Design Challenges of Virtual Networks: Fast, General-Purpose Communication

Alan M. Mainwaring
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

David E. Culler
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

Abstract

Virtual networks provide applications with the illusion of having their own dedicated, high-performance networks, although network interfaces posses limited, shared resources. We present the design of a large-scale virtual network system and examine the integration of communication programming interface, system resource management, and network interface operation. Our implementation on a cluster of 100 workstations quantifies the impact of virtualization on small message latencies and throughputs, shows full hardware performance is delivered to dedicated applications and time-shared workloads, and shows robust performance under demanding workloads that overcommit interface resources.

Keywords

virtual networks, high-performance clusters, direct network access, application programming interfaces, system resource management, protocol architecture and implementation

1. Introduction

Whereas large-scale parallel machines were once constructed of highly-specialized nodes and run as single-user or space-shared systems, they are now almost universally built from general-purpose microprocessors with a complete operating system on, or spread across, every node. As their generality evolves, they are deployed not only for compute-bound physical simulations, but for an increasingly rich set of data intensive services and shared environments. Nonetheless, the primary distinguishing characteristic of parallel systems, as opposed to other collections of computers on a network, is their support for fast user-level communication. This capability is what allows intense sharing of resources and transfer of information within parallel applications. This paper investigates the inherent design challenges in providing high-performance communication to a broad range of applications in a general-purpose environment.

This work was supported in part by the Defense Advanced Research Projects Agency (F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, and California MICRO.

PPoPP'99 5/99 Atlanta, GA, USA

The tension between performance and generality presents an especially interesting challenge for high-performance clusters, since, on the one hand, they offer tremendous generality by using complete computers as building blocks and, on the other, seek to deliver the performance of fast, scalable system-area networks [1, 18]. The implementation of the high-speed communication substrate should not constrain the overall usage model of the parallel system. For several years it has been well-demonstrated that communication performance could be delivered by mapping network hardware directly into the address space of the user application [4, 6, 17, 25, 27, 29, 30, 32, 33]. However, providing this capability to only a single or a few prearranged parallel programs at a time severely limits how the overall system can be used. While there are times when nodes will be devoted to a single program, high-speed communication ought to be available to all components, including file systems, schedulers, debuggers, performance analyzers, parallel clients and servers, and traditional client/server applications.

Practically, the way to obtain performance and generality is virtualization. The operating system can provide the illusion of direct access to resources, but actually bind virtual resources to physical ones on demand. Much as physical memory hosts the most active pages of virtual address spaces, the physical network resources can be focused on the most active loci of communication. Virtualization facilitates the sharing and presentation of physical resources to consumers, with the operating system able to manage protection and scheduling while remaining off critical performance paths. When done well, it provides the performance of direct application-to-resource bindings when usage approximates the stand-alone case, effective sharing of resources when they are not overcommitted, and graceful degradation under heavy loads. When done poorly, it may either fail to deliver a large fraction of the hardware capability to any one application, or may degrade severely under load. Traditional protocol stacks suffer the former, because of the run-time intermediate interpretation and management that occur on every operation, even when a single user program operates well within the capability of the underlying network.

This work makes three contributions: (1) it describes the design, implementation, and evaluation of a complete, large-scale virtual network system in daily use for more than a year on a cluster 100 workstations serving a diverse user

community, (2) it isolates network virtualization techniques that deliver full hardware performance to dedicated applications and robust performance under workloads that overcommit communication resources, and (3) shows how these techniques can be integrated within existing programming interface, operating system, and network interface (NI) frameworks.

In what follows we present the design of our virtual network system in a layered fashion and examine how the integration of network virtualization impacts each of the core architectural components. Each section outlines the general principles and the critical issues in practice. After background is established in Section 2, Section 3 addresses the programming interface provisions to support general purpose use: naming, protection, delivery models, and thread-based events. Section 4 examines the key operating system support in the context of Solaris, including integration with the virtual memory system and the driver/NI protocols. Section 5 examines network interface support, including the service and queuing disciplines, and the transport protocols. Section 6 discusses system performance in three regimes using microbenchmarks, parallel applications, and macrobenchmarks that examine robustness. Finally, Section 7 discusses lessons learned and related work.

2. Background

Approaches to network virtualization have been proposed in the form of Abstract Device Channels [15], Remote Memory Mapped Regions and reflective memory channels [5, 17], Remote Queues [2], and Active Messages [26], and several prototype cluster systems have been developed [4, 6, 27]. Indeed, the results have been promising enough that a major industrial consortium recently released the Virtual Interface Architecture [8] to serve as a point of consolidation for this work. The evidence to date has focused primarily on small prototypes, point-to-point benchmarks and single application studies that demonstrate the benefit of mapping network hardware into an application's address space. However, these tests may stress little of the network virtualization, such as the mechanisms for the virtual-to-physical binding, policies for placement, replacement and scheduling, and the coordination across nodes. Thus, effectiveness of network virtualization at scale remains a largely open question.

The cluster used in this study consists of 100 167-Mhz Sun UltraSPARC-1 workstations running Solaris 2.6 with 128 MB of memory. The machines are connected by a Myrinet [1] network with 25 switches and 185 links in a fat-tree like topology. The switches have an average cut-through latency of ~300 *ns* and have 1.2 Gb/s bi-directional ports. Network paths are shallow, with ~7 bytes of buffering per-hop, and while link-by-link flow control and back pressure contribute to low overall transmission error rates, network congestion rapidly spreads through the network. Hosts have a single LANai 4.3 network interfaces that contains a 37.5-Mhz general-purpose embedded processor with 1MB of on-board memory, independent network send/receive DMA engines, and a single DMA engine for SBUS transfers. Although it contains a general-purpose microprocessor, the NI implements a controller amenable to implementation in hardware.

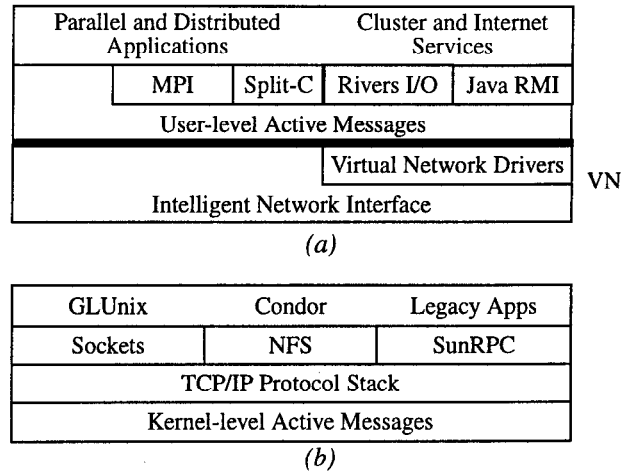


Figure 1. (a) user-level software and (b) system software operational with Virtual Networks and Active Messages.

Figure 1 shows the system architecture that we have constructed on top of virtual networks and Active Messages. This system is general-purpose, and has been operational for more than one year, supporting a diverse user community. The system provides the active subset of applications with direct, high-performance network access. At user-level, the communication programming interface supports traditional parallel libraries, such as a port of the public-domain MPICH message passing library and the Split-C language originally developed for the CM-5. It also supports high-performance parallel I/O subsystems [12], and Java-based remote-method invocations. By supporting a subset of the interface within Solaris, standard sockets, network files systems, and remote-procedure calls packages, can leverage the performance of the network.

The virtualization that delivers performance and generality in this setting raises questions at all layers of the system, *e.g.*, the nature of programming interface abstractions and operations that support both parallel and distributed applications, the realization of operating system mechanisms and policies that manage application bindings to the network hardware, and the network interface protocols and scheduling disciplines that support efficient protected network multi-programming. In developing this system, we found that many of the critical design issues arose from dealing with the requirements of full-scale use and were not revealed by simple benchmarks. Novel uses of the program-

ming interface and their compositions revealed unforeseen behaviors and interactions between layers of software within and across the nodes.

3. Programming Interface Concepts

In order to investigate techniques for supporting fast and general-purpose communication, it was necessary to define a communications programming interface that would enable a wide-range of applications.

Message passing with MPI was a standard for parallel programs, but it is oriented toward a single program with a fixed process count and imposes relatively high overhead. Sockets was the standard for client-server applications, but point-to-point connections present scalability concerns for parallel programs, and also impose large overheads. Shared memory was widely used as the mechanism for ad hoc sharing between multiple threads within a program, but it presented many open questions as a communication mechanism for a distributed environment. Active Messages were well established as a low-level programming interface on which these various popular API's have been built, with implementations on several massively parallel processors [7, 21, 28, 29, 31] and clusters [22, 23]. However, the first generation of interfaces were specialized for single parallel program and thus were not general-purpose.

Three fundamental components of the Active Message interface needed enhancement: the naming and protection model, the delivery and error model, and the integration of communication events with multi-threaded programming environments. Like its predecessors, the interface for virtual networks [26] casts communication as split-phase remote procedure calls and provides primitives for higher-level protocols and applications. However, it introduces a new abstraction, *endpoints*, which virtualize the connection to the physical network, so that many processes on a node can each have multiple endpoints. Endpoints are objects that hold message queues and associated state that resides beneath the interface. The programming interface is defined in terms of endpoints. Addressability and access rights are established among a collection of endpoints, forming a virtual network. Programming within a virtual network is then nearly identical to traditional Active Message environments. This section presents the issues raised by general-purpose use and their solutions in terms of the three components.

3.1 Naming and Protection

The interface must provide a logical communication namespace and a protection model that allows applications to control message delivery into their endpoints. We wanted the naming and protection model to allow a wide range of network technologies, *e.g.*, it should enable either send-side or receive-side address translation and protection checks. The protection model should be sufficient to catch programming mistakes, protocols errors, and potential hardware errors from which applications should be insulated. Whereas strong end-to-end security and authentication measures should be implemented in the applications where necessary, since handlers can decide what to do with the messages that are received.

Endpoint names are opaque, *i.e.*, they have no predefined internal structure, so that many of the communication naming schemes in current use can be employed, *e.g.*, (IP address: port number), and the names can be obtained by any rendezvous mechanism. Applications use endpoint relative naming for actual communication operations. An endpoint object contains a simple translation table, which allows programs to construct a logical communication namespace of small integers by associating endpoint names and protection keys. A communication operation specifies the source endpoint and a translation table index for the destination endpoint. (Clearly, traditional virtual node number addressing in parallel programs is easily realized with this approach.) The protected portion of the communication subsystem, i.e., the NI, stamps each outgoing message with the key and routes it to the destination; the receiving interface verifies the key and deposits the data. The key must match the destination endpoint key for delivery. On a connection oriented network, such as ATM, routing and verification may be implemented using virtual circuits. A virtual network consists of a collection of endpoints that refer to one another, and is constructed by configuring the individual endpoints, rather than through some specific group membership interface.

3.2 Delivery and Error Model

The delivery and error model must balance the needs of traditional parallel programs, which expect perfectly reliable message delivery in an otherwise fail-stop model, with the needs of client/server applications and cluster services that can tolerate errors and adapt to changes. We cannot assume a perfectly reliable interconnect, even though transmission errors on emerging networks are rare, because we want the communication system to support hot-swap of links and switches for incremental scaling and to adapt to changes in the physical topology transparently. Thus, the substrate should mask transient transport and reconfiguration errors, yet provide a clean way for error-aware programs to handle serious conditions, such as a remote node crashes.

The interface specifies exactly once delivery of messages, barring unrecoverable transport conditions, and that undeliverable messages are returned to their sender, where they invoke an 'undeliverable message' handler. This enables applications to control how errors are handled, e.g., abort or re-issue, without having to take pessimistic actions in the common case, such as setting time-outs and logging message contents. This "return to sender" model allows the un-

derlying, machine-specific detection and retry mechanisms to be projected upward essentially for free.

3.3 Communications Events and Threads

The interface must integrate communication events with multi-threaded applications. Many applications, such as servers, require event driven communication which allows them to sleep until messages arrive, whereas polling is more efficient in parallel applications that communicate intensely. Both modes should be supported and applications should control the mode and which endpoint state transitions generate events. Rather than define a new event model and associated concurrency controls, the interface assumes POSIX threads with mutex locks and synchronization mechanisms.

Endpoints have event masks that sensitize a synchronization variable to endpoint state transitions, such as message arrival. Threads can set and wait on these events. Applications can mark endpoints as shared or exclusive, so that operations on shared endpoints invoke code which performs the necessary synchronization while operations on exclusive endpoints avoid those overheads. Choosing to project events to applications using standard thread synchronization enables implementations within operating system where per se process signals absent. The interface provides applications with the flexibility to determine the relationships between threads and endpoints. For example, one thread may operate upon multiple endpoints and many threads may concurrently access a single endpoint.

4. OS Resource Management

The operating system challenge in virtual networks is managing the collection of endpoints so that when processes communicate they obtain the full efficiency of the network resources, and otherwise these resources are fully available to other processes.

The approach in conventional network stacks is to multiplex/demultiplex all traffic within the kernel. Virtual networks assume a capable network interface that can multiplex traffic for a limited set of endpoints, without operating system intervention. Several systems have demonstrated fixed-degree multiplexing through a NI, *e.g.,* [6, 30]. Our approach manages the resident set dynamically, with active endpoints bound on demand to the NI in response to local or to remote references. The solution is compatible with contemporary operating systems, and supported without source-level modifications to Solaris. The key is the device abstraction between the operating system and the NI.

4.1 Management Model

Endpoint management is cast as a virtual memory problem and tackled with extensions of standard virtual memory mechanisms. Binding endpoints on-demand to hardware resources is analogous to binding pages to memory frames. Non-resident endpoints reside in application memory, but are not directly accessible by the NI. When an application writes a message into a non-resident endpoint, the system traps the reference and makes the endpoint resident, *i.e.*, binds it to communication resources. What is unique is that the arrival of a message for a non-resident endpoint can also cause it to made resident. Making an endpoint resident may require evicting an endpoint to make room, and an endpoint replacement policy selects which one.

In general, a NI requires addressability for message buffers, message descriptors, and the like. In our system, the NI contains a small amount on-board memory, through which all transfers are staged; data can be moved between the host and NI memory or between NI memory and the network, but not directly between host and network. To allow the NI to process small packets as quickly as possible, resident endpoints reside physically in the NI. The interface reserves 64KB of its on-board memory for eight endpoint frames. (Newer interface hardware supports up to 96 endpoint frames). In addition, address translations are established for bulk data transfers associated with the endpoint. This organization provides the NI with single-cycle random-access to all resident endpoints. Because they remain mapped into application address spaces, applications also have fine-grained access to them with programmed I/O. Non-resident endpoints are like any other cacheable memory page in the process address space. The operating system is involved in residency transitions, but *none* of the common-case communication operations.

4.2 Integration with Virtual Memory System

This management model is realized by extending the Solaris virtual memory system with a new endpoint module. Solaris is representative of modern operating systems, in which the application virtual address spaces consists of a collection of *segments* [16]. Each segment has an associated driver that maintains the address translations for virtual memory pages and manages their physical backing store. Segments export methods for allocating, mapping, duplicating, locking, and handling access faults. Integration with the virtual memory system at the segment layer provides precise control over all facets of memory management and much richer functionality than what is exposed to device drivers.

Endpoints are memory mapped objects, represented by address space segments, and managed by the endpoint segment driver. The functionality provided by segment drivers for managing virtual memory, apply to endpoint management as well. For example, segment creation is equivalent to allocating an endpoint and initializing its messages queues. Process termination automatically invokes segment driver methods to free segments, and with endpoint segments this may cause the driver to synchronize de-allocation with the network interface before proceeding. Most importantly, the

segment driver handles page faults which permits the on-demand re-binding of endpoints.

Figure 2. shows the four-state protocol controlling endpoint virtual address translations and backing store. Endpoints reside in uncacheable endpoint frames on the network interface (on-nic), cacheable main memory (on-host), or in the system swap area (on-disk). Endpoints in interface memory have read-write (r/w) translations, endpoints in host memory can have either read-write or read-only (r/o) translations, and endpoints migrated to disk are marked as invalid (n/a).

An endpoint initially resides in the on-host r/o state. Writes by applications (or references by the network interface) generate endpoint page faults which transition it to the on-host r/w state and schedules its re-mapping to an NI endpoint frame. This allows applications to read and to write endpoints without necessarily consuming NI endpoint frames, while providing the system with a triggering event to initiate re-mapping. Eventually, the kernel makes the non-empty endpoint resident so communication can occur. When all frames are occupied, the system replaces a resident endpoint at random and returns the endpoint to the on-host r/o state. Page reclamation mechanisms may move non-resident endpoints to secondary storage should they be the least recently used pages during periods of acute memory deficits. The 'vm pageout' transitions refers to these reclamations.

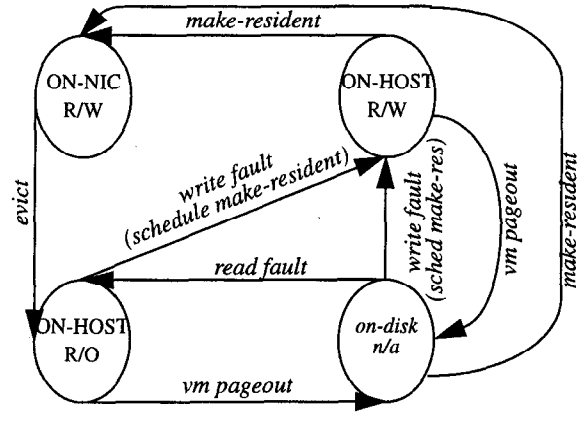


Figure 2. Operating system endpoint segment management protocol as implemented in the Solaris VM system.

The non-resident, read-write state deserves special attention. It was not in our original design but it is extremely important for robust performance under conditions of high re-mapping load. Normally, pagefaults are handled synchronously, while the faulting thread remains suspended. This state de-couples process scheduling from the binding of endpoints to the NI, and allows the application thread to continue execution immediately after a write fault. The initial page fault schedules the re-mapping operation with the system and makes the endpoint writable. The segment driver uses background kernel thread to activates non-empty endpoints, asynchronously to their initial fault handling. The thread periodically services re-mapping requests in the background, and unmaps the endpoint, moves its backing store to the NI, and updates its virtual address translations.

The activation of a non-resident endpoint in response to message arrival is also unusual, as there is no user process instruction that generates the fault, so the segment driver must simulate its effect. Again, multi-threaded operating systems provides a simple solution. The endpoint segment driver spawns a kernel thread which performs proxy operations on behalf of the NI. When requested to make an endpoint resident, it generates a software-initiate pagefault which activates the same underlying driver mechanisms.

4.3 Driver/NI Protocol

The segment driver operates concurrently with the network interface, and these two agents must coordinate their operations on shared endpoints and data structures. Both the operating system and NI make asynchronous requests to initiate operations in the other, and receive responses in return. This raises three fundamental issues: the means through which the operating system and NI communicate, the protocol that defines the possible operations and synchronizes their interactions, and how each agent performs operations on behalf of the other. Here we address the protocol and driver operations; NI operations for the driver are in Section 5.3.

The segment driver and the NI are peer agents that communicate using a simplified Active Messages interface through a dedicated system endpoint. Unlike user-level endpoints, the system endpoint is permanently resident. A small number of message handlers, in the driver and in the NI, define the protocol through which they interact. For example, the driver may request that the NI allocate an endpoint, load an endpoint from the host into an endpoint frame, or unload an endpoint to host memory. In each case, it invokes a message handler in the NI to perform part of the operation. The interface may request that the driver, for example, make an endpoint resident, notify a thread of a communication event, or manipulate DMA mappings to application memory regions. A variant of logical clocks [20] is used so that each agent can resolve the ordering of events initiated by the other, e.g., when the driver attempts to free an endpoint as the interface concurrently requests that it be made resident.

5. Network Interface

The fundamental challenge in the NI support for virtualization is balancing the demands of individual application performance with the need for fair sharing of critical hardware resources. Multiple independent flows are bound to the network interface, instead of muxed into a single flow by the operating system, and the interface obtains three core responsibilities: the efficient implementation of packet trans-

mission mechanics and protocols, fairly servicing multiple resident endpoints while retaining per-endpoint performance, and integrating driver requests with the device's ongoing communication and protocol operations.

5.1 Primitives and Transport Protocols

The mechanics of packet transmission requires driving the packet interface to send data over network links, as well as providing end-to-end transmission sequencing, flow control, error detection and handling. The interface systematically processes queues of message descriptors for resident endpoints, multiplexes packets onto the link while applying simple flow control protocols for reliable delivery, and demultiplexes arriving messages into destination endpoints.

The details of an earlier version of these protocols were previously published [10] and are only summarized here. User-level credits prevent a single endpoint from overrunning the receive queues of a dedicated destination endpoint. In addition, the network interface uses a lightweight stop-and-wait flow control protocol over multiple logical channels with positive acknowledgment. A randomized exponential back-off algorithm control packet time-outs and retransmissions. Flow control channels are self-synchronizing and automatically re-initialize sequencing state should either end enter a designated uninitialized state, *e.g.,* when a node reboots. The interface places 32-bit time stamp in the link header of each packet and receiving interfaces reflects them in their acknowledgments (see Section 5.3).

Positive acknowledgments indicate messages were written into their destination endpoint, while negative acknowledgments encode why messages could not be delivered. The prolonged absence of acknowledgments indicates an unrecoverable transport condition, e.g., disconnection or interface failure, which triggers the return of messages to their senders. Other errors, such as sending to a non-existent endpoint, do so as well. Multiple logical channels between all interfaces mask transmission and acknowledgment latencies, and take advantage of multi-path routing when available. Because flow control channels are shared physical resources, no message can occupy one for a prolonged period of time. After a bounded number of consecutive retransmissions, the NI carefully unbinds messages from channels enabling their re-use; subsequent retransmissions reacquire and rebind them. Careful engineering is required to accomplish these functions in a small number of instructions.

5.2 Service and Queueing Discipline

Because the NI services multiple resident endpoints, its service and queueing discipline can balance minimizing latency and maximizing throughput for individual endpoints while maintaining fairness and responsiveness across them. The service discipline determines the order in which endpoints are serviced while the queueing discipline determines the order in which descriptors within an endpoint are processed. Traditional protocol stacks, e.g., TCP/IP, present NIs with a unified stream of packets from a mix of processes and protocols and incoming packets are received into a single shared queue that requires further higher level processing before delivery to applications. Thus, the service and queueing discipline is determined primarily by process scheduling. Architectures that provide direct user-level messaging collapse such intermediate layers of interpretation, and the NI schedules multiple traffic flows onto the network.

The NI uses a weighted round-robin scheduler for servicing resident endpoints, and processes endpoint descriptors in a first-come first-serve manner. The algorithm cycles through resident endpoints and loiters on those with packets awaiting transmission (or retransmission). While packets remain to send, the interface processes at most 64 (the number of descriptors for sending messages) messages for at most 4 *ms* (the approximate transmission time for 64 messages of the maximum transmission unit size) before servicing other endpoints. In effect, the NI maintains a state machine per endpoint. The discipline allows the interface to cache endpoint-specific state and optimize the latency and throughput for an individual endpoint, while, at the same time, prevents endpoints sending large messages from receiving unfair proportion of attention. The in-order transmission of packets, and the sometimes out-of-order reception of their acknowledgments requires departing from the first-in first-out queueing policy for retransmission. Retransmission requires fine-grain random-access to message descriptors in endpoints. It would be costly if they were maintained in host memory.

5.3 Driver Operations

The NI interleaves the servicing of the driver endpoint among all others. While the driver/NI protocol defines a set of atomic operations in the driver, the network interface overlaps the processing of driver requests with user messages. The fundamental complication arises when the driver attempts to unload or invalidate an endpoint which has unacknowledged messages in flight. The mechanics of sending packets and protocol messages create references to physical endpoints, which must be eliminated before reusing that state. In essence, the network interface implements a lockup-free cache of the most active endpoints in the system, and while the interface must take special care while the driver modifies a particular entry, the processing of messages of all other entries continues at full speed.

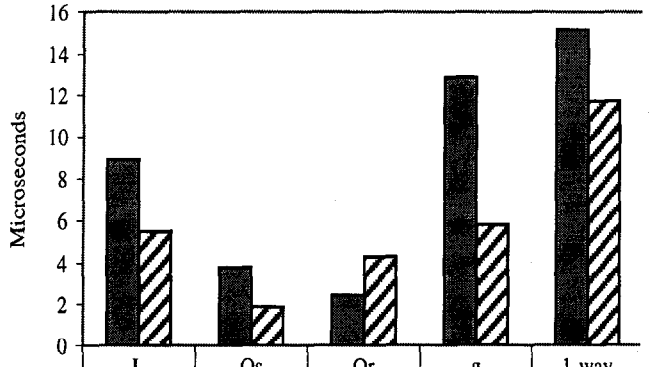
Putting an endpoint with unacknowledged messages into a quiescent state adds transient states to the dispatch loop driving the interface operation. These states prevent new messages from being sent from endpoints to be modified by the driver, while periodically retransmitting unacknowledged packets until acknowledgments are received. Once

quiescent, the driver may safely operate upon the endpoint. Because the retransmission protocol may introduce multiple copies of messages in the network, the interface must account for all such copies and their acknowledgments before responding to driver requests. For simplicity, the system statically binds flow control channels to physical network routes, and this imposes a first-in first-out ordering of messages across each logical channel. Receiving an acknowledgment for the most recent retransmission is then sufficient to account for all copies.

6. Performance

Previous sections have described a family of interrelated design choices for an effective virtual network system and the general rationale behind them. This section presents empirical measurements to evaluate its performance in three regimes and reflects upon the design choices made. It begins with microbenchmarks that reveal how virtualization effects point-to-point overheads, latencies, and bandwidths. It uses dedicated and time-shared parallel applications to evaluate typical workloads that operate well within the capabilities of the network. The remainder of the section examines its scalability and robustness under a demanding set of workloads that systematically overcommit physical network resources, and stress the virtualization mechanisms and policies.

6.1 Stand-alone Microbenchmarks



	L	Os	Or	g	1-way
AM	8.91	3.80	2.42	12.85	15.14
GAM	5.50	1.90	4.30	5.80	11.70

LogP parameters

Figure 3. LogP performance characterizations. AM denotes Active Messages for virtual networks. GAM refers to a single-endpoint interface with none of the necessary enhanacements of Section 3.

The LogP microbenchmark results in Figure 3. characterize performance for both virtual networks and a first-generation Active Message interface for stand-alone parallel programs, using a technique described in [9]. In the LogP model, the send and receive overheads, O_s and O_r, account for the host processor time spent writing and reading a message to the NI, respectively. The latency, L, accumulates the remaining end-to-end time. Each message incurs a total overhead of $O_s + O_r$ and experiences a one-way time $O_s + O_r + L$. The gap, g, is the time per (16-byte) message through the rate-limiting communications stage between two endpoints. Messages can be sent every g time units in steady state.

Virtualization and the increased demands placed upon the network interface increase the round-trip time by 23% and the gap by a factor of 2.21, while the total per-packet overhead remains the same. The larger gap is due primarily to the transport protocol and acknowledgment processing. Other aspects of virtualization, such as error checking and defensive firmware practices, contribute only 1.1 *usec* to L and g. Sensitivity studies [32] show that increases in gap are, in general, less detrimental than increases in overheads, because such increases only effect applications which send long, frequent bursts of small messages. Whereas the larger send overhead reflects the cost of writing bigger message descriptors to the NI, the smaller receive overhead shows the benefit of reading entire descriptors across the SBUS using a single SPARC VIS block load instruction.

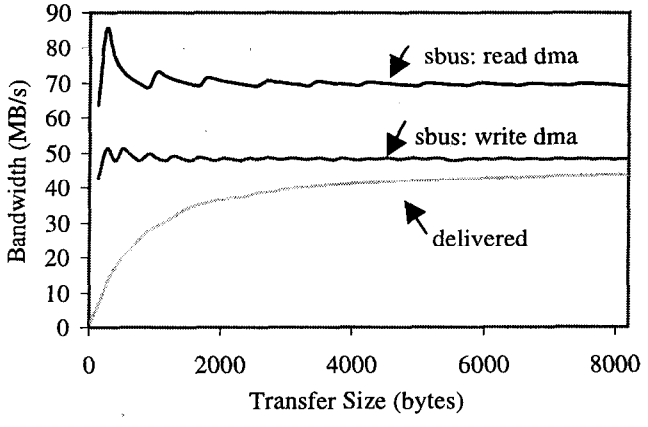


Figure 4. Transfer bandwidths. Microbenchmarks showing delivered bandwidth for 128 byte to 8192 byte messages, with maximum hardware transfer rates across the SBUS for comparison.

Figure 4. shows that virtualization has no appreciable performance impact for point-to-point bulk data transfers. The system delivers 43.9 MB/s with 8 KB messages with an $N_{1/2}$ of 540 bytes. The first-generation interface delivered only 38 MB/s for the same size message. The round-trip latencies for n-byte messages, $n \geq 128$, take $time = 0.1112(n) + 61.02$ *usec* ($R^2 = 0.99$). The SBUS exhibits asymmetric direct memory access transfer rates whether reading or writing host memory from the NI. Thus, the delivered performance approaches 93% of the 46.8 MB/s hardware limit for 8KB DMA transfers (SBUS write dma) when writing to host memory. Although the NI pipelines its processing of message descriptors to compensate for the store-and-forward delay when staging transfers through its memory, the transfer time across the SBUS dominates the added packet and protocol processing arising from virtualization.

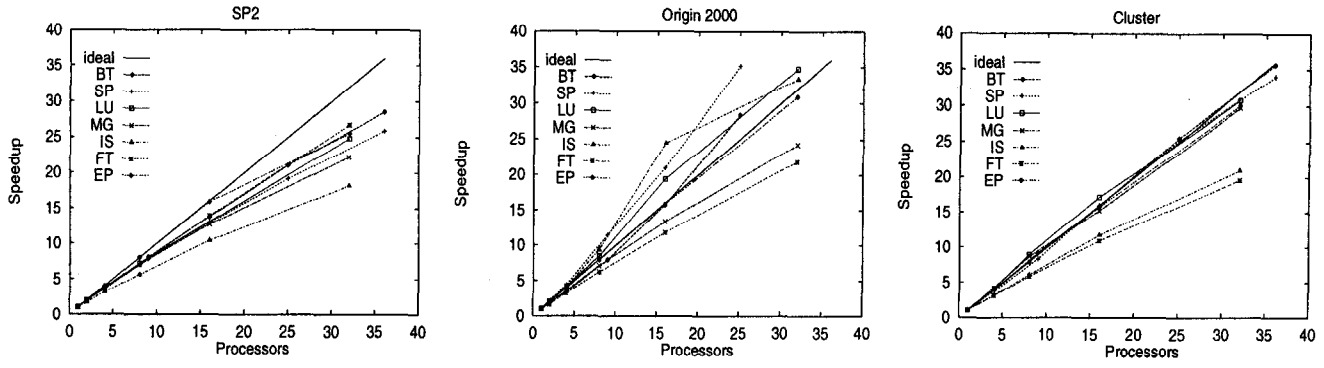


Figure 5. NPB speed-ups through 32 processors on the IBM SP-2, Berkeley NOW and SGI Origin 2000. Shows speedups for the NAS Parallel Benchmarks 2.2 (Class A) on the IBM SP-2, Berkeley NOW, and SGI Origin 2000, from 1 to 36 nodes with constant problem size scaling.

6.2 Dedicated Parallel Applications

The performance of individual parallel applications using the system in a stand-alone MPP fashion has been shown on several benchmarks. This section illustrates that the communication layer continues to deliver full hardware performance to parallel applications that use the system in a stand-alone way.

Using the public-domain Scalapack library, the optimized BLAS routines from the Sun Performance Library, and our port of the standard MPICH on Active Messages, our 100-node cluster sustained 10.14 GF on the massively-parallel linpack benchmarks, making it the first cluster on the Top-500 list [14], ranking #315 on June 19th, 1997.

We have also evaluated the performance and scalability of the NAS Parallel Benchmarks (v2). As shown in Figure 5., for Class A, the scalability is significantly better than the SP-2. All but two of the benchmarks demonstrate linear speed-ups through 32 processors. They are not embarrassingly parallel, but improved cache performance compensates for increased communication, which is even more pronounced on the Origin. The all-to-all communication within the FT and IS benchmarks was limited by the bisection bandwidth. Comparing with the newer, faster SGI Origin 2000, the execution times of all benchmarks on our cluster are at most a factor of two larger. Instrumentation reveals that not only is the relative time spent performing communications lower on the cluster, but in some instances the absolute time spent performing communication is lower.

6.3 Multiple Parallel Applications

To demonstrate support for more general workloads, we consider multiple parallel programs, each with one or more virtual networks that timeshare a partition within the cluster. This workload shows virtual networks adapting to process scheduling and not constrained in its usage model.

In general, parallel applications that communicate frequently must be co-scheduled to some degree because there are strong dependencies between the processes. A variety of mechanisms exist for co-scheduling parallel applications, either in the operating system or run-time libraries. Some systems require co-scheduling for system correctness because the communication substrate can support direct, protected access for only one process at a time [19]. Our system does not require this; it uses implicit co-scheduling which coordinates the scheduling of processes within parallel applications using conventional local schedulers. Regardless, however the scheduling system selects processes to run, the virtual network subsystem adapts the resident set to the active endpoints.

Previously published results in [12] shows that the execution time of multiple, time-shared Split-C applications (as well as synthetic benchmarks) on 16-nodes is within 15% of the time to run them in sequence. The time spent in communication remains nearly constant, which indicates that when applications communicate, they receive full network performance. In the presence of application load imbalance, time-sharing improved the throughput of some workloads up to 20%.

6.4 Virtualization at Scale and Load

We conclude this section with an examination of synthetic workloads that stress network virtualization mechanisms and reveal their robustness under severe loads. These workloads build upon a simple client/server model, with one server, and one or more client processes. The flexibility of virtual networks and the Active Message interface motivate measuring a few different workload configurations. First, to limit interactions with process scheduling and resource management external to the communication layer, the server and each of the client processes run on distinct, dedicated nodes. Second, we consider two natural designs, one where each client has an endpoint that communicates with one shared server endpoint, so only one virtual network is used, and another where each client communicates with its own unique server endpoint, so there are as many virtual network

as clients. With multiple virtual networks, we consider two natural server threading models, one where a single thread handles requests from all client (ST), and another where there is a server thread for each endpoint (MT). Each server thread waits for messages to arrive from its client, processes requests until none remain, and waits again. Finally, we consider two configurations of the server's network interface, one with only 8 endpoint frames, and the second with the default 96 frames. As the number of clients increases, the number of server endpoints increases, too. More than 8 clients cause the 8-frame configuration to overcommit its physical endpoint resources and to begin re-mapping them on-the-fly. (None occurs with the 96-frames).

In general, we might model client/server communication as alternating between computation and burst communication phases, but here we want to examine system behavior under increasingly demanding loads that overcommit NI resources and activate the underlying virtualization mechanisms. The workload is somewhat like a page thrash test. Each client sends a continuous stream of requests to its endpoints in the server, and we increase the number of clients. Each graph shows the throughput for five possible configurations over a 20 second interval in the steady state.

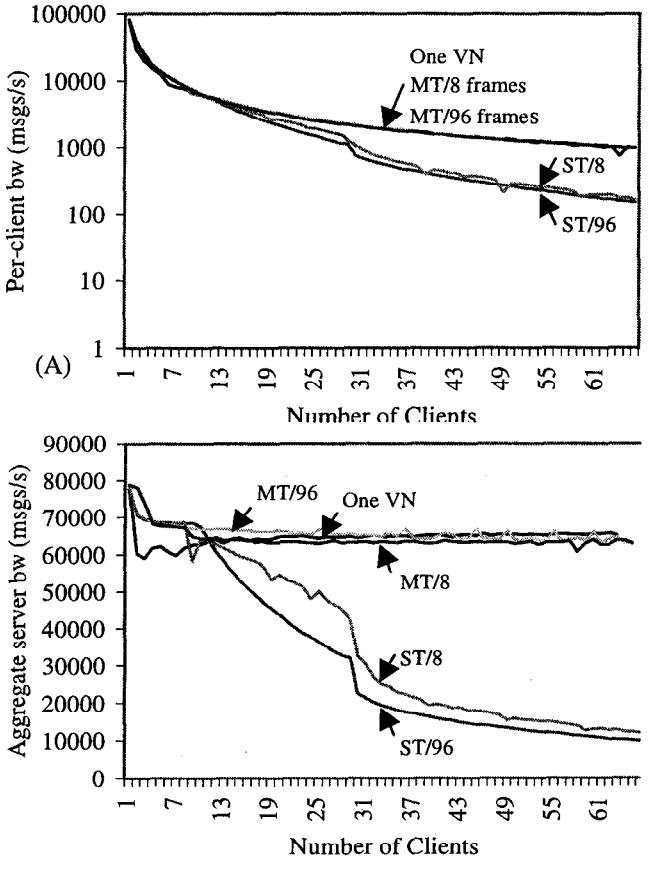


Figure 6. Small message performance under contention.

Figure 6. shows throughput for small messages across different configurations. 6a shows per-client throughput on a semi-log scale, and 6b shows the aggregate server throughput. In the OneVN configuration, each client obtains its proportional share of the server's maximum throughput of 78K msgs/s. With ST, clients continue to receive their proportional share, but two factors may contribute to the aggregate performance degradation. With 8 server frames, the server may stall when endpoint re-mapping occurs, and with 96 frames, the costs of polling resident but non-cacheable endpoints in interface memory outweigh that of polling non-resident but cacheable endpoints in host memory. In the MT configuration, performance is resilient to the number of server frames. Theads with empty endpoints remain asleep until messages arrive. Threads blocked waiting for their endpoints to become resident don't prevent threads with non-empty, resident endpoints from running.

Figure 7 shows throughput for 8KB bulk messages. 7a shows per-client throughput, and 7b shows the aggregate server throughput. In the OneVN configuration, each client obtains approximately its proportional share of the server's maximum throughput, in this case of ~42.8 MB/s. The ST configuration shows sensitivity to the number of server frames. With only 8 frames, server performance drops with 9 clients and then degrades slowly. With 96 frames, no re-mappings occur and ST performance surpasses OneVN for the reasons explained below. In the MT configuration, performance is similar to the ST configuration and remains sensitive to the number of server frames. The local scheduler does interact with the endpoint scheduling in the operating system, and the service and queueing discipline on the NI. Threads for client endpoints with pending messages can be run while threads suspended during re-mapping operations or awaiting the arrival of requests will not block them.

6.4.1 Discussion

These results demonstrate the ability to overcommit NI resources (by more than 8:1), while providing robust throughput for resident endpoints and fair service for all endpoints over time. Under these workloads, the operating system sustains approximately 200-300 endpoint re-mappings per second, but still delivers 50%-75% of its performance. Over short intervals, those clients with resident server endpoints successfully deliver messages. Messages for non-resident endpoints are negatively acknowledged and retransmitted later, while the server interface requests the driver re-map the endpoint. Similarly, round-trip latencies experienced by client requests are strongly bimodal. Those requests delivered to resident endpoints are processed quickly, while others experience re-mapping and retransmission delays.

Without the support for event-driven operation, the multi-threaded server would not be implementable. Without resorting to preemptively scheduled threads bound to light-

weight processes, a single user-level thread would either poll forever (wasting cpu cycles), or use time-outs or other clumsy notification mechanisms to wake its up periodically.

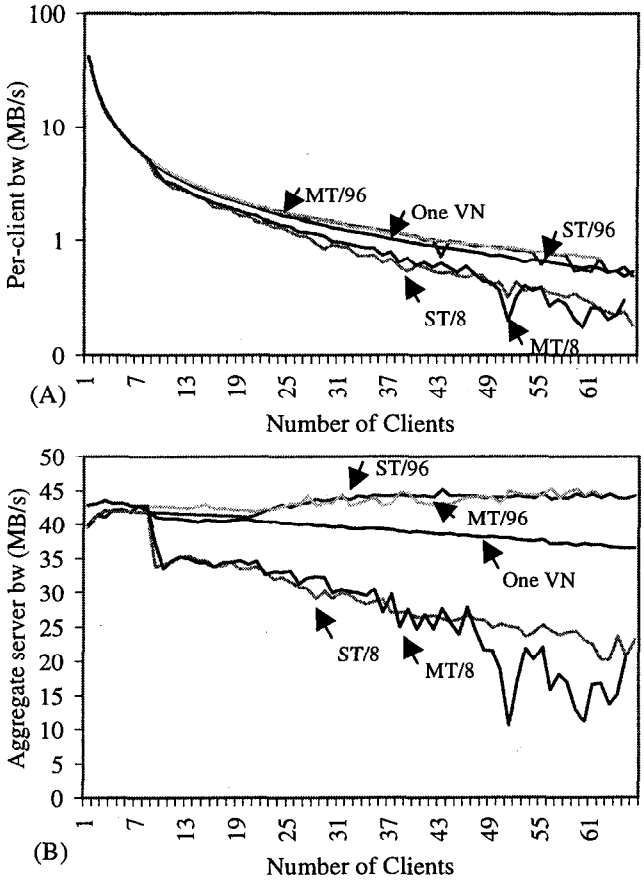


Figure 7. Bulk transfer performance under contention.

Originally, the endpoint management protocol in Section 4.2 did not include the on-host r/w state (transitions to it went directly to the on-nic r/w state). Single threaded servers fell off sharply as soon as endpoint re-mapping began with the 9th client. Only a few percent of the hardware performance was delivered. This was because the server thread blocked for the full-duration of the upload each time it wrote replies into a non-resident endpoint. However, the multi-threaded server did perform well, because it allowed multiple threads to block awaiting re-mappings, while those threads with resident endpoints could always be run and could process incoming requests from their clients.

The single virtual network configuration shows the impact of receive queue overruns, and subsequent retransmissions. Figure 6(b) shows the drop from 75K to 60K msgs/s, from 2 to 3 clients, which occurs when this lightweight mechanism no longer prevents receive queue overruns, and the link protocols begins retransmitting them. Credit-based flow control in the user-level library allows each endpoint to have 32 outstanding Active Message requests because each endpoint's request receive queue is 32 entries deep. Similarly, Figure 7(b) shows the ST and MT configurations with 96 frames outperforming the OneVN configuration for the same reason: with one-to-one "connections," overruns do not occur and retransmissions only result from the destination endpoint not being resident.

7. Related Work

Despite important differences, many similarities exist in contemporary user-level network systems. Most provide processes with direct network access, typically by mapping device hardware into virtual address spaces. The systems optimize the transfer of control, data, and status information between user-level applications and the NI. The operating system is always off the critical path.

Several prototypes have explored the integration of high-speed network adaptors into commodity operating systems. These small-scale systems provided varying degrees of protected multiprogramming. The Osiris project at the University of Arizona created Application Device Channels [15] that provided a small number of memory mappings to an ATM adaptor, using *fbufs* to manage sharing of message buffers between the device and higher-level protocols. The U-Net [30] system illustrated user-level networking with ATM and Myrinet adaptors (and also emulated it using Fast Ethernet). It demonstrated limited-degree virtualization, and rather than addressing protocol issues with the NI, each application provides all of its protocol support at user-level.

In addition to Active Messages, several high-performance communication systems have been realized on both clusters and MPP's. Remote queues [2] provide an interface to message queues with direct control over their servicing. The Illinois Fast Message system [24, 25] is similar to remote queues and Active Messages, and has been implemented successfully on clusters and massively parallel processors. With recent extensions, it, too, supports limited-degree virtualization for parallel programs. Although it assume a perfectly reliable interconnect, this allows careful use of preallocated storage and user-level credits to avoid receiver overruns. (Support for thread-based events is pending.)

The remote memory-mapped regions as used in the Memory Channel system [17] establishes import-export relationships between virtual memory regions in different applications. The SHRIMP project designed Virtual Memory-Mapped Communication [5] where reflective memory channels are established such that writes into an imported segment eventually appear in the exported one. It implemented memory-based message passing across its logical channels, as well as RPC [3] and data stream abstractions [13]. Almost all state resides in host memory, leaving the NI and OS to coordinate the management of device TLB's. The Hamlyn [4] system implemented sender-based communication.

The Virtual Interface Architecture [8] combines elements of these systems. It provides direct user-level network access to multiple applications, supports several different reliability models, and both memory-based and message-based transport primitives. The reference architecture takes a conservative position on memory management, requiring explicit memory registration and pinning before communicating. It specifies a rather complicated model for operating upon descriptors in host memory, as well as several weaker reliability models. Although it uses connections, collections of VI's may share a completion queue which provides a central location for polling. A parallel program on n nodes requires n^2 total VI's for complete connectivity, rather than a single endpoint. Resource provisioning is also done on a connection bases rather than pooling resources across a set.

8. Conclusions

We have demonstrated the feasibility of network virtualization at scale within the existing frameworks of programming interfaces, operating systems, and network interfaces. Our implementation delivers the full hardware performance to parallel applications that run in a stand-alone fashion, and adapts to process scheduling with time-shared workloads. Moreover, it continues to deliver a large fraction of the network performance even with demanding workloads that overcommit the NI resources. The fast, user-level communication that we take for granted in parallel systems can become available to all components.

Because virtual networks require dealing with the interactions between layers of the system, and across nodes in the network, their implementations are challenging. When done carefully, the necessary extensions are localized in scope and reasonable in their demands on processing and storage. The extensions to Active Messages focused on three specific areas, and left its original request/response paradigm intact. The enhancements to Solaris found existing virtual memory management and kernel thread facilities entirely adequate.

Although the NI protocols are non-trivial, we believe that their complexity remains in line with high-performance implementations in hardware. Our implementation was successful even with the significant processing and storage constraints of the LANai. Additional processing power would improve latency and gap, and recover some of the costs of virtualization. It would also enable more sophisticated algorithms, *e.g.,* round-trip times estimation for scheduling retransmissions, or piggybacking acknowledgments to reduce network occupancy.

We are currently working on applying these techniques for network virtualization to an implementation of the Virtual Interface Architecture in order to address a number of limitations from which it currently suffers. The challenges facing large-scale implementations, such as managing a large logical space of VI's given finite interface resources, and the stronger reliable delivery modes given less-than-perfect networks, have analogous solutions to those described herein.

Acknowledgments

The authors thank Bob Felderman at Myricom for his time and support, and Bill Nesheim and Madhu Talluri at Sun Microsystems for their guidance with Solaris. We also thank Eric Anderson, Phillip Buonadonna, Brent Chun, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau for their timely discussions and suggestions on drafts of this paper, and the anonymous reviewers for their comments and feedback.

9. References

[1] N. Boden, D. Cohen, R. Felderman, A.Kulawik. C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit per Second Local Area Network. In *IEEE Micro Magazine*, February 1995.

[2] E. A. Brewer, F. T. Chong, Lok T. Liu, S. D. Sharma, and J. D. Kubiatowicz. Remote Queues. Exposing Network Queues for Atomicity and Optimization. In *Proceedings of the 7th Symposium on Parallel Algorithms, and Architectures*, pages 42-53, Santa Barbara, CA, July 1995.

[3] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. Department of Computer Science, Princeton University Technical Report TR-512-96, February 1996.

[4] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender managed interface architecture. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, pages 245-259, Seattle, WA, October 1996.

[5] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21rst International Symposium on Computer Architecture*, pages 142-153, April 1994.

[6] A. Basu, M. Welsh, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V,* Stanford, CA, August 1997.

[7] C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken, "Low-Latency Communication on the IBM RISC System/6000 SP," Department of Computer Science, Cornell University, September 1996.

[8] Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture Specification Version 1.0. On-line at *http://www.viarch.org*, December, 1997.

[9] D. Culler, L. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. In *IEEE Micro Magazine*, February 1995.

[10] B. Chun, A. Mainwaring, and D. Culler. Virtual Net-

work Transport Protocols for Myrinet. In *IEEE Micro Magazine,* January 1998.

[11] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstain, D. Patterson, K. Yelick. Cluster I/O with River: Making the Fast Case Common. IOPADS '99. May, 1999, Atlanta, Georgia.

[12] A. C. Arpaci-Dusseau, D. E. Culler, A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *1998 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 24-26, 1998.

[13] S. Damianakis, C. Dubnicki, and E. W. Felten. Stream Sockets on SHRIMP. Department of Computer Science, Princeton University Technical Report TR-513-96, October 1996.

[14] J. J. Dongarra, H. W. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites. Technical Report UT-CS-97-365, University of Tennessee, June 1997.

[15] P. Druschel, L. Peterson, and B. Davie. Experiences with a High-speed Network Adaptor: A Software Perspective. In *Proceedings of ACM SIGCOMM '94 Symposium*, August, 1994.

[16] B. Goodheart and J. Cox. The Magic Garden Explained: The Internals of UNIX System V Release 4: An Open Systems Design. New York : Prentice Hall, 1994.

[17] R. B. Gillett. Memory Channel Network for PCI. IEEE Micro, 16:12-18, February 1996.

[18] R. Horst. TNet: A Reliable System Area Network. IEEE Micro, February 1995, vol.15, (no.1):37-45.

[19] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. W.Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms*, pages 272-285, San Diego, California, 1992.

[20] L. Lamport. Time, Clocks, and Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558-565, July 1978.

[21] L. T. Liu. An Evaluation of the Intel Paragon Communication Architecture. Master's report, University of California at Berkeley, Computer Science Department, Berkeley, CA, July 1995.

[22] S. Lumetta, A. Mainwaring, D. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC'97*, November 1997.

[23] R. P. Martin. HPAM. An Active Message Layer for a Network of Workstations. In *Proceedings of Hot Interconnects II*, Stanford, CA, August 1994.

[24] S. Pakin, V. Karacheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. In *IEEE Parallel and Distributed Technology*, 1997.

[25] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations. Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995.

[26] A. Mainwaring and D. Culler. Active Message Application Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[27] H.Tezuka, A. Hori, Y. Ishikawa and M. Sato. PM: A Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking '97*, April 1997

[28] L. Tucker and A. Mainwaring. CMMD: Active Messages on the CM-5. *Parallel Computing*. 20 (1994) 481-496.

[29] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[30] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[31] T. von Eicken, D. E. Culler, S. C.Goldstein, and K. E. Schauser. Active Messages. a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256-266, Gold Coast, Australia, May 1992.

[32] R. Martin, A. Vahdat, D. Culler, T. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the International Symposium on Computer Architecture*, Denver, CO. June 1997.