formed in parallel across them. Alternatively, endpoints could be dynamically bound to interfaces but load-balanced across them over time. Additionally, availability should be considered, e.g., the ability to take NIs off-line and migrate endpoints to remaining interfaces would have implications for all system layers.

Although many contemporary cluster communication efforts recognize the importance of performance and generality, these efforts need better metrics and benchmarks for their evaluation. The ping-pong benchmarks showing latency and bandwidth as a function of message size between two endpoints examines performance in a limited, probably unrealistic, scenario, although some fundamental performance parameters can be obtained. We have examined our system in two additional regimes -- more strenuous many-to-one workloads that exercise the virtualization mechanisms of one specific host, and many-to-many workloads between multiple endpoints in static but non-trivial working sets. Interesting workloads expose dependencies between the communication system and other system components, such as the process scheduling system, making the isolation and measurement of communication phenomena more difficult.

In this system, the network interface directly executes transport operations issued by applications, multiplexes and demultiplxes messages between resident endpoints and the network, and operates within the finite resources granted by the operating system over time. Building upon standard virtual memory management mechanisms, the operating system binds endpoints on-demand to the network interface in a manner transparent to applications above it and consistent with the concurrent operation of the network interface beneath it. And the programming interface provides the vocabularly and semantics of communication between endpoints using active messages for parallel and distributed applications, and at user-level and within the operating system itself. The end-to-end integration of these components allows applications to work within a large logical space of endpoints, while the operating system retains control over resource management and virtualization, and the network interface focuses its hardware on the active endpoints bound by the operating system at any time.

Microbenchmarks and dedicated parallel applications have demonstrated that the system delivers full hardware performance, while workloads that stress the virtualization mechanisms by overcommitting the finite network interface resources continue to receive fair service that gracefully degrades under load. Point-to-point microbenchmarks revealed bandwidths, latencies, and overheads equivalent to a non-virtualized active message system implemented on the same platform. The added computation in the network interface arising from endpoint multiplexing/demultiplexing and the simple transport protocol reduce small message throughput, as compared with a non-virtualized single-endpoint system specialized for parallel programs. However, the costs reflect stronger delivery guarantees, a well-specified error model, and the benefits of virtualization. Results of the Linpack and NAS Parallel Benchmarks show that the system delivers full hardware performance to dedicated applications and speedups comparable to the IBM SP-2, while the timeshared workloads of parallel Split-C applications reported in [dusseau] shows that performance scales proportionally with the number of processes. Finally, stress tests examining virtualization under workloads which overcommit network interface resources show both fairness and graceful performance degradation under load. Network virtualization can retain performance for dedicated programs, which providing a complete, general-purpose communication system available to other applications and all facets of the system at the same time.

This new interface has been used for parallel programming, client-server applications, I/O subsystems [13] and system management and network administration.

10. Discussion

Achieving the performance and generality of virtual networks required selective extensions across several layers of the system. A few, select changes to the Active Message programming interface enable its use in both distributed and parallel applications. Specific extensions to the operating system brought the complete set of virtual memory mechanisms to bare upon the integration of communication and resource management. We found that although the performance and virtualization place additional demands upon network interface processing and resources, solutions exist in-line with the capabilities of contemporary devices. While the limited number of endpoint frames in our system may be unrealistically small, it spurred the development of the general virtualization mechanisms providing a safety net and correctness when resources become overcommitted. Simply upgrading to the current generation Myrinet NI hardware will provide more than 96 resident endpoints per network interface.

We have found the performance of our system competitive with dedicated-mode communication layers, while its generality makes its performance available to a much broader range of parallel and distributed workloads. The porting of applications, message passing libraries, and run-time systems based upon previous active message interfaces has been straightforward. Developers of new applications and software components have appreciated the ability to work within a large logical space of endpoints, and have found operating within a resident endpoint set no more burdensome than using virtual memory. Extensions for multi-threaded applications and event-driven communication, originally motivated by client/server applications, have found widespread usage in I/O libraries, a run-time system for co-scheduling parallel programs, and system management daemons.

Through implementing the virtual network architecture, we have recognized the importance of virtualizing the hardware and software resources for communication both within and across the systems in clusters. While allowing applications to allocate the seven physical endpoints in our NI on a first-come first-serve basis would have been simpler, we sought to virtualize them in a manner that retained performance and afforded the generality offered elsewhere in systems when shared physical resources are virtualized such as timesharing and file systems. For communication, this requires an end-to-end solution between senders and receivers, through intermediate layers of the system. Our approach integrates NI operation with the endpoint management in a manner transparent to correct application execution.

Our results show the feasibility of dynamically managing a working set of endpoints, while providing applications with correct behavior should physical communication resources become overcommitted. By selectively faulting on application and NI references to endpoints, our segment driver binds the most recently active endpoints to the NI. The working set narrows the window of endpoints from which the NI must schedule and transmit packets. The binding of application resources to hardware makes high-level information available to the NI. We use this information to specialize the negative acknowledgments generated for messages arriving for non-resident endpoints, allowing senders to adjust their retransmission frequency given knowledge of page re-mapping costs. We could use additional high-level information to refine working sets to contain only those resident endpoints belonging to the current scheduled process.

In our experience, one of the most contentious aspects of cluster communication layers has been the delivery and error models associated with their transport primitives. Contemporary cluster communication systems exhibit a wide variety of delivery and error models. Messages arriving for a resident endpoints with receive queue space are delivered, messages arriving for resident endpoint with full receive queues or messages arriving for non-resident endpoints are retransmitted, and errant messages arriving for non-existent endpoints or with incorrect keys are returned to their senders. Barring unrecoverable application and transport errors, our system delivers messages. Our results show that performance is consistent with lightweight protocols that recover from link CRC errors, receive queue overruns and other transport conditions. We believe that system should provide a delivery and error model that's independent of how resources are virtualized.

Although we have implemented lightweight protocols in our NIs that are consistent with our transport primitives and their delivery and error models, cluster protocol architectures remain an open area of study. Systems need an active agent to time-out and retransmit messages, while overlapping (re)transmissions with computation and providing efficient atomic access to endpoint contents concurrently accessed by the host and NI. Our NI retains messages until delivered or returned-to-sender. User-level packet time-outs and retransmissions are the obvious alternative but are no panacea in the microsecond time scales of gigabit per second system area networks. Low-cost user-level retransmission introduces complexity.

Our implementation of virtual networks does not address multiple interfaces, but it statically binds endpoints to interfaces and balances endpoint allocations across them. Our design and implementation targeted systems with a single NI per host. Other models for virtualization of multiple interfaces warrant further attention and raise a family of interesting issues. For example, endpoints could span multiple interfaces and operations, e.g., a bulk transfer, could be per-

481-496.

[31] Klaus E. Schauser and Chris 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.

[32] 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.

[33] T. 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, Gold Coast, Australia, May 1992.

[34] 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.

face. Department of Computer Science, Princeton University Technical Report TR-512-96, February 1996.

[5] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John 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.

[6] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan 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.

[7] 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.

[8] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken, "Low-Latency Communication on the IBM RISC System/6000 SP," Department of Computer Science, Cornell University, September 1996.

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

[10] D. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright. The Generic Active Message Interface Specification. University of California at Berkeley, Network of Workstations Project White Paper, February 1994.

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

[12] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *IEEE Micro Magazine,* January 1998.

[13] Remzi H. Arpaci-Dusseau, Eric A. Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the Fast Case Common. Submitted for publication, October 1998.

[14] Andrea C. Arpaci-Dusseau, David E. Culler, Alan 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.

[15] Stefanos Damianakis, Cezary Dubnicki, and Wdward W. Felten. Stream Sockets on SHRIMP. Department of Computer Science, Princeton University Technical Report TR-513-96, October 1996.

[16] Jack J. Dongarra, Hans W. Meuer, and Eric Strohmaier. TOP500 Supercomputer Sites. Technical Report UT-CS-97-365, University of Tennessee, June 1997.

[17] 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.

[18] 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.

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

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

[21] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms*, pages 272-285, San Diego, California, 1992.

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

[23] Lok Tin Liu. An Evaluation of the Intel Paragon Communication Architecture. Master's report, University of California at Berkeley, Computer Science Department, Berkeley, CA, July 1995.

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

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

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

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

[28] 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.

[29] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa and Mitsuhisa Sato. PM: A Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking '97*, April 1997

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

Similarly, round-trip latencies experienced by client requests are strongly bimodal. Those requests delivered to resident endpoints are processed quickly, while others experience remapping and retranmission delays.

With our original 3-state endpoint management protocol, single threaded servers fell off severe performance cliffs as soon as endpoint remapping began with the 9th client. This was because the server thread blocked for the full-duration each time it wrote replies into a non-resident endpoint. However, the multi-threaded server allowed multiple threads to block awaiting remappings, while those threads with resident endpoints were always runnable and could process incoming requests from their clients. Regardless, without the support for event-driven operation, the multi-threaded server would not be implementable. Without resorting to pre-emptively scheduled threads bound to light-weight processes, a single user-level thread would either poll forever (wasting cpu cycles), or use timeouts or other clumsy notification mechanisms to wake its up periodically.

The single virtual network configuration shows the impact of receive queue overruns, and subsequent retransmissions. 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 deep. Figure 6(b) shows the drop from 75K to 60K msgs/s, from 2 to 3 clients, which occurs when this lighterweight mechanism no longer prevents receive queue overruns, and the link protocols begins naking and retransmitting them.

7. Related Work

Despite important differences, many similarities exist in contemporary user-level network systems. Virtually all high-performance communication systems provide applications with direct access to communications hardware using virtual memory mappings. Layers carefully manage the resources along the critical paths, and many optimizations focus upon the low-level handshake between applications and the network interface. The operating system is always off the critical path.

Several small-scale prototypes have explored facets of user-level networking. The Hamlyn [5] system implemented sender-based communication. The Osiris project at the University of Arizona created Application Device Channels [17] providing a small number of mappings to a high-performance ATM adaptor using *fbufs* to manage communication memory. U-Net [32] demonstrated user-level networking on clusters of workstations using ATM, and subsequently Myrinet and Fast Ethernet. All these systems provided protected network multi-programming.

A variety of simple and efficient communication systems have arisen both clusters and MPP's. Remote queues [3] provides an interface to message queues with direct control over their servicing. Active Messages can be viewed as imposing a simple queueing discipline on top of remote queues which avoids deadlock, and in fact, this is the approach taken in our system. The Illinois Fast Message system [27] is an important variation upon remote queues as Active Messages, and has found success both on clusters as well as MPP's.

The remote memory-mapped regions as used in the Memory Channel system [19] establishes import-export relationships between virtual memory regions in different applications. The Princeton SHRIMP project advocates the Virtual Memory-Mapped Communication [6] model where reflective memory mappings are established such that writes into an imported segment eventually appear in the original, exported one. Efficient RPC [4] and data stream abstractions [15] have been implemented using VMMC.

The Virtual Interface Architecture [9] 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 design point in the reference architecture takes a conservative position on memory management, requiring explicit memory registration and pinning before communicating. It also specifies a complicated model for operating upon descriptors in host memory. We expect VIA to serve as a point of consolidation for communication efforts into the future.

8. Conclusions

We have described a novel communication architecture that provides applications with direct network access for performance, and end-to-end virtualization of shared physical communication resources for generality. We have presented its organization in a layered fashion which highlights the fundamental design issues in principle, the implementation details in practice, and the integration of its components in operation.

9. References

[1] T. Anderson, D. Culler, D. Patterson, et al. A Case for Networks of Workstations: NOW. In *IEEE Micro Magazine*, February 1995.

[2] 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.

[3] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John 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.

[4] Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Inter-

per-client throughput, and the latter shows aggregate server throughput. The per-client results demonstrate that the single virtual network configuration, with one server endpoint shared by an increasing number of clients, performs equivalently to the multi-threaded servers regardless of the number of endpoint frames. The aggregate server results show more variation: the multi-threaded server is robust to the number of configured frames. The single virtual network performance is comparable, except for small numbers of clients. Whereas credit-based flow control at user-level prevent receive queue overruns when clients have dedicated server endpoints, it doesn't prevent such overruns with shared endpoints which activate the interface time-out/re-transmission mechanisms. With a single-threaded server, the 8-frame configuration outperforms the 96-frame configuration, because the benefits of polling non-resident but cacheable endpoints in host memory outweighs the costs of polling resident but non-cacheable endpoints on the interface.

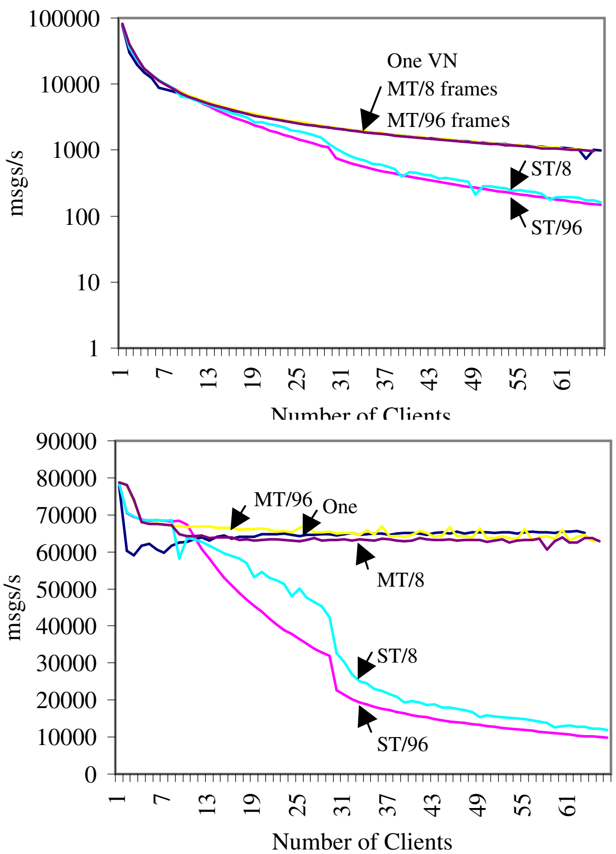


Figure 6. Small message performance under contention.

Figure 7.(a) and Figure 7.(b) show throughput for bulk messages across the same configurations. Both the per-client and aggregate server performance depends strongly the number of endpoint frames, with the 96-frame configuration that avoids endpoint re-mapping outperforming the 8-frame configuration which experiences endpoint re-mapping. Again, user-level credit-based flow control prevent clients from over-running their corresponding server endpoint receive queues, and the single virtual network configuration with a shared endpoint underperforms them. The 8-frame configuration show a sudden drop-off with the 9th client, at which the probability of a client's endpoint being non-resident increases. The local scheduler does become involved in scheduling server threads in the mult-threaded configuration. Threads for client endpoints with pending messages are runnable, and threads suspending during remapping operations also don't prevent others threads from processing client requests.

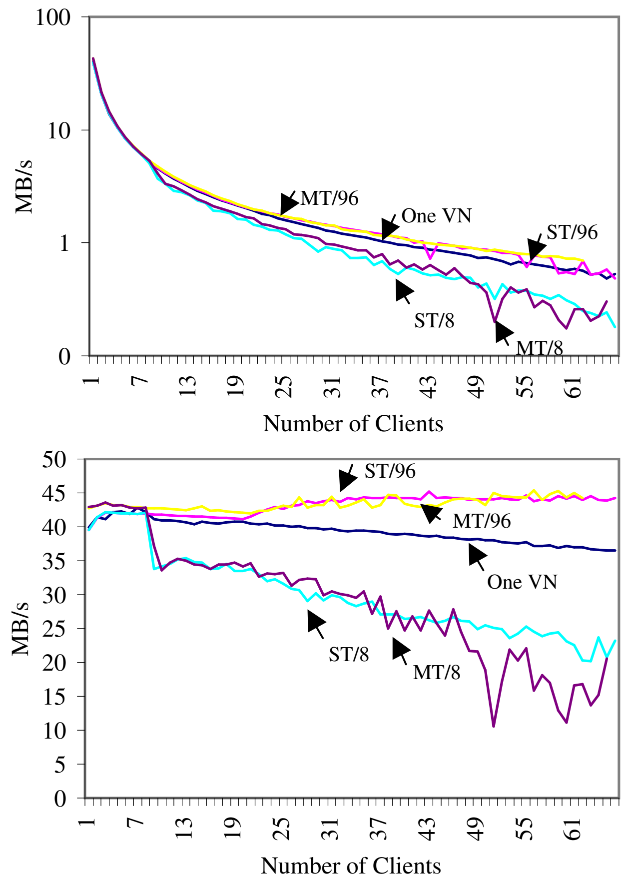


Figure 7. Bulk transfer performance under contention.

These results demonstrate the ability to overcommit NI resources by more than 8:1, yet while providing robust throughput for resident endpoints and fair service for all endpoints over time. Under these workloads, the operating system sustains approximately 200 endpoint re-mappings per second, but still delivers 50%-75% of its performance. Over short intervals, only those clients with resident server endpoints successfully deliver messages. Messages for non-resident endpoints are naked and later retransmitted, while the server interface requests the driver remap the endpoint.

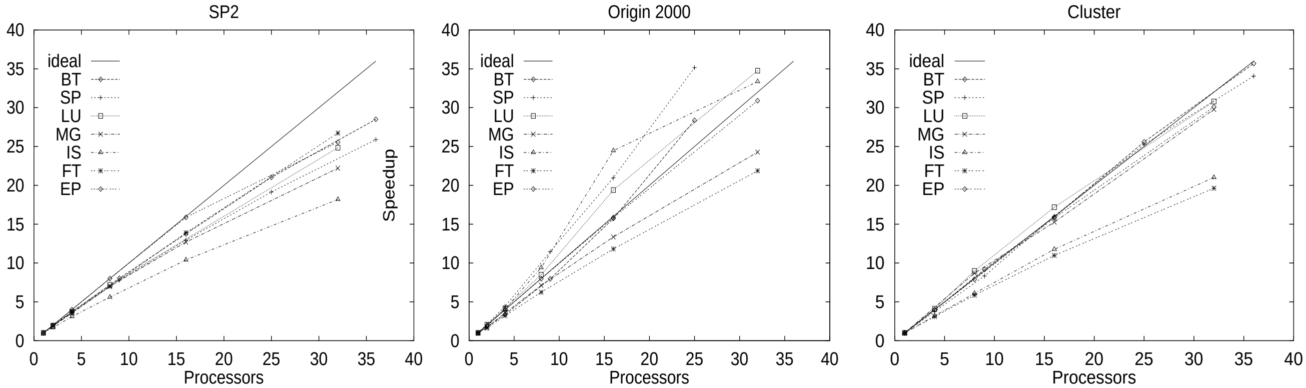


Figure 5. NPB speed-ups through 32 processors on the IBM SP-2, Berkeley NOW and SGI Origin 2000.

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 processors under constant problem size scaling.

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. In general, with time-shared, 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 [21]. Our system uses implicit co-scheduling which coordinates the scheduling of processes within parallel applications using conventional local schedulers. Whatever scheduling discipline is used, however, the scheduler determines which processes to run and the virtual network subsystem adapts the resident set to the active endpoints.

Previously published results in [14] 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%.

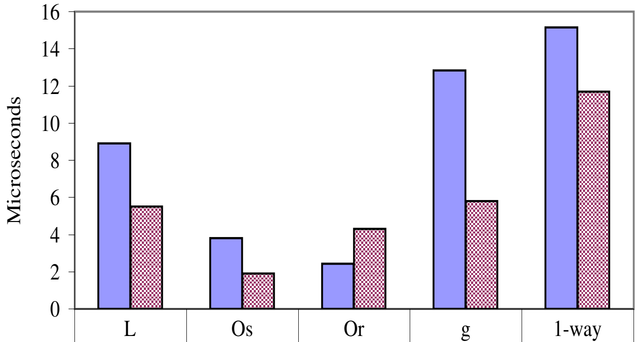
6.4 Virtualization at Scale and Load

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, subtlely 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 were client creates an endpoint which communicates with shared server endpoint, and another where each client communication with its own, unique server endpoint. Third, we consider two natural server threading models, one where a single thread handles requests from all client, and another where there is a dedicated server thread for each endpoint. Each server threads waits for messages to arrive from its client, processes requests until non 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 causes the 8-frame configuration to overcommit its physical endpoints and to begin re-mapping them on-the-fly. (None occurs with the 96-frame setup.)

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 which frequently remain dormant. Therefore, each client sends a continuous stream of requests to its endpoints in the server.

The data presented reflects the average short message rates, and 8KB bulk transfer bandwidths, for five of the possible configurations over a 20 second interval in the steady state. Figure 6.(a) and Figure 6.(b) show throughput for small messages across different configurations. The former shows

for both virtual networks and a first-generation interface for single parallel applications on our cluster. In the LogP model, the sending and receiving overheads, O_s and O_r, account for the processor time spent writing or reading a message to the network interface, respectively. The gap, *g*, accounts for the time per (16-byte) message through the rate limiting communications pipeline stage between two endpoints and the latency, *L*, accumulates remaining time. Messages can be sent every g time units, incur an overhead of $O_s + O_r$, and an end-to-end latency of $O_s + O_r + L$.



	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

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..

This shows that virtualization, and the increased demands placed upon the network interface, increased round-trip times by 23%, and increased the *g*ap by a factor of 2.21, while the total per-packet overheads remains the same. The larger gap is due primarily to the link-level protocol and acknowledgment processing. Other aspects of virtualization, such as error checking and defensive firmware practices, contributing only 1.1 *usec*. Sensitivity studies [34] show that increases in gap is in general less detrimental than increases in overheads, because increasing the former effect applications which send small messages more often than *g usec*. The larger send overhead reflects the cost of issuing larger message descriptors to the NI, but the smaller receive overhead shows the benefit of reading an entire descriptor across the SBUS with a single SPARC VIS block load instruction.

Virtualization has no appreciate performance impact for bulk transfers. The system delivers 43.9 MB/s with 8KB messages, with an $N_{1/2}$ of 540 bytes. Round-trip latency for *n*-byte bulk transfers, with *n* >= 128 bytes, takes $time = 0.1112(n) + 61.02\ usec\ (R^2 = 0.9999)$. Figure 4. shows the delivered bulk transfer performance approaches 93% of the peak performance of the hardware. The SBUS exhibits asymmetric direct memory access transfer rates when reading and writing host memory from the interface. Thus, the

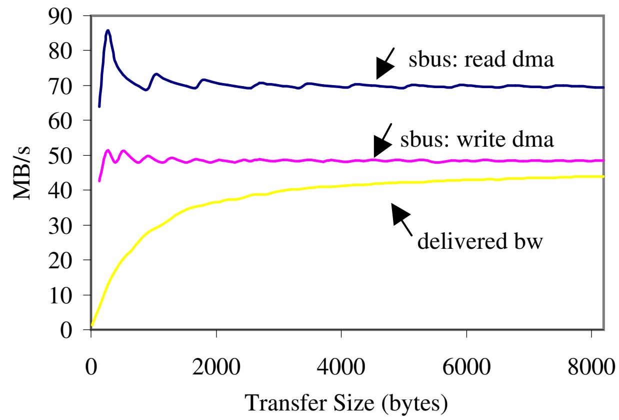


Figure 4. Tramsfer bandwidths.

Microbenchmarks showing delivered bandwidth for 128 byte to 8192 byte messages, with the average in-

delivered performance nears the 46.8 MB/s maximum for DMA transfers from the interface to the host. Although the NI pipelines its processing of message descriptors to compensate for the store-and-forward delay when staging bulk transfers through its on-board memory, the transfer time across the SBUS dominates the additional packet and protocol processing arising from virtualization.

6.2 Dedicated Parallel Applications

The performance of individual parallel applications using the system in a stand-alone MPP fashion is shown on several benchmarks. The next two results shows that the costs of virtualization in the communication substrate does not compromise the performance of an individual application significantly (while allowing greater generality and robustness.)

Using the public-domain Scalapack library, optimized BLAS routines from Sun Performance Library, and our port of the standard MPICH on active messages, our cluster sustained 10.14 GF on the massively-parallel linpack benchmarks, making it the first cluster on the Top-500 list [16], 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, 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. 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. Not only did instrumentation reveal that the relative time spent performing communications is lower on the cluster, but in

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 [12] and are only summarized here. The 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 acknowledgment indicate the message was written into the destination endpoint, and negative acknowledgments carry information describing why the message was not 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 interface carefully unbinds messages from channels enabling their re-use; subsequent retransmissions reacquire and rebind them.

5.2 Service and Queueing Discipline

Because the NI services multiple resident endpoints, its service and queueing discipline must 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. Incoming packets are received into a single shared queue that requires further higher level processing before delivery to applications. Architectures that provides direct user-level messaging necessarily collapse such intermediate layers of interpretation. Because the system binds endpoints directly to it, the NI must provide a discipline for scheduling 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 endpoints with packets awaiting (re)transmission. While packets remain to send, the interface processes up to 64 (the maximum number of posted messages per endpoint) descriptors for a maximum of 4*ms* (the approximate time to send 64 messages of the maximum transmission unit size at full link rate) before servicing all other endpoints. This discipline enables the interface to cache endpoint-specific state, and optimize the latency and throughput for an individual endpoint, while the time attenuates endpoints sending large messages from receiving unfair proportion of attention. The in-order transmission of packets and their out-of-order acknowledgments requires that departing from the first-in first-out queueing policy for retransmissions. *Retransmission requires fine-grain access to endpoint descriptors and their internal components for small message performance. It would be costly if descriptors were maintained in host memory.*

5.3 OS and Protocol Integration

The operating system and network interface operate concurrently yet synchronize some operations upon endpoints. The fundamental complication arises when the driver attempts to unload or invalidate an endpoint while it 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.

Putting an endpoint with unacknowledged messages into a quiescent state adds transient states to the finite state machine that controlling the interface operation. These states prevent new messages from being sent, while periodically retransmitting unacknowledged packets until acknowledgments are received. However, 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 simplify, 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 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 that evaluate performance in three distinct regimes, and reflects upon the design choices made. It starts with microbenchmarks that reveal the impact of virtualization upon the base overheads, latencies, and bandwidths. It uses both dedicated and time-shared parallel applications performance that operate well within the capabilities of the underlying network, and finishes with workloads that examine robustness with workloads that overcommit resources.

6.1 Stand-alone Microbenchmarks

The LogP microbenchmark in Figure 3. characterizes active message performance, using a technique described in [11],

translations, and endpoints paged to disk are marked invalid (n/a). The 'vm pageout' transitions refers to page reclamations that migrate pages to secondary storage.

An endpoint initially resides in the on-host r/o state. Writes by applications (or references by the network interface) generate endpoint faults which transitions 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 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.

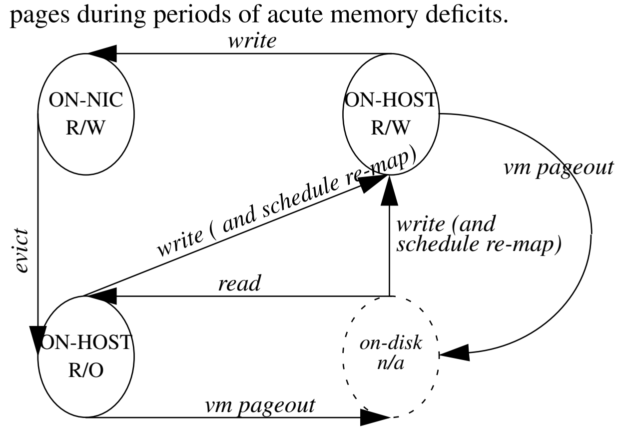


Figure 2. Endpoint management protocol for Solaris VM system.

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 remapping load. This state de-couples process scheduling and binding of endpoints to the NI. It allows the application thread to resumes execution; later the system atomically unmaps the endpoint, relocates its backing store to the NI, and updates its virtual address translations. Normally, pagefault are handled synchronously and in their entirety, while the faulting application thread remains suspended. The segment driver uses background kernel thread to activates non-empty endpoints, asynchronously to their initial fault handling.

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 segment driver mechanisms.

4.3 Network Interface Integration

It is critically important that the segment driver operates concurrently with the network interface, yet for these two agents to synchronize their operations upon shared objects. Both the operating system and NI operate asynchronously, and each makes requests to initiate operations in the other. This raises three core issues: the means through which the operating system and NI communicate, the protocol that synchronizes their interactions, and how each agent performs operations on behalf of the other. Here we address the communication, protocol, and kernel operations; NI operations on behalf of the driver are discussed in Section 5.3.

The segment driver and the NI are peers that communicate using a simplified form of Active Messages through a dedicated system endpoint that is permanently resident. The simple protocol through which they interact are defined by a small number of management handlers in each agent. The driver may allocate an endpoint, free an endpoint, load an endpoint from host memory into an endpoint frame, or unload an endpoint from an endpoint frame into host memory. In each case, it invokes an associated handler in the NI to perform part of the operation. The interface may request that the endpoint segment driver make an endpoint resident, generate an endpoint event, or manipulate DMA mappings to application memory regions. *In addition, a variant of logical clocks [22] imposes an ordering upon their messages so that each agent can detect corner conditions, e.g., when concurrently the driver attempts to free an endpoint while the interface 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 upon critical hardware resources.Because multiple independent flows are bound to the network interface, instead of muxed into a single flow by the operating system, the interface faces three core challenges: the efficient implementation of packet transmission 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

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. Although applications may form arbitrary thread-endpoint relationship, they remain responsible for remaining attentive to endpoints either by polling or waiting on events in order to make forward progress.

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.,* [10, 29]. 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 without source-level modifications to Solaris. The key is a generalized 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. What is unique is that the arrival of a message for a non-resident endpoint has the same consequences. 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 communication transfers are staged; data can be moved between the host and NI memory or between NI memory and the 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* [18]. 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 address translations and backing store. This four-state endpoint management protocol shows endpoint states and transitions. Endpoint reside in uncacheable endpoint frames on the network interface (on-nic), cacheable main memory frames (on-host), or in the swap area (on-disk). Endpoints in memory have read-write (r/w) translations, endpoints in host memory can have either read-write or read-only (r/o)

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 it is connection-oriented and also imposes 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 [32, 30, 31, 23, 8] and clusters [25, 32, 27]. 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 development: 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 casts [28] 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 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, 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 an 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 defensive actions in the common case, such as setting time-outs and logging message contents. *This "return to sender" model allows the underlying, 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, polling is more efficient

tural components. Each section outlines the general principles and the key issues in practice. Section 3 addresses the key programming interface provisions to support general purpose use, including naming, protection, delivery models and error handling, and integration with threads and events. Section 4 examines the key operating system support in the context of Solaris, including integration of the virtual memory system with the network interface. Section 5 examines network interface support for virtualization, including the service and queuing disciplines, and the transport protocols employed. Section 6 discusses system performance in three regimes using microbenchmarks, applications, and macrobenchmarks that examine robustness. Finally, Section 7 discusses lessons learned and related works.

2. Background

Approaches to network virtualization have been proposed in the form of Abstract Device Channels [17], Remote Memory Mapped Regions and reflective memory [6, 19], Remote Queues [3], and Active Messages [28] and several prototype cluster systems have been developed [6, 27, 1]. Indeed, the results have been promising enough that a major industrial consortium recently released the Virtual Interface Architecture [9] 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 mapping network hardware into one application's address space. However many tests stress little of the network virtualization, leaving its effectiveness at scale a largely open question. In developing this system, we found that many of the critical design issues arose from dealing with the requirements of full-scale use and are not revealed by simple benchmarks.

We wanted to understand network virtualization within the hardware and software framework provided by Myrinet and Solaris. We sought to understand the system architecture, the innovations and extensions of its core components, and their integration within and between nodes of the system that realize efficient network virtualization. Therefore, rather than building custom network interface hardware, we specialized the functionality of the network interface using its embedded processor. Its slow, resource-constrained implementation severely constrained its responsibilities. Although we were philosophically willing to make arbitrary modifications to Solaris, we wanted to know if effective virtualization was realizable within the existing framework of a modern UNIX operating system. With the programming interface, we sought to understand the minimal extensions to an existing Active Message interface necessary for general-purpose, multi-programmed use.

While technological influences influenced our approach from the bottom-up, the desire to support the user-level and system software hierarchies shown in Figure 1 guided from the top-down. We sought to understand how their requirements from these diverse parallel and distributed applications would influence underlying system design and implementation. An Active Message programming interface supports traditional libraries and run-time systems, such as MPI and Split-C, as well as novel parallel I/O system and remote method invocations within Java. These in turn support parallel and distributed application, as well as novel cluster and Internet services. By supporting a subset of the programming interface within kernel, standard facilities such as sockets, network files systems, and remote-procedure calls packages, can leverage the performance of the high-speed interconnect. 6Applications such as GLUnix[], Condor[], and those using sockets can leverage it.

The cluster has 100 167-Mhz Sun UltraSPARC-1 workstations with 128 MB running Solaris 2.6, interconnected by a Myrinet network with 25 switches and 185 links configured in a fat-tree like topology. Switches have an average cut-through latency of ~300*ns* and 1.2 Gb/s bi-directional links. Each LANai 4.3 network interfaces contain a general-purpose embedded processor running at 37.5-Mhz with 1MB of on-board memory, independent network send/receive DMA engines, and a single DMA engine for SBUS data transfers.Network paths between interfaces quite shallow. While link-by-link flow control and back pressure contribute to low overall transmission error rates, network congestion rapidly spreads through the network. Although the NI contains a general-purpose microprocessor, it implements a controller amenable to implementation in hardware.

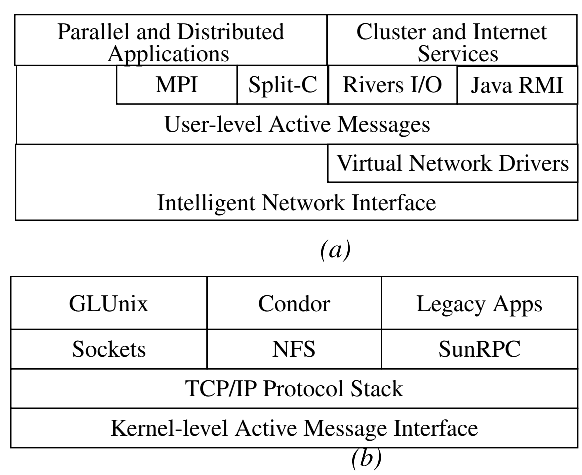


Figure 1. (a) user-level software and (b) system software heirarchies currently operational using Virtual Networks and Active Messages,

Design Challenges of Virtual Networks

Alan M. Mainwaring
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776
(510)-642-8299
alanm@cs.berkeley.edu

David E. Culler
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776
(510)-642-7572
culler@cs.berkeley.edu

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 our large-scale virtual network system, and examine the integration of communication programming interface, operating system resource management, and network interface operation. Our results from an complete implementation on a cluster of 100 workstations show: *[1] virtualization only affects the gap and latency, [2] full hardware performance delivered to dedicated applications and time-shared workloads, and [3] robustness under loads.*

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. 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 [2, 20] to a broader range of applications.

The implementation of the high-speed communication substrate should broaden, and not determines, 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 [5, 7, 17, 25, 26, 29, 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 of t*he system both for the operating system, its network file systems, process schedulers, and sockets within the network, as well as for user-level processes, to debuggers and performance analyzers, to parallel clients and servers, and for traditional client/server applications.*

Generally, the way to obtain performance and generality is virtualization. Systems 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 provides generality and completeness by facilitating the sharing, scheduling, and presentation of physical resources to their consumers, rather than maximizing performance for one specific usage model, such as stand-alone use.* However, when done well, it retains most of the performance afforded by direct application-to-resource bindings. 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 excessive intermediate interpretation and management occur even when a single user program operates well within the capability of the underlying network.

This work makes three fundamental contributions: (1) it describes the design, implementation, and measurements of a complete, large-scale virtual network system in daily use for more than a year on a cluster one hundred workstations serving a diverse user community, (2) it shows efficient network virtualization that delivers full hardware performance to dedicated applications, and robust performance under workloads that overcommit communication resources, *and (3) that integration across the programming interface, operating system, and the network interface (NI) enables efficient virtualization within their existing 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 architec-