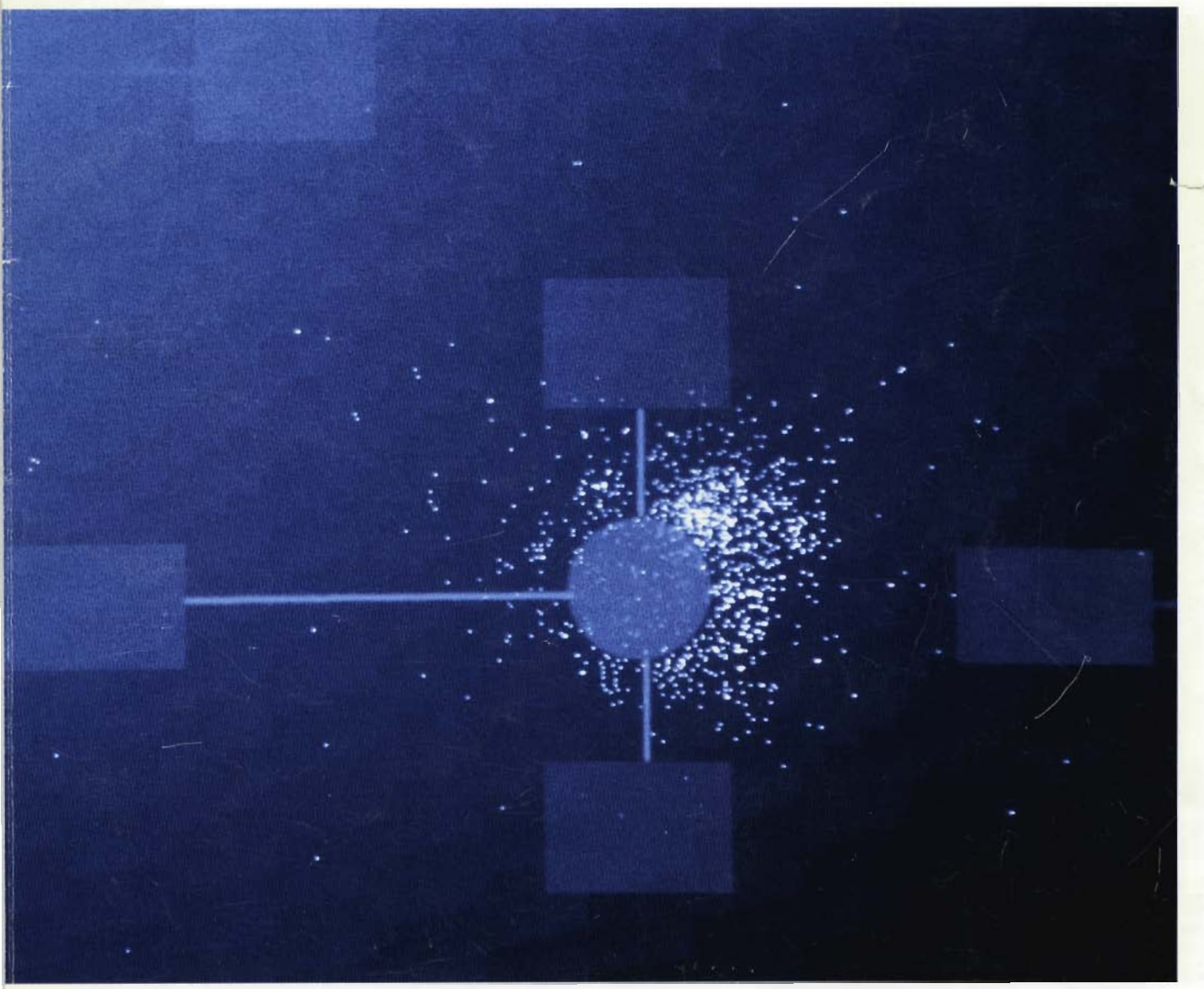
VAXcluster Systems

Digital Technical Journal

Digital Equipment Corporation



Number 5

September 1987

The *Digital Technical Journal* is published by Digital Equipment Corporation, 77 Reed Road, Hudson, Massachusetts 01749.

Changes of address should be sent to Digital Equipment Corporation, attention: Media Response Manager, 444 Whitney Street, NRO2-1/J5, Northboro, MA 01532-2599

Comments on the content of any paper are welcomed. Write to the editor at Mail Stop HL02-3/K11 at the published-by address. Comments can also be sent on the ENET to RDVAX::BEANE or on the ARPANET to BEANE%RDVAX.DEC@DECWRL.

ISBN 1-55558-004-1

Documentation Number EY-8258E-DP

Book production was done by Educational Services Media Communications Group in Bedford, MA.

Cover Design

VAXcluster systems are featured in this issue. The central connection between the elements in a cluster is called the Star Coupler. Our star-filled cover evokes the thousands of VAXcluster systems now operating worldwide. The image was created using the Lightspeed System.

The cover was designed by Barbara Grzeslo and Tim Roberts of the Graphic Design Department.

Contents

VAXcluster Systems

Editor's Introduction

Richard W. Beane
Editor

VAXcluster systems are closely coupled configurations of VAX CPUs and storage devices. The VAX CPU at any node can communicate with the processor and storage devices at any other node in the cluster. The interconnects and software used to activate this unique concept allow data transfers at up to 70 megabits per second between nodes. This issue of the *Digital Technical Journal* contains papers about some of the key hardware and software features in these systems, as well as some measures of their performance. Since several organizations within Digital are responsible for various VAXcluster features, these papers are contributed by engineers from a wide spectrum of engineering groups.

Since the VAXcluster concept spans such a range of technologies, the first paper is an overview explaining generally how these systems work. Nancy Kronenberg, Hank Levy, Bill Strecker, and Richard Merewood describe the architecture, the storage control, the VMS software alterations, and the multitude of activities that control access to the storage devices.

The System Communication Architecture, described by Darrell Duffy, is the structure that allows the nodes in a VAXcluster system to cooperate. This relatively simple framework governs the sharing of data between resources at the nodes and binds together applications that run on different VAX CPUs.

Additional features were needed in the VMS software to accommodate accessing disks on multiple systems. The distributed lock manager, described by Sandy Snaman and Dave Thiel, provides the synchronization needed to accomplish transparent data transfers between cluster members. Other changes were also needed to broaden the file functions performed by the VMS software. Andy Goldstein relates some alternative ways to expand those functions and how the QIO processor was extended to synchronize file accesses. The resulting system of locks and queues provides a consistent sequence for managing distributed files.

The next paper, by Mike Fox and John Ywoskus, describes the extension of the VAXcluster concept to systems connected with an Ethernet. These Local Area VAXcluster systems use special software to provide functions needed by clusters, but not provided by Ethernet software. Thus, MicroVAX II and other small VAX systems can be clustered to yield significant amounts of processing power.

The last three papers deal with performance aspects of VAXcluster systems. The paper by Ed Balkovich, Prashant Bhabhalia, Dick Dunnington, and Tom Weyant discusses the results of a VAXcluster model that demonstrates how redundancy improves availability. Then, Dale Park, Rekha Von Ehren, T-J. Wang, and Nii Quaynor describe two models they developed to measure the performances of VAX 8974 and 8978 systems. These models, based on benchmarks run in different environments, use a VAX 8700 CPU for a baseline comparison.

The final paper relates the results of a model to measure the characteristics of the CI bus. Xi-ren Cao, Nii Quaynor, and Fernando Colon Osorio describe how their model measures the performance of the arbitration algorithm in this bus. They suggest some interesting schemes to improve utilization and reduce response time.

Dick Beane

Biographies

Edward E. Balkovich Ed Balkovich is the manager of VAXcluster System Engineering, which addresses issues of VAXcluster performance, availability and architecture for High Performance Systems. He was Digital's associate director of Project Athena at M.I.T. and is an Adjunct Associate Professor at Brandeis University. Before joining Digital in 1981, Ed was a faculty member at the University of Connecticut. He earned his B.A. degree (1968) from the University of California at Berkeley, and his M.S. (1971) and Ph.D. (1976) degrees from the University of California at Santa Barbara. He is a member of the ACM and IEEE.

Prashant Bhabhalia A principal engineer in VAXcluster Systems Engineering, Prashant Bhabhalia develops and interprets reliability and availability models. Earlier, he was a program manager in Computer Systems Manufacturing and a senior engineer in GIA Manufacturing. Before joining Digital in 1980, Prashant was an industrial engineer at Norton Company and Gits Plastic Corporation. He holds an M.S.I.E. degree (1974) from the Polytechnic Institute of Brooklyn and a B.S.M.E. degree (1972) from the M.S. University in India. Prashant is a senior member of I.I.E.

Xi-Ren Cao As a principal software engineer in the High Performance Systems and Clusters Group, Xi-Ren Cao models and evaluates VAXcluster configurations. Before joining Digital in 1986, he was a research fellow at Harvard University. Xi-Ren has published over 20 technical papers on performance evaluation, simulation, stochastic systems, queuing networks, and control theory, and has co-authored a book "Perturbation Analysis of Discrete Event Systems," to be published in 1988. He received his Ph.D. degree from Harvard University in 1984 and is a member of IEEE.

Fernando C. Colon Osorio Fernando Colon Osorio graduated from the University of Puerto Rico (B.S.E.E., 1970) and the University of Massachusetts (M.S., Ph.D., 1976). Joining Digital in 1976, he helped design the PDP-11/60 and PDP-11/74 systems and managed the LAN group in Corporate Research. Fernando also managed the overall design verification for the VAX 8600 project. In High Performance Systems, he now manages the systems research and advanced development group, responsible for VAXclusters, fault tolerance, advanced architectures, and performance analyses. He was Associate Editor of the IEEE Transactions on Computers and is the co-author of "Engineering Intelligent Systems."

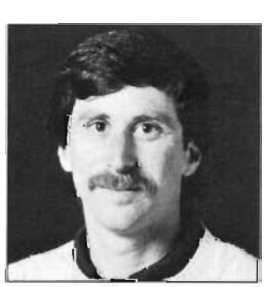
Darrell J. Duffy As a consulting software engineer, Darrell Duffy works on the network architecture for VAXcluster systems. On previous projects, he led the development of operating systems for parallel processors and wrote software for the Local Area Terminal protocol. Darrell helped to develop DECnet software after joining Digital in 1977. He received a B.S. in computer science from West Virginia University in 1972 and worked at the University of Florida. Darrell and three other Digital engineers have applied for a patent on the LAT protocol.

William R. Dunnington Dick Dunnington is a principal quality engineer working on availability modeling in the Computer System Manufacturing Group. Previously, he was a quality engineer in the Far East Manufacturing Group, working on personal computer memories. Before joining Digital in 1979, Dick was a captain in the U.S. Army. He received an Associates degree in liberal arts from S.U.N.Y. (1973) and a B.S. degree in engineering science from the University of Nebraska (1974). Dick, a member of SIAM and ASQC, is also a Certified Quality Engineer.

Michael S. Fox In 1977, Mike Fox joined Digital after earning his M.S. (1977) and B.S. (1976) degrees in computer science from Rensselaer Polytechnic Institute. Initially, he helped to develop the RSX11M-PLUS software, then served as architect and supervisor on the PRO/SERVER project. In Digital's Graduate Engineering Education Program, Mike returned to Rensselaer for a year as a faculty member in computer science. Back at Digital, he joined the VMS Engineering Group and lead the project that developed the Local Area VAXcluster software. Mike is now a consulting software engineer.

Andrew C. Goldstein Andy Goldstein received his B.S.E.E. and M.S.E.E. degrees from M.I.T. in 1971, and joined Digital in 1973. He was initially responsible for the file system in the RSX-11D and RSX-11M systems, and became a charter member of the VMS Development group. Andy designed and implemented the VMS file system, and worked as well on the VMS I/O and executive software. More recently, he designed the security features in VMS version 4.0 and helped with the VAXcluster file system. Andy is now a senior consultant software engineer, and is a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and ACM.

Nancy P. Kronenberg Nancy Kronenberg is a senior consultant software engineer in the Advanced VAX Development Group. She is currently project leader of the microcode team for a new VAX CPU. Previously, Nancy worked in the VMS Development Group where she assisted with the SCA specification and wrote the CI port driver and part of the VMS SCA services. Before joining Digital in 1978, she was a systems analyst at Massachusetts Computer Associates and at Applied Data Research. Nancy earned her AB degree in physics from Cornell University in 1967.



Henry M. Levy A consultant engineer on leave from Digital, Hank Levy is currently an Assistant Professor working on distributed systems and computer architecture research at University of Washington. Hank joined Digital in 1974. He was a member of the original VAX/VMS team and later worked for the VAX Architecture Group on interconnect and workstation architectures. He has published over a dozen papers and the books Capability-Based Computer Systems and Computer Programming and Architecture: The VAX-11. Hank holds a B.S. degree (1974) from Carnegie-Mellon University and an M.S. degree (1981) from University of Washington.

Richard J. Merewood Richard Merewood is the software development manager for the DECnet-VAX, Local Area VAXcluster, and VAXcluster software projects. In Reading, England, he managed the development of Digital's X.25 networking products, performed advance development on the ISDN project, and supervised a modem development project. Before joining Digital in 1980, Richard was an international consultant in data communications and transaction processing. He studied electrical engineering at the Imperial College of Science & Technology, London.

Daeil Park As a principal software engineer in the Systems Performance Group, Dale Park executes and analyses tests to determine VAXcluster performance. He is particularly involved with measuring the performance of application programs on these systems. Dale joined Digital in 1983 after receiving his M.S. degree in computer engineering (1983) from Case Western Reserve University. Earlier, he was a system design engineer at Samsung Electronics Co. Ltd, in Korea. Dale earned his B.S. degree in electrical engineering (1977) from Seoul National University in Korea.

Nii N. Quaynor After earning his B.E. degree from Dartmouth College in 1973 and his Ph.D. from S.U.N.Y. at Stony Brook in 1977, Nii Quaynor joined Digital in 1978. He first worked in corporate research on multimicro systems. In 1982, Nii joined the VAX 8600 project as a consulting software engineer and created models for large-scale CAD applications using a register transfer language. Later, he worked on the verification of the VAX 8600 design. Nii is now the manager of the System Performance Group in High Performance Systems.

William E. Snaman, Jr. Sandy Snaman is a principal software engineer in the VMS Development Group, currently working on software for VAXcluster systems and the distributed lock manager. Sandy has also developed and taught VAXcluster courses in Educational Services and was a software maintainability engineer for Customer Services Systems Engineering. He joined Digital in 1980 after eight years in the U.S. Navy. Sandy holds a B.S. degree (1985, Magna Cum Laude) from the University of Lowell, where he is now completing his M.S. degree in computer science.

William D. Strecker Bill Strecker, vice president for Product Strategy and Architecture, joined Digital after receiving his B.S., M.S., and Ph.D. degrees from Carnegie-Mellon University. Bill's work on cache memories led to the PDP-11/70 system, and he also led the team that developed the VAX architecture. Bill guided Digital's interconnect strategy, which lead to the computer interconnect (CI) and the Systems Communication Architecture. He holds several patents on CPU designs and computer interconnects. Bill and was elected to the National Academy of Engineering in 1986.

David W. Thiel Dave Thiel, a consulting software engineer, is currently studying future directions for VAXcluster systems in the VMS Development Group. He was project leader for the initial VAXcluster support in VMS version 4.0. Dave also worked on the executive and data compression areas of the VMS software. Dave joined Digital in 1980 from GenRad, Inc., where he was a principal software engineer. He earned his B.S.E.E., M.S.E.E., and Electrical Engineer degrees from M.I.T. in 1972. He is a member of Tau Beta Pi, Eta Kappa Nu, ACM, and IEEE.

Rekha D. Von Ehren As a senior software engineer in the Systems Performance Group, Rekha Von Ehren works on performance measurements and analyses for VAXcluster systems. Previously, she analyzed the performance of VAX 8600 and 8650 CPUs. Rekha joined Digital in 1983 after receiving her M.S. degree in industrial engineering from the University of Wisconsin. She also earned an M.S. degree (1981) in operations research from the London School of Economics and a B.S. degree in statistics and computing from North London Polytechnic. Rekha has just given birth to her first child, a baby boy, named Samuel.

Tzyh-Jong Wang As a principal engineer in the Systems Performance Group, Tzyh-Jong Wang conducts modeling studies to measure system performance. He analyzes VAXcluster configurations, on-line transaction processing, and other advanced systems. Before joining Digital in 1987, Tzyh-Jong was a lecturer at the University of Wisconsin at Madison, where he received his M.S. and Ph.D. degress (1987) in information systems. He also earned a B.S.I.E. degree (1978) from the National Tsing-Hua University, Taiwan. Tzyh-Jong is a member of ACM, IEEE, ORSA, and TIMS.

Thomas F. Weyant Tom Weyant is the manager of the Systems Reliability Engineering Group in Computer Systems Manufacturing. As a consulting engineer, he worked on systems reliability and availability modeling, computer-interconnect reliability, infant-mortality and long-term failure-rate modeling, and was the manager of advanced development. Before joining Digital in 1985, Tom worked for ten years at AT&T Bell Laboratories and Hughes Aircraft Company. He earned his B.S.M.E. degree (1975) from the University of California at Santa Barbara, and his M.S. and Ph.D. degrees (1981) in operations research from UCLA.

John A. Ywoskus John Ywoskus is a principal software engineer with the VAX/VMS Development Group. He is currently project leader of the Local Area VAXcluster development effort and was lead technical contributor in the development of the first release of this product. Before joining the VMS group in 1985, John worked as a developer on the LAT-11 terminal server project and as project leader of the LATplus V1.0 application terminal project. John came to Digital in 1981 from the Charles Stark Draper Laboratory, where he worked on CAD system software development. He earned a B.S. degree in Applied Mathematics from Harvard College in 1979.

Nancy P. Kronenberg
Henry M. Levy
William D. Strecker
Richard J. Merewood

The VAXcluster Concept: An Overview of a Distributed System

A VAXcluster system is a highly available and extensible configuration of VAX computers that operate as a single system. To achieve high performance in a multicomputer environment, a new communications architecture, communications hardware, and distributed software had to be jointly designed. The software is the VAX/VMS operating system, using a distributed lock manager to synchronize access to shared resources. The communications hardware includes a 70-megabit per second message-oriented interconnect, and an interconnect port that performs communications tasks traditionally handled by software. The Local Area VAXcluster system, an implementation of the VAXcluster architecture, uses a standard Ethernet as its interconnect. This development provides VAXcluster functions for the MicroVAX family.

Contemporary multicomputer systems typically lie at the ends of the spectrum delimited by tightly coupled multiprocessors and loosely coupled distributed systems. Historically, loosely coupled systems have been characterized by the physical separation of processors, low-bandwidth message-oriented interprocessor communication, and independent operating systems.[1,2,3,4] Conversely, tightly coupled systems have been characterized by close physical proximity of processors, high-bandwidth communication through shared memory, and a single copy of the operating system.[5,6,7]

An intermediate approach taken at Digital Equipment Corporation was to build a "closely coupled" structure of standard VAX computers,[8] called a VAXcluster system. By closely coupled, we imply that a VAXcluster system has characteristics of both loosely and tightly coupled systems. On one hand, a VAXcluster system has separate processors and memories connected by a message-oriented interconnect, running instances of the same copy of the distributed VAX/VMS operating system. On the other hand, the initial implementation of the cluster relied on close physical proximity, a single (physical and logical) security domain, shared physical access to disk storage, and high-speed memory-to-memory block transfers between nodes.

The goals of the VAXcluster multicomputer system are high availability (in suitable configurations) and easy extensibility to a large number of processors and device controllers. In contrast to other highly available systems,[9,10,11,12] a VAXcluster system is built from general-purpose, off-the-shelf processors ranging in size from MicroVAX workstations[13] to high-performance VAX CPUs, and a general-purpose operating system.

A key concern in this approach is system performance. Two important factors in the performance of a multicomputer system are the software overhead of the communications architecture and the bandwidth of the computer interconnect. To address these issues, several developments were undertaken as part of the original VAXcluster design, including

- A simple, low-overhead communications architecture whose functions are tailored to the needs of highly available, extensible systems. This architecture is called the System Communication Architecture (SCA).
- A very high speed message-oriented Computer Interconnect, called the CI bus

The original version of this paper appeared in "VAXclusters: A Closely-Coupled Distributed System," by Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker, published in *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986.

- An intelligent hardware interface to the CI bus, called the CI port, that implements part of the SCA in hardware
- An intelligent, message-oriented mass storage controller that uses both the CI bus and the CI port interface

This combined software and hardware architecture supports a high-performance communications structure for interconnecting high-performance VAX systems. For low-end VAX CPUs, the Local Area VAXcluster system has been developed to permit workstations interconnected by means of the Ethernet to share a common file system, printers, and batch processing. Workstation users can derive the benefits of centralized timesharing without sharing a CPU and without system management overhead. A Local Area VAXcluster system is supported by software that emulates some of the CI functions, thus making the difference between CI-based and Ethernet-based VAXclusters largely invisible to higher level software. Local Area VAXcluster systems can be formed from and coexist with existing Ethernet networks without the need for special-purpose hardware.

This paper describes the communications hardware developed for VAXcluster systems, the hardware-software interface, the Local Area VAXcluster system, and the structure of the distributed VAX/VMS operating system. The developments described in this paper are part of Digital's VAXcluster product; there are, as of mid-1987, approximately 6,000 VAXcluster and Local Area VAXcluster systems in operation.

VAXcluster Hardware Structure

The CI-based VAXcluster System

Figure 1 shows the topology of a typical CI-based VAXcluster system. The components include the CI bus, VAX hosts, CI ports, and Hierarchical Storage Controllers (HSC) for mass storage (i.e., disk and tape). For high-reliability applications, a cluster must contain a minimum of two VAX processors and two mass storage controllers with dual-ported devices. The preferred method of attaching terminals is through a Local Area Transport (LAT) server (not shown), which allows a terminal to connect to any host in a VAXcluster system.

The CI bus is a dual-path serial interconnect with each path supporting a transfer rate of 70-megabits per second. The primary purpose of

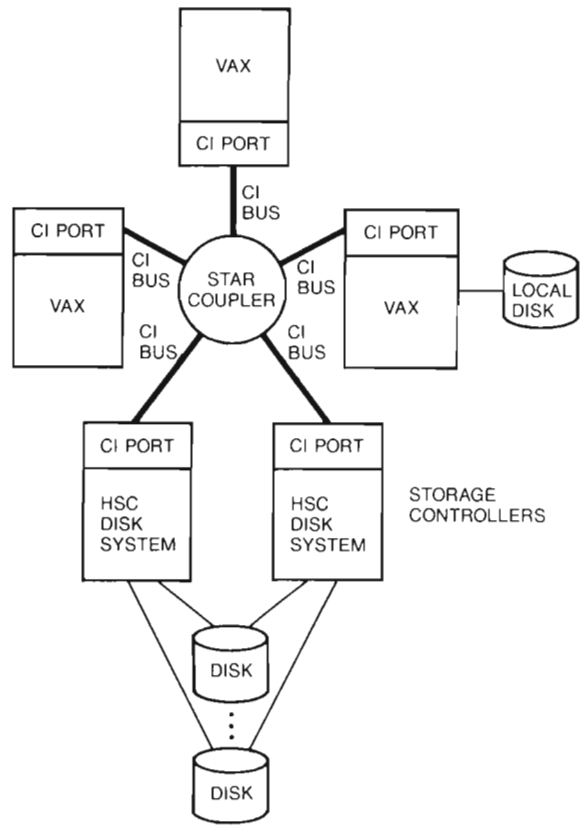


Figure 1 VAXcluster Hardware Topology

the dual paths is to provide redundancy in the case of path failure; when both paths are available, they are usable concurrently. Each path is implemented in two coaxial cables; one for transmitted and one for received signals. Baseband signaling with Manchester encoding is employed.

While the CI bus is logically a bus, it is physically organized as a star topology. A central hub called the Star Coupler connects all of the nodes through radial CI paths of up to 45 meters. The current coupler is a passive device that supports a maximum of 16 nodes; node addresses are 8 bits, providing an architectural limit of 256 nodes.

The selection of a star topology was chosen over a conventional linear topology for several reasons. First, the efficiency of a serial bus is related to the longest transit time between nodes. The star permits nodes to be located within a 45-meter radius (an area of about 6400 square meters) with a maximum node separation of 90 meter radius (an area of about 6400 square meters) with a maximum node separation of

90 meters. Typically, a linear bus threaded through 16 nodes in the same area would greatly exceed 90 meters. Second, the central coupler provides simple, electrically and mechanically safe addition and removal of nodes.

The CI port is responsible for arbitration, path selection, and data transmission. Arbitration uses carrier sense multiple access (CSMA) but is different from the arbitration used by the Ethernet.[14,15] Each CI port has a node-specific delay time. When wishing to transmit, a port waits until the CI bus is quiet and then waits its specific delay time. If the CI bus is still quiet, the node has won its arbitration and may send its packet. This scheme gives priority to nodes with short delay times. To ensure fairness, nodes actually have two delay times — one relatively short and one relatively long. Under heavy loading, nodes alternate between short and long delays. Thus the bus is contention driven under light loading and round robin under heavy loading.

Upon winning an arbitration, a port sends a data packet and waits for receipt of an acknowledgment. If the data packet is correctly received, the receiving port immediately returns an acknowledgment packet *without* re-arbitrating the CI bus. This action is possible because the CI port can generate an acknowledgment in less time than the smallest node-specific delay. Retries are performed if the sending CI port does not receive an acknowledgment.

To distribute transmissions across both paths of the dual-path CI bus, the CI port maintains a path status table indicating which paths to each node are currently good or bad. Assuming that both paths are marked good, the CI port chooses one randomly. This provides statistical load sharing and early detection of failures. Should repeated retries fail on a path, it is marked bad in the status table and the other path is tried.

The Ethernet-based VAXcluster System

Figure 2 shows an example of a Local Area VAXcluster system. The CI bus of Figure 1 has been replaced by an Ethernet, and the VAX hosts (referred to as satellite nodes) are MicroVAX computers and workstations. Satellite nodes may be diskless, in which case one or more VAX hosts act as storage servers, serving a function analogous to the HSC controllers in CI-based configurations. One or more storage servers, called boot nodes, are responsible for loading satellite nodes with the VMS operating system and for storing crash dumps from those nodes. Satellite nodes may use remote disks for process swapping and virtual memory backing storage.

The important difference between the CI-based and the Local Area VAXcluster systems is that the communication functions performed by the CI hardware are emulated in the latter by software within the VMS operating system. The Ethernet is an industry-standard, 10-megabit per second baseband local area network[15] that uses the carrier sense multiple access with collision detection (CSMA/CD) technique for arbitration. Unlike the CI bus, an Ethernet may be used to carry multiple protocols simultaneously. (Note that this allows a cluster to share the Ethernet with other protocols, such as the LAT and DECnet protocols.)

A new Ethernet protocol, which is an extension of SCA, was designed for Local Area VAXcluster system. Using this protocol, a VMS software component emulates the CI port interface, which is to say that the higher level software interface is identical to that of the CI bus, but the Ethernet is used to carry data. This approach eliminated the need for any special hardware and allowed the software modifications needed to be mostly limited to a single VMS component.

Exactly the same approach was used for loading the VMS system into satellite nodes. Here, a special port emulator was developed to operate in the booting and system-initialization environment. This boot driver forms part of a vestigial VMS environment whose function is to read, initialize, and start the VMS system image from the remote disk. These modules are themselves loaded by means of the Digital Network Architecture maintenance operations protocol (MOP).[16]

The CI Port Architecture

Each VAXcluster host and mass storage controller connects either to the CI bus through a CI port or to the Ethernet by means of a standard Ethernet adapter. CI ports have been implemented for the HSC50 and HSC70 mass storage controllers, and the VAX-11/750, 11/780, 11/782, 11/785, and VAX 8000 series hosts. Ethernet adapters have been implemented for all VAX processors. VAX CI ports implement a common architecture, whose goals are to

- Off load much of the communications overhead typically performed by nodes in distributed systems

- Provide a standard, message-oriented software interface for both interprocessor communication and device control

The design of the CI port is based on the needs of the VMS System Communications Architecture. SCA is a software layer that provides efficient communications services to low-level distributed applications (e.g., device drivers, file services, and network managers). SCA supports three communications services: datagrams, messages, and block data transfers. In a Local Area VAXcluster system, the SCA functions performed by the CI port are performed by software in the port emulator module.

SCA datagrams and messages are information units of less than 4,000 bytes sent over a connection. They differ only in reliability. The delivery of datagrams is not guaranteed; they can be lost, duplicated, or delivered out of order. The delivery of messages is guaranteed, as is their order of arrival. Datagrams are used for status and information messages whose loss is not critical, and by applications like the DECnet software that have their own high-level reliability protocols. Messages are used, for example, to carry disk read and write requests.

To simplify buffer allocation, hosts must agree on the maximum size of messages and datagrams that they will transmit. VAXcluster hosts use standard sizes of 576 bytes for datagrams and 112 bytes for messages.

To ensure the delivery of messages without duplication or loss, each CI port maintains a vir-

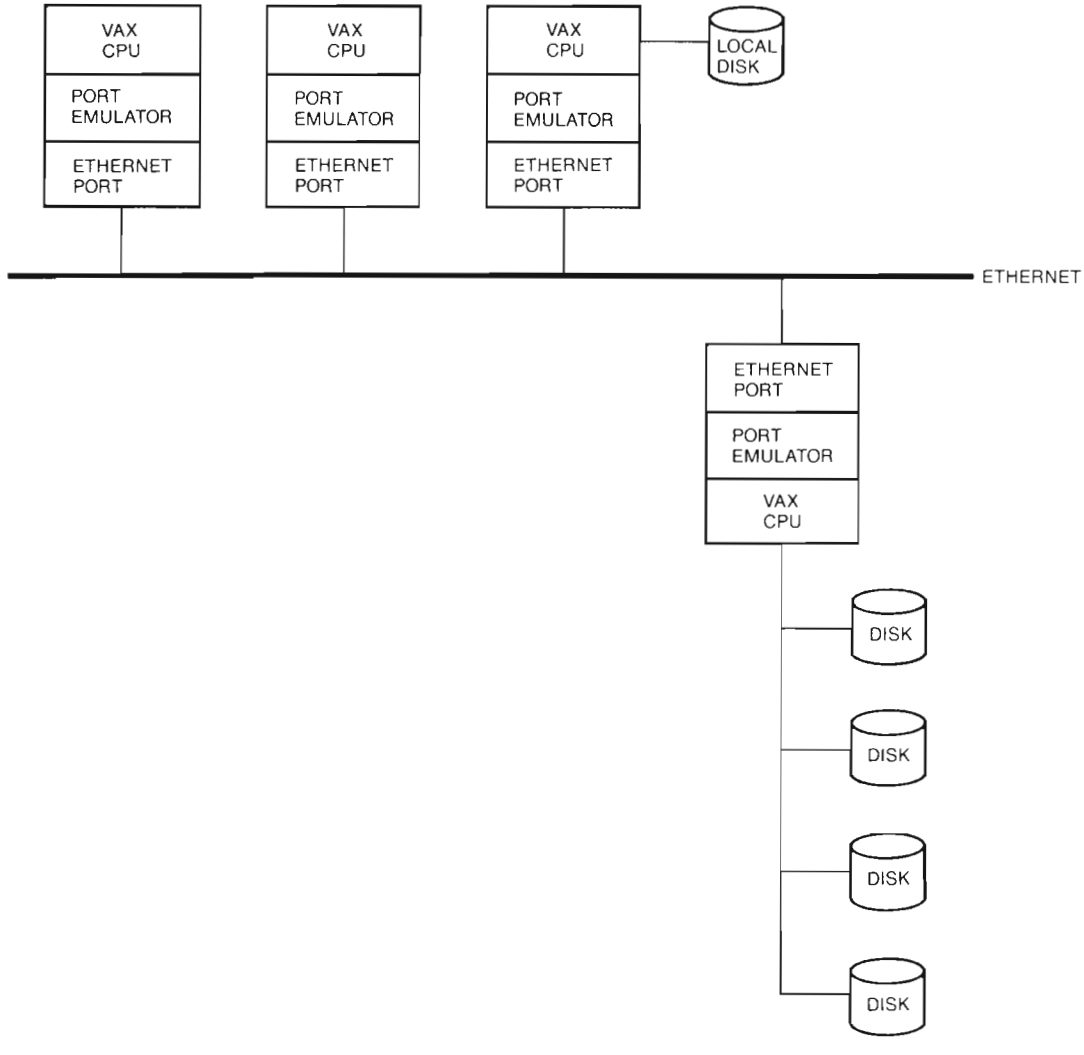


Figure 2 Local Area VAXcluster Topology

tual circuit with every other remote CI port. A virtual circuit descriptor table in each port indicates the status of its port-to-port virtual circuits. Included in each virtual circuit descriptor are sending and receiving sequence numbers. Each transmitted message carries a sequence number enabling duplicate packets to be discarded.

Block data is any contiguous data in a process' virtual address space. There is no size limit except that imposed by the physical memory constraints of the host. The CI port hardware is capable of copying block data directly from the process virtual memory on one node to the process virtual memory on another node. For the Ethernet, this function is performed in software by the port emulator.

The delivery of block data is guaranteed. The sending and receiving ports and the port emulators cooperate in breaking up the transfer into data packets and ensuring that all packets are correctly transmitted, received, and placed in the appropriate destination buffer. Virtual circuit sequence numbers are used on the individual packets, as with messages. Thus the major differences between block data and messages are the size of the transfer, and the fact that block data need not be copied by the host operating system. Block data transfers are used, for example, by disk subsystems and disk servers to move data associated with disk read and write requests.

CI Port Interface

The VAX CI port interface is shown in Figure 3. The interface consists of a set of seven queues: four command queues, a response queue, a datagram free queue, and a message free queue. The queues and queue headers are located in host memory. When the port is initialized, the host software loads a port register with the address of a descriptor for the queue headers.

Host software and the port communicate through queued command and response packets. To issue a port command, the port driver software queues a command packet to one of the four command queues. These four queues accommodate four priority levels; servicing is FIFO within each queue. An opcode within the packet specifies the command to be executed. The response queue is used by the port to enqueue incoming messages and datagrams, while the free queues are a source of empty packets for incoming messages and a sink for transmitted message packets.

For example, to send a datagram, software queues a SEND DATAGRAM packet onto one of

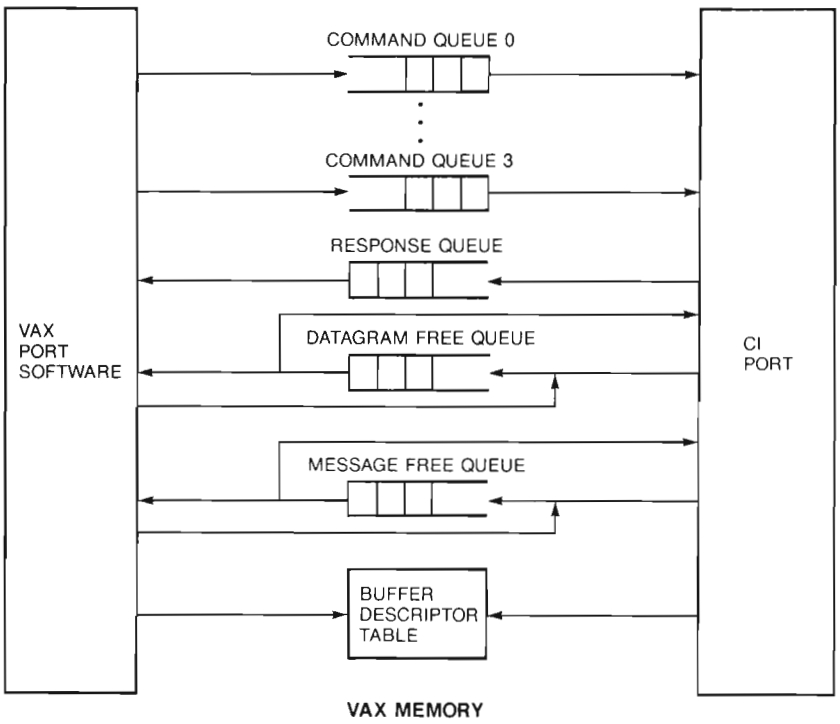


Figure 3 The CI Port Interface

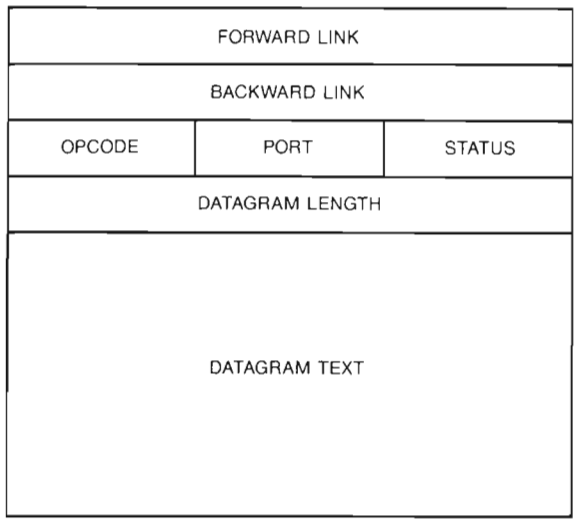


Figure 4 CI Port Command Jacket

the command queues. The packet contains an opcode field specifying SEND DATAGRAM, a port field with the destination port number, the datagram size, and the text of the datagram. The packet is doubly linked through its first two fields. This structure is shown in Figure 4.

If the host software needs confirmation when the packet is sent, it sets a response queue bit in the flags field. This bit causes the port to place the packet in the response queue and interrupts the host after the packet has been transmitted. The response packet is identical to the SEND DATAGRAM packet, except that the status field indicates whether or not the send was successful. Had the response queue flag bit been clear in the SEND DATAGRAM command (as it typically is), the port would instead place the transmitted command packet on the datagram free queue without causing a host interrupt.

Upon receiving a datagram, a CI port takes a packet from its datagram free queue. Should the queue be empty, the datagram is discarded. Otherwise, the port constructs a DATAGRAM RECEIVED packet that contains the datagram and the port number of the sending port. This packet is then queued on the response queue.

Messages operate in a similar fashion, except that they have a different opcode, and the message buffers are dequeued from the message free queue. If the message free queue is empty when a message arrives, the port generates an error interrupt to the host. The high-level SCA flow control ensures that the message free queue does not become empty.

Block transfer operations are somewhat more complicated. Each port has a data structure called a buffer descriptor table. Before performing a block transfer, host software creates a buffer descriptor that defines the virtual memory buffer to be used. The descriptor contains a pointer to the first VAX page table entry mapping the virtually contiguous buffer. In addition, the descriptor contains the offset (within the first page) of the first byte of the buffer, the length of the buffer, and a 16-bit key. The data structures for a block transfer are illustrated in Figure 5.

Each buffer has a 32-bit name, consisting of a 16-bit buffer descriptor table index and the 16-bit buffer key. The key is used to prevent dangling references and is modified whenever a descriptor is released. To transfer block data, the initiating software must have the buffer names of the source and destination buffers. The buffer names are exchanged through a high level message protocol. A host can cause data to be moved either to another node (SEND DATA) or from another node (REQUEST DATA). A SEND DATA or REQUEST DATA command packet contains the names of both buffers and the length of the transfer. In either case (send or request), a single command packet causes the source and destination ports to perform the block transfer. When the last packet has been successfully received, the initiating port places a response packet on its response queue, indicating that the transfer is complete.

The goal of reducing VAX host interrupts is met through several strategies and mechanisms. First, the block transfer mechanism minimizes the number of interrupts necessary to transfer large amounts of data. Second, at the sending port, DATAGRAM SENT/MESSAGE SENT confirmation packets are typically generated only when a failure occurs. Third, a receiving port interrupts the VAX host only when the port queues a received packet on an empty response queue. Thus when software dequeues a packet in response to an interrupt, it always checks for more packets before dismissing the interrupt.

Port Emulation for the Ethernet

Figure 6 shows the relationship of the port emulator to the VMS operating system functions that use that emulator. For comparison, the CI port interface is also shown in this diagram. The port emulator implements the same functions as the

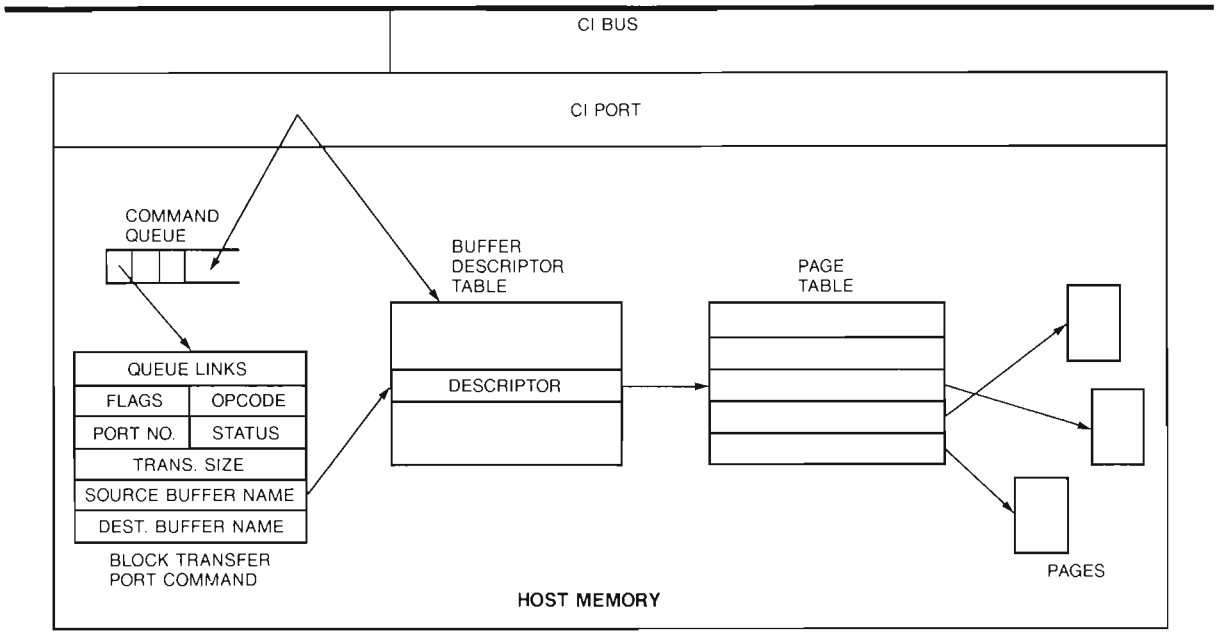


Figure 5 CI Port Block Data Memory Mapping

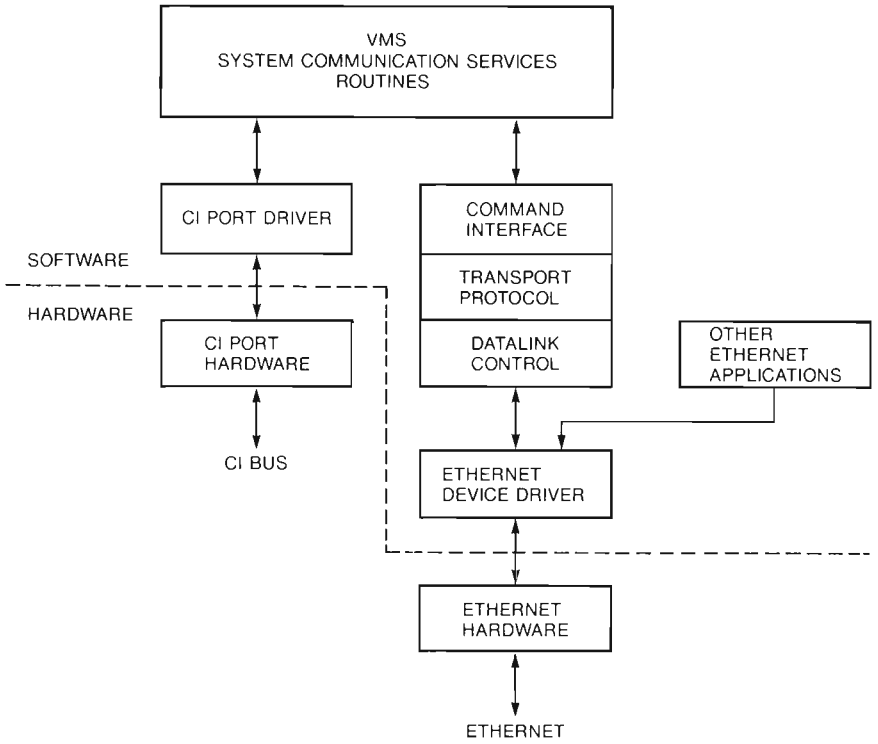


Figure 6 CI Port Emulation Using Ethernet

emulator implements the same functions as the CI port and its associated driver. The emulator also operates the SCA protocol across the Ethernet and manages its interface with the Ethernet datalink driver. Thus the emulator is responsible for

- The provision of a compatible command interface to the system communication services (SCS) module
- The operation of a transport protocol that imitates CI behavior
- Node authentication and topology control functions
- Propagation of Ethernet datagrams and datalink control

The port emulator must deal with an underlying datalink layer whose characteristics are somewhat different than those of the CI bus. The Ethernet datalink can transmit datagrams between 64 and 1,536 bytes in length in either a point-to-point, multicast, or broadcast fashion. The Ethernet provides neither automatic acknowledgment nor flow control, and Ethernet adapters do not handle either buffer segmentation or different message types. The CI functions of datagram transmission, sequenced messages, and block transfers must be implemented by the emulator and translated into requests that can be processed by the standard VMS Ethernet device drivers.

Port emulation can be viewed conceptually as three separate layers. The highest layer provides a command interface for the higher level SCS routines. That interface is compatible with that used for CI ports. This layer is also responsible for the fragmentation and re-assembly of block transfer buffers that are larger than the maximum Ethernet message size.

The transport layer provides a sequenced message and datagram service to the corresponding layer in the remote node. Its handling of datagrams amounts to little more than a pass-through function; the handling of sequenced messages and block transfers, however, is more complex. In the latter case, the transport layer must ensure that messages are transmitted and received in the correct order, ensure that acknowledgments are sent and received, and retransmit messages that have been lost. The transport layer operates a simple pipeline flow control scheme that allows a fixed window of unacknowledged messages. Acknowledgments can be "piggybacked" on returning messages.

Last, the datalink control layer is responsible for passing messages between the Ethernet device drivers and the transport layer and control of the Ethernet datalink service. The datalink control layer also maintains a record of the cluster's topology by exchanging multicast messages with other cluster members.

Below the port emulator module is the standard VMS Ethernet device driver, which can also be used simultaneously by other applications like the DECnet, LAT, and ISO transport protocols. These protocols are multiplexed and demultiplexed by the Ethernet device driver using the Ethernet standard protocol type.

The CI port emulation function for the Local Area VAXcluster system has a higher system overhead than the equivalent CI connection since the operations involved are performed by the host VAX processor. Since the Ethernet has lower bandwidth and longer response times, however, the demand for host system resources is moderated. The Local Area VAXcluster performance is acceptable for typical customer workloads in which most nodes are single-user workstations. The CPU time overheads are most noticeable on nodes that serve disks to multiple users; those nodes are typically dedicated processors.

Mass Storage Control

The move from control- and status register-activated storage devices to message-oriented storage devices offers several advantages:

- Sharing is simplified since several hosts can queue messages to a single controller. In addition, device control messages can be transmitted to and executed by hosts with local disks.
- Extension to new devices is easier. In contrast to conventional systems where there is a different driver for every type of disk and disk interface, a single disk class driver simply builds message packets and transmits them using a communications interface. The disk class driver is independent of drive specifics (e.g., cylinders and sectors). New disk and tape devices and controllers can be added with little or no modification to the host software.
- Performance is improved. The controller can maintain a queue of requests from multiple hosts and can optimize disk performance in real time. The controller can also handle error recovery and bad-block replacement.

The HSC family, shown in Figure 1, is a CI-based controller for both disks and tapes. A single HSC70 controller can handle up to 32 disk drives. Multiple HSC controllers with dual-ported disks provide redundancy in case of failures. Further redundancy can be provided by grouping disk volumes together in shadow sets to form a single virtual volume in which all members contain exactly the same data. If one member of the shadow set fails, the virtual disk volume continues to be available.

The protocol interpreted by the HSC controller is called the Mass Storage Control Protocol (MSCP), which provides access to mass storage volumes at the logical block level. The MSCP model separates the flow of control and status information from the flow of data. This distinction has been used in other systems to achieve efficient file access[17] and corresponds to the CI port's message and block data mechanisms; messages are used for device control commands while block transfers are used for data.

The same control protocol is used to provide clusterwide access to CI-based controllers like the HSC devices, and to disks connected directly to a VAX processor (See Figure 7). In a Local Area VAXcluster system, all mass storage is connected directly to the boot node and to zero or more other storage server nodes. Messages are routed from the disk class driver in the requesting node

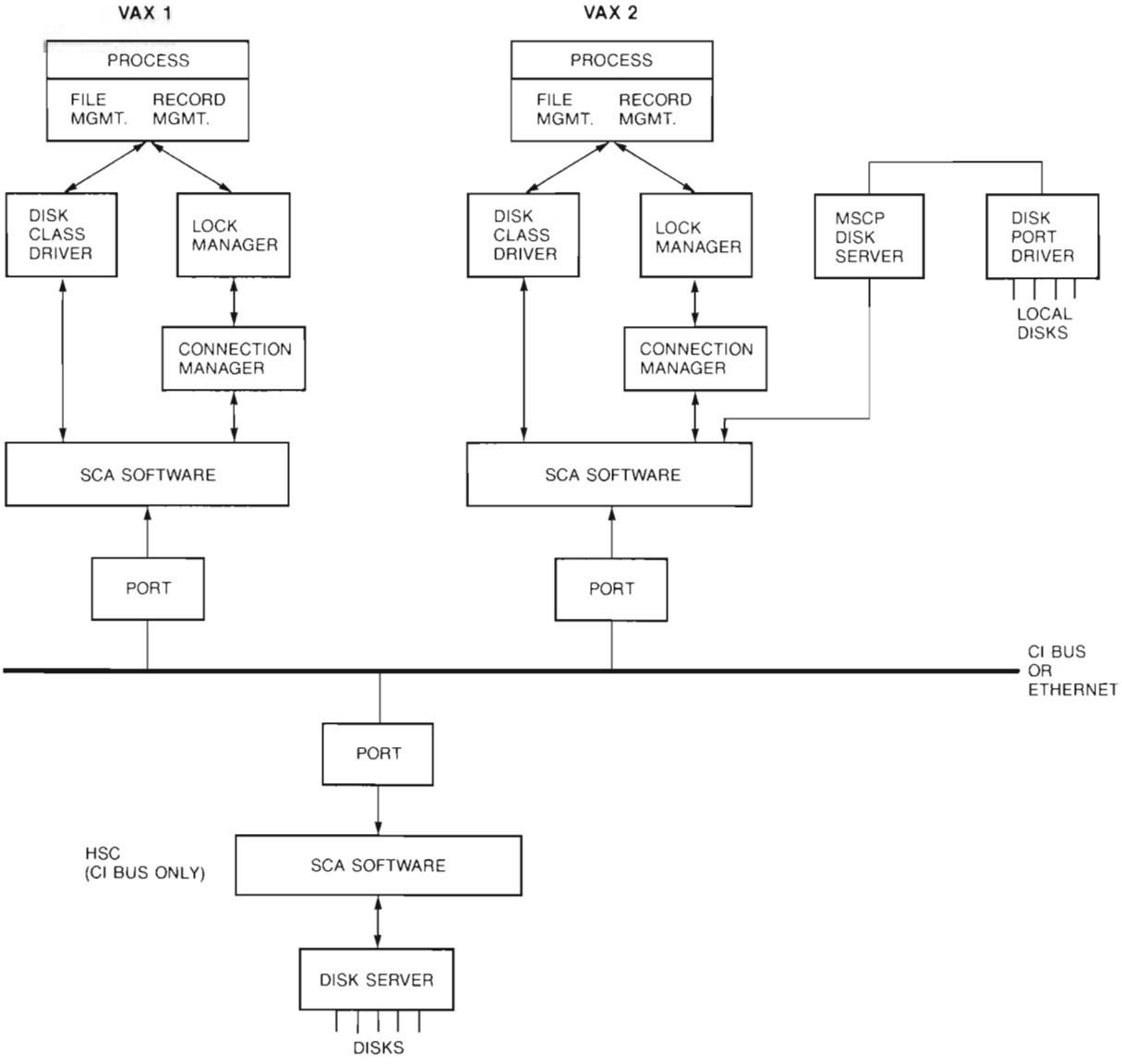


Figure 7 VAXcluster Software Structure

to an MSCP server on the node with the local disk. This server then parses the MSCP message, issues requests to its disk, and initiates the block transfer through its SCA interface. Thus in either a CI-based or a Local Area VAXcluster system, all locally attached disks can be made transparently available to all other VAX hosts in the cluster.

VAXcluster Software

From a user's point of view, a VAXcluster system is a set of nodes cooperating through the VAX/VMS distributed operating system software to provide sharing of resources among users on all nodes. Shared resources include certain devices, files, records within files, and system batch and print queues. Typically, user account and password information resides in a single file shared by all cluster nodes. A user obtains the same environment (files, default directory, privileges, etc.) regardless of the node to which he or she is logged into. In many respects, the VAXcluster system "feels" like a single system to the user.

This sense of a single system results from the fact that the VAXcluster system is symmetrical with respect to the participating VAX processors. In other words, there is no specialization of function designed into the software (although an installation may choose to configure certain CPUs differently according to the special needs of that installation). The VMS and VAXcluster file system architecture is based on the concept of clusterwide and uniform logical block access to the mass storage managed by a distributed file system. This concept contrasts with file server-based distributed systems.

Figure 7 shows an example of a small VAXcluster system and some of its major software components. Note that the operation of the VMS software in the VAXcluster environment is exactly the same for both Local Area and CI-based VAXcluster systems. The diagram shows an underlying interconnect that may be either the CI bus or the Ethernet, both of which use the port interface methods described above. HSC disk controllers connect only to the CI bus.

At the highest level, multiple user processes on each node execute in separate address spaces. File and record management services are implemented as procedure-based code within each process. The file and record services rely on lower level primitives, such as the lock manager[18] and disk class driver. The lock manager is the foundation of all resource sharing in both clustered and single-node VMS systems. It provides services for naming, locking, and unlocking clusterwide resources. The disk class driver, mentioned earlier, uses the MSCP to communicate with disk servers. The disk class driver runs in both clustered and nonclustered environments and contains no knowledge of the VAXcluster configuration. SCA software below the driver is responsible for routing driver messages to the correct device controller.

A distributed connection manager is responsible for coordinating the cluster. Connection managers on all cluster nodes collectively decide upon cluster membership, which varies as nodes leave and join the cluster. Connection managers recognize recoverable failures in remote nodes; they also provide data transfer services that handle such failures transparent to higher software levels.

Forming a Cluster

A VAXcluster system is formed when a sufficient set of VAX nodes and mass storage resources becomes available. New nodes may boot and join the cluster, and members may fail or shut down and leave the cluster. When a node leaves or joins, the process of reforming the cluster is called a cluster transition. Cluster transitions are managed by the connection managers.

In an operating cluster, each connection manager has a list of all member nodes. The list must be agreed upon by all members. A single node can be a member of only one VAXcluster system; in particular, the same resource (such as a disk controller) cannot be shared by two clusters or the integrity of the resources could not be guaranteed. Therefore, connection managers must prevent the partitioning of a cluster into two or more clusters attempting to share the same resources.

To prevent partitioning, the VMS system uses a quorum voting scheme. Each cluster node contributes a number of votes, and the connection managers dynamically compute the total votes of all members. The connection managers also maintain a quorum value. As transitions occur, the cluster continues to run as long as the total number of votes present equals or exceeds the quorum. Should the total number of votes fall below the quorum, the connection managers will suspend VAXcluster activity. When a node joins and brings the total votes up to the quorum, cluster activity will resume.

A cluster member may have a recoverable error in its communications. Such an error leaves the node's memory intact and allows the operating system to continue running after the error condition has disappeared. These errors can cause termination of a virtual circuit and a corresponding loss in communication. When cluster members detect the loss of communication with a node, they wait for a short period (specified by the system manager) for the failing member to re-establish contact. If the failing member recovers within this period, it rejoins the cluster. Users may experience a brief interruption of service when this happens. If the failing member does not recover in time, the surviving members remove the failed node from the cluster and continue operating (assuming sufficient votes are present). A node that recovers after it has been removed from the cluster is told to re-boot by the connection managers.

Shared Files

The VAXcluster system provides a clusterwide shared file system to its users.[19] Cluster accessible files can exist on CI-based disk controllers or on disks local to any of the cluster nodes. Each cluster disk has a unique and location-independent name. A complete cluster file name includes the disk device name, the directory name, and the file name. Using the device name for a file, the cluster software can locate the node (either a CPU or a disk controller) on which the file resides.

Cluster file activity requires synchronization; exclusive-write file opens, coordination of file system data structures, and management of file system caches are a few examples. However, despite the fact that files can be shared clusterwide, the file management services are largely *unaware* of whether they are executing in a clustered environment. These file managers synchronize through the VMS lock manager, described later. The lock manager handles the locking and unlocking of resources across the cluster. At the level of the file manager, then, cluster file sharing is similar to single-node file sharing. Lower levels handle the clusterwide synchronization and routing of physical-level disk requests to the correct device.

Distributed Lock Manager

As previously described, the VMS lock manager is the basis for clusterwide synchronization. Several goals influenced the design of the lock manager for a distributed environment. First, programs using the lock manager must run in both single-node and cluster configurations. Second, lock services must be efficient to support system-level software that makes frequent short-duration accesses. Therefore, in a VAXcluster system, the lock manager must minimize the number of SCA messages needed to manage locks. In a single-node configuration, the lock manager must recognize the simpler environment and bypass any cluster-specific overhead. Finally, the lock manager must recover from failures of nodes holding locks so that surviving nodes can continue to access shared data in a consistent manner.

The VMS lock manager services allow cooperating processes to define shared resources and synchronize access to those resources. A resource can be any object an application cares to define. Each resource has a user-defined name by which it is referenced. The lock manager provides basic synchronization services to request and release locks. Each lock request specifies a locking mode, such as exclusive access, protected read, concurrent read, and concurrent write. If a process requests a lock that is incompatible with existing locks, the request is queued until the resource becomes available. In many applications, resources may be subdivided into a resource tree, as illustrated in Figure 8.

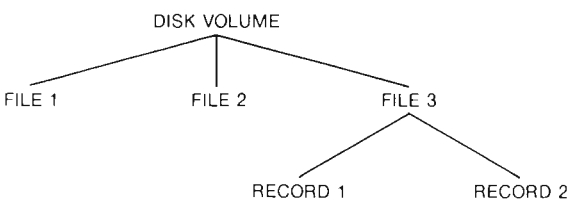


Figure 8 VAXcluster Locking Structure

In this example, the resource Disk Volume contains resources File 1 through File 3; resource File 3 contains resources Record 1, Record 2, and so on. The first locking request for a resource can specify the parent of that resource, thereby defining its relationship in a tree. A process making several global changes can hold a high-level lock (e.g., the root) and can make them all very efficiently. A process making a small, low-level change (e.g., a leaf) can do so while still permitting concurrent access to other parts of the tree.[20]

The lock manager's implementation is intended to distribute the overhead of lock management throughout the cluster while still minimizing the internode traffic needed to perform lock services. The database is therefore divided into two parts: the resource lock descriptions, and the resource lock directory system, both of which are distributed. Each resource has a master node responsible for granting locks on the resource; the master maintains a list of granted locks and a queue of waiting requests for that resource. The master for all operations for a single tree is the node on which the lock request for the root was made. While the master maintains the lock data for its resource tree, any node holding a lock on a resource mastered by another node keeps its own copy of the resource and lock descriptions.

The second part of the database, the resource directory system, maps a resource name into the name of the master node for that resource. The directory database is distributed among nodes willing to share this overhead. Given a resource name, a node can trivially compute the responsible directory as a function of the name string and the number of directory nodes.

To lock a resource in a VAXcluster system, the lock manager sends a lock request message through the SCA to the directory for the resource. The directory responds in one of three ways:

1. If located on the master node for the resource, the directory performs the lock request and sends a confirmation response to the requesting system.
2. If the directory is not on the master node but finds the resource defined, it returns a response containing the identity of the master node.
3. If the directory finds the resource to be undefined, it returns a response telling the requesting node to master the resource itself.

In the best cases (1 and 3), two messages are required to request a lock; case 2 takes four messages. An unlock is executed with one message. If the lock request is for a subresource in a resource tree, the requesting process will either be located on the master node (i.e., the request is local) or will know who the master for its parent is, allowing it to bypass the directory lookup. In all cases the number of messages required is independent of the number of nodes in the VAXcluster system.

In addition to standard locking services, the lock manager supports data caching in a distributed environment. Depending on the frequency of modifications, caching of shared data in a distributed system can substantially reduce the I/O and communications workload.

A 16-byte block of information, called a value block, can be associated with a resource when the resource is defined to the lock manager. The value in the value block can be modified by a process releasing a lock on the resource and can be read by a process when it acquires ownership. Thus this information can be passed along with the resource ownership.

In the case of a file buffer, for example, a version number is maintained in the value block. When caching a buffer, a process saves the current version number. To modify the buffer, the process obtains an exclusive lock and receives the current version number. If the current version number equals the version number of the cached data, the cache is valid. Several updates can then be made on the cached data before it is written back to disk. When the modified data is written, the process increments the version number and releases its lock.

Another mechanism used in buffer caching is a software interrupt mechanism. When requesting an exclusive lock, a process can specify that it should be notified if another lock request on the resource is forced to block. A process can then hold a modified copy of the data without writing it back. When another process wants access, the owner writes the modified data and releases its lock.

In the case of cluster transitions (e.g., failure of a node), the connection manager notifies the lock manager that a transition has started. Each lock manager performs recovery action, and all lock managers must complete this activity before cluster operation can continue.

As the first step in handling transitions, a lock manager deallocates all locks acquired on behalf of other systems. Only local lock and resource information is retained. Temporarily, there are no resource masters or directory nodes. In the second step, each lock manager re-acquires each lock it had when the cluster transition began. This step establishes new directory nodes based on a new set of eligible cluster members and rearranges the assignment of master nodes. If a node

has left the cluster, the net result is to release locks held by that node. If no node has left the cluster but nodes have joined, this recovery is not necessary from an integrity point of view. It is performed, however, to keep the directory and lock mastering overhead evenly distributed.

Some resources, depending on how they are modified, might be left in an inconsistent state by a cluster transition. To ensure the proper handling of such resources, users can define a class of locks that are not released on a cluster transition. In this case a special process can search for such locks and perform needed consistency checks before releasing them.

Batch and Print Services

In a VAXcluster system, users may either submit a batch job to a queue on a particular node (not necessarily their own node), or submit a job to a clusterwide batch queue. Jobs on the clusterwide queue are routed to queues attached to specific nodes for execution. The algorithm for assigning jobs to specific nodes is a simple one based on the ratio of executing jobs compared to the job limit of the queue.

The management of batch jobs is the responsibility of a VMS process called the job controller. Each VMS node runs a job controller process, which acquires work from one or more batch queues. Batch queues are stored in a disk file that may be shared by all nodes. The synchronization of queue manipulation is handled with lock manager services.

Print queues are similar to batch queues. Users may queue a request for a specific printer (not necessarily physically attached to their own node) or may let the operating system choose an available printer from those in the cluster.

Both batch and print jobs can be declared restartable. If a node fails, restartable jobs are either requeued to complete on another node in the cluster or executed when the failed node reboots (for jobs that must execute on a specific node).

DECnet Communications

Each member of a VAXcluster system can also participate in a DECnet network as an individual node. Simultaneously, the cluster as a whole may participate in the network as a single node. The cluster's system manager may select an additional DECnet node name and address, known as the cluster's alias, to be assigned to the cluster. DECnet connections originating from a cluster member can be made to appear as if they came from the alias node, regardless of the true originator. Connections addressed to the alias will be directed to any cluster member that has declared itself willing to receive them. This concept is particularly useful for sending and receiving network mail. All mail sent from the cluster will appear to have come from a single node. All replies will be delivered to the cluster's mail files even when the node from which the first message was sent is unavailable (provided that the disk remains available).

The VAXcluster DECnet alias address requires the presence of at least one routing node in the cluster. DECnet routing nodes maintain tables describing the topology of the network and communicate this information to other nodes. The existence of the cluster's alias address is thus propagated in control messages to other nodes in the network. Although the alias node does not actually exist, a path to it via the cluster's router is apparent. The router maintains a table of connections to the alias node by means of the distributed lock manager. When a connect request for the alias arrives at the router, it passes the request to another node in the cluster, distributing the connections in a round-robin fashion. Connect requests originating from the cluster members are simply set up as if they came from the alias.

Terminal Support

The optimum method for connecting users' terminals to a VAXcluster system is through the LAT server. Terminals are connected to the LAT server, which is attached to the VAX systems by the Ethernet. In a Local Area VAXcluster system, this connection can be the same Ethernet used to interconnect the members of the cluster. Users command the LAT server to connect them either to a specific node or to any node in the cluster. The ease of switching nodes leads users to find and use the least busy node. The server also allows users to quickly move from a failed node to one that is still running. If the LAT server is directed to select a node, it attempts to find the least busy one. Its choice is based on node CPU type (a measure of processing power) and recent idle time.

Performance

Performance measurements using a CI-based VAXcluster system of two VAX-11/780 systems have shown it is possible to achieve 3,000 message round-trips per second.[21] A round-trip is defined as the transmission of a message and the receipt of its acknowledgment from the remote system. This performance provides a basis for efficient execution of higher level distributed services, such as the VMS distributed lock manager and the MSCP logical block service used for access to mass storage. The performance characteristics of CI-based VAXcluster systems vary almost linearly in relation to the number of CI nodes in the system. From this it can be concluded that the underlying communications architecture upon which the VAXcluster system is based scales well with an increasing number of nodes. Measurements with up to twelve VAX-11/780 nodes showed nearly linear performance in cluster round trips per second.

The performance characteristics of a Local Area VAXcluster system are somewhat different for the following reasons:

- The interconnect speed is limited to 10 megabits per second, as opposed to 70 megabits per second for the CI bus.
- The delay (i.e., latency) for message round trips in the Ethernet network is somewhat greater.

Because VMS VAXcluster systems attached to the Ethernet are optimized as single-user workstations, the limits of throughput and latency do not present a problem. Workload studies have shown that the limiting factor in Local Area VAXcluster performance is the rate at which the boot node can service the satellites' mass storage I/O requests. These studies further indicate that this limit in turn depends upon the CPU speed of the boot node while executing both the CI port emulation code and the MSCP server code. For a fast VAX system (e.g., a VAX 8700 CPU), the next limit is imposed by the throughput of the Ethernet adapter used by the boot node. The final limit to be encountered is the saturation of the Ethernet network itself. This limit is reached at approximately 100 typical VMS I/O requests per second and is largely independent of the number of satellite and boot nodes accommodated by the network. Note that the factors limiting the number and size of Local Area VAXcluster systems that can be sustained by a single Ethernet segment is heavily dependent upon the nature of the applications being run.

Summary

A principal goal of VAXcluster systems was the development of an available and extensible multicomputer system built from standard processors and a general-purpose operating system. Much was gained by the joint design of distributed software, communications protocols, and hardware aimed to meet this goal. For example:

- The CI interconnect supports the fast message transfer needed by the system software.
- The CI port implements many of the functions needed by the SCA software.
- The HSC controllers, with their message-protocol and request-queuing optimization logic, support a large pool of disks for multiple hosts.

Designing hardware and software together allows for system-level trade-offs; the software interface and protocols can be tuned to the hardware devices.

An important simplifying aspect of the VAXcluster design is the use of a distributed lock manager for resource synchronization. In this way, higher level services such as the file system do not require special code to handle sharing in a distributed environment. However, the performance of the lock manager becomes a crucial factor. The performance of the distributed lock manager has been attacked with the design of a locking protocol requiring a fixed number of messages, independent of the number of cooperating nodes.

The system design of the original VAXcluster implementation also allowed its straightforward migration to the Ethernet without the need for extensive hardware and software modification. The Local Area VAXcluster product allows workstation users to enjoy the benefits of a large, centrally managed timesharing system on their individual office system without having to deal with the various system management tasks.

Finally, we believe that performance measurements show the extent to which the VAXcluster system has succeeded in implementing an efficient communications architecture that is applicable to both a high-speed dedicated LAN (the CI bus) and a general-purpose shared LAN (the

Ethernet). This feat is particularly impressive when considering that the VMS software is a large, general-purpose operating system.

Acknowledgments

VAXcluster systems are the result of work done by many individuals in several engineering groups at Digital Equipment Corporation. We would particularly like to acknowledge the contributions of Richard I. Hustvedt to the VAXcluster design.

References

1. G. Almes et al., "The EDEN System: A Technical Review," *IEEE Transactions on Software Engineering SE-11* (January 1985): 43–59.

2. *Apollo Domain Architecture* (North Billerica: Apollo Computer Corporation, 1981).

3. A. Brownbirdge, A. Marshall, and A. Randell, "The Newcastle Connection or UNIXES of the World Unite!," *Software — Practical Experiments 12* (1982): 1147–1162.

4. G. Popek et al., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating Systems Principles, ACM* (1981): 169–177.

5. G. Fielland and D. Rodgers, "32-bit Computer System Shares Load Equally Among Up to 12 Processors," *Electrical Design* (September 1984): 153–168.

6. K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, (New York: McGraw-Hill, 1984).

7. M. Satyanarayanan, *Multiprocessors: A Comparative Study*, (Englewood Cliffs: Prentice-Hall, 1980).

8. W. Strecker, "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proceedings of AFIPS NCC* (1978): 967–980.

9. J. Bartlett, "A Nonstop Kernel," *Proceedings of the 8th Symposium on Operating Systems Principles, ACM* (1981): 22–29.

10. A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proceedings of the 9th Symposium on Operating Systems Principles, ACM* (1983): 90–99.

11. D. Katsuki et al., "PLURIBUS — An Operational Fault-tolerant Multiprocessor," *Proceedings of the IEEE 66* (October 1978): 1146–1159.

12. J. Katzman, "The Tandem 16: A Fault-tolerant Computing System," *Computer Structures: Principles and Examples*, ed. D. Siewiorek (New York: McGraw-Hill, 1982).

13. M. Fox and J. Ywoskus, "Local Area VAXcluster Systems," *Digital Technical Journal* (September 1987, this issue): 56–68.

14. R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM 19* (July 1976): 395–404.

15. *The Ethernet: A Local Area Network, Data Link Layer and Physical Layer Specification, Version 2.0* (Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, Order No. AA-K759B-TK, 1982).

16. *DECnet Digital Network Architecture (Phase IV) Maintenance Operations Functional Specification* (Bedford: Digital Equipment Corporation, Order No. AA-X436A-TK, 1983).

17. D. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations," *Proceedings of the 9th Symposium on Operating Systems Principles, ACM* (1983): 129–140.

18. W. Snaman, Jr. and D. Thiel "The VAX/VMS Distributed Lock Manager," *Digital Technical Journal* (September 1987, this issue): 29–44.

19. A. Goldstein, "The Design and Implementation of a Distributed File System," *Digital Technical Journal* (September 1987, this issue): 45–55.

20. J. Gray et al., "Granularity of Locks and Degrees of Consistency in a Shared Data Base," *Modelling in Data Base Management Systems*, ed. G. Nijssen (Amsterdam: North Holland, 1976).

21. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems*, vol. 4, no. 2 (May 1986): 130–146.

Darrell J. Duffy

The System Communication Architecture

The System Communication Architecture defines how data traffic is handled among host systems and their disk systems over the CI interconnect in a VAXcluster configuration. Low CPU overhead was a key design goal. The SCA supports the management of cluster configurations, buffers, and connections. It also supports directory services, datagram and sequenced-message services, and named-buffer transfer services. The SCA can be extended to connections between hosts and locally attached storage controllers, and to Local Area VAXcluster systems, which use the Ethernet. Each CI port is capable of sustaining about two megabytes per second of bandwidth with minimal overhead required from a CPU.

The System Communication Architecture (SCA) defines the network architecture for VAXcluster systems, much like the Digital Network Architecture (DNA) defines the network protocols for Digital's wide area networks.[1]

In 1981, as the Computer Interconnect (CI) hardware was being developed, it became clear that some type of network architecture was needed to bind the CI subsystems together. This architecture required a relatively simple structure so that little overhead would be needed in either the VAX host computers or the Hierarchical Storage Controllers (HSC). Many of the system processes within the systems and controllers would have to communicate in, at that time, unforeseen ways. Therefore, the SCA architecture had to support all the features and performance of the CI hardware so they could be used by the system processes.

The CI Interconnect

The CI interconnect provides the following basic services:[2]

- Sending datagrams, which are not guaranteed against loss and duplication
- Sending sequenced messages, which are guaranteed against loss and duplication (If an error occurs, the sending node on the CI interconnect will be notified.)
- Named-buffer transfers, which are potentially large data transfers between process buffers in virtual memory (These transfers are also guaranteed against loss and duplication.)

These services are very useful to the operating system software when VAXcluster and other distributed systems are built. However, in the form that the CI port provided those services, they could not be shared conveniently by the many parts of the operating system needing them.

The SCA architecture provides a simple and efficient means for the various parts of the operating system and the disk-controller software to use these services.

SCA Goals

SCA was developed from the beginning with the following set of goals:

- To provide a high-performance means of accessing and directing mass-storage controllers, and of transferring data
- To facilitate access to and sharing of all the capabilities of the CI ports among many processes within the operating systems of the host computers
- To provide a way for each system on the CI interconnect (e.g., VAX host systems, disk and tape controllers) to obtain configuration infor-

mation about every other system and which functions each system performs

- To establish a means of binding together system applications (SYSAPs) in two different systems over the CI interconnect so that the SYSAPs can communicate using their names

SYSAPs are functions within the operating systems of hosts and within the firmware of disk and tape controllers. In host systems, those functions include disk and tape class drivers, DECnet software, and the VAXcluster connection manager, among others.[3]

In single computer systems, command status registers are used to direct the mass-storage controllers and other devices. In VAXcluster systems, however, the SCA network architecture would now direct the traffic between host systems and disk systems. One important design goal of SCA was to make it operate as efficiently as possible, that is, with low overhead on the systems.

SCA Services

The SCA architecture supports the performance of six different functions.

1. Cluster configuration management
2. Buffer management
3. Connection management
4. Directory services
5. Datagram and sequenced-message services
6. Named-buffer transfer services

The following sections describe each of these functions and show how they interoperate to provide a coherent scheme for system communication.

Cluster Configuration Management

A node on the CI interconnect is either a VAX computer system or an HSC controller supporting disc or tape devices. Within the cluster, a node cannot communicate with another node until it has established that node's location on the CI interconnect. At present, 16 nodes is the maximum number the CI interconnect can support, although the architecture can support 224. Since this current number is small, polling is an efficient method for each node to determine which of the potential nodes are present. There is an "instance" of the SCA software within each of the hardware components connected to the CI interconnect. Using the ID request/response feature of the CI ports, SCA software periodically polls each of the other nodes on the CI and keeps a list of the active members in the hardware cluster. Using the information in this list, the SCA software keeps a port-to-port virtual circuit open to every other node on the interconnect.

SCA software opens this port-to-port virtual circuit by using a series of messages, called a handshake, between itself and another SCA software instance in a partner node. The handshake allows the two SCA instances to first synchronize and then exchange information. At the end of the handshake each node will direct its local CI port to enable the virtual circuit state with the other node's CI port. This enabling allows the guaranteed exchange of sequenced messages and named-buffer transfers between the two ports.

The information exchanged in the handshake gives to each node the software type and SCA version running on the other node. That allows nodes with different SCA versions to interoperate. Other information, such as the time of day and the time the node last booted, is also exchanged.

A node with multiple CI ports will use all its ports to form port-to-port virtual circuits to all the other remote nodes. Each node will store information about each of the remote nodes in a system block for that remote node. Each port-to-port virtual circuit is called a path. The information blocks representing these paths, called path blocks, are chained together to the system block for a particular remote node. In that way, SCA can maintain the exact relationships among the paths and nodes.

The total number of paths between two nodes is equal to the number of CI ports on the local node times the number of CI ports on the remote node. SYSAPs in both the local and remote nodes can determine the topology of the CI interconnect by making special calls to SCA software. Figure 1 depicts an example of the relationship between system blocks and path blocks for a network.

Buffer Management

One of SCA's most important properties is its close control over how the communications buffers are used within the nodes. This control is important because node activity normally occurs at very high data rates. The buffers could be

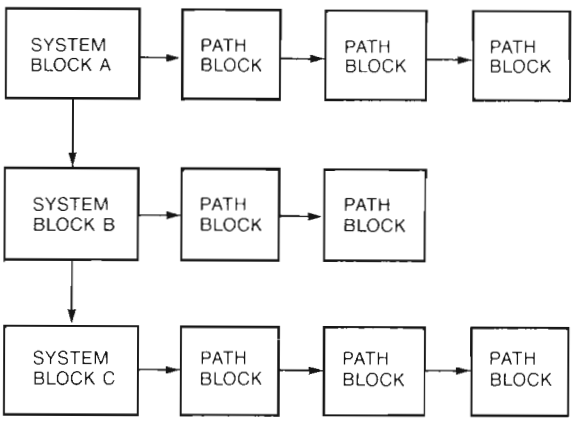


Figure 1 Connections between System and Path Blocks

quickly overrun if data transmission were not strictly controlled from the source. Recovery from buffer exhaustion is not a rapid process. During periods of high load within the node, these delays yield further delays and thus increase the requirements for buffering.

SCA software controls the buffers for two types of traffic: SCA control messages, and SYSAP data messages. SCA control messages are used to establish and remove SYSAP-to-SYSAP connections and to control buffer usage on those connections. The SCA control-message protocol is structured so as to simplify the control of buffer usage.

Control messages come in pairs, a command and its response. A response is expected for each command sent, and a buffer must be available to receive it. The SCA architecture specifies that a response will be received for each command sent. Therefore, a command buffer is made available on the free queue of the CI port to receive the response. Thus each SCA path has two buffers available for control messages, one for sending a command and receiving its response, the other for receiving a command and sending its response.

Allocating buffers for SYSAP dialogues is not as simple as the command/response allocation. In this case, the buffer allocation must be based on the needs of the protocol used by the SYSAPs. Some protocols are command/response in nature, such as the Mass Storage Control Protocol (MSCP) used for the HSC and other storage controllers. Others are not, such as the VMS connection-manager protocol used for VAXcluster systems.

SCA architecture enables the SYSAPs on a node to allocate as many receive buffers as are needed for each connection. Each SYSAP provides these buffers to SCA, which then keeps track of them. Each receive buffer acts as a "credit" to allow the or her node to send one message over that connection. The node's SCA software informs the remote SCA software of the number of credits available for each connection. If a credit is not available, the remote SYSAP will suspend sending its message. This style of buffer management is called "pessimistic flow control." It is normally unsuitable for use in general networks involving routing messages between nodes. However, since routing is not done in the SCA environment, this style has the advantage of being completely predictable. If a node momentarily lags in satisfying communication requests made upon it, the other nodes simply wait until the lagging node recovers. Thus no additional buffer management is required.

The cost of these tight controls on buffer management is some additional overhead to communicate the credits to the sending node. These credits are "piggybacked" onto messages going to the correct node by including a credit field in all SCA messages. When the SYSAP protocol does not contain returning traffic, however, additional control messages are required.

The command/response nature of SCA control messages and the pessimistic flow control for SYSAP messages remove much of the time-related behavior from the SCA architecture. That means the SCA operation is relatively independent of the exact timing of the arrival of messages and the speed of response of the nodes involved in the communication. These factors make it relatively easy to implement and verify the SCA software.

Connection Management

A connection between two SYSAPs in different nodes is a correspondence between two connection identifiers, one from each SCA instance. These connection identifiers allow the SCA software to multiplex its services onto the underlying virtual circuit by dispatching the messages to the correct connection based on the connection identifiers. Each SCA message has a header containing these connection identifiers. Figure 2 shows the layout of an SCA message with the format of the protocol header.

When a node receives a message, SCA will dispatch it based on the message type. For SYSAP-

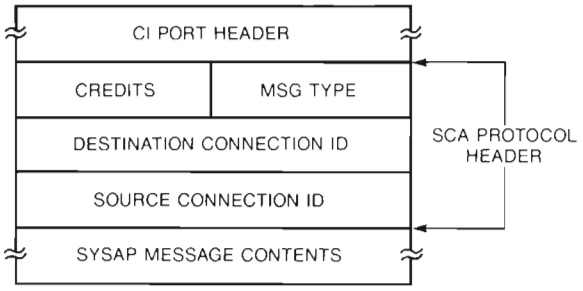


Figure 2 SCA Message with Protocol Header

related messages, SCA uses the ID of the destination connection to dispatch further to the correct SYSAP. As mentioned earlier, the credit field in each message header allows credits to be piggybacked in message traffic.

A SYSAP signals its willingness to receive connections from other SYSAPs by initiating a "listen" call to its own SCA so ftware instance. This call establishes the name of that SYSAP in a list of names of waiting processes. SYSAP names are defined by the architecture as strings of up to 16 characters. Some of the currently defined names are MSCP$DISK and MSCP$TAPE for the disk and tape servers, VMS$VAXcluster for the VAXcluster connection manager, and SCS$DIRECTORY for the SCA directory server.

A SYSAP from another node, the source node, can establish a connection to a listening SYSAP in a destination node by issuing a connect call to SCA, giving the node address of the destination node and the name of the listening SYSAP. Two SCA control-message pairs are required to establish a connection. The first command/response pair from the source establishes the connection at the destination end; the second pair from the destination to the source either accepts or rejects the connection. This separation into pairs allows the destination SYSAP to decide, based on the information passed with the connect request from the source and on its current resources, whether or not to accept the connection.

Figure 3 illustrates the events required to establish a connection between two SYSAPs. The sequence of messages is as follows:

1. A connect-request message is sent from the source node to the destination node. This message contains the source and destination SYSAP names and 16 bytes of additional information from the source SYSAP.
2. A connect-response message is sent from the destination node to the source node. This message indicates that a SYSAP with the requested name exists and that enough resources are present for SCA to honor a connection. If there are not enough resources, then the connection is refused.
3. Later, the destination SYSAP performs either an accept or a reject call, and its SCA software responds by sending either an accept-request message or a reject-request message to the source node.
4. If the message was accept request, the source will respond with an accept-response message and notify its SYSAP that the connection is open. If the message was a reject request, the source SCA software will respond with its own reject response, and the connection will not be opened.

The accept and reject responses by the receiving SYSAP are separated from the connect-request and connect-response message pair. That separation allows the SYSAP to initiate a potentially

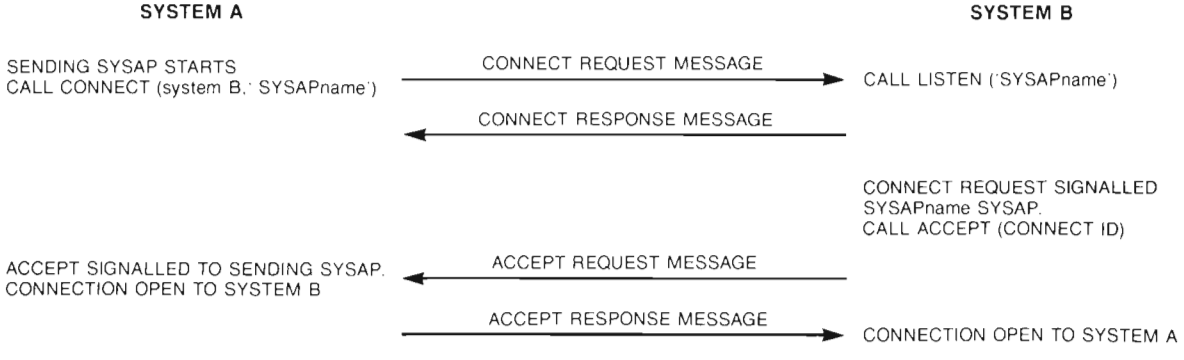


Figure 3 Events to Open a Connection

time-consuming operation without tying up the SCA control-message buffer of the sending SCA instance.

When either member of a pair of SYSAPs holding an open connection wishes to break that connection, that member performs a disconnect call to its SCA software. The SCA software will inform the SYSAP in the other node, which must then perform its own disconnect call to synchronize the dismantling of the connection. Each side informs the other of the disconnect call by exchanging a disconnect-request and disconnect-response message pair.

Directory Services

To accomplish their tasks, the various SYSAPs running within a node need the help of SYSAPs in other nodes. These SYSAPs operate either in a peer-to-peer relationship, such as the VAXcluster connection manager,[3] or in a client-to-server relationship, such as the disk class driver and the MSCP disk server. The method by which SYSAPs find those other SYSAPs within the context of SCA is called the directory service. This service is itself implemented as a SYSAP that listens for incoming connections. The service responds to a simple protocol of requests for information about which SYSAPs on this node are listening for connections from other nodes.

To query the directory service, a SYSAP must request an SCA connection to another node with a destination process name of SCS$DIRECTORY. This special process name is reserved for use by the directory services. The requesting SYSAP can then inquire if a SYSAP with a particular name is listening for a connection and also ask for a list of all SYSAPs currently listening for connections. Figure 4 shows two VAX systems and an HSC device in a cluster, with the SYSAP processes listening in each node.

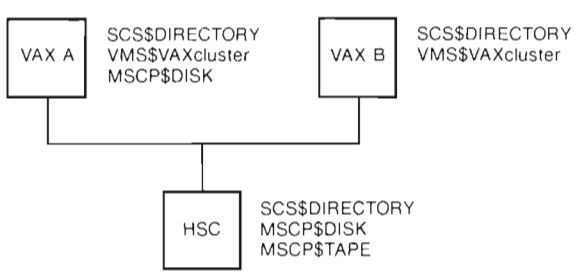


Figure 4 SYSAP Processes among Three Nodes

Every implementation of a SYSAP has the problem of finding partner SYSAPs of the same name to communicate with in the cluster. To centralize the software performing this function, the VAX/VMS software implements a general facility for SYSAPs to find other SYSAPs. This facility periodically polls other nodes through the directory service to determine which listening SYSAPs are present. This process poller is a powerful tool that simplifies the design of the SYSAPs and the operating system software by allowing various SYSAPs to start in one node without depending on whether or not other nodes are working yet. When new nodes — and the SYSAPs within those nodes — are added to the cluster, all the SYSAPs currently running will find each other and communicate automatically.

Datagram and Sequenced-message Services

The CI port and the CI interconnect provide the capability to exchange datagrams and sequenced messages between ports. Datagram and sequenced-message services are both provided by SCA in the context of a connection. A SYSAP establishes a connection with another SYSAP and then sends datagrams or messages over that connection. In the context of SCA, datagrams and messages, by convention, differ in size as well as in their delivery mechanisms. Datagrams are 576 bytes in length so that they are suitable for use by the DECnet protocol as datalink buffers. Messages are 112 bytes in length to accommodate MSCP control messages and VAX/VMS lock manager messages.

Controlling the flow of credits for datagrams and messages is done separately by SCA. Datagram credit controls operate at the receiver. The receiving of datagrams is not guaranteed. Upon receiving a datagram, a SYSAP must have available a datagram-receive credit; otherwise, the datagram is discarded. The receiving of messages, however, is guaranteed. Message-credit controls are instituted at the sending node. When a SYSAP wants to send a message, the receiving node must have a credit available. If not, the sending SYSAP waits and does not send the message until informed that the credit is available.

As mentioned earlier, the port-to-port virtual circuit provided by the CI port hardware controls the loss of sequenced messages between nodes. The circuit retransmits these messages as necessary to guarantee their delivery. In fact, the hard-

ware performs this task for CI datagrams as well, but higher layers of software do not take advantage of this fact.

Datagrams are used to log events and other communications, such as from DECnet nodes, that control the loss of datagrams in other ways. It is useful in these applications to discard information when buffering becomes a problem so that too many buffers are not consumed. In the case of event logging, the lost messages are likely to be duplicates anyway. In the case of the DECnet software, higher layers of DECnet protocol control the loss, and discarding the datagrams prevents congestion at intermediate nodes.

SCA and the CI port work together to make message transfer more efficient by eliminating transmit-done interrupts. When a node expects a response to a message, SCA and the CI port cooperate to queue the buffer sending the message to the free queue. That buffer can then be used to receive the response. Thus in a command/response exchange of two messages, the sending and receiving nodes each experience only one receive interrupt.

Named-buffer Transfer Services

One striking feature of the CI port hardware is its ability to transfer large amounts of data between named buffers in the virtual address space of processes within a node.[2] This feature is the most useful one for disk and tape transfers.

SCA provides services for the two named-buffer transfer commands available in the CI port: the send-data command, and the request-data command. The send-data command transmits the contents of a segment of a local named buffer into a segment of a named buffer in a remote node. The parameters for the send-data command are the transfer length in bytes, and the names and byte offsets of the sending and receiving buffers. The request-data command asks the remote port to transmit data from a remote named buffer to a local named buffer. The send-data command performed by a disk controller corresponds to a disk read function, and the request-data command to a disk write function.

Of course, named-buffer transfers can be used by any SYSAP, not just the ones communicating with disk controllers. Using named-buffer transfers, it is possible for two VAX systems in a cluster to exchange memory data at a transfer rate of over 2 megabytes per second at the CI ports.

Extensions to Other Interconnects

To this point, only the CI implementation of SCA has been discussed. However, the utility of SCA is not limited solely to the CI interconnect. SCA is a general network communication architecture that can serve a number of interconnects. For example, it is currently used in locally connected storage controllers and on Ethernet for low-end VAXcluster systems.

Locally Connected Storage Controllers

The UDA50 UNIBUS and KDB50 BI disk controllers are locally connected storage controllers that connect Digital Storage Architecture (DSA) disk drives to VAX computers without an intervening CI interconnect. These devices are intelligent controllers that incorporate the SCA and MSCP protocols, just as does the HSC50 CI-based disk controller. The use of SCA in these controllers has proven to be an efficient means to communicate with disk controllers in which a direct bus interface has traditionally been used.

In controllers, there is no interconnect between the host adapter and the disk controller; both functions are performed by the same controller. Although the port header has been simplified because it does not have to address multiple ports on an interconnect, the basic SCA functions still operate. The use of SCA allows multiple functions to be placed in a controller and used separately by having them appear as SYSAPs with different names. For example, disk and tape controller functions can both co-reside in a controller but are accessed via different SYSAPs.

Adapting SCA to Ethernet

Digital decided to extend the VAXcluster architecture to the Ethernet in order to support workstations and other Ethernet-based systems. The most obvious way to accomplish that extension was to build a port emulator for the CI capabilities on top of the datagram capabilities of the Ethernet adapters. Such a port emulator performs the functions of a CI port in software written as a driver running under the VMS system. SCA extends naturally in this way since the Ethernet has the fundamental properties expected of a network to be used by SCA. That is, Ethernet is a multiaccess media in which the nodes need not be concerned with how packets are routed to their final destinations.

SCA Performance

VAXcluster performance greatly depends on the performance of SCA, in terms of messages and bytes transferred per second, and on the overhead on the system software performing the transfer. Not only does SCA perform storage access, it also sends the lock manager messages that allow VAXcluster systems to share devices and files. SCA, together with the CI port design, is indeed a high-performance and low-overhead interconnect. For example, on a VAX-11/780 system, over 3000 sequenced-message round trips per second can be exchanged with another VAX system. Yet, only about 300 microseconds of CPU overhead are required to send and receive each message pair. Each CI port will sustain approximately 2 megabytes per second of named-buffer transfer bandwidth with no overhead on the part of the CPU. Each mass storage operation requires a sequenced-message pair and a named-buffer transfer initiated by the HSC50 disk controller. Therefore, the CPU overhead of SCA software alone for these functions is only about 300 microseconds. The storage transfer itself can proceed at the rate of about 2 megabytes per second for long transfers to disk or between host systems.

Summary

SCA is a high-performance network architecture developed to allow the CI interconnect to be shared among the various functions required in VAXcluster systems. Among these functions are mass-storage and tape-storage access, which had traditionally been done using direct control over a bus instead of a network message-passing protocol. SCA has proven to be a highly efficient means both to control storage access and to allow VAX host systems to communicate.[4] Its flexibility permits its use to be extended to direct local-storage controllers and to other interconnects such as Ethernet.

Acknowledgments

A large number of people contributed to the SCA architecture and its implementations. Without their efforts, SCA could not have met its goals and would not be so widely used. Thanks also to all the folks who have reviewed and contributed helpful suggestions to this paper.

References

1. A. Lauck, D. Oran, and R. Perlman, "Digital Network Architecture Overview," *Digital Technical Journal* (September 1986): 10–24.

2. N. Kronenberg, H. Levy, W. Strecker, and R. Merewood, "The VAXcluster Concept: An Overview of a Distributed System," *Digital Technical Journal* (September 1987, this issue): 7–21.

3. W. Snaman, Jr. and D. Thiel, "The VAX/VMS Distributed Lock Manager," *Digital Technical Journal* (September 1987, this issue): 29–44.

4. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distribution System," *ACM Transactions on Computer Systems*, vol. 4, no. 2 (May 1986): 130–146.

William E. Snaman, Jr.
David W. Thiel

The VAX/VMS Distributed Lock Manager

The VMS distributed lock manager provides the synchronization mechanism needed to ensure transparent and reliable data sharing between nodes in a VAXcluster system. The lock manager provides services for mutual exclusion and event notification, and achieves high performance by minimizing the number of messages sent between nodes. The lock manager also handles deadlock situations with a minimum of messages exchanged. Since processors systems can join or leave a cluster at any time, a connection manager was developed to handle reconfigurations in a dynamic, efficient manner.

Development Background

As people and organizations came to depend heavily on computer systems to perform their daily activities, it became increasingly obvious that they needed continuous access to the vital data stored in those computer systems. Moreover, growing organizations were faced with a need to incrementally increase the amount of computing power available to them over an extended period of time. In the past, their options were usually limited to either buying more than needed initially or facing painful upgrades and application conversions as the systems were outgrown. The emergence of bus technologies, such as Digital's Computer Interconnect (CI) and the Ethernet, provided an opportunity to combine multiple processors and storage controllers into closely coupled distributed systems. Such systems could provide the needed data availability and incremental growth characteristics. The VAXcluster system was developed to answer those needs.[1]

To encompass the VAXcluster concept, the VMS operating system was extended to provide transparent data sharing and dynamic adjustment to changes in the underlying hardware configuration. These extensions make it possible for multiple processors, storage controllers, disks, and tapes to be dynamically added to a VAXcluster system configuration. Thus a small system can be purchased initially and expanded as needed by adding computing and storage resources with no software modifications or application conversions. New devices can even be added without shutting down operations. The ability to use redundant processors and storage controllers virtually eliminates single points of failure.

The VMS software running on each processor node in a VAXcluster system provides a high level of transparent data sharing and independent failure characteristics. Each processor runs its own copy of the operating system and interacts with the other processors to form a cooperating distributed operating system. In this system, all disks and the files residing on them are accessible from any processor in exactly the same fashion as if those files were connected to a single processor. They can be transparently shared at the record level by application software.

One of the challenges of putting together such a system is to provide both maximum performance and a very high level of reliability. A data-sharing model was chosen as the design center rather than a client-server model. In the data-sharing model, data resources are made directly available to all processors, which must coordinate their accesses to those resources. This model contrasts with that of the client-server, in which the server mediates access to the data. The data-sharing model eliminates potential bottlenecks that develop around heavily utilized servers, provides better opportunities for parallelism, and avoids the server as a single point of failure.

In 1982, the first lock manager was provided in version 3.0 of the VAX/VMS operating system.

The lock manager provided synchronization services for multiple processes residing on a single processor, as well as deadlock detection.[2] Concurrently, design work was under way for a distributed version of this lock manager. The distributed lock manager was released in 1984 with version 4.0 of the VAX/VMS operating system; the CI bus was used as the communications medium. In 1986, the Local Area VAXcluster system was released.[3] This system has the same locking and other algorithms as the CI-based VAXcluster system, but uses the Ethernet as the communications interconnect.

Lock Manager Description

This paper describes the distributed lock manager, which is the basic synchronization mechanism for VAXcluster systems. The lock manager permits the high degree of transparent data sharing attained by the VMS system by providing a set of services used by cooperating processes to synchronize access to shared resources. These processes can reside on any or all of the VAX processors that comprise a VAXcluster system. In this paper, the terms "node" and "processor" are used interchangeably to refer to VAX processors.

Each resource in a VAXcluster system is represented by a unique abstract name that is agreed upon by all the cooperating processes. This name is entered into a distributed global namespace that is maintained by the distributed lock manager. Cooperating processes can use the lock manager as a mechanism to mediate access to a resource by requesting locks on the abstract representation before accessing the actual resource.

The lock manager does not actually allocate or control the resource, and there is no requirement that the name represent an actual physical resource. This permits the lock manager services to be used for event notification and other communication functions, in addition to mutual-exclusion functions. Deadlock detection is also provided.

To permit maximum concurrency, resource names can be tree structured, and locks can be requested at modes that permit varying degrees of sharing. Many resources have an inherent hierarchical structure that permits different parts to be accessed by different processes at the same time. For example, a disk can contain various files, each in turn containing records. This structure allows different records of the same file, and different files to be updated concurrently. Providing tree-structured resource names permits locks to be requested at different levels of the hierarchy.[4]

In the lock manager, six lock modes are represented by an abstract matrix that defines whether or not a given mode is compatible with another mode. An application designer can interpret these modes as setting limits on how a resource can be accessed (e.g., no access, read, or write). The modes can also be interpreted as setting limits on how a resource is shared (i.e., permit read access, write access, or no access to others). Lock requests that are granted at one mode can be converted to a more or less restrictive mode. Table 1 describes the compatibility of each lock mode; Table 2 contains the suggested interpretation of each mode.

The services provided by the distributed lock manager are flexible enough to be used by cooperating processes for mutual exclusion, synchronization, and event notification. These services are known as the $ENQ (lock) and $DEQ (unlock) system services. The $ENQ system service allows a process to request a lock on a resource. The lock request is then either granted or denied by the lock manager, based on the mode of other locks that are granted on the resource. The $ENQ service allows a caller to queue a lock request and either wait for the request to be granted or continue execution. The caller can also signify that the request should not be queued. In this case the status is returned in the event that the request cannot be granted immediately.

If a caller chooses to queue a lock request and continue execution, the $ENQ service provides asynchronous notification when the lock request is granted. The caller can specify a routine to be called when the lock request is granted. This

Table 1 Compatibility of Lock Modes

Mode of Requested Lock	Mode of Currently Granted Lock					
	NL	CR	CW	PR	PW	EX
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No
PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

NL – Null lock
CW – Concurrent write
PW – Protected write
CR – Concurrent read
PR – Protected read
EX – Exclusive lock

ability to specify a routine permits queuing a request in a way that leaves the process free to carry on other functions until the request is granted. The notification mechanism used is called a completion asynchronous system trap (AST).

The $ENQ service also provides a notification mechanism whereby a process that has been granted a lock on a resource can be notified when another process is waiting for it to release the lock. This mechanism, known as a blocking AST, can provide an important performance optimization when a resource is shared infrequently. After acquiring a lock, the holder can access the resource multiple times without further locking until notified by a blocking AST that another process is waiting for it to release the lock. The holder then stops accessing the resource and releases the lock, thus permitting the lock request of the other process to be granted.

Applications can be designed that dynamically change their locking protocol from blocking ASTs (during periods of low contention) to a request-release protocol (during periods of high contention).

Another use for the blocking AST is to implement a "door-bell" notification mechanism in which a process takes out a lock and specifies a blocking AST. When another process wants to get the first process's attention, it makes an incompatible lock request that results in the delivery of a blocking AST to the first process.

A 16-byte value block associated with each resource functions as a small piece of global memory that is atomically updated. The contents of a value block are optionally returned when a lock is granted, and updated when an exclusive (EX) or protected write (PW) mode lock is released. Parameters on the lock and unlock requests control the use of a value block.

A value block can be used to help implement local caching of disk data. The resource represents the data being accessed and locks are used to provide mutual exclusion. A value block associated with the resource is used to maintain a sequence number representing the current version of data stored on the disk. Whenever data is initially read from the disk into a local buffer, a lock is first obtained, and the version number contained in the value block is saved with the data that is read. Whenever the data is to be modified, a lock is first obtained, then the buffer is updated and written back to the disk. When the lock is released, an updated version number is stored in the value block representing the new version of the data on the disk. Upon subsequent reads by this or any other node in the VAXcluster system, a lock is first obtained, and the sequence number contained in the value block is compared to the sequence number stored with the locally cached data. Whenever the sequence numbers match, the cache is valid and no disk read is required.[5]

Value blocks can also be used for communication between processes.

The $DEQ system service is used to indicate that a process no longer wants to maintain a lock on the resource. Part of its function is to optionally update the value block when the mode of the lock being released is either PW or EX.

Table 2 Modes at which Locks Can Be Requested

Mode	Suggested Interpretation of Mode
NL	Null mode grants no access to the resource; it is typically used either as an indicator of interest in the resource or as a place holder for future lock conversions.
CR	Concurrent read mode grants read access to the resource and allows its sharing with other readers. The concurrent read mode is generally used either when additional locking is being performed at a finer granularity with sublocks or to read data from a resource in an "unprotected" fashion (allowing simultaneous writes to the resource).
CW	Concurrent write mode grants write access to the resource and allows its sharing with other writers. The concurrent write mode is typically used either to perform additional locking at a finer granularity, or to write in an "unprotected" fashion.
PR	Protected read mode grants read access to the resource and allows its sharing with other readers. No writers are allowed access to the resource. This mode is the traditional "share lock."
PW	Protected write mode grants write access to the resource and allows its sharing with concurrent read-mode readers. No other writers are allowed access to the resource. This mode is the traditional "update lock."
EX	Exclusive mode grants write access to the resource and prevents it sharing with any other readers or writers. This mode is the traditional "exclusive lock."

Design Constraints and Goals

Several constraints were placed on the design of the distributed lock manager, the most important one being that it had to be extremely reliable. This constraint was vital since the VMS file system, the Record Management System, several database systems, and other critical products would depend on the lock manager to maintain the integrity of their resources. The lock manager had to be general enough so that many different applications could be built using its services, thus avoiding the creation of a separate synchronization tool for each application. Moreover, the lock manager had to have very high performance characteristics and be able to tolerate the failure of an arbitrary number of processes or nodes.

For performance reasons, it was essential to minimize the number of messages exchanged between the various nodes. This was especially important as the number of nodes increased. Additionally, minimum penalties should be imposed when all the cooperating processes reside on a single processor. The goal was to have the cost increase no more than linearly as the number of nodes increased. In fact, what was attained was a cost bounded by a small constant that is independent of the number of nodes that exist in a VAXcluster system.

Relationship between the Distributed Lock Manager and the Connection Manager

As the lock manager was being developed, it became clear that a need existed to separate the function of managing a dynamic configuration of processors from that of managing the resource namespace. This separation required the creation of a new entity, the connection manager. The distributed lock manager relies on the connection manager for several vital services.

The connection manager maintains a globally consistent list of all processors that are in the VAXcluster system at any given instant. To maintain this consistency, the connection manager utilizes a very strong notion of cluster membership and orchestrates the addition and removal of nodes. Part of that orchestration process is the coordination of the distributed lock manager's task of rebuilding a database describing the locking namespace and state whenever the configuration changes.

Another function of the connection manager is to prevent the partitioning of the namespace. This partitioning could happen if the distributed lock managers in disjoint subsets of nodes operated independently. They could do so in the event of a communications failure, or a "rolling" power failure a nd recovery cycle. In these situations, any objects accessible to multiple subsets could be inconsistently accessed and therefore corrupted. The connection manager uses a voting algorithm to ensure that the set of available processors cannot be split into two or more functioning groups if communications fail. This approach requires that only a very limited amount of global information (i.e., the number of votes held by a node and the total number of votes available to the entire set of member nodes) be known by each system. Furthermore, protection is given against a very wide set of failures because there are no additional underlying assumptions about failure mechanisms.

The final function is a communications service that provides a virtual circuit between each member node of a VAXcluster system. This service ensures the reliable delivery of sequenced messages. If messages cannot be delivered in sequence, the virtual circuit will break. The most significant characteristic of this service is that cluster membership and the existence of the virtual circuit are tightly coupled. The virtual circuit must exist for a pair of nodes to become or remain part of a VAXcluster system. A failure of the virtual circuit, therefore, requires the removal from the cluster of at least one of the nodes terminating that circuit. This approach greatly simplified the design of the distributed lock manager because only one type of communications failure is visible to it. The required action upon the occasion of such a failure is made simpler because it is certain to be followed by a change in the cluster's membership. Such a change involves rebuilding the distributed lock manager's database.

The Operation of the Distributed Lock Manager

The following section describes the operation of the distributed lock manager when all lock requests can be granted immediately. A later section discusses its operation under conditions of contention. Table 3 gives definitions of the terms used in describing these operations.

Table 3 Terms and Definitions

Term	Definition
Resource tree	The lock manager allows names to be structured in a hierarchical fashion. For example, the root resource can represent a device; its child, referred to as a sub-resource, can represent a file on that device; and another subresource beneath it can represent a record.
Lock request	The request by a process for a lock on a resource.
Root-lock	The lock request for a resource at the root of a resource tree.
Sublock	The lock request for a resource below the root of a resource tree.
Resource manager	The node that controls the granting of lock requests on a given resource tree for which it maintains information about all granted and waiting lock requests. All nodes are potentially resource managers, each handling a particular subset of the set of resource trees.
Directory service	The directory service provides a mechanism to locate the current resource manager. This service is needed because lock requests must be directed to the resource manager, which may change over time. The directory function is distributed among the various nodes in a VAXcluster system, each node providing the function for a subset of the resource trees. This distribution eliminates potential performance bottlenecks.
Lock mode	The mode of a lock request indicates the type of lock being requested, such as NL, PR, or EX. By convention, the mode represents the type of access to the resource that is being requested, such as read, write, or no access. It also indicates a willingness to permit others to share the resource.

An Initial Lock Request on a Root Resource

When a process somewhere in a VAXcluster system requests a root-lock, the distributed lock manager must first identify which node is currently managing the resource tree. The resource name specified by the lock request is hashed, and the resultant value is applied to a vector containing zero or more entries for every node currently in the cluster. The selected vector entry identifies the directory node for the resource specified. A message is then sent to this node requesting a lock on the resource. The building and sending of a message can be avoided if the node making the request is also the directory node.

The vector is maintained by the connection manager, which ensures that the vector is updated whenever a node enters or leaves the cluster. The connection manager also ensures that the vector is identical on all nodes. Each node can request that it be entered zero or more times in the directory vector, depending on the extent to which the node wants to participate in the distributed directory function.

Upon receiving the message, the directory node can respond in any of three ways. First, it can indicate that the node making the request should manage the resource itself. Second, it can indicate that the request should be re-sent to another node that is already managing the resource. Finally, it can respond to the request directly, since the directory node itself may already be managing the resource. If this lock request is the first one on the resource, the directory node will instruct the requestor to manage the resource itself. It will also create a directory entry for the resource, thus ensuring that subsequent requests from other nodes will be directed to the new resource manager. Figure 1 illustrates this case.

All subsequent lock requests for additional root-locks or sublocks on this resource from the node that originated the initial request will now be processed without further message traffic, since the node is now managing the resource itself. This action, called local locking, was developed to minimize the cost of locking should all the processes sharing a resource reside on one node. Figure 2 provides an illustration of local locking.

At this point, if a process residing on another node makes an initial root-lock request, the resource name is again hashed and the directory node identified in the same fashion as before. The request is sent to the directory node, which responds by identifying the node currently man-

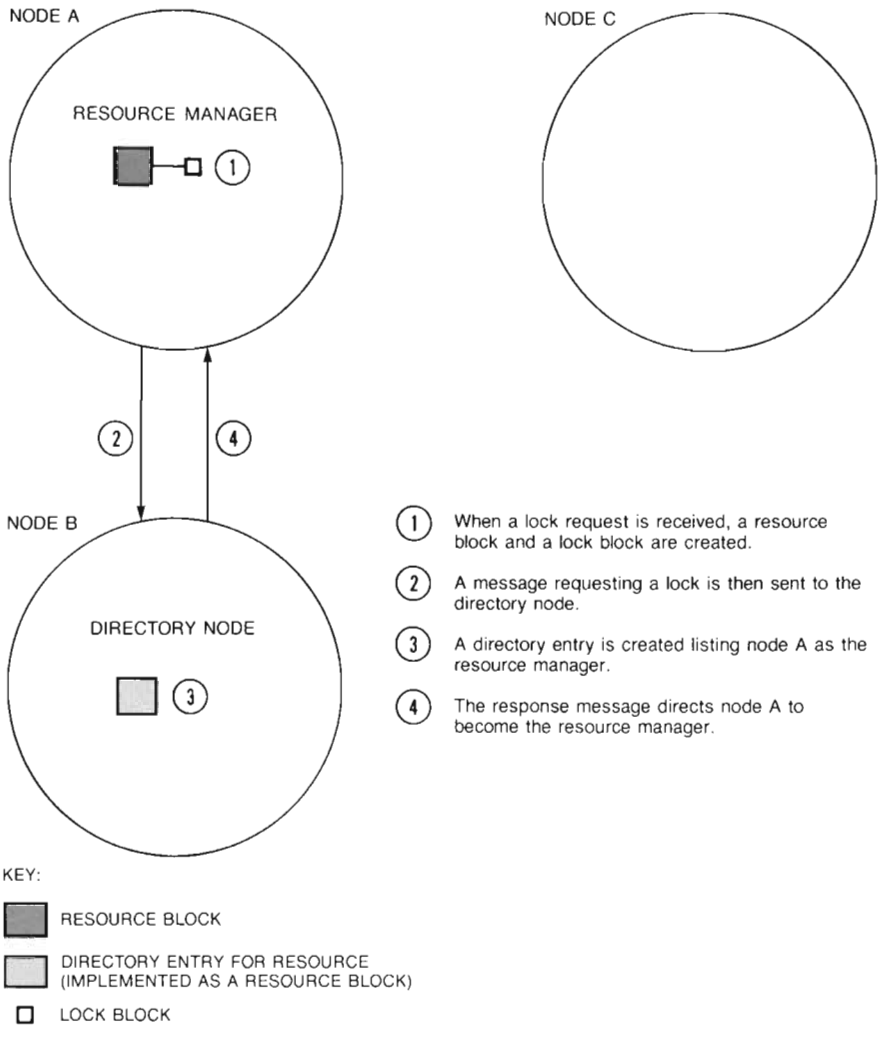


Figure 1 A Root-lock Request When No Resource Manager Exists

aging the resource. Upon receiving the response, the requestor re-sends the lock request to that node.

This case is potentially the worst with regard to messages since one round trip is required to the directory node (assuming that it is another node in the VAXcluster system) and another round trip to the resource manager. Note that this cost is bounded by a small constant with respect to the number of nodes in a VAXcluster system. Figure 3 illustrates this case.

Subsequent Root-Lock and Sublock Requests

Once a lock on a root-level resource has been established, the identity of the resource-manager node is known. After that point no further messages are sent to the directory node by that processor; all requests are sent directly to the resource manager. If the lock request is made on a node that is not the resource manager, two messages are required for every lock request after the first: a request, and a response. This process is called remote locking. Figure 4 illustrates the remote locking concept.

Releasing Lock Requests

When a process residing on the node managing the resource decides to release a lock, no messages are sent unless the lock is the last remaining one on the resource. In that event a message is sent to the directory node indicating that this

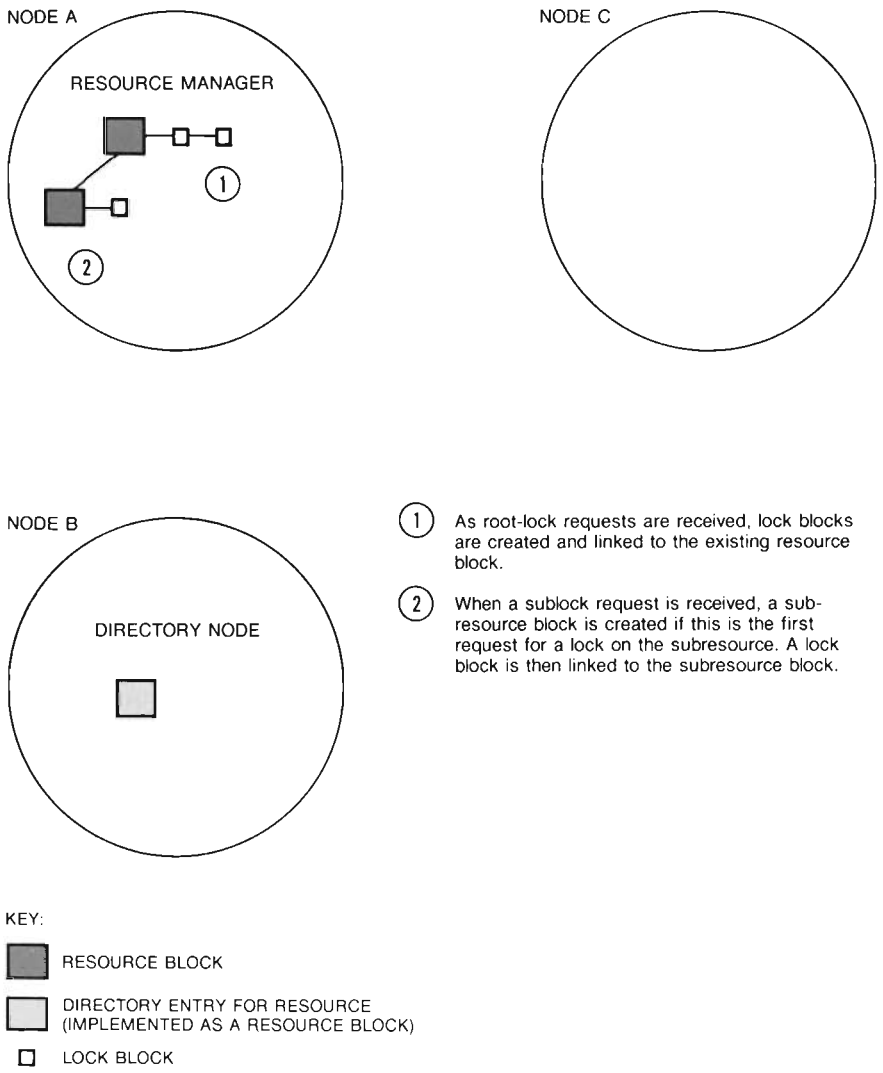


Figure 2 Root and Sublock Requests Made on the Resource Manager

node is no longer managing the resource. The directory node then deletes the directory entry for the resource. This deletion allows the next node requesting a lock on the resource to become the resource manager. No response is necessary because the message delivery is guaranteed by the connection manager.

For the case in which a process releasing a lock does not reside on the node that manages the resource, a message is sent to the resource manager. Again, if this is the last remaining lock on the resource, the resource manager sends a message to the directory node indicating that this node is no longer the resource manager. Figure 5 illustrates the concept of unlocking.

Converting Lock Requests

The lock manager also permits the mode of a granted lock to be altered. This action is called a conversion. Conversion requests can be processed more efficiently than new lock requests because all the data structures are already in place, and the resource manager has already been identified. If a conversion request is made on the node managing the resource, no messages need be exchanged. If the resource manager is not the node on which the request is being made, either one or two messages are required. For example, in some cases in which the requested mode is compatible with the granted mode, the request

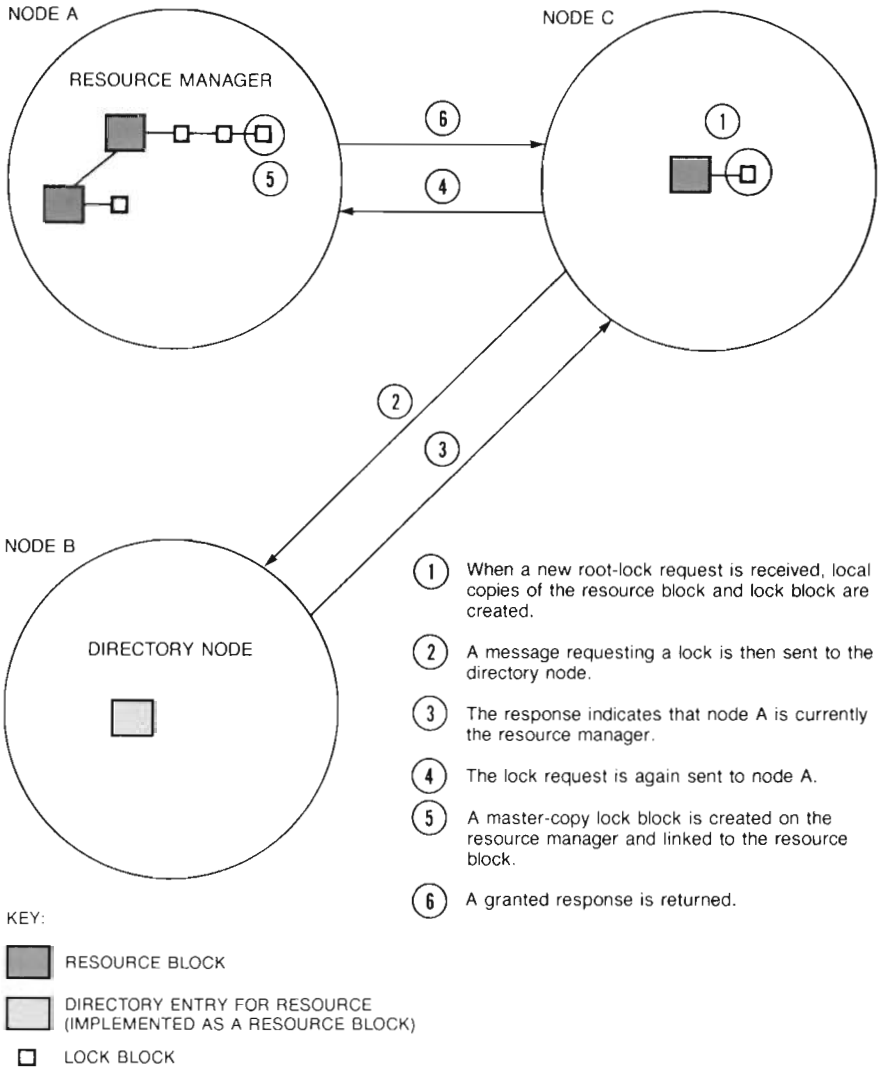


Figure 3 New Root-lock Request When a Resource Manager Exists

can be unilaterally granted, and a single message sent to notify the resource manager of the change. In others, the resource manager must make a decision based on the other requests that are granted. A request is then sent to the resource manager, who must respond. In all cases, no communications are required with the directory node. Figure 6 illustrates a conversion request.

Operation During Periods of Resource Contention

The operation is slightly more complicated during periods of contention. When a resource manager receives a lock request that cannot be granted because an incompatible lock exists, two actions are required. First, all holders of incompatible locks that have indicated a desire to receive blocking ASTs must be notified that a process is waiting. To accomplish this, a message is sent to each node where a lock holder resides. The process holding the lock is notified only once, even though it may be blocking multiple lock requests. Second, the requester of the lock must be told to wait; this is accomplished by sending a response to the lock request. When the blocking lock is later released, a message is sent to each waiting requestor indicating that the lock is now granted. Table 4 summarizes the numbers of messages used for different types of lock requests.

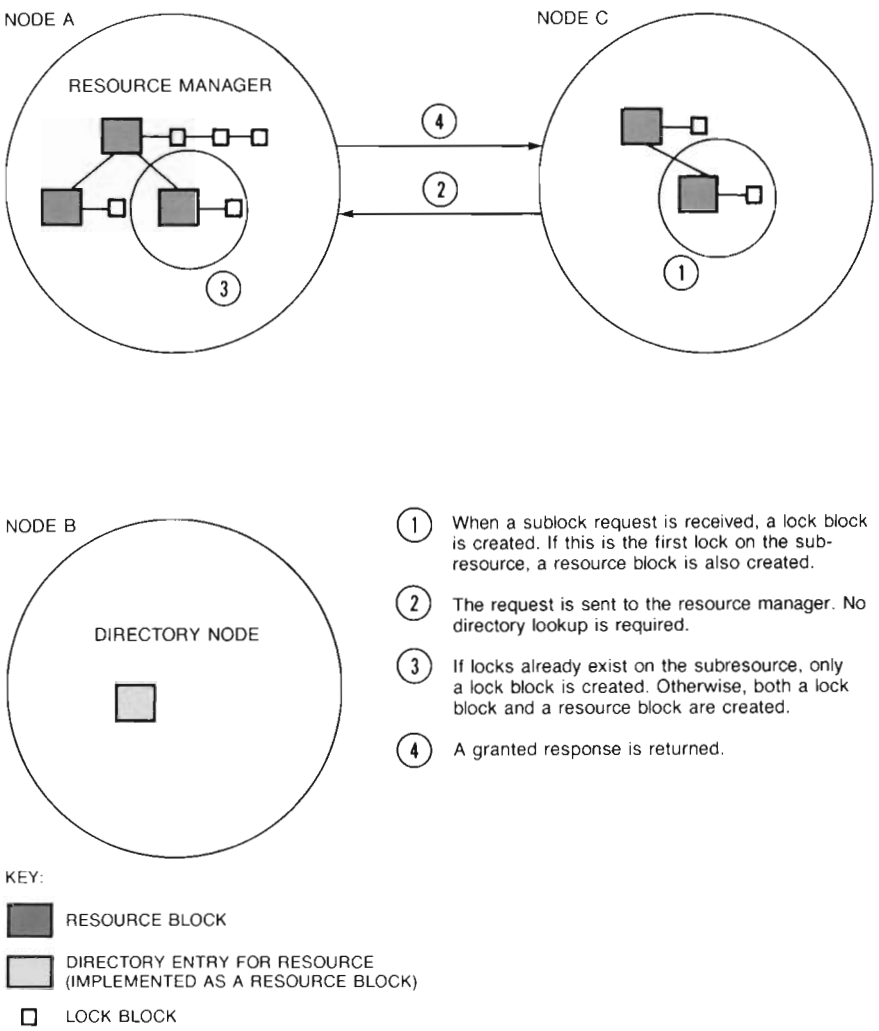


Figure 4 A Sublock Request on a Node that Is Not the Resource Manager

Scaling Behavior of the Distributed Lock Manager

It can be shown that the number of messages required for any locking operation is bounded by a small constant that is independent of the number of nodes, or cluster size, in a VAXcluster system. This section addresses how the size of the data representing the locking state and the total number of locking messages vary with a cluster's size.

The distributed lock manager uses a fixed-size control block to represent both a lock and a lock request. An instance of this control block exists on the node requesting the lock. If the resource manager is a different node, another instance exists on the resource manager. A resource is represented by another fixed-size control block. An instance of this control block exists on each node requesting the lock, on the resource manager, and on the directory node. Whenever any of these categories overlap (i.e., requestor, resource manager, and directory node), only one instance of the control block is present. The control blocks for locks and resources are dynamically allocated and deallocated.

At least one lock is represented for every resource represented. Conversely, a resource is represented for every lock represented. For each lock, the upper bound on the storage requirements is two lock control blocks and three resource control blocks. This upper bound is usually quite loose and depends on a cluster's size.

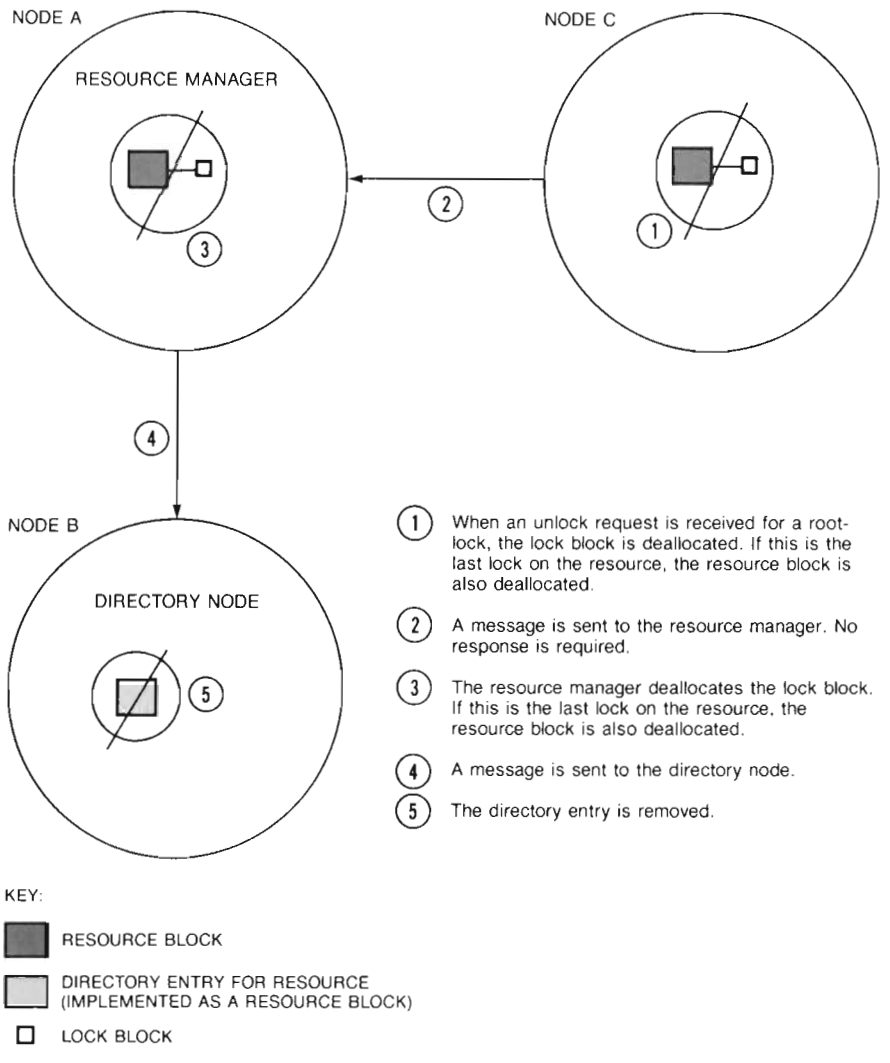


Figure 5 Unlock Request for the Last Remaining Lock on a Root Resource

VAXcluster applications are typically designed so that their algorithms do not change as the size of the cluster changes. Therefore, an instance of a typical application running on one node exhibits a behavior with respect to the number of outstanding locks and the frequency of locking operations that is independent of the number of additional instances of that application running on the same or other nodes. If multiple instances of the application are running, the number of outstanding locks and the frequency of locking operations increase in proportion to the number of copies of the application, independent of the cluster size.

Both the number of messages per locking operation and the storage requirements for a lock are bounded by constants that are independent of the cluster size. Therefore, the rate at which messages must be exchanged and the total storage required to represent the locking state are proportional to the number of instances of the application that are running, which is also independent of the cluster's size. If the number of instances of the application is proportional to the cluster size, the rate of message exchange and the total storage required to represent the locking state are both bounded by a constant times the cluster size.

This argument is also valid when multiple instances of each of several applications are present.

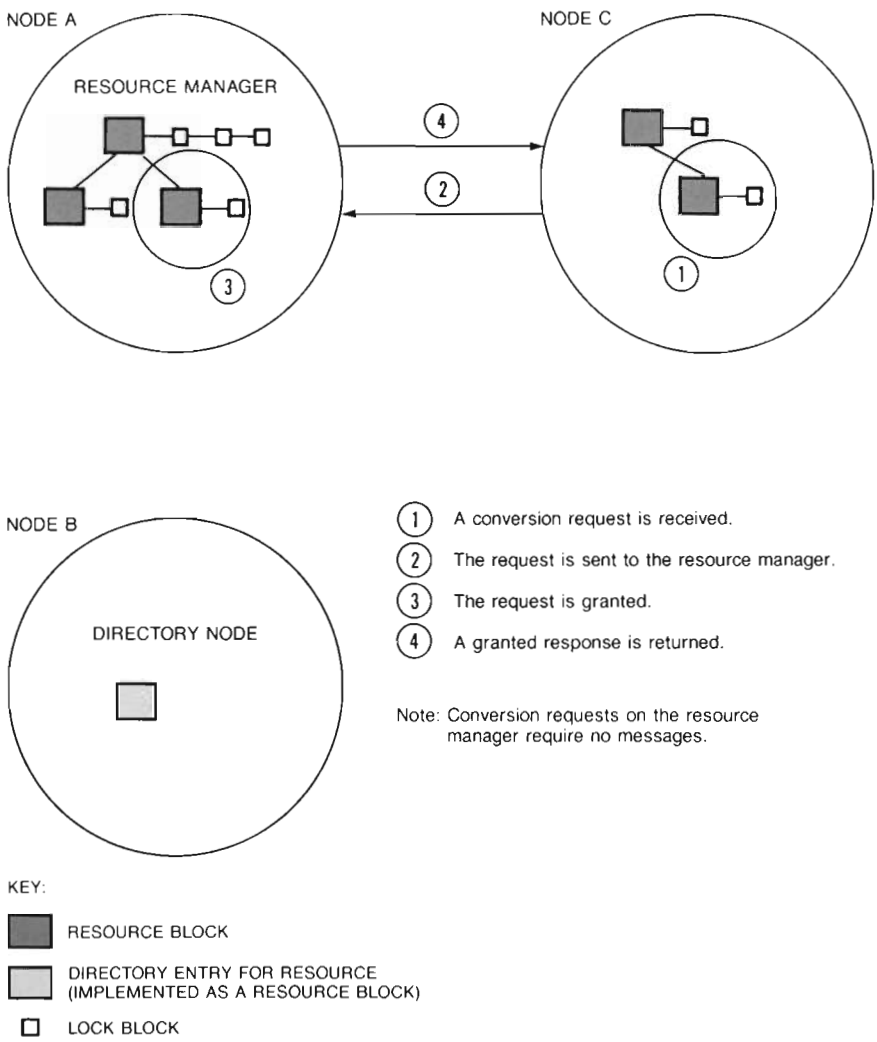


Figure 6 Conversion Request on a Node that Is Not the Resource Manager

These characteristics of the distributed lock manager (i.e., total space and message traffic behavior that is subject to a linear bound in the "workload") are a significant fact or in allowing VAXcluster systems to act as distributed operating systems. These characteristics suggest that, from the distributed lock manager's viewpoint, additional growth in the size of a VAXcluster configurations is certainly viable.

Performance Aspects of the Distributed Lock Manager

Table 5 summarizes the performance of the distributed lock manager. The measurements reflect operations that are normally done in pairs. Such operations include an $ENQ followed by a $DEQ, and a conversion to a more restrictive mode (up) followed by a conversion to a less restrictive mode (down). The operations reported in the table are performed on sublocks.

When Processors Join or Leave the VAXcluster System

The connection manager plays a major role in the lock manager's ability to deal with configuration changes when one or more nodes join or leave the VAXcluster system. When the membership of the cluster must be altered, a coordinator node is elected to lead the other nodes through the state transition. Any node can become the coordinator

Table 4 Summary of Number of Messages Used for Lock Requests

Request Type	Messages	Comments
Initial root-lock request from a system for a previously unknown resource (i.e., no manager exists)	2 or 0	Zero messages if node making the request is the directory node. Otherwise two messages; a directory lookup request followed by a "do local" response.
Subsequent root-lock requests on resource manager	0	
Sublock request on resource manager	0	
Unlock request on resource manager with locks remaining		
Unlock of last lock on resource by resource manager	1 or 0	Remove directory entry message sent to directory node. No message sent if manager is also directory node.
Initial root-lock request from a system for a resource that is known (i.e., a manager exists)	2 or 4 (1)	If requester is the directory node, two messages consisting of a lock request followed by a response from the manager. If requester is not directory node, do a directory lookup, a resend to manager response, a lock request to the manager, and a response back.
Sublock requests and subsequent root-lock requests from a system that is not resource manager	2 (1)	Lock request to manager and a response back.
Unlock request from a system that is not the resource manager	1 or 2	Dequeue message to manager. Manager may then send a remove directory message to directory node if this lock is the last one.

NOTE: If the lock request cannot be granted immediately, add one message. If the lock is granted, blocking another request, and a blocking AST was requested, add one message. In all cases the number of messages is independent of the number of nodes in the VAXcluster system.

Table 5 Performance Summary of the Distributed Lock Manager

VAX-11/780 VAXcluster System Locking Using the Computer Interconnect (CI780)

	Local Locking	Remote Locking		
	Local CPU	Local CPU	Remote CPU	Elapsed Time
ENQ + DEQ	0.6	2.7	1.5	3.9
CVT (up+down)	0.4	2.4	1.3	3.3

MicroVAX II Locking Using the Ethernet

	Local Locking	Remote Locking		
	Local CPU	Local CPU	Remote CPU	Elapsed Time
ENQ + DEQ	0.7	6.0	4.8	8.1
CVT (up+down)	0.5	5.6	4.6	7.8

- *All numbers are in milliseconds*
- *For Local Locking, Local CPU = Elapsed Time*
- *ENQ refers to a lock operation, DEQ refers to an unlock, and CVT to a mode conversion*

and it is usually the first to discover that a membership change is required. The need for a membership change can result from timing out a broken connection, or upon discovering a new node. All configuration changes are made using a two-phase commit protocol to ensure consistency on all nodes. To add or remove a node, the coordinator describes a proposed configuration to the other members. They have the option of agreeing or disagreeing with the proposed configuration.

They will disagree if they can construct a more optimal configuration based on the number of nodes they can communicate with and on the assignment of votes to those nodes. The resulting VAXcluster system can only consist of a strongly connected group of nodes where every node has a connection to each of the others.

In case of disagreement, the coordinator backs out of the operation, waits a random amount of time, and then initiates the election protocol again. During this interval other nodes can attempt to become the coordinator. Disagree-

ments are quickly resolved so that the node that can put together the most optimal configuration becomes the coordinator. At this point, the new configuration has been described to all nodes and they have agreed; therefore, commit messages are sent.

Thus the connection manager is able to provide the distributed lock manager with a consistent view of the processors that are members of the VAXcluster system. The connection manager can also ensure that the vectors used to identify the directory node for a given resource are identical on all nodes. In addition, the manager assigns a unique identifier, called the cluster system ID (CSID), to each processor admitted into the VAXcluster system.

At the completion of any change in membership, the connection manager leads the other nodes through a lock database rebuild. The node that was the coordinator now takes on the role of a synchronizer. Each node begins to execute a series of action routines that control how the lock database is to be rebuilt. Each action routine describes a particular step in the rebuild process, and all nodes execute the action routines in parallel.

One or more action routines are separated by synchronization steps. Upon reaching a synchronization step, a node sends a message to the synchronizer indicating that that node has completed a step and is waiting for notification to proceed with the next one. After receiving this message from each processor in the VAXcluster system, the synchronizer sends a message to each node telling it to proceed with the next step. This process continues until all action routines have been executed and the lock database has been rebuilt on all nodes.

From the viewpoint of the distributed lock manager, the actions taken are identical when nodes are added or removed. This redistributes the management of resource trees to prevent the management of most of them from migrating to the "oldest" member of the VAXcluster system.

Upon discovering a broken connection to a remote node, the connection manager initially assumes that this condition is temporary and attempts to restore the connection for a specified interval that depends on the installation. During this interval, normal activity can generally proceed. Lock-request and other messages addressed to the remote node and sent using the connection manager's message delivery service are queued pending the re-establishment of the connection. If the connection is re-established, the queued messages are sent in the original order, and the sender remains unaware that a problem existed.

If the connection cannot be re-established within a specified interval, the connection is declared irrevocably broken, and a cluster reconfiguration is required. Locking is disabled on all nodes during a reconfiguration. Lock requests can still be made, but the processes making them will be blocked pending completion of the state transition.

The lock database is rebuilt in the following fashion by each node. First, new lock requests are disabled. Then, the lock database is scanned and all directory information is removed, since a change in membership redistributes the directory functions. Information about locks that are either held or requested by processes on other nodes is also discarded. These actions result in a period of time during which no directory nodes and no resource managers exist. The only information retained concerns the lock requests made by processes actually residing on a node.

At this point the nodes re-acquire all the locks held before the membership changed, using the same algorithm by which the locks were initially acquired. Locks that were waiting to be granted are re-ordered by a sequence number that was assigned when they were queued so that the order in which they wait is preserved. By the process of re-acquiring locks, new directory entries are created and new resource managers chosen. Since each node re-acquires its own locks, the locks held by nodes that are no longer members of the VAXcluster system are released. Once all locks have been re-acquired, an attempt is made to grant waiting locks since the removal of lock requests contributed by a failed node may permit waiting requests to be granted. Once these actions have been accomplished, locking is enabled and activity proceeds normally.

Distributed Deadlock Detection

The requirements for a distributed deadlock algorithm were to minimize the number of messages involved in a deadlock search, find all deadlocks, and not find false deadlocks. Since the distributed lock manager was to be a general-purpose synchronization tool used by many

applications, simplifications based on assumptions about the way it was used could not be made.

From the lock manager's perspective, there are two classes of deadlocks: conversion, and multiple-resource. This distinction is made because conversion deadlocks are easily detected by the resource manager whereas multiple-resource deadlocks are detected by a more complex distributed deadlock algorithm.

A conversion deadlock involves multiple conversion requests on a single resource so that all information will be readily available for the resource manager to identify them. Let us consider a request to convert a lock held at one mode to another more restrictive mode (e.g., from CR mode to EX mode). If another lock is also held at CR mode, the conversion request must wait for the second lock to be released or converted to a compatible mode. If an attempt is then made to convert the second lock from CR mode to EX mode, a conversion deadlock results. The first conversion request cannot be granted while the second lock is still held at the original mode and the second conversion request cannot be granted because it must wait for the first lock to be granted.

A multiple-resource deadlock can be identified by searching for cycles in a "wait-for" graph of processes. A simple example can be constructed with two processes and two resources. Suppose a process P1, which is already holding a granted lock on resource R1, waits for a lock request to be granted on resource R2. A deadlock results if a process holding a lock on R2 that is blocking P1's request attempts to acquire a lock on R1 that is incompatible with the granted lock held by P1.

Distributed deadlock detection is implemented with an algorithm that searches the clusterwide wait-for graph by sending messages to traverse arcs that cross system boundaries. The algorithm using messages to traverse arcs between systems was developed independently both at Digital and at IBM Corporation.[6,7]

One of the assumptions that was made in the design of the lock manager was that deadlock searches would be an infrequent occurrence and relatively costly. This being the case, deadlock searches are initiated only after a process has waited longer than a configuration-specified period. This has the effect of greatly reducing the number of searches that are initiated. For example, if process A on system 1 has a lock request waiting for longer than the deadlock wait interval, then a deadlock search is initiated on its behalf.

Time-outs are detected on the node that is managing a resource so that information about all lock requests on the particular resource is available for the deadlock search. If a conversion request has timed-out, the queue of conversion requests is searched to identify whether the granted mode of any conversion request made after the timed-out conversion request is incompatible with the requested mode of the timed-out conversion request. If one is found, a conversion deadlock exists and a victim is selected. The waiting lock request of the victim is then completed with a error status indicating that a deadlock was found. Granted locks are never affected by victim selection.

If no conversion deadlock is found, a more extensive multiple-resource deadlock search is initiated. The wait-for graph of processes is traversed, beginning with the process owning the timed-out lock request and searching for a path back to that same process. Beginning with the lock request, each process holding a blocking lock on the resource is tested to determine if the process has waiting locks on other resources. For each waiting lock found, the algorithm is applied recursively until either no more waiting locks are found or the initial process is found. In the former case no deadlock exists because no cycle exists. In the later case a deadlock exists because a cycle was found to include the process owning the lock that timed out.

If the arcs of the wait-for graph traverse processor boundaries in the VAXcluster system, messages are sent indicating that the search should be continued on the destination processor. The messages indicate that the search should commence with a certain lock and continue with the ultimate goal of discovering a path to the process owning the timed-out lock request.

In the implementation, two possibilities exist that must be accounted for. In the first, a blocking lock is found that is owned by a process residing on a remote system. In this case the search must be continued on the remote system by identifying all locks that the process is waiting for. In the second, a process is waiting for a lock managed on a remote system. In this case the search must be continued on the remote system by identifying all locks that are blocking the waiting lock.

Let us consider the following example. A waiting lock request L1 owned by process P1 on node N1 times out, and a deadlock search is initiated. The search is initiated on node N2, which manages the resource tree. A blocking lock L2 owned by process P2 located on node N3 is discovered on the resource. A message is then sent to node N3, indicating that a search should be continued there, beginning with the lock L2, with the goal of finding a path to process P1. Upon receiving the message, node N3 determines that process P2 is waiting on lock L3 managed by node N4. A message is sent to node N4 to continue the search starting with lock L3 with a goal of finding process P1. Lock L3 is discovered to be blocked by lock L4 that is owned by process P1. Since a cycle has been discovered, a victim is selected, and its waiting lock request is completed with deadlock status. Deadlock messages contain the identity of the best victim found so far, and a message is sent to the node in which the victim resides.

An interesting extension to the similar algorithm described in reference 6 is used in the deadlock search. To prevent looping on cycles that do not include the process with a timed-out lock request and to greatly reduce the worst-case search time, a bitmap is used to indicate if a process has already been visited in the search. Each node in the VAXcluster system has a bit map with one bit for every process on that node. When the search is initiated, all bits are cleared. If a process has been involved in the deadlock search, its corresponding bit is set. If a message then arrives that indicates that this process should be involved in the search, the message is ignored since all paths from this process have been searched already.

A node never knows when a deadlock search is completed because the messages simply die out when no deadlock is found. Therefore, some way must be provided to determine when the bitmap can be reused for a new search. That is accomplished by assigning a "timestamp lifetime" to the deadlock search. In this scheme, one node is assigned the role of a timestamp server by the connection manager whenever the cluster membership changes. To initiate a deadlock search, a node requests a timestamp from the timestamp server. The timestamp represents a time slightly in the future. Once that timestamp has been issued, the timestamp server will not issue another until that time has passed (i.e., the timestamp has expired). The initial value of the timestamp is 50 milliseconds, based on an estimate of a reasonable worst-case search time. The timestamp is used in the deadlock messages to indicate a specific deadlock search.

Whenever a deadlock message is received, its timestamp is compared to a timestamp stored with the bitmap. The comparison determines how the bitmap is to be used. There are three possible cases, described as follows:

- The message value exceeds the bitmap value – The bitmap was being used by a previous deadlock search and its timestamp lifetime has expired. In this case the bitmap is available for use by the new deadlock search. The bitmap is cleared and the timestamp from the message is saved with it. The new search is then continued.
- The bitmap equals the message value – The bitmap is available and has already been used by an earlier message involved in this search. Proceed with the search. If the bit corresponding to the process requesting the lock is already set, then ignore this message since all paths from this process have already been searched.
- The bitmap value exceeds the message value – The bitmap has been preempted by a subsequent deadlock search. The timestamp assigned to this message expired before the search completed. Abort this deadlock search for now but reinitiate it later with a new timestamp that is double the last timestamp's lifetime.

The bitmap optimization provides not only the performance benefits noted above, but also prevents the algorithm from looping when it encounters unsuspected deadlocks. For example, suppose process A is waiting for B which waits for C which waits for B. Processes B and C have a deadlock that will not be discovered when searching on behalf of process A since the ultimate destination of the search is process A. However, the deadlock will be found when searching on behalf of B or C. The use of bitmap optimization prevents the search from looping when searching on behalf of process A.

Acknowledgments

The authors would like to acknowledge all those who worked to make VAXcluster systems a reality. We especially want to acknowledge Steve

Beckhardt, who designed and implemented the distributed lock manager. Also, our thanks to Steve Neupauer, who supplied the performance figures, and to all those who reviewed this paper.

References

1. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-coupled Distributed System," *ACM Transactions on Computer Systems*, vol. 4, no. 2. (May 1986): 130–146.

2. L. Kenah and S. Bate, *VAX/VMS Internals and Data Structures*, (Bedford: Digital Press, 1984).

3. M. Fox and J. Ywoskus, "Local Area VAXcluster Systems," *Digital Technical Journal* (September 1987, this issue): 56–68.

4. J. Gray, R. Lorie, G. Putzolu, and I. Traiger, "Granularity Of Locks and Degrees of Consistency in a Shared Data Base," IBM Research Report RJ1654 (1975).

5. *VAX/VMS System Services Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-Z501C-TE, 1986).

6. R. Obermarck, "Global Deadlock Detection." IBM Research Report RJ2845(36131) (June 1980).

7. S. Beckhardt, Digital Equipment Corporation Internal Memorandum describing the deadlock detection algorithm used by the VMS operating system.

Andrew C. Goldstein

The Design and Implementation of a Distributed File System

The advent of VAXcluster systems, with their simultaneous requests for storage data, altered the requirements of the file functions in the VMS software. To replace the single-system process, an extended QIO processor was developed to synchronize file accesses. The locks in the VMS lock manager provide that synchronization by arbitrating and blocking requests. Deadlock is prevented by taking out locks in a consistent order. Proper cache management is ensured by locks with sequence counters and a set of synchronization queues. This total scheme works so well that, in addition to VAXcluster hosts, it is used for single systems as well.

The VMS file system provides basic file-management facilities to all VMS users and to many other components of the VMS system itself. From a raw disk, which consists simply of a series of data blocks, this file system provides files and file management, directories, security enforcement, and a variety of functions related to the intricacies of managing a file structure. The VMS interface to the file system is the $QIO system service.[1] The $QIO read and write functions provide block-level access to file data. Other $QIO functions specific to the file system create, access, modify, and delete files.

The $QIO service normally leads to the VMS driver context. This context consists of initial kernel-mode execution in the process context, with few system services allowed, followed later by interrupt-level execution. The complexity of the file system makes it impractical to execute in the normal driver context. Therefore, the VMS system provides two methods for extending the operating context of the file system to provide the richness needed to support its complexity.

The Ancillary Control Process

In VMS releases 1 through 3, a technique called the ancillary control process (ACP) extended the file system's context. An ACP is a separate VMS system process that executes in a privileged context. All the VMS services normally available to processes are available to the ACP, thus making feasible the implementation of complex code. The I/O processing routines (the FDT routines) in a process context send $QIO functions for the file system to the ACP. In turn, the ACP executes the functions in its own context, returning completion data and status to the caller by using the I/O completion routines in the VMS kernel. An extension of the VMS buffered-I/O mechanism copies both the caller's arguments to the ACP and the return parameters back to the caller.

In addition to the extended execution environment, the ACP concept provides an important facility to the file system: synchronization. The VMS file system ACP executes user functions in a single stream, completing each function before starting the next one. Thus all file functions are inherently synchronized because only one ACP performs file management on a volume. Moreover, the implementation of a file system cache becomes quite simple and straightforward when operating in the single-process context. Figure 1 depicts the ACP-based file system.

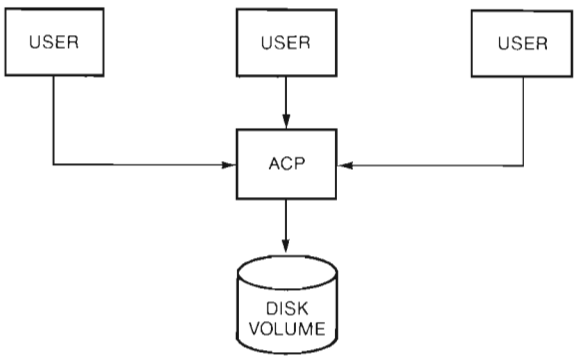


Figure 1 ACP-based File System

Cluster Alternatives

Many of the attributes that made the ACP concept attractive were invalidated when the VMS software had to support the VAXcluster concept. VAXcluster systems require that each disk volume be accessible to all host systems in the cluster. Therefore, a disk volume can no longer be served by a single process. We examined other concepts, including having a single "master ACP" for a volume on one member of the cluster. That ACP would then execute all file functions for all cluster members. We rejected this approach, however, because of the high availability requirements of VAXcluster systems. Transferring the file system context to another cluster member in the event of a failure would have been very difficult.

Based on those considerations, we chose an approach that uses a symmetrical file-management design in which the file functions execute on the cluster member on which they originate. No longer having the implicit synchronization and cache management of a single ACP, we were now forced to address those issues explicitly in the distributed system.

The Extended QIO Processor

Our need for an explicit synchronization scheme eliminated one of the major attractions of the ACP: its implicit synchronization. In addition, it seemed redundant to have two schemes — one implicit (ACP) and one explicit — to manage file operations. Therefore, rather than using explicit synchronization only between cluster members, we chose to use it for all operations, including those local to one processor. As a result, we developed the second operating context for the file system now available in the VMS software: the extended QIO processor, or XQP. The XQP executes as an asynchronous system trap (AST) thread at the kernel level in the context of the calling process. An extended kernel stack and a data area located in the process's P1 region provide the necessary execution context. Since execution occurs at interrupt priority level (IPL) 0, all the basic system services can be used. Figure 2 depicts the XQP-based file system.

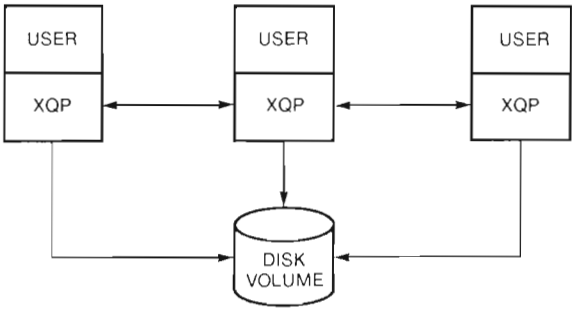


Figure 2 XQP-based File System

The XQP design for the file system has several advantages over the distributed master-ACP design:

- Consistency – All file operations are synchronized in the same way, whether the volume is accessible clusterwide or not. This technique simplifies the synchronization design and provides fewer opportunities for bugs.
- Performance – We eliminated the process context switch associated with an ACP call by running the file system in the context of the caller.
- Concurrency – Multiple file operations can proceed concurrently, in many cases, by implementing explicit synchronization where it is needed, thus improving system performance.

The remainder of this paper concentrates on the problems unique to the VAXcluster distributed-file system: synchronization, and cache management.

Synchronization

The file system requires synchronization for two basic reasons:

1. File structure integrity – Multiple users must be prevented from simultaneously modifying the same parts of the file structure (e.g., attempting to find and allocate the same piece of free disk space to different files).
2. File system semantics – Certain file operations provide user-level synchronization (e.g., preventing two users from simultaneously accessing the same file in a conflicting manner).

Synchronization is achieved first by organizing the file structure into units that can be synchronized, then by using an underlying facility to control concurrency. The VMS lock-management services provide an ideal synchronization facility for VAXcluster systems.[2] The VMS file structure readily decomposes into manageable units. In fact, all units are files. Naturally, a file itself is a

file. A directory is a file. Even the volumewide management structures (e.g., the quota file and the storage bitmap) are files. Thus the file is the natural unit of synchronization for most aspects of file operations.[5]

Each file has a 48-bit file ID that uniquely identifies the file within a volume or volume set. Removing the sequence number from the file ID leaves a 32-bit integer that uniquely identifies the file at any instant of time. This integer, the file number, forms the resource name that synchronizes operations on the file. A file consists of its contents plus a file header, both of which are synchronized by a single lock. Not all locks are based on individual files. For example, for convenience and efficiency, a single-volume synchronization lock controls the allocation and deallocation of all free space and file headers.

Armed with this introduction, we can now examine in detail how each lock is used to synchronize the operations of the file system.

Device Lock

The device lock manages the states of devices accessed by the cluster. The resource name of the lock is derived from the device name, prefixed with the text string SYS$. The following lock modes represent the device state:

Lock Mode	Device State
(No lock)	Idle
CR	Volume has channels assigned and/or is mounted for shared access
PW	Mount in progress
EX	Volume allocated or mounted privately

These lock modes provide the same device arbitration that is available on single-CPU VMS systems. The value block of the device lock contains additional details about the device state (device ownership and protection, whether mounted or not, whether mounted on a foreign system or not, etc.).

Mount Lock

The device arbitration semantics in the VMS system dictate that the device lock may not be waited upon; any attempt at a conflicting access to a device yields a lock error. Therefore, an additional mount lock will serialize concurrent attempts to mount the same device. The resource name of the mount lock is again derived from the device name, prefixed with the text string MOU$. The mount lock is held in EX mode while a user mounts a device, thus allowing others in the cluster to queue behind the current mount operation.

Volume Synchronization Lock

Mounting a volume creates the volume synchronization lock in CR mode. This lock represents the mounted volume and associates one for one with the device on which the volume is mounted. The lock's resource name is derived for shareable volumes from the volume label, prefixed with the text string F11B$v. This derivation guarantees that all shareable volumes mounted in the cluster will have unique volume labels. Nonshareable volumes use the system address of the unit control block (UCB, the VMS data structure representing the device) as the volume lock name, thus allowing volumes with duplicate names to be mounted. The value block of the volume lock contains additional flags to describe the state of the volume as well as the allocation and buffer-management states.

Both the device lock and the volume lock must be held by a cluster member for the total length of time a volume is mounted. This period will usually exceed the lifetime of any process in the system. Therefore, normal locks, which are associated with an owner process, cannot be used. Instead, the file system uses system-owned locks, which are held by the system as a whole, not by any particular process. As a result, they survive the life of any and all processes in the system. These locks are released only when explicitly commanded by the system software or when the system leaves the cluster (e.g., it crashes).

The volume synchronization lock also synchronizes the allocation and deallocation of all space on the volume. When the XQP wishes to allocate space (e.g., to create a file), it takes a separate copy of the volume lock in PW mode. (Note that PW mode is compatible with the CR-mode lock representing the mount, but incompatible with itself. That ensures that only one process will attempt to allocate or deallocate space at the same time.) This form of the volume lock is held as a process lock, but only for short periods of time (the duration of a single file function or less). Part of the value block for the volume lock controls the allocation of space and contains the current count of free blocks as well as pointers

into the space-allocation bitmaps. Upon raising the volume lock to PW mode, the XQP reads this value block and writes it back to the lock manager upon release.

File Serialization Lock

The file serialization lock synchronizes all operations that affect the state of an individual file. The resource name of the file serialization lock is simply the file number, prefixed with the text string F11B$s. The resource name is qualified by the volume name by virtue of being a sublock of the volume synchronization lock. By holding the file serialization lock at PW mode, the XQP ensures that only one operation (opening, closing, extending, deleting, etc.) is performed at a time on any one file. The serialization lock also ensures that only one operation is performed at a time on any one directory. The file serialization lock, a process lock, is held only for the duration of a single file operation.

The locks described so far are sufficient to assure the integrity of the file structure in the face of concurrent operations. However, two additional locks are required to support the synchronization semantics that the file system provides to its users.

Arbitration Lock

The file system provides access arbitration for files; that is, users may open files for read or write operations and can specify whether other users may open the file concurrently. An arbitration lock is used to arbitrate file access across a cluster. The resource name of the arbitration lock is the file number, prefixed by the text string F11B$a and the volume lock name (the resource name of the volume lock). The arbitration lock is held as a system-owned lock in any of the available lock modes, depending on the state of access of the file. These states of access are

- NL – No-lock file access
- CR – Open for read, allowing other reads/writes
- CW – Open for read/write, allowing other reads/writes
- PR – Open for read, allowing other readers
- PW – Open for read/write, allowing other readers
- EX – Open for exclusive access

Since the arbitration lock is held for the entire time that a file is open, its use is optimized. One system-owned lock represents the state of all accesses to the file on each cluster node. The lock mode represents the "highest" mode of access to the file on that cluster member.

Blocking Lock

Certain maintenance operations on the file structure require it to be held stable for a period of time. For example, the ANALYZE/DISK utility will lock out all file operations during a disk-rebuild operation by using privileged file functions to lock the volume. To implement the locking function clusterwide requires another volume-specific lock, the blocking lock. The resource name of the blocking lock is the volume lock name, prefixed by the text string F11B$b.

Since performance degrades if the lock manager checks on the blocking lock as each file function starts, this lock is managed in an optimized fashion. Under normal conditions, each cluster member holds the blocking lock as a system-owned lock in CR mode. This state is noted in the volume control block (VCB). Thus the start of every file function requires only a local state check. When a lock-volume function executes, it attempts to raise the blocking lock to EX mode. Since the EX lock is incompatible with the CR locks, a system-blocking AST routine will be executed on each cluster member holding the lock at the CR mode. This AST routine executes as a subroutine called at IPL 8 using the JSB subroutine call instruction. The routine acquires process context by "borrowing" the swapper process. A kernel AST is then queued to the swapper, causing another routine to execute in the swapper's process context. This other routine releases the CR-mode blocking lock and updates the VCB context accordingly. When all the CR-mode locks have been released, the EX lock will be granted and the lock-volume function completes.

The volume will remain locked because the blocking-lock check at the start of every file function will now fail. When that happens, the XQP will attempt to reacquire the blocking lock. This attempt causes the process to stall because the blocking lock is still held elsewhere in EX mode. When an unlock-volume function finally releases the blocking lock, all processes waiting for the lock will also be released and the CR mode lock is re-established. Normal file operations can then proceed.

Deadlock Prevention and Locking Order

The execution of a single file function can involve taking out several locks. Holding more than one lock at a time always presents the potential for deadlock. The XQP avoids deadlocks, however, by taking out locks in a consistent order, as follows:

1. Blocking lock
2. Directory serialization lock
3. File serialization lock
4. Volume lock
5. Other special locks

Note that the ordering of the directory and file locks assumes a truly hierarchical directory structure. The VMS file structure allows the creation of arbitrary links; thus directory links can point "upward" in the directory hierarchy. Any attempt to traverse an upward link while another process is traversing the corresponding downward link can result in a deadlock error. The VMS system views such deadlocks as an exceptional circumstance and returns them to the caller.

Caching

The file structure of the VMS file structure is complex.[3] Typical file operations require the examination or modification of several separate components of the file structure. To achieve acceptable performance, the VMS file system has always maintained extensive caches of components of the file structure. These caches include the following:

- A general-purpose block-buffer cache holds recently read disk blocks containing file structure components.
- A file control block (FCB) list describes the attributes and states of all open files and recently referenced directories.
- An extent cache holds a portion of the disk's free space for fast allocation and deallocation. Space held in the extent cache is marked "in use" in the disk's storage bitmap (the primary structure that controls space allocation) to ensure safety if the system crashes. Should the system crash, the space in the extent cache will be temporarily lost. Because this space has been marked "in use," there is no possibility of space that was allocated to files before the crash being again allocated to other files after the crash. Lost space is usually recovered with a disk rebuild operation after the volume is mounted.
- A file-ID cache holds a set of free file numbers for fast allocation and deallocation of file headers. Similar to those in the extent cache, file numbers held in this cache are marked "in use" in the disk's file-number bitmap.
- When quota management is in effect, a quota cache holds quota records for currently active users.

Together, these caches absorb over 75 percent of the disk I/O that the file system would otherwise incur in performing file management functions.

Implementing these caches in the single-system ACP context was relatively straightforward. The block-buffer cache was located in the ACP's process context; the remaining caches occupied small portions of the system nonpaged pool.

The advent of clusters and the XQP introduced the traditional problems of maintaining cache coherency in a distributed environment. These problems were solved by using traditional cache-consistency techniques and both traditional and nontraditional application of the VMS lock manager. Many of the synchronization locks described so far also play a second role in managing the caches.

To put the block-buffer cache into a shared context, we moved this cache from the ACP process context to the system paged pool. The other caches remained in their existing locations. Since each CPU in a cluster has its own set of caches, all were synchronized with locks using a combination of sequence counters and blocking ASTs.

Because major changes were involved, we took the opportunity to examine some of the design decisions made in VMS version 1. Based on this examination, we made some alterations to reflect the changes in scale that have taken place in the VMS software since its initial release. For example, the original block-buffer cache had used linear searching on its descriptor tables. The new block-buffer cache uses descriptors based on a hash table to allow faster access to a large cache.

Previous versions of the VMS system used a simple directory-index mechanism built into the directory's file control block. In effect, this mechanism kept a small table of contents that allowed faster access to the entries of a directory

file. In the XQP conversion, this index was moved into the block-buffer cache to increase the space available to each directory index, thus improving its effectiveness.

Block Buffer Cache

The block-buffer cache consists of a collection of 512-byte buffers for disk blocks, plus the necessary collection of descriptors and hash tables. Cache coherency is maintained using the traditional lock and sequence-number technique.

Every file structure block processed by the XQP is governed by some synchronization lock. The value block of the lock contains a sequence number representing the last update to blocks governed by that lock. Upon reading a block, the file system associates the current sequence number with the copy of the block held in the cache. Upon modifying a block, the file system increments the sequence number and, at the end of the file operation, releases the lock with the updated sequence number. The corresponding locks are not fully released if any data blocks remain in the cache. Instead, the locks are demoted to NL mode to preserve the continuity of the value block.

If another system's XQP subsequently references this file structure block and finds an old copy of it in its own block-buffer cache, that system will find that the sequence numbers in the cache descriptor and in the value block of the lock do not match. This mismatch indicates that the block has been modified, and that the cache contents are invalid and must be refreshed from the disk.

We observed earlier that the volume synchronization lock and the file serialization lock are the only ones strictly necessary to ensure the integrity of the file structure. Consequently, all file structure data is read and written under these two classes of locks, which govern cache coherency. Blocks related to space allocation on the volume, such as the storage and file-number bitmaps, are processed under the volume lock. All other blocks, such as file headers and directory contents, are processed under the file serialization lock of the file to which they belong. The file serialization lock carries two sequence numbers to discriminate between updates to file data (e.g., directory contents) and updates to file headers (e.g., the directory file header).

Detailed Cache Organization

The buffers of the cache are partitioned into four buffer pools. These pools contain

- File headers and file-number bitmap blocks
- Storage bitmap blocks
- Directory, quota file, and miscellaneous data blocks
- Directory index blocks

This partitioning is needed because one or two buffers of each type may have to be available concurrently. For example, creating a file might

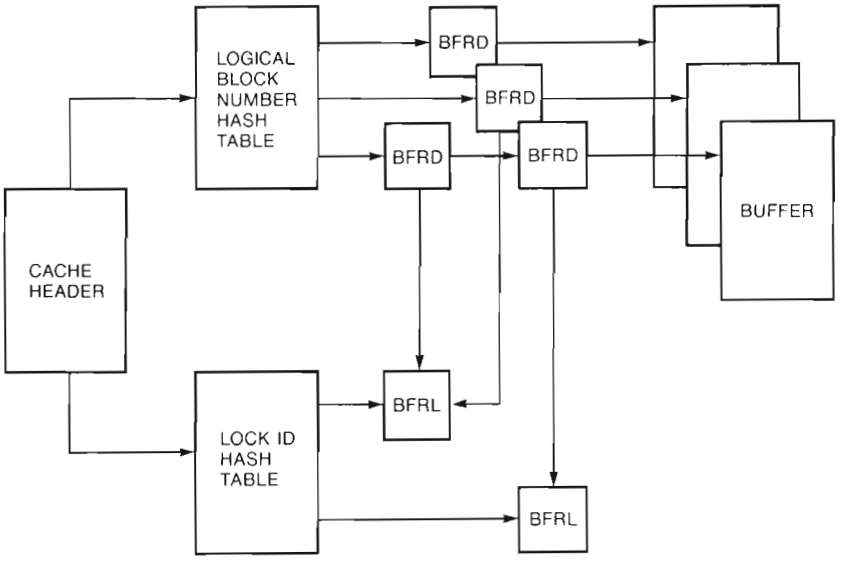


Figure 3 Buffer Cache Structure

require concurrent access to the file header, the storage bitmap for space allocation, and the directory to create the directory entry. Each buffer pool is managed using a variant of least recently used (LRU) replacement. Consequently, the buffer manager can guarantee concurrent access to one or two buffers of each type without any explicit buffer lock and release mechanism. (Certain file and directory operations require concurrent access to two file headers or two directory blocks.) The structure of the buffer cache is shown in Figure 3.

Each buffer has a buffer descriptor (BFRD), which contains the information needed to identify and manage the current buffer contents, as shown in Figure 4. The BFRD contains the following information:

- An logical block number (LBN) and a unit control block (UCB) to identify the disk address and the volume of the block contained in the buffer
- The lock basis (i.e., the root of the resource name for the lock governing the buffer)

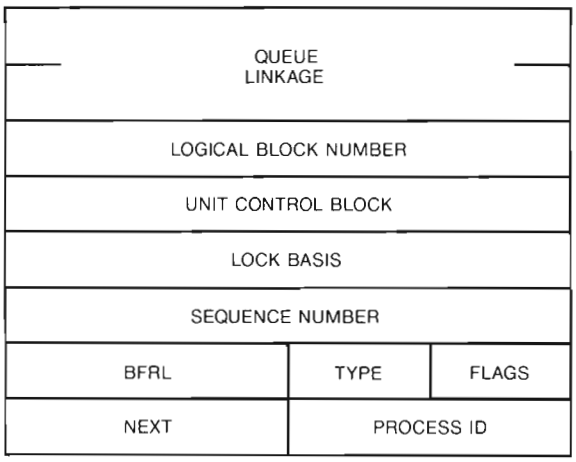


Figure 4 Buffer Descriptor Block

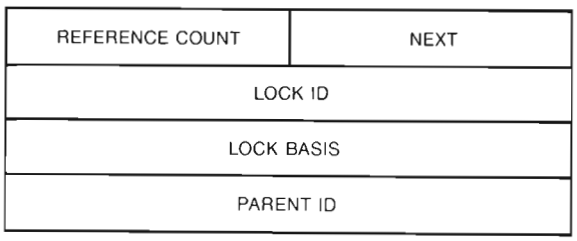


Figure 5 Buffer Lock Block

- The buffer sequence number from the value block of the lock
- A pointer to the lock block of the buffer
- Flags, including valid and modified
- A process ID of the buffer's owner
- Queue pointers for state queue linkage
- A hash-chain link pointer

In addition, a buffer lock block (BFRL), shown in Figure 5, is associated with each buffer, several of which may be processed under the same lock. Thus the BFRL identifies the lock under which some set of buffers is managed and contains the following information:

- The lock ID of the lock
- The lock ID of the parent lock
- The lock basis
- A reference count
- A hash-chain link pointer

Buffers and locks are found using two hash tables, one each for BFRDs and BFRLs. The disk block LBN is used to hash into the BFRD hash table; the lock basis is used to hash into the BFRL hash table. Each entry in the table forms the head of the hash chain for a set of BFRDs or BFRLs.

The cache header ties together the components in the block-buffer cache. The cache header contains

- Base pointers for the hash tables
- The BFRD and BFRL lists
- Availability counts and descriptors to form the four partitions of the buffer cache
- Performance counters
- Several synchronization queues

Each synchronization queue is described as follows:

- Cache synchronization queue – Changes to the cache descriptors (e.g., signing a buffer out of the cache for process use or changing the contents of a buffer) must be serialized.
- Pool wait queues – If insufficient buffers are left in the buffer pools, the XQP must wait before processing a file function.

- Ambiguity queue – The lock name used to synchronize a file header sometimes changes. For example, all headers of a multiheader file are synchronized under the serialization lock of the primary file header. Therefore, the lock name for an extension header will change when the file is deleted and the header reused for another file. The ambiguity queue is used when the VMS software finds that a file header buffer is owned by another process under a different lock. Thus the queue allows the currently executing XQP to wait until the state of the header buffer stabilizes.

Since each host CPU has a buffer cache, access to it is not synchronized by the lock manager. Rather, an informal queuing mechanism, which saves considerable overhead, is used. When an XQP must wait on one of the buffer header queues, it simply sends the I/O packet representing the current file operation into the appropriate queue and suspends execution. Some time later, another process in the system will rectify whatever condition the first process was waiting for (e.g., making buffers available). Having done so, the other process checks the appropriate queue to detect that the first process is waiting. The first process is then restarted by removing its I/O packet from the synchronization queue and using the I/O packet to queue an AST.

Buffer Management

In the block-buffer cache, each buffer is in one of two states: either it is available for use (and may or may not contain valid disk data), or it is owned by a process (and only one process). The cache is carefully managed to avoid resource deadlocks and to prevent individual processes from "hogging" it.

A resource deadlock happens when a process partially executes a file function, then discovers the need for an additional I/O buffer. Being partially complete, the process probably holds some locks. If no more buffers were available, the process would have to wait, holding its locks. In the meantime, some other process, also holding some I/O buffers, might attempt to acquire a lock that the first process is holding. In this case, that other process will stall. This situation is the classic deadlock of "A is holding X and waiting for Y, B is holding Y and waiting for X." Yet the VMS lock manager would not detect this deadlock because some of the entities involved are not locks.

Resource deadlocks are avoided by reserving sufficient buffers before starting a file function. Thus the file system is designed so that all file functions can be completed using a known minimum number of buffers. If this minimum number is not available, the XQP must wait on the pool wait queue. Therefore, deadlocks cannot occur because the XQP is not yet holding any locks.

Buffers are reserved by simply decrementing the pool availability counters in the cache header. Individual buffers are not actually taken by the process until needed. The state queue linkage and the owner process ID (PID) represent the state of a buffer. An available buffer is linked into the LRU list corresponding to the buffer pool; this buffer has a zero-owner PID. A process takes a buffer when the process wishes to read a particular disk block. The process selects an appropriate buffer either by finding the desired disk block in the LBN hash table, or, if the block is not found, by removing the oldest buffer from the front of the LRU list. Taking a buffer for process use involves first removing it from the LRU list and entering it into the process's in-process list, then entering the process ID into the buffer's owner PID field.

A buffer is never taken if marked with a different owner PID (i.e., owned by another process). If the buffer is for a file header, the lock basis for the header could be changing; therefore, the XQP must wait on the ambiguity queue. The lock basis for other types of buffers never changes while the buffer is owned. Therefore, finding a buffer owned by another process indicates that file synchronization has been violated, which causes a system crash.

In many cases, more buffers than the necessary minimum may be useful in processing a file function (e.g., when a file has many headers or a large directory must be searched). If more buffers are available in the cache, the XQP will continue to reserve and take them for process use. Once the cache availability counters fall below a minimum threshold, however, the XQP will stop reserving additional buffers. In this case, the XQP must return a buffer from its in-process list for each new buffer taken. This swap prevents one very complex file operation from hogging all available buffers and guarantees a minimum level of operational concurrency.

At the end of a file operation, all buffers held on the in-process list must be returned to the cache. Since modified buffers are not held in the

cache, any on the in-process list are written back to the disk as they are returned. As the buffers are returned, the XQP ensures that each one is associated with a BFRL corresponding to the synchronization lock under which the buffer was read. The XQP will release all synchronization locks when all buffers have returned. Locks corresponding to buffers remaining in the cache are not released but are demoted to NL mode to preserve the buffer sequence number.

The inability to hold modified buffers in the general cache is a small regression from the ACP-based file system. VMS versions 2 and 3 could hold modified file headers of files currently open for write in the cache. That ability saved a write operation when such a file was modified (e.g., extended). Now, the technique of holding modified buffers and flushing them under a blocking AST is well understood. It is possible to add the necessary mechanism to the new buffer manager. However, development time constraints prevented us from including this capability in VMS version 4.

User Interference

The file system is designed to tolerate the modification of the file structure components by user-level software (such as the disk-rebuild utility). Therefore, when a user process opens the storage bitmap file for a write operation, for example, any updates to that file must be accounted for in the block-buffer cache. This task is accomplished by first recognizing files that constitute components of the file structure when they are opened for write, then routing all writes through the XQP. The XQP checks all blocks written against the cache and invalidates matching cache buffers.

File Control Blocks

Like the block-buffer cache, the file control blocks for open files and directories represent replicated cache data that must be kept coherent. The blocking AST mechanism in the lock manager solves this coherency problem. Recall that each cluster member holds an arbitration lock for each open file on a cluster-accessible volume. Associated with the arbitration lock is a system blocking AST routine. File access arbitration never invokes this routine because arbitration does not wait for file accessibility. (File access conflicts are returned as errors to the caller.)

When a user modifies the attributes of a file (its size, protection, etc.), the various file control blocks across the cluster must be updated. This task is done by queuing an EX-lock request for the arbitration lock, thus causing the blocking AST routine to execute. The AST routine simply marks the local file control block "stale." Once queued, the EX-lock request will be immediately canceled since it will normally never be granted. On the other cluster nodes, the next operation on the file will update the file control block. The XQP, finding the file control block marked stale, will refresh it with file data read from the disk and rearm the blocking AST by re-establishing the arbitration lock.

Quota Cache

The quota cache presents a unique cluster-synchronization problem. The quota cache contains a small number of currently active quota records, each representing a file owner to whom file space has been charged. Now, users normally modify files owned only by themselves. Therefore, a small cross section of the quota file, representing the set of users currently logged into the system, can be cached with excellent locality. The quota cache is especially effective because quota changes are reflected only in the cache entries. These changes are written back to the quota file only when replacement removes them from the cache. As a result, a properly sized cache eliminates almost all the overhead of quota management. Figure 6 illustrates the access to the quota cache, and Figure 7 the entry to that cache.

Preserving the performance characteristics of the quota cache presented us with a unique problem. The locality of use of file owners does not in any way reflect back into locality of use of quota file blocks. Thus the cache entries must be handled on an individual basis. Quota-cache coherency across the cluster is maintained by using a separate lock for each quota-cache entry. The dynamic part of a quota record (quota, overdraft, and usage, plus some flags) just fits into the 16-byte value block of the lock. The resource name of the lock is the file owner, plus the volume name and the text string F11B$q.

A lock held at PW mode backs up each valid entry to the quota cache. When another XQP in the cluster wishes to use the same quota record, that XQP must find a suitable cache entry (by finding the file owner in its cache or taking the LRU cache entry) and then enqueue for the lock at PW mode. This action triggers a blocking AST

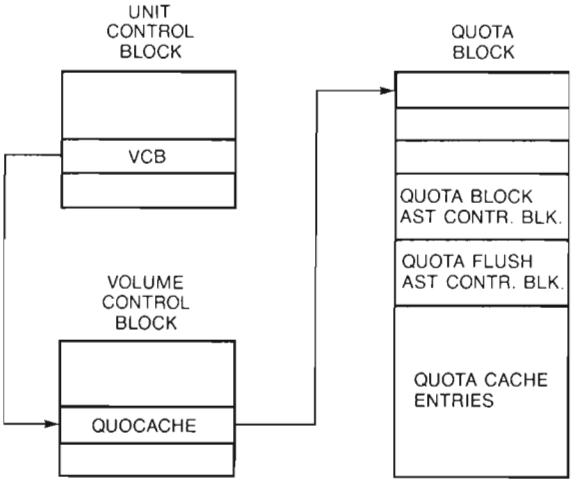


Figure 6 Quota Cache

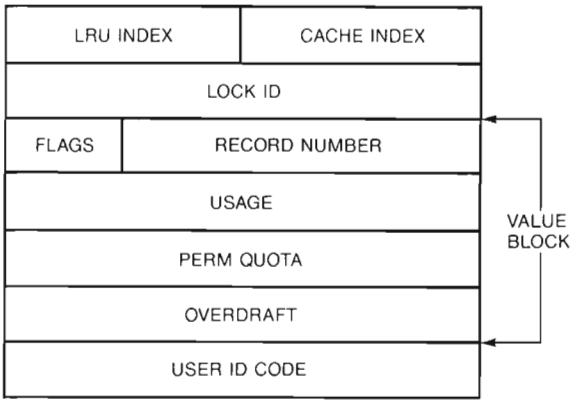


Figure 7 Quota Cache Entry

on the node currently holding the lock at PW mode. Because the quota-cache lock is system owned, the blocking AST routine will execute at IPL 8. Using an AST control block built into the quota-cache structure, the routine queues an AST to the swapper process to borrow its process context. The swapper AST executes another subroutine that releases control of the entry to the quota cache. This subroutine marks this entry "invalid" and demotes the PW lock to CR mode, in the process writing the entry contents into the value block of the lock. Upon release, the lock is granted to the requesting process, which transfers the lock's value block into its cache entry. As a result, the lock manager can transfer quota-cache entries about the cluster without incurring any disk I/O.

File Number and Extent Caches

During normal operation, the file-number and extent caches, shown in Figure 8, do not present any synchronization or coherency problems in the cluster. Since the cache contents are marked "in use" in the appropriate bitmap, each cache in each cluster member simply contains a different collection of free disk space or free file numbers.

The cache may have to be emptied, however, and its contents written back to the bitmap. There are two reasons for these actions. First, the file system will tolerate the modification of the file structure components by user-level software (e.g., the disk-rebuild utility). Therefore, when a user process opens the storage bitmap file for a write, for example, all instances of the extent cache must be flushed to the bitmap. That does two things:

1. It presents the user with a correct view of the bitmap.
2. It prevents the cache from containing stale data in the event the user modifies the bitmap.

Note, by the way, that the quota cache is affected by all these considerations as well.

Second, resource exhaustion must be handled as gracefully as possible. With the extent caches in operation, the available free space on the disk is distributed in the various extent caches across the cluster. If a user makes an allocation request for all the remaining free space on the disk, that

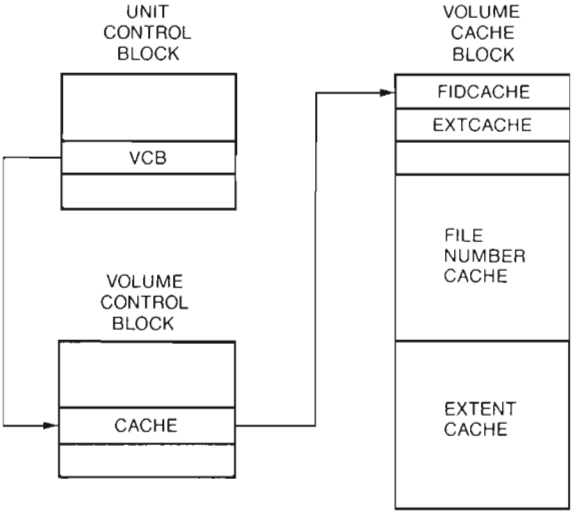


Figure 8 File Number and Extent Caches

request cannot be satisfied without emptying the extent caches on the other cluster members.

A cache-flush lock will handle both situations stated above. The quota, file-number, and extent caches are each backed by a cache-flush lock. The resource name is derived from the file number of the related file, plus the text string F11B$c. While a cache is active, the cache-flush lock is held as a system-owned lock at PR mode.

When wishing to cause a cache flush for a certain type of cache across the cluster, the XQP enqueues for the related lock at CW mode. This action causes the blocking AST associated with the PR lock to execute as a fork IPL 8 routine. This routine uses an AST control block built into the cache structure to queue an AST to the CACHE_SERVER process of the file system. One such process runs on each node in a cluster; its sole responsibility is to respond to cache-flush requests.

The parameters associated with the AST identify which cache is involved and the volume for which the cache is to be flushed. The CACHE_SERVER process then executes a privileged file system control function that causes the file system to empty the specified cache. Having emptied the cache, the XQP releases the PR lock, thus allowing the process requesting the CW lock to proceed. If a cache flush is requested simply to make all free space available, the CW lock will be immediately released. If the cache flush is associated with opening a piece of the file structure for a write, however, the CW lock will be held as a system-owned lock until the file is closed. Since any attempt to refill the cache must first acquire the PR lock, such attempts will fail until the file is closed and the CW lock released.

Summary

The distributed file system was one of the most challenging aspects in developing VAXcluster systems. Starting from a file system that was process based and single threaded, we developed one that is procedure based and multithreaded. The major challenges lay in developing the necessary synchronization and in redesigning the caches to work correctly in the distributed environment. We solved these problems by extensively employing the VMS distributed-lock manager in new and creative ways. The result is a file system that works effectively in the cluster environment. What's more, this file system displays better performance and concurrency in the single-system environment as well.

References

1. *VAX/VMS I/O User's Reference Manual, Part 1* (Maynard: Digital Equipment Corporation, Order No. AA-Z600C-TE, 1986).

2. *VAX/VMS Systems Services Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-Z501B-TE, 1986).

3. *Guide to VAX/VMS Disk and Magnetic Tape Operations* (Maynard: Digital Equipment Corporation, Order No. AI-Y506B-TE, 1986).

Michael S. Fox
John A. Ywoskus

Local Area VAXcluster Systems

Local Area VAXcluster systems use the Ethernet rather than the CI bus as their interconnect between nodes. This makes it possible to include MicroVAX systems and workstations in a VAXcluster environment. The key technical issues that had to be solved were to provide an Ethernet base equivalent to the CI bus for the cluster's System Communication Architecture protocols and to allow the VMS software to boot on a diskless system using the Ethernet as a link to a remote system disk. This paper describes the work done to satisfy these two design issues: providing robust cluster communication on the Ethernet as a means of performing remote disk access, and network booting of the VMS system.

The Local Area VAXcluster (LAVc) software is a new product that brings VAXcluster functionality to the full range of VAX processors. A LAVc uses the Ethernet instead of Digital's proprietary Computer Interconnect called the CI bus, thus making possible the inclusion of small systems like the MicroVAX II CPU in the VAXcluster configuration. This paper describes the benefits provided by a LAVc, the concepts on which it was built, and the technical details of the two new major internal capabilities added to the VMS operating system.

VAXcluster System Definition

A VAXcluster system is a distributed system made up of VAX computers and their associated storage elements, all linked in a closely coupled arrangement.[1] VAXcluster members cooperate with each other on a peer-to-peer basis. They all share a common file system, print and batch queue operations, and comprise a single management domain (the cluster is managed as a single-system entity) enclosed by a single security perimeter.

A VAXcluster system differs from a more tightly coupled multiprocessor arrangement in several ways. First, the VAX systems communicate over a fast, efficient network link instead of sharing memory. Second, each system has its own copy of the VMS system in memory (possibly loaded from the same shared disk image). Third, the members may boot and shut down independently. Finally, the clusterwide file system, single security and management domains, and other VAXcluster features are much closer to those offered by a traditional single timesharing system than to the capabilities offered by traditional networks.

The first VAXcluster implementation (VMS version 4.0) operated only on the CI bus, a limited-distance LAN connecting up to sixteen nodes at 70 megabits per second. CI adapters are highly intelligent, and hence relatively complex and expensive. They were built expressly for large systems located in machine rooms. With the advent of small desktop VAX processors, some new interconnect was needed for bringing them the benefits of cluster functionality. The CI bus could meet neither the geographical criteria nor the low cost required in an office (as opposed to a computer room) environment, nor could it support enough nodes.

The VAXcluster support in VMS version 4.4 had matured enough so that extending it to another interconnect became feasible. The Ethernet, already Digital's standard for network communication, was the obvious choice for this new interconnect. Ethernet's cost, distance, speed, connection capabilities, and existing hardware base allowed the VAXcluster functions to move out of the machine room and effectively support smaller systems.

LAVc Goals, Requirements, and Configurations

The overall LAVc goal was to bring the benefits of VAXcluster systems to low-end and desktop

systems. The benefits of this goal included the following:

- A single, clusterwide common file system with disks connected to any CPU
- Fully integrated and synchronized file sharing at the record level among users on any member in the cluster
- Clusterwide availability of print and batch queues (Print and batch execution facilities can be located on any set of members.)
- A single security domain
- The simplification (or even elimination) of the end user's system-management responsibilities

With this goal in mind, we drew up a list of requirements for such a product. These requirements included

- Support the Ethernet instead of the CI bus as a cluster interconnect, yet allow simultaneous use by other clusters and networks
- Boot the VMS software over the Ethernet
- Simplify cluster management and installation by providing tools and limiting configurations
- Provide clusterwide disk access by means of the software Mass Storage Control Protocol (MSCP) server instead of the HSC controllers
- Retain all the existing VAXcluster software capabilities and as much of the implementation as possible
- Support diskless systems

The first three requirements had the largest impact on the LAVc development. In fact, the first two required the most engineering effort to develop new software. After a brief description of the resulting LAVc product, the remainder of this paper will describe the technical work done to meet those two requirements.

Configurations

The configuration supported by the initial releases of LAVc utilizes a single Ethernet as the cluster interconnect. Conservative restrictions were imposed where necessary to limit the complexity and to allow thorough testing and performance analysis of almost all supported cluster configurations. The result is the configuration shown in Figure 1. Future extensions to increase the number of members, allow both CI and Ethernet in the same cluster, and multiple Ethernets are being planned. They will not be addressed further in this paper.

The members cooperate with each other in a peer-to-peer relationship. They are managed by a cluster connection manager and synchronized by a distributed lock manager without regard for the roles they play in an operating LAVc.[2] That is a key difference between the LAVc and other "client/server" products. Any system in the cluster can provide or consume resources provided by the other systems. To simplify the resulting supported configurations, however, we chose to assign certain roles to the systems. The boot member and satellite roles merely describe the jobs those systems perform; the roles are not known by the VAXcluster software. The cluster software cares only where the resources are located and which systems have access to them.

Each boot member is a management center of the cluster. The VMS system disks connected to each boot member makes them available to other cluster members by means of the MSCP server software. The initial LAVc releases limit the number of boot members and system disks to reduce the complexity of installation and management.

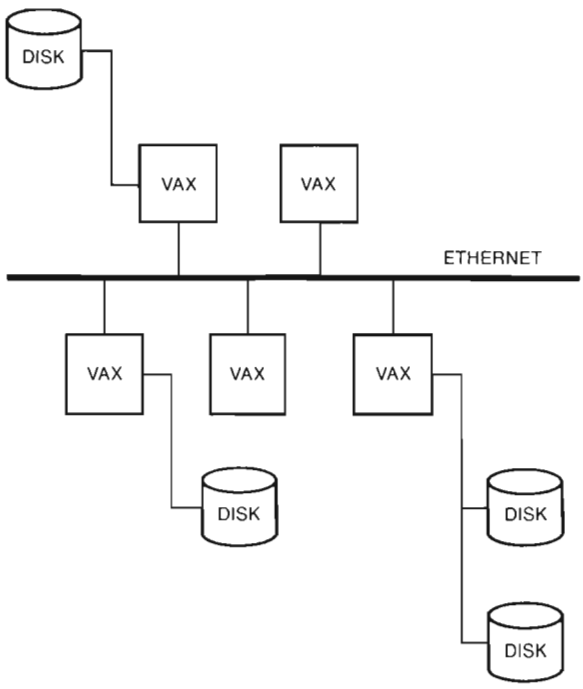


Figure 1 LAVc Configuration

Boot members may also serve other data disks in the cluster.

A boot member also functions as a load host during an Ethernet boot operation. This role is discussed further in the sections on remote booting.

Satellite systems boot off the system disk provided by a boot member and generally depend on that member for other resources as well (data disks, printers, etc.). On the other hand, satellites may serve data disks to the cluster, as well as provide print or batch resources. The satellites are configured by the cluster manager to best meet the needs of the application.

To date, only members of the MicroVAX II family of systems and workstations (MicroVAX II, VAXstation II, VAXstation II/GPX, MicroVAX 2000, and VAXstation 2000 systems) can be satellites. This restriction results from the need for specific code to be written to support remote booting for the CPU and Ethernet adapters. Satellite support for other CPUs (both new and existing) will be considered in the future.

Disk Access

In a CI cluster, the HSC disk controllers connect to the CI bus in the same manner as do the VAX systems. I/O requests originating in any VAX CPU are passed to the disk class driver (DUDRIVER), which encodes them into MSCP packets. These packets are sent over the CI network to the appropriate HSC controller for execution. All VAX CPUs in the cluster therefore have equal access to the HSC controllers and the disks connected to them. However, an HSC controller cannot connect to an Ethernet. Therefore, some other method is needed in a LAVc to allow disk access to all systems.

In the absence of HSC controllers, each disk must be connected to the system by some controller, such as a UDA, KDA, or UNIBUS controller. Making these disks accessible to other VAX systems in the cluster requires a software emulation of the HSC controller. This need is filled by the MSCP server software.

The VAX CPU originating the I/O request merely sends an MSCP packet over the network to the target VAX CPU with the desired disk. The packet is identical to the one DUDRIVER would have sent to an HSC controller. The MSCP server software on that target CPU receives the packet, performs the operation, and returns the results just as an HSC would do. The class driver on the originating VAX cannot tell the difference between the MSCP server and an HSC controller. The result, as shown in Figure 2, is that disks served by the MSCP server appear to be equally available to all systems in the cluster, independent of which system they are actually cabled to and the type of interconnect.

System Management

The LAVc configurations described above were designed so that all system management activities would take place on the boot member. Although the cluster can be configured differently, that configuration is the simplest. It is also what most users would want when the satellites are personal workstations.

The VMS, satellite system, and application software installations are all controlled by command procedures executed on the boot member. Disk backups are done mostly on the boot member, on which the backup device (usually tape) is located. Data disks can be located anywhere in the cluster. If the satellite is a single-user workstation, we recommend that applications and user data not be put on any of its disks. Using a workstation's local disks only for page and swap files eliminates the need for backups, thus freeing the owner of all system-management responsibilities.

The overall product simplicity goal is clearly facilitated by configuring the cluster in this manner. All management activity is local to one system and remains under the control of a limited number of people. Cluster users should have no

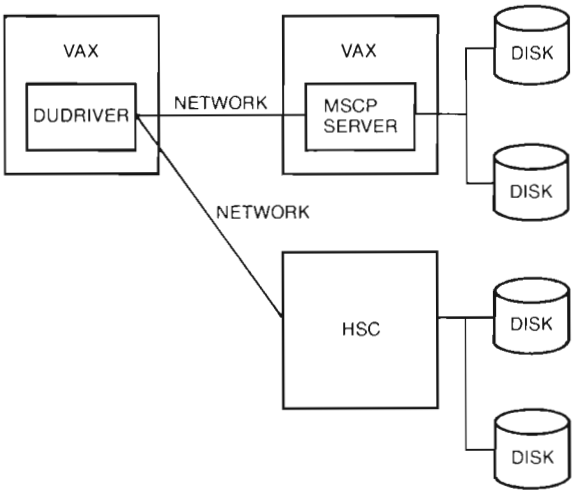


Figure 2 Disk Access

more system-management responsibilities than users of dumb (e.g., VT220) terminals would have.

LAVc's Use of the Ethernet

The Ethernet is used as the cluster communication mechanism because it is compatible with the LAVc's requirements for cost and system environment (non-computer room). There are, however, significant tradeoffs inherent in substituting the Ethernet for the CI bus. For example, communication over the Ethernet is slower and more CPU intensive than over the CI bus. The Ethernet's advantages are lower expense, much greater geographic distance, and the ability to connect many more systems.

The VMS port driver that provides reliable cluster communication utilizing the Ethernet is called PEDRIVER. It provides communication in such a way that the rest of the VMS software is unaffected. This section describes PEDRIVER's role within a LAVc, the PEDRIVER protocol, and some technical details about its internal structure.

The PEDRIVER

Communication services within a VAXcluster system are described by the System Communication Architecture, or SCA.[3] The SCA model consists of the four layers shown in Figure 3.

The system application (SYSAP) layer consists of users of the connection services provided by the systems communication services (SCS) layer. Examples of SYSAPs are the disk class driver (DUDRIVER), the MSCP server, and the cluster connection manager.

The SCS layer provides network resources to the SYSAPs. It multiplexes the underlying communication service, provided by the port-to-port communication layer, into several connections. These connections link a number of entities, including the connection managers between two members, the class driver to the MSCP server (or HSC device), and so forth. The SCS layer also provides flow control, buffer management, notification of new SYSAPs registering with it, and notification of connection breakage.

SYSAP	SYSTEM APPLICATIONS LAYER
SCS	SYSTEMS COMMUNICATION SERVICES LAYER
PPD	PORT-TO-PORT COMMUNICATIONS LAYER
PI	PHYSICAL INTERCONNECT LAYER

Figure 3 SCA Layers

The port-to-port communications (PPD) layer maintains a single communications path, called a virtual circuit, with every other VAX system or HSC controller in the cluster. On a CI cluster, this layer is the lowest software layer within the VMS system. It is implemented by the CI port driver, called PADRIVER. PADRIVER knows how to interface with the CI adapter and is responsible for discovering new nodes, forming virtual circuits with them, detecting communication failures, and signaling these events to the SCS layer.

In a LAVc, PEDRIVER provides much of the same PPD functionality as does PADRIVER. Since the Ethernet hardware offers only a datagram service (instead of the reliable communication path offered by the CI bus), PEDRIVER uses a networking protocol to provide a reliable communications service. Unlike PADRIVER, PEDRIVER is device independent, utilizing an underlying datalink driver to control the Ethernet adapter.

The physical interconnect (PI) layer represents the medium over which packets are sent and received. A complete specification for this layer includes the mechanisms for clocking bits on the wire, the framing of bits into bytes and bytes into messages, electrical signal requirements, cabling, and so forth.

Ports

A port is a software interface between the port driver and a communications entity, usually an adapter. A port is implemented as a set of queues whose use is rigorously defined. Access to these queues is by means of interlocked instructions; thus no other synchronization mechanisms are required. The port driver manages the port. The driver receives requests from the SCS layer, formats them, then passes them across the port by linking a packet in a prioritized command queue. The driver then sets a control bit to inform the port of this action. The entity behind the port dequeues the command packet, executes it, and either returns it to the driver with a status message or places it in the appropriate free queue. Packets being delivered across the port to the driver are linked into a response queue. An interrupt is generated if the queue was previously empty.

In the CI case, this port structure is used to communicate between PADRIVER and the CI hardware. The hardware guarantees the delivery of sequential messages. It also moves user data into or out of the virtual address space of a target node during block transfers. Thus the CPU overhead is kept to an absolute minimum. The CI adapter is intelligent enough to perform these functions on its own and to interrupt the CPU when the operation is finished.

Ethernet adapters do not fit this model. They are typically packet-oriented devices that transmit or receive using discrete, limited-size buffers. The adapters do not guarantee sequential delivery. Since VAXcluster systems require these features, they must be replaced with software, at a corresponding increase in CPU overhead.

To preserve the same port interface, however, we put the software providing these services below the port interface. The port then becomes an interface between SCS and a port driver above the port, and a port emulator below. Preserving the same level of functionality at the port interface eliminated the need for extensive software modifications to the SCS and higher software layers. Figure 4 shows the port structure for both the CI and Ethernet cases.

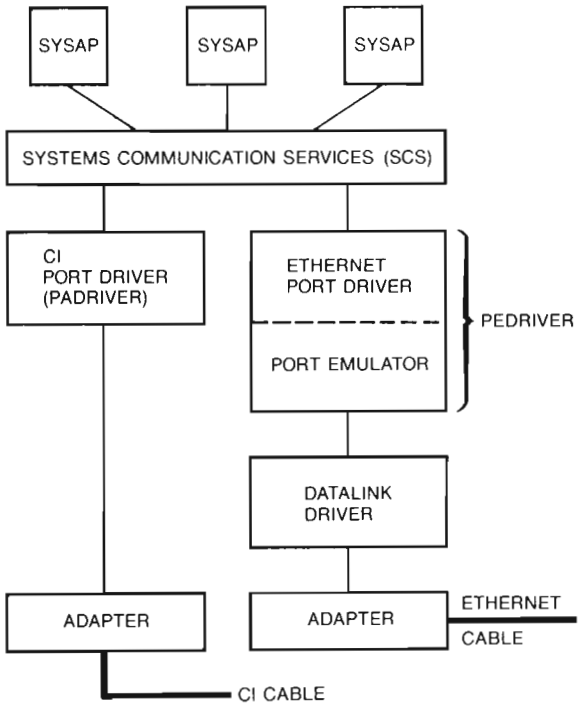


Figure 4 VAXcluster Software Structure

PEDRIVER Functions

PEDRIVER is used instead of PADRIVER as the port driver in a LAVc. PEDRIVER contains two major segments: a port manager that receives packets from SCS and queues them to the port, and a port emulator that operates below the port interface. This port emulator effectively emulates the behavior of the CI hardware, utilizing a still lower level datalink driver for access to the Ethernet adapter, as shown in Figure 5. Since the port emulator is the key to the LAVc's use of the Ethernet, its design and implementation will now be described in detail.

NI-SCA is the name of protocol used by the port emulator to communicate with its peers on other nodes. This protocol extends the SCA so that systems can be connected by the Ethernet (also known as the NI). This extension is achieved at the cost of reduced CPU efficiency, since the software is doing more work, and lower I/O bandwidth, since the Ethernet is slower than the CI bus. In addition, the public access nature of the Ethernet introduces security and configuration problems not encountered on the CI bus.

Major Objectives

The goals of the NI-SCA port design are

- Compatibility – The interface to the NI-SCA port must have a strong resemblance to that of the CI port to minimize the impact on the system software directly using the port. In particular, the functions required by the SCS layer and provided by the port should be operationally equivalent to their CI port counterparts so that the SCS layer need not be changed.
- Performance – The port architecture has to address two performance problems. First, the low Ethernet bandwidth may very well be a bottleneck in some configurations, especially

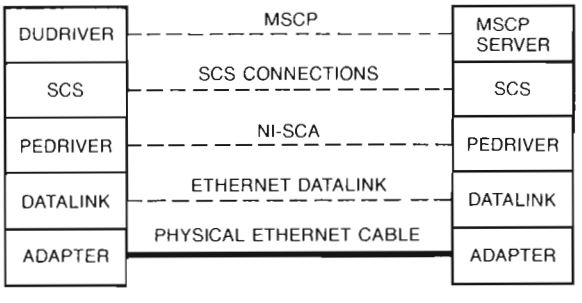


Figure 5 Protocol Layering

as CPU speeds increase. Second, the low bandwidth affects both the aggregate throughput and the response time between a transmitted message and the subsequent response.

- Security – Provisions for authenticating remote nodes are required. (Software data encryption is not currently part of the port design.)
- Simplicity – The port architecture should be defined so that implementations may substitute performance for simplicity. Ports implementing different subsets of the architecture must be able to communicate with each other.

Differences between the CI Bus and Ethernet

The NI-SCA architecture must address several areas that result from the fundamental differences between the CI and Ethernet buses and their existing adapters.

- Locating other nodes – The CI polling for the existence of other nodes does not work in the larger Ethernet environment.
- Data transport – The NI-SCA port emulator must make the data transfer limitations of the Ethernet transparent. Data segmentation and reconstruction must be handled efficiently.
- Multiple paths – Any given node may interact with more than one Ethernet through more than one Ethernet adapter. The port emulator must allow an implementation to exploit such configurations transparently to achieve the requirements of efficiency and redundancy. The current implementation of PEDRIVER does not support this.
- Detection of communication failures – The port emulator must detect node or communications failures and signal them to the SCS layer.
- Ethernet coexistence – The NI-SCA protocol must allow multiple clusters to coexist on the same Ethernet and to share that Ethernet with other network protocols.
- Security – Secure communication between nodes must be addressed since the Ethernet spans a wider and less secure environment than does the CI bus, which is typically protected by the security of the computer room.

Locating Other Nodes and Virtual Circuit Formation

The address space on the CI bus is currently implemented as a four-bit field. The resulting maximum of 16 possible addresses and the limitation of one cluster per CI bus makes polling all possible addresses to locate other nodes an attractive solution. Polling is clearly not practical on the Ethernet, however, where there are 2^{47} possible addresses, multiple clusters, and nodes totally unrelated to clusters.

PEDRIVER replaces the CI bus polling with a multicast scheme to a cluster-specific multicast address. A large block of consecutive multicast addresses have been reserved for NI-SCA. The lowest address in the block is hard coded into PEDRIVER. During installation, the user assigns a group number to the cluster. PEDRIVER adds this group number to the base address to generate that cluster's unique multicast address within NI-SCA's reserved block.

PEDRIVER enables the reception of this multicast address and transmits a HELLO multicast to it every three seconds. PEDRIVER will attempt to create a circuit upon receiving a HELLO message from a node with which it does not currently share an open virtual circuit. HELLO messages received from nodes with a currently open virtual circuit indicate that the remote node is still operational.

A standard three-message-exchange handshake is used to create a virtual circuit, as shown in Figure 6.

The START_VC and START_ACK contain information about the transmitting system, and what it believes the cluster password to be. These parameters are verified at the receiving system, which continues the handshake only if its verification is successful. Thus each system authenticates the other. After the final ACK message, the virtual circuit is open for use by both systems.

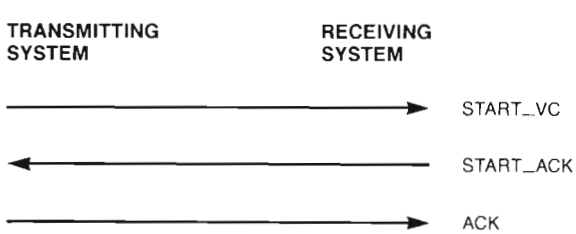


Figure 6 Standard Handshake

Data Transport

PEDRIVER uses the virtual circuit to provide the three SCA port data transfer services described below. The SCS layer does not need to distinguish between the CI hardware or the NI-SCA port emulator version of these services.

1. Datagrams – Packets to be delivered on a "best effort" basis. No guarantees are made about delivery, sequentiality, or replication.
2. Sequenced messages – The port guarantees the sequential delivery of exactly one copy of the packet.
3. Block transfers – The port moves a large amount of data in either direction. Segmentation, handled below the port, is invisible to the port driver and everything above it.

Datagrams are sent as Ethernet packets, which are sufficient since no delivery guarantees are assumed.

PEDRIVER uses a standard networking protocol to provide reliable communications when necessary. A sequence number is included in each packet so that lost or out-of-sequence packets can be detected. Each packet requiring reliable delivery must be acknowledged by the receiving port emulator. To improve efficiency, several packets may be sent without waiting for an ACK. Whenever possible, the recipient will also bundle the ACK into a message to be sent back to the original source, thus saving the cost of an explicit ACK. Timers are used in both the source and destination systems to generate a retransmission if an ACK does not arrive after a specified time period has elapsed. These timers also initiate the transmission of an explicit ACK in the absence of any reverse traffic.

To send relatively small amounts of data, SYSAPs use sequenced messages, generally holding up to about 120 bytes. PEDRIVER sends these messages with a sequence number over the virtual circuit, and they must be acknowledged by the recipient as described above. PEDRIVER can therefore guarantee reliable message delivery to the destination SYSAP.

To send large amounts of data, SYSAPs use block transfers. In a VAXcluster system, the disk class driver and the MSCP server use block transfers to move data being read from or written to a disk. PEDRIVER's port emulator implements block transfers by segmenting the data in 1300-byte chunks. Each chunk is copied out of the source buffer into a datalink packet and transmitted over the virtual circuit as a sequenced message. The receiving port emulator copies the data out of the Ethernet packet into the user's buffer. The virtual circuit guarantees the sequential delivery of these packets, thus maintaining data ordering and integrity.

The CI adapter can copy data into or out of the virtual address space of a target node by using direct memory access (DMA). Thus the CPU is not involved in block transfers. Ethernet adapters, however, access data in specific buffers; therefore, PEDRIVER must copy data using a MOVC instruction. This scheme adds a lot of CPU overhead to Ethernet block transfers.

Detection of Communication and Node Failures

Communication can be lost between nodes for several reasons: a node shutdown, a system crash, or a hardware failure. PEDRIVER must detect these events and signal their occurrences to the SCS layer.

A system generally transmits a node-stop (or last gasp) datagram upon learning it will shut down. This shutdown could be a planned event by an operator or a system software crash. The SCS layer acts upon a received node-stop datagram. SCS breaks all connections with SYSAPs on the originating system and tells PEDRIVER to break the virtual circuit. Cluster reconfiguration occurs much faster when a last-gasp datagram is received because no time-outs are required.

Communication can be lost, however, without the receipt of a node-stop datagram. Both a hardware failure and tripping a system's halt switch will break contact, or the node-stop datagram could be lost on the Ethernet. Therefore, other ways of detecting a breakage are needed. In general, PEDRIVER detects a breakage by checking for the HELLO multicasts being transmitted every three seconds. One eight-second timer checks for the arrival of HELLO messages for all virtual circuits. If two ticks of this timer (eight to sixteen seconds) occur without receiving a HELLO message from a system, that system is assumed to have failed. The SCS layer is then notified of this occurrence.

Certain hardware failures may cause a node to continue sending but to be unable to receive HELLO messages. Therefore, still another failure detection method is used: the counting of retransmission attempts for a sequenced packet.

If a sending node makes 30 attempts (at one-second intervals) without receiving an ACK, the recipient node is presumed dead and SCS notified of the failure.

Sharing the Ethernet

The Ethernet is designed as a shared-communications bus. Any NI-SCA architecture that precludes its use by other clusters or networks is unacceptable.

Multiple LAVcs coexist on the same Ethernet by using different group numbers. Thus each LAVc uses different multicast addresses to transmit and receive its HELLO messages. As a result, it does not "hear" messages from other LAVc's or attempt to form virtual circuits with them. Multicast messages on one Ethernet are not passed to other Ethernets that are linked by means of traffic routers or gateways utilizing other communications media. Therefore, group numbers must be unique only on each Ethernet. Different clusters on other Ethernets may use the same group number. The group-number space is large enough so that ranges of numbers can be given to different branches of a business organization, thereby reducing the need for networkwide administration.

NI-SCA is registered as Ethernet protocol type 60-07. This registration allows the datalink driver to distinguish NI-SCA packets from those sent by the DECnet, LAT, or other protocols. PEDRIVER's use of the Ethernet has no effect on any other protocol, regardless of how the packets are multiplexed on the single Ethernet.

Security

The VAXcluster system itself is one VMS security domain. All the security control and alarm features in the VMS system work on a clusterwide basis. These features can be used with an appropriate degree of physical security (around the systems and Ethernet cable) to achieve a desired level of overall security.

Unauthorized systems are prevented from joining the cluster because a cluster password is required to establish communications. That password is validated by both nodes during the initialization handshake to create the virtual circuit. The password prevents an unauthorized user from booting off a privately created local disk with a local authorization file (instead of a boot member) and joining the cluster. Satellite systems booting off the boot member must have been configured into a database by the system manager, effectively authorizing their entry into the cluster. A means is also provided to prevent users from performing conversational bootstraps to alter system parameters.

Ethernet cables are subject to unauthorized taps and eavesdropping. The LAVc assumes the presence of an appropriate level of physical security around the systems and Ethernet cables, as these problems cannot be solved in software. Encryption hardware is the only truly effective counterweapon to these attacks. Exploiting the vulnerabilities of Ethernet in the absence of encryption could be done, but it would require substantial time, energy, and expertise.

Internal Structure of PEDRIVER

When extending SCA to include the Ethernet, we found the layering of the original model to be somewhat inconvenient. For one thing, the PPD layer performed too many functions to be thought of as a single layer. This problem was further compounded when additional functions, such as node authentication, were included. Therefore, the approach taken was to adhere generally to the original model, but to replace the PPD and PI layers with several layers.

In the NI-SCA model, the PPD layer was replaced with the layers from the port command interface (PCI) to the datagram propagation (DX) layers. The PI layer was replaced with the datalink and physical link (PL) layers. The resultant layering may seem a bit excessive — seven layers replacing two — but is nevertheless a natural partitioning of the activities below the SCS layer. Increasing the number of layers for NI-SCA does not increase the intrinsic complexity of the port; it merely facilitates the port's description. The new NI-SCA model is shown in Figure 7, together with a brief description of each new layer.

The Port Command Interface (PCI) Layer

The PCI layer effectively implements the port by defining the interface between the port and the port driver. Normally, the modules of a given layer communicate with modules in the corresponding layer on remote nodes. Lacking this characteristic, the PCI is not a layer in the strict sense of the word but is merely an interface between the SCS and the port-to-port communications (PPC) layers.

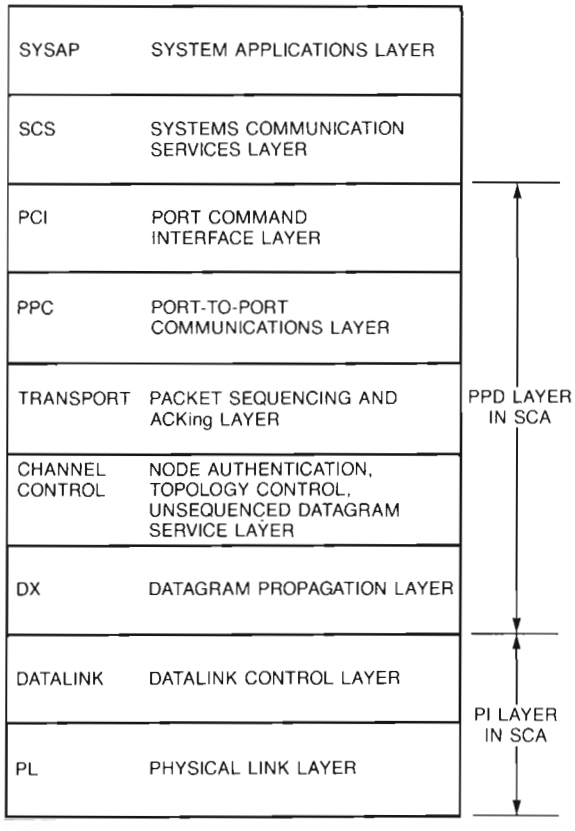


Figure 7 NI-SCA Layers

The PCI layer is the set of queues used to pass command packets down to and response packets up from the port emulator. Each packet consists of two regions:

- The port interface region is comprised of command and status information between the port and the port driver. The specifics of this region are private to PEDRIVER.
- The PPC region is comprised of the information used by the local PPC layer to communicate with a remote PPC layer. The specifics of this region are not private to PEDRIVER since the region is interconnect independent. The PPC region is the same for the Ethernet as it is for the CI bus.

The Port-to-Port Communication (PPC) Layer

The PPC layer exists below the port interface. This layer provides port services (datagrams, sequenced messages, and block transfers) to the PCI layer by translating between PCI packets and a series of PPC messages exchanged with the remote port. The PPC layer also segments block transfers into a series of sequenced messages. The datagram and sequenced services provided by the transport layer are used to exchange these messages. To be consistent with the CI bus, any errors detected at the PPC layer in a packet sent or received in sequenced mode cause the virtual circuit to be disconnected.

The Transport (TR) Layer

The transport layer uses one or more paths to the remote node to provide the local PPC layer with a sequenced-message and datagram connection to a remote PPC layer. For datagrams, the transport layer is little more than a conduit to the channel control layer. For sequenced messages, the transport layer handles all the sequencing, sending and receiving ACKs, and retransmissions required to provide guaranteed message delivery and sequentiality. Although multiple Ethernets are not currently supported in a cluster, this layer would be responsible for that functionality.

The Channel Control (CC) Layer

A channel is a path that utilizes a single Ethernet to join two ports with an authorized datagram service. To accomplish that service, the channel uses the datagram service provided by the DX layer. The channel control layer manages the network topology and therefore provides such services as node authentication, access control, and virtual circuit initialization.

The Datagram Exchange (DX) Layer

The DX layer attempts to transmit packets from the source port to the destination port. On any given system, the DX layer is the interface between the ports and the datalinks. As such, this layer is basically a switch; many ports may be above it, many datalinks below it. Note that on a single system, the DX layer may be shared among multiple ports and is not owned by any one port.

The DX layer determines which systems are on which Ethernet and transmits packets correctly to their destinations by managing the group number and multicast HELLO messages. This layer includes the group number in all the packets it transmits and checks the numbers on received packets.

The Datalink Control Layer

The datalink layer provides access to the physical link and the functions at the packet level. These

functions include the hardware adapter control, the minimum and maximum length requirements of packet, provisions for data-integrity checking, datalink header formats, and multicast addressing. For NI-SCA, this layer is provided by a separate datalink driver. This driver controls the Ethernet adapter hardware and is shared by all Ethernet users (LAVc, DECnet, LAT systems, etc.) on the system.

The Physical Link (PL) Layer

The PL layer represents the medium over which packets are sent and received. A complete specification for this layer would include the mechanisms for clocking bits on the wire, the framing of bits into bytes, electrical signal requirements, cabling, and so forth. For NI-SCA, this layer is defined by the Ethernet standard.

Network Booting of the VMS Software

Two LAVc requirements are met by booting the VMS software over the Ethernet: simplifying system management by requiring only one VMS system disk, and making possible diskless systems. The software engineering effort required during LAVc development to provide this functionality was second only to that needed to develop PEDRIVER.

Normal VMS Booting

Booting a system on a VAX processor takes place in several stages. Each stage is characterized by a loaded program that performs some prescribed function, which in turn loads and transfers control to another program.

The first such program to run is the console program, which is different on different processor types. Its basic role with respect to booting is to retrieve the input parameters, store them in the first six general-purpose registers, and then load and transfer control to VMB. VMB, referred to as either the primary bootstrap or primary loader, is the first program that is more or less common across all processor types. Depending on the processor type, VMB is retrieved either from ROM (the MicroVAX II class of systems) or the console block-storage device (other VAX systems).

Although the partitioning of work between the console program and VMB differs slightly with processor type, together they accomplish the following:

- Locate a block of memory to use during the boot
- Locate and establish an access path to the system disk
- Provide a primitive I/O system consisting of a boot driver for the system device, a file system, and the $QIO access routine
- Locate, load, and transfer control to the secondary bootstrap, called SYSBOOT.EXE for the VMS system, or DIAGBOOT.EXE for diagnostics

SYSBOOT is the secondary bootstrap selected to run when VMB is directed to load the VMS software. SYSBOOT performs the following actions:

- Loads the VMS images into memory
- Reads the system parameter file, accepts any user specified parameter changes if this is a conversation boot, and configures the system accordingly
- Allocates memory for and loads the terminal and system disk drivers
- Transfers control to the INIT module of the VMS system

The VMS INIT module initializes the now running VMS system.

- Loads the processor dependent code (SYSLOAxxx) and other loadable components into memory
- Copies the boot I/O routines to the nonpaged pool for use during any system crash
- Tries to form a new VAXcluster system or join an existing one if the parameters are set to do this
- Transfers control to the system scheduler to initiate process execution

Remote Booting Requirements

The actions performed during each of the three stages of a network boot are the same as those in a local disk boot. No modifications were required in the functional operation of these programs. What was needed was the ability to contend with an Ethernet linking the booting system with its system disk. The Ethernet has totally different characteristics than those of the block-structured disk device previously present. The plan, then, was to load a piece of software that makes the Ethernet look like a disk, thus enabling the rest of the VMS boot sequence to proceed normally.

The three primary requirements for the remote booting design and implementation were to

- Change the existing boot process as little as possible
- Require no initial state or context information on the satellite system
- Work with the existing MicroVAX II boot ROMs (Required hardware upgrades in the field would make a LAVc much more difficult to install).

The existing boot ROMs on MicroVAX II systems include an Ethernet device boot driver capable of transmitting and receiving packets, plus a VMB program containing the DECnet maintenance operation protocol (MOP). MOP locates a boot host system on the Ethernet network, uses a simple, synchronous ping-pong protocol to copy an image from the host into local memory, and then transfers control to that image.

The existing SYSBOOT program could not be loaded directly by a MOP exchange. SYSBOOT expects to be able to access the boot device as a block-structured storage device; it does not understand the various types of Ethernet adapters that may be present. Moreover, SYSBOOT would not have enough information to locate the system disk. Therefore, another image called NISCS_LOAD is inserted into the boot sequence between VMB and SYSBOOT. NISCS_LOAD provides the environment that SYSBOOT needs to do its job correctly. As a result, minimal modifications to SYSBOOT and VMS INIT were necessary.

Remote Booting Operation

The user starts the satellite boot sequence with the appropriate BOOT command on the system console. From thereon, the process is automatic.

Satellite Operation during the MOP Exchange

The VMB program in the satellite system's boot ROM interprets the boot command and attempts an Ethernet boot. VMB starts by transmitting a multicast message requesting an operating system load. This message is multicast to an architecturally specified address because the ROM cannot have any knowledge of the network configuration. This "please boot me" request is received by host systems on the Ethernet that are willing to service network boots. If the requesting satellite is one that the host is willing to service, it responds to the request with an "assistance volunteer" packet. The satellite responds to the first "assistance volunteer" packet received and ignores any others. That response causes the host to send the NISCS_LOAD image to the satellite.

Boot Member Operation during the MOP Exchange

The host side of the MOP exchange is handled by the DECnet-VAX software, which must be running on the boot member. Each boot member in all clusters on the Ethernet will hear the operating system request multicasts sent out by every satellite. Other systems that are not boot members will not have enabled reception of this multicast address.

The DECnet software responds to an incoming boot request multicast by extracting the source address of the multicast from the packet and searching the node database for a match. This 48-bit hardware address of the transmitting satellite is guaranteed to be unique on every Ethernet adapter. This address is not normally present in the database since it is not used for DECnet (or other) communication under the VMS system. Only those nodes that have been configured into the boot member's cluster by the cluster manager will have their hardware address entered into the database. The request is ignored if the multicast source does not match an address in the database. Therefore, satellites will be booted only by a boot member in the appropriate cluster.

If the source address does match an address in the database, the DECnet software starts running the maintenance operations module (MOM). This program handles the host end of the MOP exchange. MOM also looks up the satellite in the node database to get other information stored there, including the name of a load assist agent (LAA) program, which is used to customize the load procedure for a LAVc. MOM cannot do this customizing because it is a general-purpose MOP facility. MOM invokes the LAA by merging it into MOM's address space and then calling it.

The LAA was written specifically to handle the loading of NISCS_LOAD. LAA customizes the NISCS_LOAD image for the booting satellite by appending necessary information to it, including

- The name and unit number of the satellite's system disk

- The name of that satellite's root directory on that disk
- The cluster group number
- The cluster password
- A flag allowing or disallowing conversational bootstraps

The NISCS_LOAD image and appended data are then passed to routines within MOM that transmit them to the satellite using the MOP protocol. When NISCS_LOAD starts executing on the satellite, it can use this information for the next phase of the boot.

After NISCS_LOAD has been successfully transmitted, the MOP phase of the boot (and the involvement of DECnet-VAX) is complete. The boot member no longer knows that the satellite is booting, and it does not need to provide the satellite with additional special services.

NISCS_LOAD, Loading SYSBOOT, and VMS Software

The VMS system will not have been loaded into the satellite when NISCS_LOAD executes. Therefore, NISCS_LOAD is designed to run in a bare machine environment; that is, NISCS_LOAD must be specifically programmed to handle any Ethernet adapter or CPU it is to support. To date, only support for the MicroVAX II CPU has been included, along with the Q-bus adapter and the MicroVAX 2000 and VAXstation 2000 Ethernet adapters.

The NISCS_LOAD image contains four components:

- Datalink boot drivers for all supported Ethernet adapters
- A boot driver version of PEDRIVER, called PEBTDRIVER
- Primitive "class driver" MSCP code
- Parameter values assembled by the load assist agent on the boot member

PEBTDRIVER retrieves the boot member's Ethernet address, the group number, and the cluster password from the NISCS_LOAD parameter list. A virtual circuit back to the boot member is set up by transmitting a START_VC packet, which starts the normal initialization sequence. The boot member does not know that the system at the other end of this virtual circuit is booting since the virtual circuit and I/O requests sent over it are identical to those sent by a running VMS system.

Upon setting up the virtual circuit, PEBTDRIVER has a path to the system disk that NISCS_LOAD will need to continue the boot. The primitive class driver now issues a normal MSCP command to read the SYSBOOT.EXE image from that disk into memory and transfer control to that image. PEBTDRIVER remains in memory to serve as SYSBOOT's "driver" for accessing the system disk, hiding all knowledge of the Ethernet adapter. The presence of the primitive class driver makes SYSBOOT "see" the expected block-structured device interface. SYSBOOT can now load the VMS software normally by issuing a read operation over the virtual circuit set up by PEBTDRIVER.

After being loaded by SYSBOOT, the VMS system can initialize normally because the Ethernet path to the system disk is totally hidden. No operational changes to SYSBOOT or VMS INIT were necessary. The runtime PEDRIVER takes over from the boot driver during the initialization of the VMS software, thus breaking the boot driver's virtual circuit and establishing a new one.

The PEBTDRIVER portion of NISCS_LOAD remains permanently in memory. If the system crashes, that portion is activated again to write the contents of memory into the dump file. The runtime driver is not used because the state of the VMS system, the drivers, and the data structures cannot be trusted in a crashed system. The boot driver is totally ignored while the system is up; therefore, its integrity is usually left intact by the crash. As with any other boot driver, the system disk is the only known device. Therefore, the dump file must be on that disk.

Summary

We have shown how Local Area VAXcluster systems are a natural follow-on to the original VMS VAXcluster implementation using the CI bus. The cluster architecture and implementation were generally independent of the interconnect specifics; therefore, the switch to Ethernet was confined to the port driver layer. The replacement of PADRIVER with PEDRIVER and the addition of Ethernet booting was all that was required to make the product work. This combining of VAXcluster functionality with the MicroVAX systems and workstations now available, plus the

low cost and flexibility of the Ethernet, brings new power to low-end systems. These benefits include both the data and resource-sharing capabilities of VAXcluster systems, and the ability to isolate workstation users from system-management responsibilities.

The LAVc has a bright future planned. Work is in progress to allow both CI and Ethernet interconnects to coexist in the same cluster. When this work is completed, workstation users will be able to draw upon the power, resources, and speed of the large VAX machines, HSC controllers, and disk farms in the computer room. In addition, users will have full access to the same data files as do users on those mainframes. All these systems will be running the same operating system, be centrally managed, be highly available, and offer the same software environment to all users. No other product comes close to offering such total system integration from the data center to the desk top.

References

1. N. Kronenberg, H. Levy, W. Strecker, and R. Merewood, "The VAXcluster Concept: An Overview of a Distributed System," *Digital Technical Journal* (September 1987, this issue): 7–21.

2. W. Snaman and D. Thiel, "The VAX/VMS Distributed Lock Manager," *Digital Technical Journal* (September 1987, this issue): 29–44.

3. D. Duffy, "The System Communication Architecture," *Digital Technical Journal* (September 1987, this issue): 22–28.

Edward E. Balkovich
Prashant Bhabhalia
William R. Dunnington
Thomas F. Weyant

VAXcluster Availability Modeling

VAXcluster systems use redundant hardware—processors, interconnects, and storage elements—and software to achieve high system availability. No special hardware or software is required. A simple, first-order availability model is used to illustrate how this redundancy improves availability. Four VAXcluster configurations are analyzed to show that redundancy decreases system unavailability by two orders of magnitude. Decomposition techniques were used to develop these first-order availability models, which were then analyzed using "textbook" reliability analysis techniques. More complex configurations and models of broader classes of faults will require the support of more sophisticated modeling tools.

An increasing number of specialized computer systems are being dedicated to tasks that are critical to the success of an organization. For example, in the financial services industry or in manufacturing, it must be possible to access a computing system to deliver a service or to manufacture a product. Any loss of access to the computing system adversely impacts business. The ability to access a computing system when it is needed (commonly referred to as availability) is becoming an important metric used to select such computer systems. Obviously, high availability also improves the quality of service provided by general-purpose computing systems, such as those providing timesharing services.

VAXcluster systems provide high availability.[1] They can be configured so that there is no single point of failure. Each cluster is a multiple-computer system, built from standard hardware and software elements. VAXcluster systems can be expanded in increments to provide the computing power, data resources, and storage capabilities typically associated with mainframe systems.

Although these systems are not fault tolerant, they can detect, isolate, and recover from faults in their processor, interconnect, and storage subsystems. (Fault tolerance generally implies that a recovery from a fault is completely invisible to an application.) While VAXcluster systems can detect, isolate, and recover from faults, the recovery from some types of faults impacts the applications and their design. For example, a VAXcluster system will retry an I/O operation if a fault is detected in either the interconnect or storage subsystems.

The integrity of the I/O operation is ensured by the operating system. If a processor fails, however, the computations hosted by it are lost. A user must start a new session on another (available) processor. The user must depend on an application, not the operating system, to recover the state of the computation to the point at which the fault occurred. For example, a journal file can be used to recover an editing session or database transaction. In this case, the integrity of the computation is assured by the application, not by the operating system.

This paper documents a study using simple first-order models to show how the inherent redundancy of VAXcluster systems is used to achieve high availability. Although more sophisticated models are possible, the models used in this study were sufficient to illustrate the main points. It is assumed that the reader is familiar with the basic technical concepts of VAXcluster systems presented in our companion papers.[2,3] It is not assumed that the reader is familiar with the standard methods of analyzing availability used to illustrate the points of this study.

VAXcluster Structure

Figure 1 illustrates a simple VAXcluster system with terminals connected to the system via a LAT server. Either processor is accessible through that server, and dual-ported disks are accessible through either Hierarchical Storage Controller (HSC). The HSC devices and the processors are

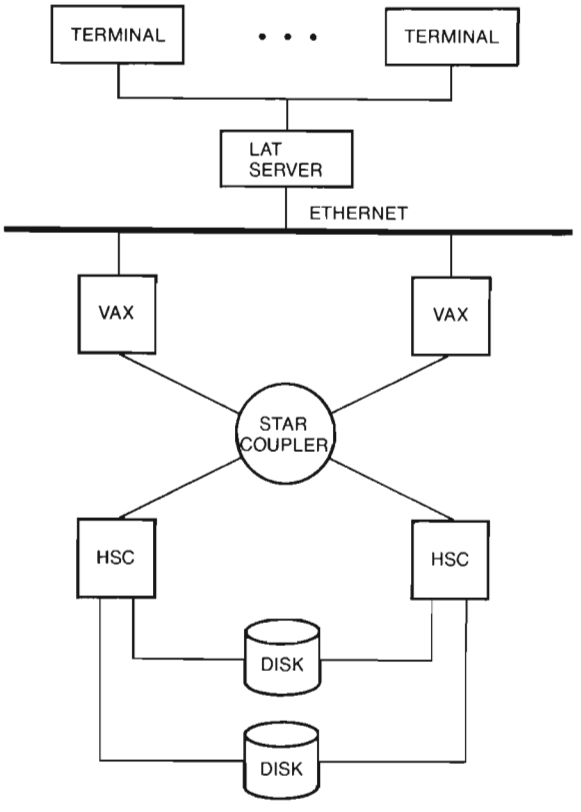


Figure 1 Simple VAXcluster Configuration

connected by a Star Coupler, a passive device offering two independent datapaths between each node of the system. Multiple disks are used to shadow a volume of information. This simple system illustrates all the basic forms of redundancy in VAXcluster systems.

Processor Failures

If a processor or its Computer Interconnect (CI) adapter fails, all computations in progress on that processor will be lost. The processor and the adapter can detect some types of faults and inform the VAXcluster system of them immediately. Other types of faults are detected by the other VAXcluster processors by way of time-outs.

When other processors detect a fault in a processor or its adapter, they reconfigure themselves to remove the failed processor from the cluster. The reconfiguration times depend on the number of locks in the system and on the number of I/O devices in the configuration. The average reconfiguration time after a processor failure is a small number of seconds.[4] After the reconfiguration is complete, the user can begin a new session on the remaining processor. Appropriately constructed applications, such as those employing journaling, can then be recovered to the point of the failure.

Interconnect Failures

The Star Coupler, a passive device, has a negligible failure rate compared with the other elements. The individual CI paths attached to a single adapter have active elements, however, and the failure rates for those paths must be considered.

If a single path fails, the CI adapter will retry the transmission on the redundant path. The retry is invisible to both the processor and the HSC device using the adapter.

If both paths fail, neither the processor nor the HSC device attached to the adapter can communicate with other elements of the VAXcluster configuration. The effect is similar to a processor or HSC failure. However, other processors and HSC devices can continue to communicate with each other.

Hierarchical Storage Controller Failures

HSC failures are managed by the VAX processors. The HSC device can detect some faults and inform the cluster about them immediately. Other types of faults are detected by the VAX processors and the disks by time-outs. When a fault is detected in an HSC device, the VAX processors will retry any I/O operations in progress by using the redundant HSC device. An HSC failure is invisible to the process issuing the QIO operation. The times required to reconfigure the system after an HSC failure depend on the number of outstanding I/O operations, the number of I/O devices, and the use of volume shadowing. The average time is typically a small number of seconds.

Volume shadow sets, hosted by an HSC device, must be reconstructed if that device fails. Although the shadow set is available during reconstruction, this process involves additional I/O that competes with user requests to read or write to the volume shadow set.

Disk Failures

HSC devices detect disk failures. Volume shadowing allows an HSC device to retry a failed I/O operation using another member of the volume shadow set. The failure of a disk in a shadow set is invisible to the process issuing the QIO operation. When a fault is detected, the volume

shadow set will be reconfigured to remove the failed volume. Once again, the average time required to reconfigure the shadow set after a disk failure is a small number of seconds.

VAXcluster Configurations Considered

Modeling Procedure

This paper focuses on the availability modeling of four simple VAXcluster configurations. The goals of the study were to

- Demonstrate the sensitivity of different reliability and availability parameters
- Demonstrate how different types of redundancy improve VAXcluster availability

These goals were achieved by first modeling the availability of a baseline configuration consisting of a VAX processor, an HSC storage controller, and a disk drive. Each element in the configuration represented a single point of failure. Next, redundancy in the form of a second VAX processor was added to the baseline configuration to create a second configuration. Another HSC storage controller was then added to create a third configuration. Finally, a disk drive and volume shadowing were added to create a fourth and fully redundant configuration. These four simple configurations were used to study the principal forms of redundancy in a VAXcluster system.

Referring to Figure 1, the configurations considered here consisted of VAX processors, a Star Coupler, HSC storage controllers, and disk drives; they did not include the Ethernet, the LAT server, or the user terminals.

Baseline Configuration — Model 1

The baseline configuration, Figure 2, consisted of a VAX processor, an HSC storage controller, and a disk drive. The processor and the storage controller were connected by way of a Star Coupler whose failure rate is negligible compared to that of the other elements. Figure 2 also shows the configuration diagram translated into a reliability block diagram in which the series positioning of each element represents a single point of failure for the configuration.

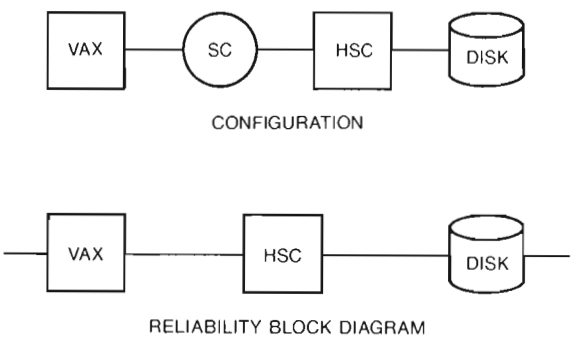


Figure 2 Baseline Configuration (Model 1)

Redundant Processor Configuration — Model 2

The second configuration considered in the study, Figure 3, added redundancy in the form of a second VAX processor. The failure of either processor or its CI adapter requires a failover process to the redundant processor with its associated VAXcluster reconfiguration activities. These activities usually complete in a matter of seconds.

In the reliability block diagram for the hardware model, the redundant VAX processors are shown in parallel because both must fail for the configuration to fail. However, the HSC device

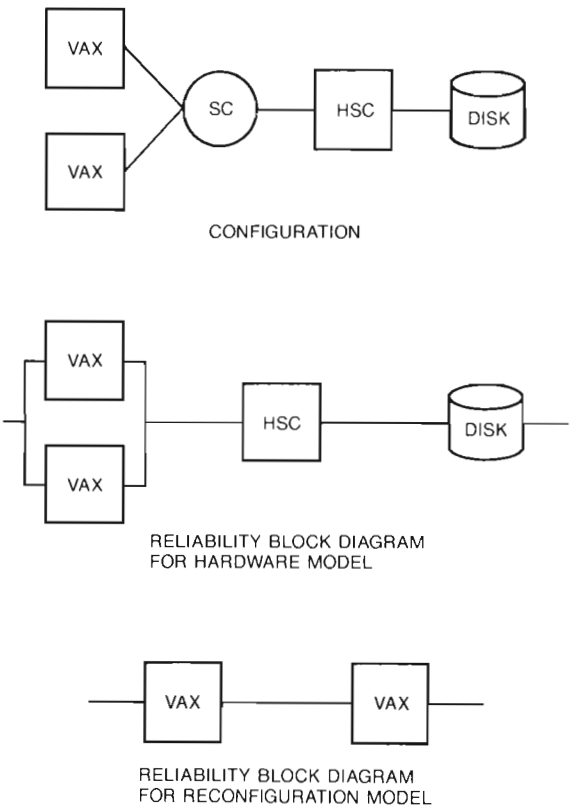


Figure 3 Configuration with Redundant Processor (Model 2)

and the disk drive are still shown as single points of failure.

If either processor fails, the VAXcluster system will undergo a reconfiguration. Depending on the user application, the system may be unavailable during the failover process.[5] This condition is represented in the reliability block diagram by the two VAX processors in series.

Similarly, the reconfiguration operation is repeated when a repaired VAX processor is re-established in the VAXcluster system. Again, depending on the user application, the system may be unavailable until the reconfiguration completes. Since either VAX processor could fail, the reliability block diagram is again valid for this condition.

Redundant Storage Controller Configuration — Model 3

In the third configuration, Figure 4, additional redundancy in the form of a second HSC storage controller was added to the Model 2 configuration, which already had a redundant VAX processor. Now the failure of either a VAX processor or an HSC storage controller requires a failover process to either the redundant processor or the controller with the associated VAXcluster reconfiguration activities.

When a repaired HSC storage controller is re-established in a VAXcluster system, there is no reconfiguration operation. Instead, the HSC device is placed in "warm stand-by" redundancy. That is, the device is not actively re-established in the VAXcluster system unless the other HSC device fails. This situation contrasts with that of the active redundancy of the VAX processor, which is immediately reconfigured back into operation as soon as it is repaired.

Fully Redundant Configuration — Model 4

A fourth configuration, Figure 5, added further redundancy in the form of a second disk drive and volume shadowing to the Model 3 configuration, which already had a redundant VAX processor and HSC storage controller.

In volume shadowing, write commands are applied to all available volumes in the shadow set. Read commands are accomplished using any available volume. A fault in a disk causes it to be removed from the shadow set. A repaired volume is merged back into a shadow set by first copying the data from an available volume as a background activity. Only upon becoming identical to existing members of the set will the repaired volume again become an available member of the shadow set.

A detailed description and analysis of the Model 4 configuration is given later.

Modeling Approach

Several formal definitions are needed to quantify VAXcluster availability.

Availability is the proportion of time that service is available from a VAXcluster system to perform a user application.

It is important to remember that this definition of availability is a general one. As the nature of the application, the size of the VAXcluster configuration, and the amount of redundancy change, availability can be defined in more complex

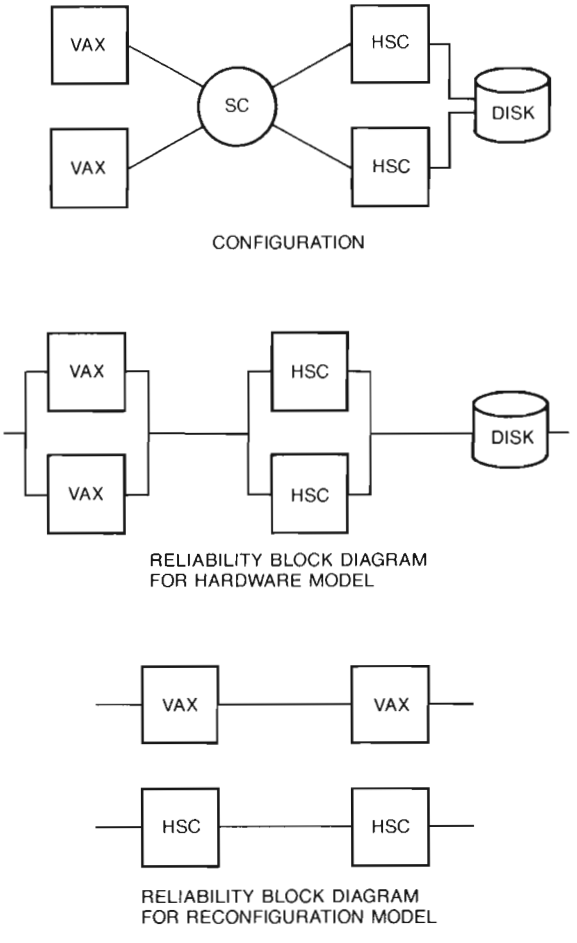


Figure 4 Configuration with Redundant Processor and Storage Controller (Model 3)

ways. For the configurations used in this study, at least one of each type of element must be running for the VAXcluster system to be operational.

Unavailability is the proportion of time that service is interrupted and that a VAXcluster system cannot perform a user application.

In this study, the related metric of downtime in minutes per year will be used rather than the system unavailability.

Reconfiguration time is the time taken to initially detect a failed element and remove it from the VAXcluster system. For a failed VAX processor, this time also includes the time taken later to re-establish the repaired element's membership in the cluster.

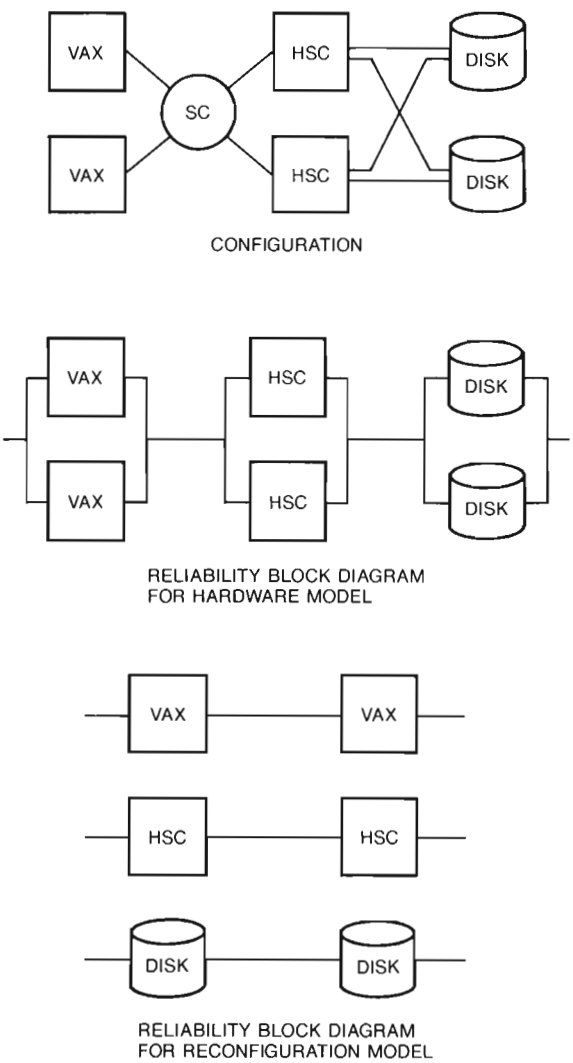


Figure 5 Configuration of Fully Redundant System (Model 4)

Note that the HSC device employs "warm stand-by" redundancy and therefore does not have any significant reconfiguration time associated with re-establishing membership in the cluster.

VAXcluster reconfiguration activities usually complete in a matter of seconds; however, in extremely rare cases, much longer times are possible.

Overview

The most common approach to modeling complex systems consists of structurally dividing a system into smaller subsystems, such as processors, controllers, and disks.[6] The availability of each subsystem is then analyzed separately, and the individual subsystem solutions are combined to obtain the system solution. One important assumption must be made to achieve a solution: the behavior of each subsystem must be independent from that of any other subsystem.

Furthermore, a decomposition technique can be applied to certain behaviors that cause system outages due to failures in redundant subsystems. In these cases, the recovery to an operational system happens quickly. Similar behavior is also present when the failed subsystem is repaired and is ready to rejoin the system to make it a fully configured system. This type of decomposition is called behavioral decomposition.

With this approach to structural and behavioral decomposition, hardware failures and VAXcluster reconfigurations are modeled separately. Such a decomposition allows the model to analyze both VAXcluster reconfigurations and complete system failures due to hardware failures. It also allows the model to analyze the sensitivity of system availability to each factor.

In this study, availability modeling captured the following factors:

- Hard failures requiring a repair call
- VAXcluster reconfigurations during which the VAXcluster system was assumed to be unavailable in this analysis
- Response time for maintenance personnel
- Time-to-repair

The following factors were not considered (except for the impact of reconfigurations due to hardware failures):

- Intermittent failures
- Transient failures

- Quorum disks
- Operational errors
- Software errors

The following modeling parameters were used:

- The mean time-between-failures (MTBF) and mean time-to-repair (MTTR) of each of the following elements:
  - VAX processor
  - HSC storage controller
  - Disk drive
- VAXcluster reconfiguration times caused by
  - VAX processor failure
  - Re-establishment of the repaired VAX processor into the VAXcluster configuration
  - HSC storage controller failure
  - Disk drive failure
- Response time for maintenance

The remainder of this section describes in detail the modeling of the fourth configuration (Model 4).

Analysis of Hardware Failure

Consider the structural decomposition of the VAXcluster configuration. Three subsystems were connected in series, each consisting of two elements in parallel. At least one element in each subsystem had to be operational for the VAXcluster system to be operational. The hardware reliability block diagram is shown in Figure 5.

Repairable systems are those for which an automatic or manual repair can be made if an element fails. Assume that each element is subject to failure and has its own repair facility.[7] If the time-to-failure of element i is exponentially distributed with failure rate λ_i, and the time-to-repair of element i is exponentially distributed with repair rate μ_i, the instantaneous availability can be obtained by the following equation:

$$A_i(t) = \frac{\mu_i}{\lambda_i+\mu_i} + \frac{\lambda_i}{\lambda_i+\mu_i} e^{-(\lambda_i+\mu_i)t}$$

As t approaches infinity, $A_i(t)$ approaches the steady-state availability and A_i equals $\mu_i/(\lambda_i+\mu_i)$.

The steady-state availability of a single element is given by the following equation:

$$A = \mu/(\lambda+\mu)$$

in which λ is the failure rate of the element and μ is the repair rate of the element. The time-to-failure and the time-to-repair are assumed to be exponentially distributed.

The steady-state availability of two elements in parallel is[8]

$$A = 1-(1-A_1)(1-A_2)$$

In Model 4, the elements in each subsystem are two VAX processors, or two HSC storage controllers, or two disk drives. Using the equation above, the availability of the processor subsystem, A_p, can be expressed as

$$A_p = 1-\left(\frac{\lambda_p}{\lambda_p+\mu_p}\right)^2$$

Similarly, the availability of the HSC storage controller subsystem, A_b, and the availability of the disk drive subsystem, A_r, can be expressed as

$$A_b = 1-\left(\frac{\lambda_b}{\lambda_b+\mu_b}\right)^2$$

and

$$A_r = 1-\left(\frac{\lambda_r}{\lambda_r+\mu_r}\right)^2$$

The aggregate availability of the VAXcluster system is

$$A_s = A_p \times A_b \times A_r$$

For exponentially distributed times, the failure rate, λ, is $1/MTBF$ and the repair rate, μ, is $1/MTTR$.

Analysis of Reconfiguration Times

Next, consider the behavioral decomposition caused by the reconfiguration that occurs when one element in a subsystem fails and an automatic failover to a second (redundant) element takes place. During this process, a reconfiguration occurs when a failed element leaves the VAXcluster system. For processors only, another reconfiguration occurs when a repaired processor later rejoins the VAXcluster system. Depending on the user application, the VAXcluster system may be unavailable to perform user applications during these reconfigurations.

For example, consider the following time line:

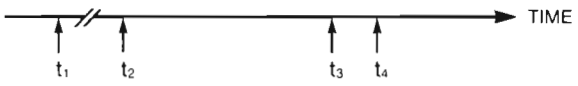


Figure 6

Time t_1 to t_2 is the VAXcluster reconfiguration time for a failed VAX processor to be detected and removed from the VAXcluster membership. Time t_2 to t_3 is the repair time for the failed hardware element. Time t_3 to t_4 is the time for the repaired VAX processor to be re-established in the VAXcluster membership.

Figure 5 includes the reliability block diagram representing the VAXcluster reconfiguration behavior of the Model 4 configuration. Each subsystem is shown as two elements in series. If any single element is not operational, the subsystem can be unavailable due to a VAXcluster reconfiguration.

For two elements in series, the availability is[8]

$$A = A_1 \times A_2$$

In model 4, the elements in each subsystem are two VAX processors, or two HSC storage controllers, or two disk drives.

Applying the equation above for elements in series, the availability of the processor subsystem, A_p, is

$$A_p = \left\{\frac{\mu_p}{(\lambda_p + \mu_p)}\right\}^2$$

Note that for the VAX processor, the rate μ_p is the reciprocal of the sum of the times t_1 to t_2 and t_3 to t_4.

Similarly, the availability of the HSC storage controller subsystem, A_b, and the availability of the disk drive subsystem, A_r, is

$$A_b = \left\{\frac{\mu_b}{(\lambda_b + \mu_b)}\right\}^2$$

and

$$A_r = \left\{\frac{\mu_r}{(\lambda_r + \mu_r)}\right\}^2$$

The aggregate availability of the VAXcluster system is

$$A_s = A_p \times A_b \times A_r$$

Assuming an operation running 24 hours a day, 365 days per year, the downtime equals $(1 - A_s) \times 525{,}600$ minutes per year. This figure is the downtime caused only by reconfigurations. The total downtime is the sum of the downtime caused by hardware failures and the downtime caused by VAXcluster reconfigurations.

Extensions to the Models

The simple models considered in this study can be extended in several dimensions.

The complexity of the configurations can be increased either by adding more VAXcluster elements or by extending the bounds of the models to include the Ethernet and its attachments. A complex configuration could include multiple clusters and multiple Ethernet segments. More complex definitions of availability are needed as the configurations increase in complexity. These definitions range from the single-user view to a measure of system productivity.

Only permanent (hard) hardware failures are considered in this study. Intermittent and transient hardware and software failures, as well as operational errors, can be added as extensions to future models. The downtime allocation reported in the literature typically attributes about one third of the total to each of the hardware, software, and operator-induced failures.[9] This result includes the effectiveness of system recovery that can be hardware based, software based, or both. Certain insidious failures can result in ineffective recovery, even in the presence of hardware or software redundancies. The term "fault coverage" represents the joint probability of fault detection and successful failover to a redundant element. A fault-coverage factor of one is assumed in this study.

This study also assumes that the subsystems of VAX processors, HSC storage controllers, and disk drives are independent. Relaxing this assumption adds to the complexity of the modeling approach. Similarly, a simplistic maintenance strategy is assumed in which each cluster element has its own repair facility.

The extensions described above add more realism to the modeling approach at the expense of added complexity in both model formulation and solution technique. Moreover, the textbook formulae used in this study are limiting and often inappropriate.

Markov modeling is a particularly useful analytic technique for formulating and solving these complex models.[7] Simulation is an alternative but computationally less efficient technique.

Another valuable industry-wide tool is the Symbolic Hierarchical Automatic Reliability and Performance Evaluator (SHARPE) software.[10] SHARPE's hierarchical feature allows complex subsystem models to be combined into a system model for efficient solution. SHARPE also employs state-of-the-art matrix-solving routines to solve large and often ill-conditioned problems arising from the Markov model formulation of these complex configurations.

Results and Conclusions

This section discusses the results of this study in detail.

The Impact of Initial Redundancy

In Model 1, no redundancy exists in the system.

In Model 2, the redundancy of the additional VAX processor reduces the total downtime to 16 percent of the downtime in Model 1.

In Model 3, the redundancy of an additional VAX processor and an HSC storage controller reduces the total downtime to almost 7 percent of the downtime in Model 1.

In Model 4, the total redundancy of an additional VAX processor, an HSC storage controller, and a disk drive reduces the total downtime to slightly under 1 percent of the downtime in Model 1.

These results show that redundancy does work to increase the availability of the system. Figure 7 shows the effect on total downtime as different forms of redundancy are introduced. A fully redundant configuration reduces system downtime by two orders of magnitude.

VAXcluster Reconfiguration Downtime

Figure 8 is an expanded view of the decrease in total downtime for the three models that include redundancy. It also shows the contribution of VAXcluster reconfigurations to total downtime. Here the typical duration of reconfiguration is used. Since Model 1 has no redundancy, the VAXcluster reconfiguration downtime is zero.

Impact of Increased Frequency of Reconfigurations

Since the previous results considered the frequency of reconfigurations equal to that of hardware failures, it was necessary to study the impact of an increased frequency of reconfigurations on downtime.

Figure 9 shows the linear relationship between reconfiguration downtime and an increase in the frequency of reconfigurations. It also shows the trend in the reconfiguration downtime as the duration of reconfiguration is first varied to three and then to six times the typical value. As shown, the key to reduced downtime is keeping the duration and the frequency of reconfigurations as low as practical. High-reliability hardware is a major factor in keeping the frequency of reconfigurations low.

Contribution of Individual VAXcluster Elements

This study also examined how much downtime an individual VAXcluster element contributes toward the total downtime.

Figure 10 shows the contribution of each element (CPU, HSC, and disk) toward the total downtime for Model 4. At a given MTBF, the VAX processor contributed 82 percent of the total

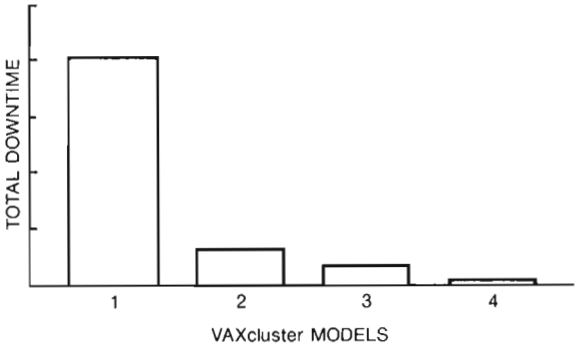


Figure 7 Impact of Initial Redundancy

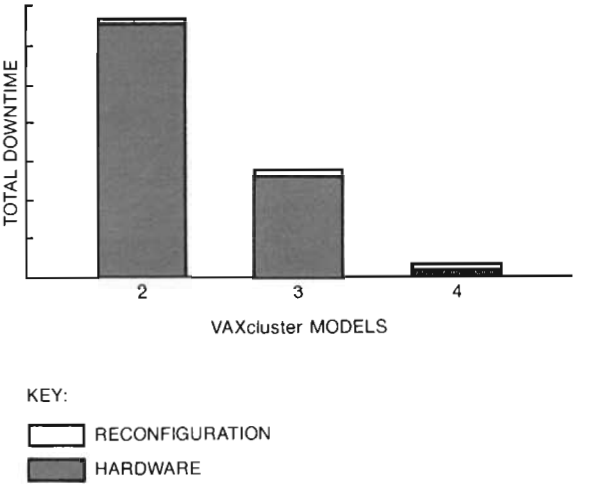


Figure 8 Total System Downtime by Model

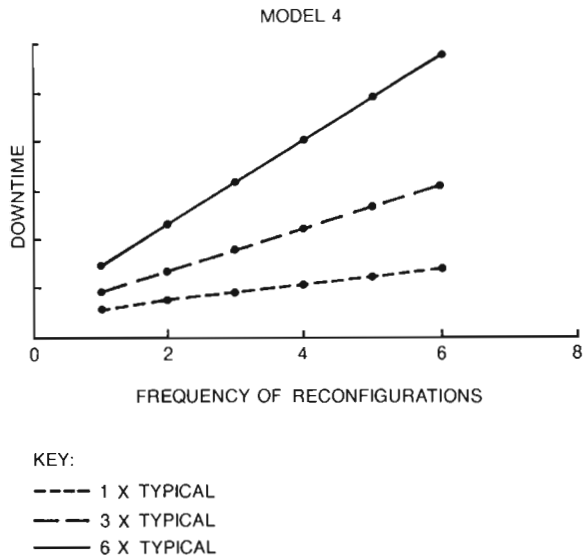


Figure 9 Reconfiguration Downtime by Frequency of Reconfigurations

downtime. When the MTBF of that particular VAX processor was improved, its contribution dropped to 57 percent.

Typical VAXcluster configurations would generally include more than the two disks used in this study. Having more disks would change the contribution of the disk subsystem to the system unavailability. (Analyzing the impact of additional disks is outside the scope of this paper.)

The reliability improvement in the MTBF of the VAX processor decreased both the hardware and the reconfiguration downtime. Figure 11 shows a decrease of approximately 58 percent in total downtime.

Hardware Downtime versus Response Time

This study included a response time for maintenance for each call as part of the recovery time. If an on-site maintenance person were available, the response time would be eliminated, thus speeding the recovery of a failed element. When this strategy is considered, the hardware downtime drops by almost 60 percent. Figure 12 shows this reduction as applied to Model 4.

The N of M Redundancy Case

The results given so far have been for (1 of 1) and (1 of 2) configurations of VAX processors, storage controllers, and disks. In this section, the hardware downtime results for VAX processors are generalized to the (N of M) redundancy case. The assumption is that N processors are required for capacity and M processors represent $M-N$ redundancy. The steady-state availability is defined as the probability of at least (N of M) processors working. The cluster is assumed to be unavailable when less than N processors are working. Note that, depending on the configuration and application, clusters with less than N working could be considered as partially available. The case of the partially available cluster is not considered here.

The (N of M) availability, as defined above, is

$$Availability_{(N\ of\ M)} = \sum_{i=0}^{M-N} \frac{M!}{i!(M-i)!} \cdot \left(\frac{\mu}{\mu+\lambda}\right)^{M-i} \left(1-\frac{\mu}{\mu+\lambda}\right)^{i}$$

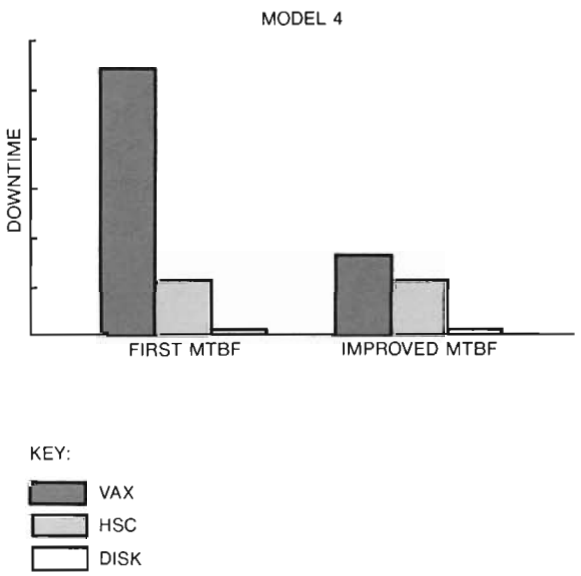


Figure 10 Contributions of Individual VAXcluster Elements to Downtime

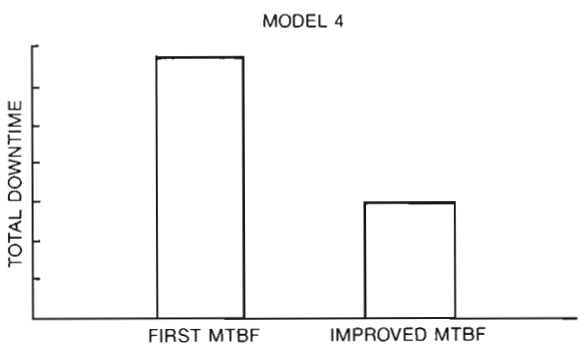


Figure 11 Total System Downtime by VAX Processor MTBF

An application of the (N of M) availability expression for VAX processors is shown in Figure 13. The number of VAX processors required to run applications to capacity was set to 1, 2, 3, and 4. The values for M were set to $N+0$, $N+1$, and $N+2$. High availability is typically measured in values much greater than 0.99. Therefore, to distinguish the variation in availability, the origin in Figure 13 is not zero but much greater than 0.9. With no redundancy ($M=N+0$), availability decreases with an increase in the number of processors. That decrease occurs because more CPUs must be available to deliver the application, bringing about a greater likelihood of failure and outage. This result is shown in the graph by the downward trend of the "$N+0$" bars. Adding a single redundant CPU ($M=N+1$) greatly improves system availability. Adding a second redundant CPU ($M=N+2$) has little additional effect on availability. The additional improvement is not visible on the graph, even with the expanded vertical scale. It can therefore be assumed that "$N+1$" redundancy is sufficient for most applications.

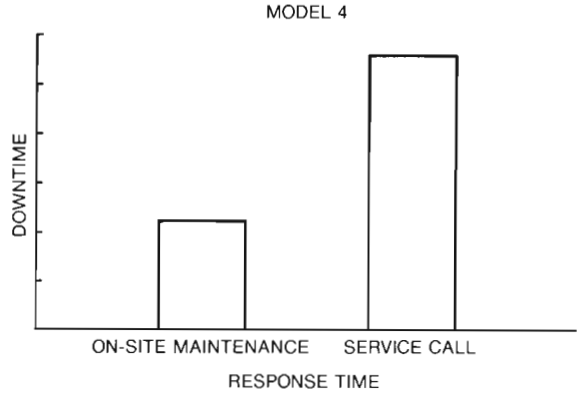


Figure 12 Hardware Downtime versus Response Time

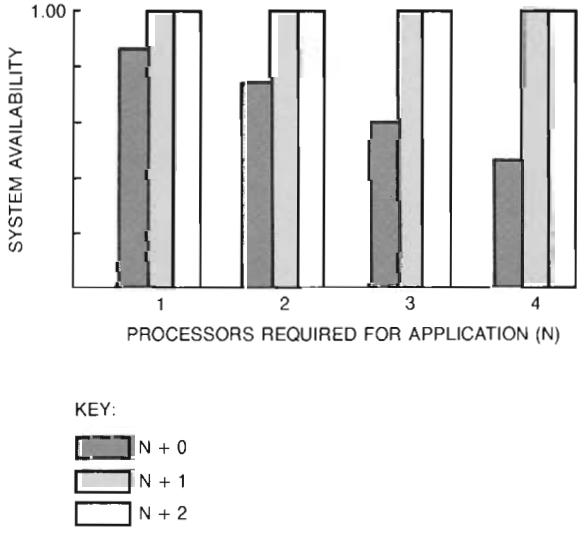


Figure 13 The (N of M) VAX Processor Redundancy Case

Summary

VAXcluster systems achieve high availability by eliminating single points of failure with redundant hardware. Redundancy is introduced at the level of standard processors, interconnects, storage elements, and software. No special-purpose hardware or software is required. The same hardware and software could be used to construct a less available uniprocessor system without volume shadowing.

The simple analytic models of VAXcluster availability developed in this study show that redundancy yields dramatic improvements in system availability for the system configuration shown in Figure 1. The average downtime of the system is reduced by nearly two orders of magnitude from that of a similar uniprocessor system without volume shadowing.

Because they can be expanded incrementally, VAXcluster systems requiring a minimum number of N processors to achieve a performance goal can achieve significant improvements in availability with the addition of a single redundant processor. There is no requirement to fully replicate all the original N processors.

The system configurations analyzed in this study are simple ones designed to illustrate the most important concepts of VAXcluster systems. The downtime of a more complex VAXcluster configuration, with many additional processors, HSC devices, and disk drives, changes system downtime in complex ways. In general, additional redundant hardware causes multiple hardware failures to become less of a factor. When faults do occur, however, time is required to reconfigure the system. Some applications may view these small reconfiguration times as a source of system downtime. In such cases, additional hardware increases both the frequency of reconfigurations and their contribution to system downtime. Continuing efforts to improve hardware reliability are particularly important to

reduce the downtime due to multiple hardware failures and the frequency of reconfigurations that might be counted as downtime by an application.

The analysis used in this study uses structural and behavioral decompositions of systems. Structural decomposition is the most common approach to modeling complex systems. However, this approach assumes that each subsystem behaves independently. For the systems and phenomena considered in this study, recovery to an operational state happens quickly following a system reconfiguration caused by a fault in a redundant subsystem. Similar behavior is also present when a failed VAX processor subsystem is repaired and is ready to rejoin the system.

These modeling approaches were applied to the VAXcluster system, which was considered to be repairable. Structural decomposition was used to model the hardware failures of each VAX processor, HSC device, and disk drive in the system. Behavioral decomposition was used separately to model the reconfiguration times.

Notes and References

1. This paper is limited to CI-based VAXcluster systems. Local Area VAXcluster systems, implemented with Ethernet, are not considered in this analysis. The reader should be aware that there are significant configuration differences between CI-based VAXcluster systems and Local Area VAXcluster systems that lead to important differences in system availability.

2. N. Kronenberg, H. Levy, W. Strecker, and R. Merewood, "The VAXcluster Concept: An Overview of a Distributed System," *Digital Technical Journal* (September 1987, this issue): 7–21.

3. *VAXcluster Systems Handbook* (Bedford: Digital Equipment Corporation, Order No. EB-28858-46, 1986).

4. E. Los, S. Snaman, S. Szeto, and D. Thiel, Corrections to "Cluster State Transitions," *VAXcluster Systems Quorum*, vol. 2, issue 3 (Digital Equipment Corporation, February 1987): addendum.

5. During reconfiguration, significant processor resources are used to reconstruct the lock manager database. Some real-time applications may view the reconfiguration time as a system outage.

6. S. Bavuso et al., *Dependability Analysis of Typical Fault-Tolerant Architectures Using HARP*, CS-1986-18.

7. K. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications* (Englewood Cliffs: Prentice Hall, 1982).

8. P. O'Connor, *Practical Reliability Engineering* (Chichester: John Wiley & Sons, Ltd., 1985).

9. D. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design* (Bedford: Digital Press, 1982).

10. R. Sahner and K. Trivedi, *SHARPE: Symbolic Hierarchical Automatic Reliability and Performance Evaluator,* (Durham: Duke University Department of Computer Science, September 1986).

Daeil Park
Rekha D. Von Ehren
Tzyh-Jong Wang
Nii N. Quaynor

System Level Performance of VAX 8974 and 8978 Systems

This paper describes the results of performance tests on the VAX 8974 and 8978 systems in two different situations: a scientific environment, and a transaction processing environment. Benchmarks were run in both environments to collect application throughput, I/O activity, and other performance data. The results of a VAX 8700 were used as a baseline comparison. Based upon measured data, two models, one for each environment, were constructed to predict system performance under different configurations. These models were run with various parameters to construct performance curves. Subsequent test results showed that both models predicted performance accurately. The 8974 performed 3.2 to 4 times faster, and the 8978, 6 to 8 times faster, relative to the 8700.

The VAX 8974 and VAX 8978 systems are powerful new systems based on Digital's VAXcluster technology. These systems consist of either four or eight VAX 8700 processors respectively, packaged with an I/O subsystem of storage controllers and disk arrays. This paper presents the performance of the VAX 8974 and VAX 8978 systems in both a scientific environment and a transaction processing environment. For comparison, the corresponding VAX 8700 data is presented as the base-level performance.

The scientific environment was measured using multistream batch jobs. The transaction processing environment was measured using a multiuser interactive workload that simulated an order entry and inventory control system. The measured performance for both environments is presented in terms of user-visible performance, system behavior, and resource utilization of the applications.

Based on the measured data, performance models of VAX 8974/8978 systems under each of the two environments to predict the performance for different configurations. The construction of the model and some results are discussed following each measured performance section.

VAXcluster Performance Overview

A VAXcluster system is a highly integrated organization of VAX/VMS systems can be viewed as a single-domain information management system. It is a state-of-the-art distributed system providing full data-sharing functions. All the accesses to files and records are coordinated by locking schemes implemented by the distributed lock manager.[1] The distributed lock manager is a VMS feature that has been extended to provide synchronized read/write resource sharing among the nodes in a VAXcluster system. Being a multicomputer system of a single management domain, a cluster offers increased availability and performance.

The performance of a VAXcluster system can be observed at many levels, such as the Computer Interconnect (CI) and the System Communication Architecture.[2] The context used in this paper, however, is the system-level, or user-perceived, performance. The questions that immediately arise about VAXcluster performance are how it grows as additional processors are added, whether the performance grows in a linear scale, and if not, what performance range is expected compared to the single-system performance.

There are two primary factors that affect the performance of a VAXcluster system: a communication overhead, and a locking overhead. The first factor is related to the management of the VAXcluster system. It is the cost to maintain the multiple processors in an integrated system and includes such overhead as the compute time to maintain the connections between the nodes. A communication overhead always exists in a VAX-

cluster system, regardless of the applications and the size of the cluster, although that overhead is generally small.

The second factor comes from sharing a resource clusterwide. Every access made to a shared resource by the processes must be regulated by a certain synchronization scheme. In a VAXcluster environment, this synchronization is implemented by using locks. A lock operation may involve sending and receiving messages between processors. A previous study shows that a lock request in a VAXcluster system may take seven times as long as that in a single VAX/VMS environment.[3] Therefore, the performance of a VAXcluster system will depend upon the degree of data-sharing of a particular application.

This study has been conducted to understand what implications these factors, especially the locking overhead, have on the system-level performance of a VAXcluster system. The two applications used in this study show the extremes in terms of degrees of data-sharing. The scientific workload had no files being shared by the processes, whereas with the transaction processing workload, all the files and records are shared clusterwide by all the processes. The goal of this study was to find the relative performance range of a VAXcluster system across the entire application space by tracing the performance of the two extreme applications discussed above.

Scientific Environment

Workload Description

The scientific workload, called SCIENCE, is a suite of multistream (homogeneous) batch jobs. These jobs are well-known programs frequently used in science and research environments. Four benchmarks commonly used in physics are ISAJET and GEISHA, two Monte Carlo simulations used in high-energy physics applications, and TAIR and TWING, two tests used in aerodynamics applications. Three other programs used in chemistry are GAUSSIAN 82, a quantum chemistry package; MOPAC, a general-purpose semiempirical molecular orbital package; and RS/1, an interactive data analysis software package frequently used in chemistry labs.

Performance Metric for SCIENCE Workload

The most important performance metric is throughput. Throughput is defined as the number of jobs that the system can process in a given time. This metric was derived in the following manner, using the elapsed times extracted from the batch log files. For a closed system with one job,

$$\text{Throughput} = \frac{1}{\text{Average elapsed time}}$$

The following steps were used to apply this equation to the multinode, multistream system:

$$\text{Average elapsed time per job} = \frac{\text{Sum of elapsed times for all jobs}}{\text{Total number of jobs}}$$

in which Total number of jobs = Number of nodes × Number of streams, and

$$\text{Throughput} = \frac{\text{Total number of jobs}}{\text{Average elapsed time per job}}$$

The SCIENCE workload is a suite of representative programs, each yielding a throughput for each system. To compare the performance of systems under this workload, the multiple relative performances based on the individual throughput comparison have to be aggregated. The geometric mean is chosen to aggregate the relative performances, with equal weight on each program.[4,5]

Test Methodology

The basic methodology of this study was to increase the load on the system gradually until the processors were fully utilized, thus yielding a peak throughput for a particular configuration. Since all the benchmarks were run as batch jobs, this saturation was achieved using multistream batch jobs. Up to five batch streams on each processor were run for each benchmark tested.

Potential I/O and memory bottlenecks were minimized by allowing large sizes of user working sets and by allocating one disk per job stream for data and scratch files.

Hardware and Software Configuration

The hardware environment consisted of the following elements:

- A VAX 8700 system with one CPU, two HSC70 storage controllers, and two SA482 storage arrays
- A VAX 8974 system with four VAX 8700 CPUs, two HSC70 storage controllers, and six SA482 storage arrays

- A VAX 8978 system with eight VAX 8700 CPUs, four HSC70 storage controllers, and twelve SA482 storage arrays

The software environment consisted of the VAX/VMS version 4.4 operating system and FORTRAN version 4.3.

Characterization of the SCIENCE Workload

The seven benchmarks of the SCIENCE workload were grouped into two categories based on their I/O behavior. One group included the benchmarks with virtually no I/O activity; the other with those that generated some I/O activity.

MOPAC and TWING both generate few I/Os, thereby falling into the first category. The remaining five benchmarks, ISAJET, GEISHA, TAIR, RS/1, and GAUSSIAN 82 exhibit some I/O activity. Among all, GAUSSIAN 82 is the most I/O intensive. MOPAC and GAUSSIAN 82 were chosen as being representative of each category. Before starting the experiments, we ran the representative benchmarks on a VAX 8700 system to study the characteristics of the system resource usage. The following graphs give a profile of the two categories in terms of these studies.

Figure 1 shows the profiles of MOPAC and GAUSSIAN 82 in terms of processor utilization plotted against elapsed time. Note that a single stream of MOPAC saturated the VAX 8700 processor during the entire run of almost 40 minutes, doing virtually no I/O. On the other hand, GAUSSIAN 82 consumed the most CPU power in the first five minutes and then remained at a lower rate (67 percent) of CPU utilization for the rest of the run time. For the first five minutes, GAUSSIAN 82 generated little I/O activity. Then, however, it generated a heavy I/O load — up to 21 I/Os per second — to the user disk during the rest of the run. The I/O transfer size of GAUSSIAN 82 is the largest of all the tests, around 25 kilobytes (KB) per request. The I/O data rate of a single GAUSSIAN 82 test, collected using the Software Performance Monitor (SPM) program with 60-second intervals, shows as much as 530KB per second during this I/O intensive period.

Results and Observations

MOPAC Results. Figure 2 plots the throughput of the MOPAC benchmark against the total number of streams in the cluster. The throughput increases linearly up to one job stream per processor. Beyond this point the curves remain flat. This flattening occurs because the benchmark is very CPU intensive, and one stream saturates a single processor with an average utilization of 99.6 percent. Therefore, adding more streams does not increase throughput.

The throughputs at which the curves flatten out are 1.6, 6.4, and 12.8 jobs per hour respectively for the VAX 8700, VAX 8974, and VAX 8978 systems. In terms of relative performance, the throughput of the VAX 8974 and VAX 8978 systems were 4.0 times and 8.0 times respectively greater than the throughput of a single VAX 8700 CPU, all showing linear growth with the number of streams.

GAUSSIAN 82 Results

Figure 3 shows the throughput for the GAUSSIAN 82 benchmark plotted against the total number of concurrent streams on all the systems.

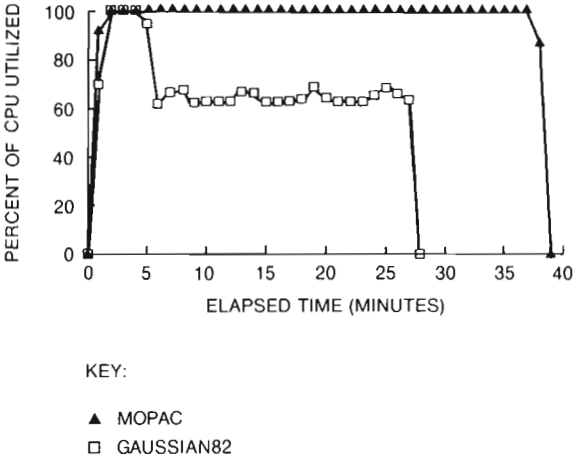


Figure 1 Transient CPU Utilization

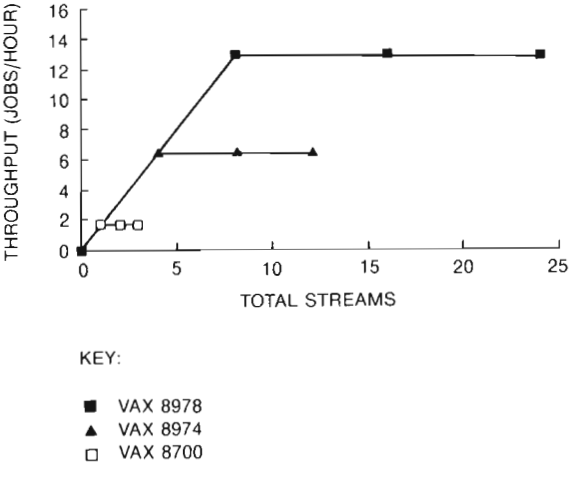


Figure 2 MOPAC Throughput

The curves show how throughput grows as the number of processors increases in the cluster. The VAX 8974 system achieved a maximum throughput of 12.1 jobs per hour with 16 concurrent streams. This throughput is 3.8 times that of the VAX 8700 CPU, which achieved 3.2 jobs per hour. The peak throughput of the VAX 8978 system was 21.9 jobs per hour, or 7.0 times that of the VAX 8700 CPU. The relative figure for the VAX 8978 system is somewhat low because there was an imbalance in the use of the I/O subsystem.

Table 1 shows the I/O activities for each HSC70 device during the five-stream run of GAUSSIAN 82 on the VAX 8978 system. All the numbers are averaged for the entire run time. One can clearly see in this table that some HSC70 devices were loaded more than others. Most disks were connected to the two HSC70 controllers, labeled HSC011 and HSC014, indicating that the other two were hot-standbys for the case of failovers. This loading variation happened because user disks were randomly assigned to the job streams. The data rate of over 2 megabytes (MB) per second on HSC011 was only the averaged number; the peak rate was close to 4MB per second, thus limiting the I/O rate. The total data rate on the CI bus of the VAX 8978 system was over 4MB per second, 2.3MB of which was through one HSC70 device. This limited the performance of five processors in the cluster.

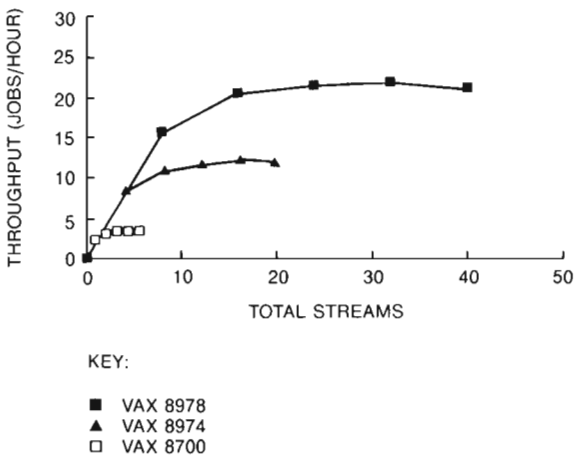


Figure 3 GAUSSIAN 82 Throughput

Note that within individual system configurations, throughput increases as the number of streams increases. With the VAX 8974 system, for example, one stream per processor produced a throughput of 2.23 jobs per hour, increasing up to 3.06 jobs per hour — a 37 percent increase — with five streams.

Performance Summary

Table 2 shows the relative performance of each benchmark in terms of maximum throughput achieved with respect to a single VAX 8700 CPU. The performance of the VAX 8974 and VAX 8978 systems ranged from 3.76 to 4.00 times, and 6.95 to 8.00 times that of the 8700, with geometric means of 3.88 and 7.40 respectively.

Simulation of the GAUSSIAN 82 Workload on the 8974/8978

Based on the measured data, a model called SIMsci was developed to describe the performance of the 8974/8978 under GAUSSIAN 82, the multistream, scientific computation workload. As described earlier, GAUSSIAN 82, a computational package for quantum chemistry, is a collection of routines for different calculation needs. The key computational behavior patterns

Table 1 I/O Activities per HSC Device

HSC70	No. of Spindles	I/O Rate (Requests/ Second)	Data Rate (KB per Second)
HSC011	24	96.4	2126.2
HSC012	2	12.7	238.8
HSC013	2	12.5	247.7
HSC014	12	65.4	1464.2

Table 2 SCIENCE Performance Relative to the VAX 8700

Program	VAX 8974	VAX 8978
GEISHA	3.76	7.02
ISAJET	3.88	7.40
TAIR	3.86	7.29
TWING	4.00	7.97
MOPAC	4.00	8.00
RS/1	3.82	7.22
GAUSSIAN 82	3.84	6.95
Geometric Mean	3.88	7.40

of this workload modeled by SIMsci are

- An executing stream places significantly different loads on the CPU and the disk at different times of execution (see Figure 1 for the transient CPU utilization pattern).
- An executing stream has a lot of I/O and CPU overlap (i.e., computation continues while I/O is in progress).

As shown in Figure 4, SIMsci consists of batch jobs (as concurrent streams), processors, and I/O devices. An executing batch job accesses both CPUs and I/O devices. The execution of a job is modeled as several interconnected stages. Each stage represents an executing interval during which the job has similar utilizations of the CPUs and the I/O devices. These stages are introduced to capture the transient behavior of GAUSSIAN 82 shown in Figure 1. Note that the number and types of stages depend on the input data to GAUSSIAN 82, which triggers different routines to execute accordingly.

The CPUs and I/O devices are the principal resources consumed by a typical batch job. SIMsci models a CPU as a single-server queue (i.e., it can serve one batch job each time). When more than one batch job competes for the same CPU, the jobs are served in a round-robin, time-sliced fashion. The CPU serves a job exclusively either for a fixed duration (e.g., 200 milliseconds) or until the job gives up the CPU (e.g., issues an I/O request), which then switches to another waiting job. The I/O device is simply modeled as a time delay since the GAUSSIAN 82 experiments are designed to avoid I/O resource contention. The presence of simultaneous CPU computations and I/O operations (over 30 percent of the time, as observed from direct measurement), was modeled. For a certain percentage of times, a job continues its computations within the CPU while its I/O request is being processed. For the rest of the times, a job is on hold while its I/O request is in progress.

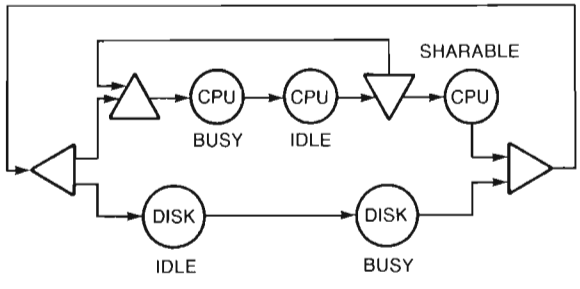


Figure 4 Model Structures of SIMsci

SIMsci uses the following model parameters to describe the interactions of job, CPU, and I/O devices:

- TotalStage, the total number of distinguishable stages of a batch job
- Nio(I), the total number of I/O requests at stage I
- TcpuUser(I), the total CPU time used by GAUSSIAN 82 at stage I
- TcpuSys(I), the total CPU time used by the VMS software at stage I
- TcpuIdle(I), the total CPU idle time due to page and swap waits at stage I
- TioWait(I), the total time that the job waits for its I/O to complete at stage I
- RTio(I), the average response time of disk I/O at stage I

The values of these parameters were derived from the measurement data. Several assumptions were made about the relationships between these parameter values and the VAXcluster configurations and job loads per node. First, it was assumed that each job's Nio, TcpuUser, and TioWait should have the same values for both the VAX 8974 and VAX 8978 configurations and for different job loads (i.e., number of streams per node). These assumptions were made because each GAUSSIAN 82 workload would always execute the same codes with the same data in any of the environments.

Second, it was assumed that TcpuSys increases as the number of nodes and the number of streams increase, thus adding communication load within the cluster and scheduling load within each node. The third assumption was that TcpuIdle increases as the number of nodes increases, since more page or swap requests would be placed on the page/swap disk, which is shared by all nodes in the cluster. It was also assumed, however, that TcpuIdle decreases as the number of streams per node increases. The more streams per node, the higher the probability that at least one job without page faults exists and can utilize the CPU while other jobs are doing paging or swapping. These assumptions were consistent with the measurement results.

SIMsci was validated against the measured data of three key metrics, job elapsed time, CPU utilization, and disk I/O rate, with less than 5 percent difference.

The performance data collected were throughput per hour and CPU utilization. Figure 5 shows that the measured and modeled results overlap for both the VAX 8700 and VAX 8974 systems, thus indicating the accuracy of the model. The 8978 curves, however, differ from each other. The previous section discussed the fact that the measured throughput of GAUSSIAN 82 was somewhat low due to the imbalanced I/O subsystem. Therefore, the model results here give us a best-case throughput when there is no I/O bottleneck. Although SIMsci produces reasonably accurate results with little effort, it does have its limitations. One major one is that SIMsci cannot predict the saturation of the I/O subsystem.

SIMwic assumes that I/Os are always free of bottlenecks; thus it cannot predict the performance of the VAX 8974/8978 systems under heavy workloads (e.g., 10 or more streams per node).

Transaction Processing Environment

Workload Description

The warehouse and inventory control (WIC) workload is a transaction processing program based on the on-line support required to manage the movement of items into and out of a warehouse. Although WIC is a warehouse application, it is a representative transaction processing application.

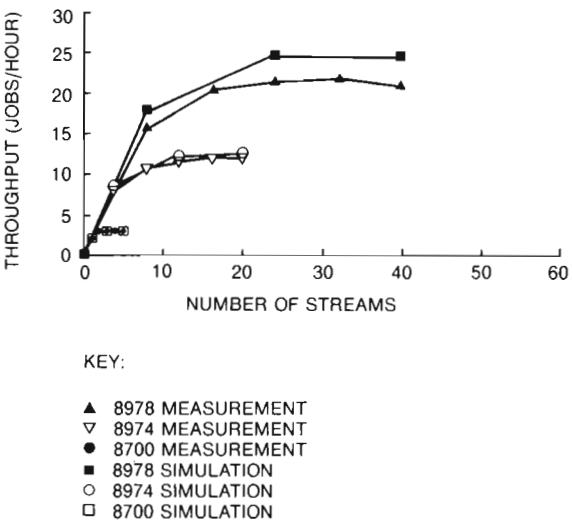


Figure 5 GAUSSIAN 82 Throughput—Model versus Measured

A WIC workload is divided into five functional parts, each associated with one task type. The five task types and the percent of total tasks represented by each type are given as follows:

- Receiving – Performs the functions needed to log the receipt of parts from the loading dock into the warehouse (17 percent)
- Inventory – Queries and updates the files containing inventory information (10 percent)
- Warehouse – Performs the functions needed to pick parts based on selected orders (10 percent)
- Order entry – Places orders to be filled by the warehouse (46 percent)
- Purchase order – Composes purchase orders (with outside vendors) for parts to be stocked in the warehouse (17 percent)

Each task is performed a specified proportion of the execution time. The task selection percentages reflect the assumption that the average flow of items into the warehouse equals the flow out of the warehouse during peak-hour operations.

Each task consists of a number of transactions. A transaction is defined as one or more user input steps followed by computation, database I/O, and output to the terminal user. Each task has an average of 7.8 transactions in the WIC application. Since a transaction implies the initiation of work by the system, throughput is measured in terms of transactions per second.

All menus and forms are implemented by requests to the VAX Transaction Data Management System. Inquiry and update operations take place on seven different application files in the VAX Record Management Services (RMS) software.

Performance Metrics for WIC Workload

- System throughput is defined as the total number of transaction processed systemwide in constant time (one second), or transactions per second (TPS). This number includes all types of transactions. Figure 6 illustrates the user and system actions needed for one transaction.
- User productivity is the average number of transactions each user completes in a unit of time, expressed in transactions per user per hour.

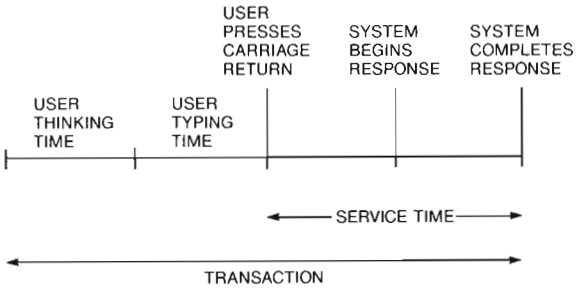


Figure 6 Transaction

- Mean service time is defined as the average time required to complete a transaction. This time does not include the input typing time or think time, but does include the time taken for screen output. A specific receiving transaction, called REC3, was chosen for the evaluation of this metric. REC3 involves updating three records and writing one record several times, which represents a moderately complex unit of work.

Test Methodology

The transaction processing environment was created by using remote terminal emulators (RTEs), which emulated all activities of terminal users. The RTEs also kept track of each transaction and the time of its occurrence and maintained the transaction mix throughout the experiment. Several systems of the VAX 8600 class were used as RTEs to load the systems under test, called SUTs.

To establish a base level of performance, the initial set of experiments was carried out with one VAX 8700 CPU as the SUT. The VAX 8974 and VAX 8978 systems were then tested by varying the number of users, and hence the number of transactions.

The RTEs logged users into the SUTs in the cluster at four-second intervals (users were evenly distributed between the SUTs in the cluster for all the configurations tested). After logging in, each user started his application, also at four-second intervals. After the SUTs reached a steady state, data was collected for 20 minutes on both the SUTs and the RTEs.

Hardware and Software Configuration

The hardware environment for each VAXcluster configuration included the same I/O subsystem. The hardware components of the configurations consisted of the following elements:

- A VAX 8974 system with four VAX 8700 CPUs, each with 32MB of memory, two HSC70 controllers, one SA482 storage array for the system; and the paging/swapping software, and three SA482 arrays for the database.
- A VAX 8978 system with eight VAX 8700 CPUs; the other hardware was the same as the VAX 8974 system's above.

The software environment consisted of the VAX/VMS version 4.5 operating system, VAX-11 ACMS version 2.0, VAX-11 TDMS version 1.4, VAX-11 CDD version 3.1, VAX-11 COBOL version 3.1, and SPM version 3.0.

In addition to the general tuning of the SYSGEN parameters, several application-specific parameters were adjusted for the best performance. These include the number of application server processes, and the size of the RMS global buffer used to buffer some portion of each RMS file. In a distributed system like a cluster, increasing the buffer size can result in additional I/O requests caused by more frequent buffer invalidations. The database consisted of 14 RMS indexed-sequential files spread over 12 disk spindles to balance the I/O rates.

Performance Results and Observations

System Throughput

Figure 7 displays the system throughput (the number of exchanges processed) at different user loads on the different configurations. These curves give a global indication of the overall relative performance of the VAX 8974 and VAX 8978 systems.

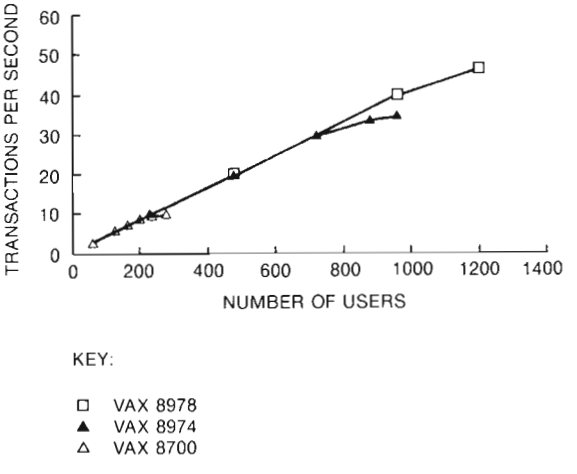


Figure 7 WIC Throughput

The VAX 8700 CPU peaked at 10.5 transactions per second (TPS) while servicing 280 users. The VAX 8974 configuration achieved its maximum throughput rate of around 34.5 TPS while servicing 960 users. Thus the maximum throughput of the VAX 8974 system is about 3.3 times that of a single VAX 8700 CPU. The performance gain is not linear in this case because the degree of data-sharing is quite high in the WIC application, causing the locking overhead typical in a cluster environment.

The limiting resource for the VAX 8974 system and the VAX 8700 CPU was processor power. The 8700 and each processor in the 8974 were fully utilized at around 960 users for the 8974 and 280 users for the 8700. The corresponding I/O rates for the peak user levels were 220 and 60 respectively for the 8974 and the 8700.

The VAX 8978 system achieved a maximum throughput of 47.5 TPS while servicing 1,200 users, which is only 4.5 times the VAX 8700 throughput. Even taking into account the cluster overhead, this result is a very low relative performance gain. Clearly, this result indicates that with the current implementation of the application the VAX 8978 performance was limited by some resource.

After more investigation, we found that the disks were this limiting resource. We observed a peak of 320 disk I/Os per second at 1,200 users on the VAX 8978 system. Let us assume that the I/Os were uniformly distributed between the 12 spindles (which they were not). In this case, dividing the peak of 320 I/Os between the spindles yields 27 I/Os per spindle. However, the actual maximum observed on any one spindle was actually around 35 I/Os per second. Figure 8 plots the four highest I/O rates.

Investigating further, we found that these disks also had large queue lengths associated with them (up to 4 requests at 1,200 users). Clearly, the I/O rates above coupled with the large queue lengths established that disk I/Os were the limiting resource for the VAX 8978 configuration. In the section Simulation of the WIC Workload, where the modeling of VAXcluster systems is discussed, more data on the VAX 8978 performance will be presented without this limiting factor.

Figure 9 gives a view of system performance in terms of throughput and processor utilization. Note that the more processors there are in the system, the more processor power it takes to do the same amount of work. For example, to obtain a throughput level of 30 TPS, the VAX 8974 system required 300 percent of the processor power and the VAX 8978 system required around 340 percent. This extra power is needed by the cluster overhead, which involves locking activities and message transfers between the processors.

User Productivity

Figure 10 provides another view of throughput in terms of user productivity, defined as the throughput per user (the throughput in Figure 7 divided by the number of users).

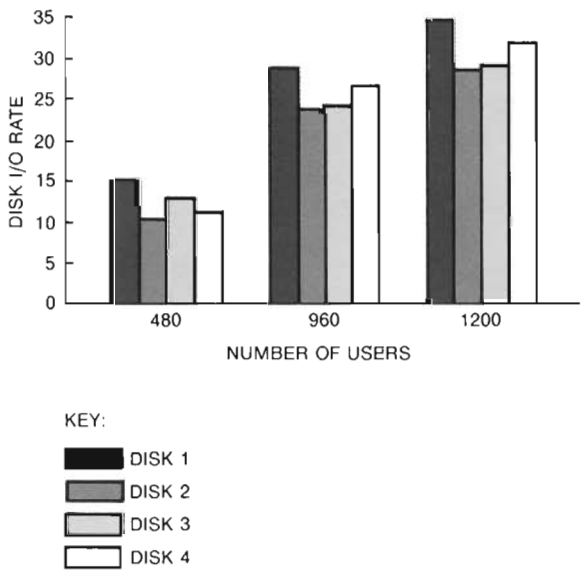


Figure 8 Disk I/O Rates for WIC (Top Four Disks)

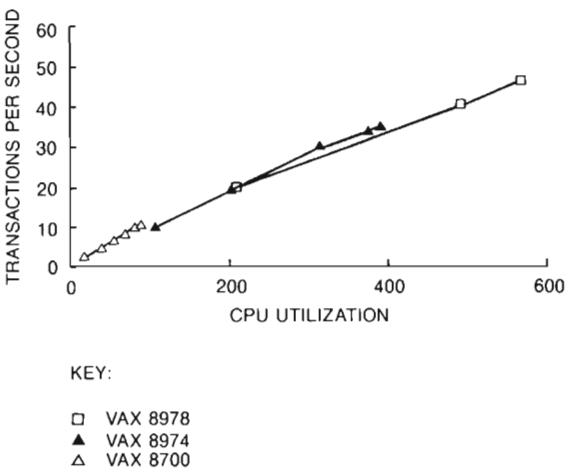


Figure 9 Throughput versus CPU Utilization

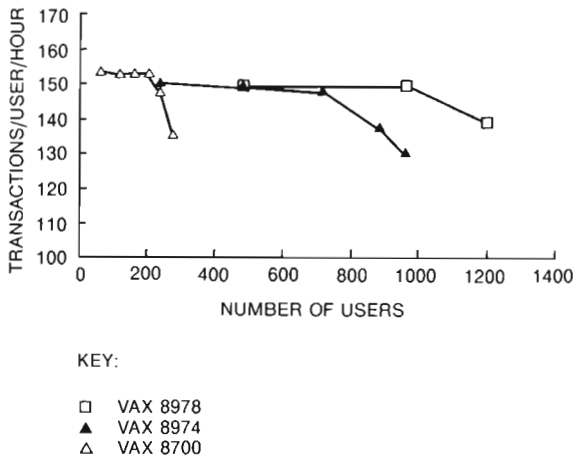


Figure 10 User Productivity

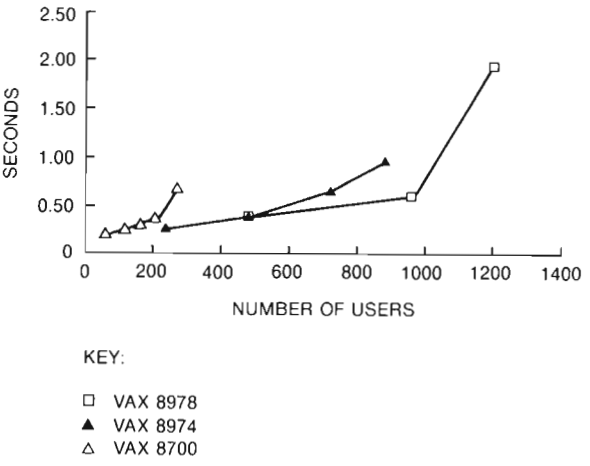


Figure 11 WIC Service Time

This figure shows that the maximum throughput per user for this workload is around 150 TPS for any configuration. This graph also indicates the number of users that can be supported by each system while maintaining a certain level of user productivity. For example, at 140 TPS, the 8700, 8974, and 8978 support 250, 850, and 1,200 users respectively. More users can be supported at lower user productivity levels.

Figure 10 also indicates the level of users at which one might consider switching to a larger system to maintain a certain level of user productivity. For example, to maintain a user productivity level of approximately 150 TPS, one must switch to a VAX 8974 system at around 240 users, and to a VAX 8978 system at around 720 users.

Mean Service Time

The VAX 8700 and VAX 8974 service times remained under one second for all user levels tested. The VAX 8978 service-time curve also followed this trend up to the 960-user level. However, after that level, the service time degraded quickly due to the large number of I/Os and queue lengths at the disks as the 1200-user level was approached. These patterns are shown in Figure 11.

ENQ Rate

So far, only user visible performance and some system behavior has been discussed. Now some of the cluster aspects of the systems are examined, mainly the locking activities.

As mentioned at the beginning of this paper, the WIC workload assumes full data-sharing (i.e., all the database files are shared by all users). This sharing involves locking and unlocking files and records every time they are accessed. The locking and unlocking operations are performed by system services called ENQ and DEQ. An ENQ request is serviced by the distributed lock manager, which examines outstanding locks to the resource and allows access if there is no conflict.

The SPM software records the the number of ENQs on a particular processor. The total ENQ rates at different user levels for different configurations were extracted from SPM data and graphed in Figure 12. This curve closely resembles the throughput curve, implying a strong correlation between locking activities and throughput. Around 26 ENQ operations were required on the average to perform each exchange.

Total Remote ENQ Rate

A remote ENQ occurs when the resource of interest is mastered by a process that runs on another processor in the cluster. Remote locks are more costly than local locks because additional interprocessor communication over the CI bus is required between the requesting and mastering nodes.

Figure 13 plots the remote ENQ rates against the total ENQ rates for different configurations.

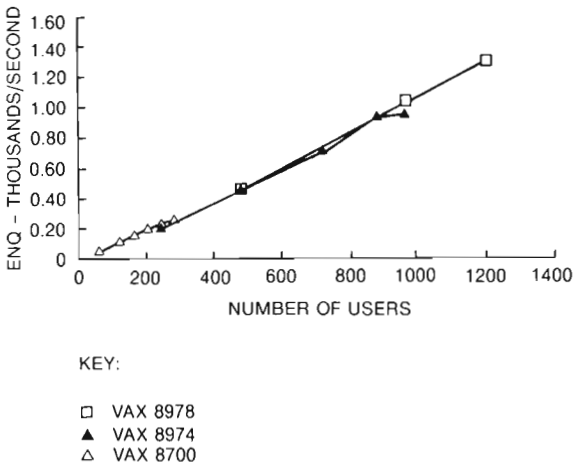


Figure 12 Total ENQ Rate

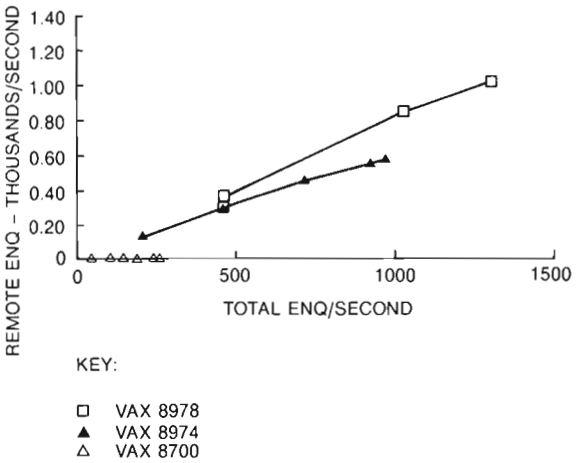


Figure 13 Remote versus Total ENQ Rates

The increasing slopes of the different curves indicate that the remote ENQ rate also increases with the number of processors in the system as well as with the total number of users. Generally, in an N-processor homogeneous distributed system in which all resources are equally accessed by all processors and all accesses require locking operations, the remote locking operations will equal $(N-1)/N$ times the total locking activity. This result occurs because each processor has an equal opportunity to master a particular resource. This relationship held in the case of the remote versus the total new ENQ rates observed in the VAX 8974 and VAX 8978 systems, in which the ratios were 75 percent and 87.5 percent respectively. Figure 13 shows, however, that on the average only 60 percent and 80 percent of the ENQs were remote for the 8974 and the 8978 respectively. These results occurred because the plotted ENQ rate includes the converted ENQ rate as well as the new ENQ rate; most converted ENQs were found to be local.

Interprocessor Communication

The communications between the processors are achieved by the Systems Communication Architecture by way of transmitting and receiving sequenced messages. Figure 14 shows the number of sequenced messages transferred between the processors every second. Most of these messages are generated by the distributed lock manager for clusterwide locking purposes.

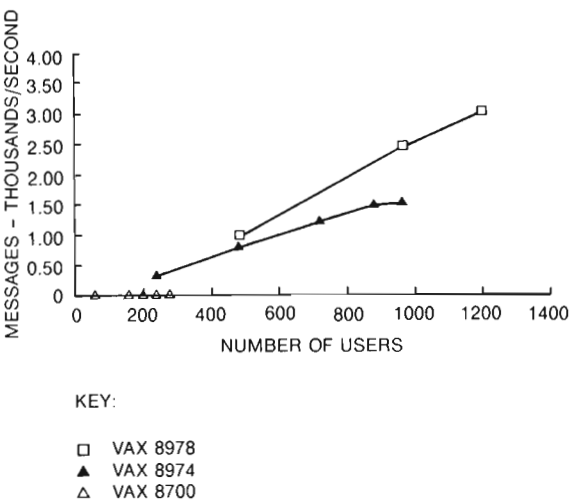


Figure 14 Message Rate between Processors

CI Traffic

The traffic on the CI consists of three packet types: datagrams, sequenced messages, and block transfer messages. In this application, datagrams were used only for error logging and therefore did not exist. Sequenced messages are used for communications between the processors and the HSC70 controllers. Most of these short packets are either packets between the distributed lock managers to perform clusterwide locking (discussed earlier) or packets between a processor and an HSC70 controller to request and response to I/O operations. Each I/O request to the disks

or tapes controlled by an HSC70 device requires a pair of messages to be exchanged between the processor and the controller. Block transfer messages are data packets for I/O operations. The transfer rates of each message type are recorded by the SPM software. Figure 15 plots the CI traffic against the number of users. The CI traffic, expressed in KB per second, is calculated from the data collected by the SPM software.

This figure shows that, in general, the CI bus is rather underutilized, peaking around 1,265KB per second at 1,200 users for the VAX 8978 system. This utilization is less than 15 percent of the raw bandwidth of a single CI wire, or 7.5 percent of the bandwidth on each CI path. It should be noted, however, that this data includes neither the extra bytes of the lower level protocol overhead nor the additional traffic incurred by retransmissions. Thus the actual CI utilization will be a little higher than these figures.

WIC Database Partitioning — Extended Study

The results presented in the previous section indicate that the application as currently implemented presented a problem with the disk I/O. More I/Os were being generated to several files, resulting in too many disk I/Os to several spindles. To reduce the number of I/Os, we partitioned both the application and the database, anticipating that the number of I/Os to each spindle would be reduced. This section summarizes the results from this study.

The main difference between this study and the previous one is the number of disk spindles used. This study used 24 spindles (6 SA482s), whereas the previous study used only 12 (3 SA482s). The throughputs achieved with this new configuration are plotted in Figure 16.

It is clear that with this configuration the VAX 8978 system performed much better with 24 spindles than with 12. The system achieved a peak throughput of 66 transactions per second with 1,600 users, which was 6.3 times the throughput of the VAX 8700 CPU. This result illustrates the importance of having a system balanced in regards to its processing power and I/O capacity.

Simulation of the WIC Workload

Based on the measurement data, a model called SIMwic was developed to describe the performance of VAX 8974/8978 systems under WIC, the multiuser, on-line transaction processing workload. WIC characterizes the on-line transaction processing of items (i.e., parts) that flow into and out of a warehouse and supports multiple concurrent access to the WIC database. The model structure of SIMwic is shown in Figure 17.

The following components of WIC were modeled in SIMwic:

- Users (who generate transactions)
- Lock messages
- CPUs
- Shared I/O passages (CI bus, HSC70 controller, channel)
- Disks

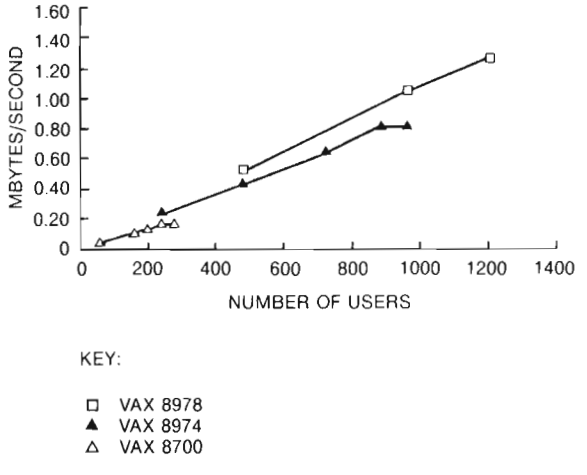


Figure 15 CI Traffic

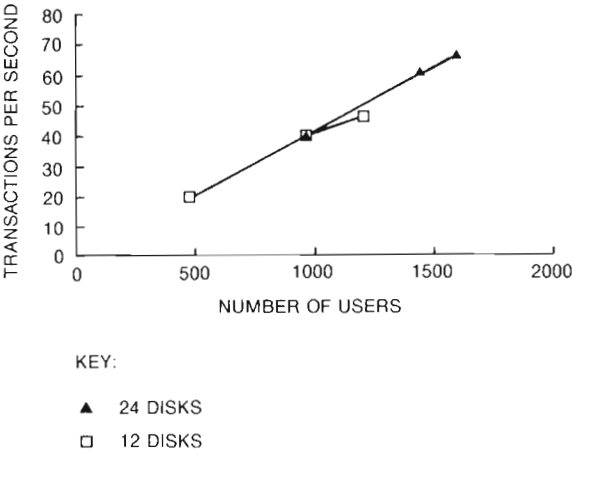


Figure 16 WIC Throughput for 12 versus 24 Disks

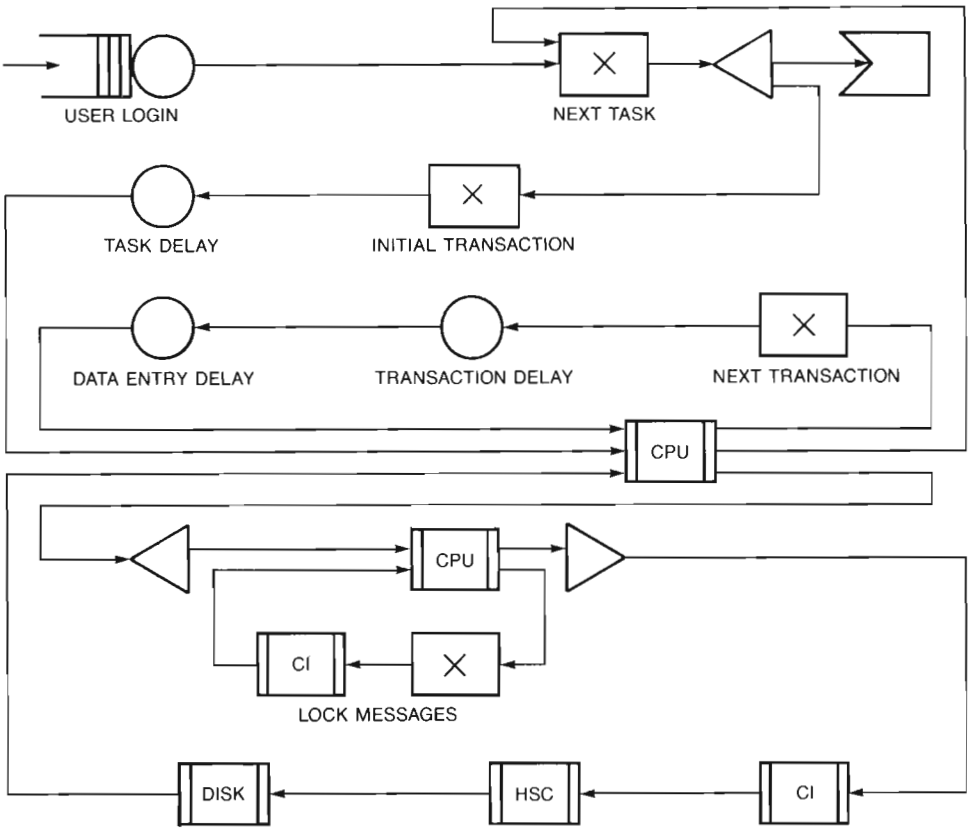


Figure 17 Model Structure of SIMwic

A user generates one task at a time to access the WIC database, each task consisting of several transactions. Each transaction uses the CPU for a certain amount of time and sends several I/O requests through the shared I/O passage to access the WIC database disks. Each I/O request will first send lock messages to ensure that the data is accessible and then initiate the I/O operations.

The following parameters are used by SIMwic to describe the interactions of the users, lock messages, CPUs, shared I/O passages, and disks:

- Intertask Delay, the delay after the completion of a task prior to the initiation of another task by the same user
- Intertransaction Delay, the delay after the completion of a transaction but prior to the initiation of the next transaction by the same task
- Task Mix, the percentages of each task type of the WIC workload
- Total Transaction, the total number of transactions for each task type
- Total Disk I/O, the total number of disk I/O for each transaction
- ProbDisk, the probability of selecting disk I for I/O
- CPU Delay, the CPU time to process a transaction on each visit
- Lock Delay, the CPU time to process lock messages due to an I/O request
- CI Delay, HSC delay, Disk Delay, and Channel Delay, delays due to data transfer and disk seeks

The values of these parameters were obtained from several sources, including workload specifications, direct measurements, other performance studies, and hardware specifications.

SIMwic was validated on measurements of CPU utilization, throughput, and disk I/O rates. The differences between simulated and direct-mea-

sured results were within five percent, as shown in Figure 18. The performance data collected were task life-cycle, throughput rate, CPU utilization, and disk I/O rate.

As discussed earlier, the performance of the VAX 8978 system under the WIC workload can be significantly improved by spreading the database over 24 disks instead of 12. SIMwic modeled such a database expansion and confirmed the performance improvements on the throughput, as plotted in Figure 19.

Summary

The performances of VAX 8978 and VAX 8974 systems were studied in two environments: a scientific, compute-intensive batch environment using the SCIENCE workload, and an on-line transaction processing environment using the WIC workload. These two environments were chosen to capture the range of the relative performances VAXcluster systems can achieve compared with the performance of a single system. Using both measurement and modeling approaches, it was shown that the 8974 has from 3.3 to 4.0 times the performance of a single VAX 8700 CPU, depending on the degree of file sharing, when there is no substantial bottleneck in the I/O subsystems. A 8978 was shown to have between 6.0 and 8.0 times the performance of the VAX 8700 CPU, again depending upon the application's characteristics, especially the amount of remote locking activity.

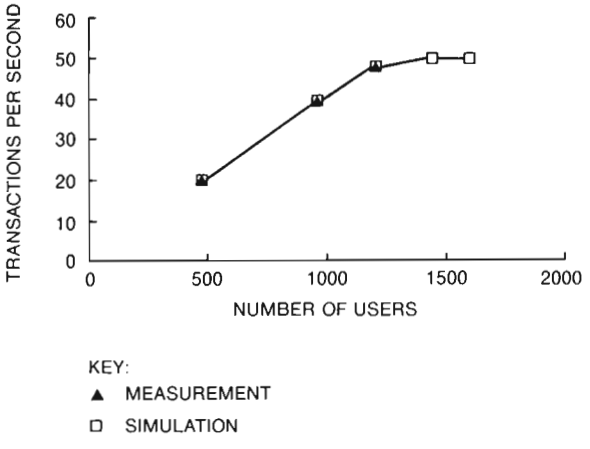


Figure 18 WIC Throughput — Model versus Measured

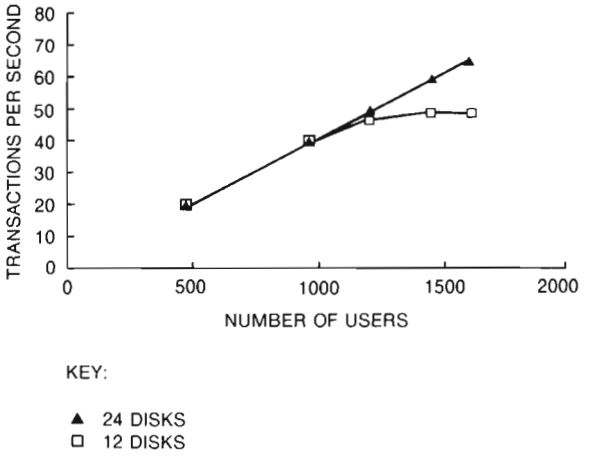


Figure 19 Model Results: Throughput with 12 versus 24 Disks

Acknowledgment

The authors wish to thank Joe Marconis for his support with the WIC workload, and Bill Youngs for providing a suit of scientific programs. Also thanks to Jory Tsai for the discussion on the VAX 8650 cluster model, to Hossein Hosseini for the WIC experiments, and to Ray Kopacko, who developed the SCIENCE workload and performed the experiments using it.

References

1. W. Snaman and D. Thiel, "The VAX/VMS Distributed Lock Manager," *Digital Technical Journal* (September 1987, this issue): 29–44.

2. D. Duffy, "The System Communication Architecture," *Digital Technical Journal* (September 1987, this issue): 22–28.

3. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems,* vol. 4, no. 2 (May 1986): 130–146.

4. P. Fleming and J. Wallace, "How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results," *CACM,* vol. 29, no. 3 (March 1986): 218–221.

5. F. Colon Osorio, N. Quaynor, D. Park, X. Cao, "Axiomatic Approach to Summarizing Benchmark Results," Annual Review/Reports, System Performance Group (1986).

Xi-Ren Cao
Nii N. Quaynor
Fernando C. Colon Osorio

CI Bus Arbitration Performance in a VAXcluster System

CI bus performance is difficult to evaluate with a conventional queuing network approach. Therefore, a new model, a generalized semi-Markov process, is used to model the process on the CI bus under its arbitration algorithm. This new model is implemented in a PASCAL program that is run for different configurations of VAXcluster systems. The simulation results demonstrate the properties of the arbitration algorithm. The results also suggest that a centralized control scheme could improve the CI utilization, and that some load-balance schemes can reduce the average response time. The method may be useful for designing other products.

This paper relates the study of performance of the CI bus in a VAXcluster environment. The cluster nodes (computers and storage controllers) are connected through a Star Coupler by a dual-path CI bus. An arbitration algorithm determines which node will be allowed to send packets over that CI bus. The performance of the CI bus may directly affect the cluster's performance, and studying the performance of the CI bus algorithm should yield some useful insights to enhance the designs of future computer-interconnect products.

Our approach is first to build a model that captures the main feature of the algorithm,[1] and then to consider other aspects as parameters of the model. The most important parameters are the length of the packets and the length of the quiet slot.

Because arbitration is complicated, a conventional queuing network model would be inadequate for modeling the CI process. For example, the CI bus could not be modeled as a server since packet transmission cannot start immediately after a request arrives, even if the CI bus were idle. Thus we propose another model based on the generalized semi-Markov process (GSMP). Moreover, this model may be useful for studying other processes in VAXcluster systems.

CI Bus Arbitration Algorithm

Here, we briefly review aspects of the CI arbitration related to the performance study. Reference 1 contains details of the CI bus arbitration.

A Simple Description of a CI Bus

Let us assume a VAXcluster system in which there are N nodes attached to a CI bus. Each node can send both information and acknowledge packets through the bus to any other node. Upon receiving an information packet, a node first checks the cyclical redundancy check (CRC) information in that packet. If the CRC succeeds, the receiving node will immediately send back to the transmitting node an acknowledge packet with either an acknowledgment (ACK) if the node accepts and stores the packet correctly, or a non-acknowledgment (NAK) if not. If the CRC fails, the node will send no response.

A time period, called the quiet slot, is reserved to guarantee the transmission of the acknowledge packet. The quiet slot (QS) is defined as the period of time needed to accommodate the time delay through a node's front-end logic, plus the round-trip cable and coupler delays for the longest path in a CI cluster installation. Only the node that generates the acknowledge packet for the information packet just received can grasp the CI bus during the quiet slot following the transmission of any information packet. Thus, as an approximation, the transmission time of the information packet may be extended to include the transmission time of the acknowledge packet.

After sending an information packet, the transmitting node waits for the length of an acknowledge time-out period. If that node receives an ACK during that period, the transmit is completed. Upon receiving a NAK or no response within the time-out period, however, the trans-

mitting node must retransmit the packet. The acknowledge time-out period is greater than the sum of one quiet slot, plus the CI bus turnaround time, plus the time to verify and accept the acknowledge packet at the transmitting node.

In addition, in any such "shared" multinode bus structure, the arbitration for use of the bus so as to avoid collisions is a critical element of the design. The CI bus architecture implements the distributed arbitration scheme discussed below.

CI Bus Arbitration

Two identical CI paths are used in a VAXcluster system, and all nodes are connected to both of them. Each node can randomly pick one path before transmitting an information packet. Once having chosen a path, the node will use it until an acknowledge packet from the destination node has been received. However, each node cannot transmit and receive simultaneously using two different paths. Figure 1 illustrates the structure of a VAXcluster system in which VAX CPUs and HSC devices are connected to one CI path.

Arbitration must be performed by all nodes prior to the transmission of any information packet. The acknowledge packet, following receipt of an information packet, does not require arbitration. This method is called a slotted-carrier sense multiple access (CSMA) protocol, also referred to as dual-count round robin. The following parameters are used in current VAXcluster systems:

- The clock unit (TCLK) is set at 114.28 nanoseconds (ns).
- The value of the quiet slot can range from 7 to 64 TCLKs, or 800 to 7,314 ns, depending on the cable length of the cluster. The QS for the for the discussion of this paper simulation is 1,143 ns.

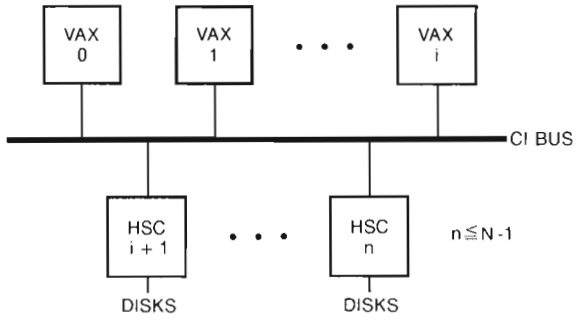


Figure 1 A Typical VAXcluster System

- The maximum number of nodes in the cluster, N, is 16 for the current algorithmic implementation.
- The ID numbers of the nodes are $I=0, 1, \ldots, N-1$, one for each node.

The arbitration algorithm operates as follows:

1. Upon starting a transmit operation, node I chooses randomly one of the CI paths and sets the value of its arbitration counter, C, to $N+I+1$.
2. In each TCLK period, the node determines whether or not the CI bus is busy. If it is busy, the arbitration counter will remain unchanged.
3. Once the node senses that the CI bus is not busy, it will start counting quiet slots. That is, the arbitration counter is set to $C-1$, and the node then waits for one QS period.

   If $C > 0$ at the end of one QS period, the node will inquire if the CI bus is busy. If it isn't busy, C is set to $C-1$, and the node waits during one additional QS period. If the CI bus is busy, the arbitration counter is set to another value that depends on the node ID.

   - If the CI bus is occupied by a node whose ID is greater than I, or if this is the node's first attempt to grasp the CI bus, then C is set to $N+I+1$ (i.e., the initial value of C for this node).
   - If the CI bus is occupied by a node whose ID is less than I and this is not the first attempt of node I to grasp the CI bus, then C is set to $I+1$. After the arbitration counter is reset, control returns to step 2 above.

   If $C=0$ at the end of the QS period, the node inquires again if the CI bus is busy. If so, the arbitration counter is set to another value that depends on the node ID, as explained just above. If the CI bus is not busy, the node inquires if a packet is being received from the other path.

   - If the node is receiving from the other path, C is reset to N, and control goes to step 2 above.
   - If the node is not receiving, it starts the transmission immediately.

Figure 2 shows a possible case of CI arbitration. This figure depicts a short history of the arbitration times for three nodes, labeled 2, 6, and 8. During the period $[0,t_1]$, the CI bus is transmitting a packet from some other node while both node 2 and node 6 have requested to transmit. The arbitration counters of these two nodes are set respectively to 19 (16+2+1) and 23 (16+6+1). At time t_1, the CI bus becomes idle, and nodes 2 and 6 both start counting quiet slots. At time t_2 ($t_2-t_1=19QS$), the arbitration counter of node 2 becomes zero; hence node 2 wins the bus. At this instant, the arbitration counter of node 6 is 4. After detecting that the bus has been captured by a node whose ID is less than its own, node 6 sets its arbitration counter number to 7 (6+1). (Assume that this is not the first attempt of node 6.)

The transmission of the packet from node 2 ends at t_4. Node 6 starts counting again at t_4 with an arbitration counter of 7 and wins the bus at t_6 ($t_6-t_4=7QS$). Figure 2 also shows that requests arrive at the ports of nodes 2 and 8 at t_3 and t_5 respectively. At t_6, the arbitration counter of node 2 becomes 19 (16+2+1) since the bus was won by a node whose ID is bigger than that of node 2. The arbitration counter of node 8 is set to 25 (16+8+1) since this is node 8's first attempt to occupy the CI bus.

For simplicity, we will study the properties of only one CI path in this report. The principle for studying two CI paths should be the same.

Some Preliminary Analysis

Although a complete analysis of the CI bus process is difficult, some preliminary analyses may help us to understand the properties of this process and perhaps validate the simulation results.

When two packets attempt to pass through the same path of the CI bus simultaneously, both packets will be destroyed. Therefore, packets can be passed successfully only if, before sending a packet, each node determines whether the CI bus is busy. Even with this check, two nodes can still send their packets simultaneously if each node detects at the same instant that the CI bus is idle. The situation is even worse because of the propagation time of a packet from the transmitting node to the detecting node.

The introduction of the QS concept into the arbitration algorithm almost eliminates the possibility of packet collisions when the CI bus is saturated. In this case nearly every transmit request will find the CI bus busy and must wait until the end of the transmission of the current packet. At the end of a transmission period from a node, denoted as I_0, all other nodes having an outstanding transmit request will start counting quiet slots simultaneously. The arbitration counters of nodes whose transmit requests are made during the transmission period have the form $N+I+1$. The arbitration counters of those nodes whose transmit requests were made in previous transmission periods have the form $N+I+1$ if $I > I_0$, or the form I if $I < I_0$. Thus at a given time, each node has a unique arbitration count. The node whose arbitration counter reaches zero first will grasp the CI bus.

After each transmission period, there is a short interval (16 quiet slots) in which no transmissions occur on the CI bus. However, every requesting node is still counting the quiet slots during this period. For example, suppose that in one QS, the smallest arbitration counter is

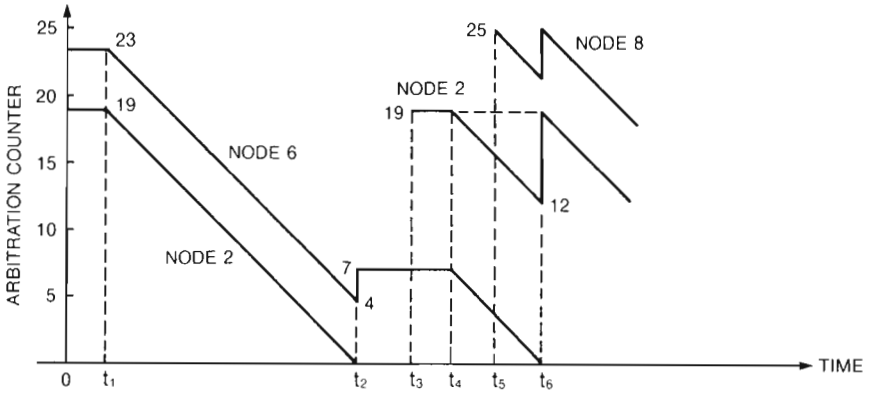


Figure 2 Arbitration among Three Nodes

$N+I+1$, and that in the same QS, node I initiates a transmit operation. In this case the arbitration counters of node I and the node whose arbitration counter is $N+I+1$ in that QS are always the same. Therefore, these two nodes could start to transmit at the same time, and a collision could occur, even though its probability is very small.

The CI bus could be considered as a server. From the arbitration scheme discussed above, however, customers do not start services immediately after arriving at the server, even if it is idle. One may argue that the arbitration time can be modeled by a separate server. In this case, however, the customer in that server does not have a fixed service time (the arbitration counter needs to be reset frequently). Therefore, the CI bus cannot be modeled as a standard queuing system. Fortunately, many stochastic processes exist that can be used to model real-world processes. One stochastic process, called the generalized semi-Markov process, has a characteristic very similar to the process on the CI bus under the above arbitration rules.

In the next section, we give an description of this process.

Generalized Semi-Markov Processes

The generalized semi-Markov process, or GSMP, is one of the most promising stochastic processes in operations research for modeling complex phenomena. GSMP was introduced by Matthes,[2] and investigated further by other researchers, among them Schassberger,[3] and Whitt.[4]

A GSMP can be described as follows. Let S and R be subsets of positive integers. We regard the elements s of subset S as possible states of the GSMP. Some events may occur at each state. R denotes the indices of all possible events that may occur during the evolution of a GSMP. All events that can occur in state s are denoted as set $E(s)$, which is a subset of R.

The system will stay in a state s until an event $i \in E(s)$ triggers a transition of the system to another state s'. Let $p(s',s,i)$ be the probability that the new state is s', given that event i triggers a transition from state s. An event can trigger a transition only at the end of its lifetime. Associated with each event i is a clock whose reading is denoted as c_i. The clock runs at a speed $r(s,i)$, which depends on both the event i and the state s. If at time 0 the clock is set to c_i, then at time t the reading of the clock will be $c'_i = c_i - r(s,i) \times t$. The lifetime of an event ends when the associated clock reading reaches zero. We assume $r(s,i) > 0$ for some $i \in E(s)$. When $r(s,i) = 0$ for $i \in E(s)$, event i is regarded as inactive in state s.

The events associated with state s' are in the set $E(s')$. The clock readings after the transition are determined as follows. New clock readings are independently generated for each $j \in N(s',s,i) \equiv E(s') - (E(s) - i)$. The new clock reading for event $j \in N(s',s,i)$ has a cumulative probability distribution, or c.p.d., of $F(x;s',j,s,i)$. For events in both $E(s)$ and $E(s')$, except for event i, the old clock readings are kept after the transition, i.e., for

$$j \in O(s',s,i) \equiv E(s') \cap (E(s) - i),\ c_j = c_j^{\bullet}\ (s,c).$$

For events in $E(s)$ but not in $E(s')$, the clocks are set equal to zero (i.e., if $j \in (E(s) - i) - E(s')$, then $c_j = \infty$ after the transition.)

For the purpose of modeling the CI process, the above scheme of determining the clock readings has to be modified slightly. We associate each event i with a set of events $H(i)$. Only for events in $j \in O(s',s,i) \equiv E(s') \cap \{E(s) - H(i)\}$ are old clock readings kept (i.e., $c_j = c_j^{\bullet}\ (s,c)$). For events in $N(s',s,i) \equiv E(s') - \{E(s) - H(i)\}$, new clock readings have to be assigned according to the c.p.d. $F(x;s',j,s,i)$. We call the process with this clock-reading assignment scheme a modified GSMP. A block diagram is shown in Figure 3.

The next transition occurs according to the same rules. These transitions describe the evolution of the system.

The Stochastic Process on the CI Bus

To describe the process on the CI bus, we use a continuous time domain as opposed to a discrete domain (i.e., we consider the clock unit 114.28 nanoseconds to be infinitesimally small compared with other event times, such as transmission times.) Furthermore, to make the problem tractable, we make the following stochastic assumptions:

- The transmission times required by every node are independent of each other.
- The times between two successive transmission requests are independent.
- The destinations of the transmitted packets are independent of the transmitting node and the transmission time.

Under the above assumptions, the CI system can be characterized by the following items:

- The number of nodes, N
- The cumulative distribution functions of the transmission time of each node, denoted as $F_i(x), i=0,1,\ldots N-1$
- The cumulative distribution functions of the time between two successive transmission re-quests of each node, denoted as $G_i(x), i=0,1,\ldots,N-1$
- The probability that a packet from node i will go to node j, denoted as $p_{i,j}$

The state x of the CI bus consists of the following elements:

- An index j, indicating the node that is transmitting a packet (We use $j=N$ to indicate that the CI bus is idle.)
- The number of transmission requests made by nodes $i=0,1,\ldots,N-1$, denoted as $n_0,n_1,\ldots,n_{N-1}$

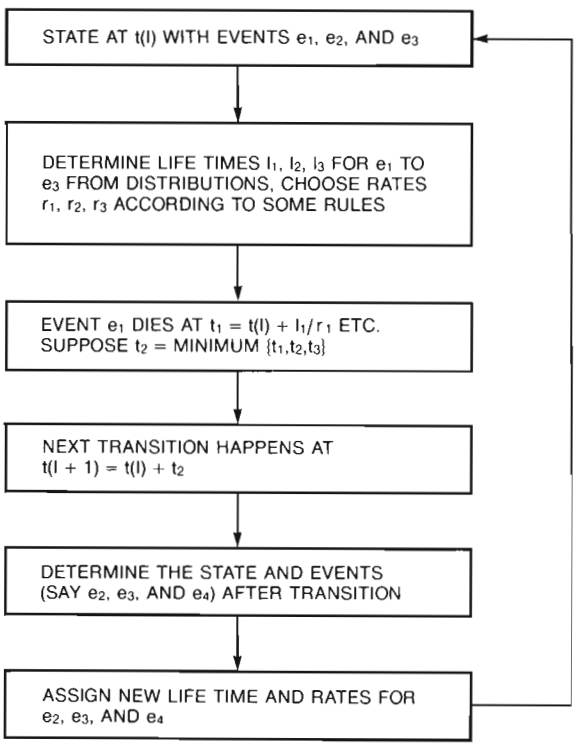


Figure 3 Block Diagram of Modified GSMP

- The residual transmission times of nodes $i=0,1,\ldots,N-1$, denoted as $b_0,b_1,\ldots,b_{N-1}$ (Except for node j, these values are the same as the transmission times.)
- The residual times between two transmission requests of nodes $i=0,1,\ldots,N-1$, denoted as $t_0,t_1,\ldots,t_{N-1}$
- The arbitration counters for the first request of nodes $i=0,1,\ldots,N-1$, denoted as $a_0,a_1,\ldots,a_{N-1}$ (Note that $a_i=0$.)

The process on one CI path can be described as a modified GSMP. Let L be the length of a QS period. The arbitration counters can be translated into continuous numbers $d_i=a_i\times L$. These continuous numbers can be viewed as clock readings. When $j=N$ (i.e., no packet is being transmitted on the CI bus), these clocks run at a rate $r=1$ until one of the readings reaches zero. When $j\neq N$, then these clocks run at a rate $r=0$; this means that when a server is transmitting packets, all arbitration counts do not change. The clock readings may jump to some other values at some transition times.

Now we can describe the process on one CI path. Let $s=\{j,n_0, n_1, \ldots, n_{N-1}\}$. Using the terminology of GSMP, we call s the state of the process. Associated with each state s, there are at most $3N$ events in $E(s)$ (i.e., the end of a transmission from each node, the grasp of the CI bus by each node, and a new request arrival at each node). The clock readings corresponding to these events are b_i, t_i, and d_i, $i=0,1,\ldots,N-1$. The clock rates are always one for all t_i, one for b_j, zero for b_i if $i\neq j$, and one for all d_i if $j\neq 0$ and zero for all d_i if $j=N$. For convenience, we also use b_i, t_i, and d_i to denote the corresponding events. Thus

$$E(s)=\{b_i,\ i{:}n_i>0;\ t_i, \text{all } i;\ d_i,\ i{:}\ n_i>0\}.$$

The only remaining work for specifying the GSMP on the CI path is to determine the clock rates $r(s,c)$, transition rules $p(s,s',i)$, and clock reading distributions $F(x,s',j,s,i)$. These can be done by examining carefully the arbitration scheme. The details can be found in reference 7.

We have modelled the CI process as a modified GSMP. This concept helps us to simplify the underlying mechanism of the process. This mechanism is no more complicated than state transition and clock readings. A simulation algorithm based on this model is given in the next section.

Simulation Algorithm

Although the GSMP concept looks sophisticated, its simulation is not difficult. In fact, the simulation of a GSMP consists mainly of two steps:

1. Use the clock readings and clock rates to determine the next transition time and the event that triggers this transition.
2. Determine the new state and the new clock readings after each transition.

Thus the GSMP model simplifies the concept of the mechanism of CI arbitration to these two steps.

The specific rules and distributions for determining the process on one CI path were described in detail in the previous sections. The simulation algorithm is given as follows:

1. Initialize the system.
   - Choose an initial state $s = \{j; n_0, n_1, \ldots, n_{N-1}\}$. n_i is the number of transmission requests of node i. j is the node transmitting, and $j = N$ means that the bus is idle.
   - Assign initial clock readings for events. For all nodes, the next transmission request happens at a time with distribution $G_i(x)$. The transmission time of a request on each node has a distribution $F_i(x)$. Set the arbitration counts according to the arbitration rule.
   - Set the value of the simulation clock, v, to 0.
2. Determine the clock rates for events according to the state s. The rates for the next transmission request are always 1. The rates for the transmission completion are 1 for node j, and 0 for all other nodes. The rates for arbitration counters are 1 for all nodes if $j = N$ (CI bus idle), 0 if $j \neq N$ (CI bus busy).
3. Using the clock rates, find the event whose clock reading reaches zero the earliest. This event triggers the transition. Set the simulation clock to the time when this reading reaches zero.
4. Using the transition probabilities, determine the next state of the process.
5. Assign new clock readings and rates for the new state. (This can be done as described in steps 1 and 2 above.)
6. If the terminating condition is not met, go to step 3. If the condition is met, stop the simulation.

There are some points that should be noted about this algorithm.

First, the model for two CI paths can be easily obtained by combining two models for one CI path and making the following modification. At the end of the arbitration of each node, the model checks to determine if the node is receiving from the other path. If not, the node starts transmission; otherwise, the model sets the arbitration count C of that node to N and starts the counting again.

The second point is, the ACK or NAK transmission times are included in the information packet transmission times (i.e., the distribution $F_i(x)$ describes the total transmission times of both an information packet and its ACK or NAK).

As mentioned earlier, we wrote a PASCAL program to implement this algorithm. The next two sections discuss the problems of choosing parameters for this model and the performance results obtained.

Choosing Parameters

As mentioned earlier, the maximum number of nodes in a CI-based VAXcluster system is 16; therefore, N is set to 16 in the simulation. QS is set to 1,143 ns.

The remaining problem is choosing the mean transmission times and the mean interrequest times, all of which depend on the node types and specific applications. In this simulation, these values are taken from the results of two previous experiments performed at Digital.[5,6] The first of those observes the CI packet traffic in a system running ASYNCQIO; the second measures the I/O performance of a system running IOX. (ASYNCQIO and IOX are both workload programs used for simulations.) The following is the mean interrequest and the mean transmission times of these two experiments; we use them as parameters in our simulation.

For ASYNCQIO, we have:

- The mean interrequest time of a VAX 8600 CPU with a CI780 bus is $r_{V,1} = 8{,}300$ microseconds (μs).
- The mean transmission time is $s_{V,1} = 6.4$ μs.
- The mean interrequest time of the HSC device is $r_{H,1} = 1{,}400$ μs.

- The mean transmission time of a packet from an HSC device is $s_{H,1}=60.5$ μs.

For IOX, we have:

- The mean interrequest time of a VAX 8600 CPU with a CI780 bus is $r_{V,2}=22{,}900$ microseconds (μs).
- The mean interrequest time of the HSC device is $r_{H,2}=3{,}800$ μs.

Since we assume that IOX reads the same number of blocks per request as ASYNCQIO, the mean transmission times $s_{V,2}$ and $s_{H,2}$ are the same as $s_{V,1}$ and $s_{H,1}$.

These values are obtained by assuming that the VAX CPU runs only one stream of ASYNCQIO or IOX on one disk. If the CPU runs m streams simultaneously, it is reasonable to take $r_{V,i}/m$ and $r_{H,i}/m$, for $i=1,2$, as the mean interrequest times.

Finally, both ASYNCQIO and IOX are I/O intensive workloads. Therefore, the simulations described in the next section, using the data derived from these two workloads, represent the performance of I/O intensive programs. The calculations here just yield reasonable values for parameters.

Simulation Results

The mean values obtained in the previous section were used in the simulations. In each simulation run, half the nodes were VAX systems, the other half were HSC devices. Also, half the VAX systems ran ASYNCQIO, the other half ran IOX. To study the CI performance, we ran four sets of simulations.

The first set had 16 nodes, or eight VAX systems and eight HSC devices. The average transmission time for the VAX systems was 6.4 μs, and for the HSC devices 60.5 μs. The interrequest times were chosen to model the systems in which each VAX system runs from one to three streams of the I/O intensive workloads. Specifically, the mean interrequest times for a system running two streams are half those for a system running only one stream, and so forth.

The CI utilization rates of this first set of simulations are shown in Figure 4, the other results in Table 1. The CI bus transmits packets during busy time, arbitration occurs during arbitration time, and the bus is idle during idle time. Idle time does not include any arbitration time. The busy, idle, and arbitration rates are the ratios of

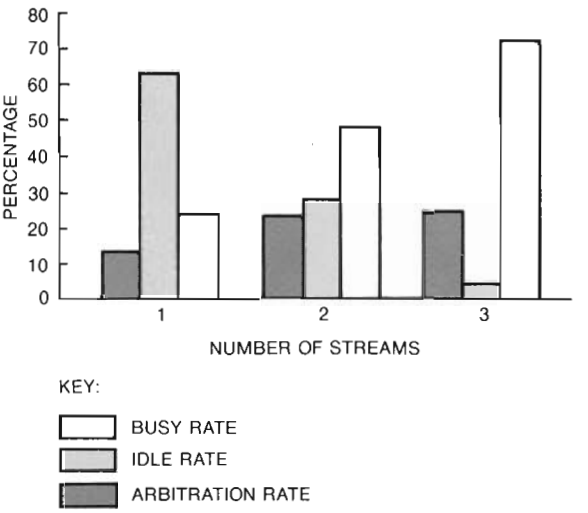


Figure 4 CI Performance for First Simulation

Table 1 First Set of Results

Simulation	1.1	1.2	1.3
No. of nodes: n	16	16	16
No. of streams	1	2	3
s_1 to s_8 – μseconds	6.40	6.40	6.40
s_9 to s_{16}	60.50	60.50	60.50
r_1 to r_4 – μseconds	8,300.00	4,150.00	2,800.00
r_5 to r_8	22,900.00	11,450.00	7,600.00
r_9 to r_{12}	1,400.00	700.00	470.00
r_{13} to r_{16}	3,800.00	1,900.00	1,270.00
Total time – seconds	43.79	21.90	14.69
Busy time – seconds	10.56	10.56	10.57
Idle time	27.27	5.95	0.42
Arbitration time	5.96	5.38	3.70
Busy rate – %	24	48	72
Idle rate	62	27	3
Arbitration rate	14	25	25
Arbitration/busy ratio	0.58	0.52	0.35
Response time – μseconds			
RE_1	46	80	213
RE_2	48	86	235
RE_3	48	86	246
RE_4	50	92	261
RE_5	51	95	256
RE_6	55	95	256
RE_7	57	97	265
RE_8	61	101	269
RE_9	115	181	1,015
RE_{10}	120	212	1,179
RE_{11}	127	238	1,171
RE_{12}	135	269	1,281
RE_{13}	139	257	539
RE_{14}	143	263	545
RE_{15}	147	270	555
RE_{16}	150	278	552

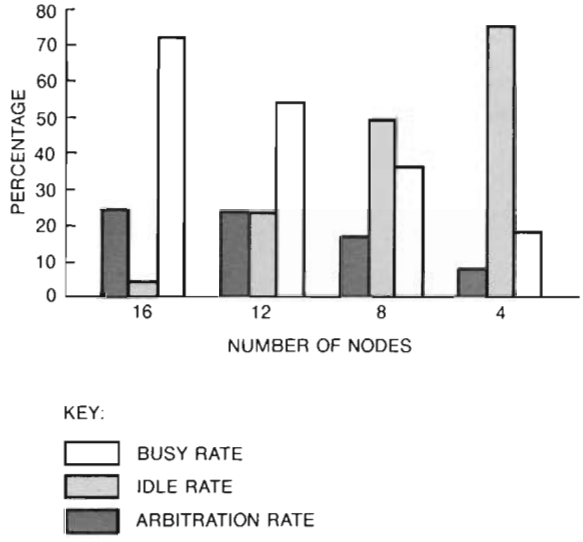


Figure 5 CI Performance for Second Simulation

Table 2 Second Set of Results

Simulation	2.1	2.2	2.3	2.4
No. of nodes: n	16	12	8	4
s_1 to $s_{n/2}$ – μseconds	6.40	6.40	6.40	6.40
$s_{n/2+1}$ to s_n	60.50	60.50	60.50	60.50
r_1 to $r_{n/4}$ – μseconds	2,800.00	2,800.00	2,800.00	2,800.00
$r_{n/4+1}$ to $r_{n/2}$	7,600.00	7,600.00	7,600.00	7,600.00
$r_{n/2+1}$ to $r_{3n/4}$	470.00	470.00	470.00	470.00
$r_{3n/4+1}$ to r_n	1,270.00	1,270.00	1,270.00	1,270.00
Total time – seconds	14.69	19.56	29.45	58.95
Busy time – seconds	10.57	10.55	10.55	10.56
Idle time	0.56	4.59	14.28	44.15
Arbitration time	3.56	4.41	4.61	4.25
Busy rate – %	72	54	36	18
Idle rate	4	23	49	75
Arbitration rate	24	23	16	7
Arbitration/busy ratio	0.33	0.43	0.44	0.39
Response time – μseconds				
RE_1	215	94	59	41
RE_2	237	99	60	41
RE_3	235	103	64	101
RE_4	245	103	64	108
RE_5	232	107	132	—
RE_6	235	110	153	—
RE_7	243	218	158	—
RE_8	239	249	162	—
RE_9	1,002	283	—	—
RE_{10}	1,086	248	—	—
RE_{11}	1,037	255	—	—
RE_{12}	1,091	258	—	—
RE_{13}	471	—	—	—
RE_{14}	475	—	—	—
RE_{15}	482	—	—	—
RE_{16}	481	—	—	—

the busy, idle, and arbitration times to the total time respectively.

From these results, we can see that the arbitration time takes about 23 to 24 percent of the total time if the CI bus is busy for more than 50 percent of the total time. The ratio of arbitration time to busy time decreases as the busy rate increases. We can also see that the response time is somewhat sensitive to the interrequest time. HSC controllers have a longer response time than VAX CPUs since the interarrival times of the controllers are shorter. The results also reveal that while the arbitration is almost fair for all nodes, some very small degree of unfairness still exists. For example, nodes 13 to 16 have the same mean interrequest and transmission times; however, the response times increase slightly as the ID number of the node increases. These properties will be explained later. Of course, such a small degree of unfairness will not affect the performance of the CI cluster.

The second set of simulations compared the performances of clusters with 4, 8, 12, and 16 nodes. The node ID numbers are 0 to 3 for the 4-node experiment, 0 to 7 for the 7-node experiment, and so forth. Each VAX CPU runs three streams of IOX or ASYNCQIO.

The results are shown in Figure 5 and Table 2.

These results confirm the properties observed in the first set of simulations. As far as the CI traffic is concerned, reducing the number of nodes is equivalent to decreasing the traffic intensity on the bus.

The third set of simulations examined the effect on performance of the lengths of packets transmitted on the CI bus. The average transmission times of a packet are assumed to be either 60.5, 60.5/2, 60.5/3, or 60.5/4 μs, depending on the number of streams. The results are shown in Figure 6 and Table 3.

As we expected, the ratio of arbitration time to busy time increases as the length of a packet decreases. If the average packet length is one-fourth of a block, the system will spend more time arbitrating than transmitting.

The fourth set of simulations kept the interrequest times of eight nodes constant at 1,000 μs, but varied the times of the other eight nodes from 300 to 1,000 μs. The parameters are listed in Table 4, and the results reported in Figure 7.

Figure 7 shows that if the mean interrequest times of nodes 1 to 4 and 9 to 12 are between 700 and 1,000 μs, the average response times of

all nodes will be very similar. If the interrequest times of these nodes decreases further, their response times increase rapidly. In this case a load balance scheme would be needed to achieve better performance.

CI Arbitration Properties

We can make the following observations from the simulation results:

- The response time increases rapidly if the CI bus is nearly saturated. This behavior is similar to that of a single-server queue.
- The arbitration algorithm is almost fair for all nodes. There is only a very small degree of unfairness. The response times of nodes with lower ID numbers are a little bit smaller than those of nodes with higher IDs.

  To explain this unfairness, let us consider two nodes, node 1 and node 10. Two cases in which node 1 gets higher priority than node 10 are given as follows:

  1. Assume that the CI bus is idle, and that node 10 requires a transmission at t_1 while node 1 requires a transmission at t_1+9QS. In this case, node 1 will win the bus despite the fact that node 10 submitted its request before node 1.
  2. Assume that the CI bus is busy, and that during this busy period both nodes 1 and 10 require transmissions. As soon as the CI bus becomes idle, both nodes will start counting quiet slots. In this case, node 1 will always win the bus whether or not it was the first to make the request.
- Under the current arbitration algorithm, the response times are sensitive to the interrequest times, especially when the CI bus is highly utilized. For example, in Simulation 1.3, the response times for two nodes with mean interrequest times of 470 and 7,600 μs are approximately 1,050 and 240 μs respectively.

  This result will occur because, under saturation, the arbitration is approximately a round-robin algorithm. If there are three requests in node 1 and six requests in node 2, the CI bus must serve the three requests in node 1 and the first three requests in node 2 before it can serve the last three requests in node 2. This algorithm gives higher priority to requests in node 1 than to those in node 2.

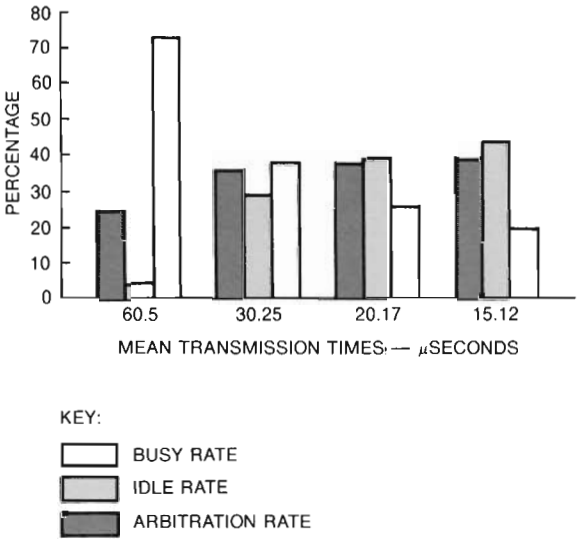


Figure 6 CI Performance for Third Simulation

Table 3 Third Set of Results

Simulation	3.1	3.2	3.3	3.4
No. of nodes: n	16	16	16	16
s_1 to s_8 – μseconds	6.40	6.40	6.40	6.40
s_9 to s_{16}	60.50	30.25	20.17	15.12
r_1 to r_4 – μseconds	2,800.00	2,800.00	2,800.00	2,800.00
r_5 to r_8	7,600.00	7,600.00	7,600.00	7,600.00
r_9 to r_{12}	470.00	470.00	470.00	470.00
r_{13} to r_{16}	1,270.00	1,270.00	1,270.00	1,270.00
Total time – seconds	14.69	14.69	14.69	14.69
Busy time – seconds	10.57	5.38	3.65	2.78
Idle time	0.56	4.16	5.59	6.34
Arbitration time	3.56	5.15	5.45	5.57
Busy rate – %	72	37	25	19
Idle rate	4	28	38	43
Arbitration rate	24	35	37	38
Arbitration/busy ratio	0.33	0.95	1.48	2.00
Response time – μseconds				
RE_1	215	56	41	36
RE_2	237	60	44	39
RE_3	235	63	46	41
RE_4	245	65	48	42
RE_5	232	65	49	44
RE_6	235	69	52	46
RE_7	243	69	54	48
RE_8	239	72	55	49
RE_9	1,002	111	76	63
RE_{10}	1,086	121	81	67
RE_{11}	1,037	131	87	72
RE_{12}	1,091	141	93	76
RE_{13}	471	137	93	77
RE_{14}	475	138	95	79
RE_{15}	482	143	98	82
RE_{16}	481	146	101	84

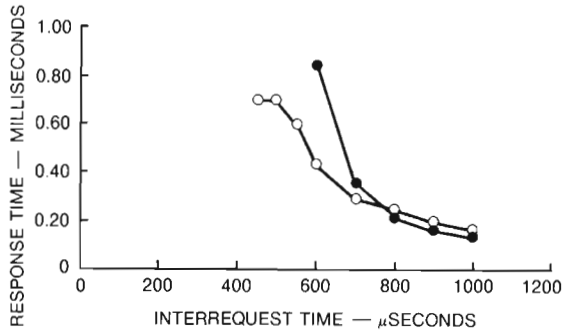


Figure 7 CI Performance for Fourth Simulation

Table 4 Fourth Set of Results

Simulation	4.1	4.2	4.3	4.4	4.5
No. of nodes: n	16	16	16	16	16
s_1 to s_8 – μseconds	6.40	6.40	6.40	6.40	6.40
s_9 to s_{16}	60.50	60.50	60.50	60.50	60.50
r_1 to r_4 – μseconds	1,000.00	900.00	800.00	700.00	600.00
r_5 to r_8	1,000.00	1,000.00	1,000.00	1,000.00	1,000.00
r_9 to r_{12}	1,000.00	900.00	800.00	700.00	600.00
r_{13} to r_{16}	1,000.00	1,000.00	1,000.00	1,000.00	1,000.00

Simulation	4.6	4.7	4.8	4.9	4.10
No. of nodes: n	16	16	16	16	16
s_1 to s_8 – μseconds	6.40	6.40	6.40	6.40	6.40
s_9 to s_{16}	60.50	60.50	60.50	60.50	60.50
r_1 to r_4 – μseconds	500.00	450.00	400.00	350.00	300.00
r_5 to r_8	1,000.00	1,000.00	1,000.00	1,000.00	1,000.00
r_9 to r_{12}	500.00	450.00	400.00	350.00	300.00
r_{13} to r_{16}	1,000.00	1,000.00	1,000.00	1,000.00	1,000.00

- Figure 7 shows the effect of the relative interrequest time on the response times. The response times of nodes 5 to 9 increase rapidly when their interrequest time is between 50 and 60 percent of the time for nodes 1 to 4.
- The results of the first and second sets of simulations show that the higher the CI busy rate, the smaller the total arbitration time. For example, in Simulation 1.1, the total arbitration time for 200,000 requests is 5.86 seconds, while that figure in Simulation 1.2 is 5.11 seconds.

  If the CI bus rate is low, the average transmission request from node I will have to wait an arbitration time of $(N+I+1)QS$. If the CI busy rate is high, however, each request can always find some node whose ID is lower and which can occupy the CI bus earlier. In this case the average request spends only $(I+1)QS$ on arbitration.
- In the third simulation, the arbitration time rate increases from 0.24 for an average packet length of one block to 0.38 for a length of one-fourth of a block. The absolute value of arbitration time also increases. This result occurs because the arbitration time is the same for packets with different lengths.

Conclusion

This paper describes the performance of the algorithm for CI bus arbitration as measured by a generalized semi-Markov process model. The simulation results show the following:

- The arbitration algorithm is almost fair to all nodes.
- The ratio of arbitration to busy times depends on the average length of packets transmitted; the smaller the length, the bigger this ratio.
- The ratio of arbitration to busy times also depends on the traffic intensity; the larger the intensity, the smaller the ratio.
- The response times of packets at a node are sensitive to its I/O rate compared to other nodes; the higher the rate, the longer its response time.
- Because of the arbitration time, the CI bus is not fully utilized. In experiment 1.3, the effective bandwidth for one path of the CI bus is about 75 percent. This effective bandwidth also depends on the average length of packets.

The results indicate where problems can be anticipated, especially when the CI bus is highly utilized, and suggest some ways to improve CI performance.

Acknowledgments

The authors are indebted to Jory Tsai for discussing the possibility of using PAWS to implement the GSMP model, and to Hossein Hosseini for assistance in preparing the report.

References

1. V. Boaen et al., "Computer Interconnect Specification," Digital Equipment Corporation Standard 161-0, 1986.

2. K. Matthes, "Zur Theorie der Bedienungsprozesse," *Transactions of the Third Prague Conference on Information Theory* (1962).

3. R. Schassberger, "Insensitivity of Steady-state Distributions of Generalized Semi-Markov Processes, Part I," *Annuals of Probability 5.* (1977): 81–99.

4. W. Whitt, "Continuity of Generalized Semi-Markov Processes," *Mathematics of Operations Research*, vol. 5, no. 4 (1980): 494–501.

5. B. Murray, "CI Traffic Observations: A Comparison of the CI780, CIBCI, and CIBCA," Digital Equipment Corporation Internal Technical Memorandum (October 1986).

6. X. Cao and H. Hosseini, "I/O Properties of a VAXcluster: Part I," Digital Equipment Corporation Internal Technical Memorandum (October 1986).

7. X. Cao, N. Quaynor, and F. Colon Osorio, "CI Bus Arbitration Performance in a VAXcluster," Digital Equipment Corporation Internal Technical Memorandum (March 1987).

digital™

ISBN 1-55558-004-1