Presented at Hot Interconnects II, Aug. 1994, Palo Alto, CA, abridged version in IEEE Micro, Feb 1995.

Low-Latency Communication over ATM Networks using Active Messages

Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch

Department of Computer Science
Cornell University
Ithaca, NY14850

Abstract

Recent developments in communication architectures for parallel machines have made significant progress and reduced the communication overheads and latencies by over an order of magnitude as compared to earlier proposals. This paper examines whether these techniques can carry over to clusters of workstations connected by an ATM network even though clusters use standard operating system software, are equipped with network interfaces optimized for stream communication, do not allow direct protected user-level access to the network, and use networks without reliable transmission or flow control.

In a first part, this paper describes the differences in communication characteristics between clusters of workstations built from standard hardware and software components and state-of-the-art multiprocessors. The lack of flow control and of operating system coordination affects the communication layer design significantly and requires larger buffers at each end than on multiprocessors. A second part evaluates a prototype implementation of the low-latency Active Messages communication model on a Sun workstation cluster interconnected by an ATM network. Measurements show application-to-application latencies of about 20 microseconds for small messages which is roughly comparable to the Active Messages implementation on the Thinking Machines CM-5 multiprocessor.

1 Introduction

The shift from slow broadcast-based local area networks to high bandwidth switched network architectures is making the use of clusters of workstations[1] as platforms for parallel processing more and more attractive. While a number of software packages [5,6] already support parallel processing on today's workstations and networks, the communication performance is over two orders of magnitude inferior to state-of-the art multiprocessors[2]. As a result, only embarassingly parallel applications (i.e., parallel applications that essentially never communicate) can make use of such environments. Networking technologies such as ATM[1] offer the opportunity to close the gap: for example, ATM cells are roughly the same size as messages on multiprocessors, it takes only a few microseconds to send or receive a cell, ATM switches can be configured to provide bisection bandwidths comparable to parallel machine networks, and routing latencies are on the order of microseconds.[3] However, to date this communication potential has not been available at the application level.

From a purely technical point of view, the gap between clusters of workstations and multiprocessors is certainly closing and the distinction between the two types of systems is becoming blurred. Differences remain: in particular, the design and construction of multiprocessors allows better integration of all the components because

1. The term *cluster* is used here to refer to collections of workstation-class machines interconnected by a low-latency high-bandwidth network.

2. This paper focuses exclusively on scalable multiprocessor architectures and specifically excludes bus-based shared-memory multiprocessors.
3. Current ATM switches have latencies about an order of magnitude higher than comparable multiprocessor networks, however, this difference does not seem to be inherent in ATM networks, at least not for local area switches.

This paper and the described software are available at URL http://www.cs.cornell.edu/Info/Projects/ATM/
Authors' email:{tve,veena,basu,buch}@cs.cornell.edu
This work is supported by an equipment gift from AT&T

they can be designed to fit together. In addition, the sharing of physical components such as power supplies, cooling and cabinets has the potential to reduce cost and to allow denser packaging. While the debate over the significance of these technological differences is still open, it is becoming clear that the two approaches will yield qualitatively similar hardware systems. Indeed, it is possible to take a cluster of workstations and load system software making it look almost identical to a multiprocessor. This means that a continuous spectrum of platforms spanning the entire range from workstations on an Ethernet to state-of-the-art multiprocessors can become available, and that any distinction between multiprocessors and clusters will be more and more arbitrary from a technical point of view.

From a pragmatic point of view, however, significant differences are likely to remain. The most important attraction in using a cluster of workstations instead of a multiprocessor lies in the off-the-shelf availability of all its major hardware and software components. This means that all the components are readily available, they are familiar, and their cost is lower because of economies of scale leveraged across the entire workstation user community. Thus, even if from a technical point of view there is a continuous spectrum between clusters and multiprocessors, the use of off-the-shelf components in clusters will maintain differences.

In fact, the use of standard components in clusters raises the question whether these can be reasonably used for parallel processing. Recent advances in multiprocessor communication performance are principally due to a tighter integration of programming models, compilers, operating system functions, and hardware primitives. It is not clear whether these advances can be carried over to clusters or whether the use of standard components is squarely at odds with achieving the level of integration required to enable modern parallel programming models. Specifically, new communication architectures such as distributed shared memory, explicit remote memory access, and Active Messages reduced the costs from hundreds to thousands of microseconds to just a few dozen precisely through the integration of all system components. These new communication architectures are designed such that network interfaces can implement common primitives directly in hardware, they allow the operating system to be moved out of the critical communication path without compromising protection, and they are well suited for high-level language implementation.

This paper examines whether the techniques developed to improve communication performance in multiprocessors, in particular, Active Messages, can be carried over to clusters of workstations with standard networks and mostly standard system software. This paper assumes the current state of the art technology in which clusters using ATM networks differ from multiprocessors in three major aspects[1]:

- clusters use standard operating system software which implies less coordination among individual nodes, in particular with respect to process scheduling and address translation,
- ATM networks do not provide the reliable delivery and flow control that are taken for granted in multiprocessor networks, and
- network interfaces for workstations optimize stream communication (e.g., TCP/IP) and are less well integrated into the overall architecture (e.g., connect to the I/O bus instead of the memory bus).

In comparing communication on clusters and multiprocessors this paper makes two major contributions:

- first, it analyzes, in Section 2, the implications that the differences between clusters and multiprocessors have on the design of communication layers similar to those used in multiprocessors, and
- second, it describes, in Section 3, the design of an Active Messages prototype implementation on a collection of Sun workstations interconnected by an ATM network which yields application-to-application latencies on the order of 20μs.

The use of Active Messages in workstation clusters is briefly contrasted to other approaches in Section 4 and Section 5 concludes the paper.

2 Technical Issues

Collections of workstations have been used in many different forms to run large applications. In order to establish a basis for comparison to multiprocessors, this paper limits itself to consider only collections of workstations (called clusters) which consist of a homogeneous set of machines, dedicated to run parallel applications, located in close proximity (such as in the same machine room), and interconnected by an ATM network. Such a cluster can be employed in a large variety of settings. The cluster could simply provide high-performance compute service for a user community to run large parallel applications.

A more typical setting would be as computational resource in a distributed application. One such example, the Stormcast weather monitoring system in Norway, runs on a very large collection of machines spread across a large portion of the country, but uses a cluster of a few dozen workstations in a machine room (without high speed network in this case) to run compute-intensive weather prediction models and to emit storm warn-

1. A discussion of differences in fault isolation characteristics is beyond the scope of this paper.

ings. The availability of low-latency communication among these workstations would enable the use of parallel programming languages and of more powerful parallel algorithms, both of which require a closer coupling among processors than is possible today.

Concentrating on the compute cluster offers the largest potential for improvement because the latency over the long-haul links is dominated by speed-of-light and network congestion issues and because the wide area communication is comparatively better served by today's distributed computing software. Note that this paper does not argue that running concurrent applications in a heterogeneous environment, across large distances, and on workstations that happen to be sitting idle is not an interesting design point (it in fact has been used successfully), but that the set of communication issues occurring in such a context cannot be compared to those in a multiprocessor.

Given that the applications for clusters considered here exhibit characteristics similar to those on multiprocessors, the programming models used would be comparable, if not identical, to those popular for parallel computing. This includes various forms of message passing (e.g., send/receive, PVM), of shared memory (e.g., cache coherent shared memory, remote reads and writes, explicit global memory), and of parallel object oriented languages (e.g., numerous C++ extensions).

On parallel machines several proposed communication architectures have achieved the low overheads, low latencies, and high bandwidths that are required for high performance implementations of the above programming models. In particular, cache coherent shared memory, remote reads and writes, and Active Messages offer round-trip communication within a few hundred instruction times, so that frequent communication on a fine granularity (such as on an object by object or cache line basis) remains compatible with high performance. In these settings, the overhead of communication, that is, the time spent by the processor initiating communication, is essentially the cost of pushing message data into the network interface at the sending end and pulling it out at the receiving end. Virtually no cycles are spent in any protocol handling as all reliability and flow control are handled in hardware. The operating system need not be involved in every communication operation because the network interface hardware can enforce protection boundaries across the network.

The above communication architectures cannot be moved in a straightforward manner from multiprocessors to clusters of workstations with ATM networks because of three major differences between the two: ATM networks offer neither reliable delivery nor flow control, ATM network interfaces provide no support for protected user-level access to the network, and the workstation operating systems do not coordinate process scheduling or address translation globally. Coping with these differences poses major technical challenges and may eventually require the integration of some multiprocessor-specific features into the clusters. The following three subsections present the nature of these differences in more detail and discuss the resulting issues.

2.1 Reliability and flow control in the network

In multiprocessor networks, flow control is implemented in hardware on a link-by-link basis. Whenever the input buffer of a router fills up, the output of the upstream router is disabled to prevent buffer overflow. The flow control thus has the effect of blocking messages in the network and eventually, as the back-pressure propagates, the sending nodes are prevented from injecting further messages. This mechanism guarantees that messages are never dropped due to buffer space limitations within the network or at the receiving end. In addition, the electrical characteristics of the network are designed to ensure very low error rates, such that the use of a simple error detection and correction mechanism (implemented in hardware) can offer the same reliability within the network as is typical of the processing nodes themselves.

In contrast, an ATM network does not provide any form of flow control and does not offer reliable delivery. Instead, higher protocol layers must detect cell loss or corruption and cause their retransmission. While this partitioning of responsibilities may be acceptable in the case of stream-based communication (e.g., TCP/IP, video, audio) it is questionable in a parallel computing setting.

The flow control and the error detection and correction in multiprocessor networks serve to cover four causes of message loss: buffer overflow in the receiving software, buffer overflow in the receiving network interface, buffer overflow within the network, and message corruption due to hardware errors. In an ATM network, simple window based end-to-end flow control schemes and a per-message CRC (as used in AAL-5) can cover the first and last cases[1] of cell loss. In addition, preventing buffer overflow in the receiving network interface can be achieved by ensuring that the rate at which cells can be moved from the interface into main memory is at least as large as the maximal cell arrival rate. Preventing buffer overflow within the network, however, is not realistically possible using end-to-end flow control. This is particularly a problem in a parallel computing setting in which all nodes tend to communicate with all other nodes in both highly regular and irregular patterns at

1. Although some transmission media may cause burst errors which cannot be corrected by most CRC codes.

unpredictable intervals. The degree of contention within the network therefore cannot be measured or predicted with any accuracy by either the sender or the receiver and communication patterns which result in high contention will result in high cell loss rates causing extensive retransmissions.

Traditional flow control schemes used in stream-based communication avoid fruitless retransmission storms by dynamically reducing the transmission rate on connections which experience high cell loss rates. This works in these settings because, following the law of large numbers, contention in a wide area network does not tend to vary instantaneously and therefore the degree of contention observed in the recent past is a good predictor for contention in the near future.

As an illustration of the difficulties in a parallel computing setting, consider the implementation of a parallel sort. The most efficient parallel sort algorithms [3] are based on an alternation of local sorts on the nodes and permutation phases in which all nodes exchange data with all other nodes. These permutation phases serve to move the elements to be sorted "towards" their correct position, The communication patterns observed are highly dynamic and their characteristics depend to a large degree on the input data. If at any point the attempted data rate into a given node exceeds the link rate, then the output buffers at up-stream switches will start filling up. Because the communication patterns change very rapidly (essentially with every cell), it is futile to attempt to predict contention, and given the all-to-all communication pattern, the probability of internal contention among seemingly unrelated connections is high.

Beyond the problems caused by contention and the resulting retransmissions, the lack of reliable delivery guarantee in ATM networks imposes a certain overhead on the communication primitives. Specifically, the sender must keep a copy of each cell sent until a corresponding acknowledgment is received, in case the cell must be retransmitted. This means that messages cannot be transferred directly between processor registers and the network interface (as is possible on the CM-5 [12]), rather, a memory copy must be made as well.

2.2 User-level access to the network interface

Recently, multiprocessor communication architectures have achieved a significant reduction of the communication overhead by eliminating the operating system from the critical path. In order not to compromise security, the network interface must offer some form of protection mechanism. In shared memory models, the memory management unit is extended to map remote memory into the local virtual user address space such that the operating system can enforce security by managing the address translation tables. Message-based network interfaces contain a node address translation table which maps the user's virtual node numbers onto the physical node address space. Again, the operating system enforces security by controlling the address translation, thereby preventing a process from sending a message to an arbitrary node. The current generation of message based network interfaces only control the destination node address and therefore require that all processes of a parallel program run at the same time. The next generation adds the sending process id to each message allowing the receiving network interface to discriminate between messages destined for the currently running process, that can retrieve these message directly, and messages for dormant processes, which must be queued (typically by the operating system) for later retrieval.

In contrast, the network interfaces available for workstations do not yet incorporate any form of protection mechanism. Instead, the operating system must be involved in the sending and reception of every message. The connection based nature of ATM networks would principally allow the design of a protection mechanism to limit the virtual circuits a user process has access to (the operating system would still control virtual circuit set-up). But because the architecture of the networking layers in current operating systems does not seem to be set-up to allow user-level network interface access, it appears unlikely that network interfaces with these features will become commonplace soon. The challenge in any high-performance communication layer for clusters is, thus, to minimize the path through the kernel by judiciously coordinating the user-kernel interactions.

2.3 Coordination of system software across all communicating nodes

In almost all communication architectures the message reception logic is the critical performance bottleneck. In order to be able to handle incoming messages at full network bandwidth, the processing required for each arriving message must be minimized carefully. The trick used in multiprocessor systems to ensure rapid message handling is to constrain the sender to only send messages which are easy to handle.

In shared memory systems this is done by coordinating the address translation tables among all processing nodes such that the originating node can translate the virtual memory address of a remote access and directly place the corresponding physical memory address into the message. The set of communication primitives is small and fixed (e.g., read and write) and by forcing the sender to perform the complicated part of a remote memory access (namely the protection checks and the address translation) the handling of a request is relatively simple to implement[1]. If the virtual address were sent, the receiving node could discover that the requested virtual memory location had been paged out to disk with the result that the handling of the message would become rather involved.

In Active Messages on multiprocessors the scheduling of processes is assumed to be coordinated among all nodes such that communicating processes execute simultaneously on their respective nodes. This guarantees that messages can be handled immediately on arrival by the destination process itself. In order to accomplish this, the sender of an Active Message specifies a user-level handler at the destination whose role it is to extract the message from the network and integrate it into the ongoing computation. The handler can also implement a simple remote service and send a reply Active Message back. However, in order to prevent deadlock the communication patterns are limited to requests and replies, e.g., a handler of a reply message is not allowed to send any further messages. An implementation of Active Messages typically reserves the first word of each message for the handler address, and the handler at the receiving end is dispatched immediately on message arrival to dispose of the message. The fact that the message layer can call upon the handlers to deal with messages in FIFO order simplifies the buffering considerably over that required by more traditional message passing models such as PVM, MPI, or NX. These models allow processes to consume messages in arbitrary order and at arbitrary times forcing the communication architecture to implement very general buffer and message matching mechanisms at high cost.

In clusters the fact that the operating systems of the individual nodes are not nearly as coordinated contradicts the assumption that messages can always be consumed quickly upon arrival. In the case of Active Messages the destination process might have been suspended and cannot run the handler, and in a shared memory model the memory location requested might not be mapped. Although exact coordination is not possible without major changes to the operating system core, an implementation of either communication model is likely to be able to perform some coordination among nodes on its own and to influence the local operating system accordingly. This may allow the communication layer to assume that in the common case everything works out fine, but it must be able to handle the difficult cases as well.

2.4 Summary

Even though superficially a cluster of workstations appears the be technically comparable to a multiprocessor, the reality is that key characteristics are different and cause significant implementation difficulties: the very comparable raw hardware link bandwidths, bisection bandwidths, and routing latencies conceal the lack in clusters of flow control, reliability, user-level network access, and operating system coordination.

These shortcomings will inevitably result in lower communication performance; their quantitative effect on performance is evaluated in the next section which presents a prototype implementation of Active Messages on a cluster of Sun workstations. However, the lack of flow-control in ATM networks poses a fundamental problem: can catastrophic performance degradation occur due to significant cell loss in particular communication patterns?

1. Cache coherent shared memory stretch this characterization given that the cache in the receiving node essentially performs another address translation which may miss and require additional communication with other nodes to complete the request.

3 SSAM: a SPARCstation Active Messages Prototype

The SSAM prototype implements the critical parts of an Active Messages communication architecture on a cluster of SPARCstations connected by an ATM network. The primary goal is to evaluate whether it is possible to provide a parallel programming environment on the cluster that is comparable to those found on multiprocessors. The prototype is primarily concerned with providing performance at par with parallel machines, while addressing the handicaps of ATM networks that have been identified in the previous section. In particular:

- the prototype provides reliable communication to evaluate the cost of performing the necessary flow-control and error checking in software,
- it minimizes the kernel intervention to determine the cost of providing protection in software, and
- the buffering is designed to tolerate arbitrary context switching on the nodes.

At this time only a limited experimental set-up (described below) is available such that the prototype cannot provide information neither on how cell losses due to contention within the network affect performance, nor on how the scheduling of processes can be coordinated to improve the overall performance of parallel applications.

3.1 Active Messages Communication Architecture

The Active Messages communication architecture [4] offers simple, general purpose communication primitives as a thin veneer over the raw hardware. It is intended to serve as a substrate for building libraries that provide higher-level communication abstractions and for generating communication code directly from a parallel-language compiler. Unlike most communication layers, it is not intended for direct use by application programmers and really provides lower-level services from which communication libraries and run-time systems can be built.

The basic communication primitive is a message with an associated small amount of computation (in the form of a handler) at the receiving end. Typically the first word of an Active Message points to the handler for that message. On message arrival, the computation on the node is interrupted and the handler is executed. The role of the handler is to get the message out of the network, by integrating it into the ongoing computation and/or by sending a reply message back. The buffering and scheduling provided by Active Messages are extremely primitive and thereby fast: the only buffering is that involved in actual transport and the only scheduling is that required to activate the handler. This is sufficient to support many higher-level abstractions and more general buffering and scheduling can be easily constructed in layers above Active Messages when needed. This minimalist approach avoids paying a performance penalty for unneeded functionality.

In order to prevent deadlock and livelock, Active Message restricts communication patterns to requests and replies, i.e., the handler of a request message is only allowed to send a reply message and a reply handler is not allowed to send further replies.

3.1.1 SSAM functionality

The current implementation is geared towards the sending of small messages which fit into the payload of a single ATM cell. Eight of the 48 available bytes of payload in an ATM cell are used by SSAM to hold flow-control information (16 bits), the handler address (32 bits), and an AAL3/4 compatible checksum (16 bits). The remaining 40 bytes hold the Active Message data.

The C header file for the interface to SSAM is shown in Figure 1. To send a request Active Message, the user places the message data into a per-connection buffer provided by SSAM and calls `SSAM_10` with a connection identifier and the remote handler address. `SSAM_10` adds the flow-control information and traps to the kernel to have the message injected into the network. It also polls the receiver and processes incoming messages. At the receiving end, the network is polled by `SSAM_10` or `SSAM_poll` (the latter only polls the network) and all messages accumulated in the receive FIFO are moved into a buffer. SSAM then calls the appropriate handler for each message, passing as arguments the originating connection identifier, the address of the buffer holding the message, and the address of a buffer for a reply message. The handler processes the message and may send a reply message back by placing the data in the buffer provided and returning the address of the reply handler (or `NULL` if no reply is to be sent).

The current prototype does not use interrupts, instead, the network is polled every time a message is sent. This means that as long as a process is sending messages it will also handle incoming ones. An explicit polling function is provided for program parts which do not communicate. Using interrupts is planned but not implemented yet.

3.1.2 Example: implementing a remote read with SSAM

The sample implementation of a split-phase remote double-word read is shown in Figure 2. The `readDouble` function increments a counter of outstanding reads, formats a request Active Message with the address to be read as well as information for the reply, and sends the message. The `readDouble_h` handler fetches the remote location and sends a reply back to the `readDouble_rh` reply handler which stores the data into memory and decrements the counter. The originating processor waits for the completion of the read by busy-waiting on the counter at the end of `readDouble`. A split-phase read could be constructed easily by exposing the counter to the caller, who could proceed with computation after initiating the read and only wait on the counter when the data is required.

3.2 Experimental set-up

The experimental set-up used to evaluate the performance of the prototype SSAM implementation consists of a 60Mhz SPARCstation-20 and a 25Mhz SPARCstation-1+ running SunOS 4.1. The two machines are connected via Fore Systems SBA-100 ATM interfaces using a 140Mb/s TAXI fiber. The interfaces are located on the Sbus (a 32-bit I/O bus running at 20 or 25Mhz) and pro-

```
/* ssam.h - SPARCstation ATM Active Messages */

/* Initialize Active Messages */
extern int SSAM_init(void);

/* Active Message handlers */
typedef void (*SSAM_reply_handler)
   (int connection, void *in_buf);

typedef SSAM_reply_handler
   (*SSAM_req_handler)(int connection,
   void *in_buf, void *reply_buf);

/* Buffers to send messages */
#define SSAM_MAXCONN (32)

extern void *SSAM_reqbuf[SSAM_MAXCONN];

extern void SSAM_10(int connection,
 SSAM_req_handler handler);

/* Poll the network explicitly */
extern void SSAM_poll(void);
```

Figure 1: C interface for SPARCstation Active Messages

```
/* Remote read of 32 bytes */
static volatile int read_cnt = 0;
typedef struct {
        double  *src, *dest;
        double  data[4];
} read32_msg;
/* Read 32 bytes from remote node */
void read32(int conn, double *src,
            double *dest)
{ read32_msg *out = SSAM_req_buf[conn];
  out->src = src; out->dest = dest;
  read_cnt++;
  SSAM_10(conn, read32_h);
  while(read_cnt) SSAM_poll();
}
/* Read request handler */
static SSAM_reply_handler
read32_h(int conn, read32_msg *in,
         read32_msg *out)
{ double *src = in->src;
  out->dest = in->dest;
  if(((long)src&7) == 0) {
    out->data[0] = src[0];
    out->data[1] = src[1];
    out->data[2] = src[2];
    out->data[3] = src[3];
  } else {
    /* non double-word aligned code omitted */
  }
  return read32_rh;
}
/* Read reply handler */
static void
read32_rh(int conn, read32_msg *in)
{ double *dest = in->dest;
  if(((long)dest&7) == 0) {
    dest[0] = in->data[0];
    dest[1] = in->data[1];
    dest[2] = in->data[2];
    dest[3] = in->data[3];
  } else {
    /* non double-word aligned code omitted */
  }
  read_cnt--;
}
```

Figure 2: Sample remote read implementation using SSAM

vide a 36-cell deep output FIFO as well as a 292-cell input FIFO. To send a cell the processor stores 56 bytes into the memory-mapped output FIFO and to receive a cell it reads 56 bytes from the input FIFO. A register in the interface indicates the number of cells available in the input FIFO.

Note that the network interface used is much simpler and closer to multiprocessor NIs than most second-generation ATM interfaces available today. The only function performed in hardware, beyond simply moving cells onto/off the fiber, is checksum generation and checking for the ATM header and an AAL3/4 compatible payload. In particular, no DMA or segmentation and reassembly of multi-cell packets is provided.

3.3 SSAM implementation

The implementation of the SPARCstation ATM Active Messages layer consists of two parts: a device driver which is dynamically loaded into the kernel and a user-level library to be linked with applications using SSAM. The driver implements standard functionality to open and close the ATM device and it provides two paths to send and receive cells. The fast path described here consists of three trap instructions which lead directly to code for sending and receiving individual ATM cells. The traps are relatively generic and all functionality specific to Active Messages is in the user-level library which also performs the flow-control and buffer management. A conventional read/write system call interface is provided for comparison purposes and allows to send and receive cells using a "pure" device driver approach.

The traps to send and receive cells consist of carefully crafted assembly language routines. Each routine is small (28 and 43 instructions for the send and receive traps, respectively) and uses available registers carefully. The register usage is simplified by the Sparc architecture's use of a circular register file, which provides a clean 8-register window for the trap. By interfacing from the program to the traps via a function call, arguments can be passed and another 8 registers become available to the trap.

The following paragraphs describe the critical parts of the SSAM implementation in more detail.

3.3.1 Flow-control

A simple sliding window flow control scheme is used to prevent overrun of the receive buffers and to detect cell losses. The window size is dimensioned to allow close to full bandwidth communication among pairs of processors.

In order to implement the flow control for a window of size w, each process pre-allocates memory to hold $4w$ cells per every other process with which it communicates. The algorithm to send a request message polls the receiver until a free window slot is available and then injects the cell into the network, saving it in one of the buffers as well in case it has to be retransmitted. Upon receipt of a request message, the message layer moves the cell into a buffer and, as soon as the corresponding process is running, calls the Active Message handler. If the handler issues a reply, it is sent and a copy is held in a buffer. If the handler does not generate a reply, an explicit acknowledgment is sent. Upon receipt of the reply or acknowledgment, the buffer holding the original request message can be reused. Note how the distinction between requests and replies made in Active Messages allows acknowledgments to be piggy-backed onto replies.

The recovery scheme used in case of lost or duplicate cells is standard, except that the reception of duplicate request messages may indicate lost replies which have to be retransmitted. It is important to realize that this flow control mechanism does not really attempt to minimize message losses due to congestion within the network: the lack of flow-control in ATM networks effectively precludes any simple congestion avoidance scheme. Until larger test-beds become available and the ATM community agrees on how routers should handle buffer overflows it seems futile to invest in more sophisticated flow-control mechanisms. Nevertheless, the bursty nature of parallel computing communication patterns are likely to require some solution before the performance characteristics of an ATM network become as robust as those of as multiprocessor networks.

3.3.2 User-kernel interface and buffer management

The streamlining of the user-kernel interface is the most important factor contributing to the performance of SSAM. In the prototype, the kernel preallocates all buffers for a process when the device is opened. The pages are then pinned to prevent page-outs and are mapped (using mmap) into the processes' address space. After every message send, the user-level library chooses a buffer for the next message and places a pointer in an exported variable. The application program moves the message data into that buffer and passes the connection id and the handler address to SSAM which finishes formatting the cell (adding the flow control and handler) and traps to the kernel. The trap passes the message offset within the buffer area and the connection id in registers to the kernel. Protection is ensured with simple masks to limit the connection id and offset ranges. A lookup maps the current process and connection ids to a virtual circuit. The kernel finally moves the cell into the output FIFO.

At the receiving end, the read-ATM kernel trap delivers a batch of incoming cells into a pre-determined shared memory buffer. The number of cells received is returned in a register. For each cell the kernel performs four tasks: it loads the first half of the cell into registers, uses the VCI to index into a table to obtain the address of the appropriate processes' input buffer, moves the full cell into that buffer, and checks the integrity of the cell using three flag bits set by the NI in the last byte. Upon return from the trap, the SSAM library loops through all received cells checking the flow-control information, calling the appropriate handlers for request and reply messages, and sending explicit acknowledgments when needed.

3.4 SSAM performance

The following paragraphs describe performance measurements of SSAM made with a number of synthetic benchmarks. The following terminology is used: *overhead* consists of the processor cycles spent preparing to send or receive a message, *latency* is the time from which a message send routine is called to the time the message is handled at the remote end, and *bandwidth* is the rate at which user data is transferred. The performance goal for SSAM is the fiber rate of 140Mbit/s which transmits a cell every 3.14µs (53+2 bytes) for an ATM payload bandwidth of 15.2MB/s[1].

3.4.1 ATM traps

A detailed cost breakdown for the operations occurring in each of the traps to send and receive cells is shown in Table 1. The two timing columns refer to measurements taken on the SPARCstation 1+ and on the SPARCstation 20, respectively. The times have been obtained by measuring repeated executions of each trap with `gettimeofday` which uses a microsecond-accurate clock and takes 9.5µs on the SS-20. The time breakdown for each trap was measured by commenting appropriate instructions out and is somewhat approximate due to the pipeline overlap occurring between successive instructions.

The write trap cost is broken down into 5 parts: the cost of the trap and return, the protection checks, overhead for fetching addresses, loading the cell into registers, and pushing the cell into the network interface. The SS-20 numbers show clearly that the fiber can be saturated by sending a cell at a time from user level. It also indicates that the majority of the cost (75%) lies in the access to the network interface across the Sbus. The cost of the trap itself is surprisingly low, even though it is the second largest item. In fact, it could be reduced slightly as the current implementation adds a level of indirection in the trap dispatch to simplify the dynamic loading of the device driver.[2]

The read trap is itemized similarly: the cost to trap and return, fetching the device register with the count of available cells, additional overhead for setting-up addresses, loading the cell from the network interface,

1. All bandwidths are measured in megabytes per second.
2. The kernel write-protects the trap vectors after boot-up. The SSAM prototype uses a permanently loaded trap which performs an indirect jump via a kernel variable to allow simple dynamic driver loading.

demultiplexing among processes, and storing the cell away. The total cost shows a trap which receives a single cell, as well as the per-cell cost for a trap which receives 16 cells. Here again the access to the device dominates due to the fact that each double-word load incurs the full latency of an Sbus access. The total time of 4.61µs on the SS-20 falls short of the fiber's cell time and will limit the achievable bandwidth to at most 68% of the fiber.

The write-read trap first sends a cell and then receives a chunk of cells. This amortizes the cost of the trap across both functions and overlaps checking the cell count slightly with sending. The last item in the table shows the cost of a null system call for comparison purposes (a write to file descriptor -1 was used). It is clear that a system call approach would yield performance far inferior to the traps and would achieve only a fraction of the fiber bandwidth.

3.4.2 ATM read/write system calls

In addition to the direct traps, the device driver allows cells to be sent and received using traditional read and write system calls on the device file descriptor. At this time this conventional path is provided for comparison purposes only and the read and write entry points into the device driver are limited to sending and receiving single cells. Multi-cell reads and writes could be supported easily. The read and write entry points perform the following operations:

- check for the appropriateness of the file descriptor,
- transfer data between user space and an internal buffer using `uiomove`, and
- transfer data between the internal buffer and the FIFOs of the network interface.

Operation	SS-20	SS-1+
write trap		
trap+rett	0.44µs	2.03µs
check pid and connection id	0.05µs	0.49µs
addt'l kernel ovhd	0.05µs	0.50µs
load cell to push	0.13µs	3.87µs
push cell to NI	2.05µs	3.17µs
total	2.72µs	10.11µs
read trap		
trap+rett	0.44µs	2.03µs
check cell count	0.81µs	1.08µs
addt'l kernel ovhd	0.18µs	0.80µs
per cell pull from NI	4.27µs	3.68µs
per cell demux	0.09µs	0.23µs
per cell store away	0.17µs	3.50µs
total for 1 cell	5.87µs	11.32µs
per cell total for 16 cells	4.61µs	8.08µs
write_read trap		
total, 0 cells read	3.7µs	11.2µs
total, 1 cell read	8.2µs	21.4µs
null system call	6.9µs	40µs

Table 1: Cost breakdown for traps to send and receive cells.

Operation	SS-20	SS-1+
write system call		
syscall overhead	22.6µs	100µs
check fd, do `uiomove`	3.4µs	16µs
push cell into NI	2.2µs	8µs
write total	28.2µs	124µs
read system call		
syscall overhead	22.1µs	99µs
pull cell from NI	5.0µs	13µs
check fd and recv ready, do uiomove	7.0µs	25µs
read total for 1 cell	34.1µs	137µs
read total for 0 cells	28.8µs	113µs

Table 2: Cost of sending and receiving cells using read and write system calls.

The internal buffer is used because the data cannot be transferred directly between user space and the device using `uiomove` due to the fact that the device FIFOs are only word addressable. The use of an internal buffer also allows double-word accesses to the device FIFOs, which improves the access times considerably.

Table 2 shows the costs for the various parts of the read and write system calls. The "syscall overhead" entries reflect the time taken for a read (respectively write) system call with an empty read (write) device driver routine. This measures the kernel overhead associated with these system calls. The "check fd, do `uiomove`" entry reflects the time spent in checking the validity of the file descriptor and performing the `uiomove`. In the case of a read, it also includes the time to check the device register holding the number of cells available in the input FIFO. The "push/pull cell" entries reflect the time spent to transfer the contents of one cell between the internal buffer and the device FIFOs. The "write" and "read 1 cell" totals reflect the cost of the full system call, while the "read 0 cells" entry is the time taken for an unsuccessful poll which includes the system call overhead, the file descriptor checks, and the reading of the receive-ready register.

The timings show clearly that the overhead of the read/write system call interface is prohibitive for small messages. For larger messages, however, it may well be a viable choice and it is more portable than the traps.

3.4.3 SSAM

Measurements of the Active Messages layer built on the cell send and receive traps are shown in Table 3. In all cases one word of the Active Message payload carries data and the handlers simply return. The send request uses a write-read-trap and adds a little over 1µs of overhead (on the SS-20) for cell formatting and flow-control. The handling times are all roughly the cost of a read-trap (reading 16 cells per trap) plus again a little over 1µs for the flow control and handler dispatch. If a reply is sent that adds the time of a write-trap.

The measurements show that supporting only single-cell Active Messages is not optimal. Longer messages are required to achieve peak bulk transfer rates: the one-cell-at-a-time prototype can yield up to 5.6MB/s. A simpler interface for shorter messages (e.g., with only 16 bytes of payload) might well be useful as well to accelerate the small requests and acknowledgments that are often found in higher-level protocols. Unfortunately, given that the trap cost is dominated by the network interface access time and that the SBA-100 requires all 56 bytes of a cell to be transferred by the processor, it is unlikely that a significant benefit can be realized.

3.4.4 Split-C

While a full implementation of Split-C [2] is still in progress, timings of the remote memory access primitives show that the round-trip time for a remote read of 32 double-word aligned bytes takes 32µs on the SS-20 and a one-way remote store takes 22µs for the same payload.[1] Remote accesses with smaller payloads are not noticeably cheaper. A bulk write implemented with the current SSAM layer transfers 5.5Mbytes/s, but experiments show that, using long messages, this could be improved to 9Mbytes/s by using the full ATM payload and simplifying the handling slightly.

Operation	SS-20	SS-1+
send request	5.0µs	15µs
handle request, no reply sent	5.6µs	15µs
handle request and send reply	7.7µs	25µs
handle ack	5.0µs	11µs
handle reply	5.2µs	12µs

Table 3: Cost breakdown for SPARCstation Active Messages.

1. Note that in a more realistic setting a Fore ASX-100 switch will add roughly 10µs of latency to the write time and 20µs to the round-trip read time [7].

3.5 Unresolved issues

The current SSAM prototype has no influence on the kernel's process scheduling. Given the current buffering scheme the SSAM layer operation is not influenced by which process is running. The performance of applications, however, is likely to be highly influenced by the scheduling. How to best influence the scheduler in a semi-portable fashion requires further investigation. The most promising approach appears to be to use real-time thread scheduling priorities, such as are available in Solaris 2.

The amount of memory allocated by the SSAM prototype is somewhat excessive and, in fact, for simplicity, the current prototype uses twice as many buffers as strictly necessary. For example, assuming that a flow-control window of 32 cells is used, the kernel allocates and pins 8Kbytes of memory per process per connection. On a 64-node cluster with 10 parallel applications running, this represents 5Mb of memory per processor.

The number of preallocated buffers could be reduced without affecting peak bulk transfer rates by adjusting the flow control window size dynamically. The idea is that the first cell of a long message contain a flag which requests a larger window size from the receiver; a few extra buffers would be allocated for this purpose. The receiver grants the larger window to one sender at a time using the first acknowledgment cell of the bulk transfer. The larger window size remains in effect until the end of the long message. This scheme has two benefits: the request for a larger window is overlapped with the first few cells of the long message, and the receiver can prevent too many senders from transferring large data blocks simultaneously, which would be sub-optimal for the cache. However, fundamentally, it appears that memory (or, alternatively, low performance) is the price to pay for having neither flow-control in the network nor coordinated process scheduling.

A more subtle problem having to do with the ATM payload alignment used by the SBA-100 interface will surface in the future: the 53 bytes of an ATM cell are padded by the SBA-100 to 56 bytes and the 48-byte payload starts with the 6th byte, i.e., it is only half-word aligned. The effect is that bulk transfer payload formats designed with the SBA-100 in mind (and supporting double-word moves of data between memory and the SBA-100) will clash with other network interfaces which double-word align the ATM payload.

3.6 Summary

The prototype Active Messages implementation on a SPARCstation ATM cluster provides a preliminary demonstration that this communication architecture developed for multiprocessors can be adapted to the peculiarities of the workstation cluster. The performance achieved is roughly comparable to that of a multiprocessor such as the CM-5 (where the one-way latency is roughly 6µs), but it is clear that without a network interface closer to the processor the performance gap cannot be closed.

The time taken by the flow-control and protection in software is surprisingly low (at least in comparison with the network interface access times). The cost, in effect, has been shifted to large pre-allocated and pinned buffers. While the prototype's memory usage is somewhat excessive, other schemes with comparable performance will also require large buffers.

Overall, SSAM's speed comes from a careful integration of all layers, from the language level to the kernel traps. The key issues are avoiding copies by having the application place the data directly where the kernel picks it up to move it into the device and by passing only easy to check information to the kernel (in particular not pass an arbitrary virtual address).

4 Comparison to other approaches

The ATM network communication layer most directly comparable to SSAM is the remote memory access model proposed by Thekkath *et. al.* [10,11]. The implementation is very similar to SSAM in that it uses traps for reserved opcodes in the MIPS instruction set to implement remote read and write instructions.[1]

The major difference between the two models is that the remote memory operations separate data and control transfer while Active Messages unifies them. With remote memory accesses data can be transferred to user memory by the kernel without the corresponding process having to run. But the model used does not allow remote reads and writes to the full address space of a process. Rather, each communicating process must allocate special communication memory segments which are pinned by the operating system just as the buffers used by SSAM are. The communication segments are more flexible than SSAM's buffers in that they can directly hold data structures (limited by the fact that the segments are pinned).

The advantage of SSAM over the remote memory accesses is the coupling of data and control: each message causes a small amount of user code to be executed, which allows data to be scattered into complex data structures and the scheduling of computation to be directly influenced by the arrival of data. In the remote memory access model a limited control transfer is offered through per-segment notification flags in order to to cause a file descriptor to become ready.

1. One could easily describe the traps employed by SSAM as additional emulated communication instructions.

Operation	SSAM	Remote mem access
read latency	32µs	45µs
write latency	22µs	30µs
addt'l control transfer ovhd	none	260µs
block write	5.5MB/s	4.4MB/s

Table 4: Comparison of SSAM to Remote Memory Accesses between 2 DECstation 5000s over ATM [11].

Finally, SSAM provides a reliable transport mechanism while the remote memory access primitives are unreliable and do not provide flow-control.

Table 4 compares the performance of the two approaches: Thekkath's implementation uses two DECstation 5000 interconnected by a Turbochannel version of the same Fore-100 ATM interface used for SSAM and performs a little worse than SSAM for data transfer and significantly worse for control transfer. The remote reads and writes are directly comparable in that they transfer the same payload per cell.

The performance of more traditional communication layers over an ATM network has been evaluated by Lin *et. al.* [7] and shows over two orders of magnitude higher communication latencies than SSAM offers. Table 5 summarizes the best round-trip latencies and one-way bandwidths attained on Sun 4/690's and SPARCstation 2's connected by Fore SBA-100 interfaces without switch. The millisecond scale reflects the costs of the traditional networking architecture used by these layers, although it is not clear why Fore's AAL/5 API is slower than the read/write system call interface described in §3.4.2. Note that a TCP/IP implementation with a well-optimized fast-path should yield sub-millisecond latencies.

Communication layer	Round-trip latency	Peak bandwidth
Fore AAL/5 API	1.7ms	4MB/s
BSD TCP/IP Sockets	3.9ms	2MB/s
PVM over TCP/IP	5.4ms	1.5MB/s
Sun RPC	3.9ms	1.6MB/s

Table 5: Performance of traditional communication layers on Sun4/690s and SPARCstation 2s over ATM [7].

5 Conclusions

The emergence of high-bandwidth low-latency networks is making the use of clusters of workstations attractive for parallel computing style applications. From a technical point of view a continuous spectrum of systems can be conceived, ranging from collections of Ethernet-based workstations to tightly integrated custom multiprocessors. However, this paper argues that clusters will be characterized by the use of off-the-shelf components, which will handicap them with respect to multiprocessors in which hardware and software are customized to allow a tighter integration of the network into the overall architecture.

The use of standard components, and in particular, of ATM networking technology, results in three major disadvantages of clusters with respect to multiprocessors: (i) ATM networks do not offer reliable delivery or flow control, (ii) the current network interfaces are not well integrated into the workstation architecture, and (iii) the operating systems on the nodes of a cluster do not coordinate process scheduling or address translations.

The prototype implementation of the Active Messages communication model described in this paper achieves two orders of magnitude better performance than traditional networking layers. Table 6 shows that the resulting communication latencies and bandwidths are in the same ball-park as on state-of-the-art multiprocessors. Key to the success are the use of large memory buffers and the careful design of a lean user-kernel interface. The major obstacle towards closing the remaining performance gap is the slow access to the network interface across the I/O bus, and reducing the buffer memory usage requires coordination of process scheduling across nodes. While taking care of flow control in software does not dominate performance in this study, the behavior of ATM networks under parallel computing communication loads remains an open question.

Machine	Peak bandwidth	Round-trip latency
SP-1 + MPL/p [9]	8.3MB/s	56µs
Paragon + NX [8]	73MB/s	44µs
CM-5 + Active Mesg [4]	10MB/s	12µs
SS-20 cluster + SSAM	5.6MB/s	32µs

Table 6: Comparison of SSAM's performance with that of recent parallel machines.

6 Bibliography

[1] CCITT. *Recommendation I.150: B-ISDN ATM functional characteristics.* (Revised version), Geneva: ITU 1992.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. *Introduction to Split-C.* In Proc. of Supercomputing '93.

[3] D. E. Culler, A. Dusseau, R. Martin, K. E. Schauser. *Fast Parallel Sorting: from LogP to Split-C.* In Proc. of WPPP '93, July 93.

[4] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation.* In Proc. of the 19th ISCA, pages 256-266, May 1992.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Technical Report ORNL/TM-12187, February 1993.

[6] K. Li and P. Hudak. *Memory Coherence in Shared Virtual Memory Systems.* ACM Transactions on Computer Systems, 7(4):321-359, November 1989.

[7] M. Lin, J. Hsieh, D. H. C. Du, J. P. Thomas, and J. A. MacDonald. *Distributed Network Computing over Local ATM Networks.* IEEE Journal on Selected Areas in Communications, Special Issue on ATM LANs, to appear, 1995.

[8] P. Pierce and G. Regnier. *The Paragon Implementation of the NX Message Passing Interface.* In Proc. of SHPCC '94, May 1994.

[9] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. *The SP1 High-Performance Switch.* In Proc. of SHPCC '94, May 1994.

[10] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. *Efficient Support for Multicomputing on ATM Networks.* University of Washington, Technical Report 93-04-03, April 1993.

[11] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. *Separating Data and Control Transfer in Distributed Operating Systems.* In Proc. of the 6th Int'l Conf. on ASPLOS, To appear, October 1994.

[12] Thinking Machines Corporation, Cambridge, Massachusetts. *Connection Machine CM-5, Technical Summary,* November 1992.