Efficient Support for Multicomputing on ATM Networks

Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 93-04-03
April 12, 1993

Efficient Support for Multicomputing on ATM Networks

Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska
Department of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195

April 12, 1993

Abstract

The emergence of a new generation of networks will dramatically increase the attractiveness of loosely-coupled multicomputers based on workstation clusters. The key to achieving high performance in this environment is efficient network access, because the cost of remote access dictates the granularity of parallelism that can be supported. Thus, in addition to traditional distribution mechanisms such as RPC, workstation clusters should support lightweight communication paradigms for executing parallel applications.

This paper describes a simple communication model based on the notion of *remote memory access*. Applications executing on one host can perform direct memory read or write operations on user-defined remote memory buffers. We have implemented a prototype system based on this model using commercially available workstations and ATM networks. Our prototype uses kernel-based emulation of remote read and write instructions, implemented through unused processor opcodes; thus, applications (or runtime libraries) see direct machine support for remote memory access. We show that this model can be supported safely and efficiently on current systems; for example, a 40-byte remote write operation completes in 30 μs on our 140 Mb/s ATM-based network with 19.5 SPECint DECstation hosts. Using example applications, we show how this underlying model can provide excellent performance for both parallel applications and traditional distributed system services.

1 Introduction

Recent advances in network performance have increased the attractiveness of parallel computing on workstation clusters. Such loosely-coupled structures built from commodity parts offer several advantages over dedicated multicomputers, such as the Intel Paragon [14] and Thinking Machines CM-5 [24]: commodity parts are lower cost, they don't require dedicated use, and they are flexibly scaled and upgraded. While workstation clusters have been widely used for distributed (and sometimes parallel) applications in the past, their communication environment relies on heavyweight client/server models and message-based (RPC) communication. In contrast, parallel applications have favored simpler models that involve partitioning data among processors and more direct inter-processor data access. Thus, for workstation clusters to facilitate parallel programming, we believe that they should support finer-grained, lower-cost network access, which is crucial to application performance.

This paper presents a model of network communication based on *remote memory access* that can be an effective basis for communication in a workstation cluster for both parallel and distributed environments. We demonstrate how this access model can be efficiently and safely implemented on current generation

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. C. Thekkath is supported in part by a fellowship from Intel Corporation. H. Levy is supported in part by a Fulbright research award and by INRIA.

processor and ATM networks using a software approach. To show the performance potential of this model, we present measurements of two parallel applications running on a small ATM network; in addition, we provide measurements of an RPC implementation we have layered on top of our remote access model. In effect, we show how shared-memory primitives can be implemented and used effectively at a coarse granularity (relative to shared-memory multicomputers such as the Dash [18] or KSR [16]), in support of parallel and distributed programming using current technology.

The notion of network remote memory access is not new: Spector first explored the idea on the experimental Ethernet [23], concluding that the small data packet and high bandwidth requirements ruled out Ethernet-style networks as a basis for realistic loosely-coupled multicomputing. More recently, multicomputer researchers have incorporated some of these ideas into reliable networks for distributed-memory multicomputers [25]. The Nectar project at CMU connected workstations using a high-performance custom network [3], while DEC's VAXclusters system [17] used complex communication controllers to provide distribution services for a cluster. At a coarse granularity, systems like Ivy [19], Munin [8], and Memnet [11] provide coherent remote page access using high-level software or hardware. We expect that our model could be used as a lower-level mechanism for alternative implementations of shared, coherent, distributed virtual memory systems such as these.

In general, simultaneously providing high performance, protection, shared access, and multitasking has always been a difficult goal to achieve. Our work addresses these issues in the context of current generation general-purpose workstations and networks. We argue that networked systems should support low-latency remote memory access, and demonstrate how that support can be easily integrated into operating systems and hardware.

This paper is organized as follows. Section 2 describes our network access model, the interface we have implemented, and the performance of that implementation. We show that this model can be efficiently realized using commercially available hardware. Section 3 reports our experience with the implementation; in order to test the feasibility of this model as the basis for higher-level software, we implemented and measured parallel applications, and built an RPC system as well. Section 4 presents our conclusions and summarizes our work.

2 The Network Access Model

At an abstract level, our network access model consists of a network interface or controller that provides a set of remote memory *segments*. Segments are contiguous pieces of user virtual memory; they are defined by user applications and controlled through *descriptors* maintained by the interface and privileged software. Applications exchange segment information through a higher level protocol. Once exchanged, descriptors can be named by the communicating parties, permitting direct *reads* and *writes* of data at specified offsets within the remote segments. Read and write operations are supported through special meta-instructions, described in detail below. Segments are protected from unauthorized access, because applications can selectively grant or revoke access rights to their exported segments.

While it is feasible for applications to directly use this access model, we expect that higher layers of software will typically hide many of the details from applications. For example, a stub compiler could use our low-level instructions to provide standard, at-most-once, RPC semantics; or, a communication compiler [12] could automatically generate high-performance communication code tailored to specific application requirements. In both cases, the mechanized approach relieves the programmer from the details of descriptor exchange, remote memory offsets, message loss, synchronization, and so on. High-level software also handles the effect of unreliable networks in our model, simplifying its implementation.

Modern LANs have good loss characteristics and thus there is no reason to burden the network interface with functionality that is unlikely to be used. In Section 3 we demonstrate prototype systems that use the underlying model to achieve good performance.

2.1 The Interface

It is most convenient to think of the interface as the instruction set of a communications co-processor. The interface supports two non-privileged instructions: `WRITE` and `READ`. The write instruction has the form: `WRITE` *rd*, *off*, *count*, *intr*. *Rd* specifies a descriptor register in the co-processor that identifies a remote memory segment. The descriptor includes the destination segment size, remote node address, and protection information. *Off* denotes the starting byte offset in the segment for the write. *Count* specifies the number of bytes to be written. The data for the write is taken from a set of message registers shared between the sending processor and co-processor. *Intr* indicates whether the remote destination is to be interrupted when the data reaches the destination.

On a write instruction, the co-processor verifies the rights of the sender. If the check is successful, it formats the data from the registers and sends it to the remote destination together with a descriptor identifier, offset, and count. If the network cannot be immediately accessed, the co-processor buffers the data in an internal FIFO, delivering an exception should the FIFO overflow.

On receiving a write request, the remote co-processor uses the segment descriptor number, the offset, and the size to validate the request. The descriptor identifies a virtual address range within some process. The co-processor reads the address translation tables for that process and writes the data to memory. The host processor is not necessarily interrupted on data delivery.

The read instruction has the form: `READ` *rs*, *soff*, *count*, *rd*, *doff*, *intr*. *Rs* and *soff* specify the remote source segment and offset where the data to be read can be found. *Rd* and *doff* define a local destination segment and offset where the data is deposited. The `READ` request is non-blocking, i.e., the issuing process is allowed to proceed. When the data is returned from the remote processor, it is deposited in the reader's address space. No message registers need to be specified for a `READ`, which simplifies its operation.

We chose a multi-word load/store model as opposed to a traditional message passing model such as send/receive for reasons of efficiency. Loads and stores specify the ultimate destination of the data in memory. In contrast, message passing models specify only communication end points, which necessitates overhead in demultiplexing and data copying. We have also biased our scheme towards small message packets or cells, such as those seen on ATM networks. This gains implementation performance by allowing common-case optimizations. Larger, variable-length packets can be accommodated as straightforward extensions of our basic scheme. Overall, we believe that this interface model is beneficial because it is easy and natural for programmers and run-time systems to use, it corresponds to the model used by many parallel programs, and it permits highly-efficient kernel-based implementations, as we show in the following sections.

Many characteristics of a workstation cluster make it qualitatively different from a traditional closely-coupled fine-grained multicomputer. These are related to sharing, protection, independent time-slicing among nodes, and other factors. Thus, beyond simple read and write instructions, our model (and implementation) explicitly supports additional mechanisms to accommodate these special needs. These mechanisms include (1) descriptor maintenance, (2) export and import of segments, (3) application-based pinning/unpinning of virtual memory pages, (4) segment write inhibit for synchronization, and (5) interrupt control. Support for these facilities is an important part of our model, and differentiates our work from previous efforts (e.g., [23]); however, space prohibits a complete discussion of these operations in this paper.

2.2 The Testbed

We have implemented our network access model using a software layer on top of an existing ATM host-network interface. Our implementation is optimized around the expectation that (1) in general, many operations are performed on a segment once descriptors are exchanged, and (2) each operation transfers a relatively small amount of data, e.g., on the order of tens of bytes. In this section we briefly summarize the characteristics of the interface and the host architecture before describing our implementation.

We use FORE host-network interfaces to connect our DECstation 5000s to a 140 Mb/s ATM network [13]. The interface is located on the TurboChannel I/O bus and does not have DMA capabilities. Instead, it implements two FIFO queues, one for transmitting ATM cells to the network and the other to buffer received cells. Processor accesses to these FIFOs are performed a word at a time and run with no wait states. Thus, to read a standard 53 byte ATM cell from the FIFO, the processor requires 14 (40 nanosecond) I/O bus read cycles. The receive FIFO is deep enough to store nearly 300 cells while the transmit FIFO has space for about 40.

We use a FORE ASX-100 switch with 4 ports, each running at 140 Mb/s. We measured a latency of about 7 μs for each cell going through the switch. However, this is not a fundamental limitation of ATM switches. We expect that as the technology matures, switching times can be brought down to about a few hundred nanoseconds with faster datapaths, deeper pipelining of routing table lookups, and virtual cut-through.

The DECstations contain 25 MHz MIPS R3000 processors rated at 19.5 SPECint. Many exceptions, including all I/O interrupts, are vectored to a single handler, which must then identify both the type of the exception and the originating device for interrupts. This requires reading the system's control and status registers as well as accessing the on-chip control co-processor's registers. Thus, servicing an interface interrupt requires instructions in addition to those required to handle the device specific operations.

2.3 Implementation Overview and Low-Level Performance

The bulk of the network access model is implemented in system software running inside the kernel. The performance critical meta-instructions (e.g., remote read and write) are implemented as MIPS machine instructions using unused opcodes from the R3000 instruction set. The instructions are emulated in the kernel using carefully tuned assembly routines. All the system software and emulated instructions are integrated into an otherwise standard DEC Ultrix 4.2A kernel. Protection is provided by the emulation code, which checks the validity of all accesses.

While the cost for kernel entry/exit often reduces performance for latency-critical operations, this need not be the case. The poor kernel entry/exit performance observed in conventional operating systems results from two main sources: the kernel typically (1) executes many memory access instructions to save and restore state, and (2) must validate pointer arguments, because applications are untrusted. Neither of these factors is insurmountable, however. In our implementation the amount of state saved and restored was minimized by carefully writing specialized kernel entry and exit points for the network instructions. Further, since the network instructions contain no pointer arguments (they only specify registers containing data), there is no overhead in validating pointers.

Information about segments is maintained in a simple table in the kernel; the segment descriptor name is used as a table index to locate the segment record. Each segment record contains a set of address translation entries that are valid at a given time. These entries are consulted before incoming data is written into an address space.

Number of Instructions for a 40-byte Remote WRITE Operation	
Operation	**Inst. Count**
Protection Crossing	25
Access Check & Writing Data to Device	32
MIPS Interrupt Dispatcher (on Destination)	73
Device Interrupt Handler	21
Reading Data and Writing Destination Addr	58
Total	209

Number of Instructions for a 40-byte Remote READ Operation	
Operation	**Inst. Count**
Protection Crossing	25
Access Check & Writing Read Request to Device	37
MIPS Interrupt Dispatcher (on Destination)	73
Device Interrupt Handler	21
Reading Request, Accessing Memory & Writing Data to Device	67
MIPS Interrupt Dispatcher on Source	73
Device Interrupt Handler	21
Reading Data and Writing Destination	58
Total	375

Table 1: Instruction Counts

In our implementation, each read and write operation is of up to 10 4-byte words. Table 1 summarizes the instruction counts for remote read and write instructions. As indicated in the first row, the cost of crossing protection boundaries is only 25 instructions. While the instruction counts for read and write are quite small, we still pay a price for the software implementation. One cost is the penalty incurred by the interrupt dispatcher in decoding the exception. Another cost is not evident from the table: nearly 50% of the instructions perform memory loads or stores, which tend to slow down execution. Despite these factors, a remote write operation containing one ATM cell (40 data bytes) takes 30 μs. For comparison, a processor-local write of that size, assuming 200 ns access time, is only 15 times faster. A remote read takes longer than a write — 45 μs — since two cells must be exchanged. These measurements are between two hosts connected directly without a switch; however, as noted above, we expect next-generation switches to introduce only small additional latency.

If the cell size were increased, the *incremental* cost per word would be 3 or 4 instructions. In addition, Table 1 represents the worst case, because every cell arrival need not cause an interrupt. If cells arrive back-to-back, only the first arrival would interrupt; the interrupt handler would then drain the input queue of all packets without further interrupts. Further, when streaming multiple cells, sending and receiving code can be pipelined.

It is instructive to compare these measurements with the raw performance of the underlying FORE host-network interface. To obtain this best-case performance, we constructed a simplified environment, ignoring issues of protection, sharing, and demultiplexing of incoming packets. The device was mapped into an address space, which accessed the network directly. We removed interrupt handling costs as well, because the address space polls the network constantly. Running two hosts in this configuration, we measured the time to do a write operation at about 11 μs. Thus, our software implementation has added about 19 μs to the write operation in order to multiplex multiple users and provide protected accesses. For read operations

System	Latency (μs)	
	40 bytes	**80 bytes**
DECstation 5000/200 ATM	37	55
Intel Touchstone DELTA	71	75
CM-5	91	96
CM-5 with Active Messages	28	33

Table 2: Latency Comparison

the increased cost is greater (about 23 μs), because the interrupt handling code is invoked twice, once on each end of the transfer.

Simple processor support for demultiplexing I/O interrupts, e.g., as suggested in [2], would reduce the costs of a software approach. Given low-cost interrupt handling, the software approach has a flexibility advantage over a tightly integrated, pure hardware approach providing a few special network instructions. Application-specific instructions can be easily added using the software approach. For instance, with very little effort, we synthesized a compare-and-swap instruction that we use in the applications described in Section 3. (Unlike the read/write primitives, which are only as reliable as the network, the compare-and-swap operation is reliable in the face of cell loss.) Beyond the cost of interrupt handling, additional performance gains are possible by redesigning the network interface to avoid using the I/O bus, using network interfaces connected to the cache bus instead, as is done in dedicated multiprocessors such as Alewife [1].

One might ask whether a design optimized for cell-size writes suffers on larger block transfers, relative to a specialized block transfer primitive. To answer this question, we implemented and measured a block transfer instruction that transfers data from an arbitrary-sized user buffer rather than a fixed set of registers. We measured the throughput achieved for a 4K remote block transfer to be 35.4 Mb/s, compared to 34.7 Mb/s using remote writes. Thus, our remote write interface loses little in the way of throughput relative to a block transfer primitive.

The reason for the behavior of block transfers is the following. On the sending side, only a single emulation trap is made for block transfers, compared to the multiple traps needed for remote writes. However, the overhead on the receiving side is on a per cell basis, regardless of the approach. Thus, any performance difference is only due to the decreased number of send side traps, which is small in our implementation. On the other hand, using the block transfer primitive to transfer single cells leads to higher latency because of the overhead of pointer checking inside the kernel.

2.4 Comparative Microbenchmarks

To put the performance of our implementation in perspective, it is useful to compare it with dedicated architectures, such as the CM-5 and the Intel Touchstone DELTA. (The Intel machine is a precursor of the Intel Paragon for which suitable microbenchmarks are not yet available.)

Table 2 compares the latency of sending small amounts of data in the various systems. To measure the performance of our DECstation/ATM implementation, we connected a pair of workstations to the ATM network via a switch. The average time required to perform a remote write of a given size is shown in Table 2. We did not perform measurements on the CM-5 and the DELTA, but use values reported elsewhere [20, 25].

Table 3 compares the sustained throughput achieved by the various systems. Our implementation overlaps multiple write requests; switch overhead is included in the measurement. Once again, we did not measure the CM-5 and the Intel figures, but quote from the literature.

System	Sustained Throughput Mbytes/s
DECstation 5000/200 ATM	4.3
Intel Touchstone DELTA	10.0
CM-5	8.3
CM-5 with Active Messages	8.3

Table 3: Throughput Comparison

In relation to the other systems, our implementation has traded off bandwidth for latency. We chose this bias, because we expect applications to use many exchanges of small amounts of data rather than being throughput intensive. Given better interrupt vectoring support, we could increase our bandwidth as well. It is important to note, however, that although the FORE ATM network has a bandwidth of 140 Mb/s, the best achievable memory-to-memory throughput on DECstations is considerably less than this (cf. Section 2.3). Our implementation achieves 70% of what would be possible if we directly used the FORE host-network interface to transfer data to remote memory.

3 Application Performance

This section demonstrates the use of our access model for supporting both traditional distributed system services and parallel applications. We first examine the implementation and performance of RPC built on top of the remote memory access interface. We then describe two parallel applications that we have implemented using the model, and show their speedups on the network.

3.1 RPC Support for Distributed Systems

It is possible to bypass our interface and build RPC using a conventional network model and well-known techniques [22]. However, it is convenient to have one interface for all workloads if performance is not impacted. We demonstrate in this section that an efficient RPC based on the shared memory primitives is in fact feasible.

The performance of an RPC system is very sensitive to two factors — marshaling and transport — which in turn are influenced by the access model and the network type. An advantage of the remote memory model is that it permits shared-memory, same-machine optimization techniques to be used for the cross-machine case. On the other hand, the smaller cell sizes of the ATM network require some of the common case assumptions of the transport layer to be re-evaluated. We now examine marshaling and transport in the context of our RPC implementation.

3.1.1 Marshaling

Given the notion of protected, remote memory that can be shared between a client and server, it is natural to extend techniques used for high-performance same-machine RPC, such as LRPC [5] and URPC [6]. In our system, the server exports stacks that are then imported by clients at bind time. On an RPC call, the client stub picks an available stack for the server and builds a call frame on that stack using the remote write operation. In the absence of call-by-reference, the call frame is identical to that for a local call. There is really no marshaling or demarshaling *per se*; data moves directly from source memory to destination memory without unnecessary copying or buffering. (Using writable stacks to simplify demarshaling costs can be done even without the remote memory model [15], however doing so is more complex.) Once the stack is ready, the client activates the server by writing a flag word in the server, for which the server polls. Call-by-reference is straightforward to provide through the remote read and write primitives. In this case, the references placed on the call frame must contain a segment number (descriptor) and offset into an exported client segment.

The technique of sharing a writable stack is most beneficial when the underlying compiler and processor support separate argument and execution stacks. In this case, only the argument stacks are shared between the client and the server. Contemporary RISC architectures and compilers, however, favor a unified argument and execution stack, which would permit a malicious or incorrect client to crash the server. The execution stack must therefore be protected during server execution. Our implementation allows efficient revocation of segment write access, enabling the server to execute safely. The server exports each stack as a separate segment, write-protecting a stack segment before an RPC procedure begins. The standard argument checking that is performed by the server is then adequate protection against ill-behaved clients.

3.1.2 The Transport Layer

Modern LANs have low loss rates, however no network is completely reliable. The function of the transport layer is to provide an efficient mechanism to cope with data loss. Traditional RPC transports were designed for the Ethernet, where RPC calls typically fit into a single packet. Consequently RPC protocols have been optimized for single packet exchanges [7]. With the small cell size of an ATM network, however, data for typical RPC packets may not fit into a single ATM cell.

There are at least two alternative schemes to optimize for the common case of multi-packet exchange on ATM networks. One alternative is to treat an aggregate group of cells as one higher-level message with one acknowledge. In this approach the entire message must be retransmitted even if only one packet is lost. But, if packets are rarely lost, this scheme has the advantage of a simple design and implementation. Another alternative, and the one we use, is based on selective acknowledgements [9, 10]. A fixed number of cells (say N) are grouped into a message; the cells are then transmitted without waiting for an acknowledgement. Since ATM networks guarantee cell sequencing from a particular source, the arrival of the last cell indicates that the previous cells have either arrived or been lost. When the destination detects the Nth cell (or a timeout occurs), it writes a single cell at the source with a bit mask indicating which cells it has received. The source then rewrites only the missing cells, if any.

3.1.3 Performance

We have implemented a prototype RPC system using the mechanisms just described. Stubs were hand generated for the procedures that we measured. The entire implementation is at user level and employs a user-level threads package. To avoid the cost of context switching, we chose to spin wait the threads at user level, relying on kernel time slicing for fairness among processes. This scheme clearly favors latency over throughput, but blocking schemes could be used as well at the expense of increasing the latency by the cost of the context switches.

Using this prototype, we measured the performance of RPC on two otherwise idle workstations connected by an ATM network. The results are shown in Table 4. The column marked **Simple** shows the time required for an RPC involving four integer arguments and the return of an integer result; the computation time required to calculate the result is negligible, and the data exchanged between the hosts fits in a single ATM cell. This example does not exercise most of the transport protocol code. The other columns represent multi-cell RPCs that involve execution of the full protocol code. Arguments of 32, 64, and 128 32-bit words

System	Simple	MW-32	MW-64	MW-128	Local RPC
DEC 5000/200	93	300	488	698	57

Table 4: RPC Performance (Time in μs)

Number of Nodes	Speedup		
	Grid Size		
	20 X 20	40 X 40	80 X 80
2	1.5	1.8	2.0
4	1.9	3.0	3.6

Table 5: Ising Speedup

of user data are given to the server, which returns an integer result. The actual computation time for each RPC is again negligible. For comparison, the time to make a same-machine null RPC call using a well optimized system [4] is shown in the last column. Notice that cross-machine "null" RPC time using our approach is less than a factor of two slower than this optimized local RPC.

3.2 Supporting Medium-Grained Parallel Applications

To test the effectiveness of our model and its implementation for parallel applications we report on the performance of two parallel programs.

3.2.1 Ising Model

The first application is an optimization problem that arises in the Ising model, which is used to model the behavior of crystal lattices. Each lattice atom has a characteristic value called "spin". Atoms interact with their nearest neighbors. Each interaction has an "interaction coefficient". The lattice as a whole has an energy which is a function of the spins and the interaction coefficient. The objective is to assign spins to the atoms to minimize the total lattice energy. The problem is known to be NP hard, so our particular solution uses a heuristic.

Our program has an optimizer that scans the grid and changes spins if doing so would lower the total energy. The optimizer contains two loops that are written using red/black coloring [21], a common parallelization technique used in scientific applications.

The grid is spatially distributed amongst the processors' memories such that each processor has two nearest neighbors. The boundaries of the grid contain data that is shared between neighboring processors. During program execution, spin values of atoms that have changed and are on adjacent processors have to be exchanged. The processor that changed the spin writes the new values to the adjacent processor using remote writes. At the end of each sweep through the grid, adjacent processors synchronize with each neighbor using the compare-and-swap primitive.

Our implementation uses the network access model directly, i.e., without any mechanism to cope with lost cells (except for the compare-and-swap, which is reliable). However, a transport protocol of the type described in Section 3.1.2 can be devised so that the overhead in the common case of no cell loss is small. Thus, our performance numbers are upper bounds but are realistic.

Number of Nodes	Speedup	
	Random	Linear
2	1.7	1.8
4	3.4	3.5

Table 6: Simulator Speedup

Table 5 indicates the observed speedup of the multi-node algorithm relative to an otherwise identical single-node solution that uses local memory and no synchronization. For an N X N grid, evenly distributed among P processors, the computation per processor on each sweep is proportional to N^2/P and the data exchanged is proportional to N. For small grid sizes, the time to perform the floating-point computation to calculate the new lattice energy is small relative to the time required to communicate the spin values. Consequently, speedups are modest with increasing processor count. Our measurements indicate that 40X40 is the smallest problem size for which our implementation can be expected to achieve good speedups.

3.2.2 Circuit Simulator

The second application we developed was a general-purpose, gate-level circuit simulator that uses a conservative (Chandy-Misra), distributed simulation algorithm. The simulator reads a circuit description and distributes the gates of the circuit among the various nodes. Each gate input is modeled as a queue of events into which output gates write their signal values as the circuit is being simulated.

Each node performs a simple simulation loop. For each gate on the node whose inputs are ready, it calculates the output signal value based on the type of the gate. The output signal values are then fed to the appropriate input gates, some of which might be on a remote node. Most of the parallelism of the application is achieved because the different nodes may proceed independently if their gates have valid inputs.

Our simulator is a natural candidate for exploiting remote memory references, e.g., the current version has a much simpler communication interface than the message-based version from which it was ported. Gates whose inputs are fed by remote gates are located in an exported segment that is imported by the remote gate. New signal values are simply written by using a remote write.

Table 6 shows the results of running the simulator on a 32-bit adder circuit. Performance from two distribution of input gates are shown: linear and random. In the linear distribution, with P nodes and N gates, the first N/P gates are given to the first node, the next N/P are placed on the second one, and so on. In the random distribution, the gates were distributed using a random number generator. The particular circuit we used is quite modest and has about 225 gates.

4 Conclusions

We have presented a network access model based on remote memory access that facilitates both parallel and distributed processing on workstation clusters. Our interface is easy and natural to use, and permits highly-efficient kernel-based software implementations with modern switch-based networks and interfaces. The model can be used in conjunction with mechanized code generators as well as directly by applications without undue effort. In fact, our experience with porting applications from message-passing paradigms indicates that programming is easier using remote references.

In addition to simple read/write primitives, our model and implementation handle issues of protection, shared network access, virtual memory, and multitasking that arise in a workstation cluster. We view these mechanisms as value-added features, allowing clients to safely and conveniently use the basic remote instructions in a shared environment.

Our experiments indicate that the model can substantially help to improve the performance of critical distributed system mechanisms like RPC, with cross-machine RPC performance within 2 times a same-machine RPC on similar processors. Moreover, we have shown that with commodity parts and the proper interface, it is possible to achieve small-message latency competitive with that delivered by dedicated parallel computer systems.

Finally, the performance numbers indicate that the flexibility of the software approach can be traded for additional performance by migrating critical functions, such as memory instructions and segment access instructions, to hardware. We believe that hardware controllers based on this model can be successfully used in future workstations connected to contemporary networks such as ATM.

Acknowledgments

We received substantial assistance from Brian Bershad of Carnegie Mellon University and from Eric Cooper, Mike Oren, Mike Riger, and Matt Wartell of FORE Systems. The authors would like to thank the groups at INRIA project SOR and Bull/IMAG for their support. Thanks are also due to Anant Agarwal and John Kubiatowicz for useful discussions about Alewife. Ed Felten provided an initial implementation of the Ising application and commented on an earlier draft of the paper which greatly added to its clarity. Alex Klaiber provided the code for the circuit simulator. Radhika Thekkath helped substantially in porting it to the remote memory model and provided invaluable comments on earlier drafts of the paper.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, MIT, 1991.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[3] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H.T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[4] Brian N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Architectures*, pages 205–211, April 1992.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[6] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[7] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] David R. Cheriton and Carey L. Williamson. VMTP as the transport layer for high-performance distributed systems. *IEEE Communications Magazine*, 27(6):37–44, June 1989.

[10] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Proceedings of the 1987 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 353–359, September 1987.

[11] Gary Delp. *The Architecture and Implementation of Memnet : A High-Speed Shared Memory Computer Communication Network*. Ph.D. thesis, University of Delaware, 1988.

[12] Edward W. Felten. The case for application-specific communication protocols. In *Proceedings of Intel Supercomputer Systems Division Technology Focus Conference*, pages 171–181, 1992.

[13] FORE Systems, 1000 Gamma Drive, Pittsburgh PA 15238. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[14] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[15] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software Practice and Experience*, 23(2):201–221, February 1993.

[16] Kendall Square Research Inc., 170 Tracer Lane, Waltham, MA 02154. *Kendall Square Research Technical Summary*, 1992.

[17] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[18] Daniel Lenoski, James Loudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessey. The DASH prototype: implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[19] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[20] Richard J. Littlefield. Characterizing and tuning communication performance on the Touchstone DELTA and the iPSC/860. In *Proceedings of the 1992 Intel User's Group Meeting*, pages 4–7, October 1992.

[21] J. M. Ortega and R. G. Voight. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(149), 1985.

[22] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[23] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):246–260, April 1982.

[24] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[25] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.