussed in [6].

3 Implementation approach

Tightly-coupled distributed-memory systems are characterized by a proprietary high-speed connect and a very tight coupling of the nodes and interconnect. The interconnect is typically as reliable as memory, i.e. the end points do not have to deal with recovery from corrupted packets, and the coupling between the interconnect and the compute node is at the level of the memory bus or cache, making communication very efficient. The remote memory access model has been implemented both completely in hardware (e.g. Cray T3D [1]), Shrimp [6] and in software (e.g. deposit model on iWarp[26]). The overheads and latency of a remote write are on the order of microseconds.

In contrast, loosely-coupled systems are built from off-the-shelf nodes (workstations or PCs) and network hardware (e.g. FDDI or ATM). This architecture differs in a number of fundamental ways. First, the network and node are coupled through a standard I/O bus, increasing communication overhead. Second, since standard networks do not guarantee reliably packet delivery, the software on the end-points is responsible for error detection and recovery, again adding overhead. Moreover, the bandwidth over standard commercial network hardware is typically lower than over proprietary interconnects, because of the lack of a market for very high-speed networks. While loosely-coupled systems have in general lower communication performance, they also offer significant advantages: they are cheaper, typically by a factor of 2-3 per node, and since new nodes can be added very quickly, the CPU performance of their nodes is typically about a year ahead on the growth curve compared with tightly-coupled systems.

A number of projects have looked at software implementations of the remote access model on loosely-coupled systems, i.e. without specific support from the network adapter. There were able to demonstrate a performance improvement over traditional message passing, mainly as a result of reduced context switch overhead on the receiving node [23, 27, 28].

It is possible to implement remote memory accesses more efficiently, i.e. reduce host overhead, by using a more powerful network adapter. A first approach is to rely on a powerful outboard processor to take over some of the communication tasks normally performed by the host, e.g. similar to the original Nectar systems [2]. An alternative is to provide dedicated hardware support, e.g. similar to the Shrimp interface [6]. While both approaches reduce communication overhead significantly, they have the disadvantage that they may lead to expensive adapters, and more importantly, since the adapter becomes more special purpose, they lose some of the advantages of the loosely-coupled systems architecture.

In this paper we present an architecture and implementation of a simple adapter for ATM networks that supports efficient remote memory writes, sometimes referred to as PUT operations. "Simple" means that the adapter is not much more complex than the network adapters found in today's workstations or PCs. Central to our approach is the partitioning of tasks between the host and the adapter so that we can achieve a good balance between overhead on the host and complexity on the adapter. We will refer to this architecture as HARP for Hardware Assisted Remote Put, and we will call the host that executes the write the *execution* node, while the node whose memory is being written into will be called the *memory* node.

4 The HARP Architecture

We discuss the basic operations that implement the HARP architecture and then describe the required adapter architecture, starting with a generic architecture and augmenting it with three critical capabilities.

4.1 Operations

Communication in the HARP model is a three phase process.

1. In the *set up phase*, the application sets up memory areas on both the execution and memory nodes that can serve as the source and destination of a PUT command. These areas are located in the address space of the application and are specified by a list of pointer-length pairs. The system on both the execution and memory node translates these area descriptors into buffer descriptors that can be interpreted directly by the network adapter and pins the pages holding the buffers in memory.

2. In the *communication phase*, the application can write (send) a predefined memory area to a predefined memory area on the memory node. The destination is specified as the identifier of a pre-established (logical) connection over an ATM network, represented by a Virtual Connection Identifier (VCI).

3. In the *verification phase*, host software verifies that data was delivered correctly, and an error recovery protocol is invoked if not, i.e. if a packet was lost or data corrupted. The verification phase will typically be executed as part of a synchronization phase in the application, and it will verify the correctness of a group of messages instead of a single message. Note that in many cases, the verification phase for one set of communication operations can be combined with the set up phase for the next set of operations.

The key idea is that since both the source and destination memory area have been declared in advance, the communication operations during the communication phase can be executed very efficiently. The only overhead is a "minimal" transmit of a preallocated buffer on the execution node, and there is no software involvement on the memory node. This is described in more detail below. At the same time, the performance impact of the set up and verification phases is small, since their cost is amortized over a large number of communication operations.

4.2 System requirements

We can identify four key system features that are needed to support the HARP architecture:

- *logical connections* and *early demultiplexing* of incoming messages, so that the adapter can put the data in a specific destination area without host involvement. ATM VCIs provide an ideal demultiplexer. Although few if any current adapters for packet-based networks have the ability to interpret headers, early demultiplexing is widely recognized as a powerful mechanism, and both hardware and software solutions have been proposed [3].

- *intelligent buffer management*, so that the adapter can manage multiple buffers simultaneously. Some of this functionality is required on any ATM adapter, since the cell-based nature of the network means that multiple packets can be in transit at the same time. Although packet-based networks typically

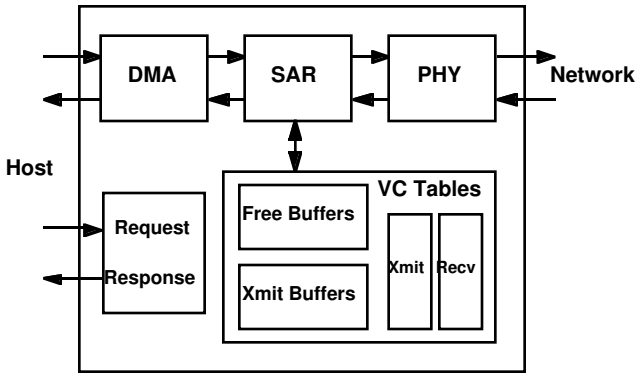


Figure 1: Generic ATM adapter architecture

only handle one packet at the time, the ability to make a list of buffers available to the adapter in advance is important for any high speed network interface.

- *user-level access* providing safe invocation of network operations.
- *low overhead error recovery* is needed to achieve a high likelihood of successful data transfers by the network. This can be achieved by driving the error rate down, i.e. low bit error rates and low loss due to congestion, or by being able to recover rapidly from errors.

The first requirements is easily supported by ATM and in the next section we describe a generic ATM adapter to illustrate this. We then describe the two additional adapter features that are needed on the adapter, flexible buffer management on receive and safe application-level access. Finally, we describe low-overhead mechanisms to recover from corrupted and lost packets.

4.3 A generic ATM adapter

Although it is possible to track logical connections in any type of network, ATM is particularly suited to this application since it provides a large number of logical connections. Since the connections are supported at the link level, i.e. the VCI is present in the cell header, the adapter can readily distinguish cells from different connections and treat them differently (i.e. early demultiplexing).

Figure 1 shows the architecture of the generic ATM adapter that we will use as the basis for our discussion in the following sections. Several research and commercial adapters [10, 12, 8] follow this architecture. The adapter sits on the host I/O bus and has the following components:

- Xmit: a *transmit VC table* with VCs (virtual connections) open for transmit. Associated with each open VC is per-VC state and, if a packet is being transmitted, a pointer to one or more buffer descriptors.
- Recv: a *receive VC table* with VCs open for receive. Associated with each open VC is per-VC state and, if the host is receiving a packet, a pointer to one or more buffer descriptors.
- SAR: a *Segmentation And Reassembly* unit that is responsible for breaking packets into cells on transmit and for recreating the packets from the (interleaved) cell streams on receive. Some ATM adapters also have a *transmit scheduler* that provides support for pacing the rate at which data is transmitted over each VC; rate control is often used to support traffic management in ATM network.
- Free buffers: a *receive buffer pool* that supplies the SAR unit with buffers for incoming packets.
- Request/Response: a *request and response interface* that allows the host to submit transmit requests of the form (VCI, buffer descriptor) to the adapter, and that allows the adapter to hand newly received packets or notification of transmitted packets to the host.
- a *DMA engine* that transfers data between buffers located in host memory and the SAR engine.
- PHY: a *physical layer interface* that transmits and receives cells over the network.

4.4 Intelligent buffer management

In the generic architecture, all incoming VCs shared the same receive buffer pool. When the first cell of a packet comes in, or when a buffer fills up in the middle of a packet, a new buffer is fetched. A simple and useful extension is to allow multiple pools, called *free lists*, and to have each VC select which free list it wants to draw buffers from. A simple use of this feature is, for example, to deposit incoming packets directly in the address space of the receiving process. In this case, each process that wanted to receive data directly would have its own free list, which would hold descriptors of buffers in that process's address space.

A second level of refinement is to allow the host to queue receive buffers directly on a specific VC, i.e. these buffers are preallocated for data coming in over that VC. In practice, this feature has to be combined with either a single free buffer pool or multiple free lists. The reason is that hosts typically have many open connections, most of which are idle at any given time. Preallocating buffers for all open VCs constitutes a large commitment of memory; sharing buffers for VCs that are not performance-critical is more practical.

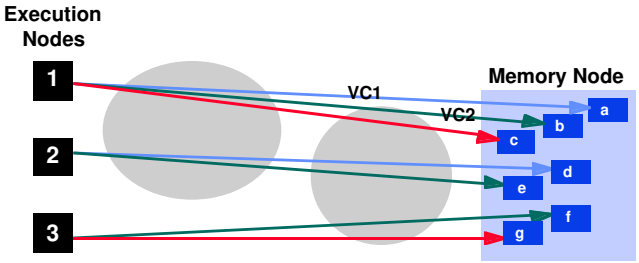
In summary, applications can elect to manage receive buffers for each VC in three different ways:

- VCs can have buffers queued directly for them, i.e. they use preallocated buffers. This results in a direct relationship between the VCI and a buffer from the perspective of the sender using the VC.
- VCs receive buffers from a specific free list that is shared with a small number of other VCs, allowing intelligent placement of data while still allowing sharing.
- VCs receive buffers from a default pool, which will usually be managed by the operating system, e.g. mbufs.

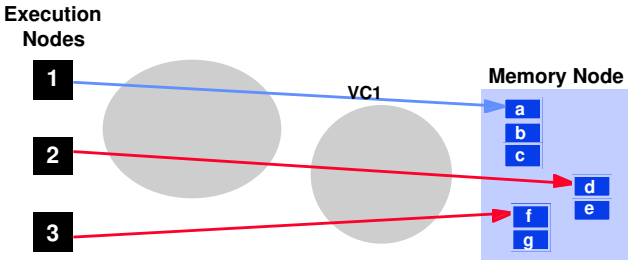
Either of the first two options (or both) is sufficient to implement HARP, since they make it possible to associated buffers with specific VCs. There are two basic approaches to implementing remote memory writes using the above buffer architecture.

In the first approach, the memory node assigns a VC to every (destination buffer, execution node) pair, and execution nodes can write to a buffer on the memory node by selecting the appropriate VC. This is illustrated in Figure 2(a). Node 1 can send to buffer **a** by using VC1 and to buffer **b** by using VC2. Note that some of the buffers might be the same, thus allowing multiple nodes to write to the same buffer. During the communication, no software involvement is needed on the receive nodes, i.e. there are no interrupts to notify the host about adapter events; all events are handled during the verification phase.

An alternative approach is to use the free lists. If the memory node knows what areas will be written by each execution node and if the nodes can agree on what order the buffers will be written, then



(a) Based on per-VC buffer allocation



(b) Based on free lists

Figure 2: Remote memory writes in HARP

the memory node can assign a free list to each execution node, queue all the buffer descriptors for the execution nodes on the appropriate free list and the execution nodes can use a single VC to write all their buffers (Figure 2(b)). For example, node 1 uses VC1 to write to buffers **a**, **b** and **c** (consecutively). This approach places more constraints on the application, but is potentially more efficient since fewer VCs have to be set up and managed.

Note that buffers that will be used for sending or receiving data have to be pinned in memory, but that is common practice for network (or other) devices that use DMA. Moreover, we assume that the host guarantees consistency between memory and the cache for accesses from the I/O bus; that is a common feature on recent workstation and PC models.

4.5 User level access to the device

One of overheads associated with send operations is the validation of the transmit request by the operating system. Safe user-level device access can eliminate this overhead and improve communication performance. Note that this feature is "optional" in the sense that even without user level access to the device, the HARP architecture provides benefits. Without this feature, the operating system has to validate transmit requests, but data can still be transferred directly from the sender's address space to the network and to the receiver's address space, and the buffer management cost can still be amortized.

On most workstations and PCs, only the operating system or a trusted server can directly transmit or receive packets through a network device. The reasons for this restriction are, first, the system wants to guarantee the correct operation of the network interface, which is a shared resource, and, second, the system wants to limit processes to only send and receive packets that they are allowed to send and receive. Any mechanism for direct application-level access to the device will have to enforce these two restrictions.

Integrity of the device is guaranteed by limiting the way applications can access the device to transmit or receive data. The simplest solution is to have simple hardware queues that can be mapped in the application's address space. A small number of such queues, each located in a separate page of the host's virtual memory system, would allow a small set of processes to access the device directly. An alternative would be a simple software queue in adapter or host memory, although this would require the adapter to make more expensive checks for correctness. In both cases, the format of the request should be simple enough that its validity can be easily checked outboard as described below.

We note, parenthetically, that another way to "corrupt" the device is to flood it with transmit requests, thus potentially preventing other applications from using the network. This is a hard problem, and most networks and network interfaces do not deal with this problem adequately, i.e. they don't guarantee fair network access. In our architecture, the transmit scheduler, which is controlled by the operating system, makes it possible to provide a degree of fairness between competing applications.

To limit an application's access to the device to valid transmit and receive requests, each application is allowed to set up a number of packet buffers in its address space. A packet buffer consists of a list of pointer-length pairs, and can be used to hold one packet for transmission or reception. Applications can also establish a number of private VCs and can allocate one or more free lists for their private use. If the adapter knows what packet buffers, free lists, and VCs belong to each application for which it support direct buffer access, then it can easily verify the following application requests:

- transmit n bytes of data from a packet buffer over a VC
- queue a packet buffer as a receive buffer on a specific VC
- queue a packet buffer on a free list.

Moreover, since each VC is associated with a specific application, responses connected with packet reception, transmit notifications, or errors can easily be queued for the right application.

Many error conditions can occur, either as a result of network errors, application errors, or malicious applications. In most cases, the adapter responds by falling back on default solutions that are supported by the operating system. For example, if a packet arrives while no packet buffer is posted, the system can drop the packet or use system buffers and post an appropriate error notification to the user. Similarly, when the application does not empty its response queue, responses can be posted to a default queue emptied by the system, or they can be dropped. Invalid transmit request can be rejected.

4.6 Reliable delivery

During the verification phase, applications have to detect and recover from possible errors. There are two types of errors: lost packets and packets corrupted by bit errors or lost cells. Lost packets can be detected using timeouts. Note that lost packets, i.e. all cells are lost, are extremely unlikely, certainly in an ATM network that uses appropriate flow control mechanisms to guard against cell loss due to network congestion. If the entire interconnect is a single ATM network, we can rely on ATM's HEC and the AAL5 CRC for error detection [29]. The 32 bit CRC over the data is very strong. A cell with an incorrect header that passes the relatively weak HEC will create an incorrect AAL5 byte count and will also nearly always cause a CRC error. Similarly, lost or discarded cells and cells with corrupted data will be caught by the AAL5 CRC. To allow the application to detect these errors, information about the arrival of packets and their correctness has to be passed up from the adapter to the application running on the host.

One common style of compiler-generated communication consists of a data exchange step (i.e. a series of PUTs) followed by a synchronization step. To ensure reliable data delivery over an unreliable network, the application will check for correct data delivery

as part of the synchronization step. In most cases, all packets will have been delivered correctly. However, when either a corrupted or lost packet is detected, as described above, the memory node has to request retransmission of the missing data. Note that this scheme requires that senders retain or copy memory areas that were transmitted until after synchronization is complete. This scheme can be viewed as a delayed protocol execution, where the protocol is executed in user space and is very light-weight. This requires both low link error rates and effective bandwidth allocation or congestion control methods that minimize the loss of data due to congestion in order to achieve good performance.

4.7 Applicability of HARP

The main application requirements for HARP are predictable data patterns so that buffers can be set up appropriately during the setup phase, and clusters of communication operations, so that the set up and verification phases can be amortized over a larger number of individual communication operations. Ideally, the communication patterns will be repetitive so that the same buffers and buffer descriptors can be reused for a large number of communication operations. Predictable and repetitive communication are very common in distributed computing applications. For example, this approach is well-suited to a parallelizing compiler or distributed object library, where the code is typically organized as a sequence of compute and communicate phases, and the communicate phases have the appropriate characteristics. Some message passing libraries, e.g. MPI [13], also directly support collective communication operations that fit the requirements.

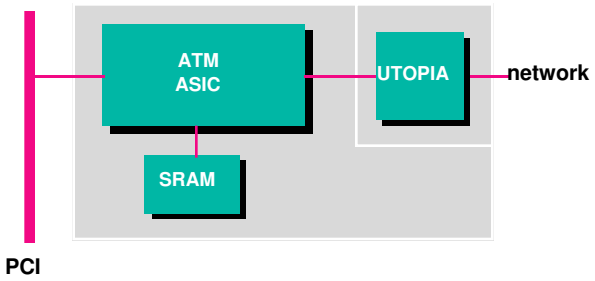
Although our focus has been on collective communication, the features of the HARP architecture also provide efficient implementation for many other styles of communication. For example, a traditional send/receive model, although it involves recipient interaction for each individual message, will still benefit from the avoidance of copying [24, 16] and potential avoidance of context switching provided by HARP's support for demultiplexing. Similarly, conventional network level protocols like IP can benefit from automatic demultiplexing [3] and application-level access. Moreover, weakly-coherent shared memory can be implemented on top of HARP, i.e. the lock and synchronization primitives can be used to implement the HARP verification phase and coherence.

5 The Credit Net Host Interface

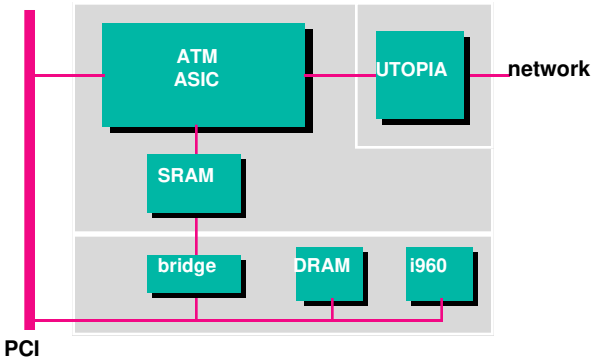
Carnegie Mellon University and Intel Architecture Labs have jointly designed an ATM network adapter for the PCI I/O bus in the context of the Credit Net project. This project has brought CMU, Intel, Harvard University and Bell Northern Research together to build a 622 Mbps ATM network that uses credit-based flow control for traffic management [17]. Adapters for standard 155 Mbps and 622 Mbps ATM/SONET links have been built by Intel [17]. Although the adapters are intended to be used as "regular network" interfaces, they support the HARP architecture. In this section we first briefly describe the Credit Net adapter, and then show how it supports the three features essential to HARP.

5.1 Adapter architecture

Figure 3(a) shows a block diagram of the Credit Net adapter. It consists of an ASIC, memory and a physical layer block. The memory holds the VC transmit and receive tables, and the buffer descriptors. All the components of the generic architecture (Figure 1), other than the physical layer interface and data structures, are on the ASIC. They include the SAR unit, a transmit scheduler, buffer management support, support for request and response mailboxes with



(a) Base adapter architecture



(b) Architecture with 960 support for experimentation

Figure 3: Credit Net ATM adapter architecture

the host, DMA support and the PCI interface. The ASIC supports both 155 and 622 Mbps links, with an aggregate (bidirectional) peak throughput of approximately 800 Mbps, given a sufficiently fast I/O bus.

A second version of the adapter is shown in Figure 3(b)b. It is identical to the adapter in Figure 3(a), except that an Intel i960 microprocessor with local memory has been added. From the perspective of the ASIC, the i960 looks like a second host. The i960 has its own set of request and response blocks that it can use to schedule cell transmission and to send/receive packets from/in the shared memory. The i960 has two major functions. First, it can serve as an accelerator for such tasks as link-by-link per-VC credit-based flow control[18, 22], ATM Forum-style rate control [7], and cell scheduling algorithms on transmit that are beyond the capability of the scheduler available on the ASIC. Second, the i960 can provide direct application access to the device by validating network operations and carrying out a request-reply protocol with the host.

Figure 4 shows the performance of the OC-3 adapter for a 90 MHz Pentium PC running NetBSD, the platform we use for the HARP measurements presented in the next section. The 90 MHz platform allows us to achieve a peak throughput using AAL5 of 124 Mbit/second using AAL5 and the peak throughput for TCP/IP measured using *ttcp* is 68 Mbit/second. The adapter is not the bottleneck in either case: AAL5 throughput is limited by the PCI chipset, while the TCP/IP throughput is CPU limited. For example, on a 133 MHz Pentium platform, which uses a faster PCI chipset, the adapter can sustain the peak OC-3 rate of 135 Mbit/second *full duplex*, i.e. an aggregate bandwidth of 270 Mbit/second, for AAL5, while the peak TCP/IP throughput is 85 Mbit/second.

5.2 HARP

We discuss how the three features that make up the HARP architecture are supported by the Credit Net adapter:

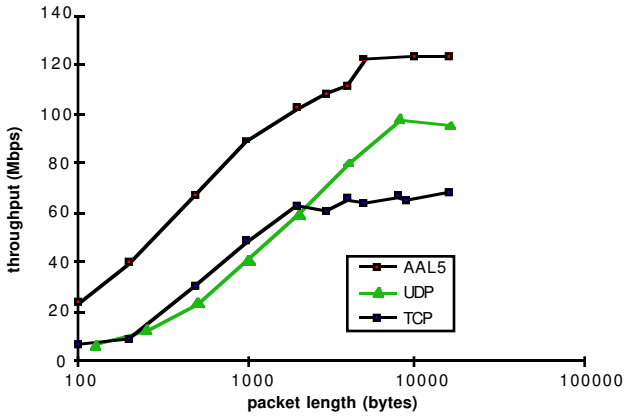


Figure 4: Throughput of Credit Net ATM adapter for 90 MHz Pentium PC

- *intelligent buffer management*: the ASIC supports a subset of the intelligent buffer management features described in Section 4.4. Specifically, the ASIC supports the direct allocation of buffers to VCs and sixteen free lists. Each VC can specify two free lists from which to draw free buffers for incoming packets: the first free list is used for the first buffer of a new packet, while the second free list is used for subsequent buffers. This mechanism was included for the purpose of header-data splitting. The only constraint relative to the most general HARP model is that at most one buffer can be assigned directly to a VC at any point in time.

- *user level device access*: the ASIC does not directly support user level device access. For example, it only supports two request/reply blocks and it does not do the necessary checking on incoming request. We plan to use the i960 to experiment with user level access: applications hand requests to the i960 which does the necessary checking and passes them on to the ASIC. Responses similarly go through the i960 on their way to the host so that they can be accessed safely by user level processes. To speed up this interaction, the host will indicate to the i960 what process is active as part of a context switch; this will avoid expensive polling by the i960 of queues owned by inactive processes.

  A hardware implementation would be based on a table that is indexed by (process id, buffer number), e.g. applications would specify buffers indirectly. The adapter would check that the specified index is valid, and it would do a length check on the amount of data sent or received. While using a general-purpose processor is expensive, it is an attractive solution for experimentation.

- *reliable delivery*: this task is mainly performed by the host, with some support from the adapter. Specifically, for incoming packets, the adapter provides the host with packet length, and checksum, so that the host can validate the contents of the packet. The Credit Net adapter provides this information to the host, which can use it to detect errors and recover if needed.

Since the adapter supports both multiple free lists and direct queueing of buffers on VCs, both methods for implementing remote memory writes (Section 4.4) are supported by the Credit Net adapter. As a result of the restriction that only a single buffer can be queued on a VC, in the approach based on direct queueing, the size of the buffers is limited to a single VM page (the adapter works in terms of physical addresses), and large application buffers will have to be broken up. Because there are only 16 free lists, the approach based on free lists will work up to 15 receivers; one of the free lists is needed for kernel buffers.

6 Performance evaluation

This section evaluates the performance of the HARP architecture using a distributed matrix transpose benchmark. A matrix is distributed row-wise over N nodes and each node has S stripes of data. This corresponds to a 1-dimensional block-cyclic distribution; for $S = 1$ it is a block distribution. The transpose operation redistributes the data to a column-wise distribution with the same properties. The matrix can be viewed as consisting of $(N * S)^2$ blocks of size B, where B represents the size of one stripe. During the transpose, each node will send S^2 blocks to the $N - 1$ other nodes.

We consider three variant implementations:

HARP: The HARP architecture: the implementation uses multi-packet queueing, automatic demultiplexing, and user-level reliability. The implementation is based on the free buffer lists. Data is not copied.

AAL5: An implementation using raw AAL5: both the sender and the receiver copy the data exactly once, but there is no protocol processing.

TCP: An implementation using conventional BSD sockets and the TCP/IP protocol for reliable delivery. This protocol stack copies the data once and also calculates a checksum.

Figures 5 and 6 show the performance for up 5 nodes, the maximum we currently have access to. The experiments were performed using the 90 MHz platforms using OC3 adapters without 960, i.e. there is no application-level access to the device and all send requests are checked by the operating system. Credit-based flow control is used to avoid cell loss due to buffer overflow in the switches. We observe that HARP performs significantly better, i.e. eliminating synchronization for every packet significantly improves efficiency. The costs that are eliminated on the receiver relative to the AAL5 implementation include copying of the data, interrupt overhead, and context switching.

7 Related work

A number of systems have implemented remote PUT (or a memory-coherent variant). The best way to distinguish between the different implementations is to break up PUT into its basic operations, and compare how these basic operations are implemented in a number of systems. Basic operations can be supported in hardware, or they can be implemented by system or user-level software on top of explicit message passing.

The basic operations can be grouped in operations at the end points related to the generation and execution of the write command, and in operations related to the transport of the command and data over the network:

1. *mapping*: how is the relationship between addresses on the execution node and the memory node established? Options are hardware, virtual memory or user-level software (e.g. compiler).

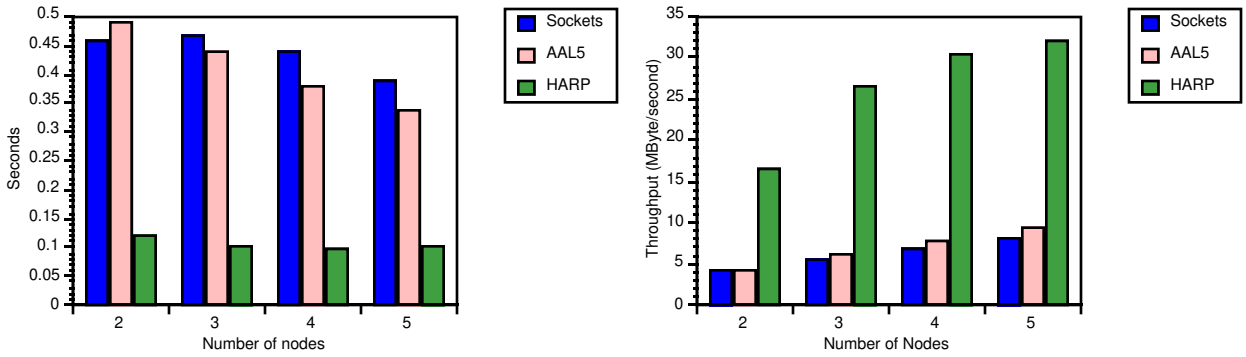


Figure 5: Time (left) and aggregate throughput (right) for transpose of 1K by 1K matrix (4MByte)

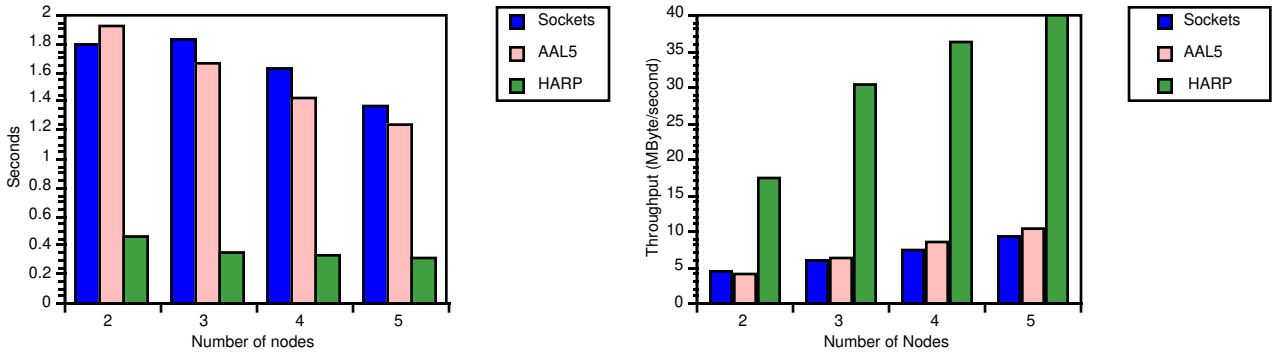


Figure 6: Time (left) and aggregate throughput (right) for transpose of 2K by 2K matrix (16MByte)

2. *generation of the write command*: options are hardware, selection using virtual memory system, or explicit user-level address calculation resulting in a (node address, memory address) pair.

3. *collecting of the data*: gathering the data and formatting it as an appropriate message. This can be done in hardware, given flexible DMA facilities, or in software by copying.

4. *decoding the write command*: interpreting the incoming message to determine what area in memory should be written. This can be done in hardware or software, where software can be VM software or user-level software.

5. *execution of the write command*: scatter the data in the incoming message to the appropriate areas in memory. This can be done in hardware (i.e. DMA engine) or software (i.e. copy).

6. *reliable completion of the write command*: while the communication subsystems of tightly coupled multiprocessors typically guarantee reliable delivery of the data, LANs (and networks in general) typically use unreliable hardware, and rely on end-to-end software protocols to detect and recover from lost and corrupted data. Note that the two types of networks differ in more aspects than how reliability is provided (e.g. routing), but we are mainly concerned with endpoint functionality in this paper. More recently, groups have proposed networks that sit in between these two extremes, i.e. they present a reliable interconnect that is flexible enough to be used in a LAN environment (e.g. [21]).

Table 1 shows how the basic operations were implemented in some example systems that implement remote write on distributed memory hardware. The left-hand columns represent systems that implement a shared-memory programming model based on a weak consistency coherency protocol (e.g. [20]): remote memory operations are completely supported in hardware. The distributed shared memory column shows systems that implement the same programming model over workstations connected by a network. The main idea is to rely on the virtual memory system to catch memory accesses to memory regions currently allocated on a different node [4] [5]. An alternative is to check memory references explicitly [30].

The other columns present levels of support for remote PUT between these two extremes. The Cray T3D supports remote PUT and GET in hardware [1]. Note that in contrast to more traditional shared memory machines, the hardware does not provide memory consistency, i.e. the software has to implement an appropriate consistency protocol. Shrimp [6] uses custom adapters to connect PCs to a custom interconnect, specifically the Paragon interconnect. Applications can set up mappings between memory areas in the private memories of different nodes. Once these mappings have been established, applications can transfer data between these memory areas in a light weight fashion; the options are automatic updates, or updates under software control (i.e. the application issues a PUT). The deposit model on iWarp implements this model entirely in software [25].

Finally in the right-most column, a number of projects have explored the support of a remote PUT and GET for networks connected by LANs [23] [28] [27]. These systems typically rely on remote procedure calls to transfer commands and data kernel-to-kernel. They can be more efficient than traditional send-receive operations because the application does not have to be scheduled (i.e. data transfer without control transfer).

Clearly, given implementations of comparable quality, the all-

	HW Shared Mem	T3D	Shrimp	iWarp Deposit	HARP	Distributed Shared Mem	Workstation LAN
mapping	hardware	software	software	software	software	VM	software
generation	hardware	hardware	hardware	software	software	VM	software
reading	hardware	hardware	hardware	software	hardware	VM	software
decoding	hardware	hardware	hardware	software	hardware	software	software
write	hardware	hardware	hardware	software	hardware	software	software
rel. del.	hardware	hardware	hardware	hardware	software	software	software

Table 1: Implementations of remote PUT

hardware solution will result in remote memory accesses with the lowest latency and overhead, but it will also be the most specialized and typically the most expensive implementation. The software/VM solutions result in more flexible and more portable implementations, but suffer from software overhead and yield significantly less efficiency and lower performance. The solution presented in this paper, HARP (Hardware Assisted Remote Put), uses simple enhancements to typical network adapters to achieve significantly better performance than typical LAN implementations at minimally higher cost.

The HARP architecture is, in a sense, a compromise between the Shrimp and workstation/LAN systems of Table 1. We use a LAN for connectivity with reliable delivery provided by the host (workstation/LAN), but the adapter supports remote PUTs in hardware.

8 Conclusion

We have presented a host-network interface architecture that makes communication on commodity workstations and PCs significantly more efficient, bringing performance closer to that of special-purpose, tightly coupled systems for a large class of applications. An important features of the adapter architecture is that minimal hardware support is required on the adapter, and that the required features are very similar to those already needed on adapters for high-speed networks.

We described an implementation of this architecture, presented measurements of communication performance, and showed that it can support finer-grain communication than can traditional adapters and networking software.

9 Acknowledgements

We acknowledge the contributions by Prashant Chandra and our colleagues at Intel Architecture Labs.

References

[1] D. Adams. *Cray T3D system architecture overview*. Cray Research Inc., September 1993. Revision 1.C.

[2] Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom, and Peter Steenkiste. The design of nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, Boston, April 1989. ACM/IEEE.

[3] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PATHFINDER: A Pattern-Based Packet Classifier. In *Proceeding Conference on Operating Systems Design and Implementation*, volume Winter, pages 115–124. Usenix, November 1994.

[4] John Bennett, John Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176. ACM, March 1990.

[5] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. In *Proceedings of COMPCON*. IEEE, 1993.

[6] Matthias Blumrich, Kai Li, Richard Alpert, Cesary Dubnicki, and Edward Felten. Virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[7] Flavio Bonomi and Kerry Fendick. The rate-based flow control framework for the available bit rate atm service. *IEEE Network Magazine*, 9(2):25–39, March/April 1995.

[8] C. Brendan, S. Traw, and Jonathan M. Smith. Implementation and Performance of an ATM Host Interface for Workstations. In *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*. IEEE, February 1992.

[9] Jose Brustoloni. Exposed buffering and subdatagram flow control for ATM LANs. In *Proceedings of the 19th Conference on Local Computer Networks*, pages 324–334. IEEE, October 1994.

[10] Eric Cooper, Onat Menzilcioglu, Robert Sansom, and Francois Bitz. Host interface design for atm lans. In *Proceedings of the 16th Conference on Local Computer Networks*, pages 247–258. IEEE, October 1991.

[11] Alan Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 106–117. ACM/IEEE, April 1994.

[12] Bruce S. Davie. A Host-Network Interface Architecture for ATM. In *Proceedings of the SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pages 307–315. ACM, September 1991.

[13] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, Oregon, November 1993. ACM/IEEE.

[14] Thomas Gross, Dave O'Hallaron, and Jaspal Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[15] Intel Corporation. *Paragon X/PS Product Overview*, March 1991.

[16] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software support for outboard buffering and checksumming. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, pages 87–98. ACM, August/September 1995.

[17] Corey Kosak, David Eckhardt, Todd Mummert, Peter Steenkiste, and Allan Fisher. Buffer management and flow control in the credit net ATM host interface. In *Proceedings of the 20th Conference on Local Computer Networks*, pages 370–378, Minneapolis, October 1995. IEEE.

[18] H.T. Kung, T. Blackwell, and A. Chapman. Credit update protocol for flow-controlled ATM networks: Statistical mulitplexing and adaptive credit allocation. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, pages 101–114. ACM, August 1994.

[19] H.T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. Parallelizing a new class of large applications over high-speed networks. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–177. ACM, April 1991.

[20] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *19th Annual International Symposium in Computer Architecture*, pages 92–102. IEEE, May 1992.

[21] Andreas Nowatzyk, Gunes Aybay, Michael Browne, Edmund Kelly, David Lee, and Michael Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences - Vol. I: Architecture*, pages 144–153. IEEE, January 1994.

[22] K. K. Ramakrishnan and Peter Neuman. Integration of rate and credit schemes for atm flow control. *IEEE Network Magazine*, 9(2):49–56, March/April 1995.

[23] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):246–260, April 1982.

[24] Peter A. Steenkiste, Brian D. Zill, H.T. Kung, Steven J. Schlick, Jim Hughes, Bob Kowalski, and John Mullaney. A host interface architecture for high-speed networks. In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages A3 1–16, Liege, Belgium, December 1992. IFIP, Elsevier.

[25] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[26] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. Intl. Conf. on Supercomputing*, page accepted, Barcelona, July 1995. ACM.

[27] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report UW-CSE-93-04-03, Computer Science and Engineering, University of Washington, April 12 1993.

[28] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[29] Z. Wang and J. Crowcroft. Seal detects cell misordering. *IEEE Network Magazine*, 6(4):8–9, July 1992.

[30] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceeding Conference on Operating Systems Design and Implementation*, volume Winter, pages 87–100. Usenix, November 1994.