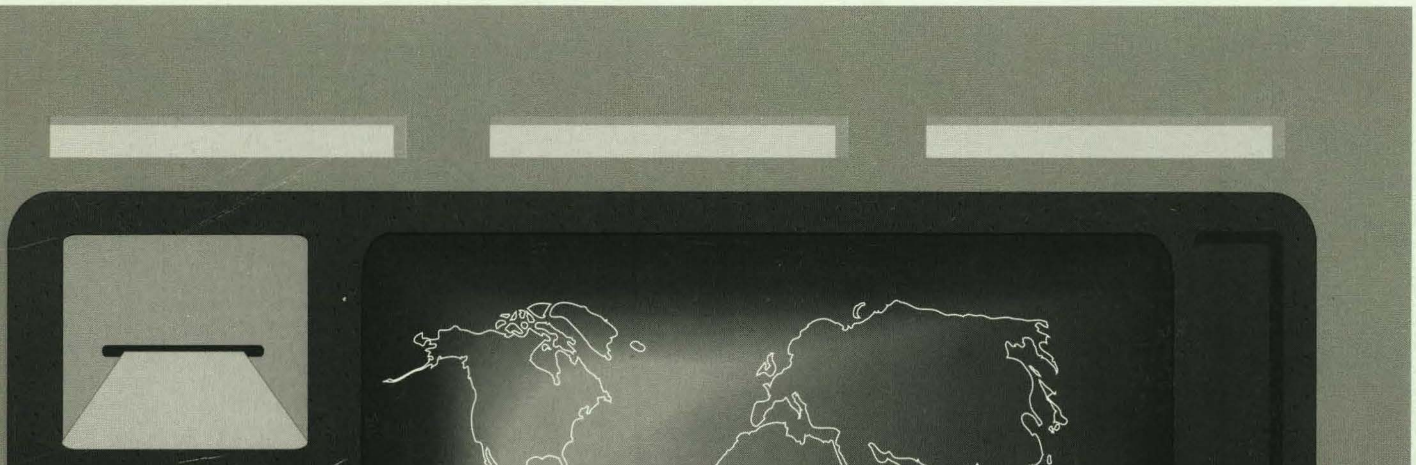
Transaction Processing, Databases, and Fault-tolerant Systems

Digital Technical Journal

Digital Equipment Corporation



Volume 3 Number 1

Winter 1991

Editorial
Jane C. Blake, Editor
Kathleen M. Stetson, Associate Editor

Circulation
Catherine M. Phillips, Administrator
Suzanne J. Babineau, Secretary

Production
Helen L. Patterson, Production Editor
Nancy Jones, Typographer
Peter Woodbury, Illustrator

The *Digital Technical Journal* is published quarterly by Digital Equipment Corporation, 146 Main Street MLO1-3/B68, Maynard, Massachusetts 01754-2571. Subscriptions to the Journal are $40.00 for four issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to The *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to DTJ@CRL.DEC.COM. Single copies and back issues are available for $16.00 each from Digital Press of Digital Equipment Corporation, 12 Crosby Drive, Bedford, MA 01730-1493.

Digital employees may send subscription orders on the ENET to RDVAX::JOURNAL or by interoffice mail to mailstop MLO1-3/B68. Orders should include badge number, cost center, site location code and address. All employees must advise of changes of address.

Comments on the content of any paper are welcomed and may be sent to the editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-F588E-DP

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

Cover Design
Transaction processing is the common theme for papers in this issue. The automatic teller machine on our cover represents one of the many businesses that rely on TP systems. If we could look behind the familiar machine, we would see the products and technologies — here symbolized by linked databases — that support reliable and speedy processing of transactions worldwide.

The cover was designed by Dave Bryant of Digital's Media Communications Group.

Contents

Editor's Introduction

Jane C. Blake
Editor

Digital's transaction processing systems are integrated hardware and software products that operate in a distributed environment to support commercial applications, such as bank cash withdrawals, credit card transactions, and global trading. For these applications, data integrity and continuous access to shared resources are necessary system characteristics; anything less would jeopardize the revenues of business operations that depend on these applications. Papers in this issue of the Journal look at some of Digital's techologies and products that provide these system characteristics in three areas: distributed transaction processing, database access, and system fault tolerance.

Opening the issue is a discussion of the architecture, DECdta, which ensures reliable interoperation in a distributed environment. Phil Bernstein, Bill Emberton, and Vijay Trehan define some transaction processing terminology and analyze a TP application to illustrate the need for separate architectural components. They then present overviews of each of the components and interfaces of the distributed transaction processing architecture, giving particular attention to transaction management.

Two products, the ACMS and DECintact monitors, implement several of the functions defined by the DECdta architecture and are the twin topics of a paper by Tom Speer and Mark Storm. Although based on different implementation strategies, both ACMS and DECintact provide TP-specific services for developing, executing, and managing TP applications. Tom and Mark discuss the two strategies and then highlight the functional similarities and differences of each monitor product.

The ACMS and DECintact monitors are layered on the VMS operating system, which provides base services for distributed transaction management. Described by Bill Laing, Jim Johnson, and Bob Landau, these VMS services, called DECdtm, are an addition to the operating system kernel and address the problem of integrating data from multiple systems and databases. The authors describe the three DECdtm components, an optimized implementation of the two-phase commit protocol, and some VAXcluster-specific optimizations.

The next two papers turn to the issues of measuring TP system performance and of sizing a system to ensure a TP application will run efficiently. Walt Kohler, Yun-Ping Hsu, Tom Rogers, and Wael Bahaa-El-Din discuss how Digital measures and models TP system performance. They present an overview of the industry-standard TPC Benchmark A and Digital's implementation, and then describe an alternative to benchmark measurement — a multilevel analytical model of TP system performance that simplifies the system's complex behavior to a manageable set of parameters. The discussion of performance continues but takes a different perspective in the paper on sizing TP systems. Bill Zahavi, Fran Habib, and Ken Omahen have written about a methodology for estimating the appropriate system size for a TP application. The tools, techniques and algorithms they describe are used when an application is still in its early stages of development.

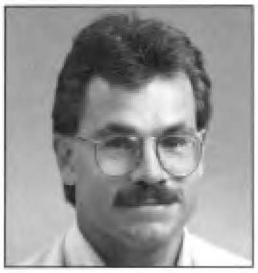
High performance must extend to the database system. In their paper on database availability, Ananth Raghavan and T.K. Rengarajan examine strategies and novel techniques that minimize the affects of downtime situations. The two databases referenced in their discussion are the VAX Rdb/VMS and VAX DBMS systems. Both systems use a database kernel called KODA, which provides transaction capabilities and commit processing. Peter Spiro, Ashok Joshi, and T.K. Rengarajan explain the importance of commit processing relative to throughput and describe new designs for improving the performance of group commit processing. These designs were tested, and the results of these tests and the authors' observations are presented.

Equally as important in TP systems as database availability is system availability. The topic of the final paper in this issue is a system designed to be continously available, the VAXft 3000 fault-tolerant system. Authors Bill Bruckert, Carlos Alonso, and Jim Melvin give an overview of the system and then focus on the four-phase verification strategy devised to ensure transparent system recovery from errors.

I thank Carlos Borgialli for his help in preparing this issue and for writing the issue's Foreword.

Jane Blake

Biographies



Carlos Alonso A principal software engineer, Carlos Alonso is a team leader for the project to port the System-V operating system to the VAXft 3000. Previously, he was the project leader for various VAXft 3000 system validation development efforts. As a member of the research group, Carlos developed the test bed for evaluating concurrency control algorithms using the VMS Distributed Lock Manager, and he designed the prototype alternate lock rebuild algorithm for cluster transitions. He holds a B.S.E.E. (1979) from Tulane University and an M.S.C.S. (1980) from Boston University.

Wael Hilal Bahaa-El-Din Wael Bahaa-El-Din joined Digital in 1987 as a senior consultant to the Systems Performance Group, Database Systems. He has led a number of studies to evaluate performance database and transaction processing systems under response time constraints. After receiving his Ph.D. (1984) in computer and information science from Ohio State University, Wael spent three years as an assistant professor at the University of Houston. He is a member of ACMS and IEEE, and he has written numerous articles for professional journals and conferences.

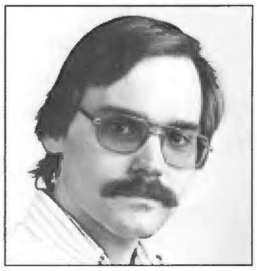
Philip A. Bernstein As a senior consultant engineer, Philip Bernstein is both an architectural consultant in the Transaction Processing Systems Group and a researcher at the Cambridge Research Laboratory. Prior to joining Digital in 1987, he was a professor at Wang Institute of Graduate Studies and at Harvard University, a vice president at Sequoia Systems, and a researcher at the Computer Corporation of America. He has published over 60 papers and coauthored two books. Phil received a B.S. (1971) in engineering from Cornell University and a Ph.D. (1975) in computer science from the University of Toronto.

William F. Bruckert William Bruckert is a consulting engineer who joined Digital in 1969 after receiving a B.S.E.E. degree from the University of Massachusetts. He received an M.S.E.E./C.E. degree from the same university in 1981. Beginning as a worldwide product support engineer, Bill later worked on a number of DECsystem-10/20 designs. He developed the cache, memory, and I/O subsystem for the VAX 8600 processor and was the system architect of the VAX 8650 processor. His most recent role was as the architect of the VAXft 3000 system. Bill currently holds seven patents.

William T. Emberton As a principal software engineer, William Emberton is currently involved in the development of Queue Management Architecture. He is also involved in X/Open and POSIX TP Standards work and is a member of the team that is developing the overall DECtp product architecture. Previously, he worked on the initial versions of the DECdta architecture. Before coming to Digital in 1987, Bill held positions as Director of Software Development at National Semiconductor and Manager of Systems Development for International Retail Systems at NCR. He was educated at London University.

Frances A. Habib Fran Habib is a principal software engineer involved with the development of transaction processing workload characterization and sizing tools. Previously, Fran worked at Data General and GTE Laboratories as a management science consultant. She holds an M.S. in operations research from MIT and a B.S. in engineering and applied science from Harvard. Fran is a full member of ORSA and belongs to ACM, IEEE, and the ACM SIGMETRICS special interest group on modeling and performance evaluation of computer systems.

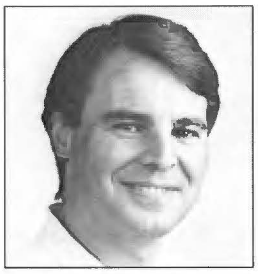
Yun-Ping Hsu Yun-Ping is currently a principal software engineer in the Transaction Processing Systems Performance and Characterization Group. He joined Digital in October 1987, after receiving his master's degree in electrical and computer engineering from the University of Massachusetts at Amherst. In his position, Yun-Ping is responsible for performance modeling and benchmark measurement of both ACMS- and DECintact-based TP systems. He also participated in the TPC Benchmark A standardization activity during 1989. He is a member of ACM and IEEE.



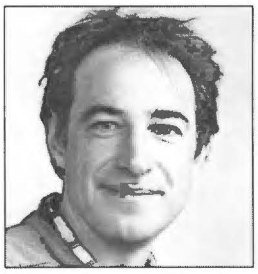
James E. Johnson A consulting software engineer, Jim Johnson has worked for the VMS Engineering Group since joining Digital in 1984. He is currently a project leader for VMS Engineering in Europe. Prior to this work, Jim led the RMS project, and after relocating to the UK three years ago, he was responsible for much of the design and implementation of the DECdtm services. At the same time, Jim was an active participant in the transaction management architecture review group. He has applied for a patent pertaining to the two-phase commit protocol optimization currently used in DECdtm services.

Ashok M. Joshi Ashok Joshi is a principal software engineer interested in database systems, transaction processing, and object-based programming. He is presently working on the KODA subsystem, which provides record storage for Rdb/VMS and DBMS software. For the Rdb/VMS project, he developed hash indexing and record placement features, and he worked on optimizing the lock protocols. Ashok came to Digital after receiving a bachelor's degree in electrical engineering from the Indian Institute of Technology, Bombay, and a master's degree in computer science from the University of Wisconsin, Madison.

Walter H. Kohler As a software engineering senior manager, Walt is responsible for TP system performance measurement and analysis and leads Digital's TP benchmark standards activities. Before joining Digital in 1988, Walt was a visiting scientist and technical consultant to Digital and a professor of electrical and computer engineering at the University of Massachusetts at Amherst. He holds B.S., M.S., and Ph.D. degrees in electrical engineering, all from Princeton University. Walt recently received the IEEE/CS Meritorious Service Award, and he has published over 25 technical articles.



William A. Laing William Laing is a senior consultant engineer based in Newbury, England. He is the technical leader for production systems support for the VMS operating system. During five years spent in the U.S., Bill was responsible for the design and initial development of symmetrical multiprocessing support in the VMS system. He joined Digital in 1981, after doing research on operating systems at Edinburgh University for nine years. Bill holds a B.Sc. (1972) in mathematics and computer science and an M.Phil. (1976) in computer science, both from Edinburgh University.



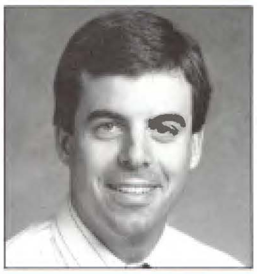
Robert V. Landau Principal software engineer Robert Landau is a member of the VMS Engineering Group, based in Newbury, England. He is currently the project leader of a VMS advanced development team investigating a high-performance, transaction-based, flat file system. Before joining Digital in 1987, Bob worked for a variety of software houses specializing in database-related products. He studied botany at London University and, subsequently, obtained a teaching qualification from Hereford College.

James M. Melvin As a principal design engineer, Jim was responsible for the specification of hardware error-handling mechanisms in the VAXft system and is presently an engineering project leader for future VAXft systems. He also specified and led the implementation of the hardware system simulation platform and the hardware design verification test plan. Jim joined Digital in 1984 and holds a B.S.E.E. (1984) and an M.S. (1989) in engineering management from Worcester Polytechnic Institute. He holds three patents on the VAXft 3000 system, all related to error handling in a fault-tolerant system.

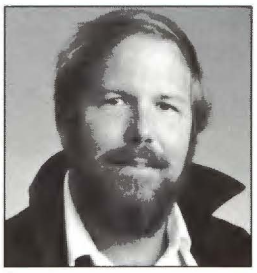
Kenneth J. Omahen A principal engineer, Kenneth Omahen is developing object-oriented queuing network solvers. He designed a variety of performance tools and performed design support studies which influenced a number of Digital products. Prior to joining Digital, Ken worked at Bell Telephone Laboratories, lectured at the University of Newcastle-Upon-Tyne, and was a faculty member at Purdue University. He received a B.S. degree in science engineering from Northwestern University and M.S. and Ph.D. degrees in information sciences from the University of Chicago.

Ananth Raghavan Since joining Digital in 1988, Ananth Raghavan has been a software engineer who has led projects for the KODA/Rdb Group. Previous to this position, he was a teaching assistant in the computer science department of the University of Wisconsin. Ananth holds a B.S. (1985) degree in mechanical engineering from the Indian Institute of Technology, Madras, and an M.S. (1987) degree in computer science from the University of Wisconsin, Madison. He has two patent applications pending for his work on undo and undo/redo database algorithms.

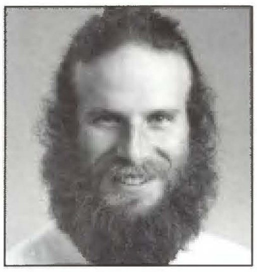
T. K. Rengarajan T. K. Rengarajan has been a member of the Database Systems Group since 1987 and works on the KODA software kernel for database management systems. He is involved in the support for WORM devices and global buffer management in the VAXcluster environment. His work in the areas of boundary element methods and database management systems is reported in several published papers and patent applications. Ranga holds an M.S. degree in computer-aided design from the University of Kentucky and an M.S. in computer science from the University of Wisconsin.



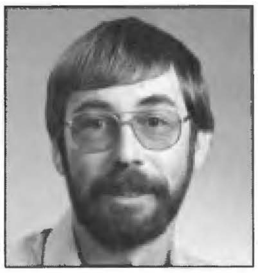
Thomas K. Rogers Thomas Rogers is a project leader for the Transaction Processing Systems Performance and Characterization Group. He is responsible for testing the VAX 9000 Model 210 system using the TPC Benchmark A standard. Prior to joining Digital in January 1988, Tom worked for Sperry Corporation as a technical specialist for the Northeast region. He received a bachelor of science degree in mathematical sciences in 1979 from Johns Hopkins University.



Thomas G. Speer As a principal software engineer in the DECtp/East Engineering Group, Thomas Speer is currently leading the DECintact V2.0 project. In this position, his major responsibility is defining the requirements for DECintact support of DECdtm services, client/server database access, and support for the DECforms product. Since joining Digital in 1981, Tom has worked on several development projects, including FORTRAN-10/20 and RMS-20. He holds degrees from Harvard University, Rutgers University, and Simmons College. He is a member of Phi Beta Kappa.



Peter M. Spiro Peter Spiro, a principal software engineer, is currently involved in optimizing database technology for RISC machines. He has worked on database facilities such as access methods, journaling and recovery, transaction protocols, and buffer management. Peter joined Digital in 1985, after receiving M.S. degrees in forest science and computer science from the University of Wisconsin. He has a patent pending for a method of database journaling and recovery, and he authored a paper for an earlier issue of the *Digital Technical Journal*. In addition, Peter enjoys the game of Ping-Pong.

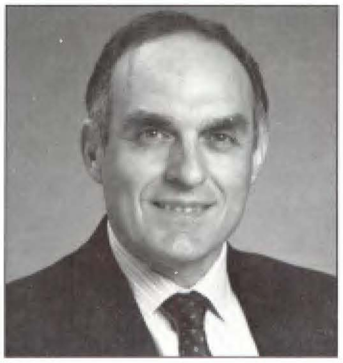


Mark W. Storm Consulting engineer Mark Storm was one of the original designers of the ACMS monitor, and he has been involved in the development of TP products for more than ten years. Currently, he is acting technical director for the East Coast Transaction Processing Engineering Group, as well as managing a small advanced development group. After joining Digital in 1976, Mark worked on COBOL compilers for the PDP-11 systems and developed the first native COBOL compiler for the VAX computer. He holds a B.S. (with honors) in computer science from the University of Southern Mississippi.

Vijay Trehan Since joining Digital in 1978, Vijay Trehan has contributed to several architecture projects. He is the technical director responsible for DECtp architecture, design, and standards work. Prior to this assignment, Vijay was the architect for the DECdtm protocol, architect for the DDIS data interchange format, and initiator of work on the DDIF document interchange format and compound document strategy. He holds a B.S. (1972) in mechanical engineering from the Indian Institute of Technology and an M.S. (1974) in operations research from Syracuse University.

William Z. Zahavi As an engineering manager, Bill is responsible for the design and development of predictive sizing tools for transaction processing applications. Before joining Digital in 1987, he was a technical consultant for Sperry Corporation, specializing in systems performance analysis and capacity planning. Bill received an M.B.A. from Northeastern University and a B.S. in mathematics from the University of Virginia. He is an active member of the Computer Measurement Group, and frequently presents at CMG conferences.

Foreword



Carlos G. Borgialli
Senior Manager, DECtp Software Engineering

Transaction processing is one of the largest, most rapidly growing segments of the computer industry. Digital's strategy is to be a leader in transaction processing, and toward that end we are making technological advances and delivering products to meet the evolving needs of businesses that rely on transaction processing systems.

Because of the speed and reliability with which transaction processing systems capture and display up-to-date information, they enable businesses to make well-informed, timely decisions. Industries for which transaction processing systems are a significant asset include banking, laboratory automation, manufacturing, government, and insurance. For these industries and others, transaction processing is an information lifeline that supports the achievement of daily business objectives and in many instances provides a competitive advantage.

Many older transaction processing systems on which businesses rely are centralized and tied to a particular vendor. A great deal of money and time has been invested in these systems to keep pace with business expansion. As expansion continues beyond geographic boundaries, however, the centralized, single-vendor transaction processing systems are less and less likely to offer the flexibility needed for round-the-clock, reliable, business operations conducted worldwide. Transaction processing technology therefore must evolve to respond to the new business environment and at the same time protect the investment made in existing systems.

Our research efforts and innovative products provide the transaction processing systems that businesses need today. The demand for distributed rather than centralized systems has focused attention on system management. Queuing services, highly available systems, heterogeneous environments, security services, and computer-aided software engineering (CASE) are a few examples of areas in which research and advanced development efforts have had and will continue to have a major impact on the capabilities of transaction processing systems.

Transaction processing solutions require the application of a wide range of technology and the integration of multiple software and hardware products: from desktop to mainframe; from presentation services and user interfaces to TP monitors, database systems, and computer-aided software engineering tools; from optimization of system performance to optimization of availability. Making all of this technology work well together is a great challenge, but a challenge Digital is uniquely positioned to meet.

Digital ensures broad application of its transaction processing technology by defining an architecture, the Digital Distributed Transaction Architecture (DECdta). DECdta, about which you will read in this issue, defines the major components of a Digital TP system and the way those components can form an integrated transaction processing system. The DECdta architecture describes how data and processing are easily distributed among multiple VAX processors, as well as how the components can interoperate in a heterogeneous environment.

The DECdta architecture is based on the client/server computing model, which allows Digital to apply its traditional strengths in networking and expandability to transaction processing system solutions. In the DECdta client/server computing model, the client portion interacts with the user to create processing requests, and the server portion performs the data manipulation and computation to execute the processing request. This computing model facilitates the division of a TP system into small components in three ways. It allows for distribution of functions among VAX processors; it partitions the work performed by one or more of the components to allow for parallel processing; or it replicates functions to achieve higher availability goals. These options permit the customer to purchase the configuration that meets present needs, confident that the system will allow smooth expansion in the future.

Further, the DECdta architecture sets a direction for its evolution through different products in a coordinated manner. It provides for the cooperation and interoperation of components implemented on different platforms, and it supports the expansion of customer applications to meet growth requirements. The DECdta architecture is designed to work with other Digital architectures such as the Digital Network Architecture (DNA), the network application services (NAS), and the Digital database architecture (DDA). Moreover, the DECdta architecture supports industry standards that enable the portability of applications and their interoperation in a heterogeneous environment, such as the standard application programming interfaces being developed by the X/Open Transaction Processing Working Group and the IEEE POSIX. Standard wire protocols that provide for systems interoperation in a multivendor, heterogeneous environment are being developed by the International Standards Organization as part of the Open System Interconnection activities.

Among the products Digital has developed specifically for TP systems are the TP monitors. These monitors provide the system integration "glue," if you will. Rather than act as their own systems integrators, customers who use Digital's TP monitors are able to spend more time on solving business problems and less time on solving software integration problems, such as how to make forms and database products work together smoothly.

Digital's TP monitors run on all types of hardware configurations, including local area networks (LANs), wide area networks (WANs), and VAXcluster systems. The DECdta client/server computing model provides the necessary flexibility to change hardware configurations, thus allowing reconfiguration without the need for any source code changes.

The two TP monitors, DECintact and VAX ACMS, integrate vital Digital technologies such as the Digital Distributed Transaction Manager (DECdtm) and products such as Digital's forms systems (DECforms) and our Rdb/VMS or VAX DBMS database products. DECdtm uses the two-phase commit protocol to solve the complex problem of coordinating updates to multiple data resources or databases.

Major developments in Digital's database products have enhanced the strengths of its overall product offerings. The two mainstream database products noted above, Rdb/VMS and VAX DBMS, layer on top of a database kernel called KODA, thus providing data access independent of any data model. The services made available by KODA, besides its high performance, allow Digital's database products to efficiently support TP applications as well as to provide rich functionality for general-purpose database applications.

For those TP systems that require user interfaces, DECforms provides a device-independent, easy-to-use human interface and permits the support of multiple devices and users within a single application.

TP systems that require high availability or continuous operations are supported by the VAX family of hardware and software. The introduction of the fault-tolerant VAXft 3000 system, added to the successful VAXcluster system, allows for a high level of system availability. Performance needs also are being met by a combination of hardware resources, including the VAX 9000 system.

This combination of architecture, software, and hardware technology, and support for emerging industry standards places Digital in an excellent position to become the industry leader for distributed, portable transaction processing systems. The papers in this issue of the Journal provide a view of the key elements of Digital's distributed transaction processing technologies.

Many individuals, teams, organizations, and business partners are responsible for bringing Digital's TP vision to fruition. Their dedication, hard work, and creativity will continue to drive the development of new technologies that enhance our family of products and services.

Philip A. Bernstein
William T. Emberton
Vijay Treban

DECdta — Digital's Distributed Transaction Processing Architecture

Digital's Distributed Transaction Processing Architecture (DECdta) describes the modules and interfaces that are common to Digital's transaction processing (DECtp) products. The architecture allows easy distribution of DECtp products. In particular, it supports client/server style applications. Distributed transaction management is the main function that ties DECdta modules together. It ensures that application programs, database systems, and other resource managers interoperate reliably in a distributed system.

Transaction processing (TP) is the activity of executing requests to access shared resources, typically databases. A computer system that is configured to execute TP applications is called a TP system.

A transaction is an execution of a set of operations on shared resources that has the following properties:

- Atomicity. Either all of the transaction's operations execute, or the transaction has no effect at all.
- Serializability. The set of all operations that execute on behalf of the transaction appears to execute serially with respect to the set of operations executed by every other transaction.
- Durability. The effects of the transaction's operations are resistant to failures.

A transaction terminates by executing the commit or abort operation. Commit tells the system to install the effect of the transaction's operations permanently. Abort tells the system to undo the effects of the transaction's operations.

For enhanced reliability and availability, a TP application uses transactions to execute requests. That is, the application receives a request message (from a display, computer, or other device), executes one or more transactions to process the request, and possibly sends a reply to the originator of the request or to some other party specified by the originator.

TP applications are essential to the operation of many industries, such as finance, retail, health care, transportation, government, communications, and manufacturing. Given the broad range of applications of TP, Digital offers a wide variety of products with which to build TP systems.

DECtp is an umbrella term that refers to Digital's TP products. The goal of DECtp is to offer an integrated set of hardware and software products that supports the development, execution, and management of TP applications for enterprises of all sizes.

DECtp systems include software components that are specialized for TP, notably TP monitors such as the ACMS and DECintact TP monitors, and transaction managers such as the DECdtm transaction manager.[1,2] DECtp systems also require the integration of general-purpose hardware products (processors, storage, communications, and terminals) and software products (operating systems, database systems, and communication gateways). These products are typically integrated as shown in Figure 1.

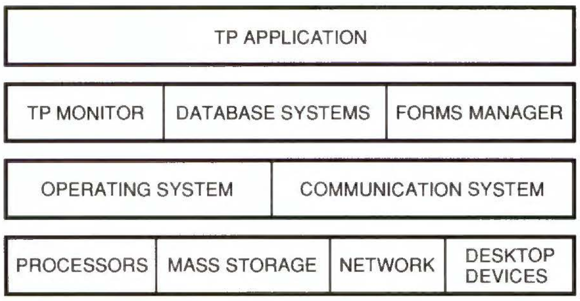


Figure 1 Layering of Products to Support a TP Application

Applications on DECtp systems can be designed using a client/server paradigm. This paradigm is especially useful for separating the work of preparing a request from that of running transactions. Request preparation can be done by a front-end system, that is, one that is close to the user, in which processor cycles are inexpensive and interactive feedback is easy to obtain. Transaction execution can be done by a larger back-end system, that is, one that manages large databases and may be far from the user. Back-end systems may themselves be distributed. Each back-end system manages a portion of the enterprise database and executes applications, usually ones that make heavy use of the database on that back end. DECtp products are modularized to allow easy distribution across front ends and back ends, which enables them to support client/server style applications. DECtp systems thereby simplify programming and reconfiguration in a distributed system.

Digital's Distributed Transaction Processing Architecture (DECdta) defines the modularization and distribution structure that is common to DECtp products. Distributed transaction management is the main function that ties this structure together. This paper describes the DECdta structure and explains how DECdta components are integrated by distributed transaction management.

Current versions of DECtp products implement most, but not all, modules and interfaces in the DECdta architecture. Gaps between the architecture and products will be filled over time. DECtp products that currently implement DECdta components are referenced throughout the paper.

TP Application Structure

By analyzing TP applications, we can see where the need arises for separate DECdta components. A typical TP application is structured as follows:

Step 1: The client application interacts with a user (a person or machine) to gather input, e.g., using a forms manager.

Step 2: The client maps the user's input into a request, that is, a message that asks the system to perform some work. The client sends the request to a server application to process the request.

A request may be direct or queued. If direct, the client expects a server to process the request right away. If queued, the client deposits the request in a queue from which a server can dequeue the request later.

Step 3: A server processes the request by executing one or more transactions. Each transaction may

a. Access multiple resources

b. Call programs, some of which may be remote

c. Generate requests to execute other transactions

d. Interact with a user

e. Return a reply when the transaction finishes

Step 4: If the transaction produces a reply, then the client interacts with the user to display that reply, e.g., using a forms manager.

Each of the above steps involves the interaction of two or more programs. In many cases, it is desirable that these programs be distributed. To distribute them conveniently, it is important that the programs be in separate components. For example, consider the following:

- The presentation service that operates the display and the application that controls which form to display may be distributed.

  One may want to off-load presentation services and related functions to front ends, while allowing programs on back ends to control which forms are displayed to users. This capability is useful in Steps 1, 3d, and 4 above to gather input and display output. To ensure that the presentation service and application can be distributed, the presentation service should correspond to a separate DECdta component.

- The client application that sends a request and the server application that processes the request may be distributed. The applications may communicate through a network or a queue.

  In Step 2, front-end applications may want to send requests directly to back-end applications or to place requests in queues that are managed on back ends. Similarly, in Step 3c, a transaction, T, may enqueue a request to run another transaction, where the queue resides on a different system than T. To maximize the flexibility of distributing request management, request management should correspond to a separate DECdta component.

- Two transaction managers that want to run a commit protocol may be distributed.

For a transaction to be distributed across different systems, as in Step 3b, the transaction management services must be distributed. To ensure that each transaction is atomic, the transaction managers on these systems must control transaction commitment using a common commit protocol. To complicate matters, there is more than one widely used protocol for transaction commitment. To the extent possible, a system should allow interoperation of these protocols.

To ensure that transaction managers can be distributed, the transaction manager should be a component of DECdta. To ensure that they can interoperate, their transaction protocol should also be in DECdta. To ensure that different commit protocols can be supported, the part of transaction management that defines the protocol for interaction with remote transaction managers should be separated from the part that coordinates transaction execution across local resources. In the DECdta architecture, the former is called a communication manager, and the latter is called a transaction manager.

Interoperation of transaction managers and resource managers, such as database systems, also affects the modularization of DECdta components. A transaction may involve different types of resources, as in Step 3a. For example, it may update data that is managed by different database systems. To control transaction commitment, the transaction manager must interact with different resource managers, possibly supplied by different vendors. This requires that resource managers be separate components of DECdta.

The DECdta Architecture

Having seen where the need for DECdta components arises, we are now ready to describe the DECdta architecture as a whole, including the functions of and interfaces to each component.

Most DECdta interfaces are public. Some of the public interfaces are controlled by official standards bodies and industry consortia; i.e., they are "open" interfaces. Others are controlled solely by Digital. DECdta interfaces and protocols will be published and aligned with industry standards, as appropriate.

DECdta components are abstract entities. They do not necessarily map one-to-one to hardware components, software components (e.g., programs or products), or execution environments (e.g., a single-threaded process, a multithreaded process, or an operating system service). Rather, a DECdta component may be implemented as multiple software components, for example, as several processes. Alternatively, several DECdta components may be implemented as a single software component. For example, an operating system or TP monitor typically offers the facilities of more than one DECdta component.

The following are the components of DECdta:

- An application program is any program that uses services of DECdta components.
- A resource manager manages resources that support transaction semantics.
- A transaction manager coordinates transaction termination (i.e., commit and abort).
- A communication manager supports a transaction communication protocol between TP systems.
- A presentation manager supports device-independent interactions with a presentation device.
- A request manager facilitates the submission of requests to execute transactions.

DECdta components are layered on services that are provided by the underlying operating system and distributed system platform, and are not specific to TP, as shown in Figure 2.

Application Program

We use the term application program to mean a program that uses the services provided by other DECdta components. An application program could be a customer-written program, a layered product, or a DECdta component.

In the DECdta architecture, we distinguish two special types of application program: request initiators and transaction servers. A request initiator is a DECdta component that prepares and submits a request for the execution of a transaction. To create a request, the request initiator usually interacts with a presentation manager that provides an interface to a device, such as a terminal, a workstation, a digital private branch exchange, or an automated teller machine.

A transaction server can demarcate a transaction, interact with one or more resource managers to access recoverable resources on behalf of the transaction, invoke other transaction servers, and respond to calls from request initiators.

For a simple request, a transaction server receives the request, processes it, and optionally returns a reply to the request initiator. A conversational request is like a simple request, except that while processing the request, the transaction

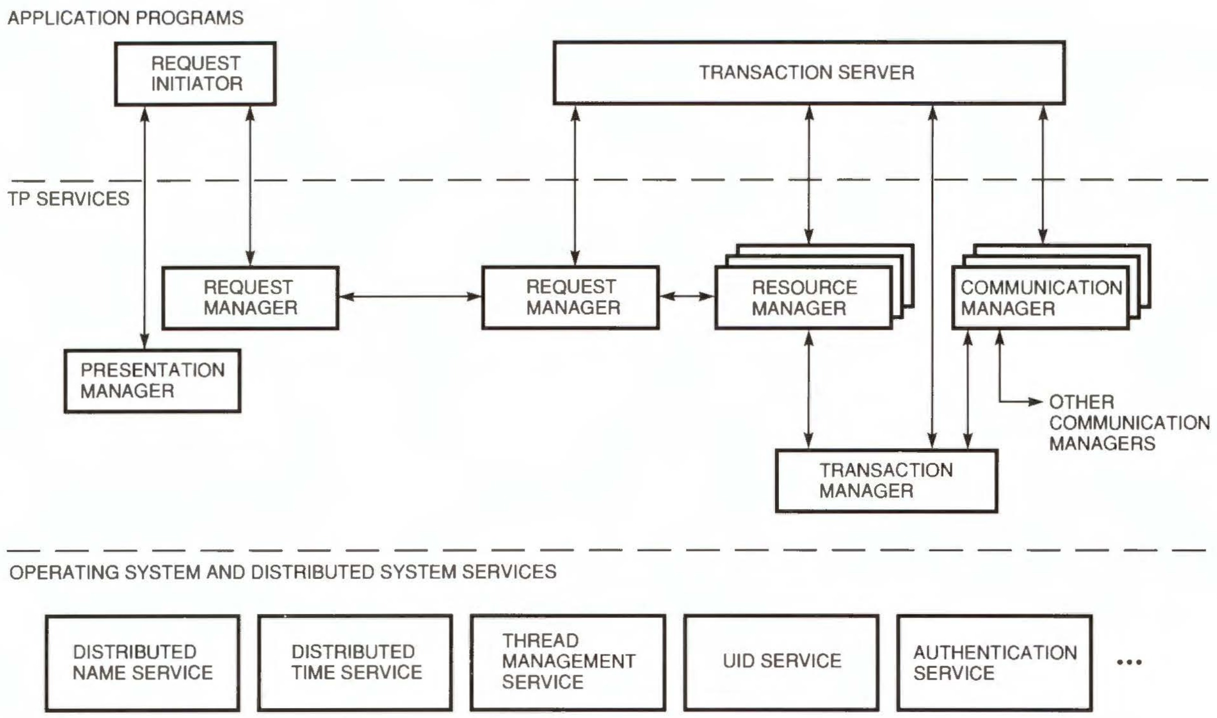


Figure 2 DECdta Components and Interfaces

server exchanges one or more messages with the user, usually through the request initiator.

In principle, a request initiator could also execute transactions (not shown in Figure 2). That is, the distinction between request initiators and transaction servers is for clarity only, and does not restrict an application from performing request initiation functions in a transaction. Architecturally, this amounts to saying that request initiation functions can execute in a transaction server.

Resource Manager

A resource manager performs operations on shared resources. We are especially interested in recoverable resource managers, those that obey transaction semantics. In particular, a recoverable resource manager undoes a transaction's updates to the resources if the transaction aborts. Other recoverable resource manager activities in support of transactions are described in the next section. In the rest of this paper, we use "resource manager" to mean "recoverable resource manager."

In a TP system, the most common kind of resource manager is a database system. Some presentation managers and communication managers may also be resource managers. A resource manager may be written by a customer, a third party, or Digital.

Each resource manager type offers a resource-manager-specific interface that is used by application programs to access and modify recoverable resources managed by the resource manager. A description of these resource manager interfaces is outside the scope of DECdta. However, many of these resource manager interfaces have architectures defined by industry standards, such as SQL (e.g., the VAX Rdb/VMS product), CODASYL data manipulation language (e.g., the VAX DBMS product), and COBOL file operations (e.g., RMS in the VMS system).

One type of resource manager that plays a special role in TP systems is a queue resource manager. It manages recoverable queues, which are often used to store requests.[3] It allows application programs to place elements into queues and retrieve them, so that application programs can communicate even though they execute independently and asynchronously. For example, an application program that sends elements can communicate with one that receives elements even if the two application programs are not operational simultaneously. This communication arrangement improves availability and facilitates batch input of elements.

A queue resource manager interface supports such operations as open-queue, close-queue, enqueue, dequeue, and read-element. The ACMS and DECintact TP monitors both have queue resource managers as components.

Transaction Manager

A transaction manager supports the transaction abstraction. It is responsible for ensuring the atomicity of each transaction by telling each resource manager in a transaction when to commit. It uses a two-phase commit protocol to ensure that either all resource managers accessed by a transaction commit the transaction or they all abort the transaction.[4] To support transaction atomicity, a transaction manager provides the following functions:

- Transaction demarcation operations allow application programs or resource managers to start and commit or abort a transaction. (Resource managers sometimes start a transaction to execute a resource operation if the caller is not executing a transaction. The SQL standard requires this.)
- Transaction execution operations allow resource managers and communication managers to declare themselves part of an existing transaction.
- Two-phase commit operations allow resource managers and communication managers to change a transaction's state (to "prepared," "committed," or "aborted").

The serializability of transactions is primarily the responsibility of the resource managers. Usually, a resource manager ensures serializability by setting locks on resources accessed by each transaction, and by releasing the locks after the transaction manager tells the resource manager to commit. (The latter activity makes serializability partly the responsibility of the transaction manager.) If transactions become deadlocked, a resource manager may detect the deadlock and abort one of the deadlocked transactions.

The durability of transactions is a responsibility of transaction managers and resource managers. The transaction manager is responsible for the durability of the commit or abort decision. A resource manager is responsible for the durability of operations of committed transactions. Usually, it ensures durability by storing a description of each transaction's resource operations and state changes in a stable (e.g., disk-resident) log. It can later use the log to reconstruct transactions' states while recovering from a failure.

A detailed description of the DECdta transaction manager component appears in the Transaction Manager Architecture section.

Communication Manager

A communication manager provides services for communication between named objects in a TP system, such as application programs and transaction managers. Some communication managers participate in coordinating the termination of a transaction by propagating the transaction manager's two-phase commit operations as messages to remote communication managers. Other communication managers propagate application data and transaction context, such as a transaction identifier, from one node to another. Some do both.

A TP system can support multiple communication managers. These communication managers can interact with other nodes using different commit protocols or message-passing protocols, and may be part of different name spaces, security domains, system management domains, etc. Examples are an IBM SNA LU6.2 communication manager or an ISO-TP communication manager.

By supporting multiple communication managers, the DECdta architecture enhances the interoperability of TP systems. Different TP systems can interoperate by executing a transaction using different commit protocols.

A communication manager offers an interface for application programs to communicate with other application programs. Different communication managers may offer different communication paradigms, such as remote procedure call or peer-to-peer message passing.

A communication manager also has an interface to its local transaction manager. It uses this interface to tell the transaction manager when a transaction has spread to a new node and to obtain information about transaction commitment, which it exchanges with communication managers on remote nodes.

Presentation Manager

A presentation manager provides an application program with a record-oriented interface to a presentation device. Its services are used by application programs, usually request initiators. By using presentation manager services, instead of directly accessing a presentation device, application programs become device independent.

A forms manager is one type of presentation manager. Just as a database system supports operations to define, open, close, and access databases, a forms manager supports operations to define, enable, disable, and access forms. A form includes the definition of the fields (with different attributes) that make up the form. It also includes services to map the fields into device-independent application records, to perform data validation, and to perform data conversion to map fields onto device-specific frames.

One presentation manager is Digital's DECforms forms management product. The DECforms product is the first implementation of the ANSI/ISO Forms Interface Management Systems standard (CODASYL FIMS).[5]

Request Manager

A request manager provides services to authenticate the source of requests (a user and/or a presentation device), to submit requests, and to receive replies from the execution of requests. It supports such operations as send-request and receive-reply. Send-request must provide the identity of the source device, the identity of the user who entered the request, the identity of the application program to be invoked, and the input data to the program.

A request manager can either pass the request directly to an application program, or it can store requests in a queue. In the latter case, another request manager can subsequently schedule the request by dequeuing the request and invoking an application program. The ACMS System Interface is an example of an existing request manager interface for direct requests. The ACMS Queued Transaction Initiator is an example of a request manager that schedules queued requests.[1]

Transaction Manager Architecture

DECdta components are tied together by the transaction abstraction. Transactions allow application programs, resource managers, request managers (indirectly through queue resource managers), and communication managers to interoperate reliably. Since transactions play an especially important role in the DECdta architecture, we describe the transaction management functions in more detail.

The DECdta architecture includes interfaces between transaction managers and application programs, resource managers, and communication managers, as shown in Figure 3. It also includes a transaction manager protocol, whose messages are propagated by communication managers. This protocol is used by Digital's DECdtm distributed transaction manager.[2]

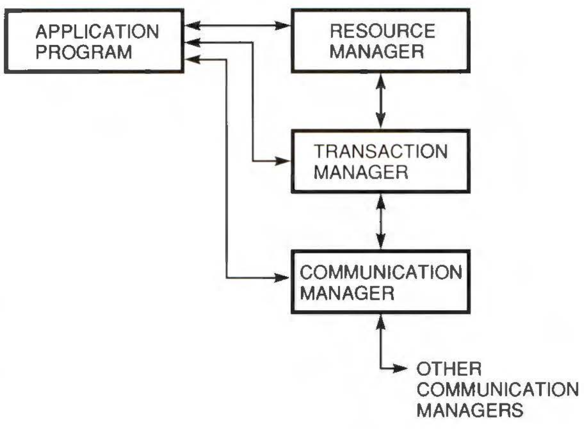


Figure 3 Transaction Manager Architecture

From a transaction manager's viewpoint, a transaction consists of transaction demarcation operations, transaction execution operations, two-phase commit operations, and recovery operations.

- The transaction demarcation operations are issued by an application program to a transaction manager and include operations to start and either end or abort a transaction.
- Transaction execution operations are issued by resource managers and communication managers to a transaction manager. They include operations
  - For a resource manager or communication manager to join an existing transaction
  - For a communication manager to tell a transaction manager to start a new branch of a transaction that already exists at another node
- Two-phase commit operations are issued by a transaction manager to resource managers, communication managers, and through communication managers to other transaction managers, and vice-versa. They include operations
  - For a transaction manager to ask a resource manager or communication manager to prepare, commit, or abort a transaction
  - For a resource manager or communication manager to tell a transaction manager whether it has prepared, committed, or aborted a transaction

- For a communication manager to ask a transaction manager to prepare, commit, or abort a transaction
- For a transaction manager to tell a communication manager whether it has prepared, committed, or aborted a transaction

- Recovery operations are issued by a resource manager to its transaction manager to determine the state of a transaction (i.e., committed or aborted).

In response to a start operation invoked by an application program, the transaction manager dispenses a unique transaction identifier for the transaction. The transaction manager that processes the start operation is that transaction's home transaction manager.

When an application program invokes an operation supported by a resource manager, the resource manager must find out the transaction identifier of the application program's transaction. This can happen in different ways. For example, the application program may tag the operation with the transaction identifier, or the resource manager may look up the transaction identifier in the application program's context. When a resource manager receives its first operation on behalf of a transaction, T, it must *join* T, meaning that it must tell a transaction manager that it is a *subordinate for T*. Alternatively, the DECdta architecture supports a model in which a resource manager may ask to be joined automatically to all transactions managed by its transaction manager, rather than asking to join each transaction separately.

A transaction, T, *spreads* from one node, Node 1, to another node, Node 2, by sending a message (through a communication manager) from an application program that is executing T at Node 1 to an application program at Node 2. When T sends a message from Node 1 to Node 2 for the first time, the communication managers at Node 1 and Node 2 must perform branch registration. This function may be performed automatically by the communication managers. Or, it may be done manually by the application program, which tells the communication managers at Node 1 and Node 2 that the transaction has spread to Node 2. In either case, the result is as follows: the communication manager at Node 1 becomes the subordinate of the transaction manager at Node 1 for T and the superior of the communication manager at Node 2 for T; and the communication manager at Node 2 becomes the superior of the transaction manager at Node 2 for T. This arrangement allows the commit protocol between transaction managers to be propagated properly by communication managers.

After the transaction is done with its application work, the application program that started transaction T may invoke an "end" operation at the home transaction manager to *commit* T. This causes the home transaction manager to ask its subordinate resource managers and communication managers to try to commit T. The transaction manager does this by using a two-phase commit protocol. The protocol ensures that either all subordinate resource managers commit the transaction or they all abort the transaction.

In phase 1, the home transaction manager asks its subordinates for T to *prepare* T. A subordinate prepares T by doing what is necessary to guarantee that it can either commit T or abort T if asked to do so by its superior; this guarantee is valid even if it fails immediately after becoming prepared. To prepare T,

- Each subordinate for T recursively propagates the prepare request to its subordinates for T
- Each resource manager subordinate writes all of T's updates to stable storage
- Each resource manager and transaction manager subordinate writes a prepare-record to stable storage

A subordinate for T replies with a "yes" vote if and when it has completed its stable writes and all of its subordinates for T have voted "yes"; otherwise, it votes "no." If any subordinate for T does not acknowledge the request to prepare within the timeout period, then the home transaction manager aborts T; the effect is the same as issuing an abort operation.

In phase 2, when the home transaction manager has received "yes" votes from all of its subordinates for T, it decides to commit T. It writes a commit record for T to stable storage and tells its subordinates for T to commit T. Each subordinate for T writes a commit record for T to stable storage and recursively propagates the commit request to its subordinates for T. A subordinate for T replies with an acknowledgment if and when it has committed the transaction (in the case of a resource manager subordinate) and has received acknowledgments from all subordinates for T. When the home transaction manager receives acknowledgments from all of its subordinates for T, the transaction commitment is complete.

To recover from a failure, all resource managers that participated in a transaction must examine their logs on stable storage to determine what to do. If the log contains a commit or abort record for T, then T completed. No action is required. If the log contains no prepare, commit, or abort record for T, then T was active. T must be aborted. If the log contains a prepare record for T, but no commit or abort record for T, T was between phases 1 and 2. The resource manager must ask its superior transaction manager whether to commit or abort the transaction.

An inherent problem in all two-phase commit protocols is that a resource manager is blocked between phases 1 and 2, that is, after voting "yes" and before receiving the commit or abort decision. It cannot commit or abort the transaction until the transaction manager tells it which to do. If its transaction manager fails, the resource manager may be blocked indefinitely, until either the transaction manager recovers or an external agent, such as a system manager, steps in to tell the resource manager whether to commit or abort.

A transaction T may spontaneously abort due to system errors at any time during its execution. Or, an application program (prior to completing its work) or a resource manager (prior to voting "yes") may tell its transaction manager to abort T. In either case, the transaction manager then tells all of its subordinates for T to undo the effects of T's resource manager operations. Subordinate resource managers abort T, and subordinate communication managers recursively propagate the abort request to their subordinates for T.

The two-phase commit protocol is optimized for those cases in which the number of messages exchanged can be reduced below that of the general case (e.g., if there is only one subordinate resource manager, if a resource manager did not modify resources, or if the presumed-abort protocol was used to save acknowledgments).[6]

Summary

We have presented an overview of the DECdta architecture. As part of this overview, we introduced the components and explained the function of each interface. We also described the DECdta transaction management architecture in some detail. Over time, many interfaces of the DECdta model will be made public via product offerings or architecture publications.

Acknowledgments

This architecture grew from discussions with many colleagues. We thank them all for their help, especially Dieter Gawlick, Bill Laing, Dave Lomet, Bruce Mann, Barry Rubinson, Diogenes Torres, and the TP architecture group, including Edward Braginsky, Tony DellaFera, George Gajnak, Per Gyllstrom, and Yoav Raz.

References

1. T. Speer and M. Storm, "Digital's Transaction Processing Monitors," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 18-32.

2. W. Laing, J. Johnson, and R. Landau, "Transaction Management Support in the VMS Operating System Kernel," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 33-44.

3. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Reading, MA: Addison-Wesley, 1987).

4. P. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," *Proceedings 1990 ACM SIGMOD Conference on Management of Data* (May 1990).

5. *FIMS Journal of Development* (Norfolk, VA: CODASYL FIMS Committee, July 1990).

6. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Transactions on Database Systems,* vol. 11, no. 4 (December 1986).

Thomas G. Speer
Mark W. Storm

Digital's Transaction Processing Monitors

Digital provides two transaction processing (TP) monitor products — ACMS (Application Control and Management System) and DECintact (Integrated Application Control). Each monitor is a unified set of transaction processing services for the application environment. These services are layered on the VMS operating system. Although there is a large functional overlap between the two, both products achieve similar goals by means of some significantly different implementation strategies. Flow control and multithreading in the ACMS monitor is managed by means of a fourth-generation language (4GL) task definition language. Flow control and multithreading in the DECintact monitor is managed at the application level by third-generation language (3GL) calls to a library of services. The ACMS monitor supports a deferred task model of queuing, and the DECintact monitor supports a message-based model. Over time, the persistent distinguishing feature between the two monitors will be their different application programming interfaces.

Transaction processing is the execution of an application that performs an administrative function by accessing a shared database. Within transaction processing, processing monitors provide the software "glue" that ties together many software components into a transaction processing system solution.

A typical transaction processing application involves interaction with many terminal users by means of a presentation manager or forms system to collect user requests. Information gathered by the presentation manager is then used to query or update one or more databases that reflect the current state of the business. A characteristic of transaction processing systems and applications is many users performing a small number of similar functions against a common database. A transaction processing monitor is a system environment that supports the efficient development, execution, and management of such applications.

Processing monitors are usually built on top of or as extensions to the operating system and other products such as database systems and presentation services. By so doing, additional components can be integrated into a system and can fill "holes" by providing functions that are specifically needed by transaction processing applications. Some examples of these functions are application control and management, transaction-processing-specific execution environments, and transaction-processing-specific programming interfaces.

Digital provides two transaction processing monitors: the Application Control and Management System (ACMS) and the DECintact monitor. Both monitors are built on top of the VMS operating system. Each monitor provides a unified set of transaction-processing-specific services to the application environment, and a large functional overlap exists between the services each monitor provides. The distinguishing factor between the two monitors is in the area of application programming styles and interfaces — fourth-generation language (4GL) versus third-generation language (3GL). This distinction represents Digital's recognition that customers have their own styles of application programming. Those that prefer 4GL styles should be able to build transaction processing applications using Digital's TP monitors without changing their style. Similarly, those that prefer 3GL styles should also be able to build TP applications using Digital's TP monitors without changing their style.

The ACMS monitor was first introduced by Digital in 1984. The ACMS monitor addresses the requirements of large, complex transaction processing applications by making them easier to develop and manage. The ACMS monitor also creates an efficient execution environment for these applications.

The DECintact monitor (Integrated Application Control) was originally developed by a third-party vendor. Purchased and introduced by Digital in 1988, it has been installed in major financial institutions and manufacturing sites. The DECintact monitor includes its own presentation manager, support for DECforms, a recoverable queuing subsystem, a transaction manager, and a resource manager that provides its own recovery of RMS (Record Management Services) files.

This paper highlights the important similarities and differences of the ACMS and DECintact monitors in terms of goals and implementation strategies.

Development Environment

Transaction processing monitors provide a view of the transaction processing system for application development. Therefore, the ACMS and DECintact monitors must embody a style of program development.

ACMS Programming Style

A "divide and conquer" approach was used in the ACMS monitor. The work typically involved in developing a TP application was divided into logically separate functions described below. Each of these functions was then "conquered" by a special utility or approach.

In the ACMS monitor, an "application" is defined as a collection of selectable units of work called tasks. A separate application definition facility isolates the system management characteristics of the application (such as resource allocation, file location, and protection) from the logic of the application.

The specification of menus is also decoupled from the application. A nonprocedural (4GL) method of defining menu layouts is used in which the layouts are compiled into form files and data structures to be used at run-time. Each menu entry points either to another menu or to an application and a task. (Decoupling menus from the application allows user menus to be independent of how the tasks are grouped into applications.)

In addition to separate menu specification and system management characteristics, the application logic is broken down into the three logical parts of interactive TP applications:

- Exchange steps support the exchange of data with the end user. This exchange is typically accomplished by displaying a form on a terminal screen and collecting the input.
- Processing steps perform computational processing and database or file I/O through standard subroutines. The subroutines are written in any language that accepts records passed by reference.
- The task definition language defines the flow of control between processing steps and exchange steps and specifies transaction demarcation. Work spaces are special records that the ACMS monitor provides to pass data between the task definition, exchange steps, and processing steps.

A compiler, called the application definition utility (ADU), is implemented in the ACMS monitor to compile the task definition language into binary data structures. The run-time system is table-driven, rather than interpreted, by these structures.

Digital is the only vendor that supplies this "divide and conquer" solution to building large complex TP applications. We believe this approach — unique in the industry — reduces complexity, thus making applications easier to produce and to manage.

DECintact Programming Style

The approach to application development used in the DECintact monitor provides the application developer with 3GL control over the transaction processing services required. This approach allows application prototyping and development to be done rapidly. Moreover, the application can make the most efficient use of monitor services by selecting and controlling only those services required for a particular task.

In the DECintact monitor, an application is defined as one or more programs written entirely in 3GL and supported by the VMS system. The code written by the application developer manages all flow control, user interaction, and data manipulation through the utilities and service libraries provided by the DECintact monitor. All DECintact services are callable, including most services provided by the DECintact utilities. The DECintact services are as follows:

- A library of presentation services used for all interaction with users. The application developer includes calls to these services for form manipulation and display. Forms are created with a forms editor utility and can be updated dynamically. Forms are displayed by the DECintact terminal manager in emulated block mode. Device- and terminal-dependent information is completely separated from the implementation of the application.

- The separation of specification of menus from the application. DECintact menus are defined by means of a menu database and are compiled into data structures accessed at run-time. The menus are tree-structured. Each entry points either to another menu entry or to an executable application image. The specification of menus is linked to the DECintact monitor's security subsystem. The DECintact terminal user sees only those specific menu entries for which the user has been granted access.
- A library of services for the control of file and queue operations. In addition to layered access to the RMS file system, the DECintact monitor supports its own hash file format (a functional analog to single-keyed indexed files in RMS) which provides very fast, efficient record retrieval. The application developer includes calls to these services for managing RMS and hash file I/O operations, demarcating recovery unit boundaries, creating queues, placing data items on queues, and removing data items from queues. The queuing subsystem is typically an integral part of application design and work flow control. Application-defined DECintact recovery units ensure that RMS, hash, and queue operations can be committed or aborted atomically; that is, either all permanent effects of the recovery unit happen, or none happen.

Because of DECintact's 3GL development environment, application programmers who are accustomed to calling procedure libraries from standard VMS languages or who are familiar with other transaction processing monitors can easily learn DECintact's services. Application prototypes can be produced quickly because only skills in 3GL are required. Further, completed applications can be produced quickly because training time is minimal.

On-line Execution Environment

Transaction processing monitors provide an execution environment tailored to the characteristics and needs of transaction processing applications. This environment generally has two aspects: on-line, for interactive applications that use terminals; and off-line, for noninteractive applications that use other devices.

Traditional VMS timesharing applications are implemented by allocating one VMS process to each terminal user when the user logs in to the system. An image activation is then done each time the terminal user invokes a new function. This method is most beneficial in simple transaction processing applications that have a relatively small number of users. However, as the number of users grows or as the application becomes larger and more complex, several problem areas may arise with this method:

- Resource use. As the number of processes grows, more and more memory is needed to run the system effectively.
- Start-up costs. Process creation, image activation, file opens, and database binds are expensive operations in terms of system resources utilized and time elapsed. These operations can degrade system performance if done frequently.
- Contention. As the number of users simultaneously accessing a database or file grows, contention for locks also increases. For many applications, lock contention is a significant factor in throughput.
- Processing location. Single process implementations limit distribution options.

ACMS On-line Execution

To address the problems listed above, Digital implemented a client/server architecture in the ACMS monitor. (Client/server is also called request/response.) The basic run-time architecture consists of three types of processes, as shown in Figure 1: the command process, execution controller, and procedure servers.

An agent in the ACMS monitor is a process that submits work requests to an application. In the ACMS system, the command process is a special agent responsible for interactions with the terminal user. (In terms of the DECdta architecture, the command process implements the functions of a request initiator, presentation manager, and request manager for direct requests.)[1] The command process is generally created at system start-up time, although ACMS commands allow it to be started at other times. The process is multithreaded through the use of VMS asynchronous system traps (AST). Thus, one command process per node is generally sufficient for all terminals handled by that node.

There are two subcomponents of the ACMS monitor within the command process:

- System interface, which is a set of services for submitting work requests and for interacting with the ACMS application

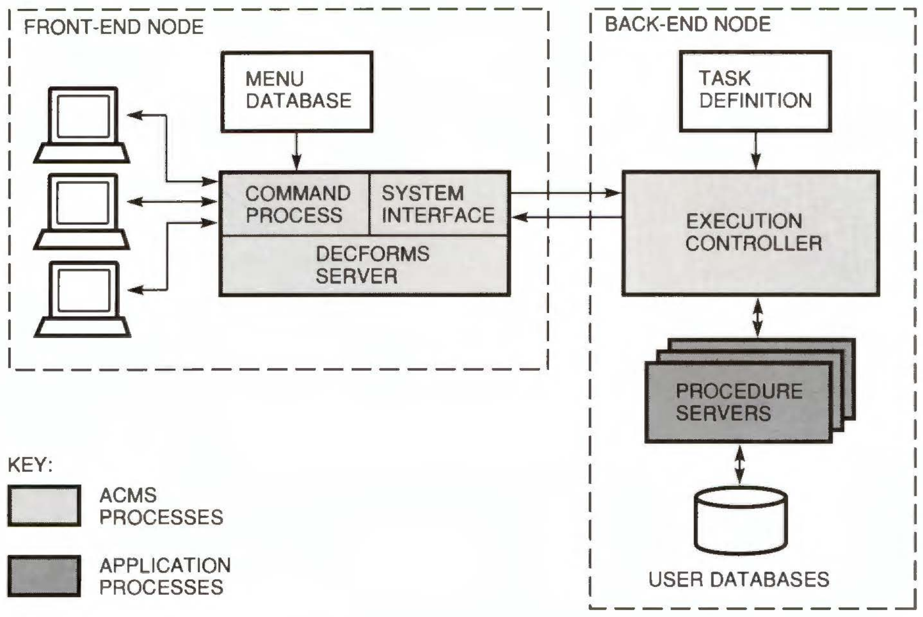


Figure 1 Basic Run-time Architecture of the ACMS Monitor

- DECforms, Digital's forms management product, which implements the ANSI/ISO Forms Interface Management System (FIMS) that provides the presentation server for executing the exchange steps

The command process reads the menu definition for a particular terminal user and determines which menu to display. When the terminal user selects a particular menu entry, the command process calls the ACMS system interface services to submit the task. The system interface uses logical names from the VMS system to translate the application name into the address of the execution controller that represents that application. The system interface then sends a message to the execution controller. The message contains the locations of the presentation server and an index into the task definition tables for the particular task. The status of the task is returned in the response. During the course of task execution, the command process accepts callbacks from the task to display a form for interaction with the terminal user.

The execution controller executes the task definition language and creates and manages procedure servers. The controller is created at application start-up time and is multithreaded by using VMS ASTs. There is one execution controller per application. (In terms of the DECdta architecture, the execution controller and the procedure servers implement the functions of a transaction server.)[1]

When the execution controller receives a request from the command process, it invokes DECdtm (Digital Distributed Transaction Manager) services to join the transaction if the agent passes the transaction identifier. If the agent does not pass a transaction identifier, there is no transaction to join and a DECdtm or resource-manager-specific transaction is started as specified in the task definition. The execution controller then uses the task index to find the tables that represent the task. When the execution of a task reaches an exchange step, the execution controller sends a callback to the command process for a form to be displayed and the input to be collected for the task. When the request to display a form is sent to the command process, the execution controller dismisses the AST to enable other threads to execute. When the response to the request arrives from the exchange step, an AST is added to the queue for the execution controller.

When a task comes to a processing step, the execution controller allocates a free procedure server to the task. It then sends a request to the procedure server to execute the particular procedure and dismisses the AST. If no procedure server is free, the execution controller puts the request on a waiting list and dismisses the AST. When a procedure server becomes free, the execution controller checks the wait list and allocates the procedure server to the next task, if any, on the wait list.

Procedure servers are created and deleted by the execution controller. Procedure servers are a collection of user-written procedures that perform computation and provide database or file accesses for the application. The procedures are written in standard languages and use no special services. The ACMS system creates a transfer vector from the server definition. This transfer vector is linked into the server image. With this vector, the ACMS system code can receive incoming messages and translate them into calls to the procedure.

A procedure server is specified with initialization and termination procedures, which are routines supplied by the user. The ACMS monitor calls these procedures whenever a procedure server is created and deleted. The initialization procedure opens files and performs database bind operations. The termination procedure does clean-up work, such as closing files prior to process exit.

The ACMS architecture addresses the problem areas discussed in the On-line Execution Environment section in several ways.

Resource Use Because procedure servers are allocated only for the time required to execute a processing step, the servers are available for other use while a terminal user types in data for the form. Thus, the system can execute efficiently with fewer procedure servers than active terminal users. Improvement gains in resource use can vary, depending on the application. Our debit and credit benchmark experiments with the ACMS monitor and the Rdb/VMS relational database system indicated that the most improvement occurs with one procedure server for every one or two transactions per second (TPS). These benchmarks equate to 1 procedure server for every 10 to 20 active terminal users.

The use of procedure servers and the multithreaded character of the execution controller and the command process allow the architecture to reduce the number of processes and, therefore, the number of resources needed. The optimal solution for resource use would consist of one large multithreaded process that performed all processing. However, we chose to trade off some resource use in the architecture in favor of other gains.

- Ease of use — Multithreaded applications are generally more difficult to code than single-threaded applications. For this reason, procedure server subroutines in the ACMS system can be written in a standard fashion by using standard calls to Rdb/VMS and the VMS system.
- Error isolation — In one large multithreaded process, the threads are not completely protected within the process. An application logic error in one thread can corrupt data in a thread that is executing for a different user. A severe error in one thread could potentially bring down the entire application. The multithreaded processes in the ACMS architecture (i.e., the execution controller and command process) are provided by Digital. Because no application code executes directly in these processes, we can guarantee that no application coding error can affect them. Procedure servers are single-threaded. Therefore, an application logic error in a procedure server is isolated to affect only the task that is executing in the procedure server.

Start-up Costs The run-time environment is basically "static," which means that the start-up costs (i.e., system resources and elapsed time) are incurred infrequently (i.e., at system and application start-up time). A timesharing user who is running many different applications causes image activations and rundowns by switching among images. Because the terminal user in the ACMS system is separated from the applications processes, the process of switching applications involves only changing message destinations and incurs minimal overhead.

Contention The database accesses in the ACMS environment are channeled through a relatively few, but heavily used, number of processes. The typical VMS timesharing environment uses a large number of lightly used processes. By reducing the number of processes that access the database, the contention for locks is reduced.

Processing Location Because the ACMS monitor is a multiprocess architecture, the command process and forms processing can be done close to the terminal user on small, inexpensive machines. This method takes advantage of the inexpensive processing power available on these smaller machines while the rest of the application executes on a larger VAXcluster system.

DECintact On-line Execution

Although the specific components of the DECintact monitor vary from those of the ACMS monitor, the basic architecture is very similar. Figure 2 shows the application configured locally to the front end. The run-time architecture consists of three types of DECintact system processes — terminal manager/dispatcher, DECforms servers, server manager — and, typically, one or more application processes. When forms processing is distributed, the same application is configured as shown in Figure 3.

The DECintact monitor can run in multiple copies on any one VAX node. Each copy can be an independent run-time environment; or it can share data and resources, such as user security profiles and menu definitions, with other copies on the same system. Thus, independent development, testing, and production environments can reside on the same node.

In the DECintact system, the terminal manager/dispatcher process (one per copy) is responsible for the following:

- Displaying DECintact forms
- Coordinating DECforms forms display
- Interacting with local applications
- Communicating, through DECnet, with remote DECintact copies
- Maintaining security authorization, including the dynamic generation of user-specific menus

Applications designated in the local menu database as remote applications cause the front-end terminal manager/dispatcher process to communicate with the cooperating back-end terminal manager/dispatcher process through a task-to-task DECnet link. (In terms of the DECdta architecture, the terminal manager/dispatcher implements the functions of presentation manager, request initiator, and request manager for direct requests.)[1]

When a user selects the remote task, that user's request is sent to the back end and is treated by the application as a local request. The terminal manager/dispatcher process is started automatically as part of a copy start-up and is multithreaded. Therefore, one such process can handle all the terminal users for a particular DECintact copy.

When the terminal user selects a menu task, one of the following actions occurs, depending on whether the task is local or remote and whether it is single- or multithreaded.

If the application is local and single-threaded, a VMS process may be created that activates the application image associated with this task. The terminal manager/dispatcher, upon start up, may create a user-specified number of application shell VMS processes to activate subsequent application images. If such a shell exists when the user selects a task, this process is used to run the application image. Each user who selects a given menu entry receives an individual VMS process and image.

If the application is local and multithreaded, the terminal manager/dispatcher first determines

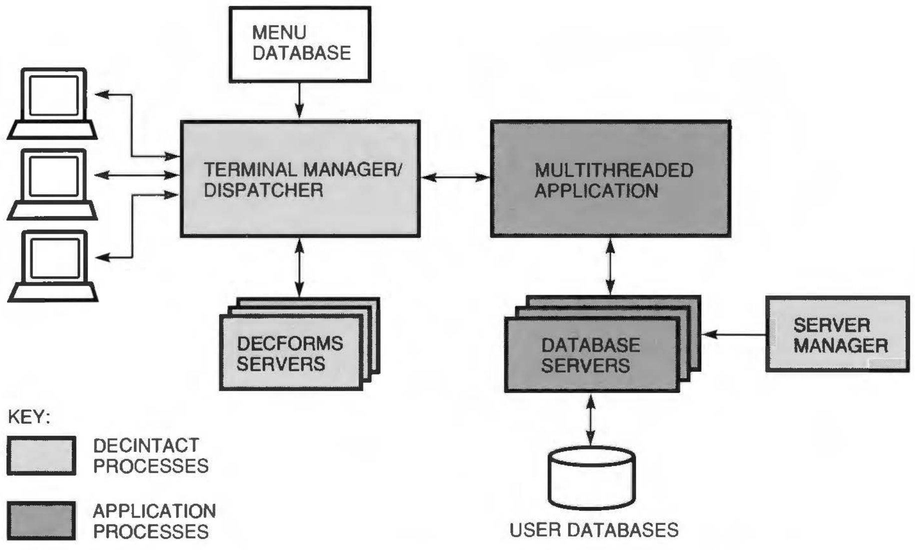


Figure 2 Basic Run-time Architecture of the DECintact Monitor

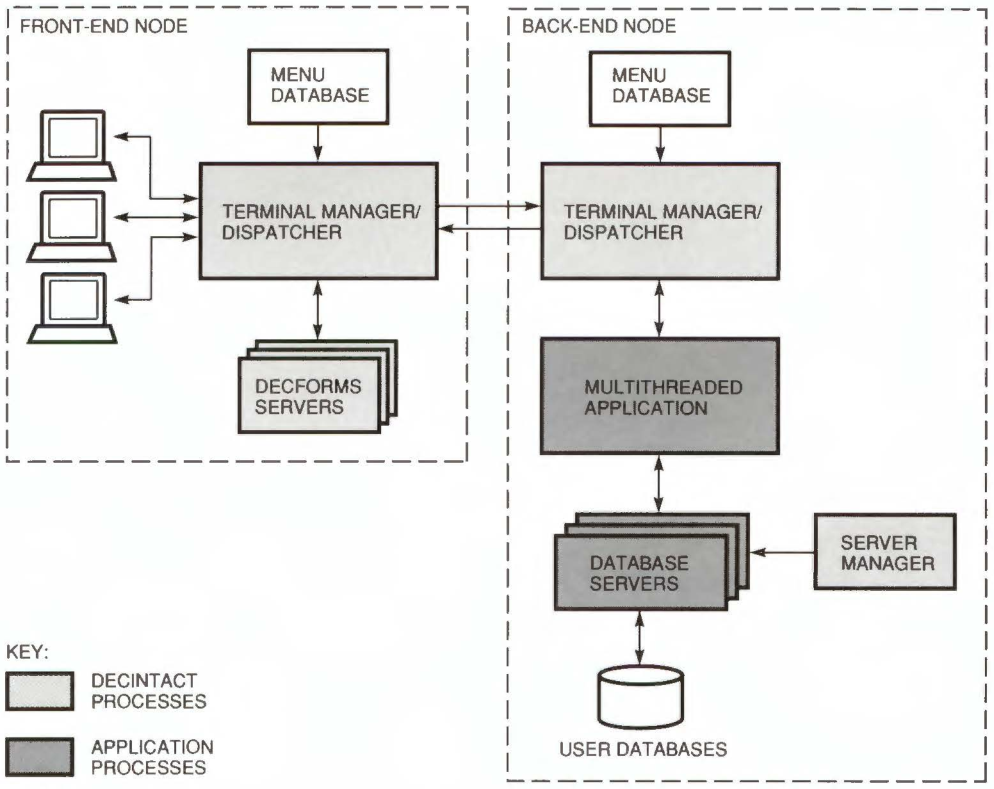


Figure 3   DECintact Basic Architecture with Distributed Forms Processing

whether this task has already been activated by previous users. If the task has not been activated and a shell is not available, the terminal manager/dispatcher creates a VMS process for the application and activates the image. If the task is already activated, the terminal manager/dispatcher connects the user to the active task. The user becomes another thread of execution within the image. Multithreaded applications handle many simultaneous users within the context of one VMS process and image.

Remote applications, whether single- or multithreaded, route the menu task selection to a remote terminal manager/dispatcher process. On receipt of the request, the remote terminal manager/dispatcher processes the selection locally by using the same procedures as described above.

Local DECintact forms interaction is handled in the following manner by the local terminal manager/dispatcher. The application's call to display a form sends a request to the terminal manager. The terminal manager locates the form in its database of active forms, displays the form on the user's terminal, and returns control to the application when the user has entered all data in the form. If the application is remote, form information is sent between cooperating local and remote terminal manager processes; the interface is transparent to the application.

In addition to supporting DECintact forms, the DECintact monitor also supports applications that use DECforms as their presentation service. The implementation of this support follows the same client/server model used by the ACMS system's support for DECforms and shares much of the underlying run-time interprocess communication code used by the ACMS monitor. Functionally, the two implementations of DECforms support are also similar to the ACMS monitor. Both implementations offer transparent support for distributed DECforms processing, automatic forms caching (i.e., propagation of updated DECforms in a distributed environment), and DECforms session caching for increased performance.

The DECintact monitor supports application-level, single- and multithreaded environments. The DECintact monitor's threading package allows application programmers to use standard languages supported by the VMS system to write multithreaded processes. Applications declare themselves as either single- or multithreaded. With the exception of the declaration, there is little difference between the way an on-line multithreaded application and its single-threaded counterpart must be coded. For on-line applications, thread creation, deletion, and management are automatic. New threads are created when a terminal user selects the multithreaded application and are deleted when the user leaves the application.

In a single-threaded application, the following occurs:

- Each user receives an individual VMS process and image context (e.g., 200 users, 200 processes).
- All terminal and file I/O is synchronous.
- The application image normally exits when the application work is completed.

In a multithreaded on-line application, the following occurs:

- One VMS process/image can handle many simultaneous users.
- All terminal and file I/O is asynchronous.
- New threads are created automatically when new users are connected to the process.
- The application image does not exit when all currently allocated threads have completed execution but remains for use by new on-line users.

For each thread in a multithreaded application image, the DECintact system maintains thread context and state information. Each I/O request is issued asynchronously. Immediately after control is returned, but before the I/O request completes, the DECintact system saves the currently executing thread's context and schedules another thread to execute. When the thread's I/O completion AST is delivered, the thread's context is restored, and the thread is inserted on an internally maintained list of threads eligible for execution.

A thread's context consists of the following:

- An internally maintained thread block containing state information
- The stack
- Standard DECintact work spaces that are allocated to each thread and that maintain terminal and file management context
- Local storage (e.g., the $LOCAL PSECT in COBOL applications) that the application has designated as thread-specific

The PSECT naming convention allows the application to decide which variable storage is thread-specific and which is process-global. Thread-specific storage is unavailable to other threads in the same process because it is saved and restored on each thread switch. Process-global storage is always available to all threads in the process and can be used when interthread communication or synchronization is desired.

The use of multithreading in the DECintact system is appropriate for higher volume multiuser applications that perform frequent I/O. Such application usage is typical in transaction processing environments. Because thread switches occur only when I/O is requested or when locking requests are issued, this environment may not be recommended for applications that perform infrequent I/O or that expect very small numbers of concurrent users, such as end-of-day accounting programs or other batch-oriented processing. These kinds of applications typically choose to declare themselves as single-threaded.

All I/O from within a multithreaded DECintact application process is asynchronous. Therefore, the DECintact system provides a client/server interface between multithreaded applications and synchronous database systems, such as VAX DBMS (Database Management System) and Rdb/VMS systems. The interface is provided because calling a synchronous database operation directly from within a multithreaded application would stall the calling thread and all other threads until the call completed. Figure 2 shows that a typical on-line DECintact application accessing Rdb/VMS, for example, is written in two pieces:

- A multithreaded, on-line piece (the client), that handles forms requests from multiple users
- A single-threaded, database server piece (a server instance), that performs the actual synchronous database I/O

This client/server approach to database access is functionally very similar to that of ACMS procedure servers and offers similar benefits. Like the ACMS monitor, the DECintact monitor offers system management facilities to define pools of servers and to adjust them dynamically at run-time in accordance with load. Similar algorithms are used in both monitors to allocate server instances to client threads and to start up new instances, as necessary. The DECintact server code, like the ACMS procedure server code, can define initialization and termination procedures to perform once-only start-up and shut-down processing. With DECintact transaction semantics, which are layered on DECdtm services, a client can declare a global transaction that the server instance will join. The server instance can also declare its own independent transaction or no transaction. (In terms of the DECdta architecture, this client/server approach implements the functions of a transaction server.)[1] The principal difference between the DECintact and ACMS approach is that DECintact clients and servers use a message-based 3GL communications interface to send and receive work requests. Control in the ACMS monitor resides in the execution controller.

As the ACMS monitor does, the DECintact architecture addresses the problem areas discussed in the On-line Execution section in several ways. Also, as with the ACMS approach, the factors we chose to trade off allowed us to achieve better efficiency, performance, and ease of use.

Resource Use The DECintact system's multithreaded methodology economizes on VMS resources. Similar to the method used in the ACMS monitor, the system reduces process creations and image activations. A major difference between the ACMS and DECintact architectures is the way the DECintact monitor implements multithreading support. The transparent implementation of threading capabilities means that coding multithreaded applications is no more difficult than coding traditional single-threaded applications. As with any application-level threading scheme, however, the responsibility for ensuring that a logic error in one thread is isolated to that thread lies with the application. The DECintact client/server facilities for accessing databases, like those used in the ACMS monitor, can realize similar benefits in process reuse, throughput, and error isolation.

Start-up Costs The DECintact architecture, like the ACMS architecture, distributes start-up costs (i.e., system resources and elapsed time) between two points: the start of the DECintact system, and the start of applications. System start-up can involve prestarting VMS process shells (as discussed previously) for subsequent application image activation. On-line application start-up is executed on demand when the first user selects a particular menu task. Multithreaded applications, once started, do not exit but wait for new user threads as users select the application. Thus, the DECintact terminal user can switch between application images and incur only an inexpensive thread creation.

Contention As in the ACMS monitor, database accesses in the DECintact client/server environment are channeled through a relatively few, but heavily used, number of processes rather than through a large number of lightly used processes. This reduction decreases lock contention.

Processing Location Forms processing can be off-loaded to a front end and brought closer to the terminal user. Thus smaller, less expensive CPUs can be used while the rest of the application executes on a larger back-end machine or cluster. In the DECintact monitor, the front end can consist of forms processing only or a mix of forms processing and application remote queuing work.

Off-line Execution

Many transaction processing applications are used with nonterminal devices, such as a bar code reader or a communications link used for an electronic funds transfer application. Because there is no human interaction with these applications, they have two requirements that differ from the requirements of interactive applications: tasks must be simple data entries, and the system must handle failures transparently.

ACMS Off-line Execution

The ACMS monitor's goal for off-line processing is to allow simple transaction capture to continue when the application is not available. A typical example is the continued capture of data on a manufacturing assembly line by a MicroVAX system when the application is unavailable. The ACMS monitor provides two mechanisms for supporting nonterminal devices: queuing agents and user-written agents.

Figure 4 illustrates the ACMS queuing model. A queuing system is a resource manager that processes entries, with priorities, in first-in, first-out (FIFO) order. (In terms of DECdta, this is the queue resource manager.)[1] The ACMS queuing facility is built upon RMS-indexed files. The primary goal of ACMS queuing is to provide a store-and-forward mechanism to allow task requests to be collected for later execution. By using the ACMS$ENQUE_TASK service, a user can write

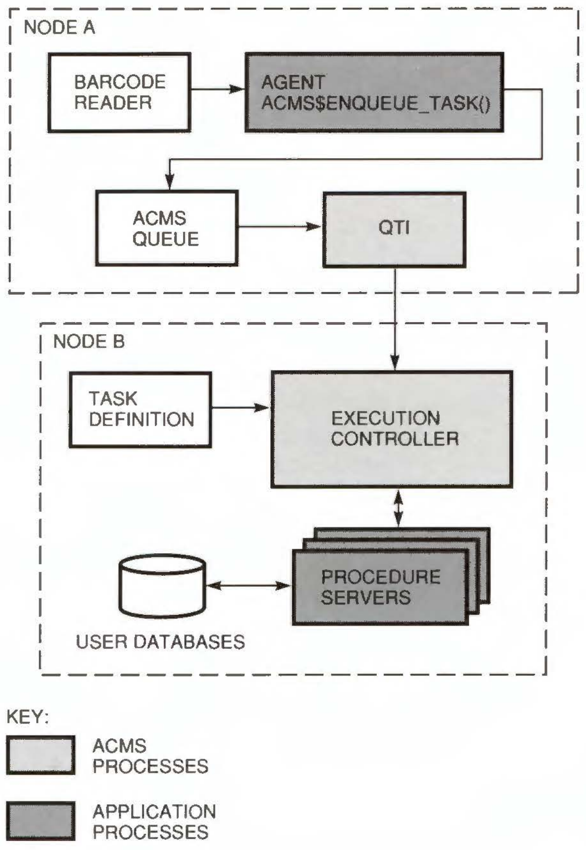


Figure 4 ACMS Queuing Agents

a process that captures a task request and safely stores the task on a local disk queue.

The ACMS monitor provides a special agent, called the queued task initiator (QTI), which takes a task entry from the queue and submits it to the appropriate execution controller. The QTI starts a DECdtm transaction, removes the task entry from the queue within that transaction, invokes the ACMS task, and passes the transaction identifier. (In the DECdta architecture, the QTI implements the functions of a request manager for queued requests.)[1] The task then joins that transaction. The removal from the queue is atomic with the commit of the task, and no task entry is lost or executed twice.

Figure 5 shows the ACMS user-written agent model for off-line processing. With the ACMS system interface, users may write their own versions of the command process. Note that because these agents cannot be safely stored on disks, this method is generally not as reliable as using queues. User-written agents can be used, however, with DECdtm and the fault-tolerant VAXft 3000 system to produce a reliable front-end system. To do so, a user writes an agent that captures the input for the task and then starts a DECdtm transaction. The agent uses the system interface services to invoke the ACMS task and passes the transaction identifier and the input data. When the task call completes, the agent commits the transaction. If DECdtm returns an error on the commit, the agent loops back to start another transaction and to resubmit the task. If a VAXcluster system is used for the application, this configuration will survive any single point of failure.

DECintact Off-line Execution

The DECintact monitor provides several facilities for applications to perform off-line processing. These facilities allow applications to

- Interface with and process data from nonterminal devices and asynchronous events
- Control transaction capture, store and forward, interprocess communication, and business work flow through the DECintact queuing subsystem

Off-line Multithreading Off-line, multithreaded DECintact applications are typically used to service

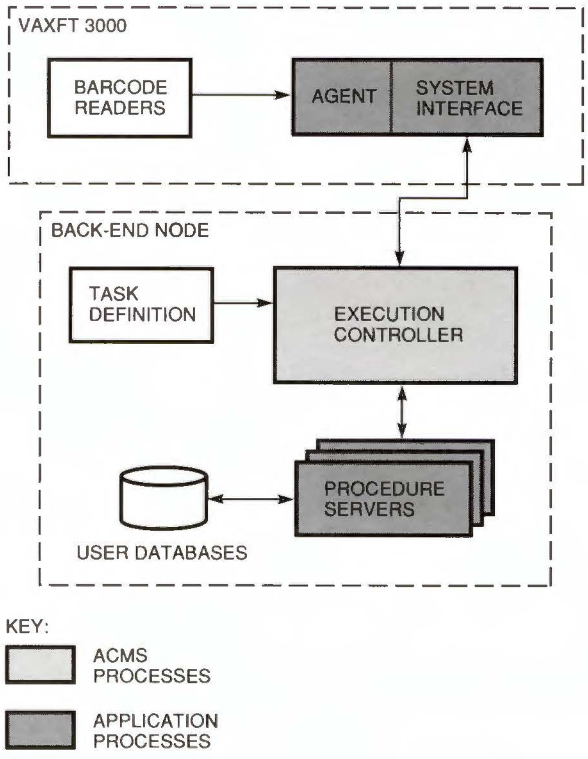


Figure 5 ACMS User-written Agent Model for Off-line Processing

asynchronous events, such as the arrival of an electronic funds transfer message or the addition to the queue of an item already on a DECintact queue. The application programmer explicitly controls how many threads are created, when they are created, and which execution path or paths each thread will follow. Off-line, multithreaded applications are well-suited to message switching systems and other aspects of electronic funds transfer in which each thread may be dedicated to servicing a different kind of event.

DECintact Queues The primary goal of the DECintact queuing subsystem is to support a work flow model of business transactions. (In the DECdta architecture, the DECintact queuing subsystem implements the functions of a queue resource manager and request initiator for queued requests.)[1] In a typical DECintact application that relies on queuing, the state of the business transaction may be represented by the queue on which a particular queue item resides at the moment. An item moves from queue to queue as the item's processing state changes, much as a work item moves from desk to desk. The superset of queue items that reside on queues throughout the application at any one time represents the state of transactions currently executing. Depending on the number of programs that need to process data during the course of a transaction, a queue item may be inserted on several different queues before the transaction completes. The application also may wish to chain together several small transactions within the context of a larger business transaction. The DECintact queuing system functions throughout the application: from the front end, where queues collect and route incoming data; to the back end, where queues can be integrated with data files in recovery units; and in between, where different programs in the application can use queues to share data.

The DECintact queuing subsystem consists of a comprehensive set of callable services for the creation and manipulation of queues, queue sets, and queue items. Queue item operations performed within the context of a DECintact transaction are fully atomic along with DECintact file operations.

In addition to overall workflow control, the DECintact queuing system allows the following:

- Deferred processing — An item can be queued by one process and then removed from the queue later by another process for processing. Deferred processing is useful when the volume of data entry is concentrated at particular times of day; applications can assign themselves to one or more queues and can be notified when an item is inserted on the queue.
- Store-and-forward processing — When users at the front end of the system write items to local queues, data entry can be continuous in the event of back-end system failure or whenever a program that is needed to process data is temporarily unavailable.
- Interprocess communication — Locally between applications sharing a node and by means of the DECintact remote queuing facility, applications can use the queuing system to reliably exchange application data between processes and applications.

A fundamental difference between ACMS queues and DECintact queues is that the ACMS system inserts tasks onto the queues, and the DECintact system inserts data items. In DECintact queuing, each data item contains both user-supplied data and a header that includes an item key and other control information. The header is used by the queuing system to control the movement of the item from queue to queue. Each queue item can be assigned an item priority. Items can be removed from the queue in FIFO order, in FIFO order within item priority, or by direct access using the item key. Queues can be stopped and started for insertion, removal, or both. Queues can also be redirected transparently at the system management level to running applications.

In the DECintact monitor, alert thresholds can be specified on a queue-by-queue basis to alert the system manager when queue levels reach defined amounts. Individual queue items can be held against removal or released. Queues can be grouped together into logical entities, called queue sets, which look and behave to the application the same as individual queues. Queue sets have added facilities for broadcast insertion on all members of a queue set and a choice of removing algorithms that can weight relative item- and queue-level priorities from the queue.

DECintact queues can be automatically distributed. At the system management level, a local queue can be designated as remote outbound. That is to say, items added to this queue are shipped transparently across the network to a corresponding remote inbound queue on the destination node. The transfer is handled by the DECintact queuing system by using exactly-once semantics (i.e., the item is guaranteed to be sent once and only once). From the point of view of the application that is adding or removing items from the queue, remote queues behave exactly as local queues behave.

To better understand some of the uses for DECintact queuing, consider a simplified but representative electronic funds transfer example built on the DECintact monitor. Figure 6 shows the elements of such an application. In this application, transactions might be initiated either locally by clerks entering data into the system from user-generated documents or by an off-line application that receives data from another branch or bank. The transactions are verified or repaired by other clerks in a different department of the bank. The transactions are then sent to destination banks over one or more network services.

To implement this application, the developer uses queues to route, safely store, and synchronize data as it progresses through the system, and to prioritize data items. Data items are given priority levels, based on application-defined criteria, such as transfer amount, destination bank, or time-to-closing.

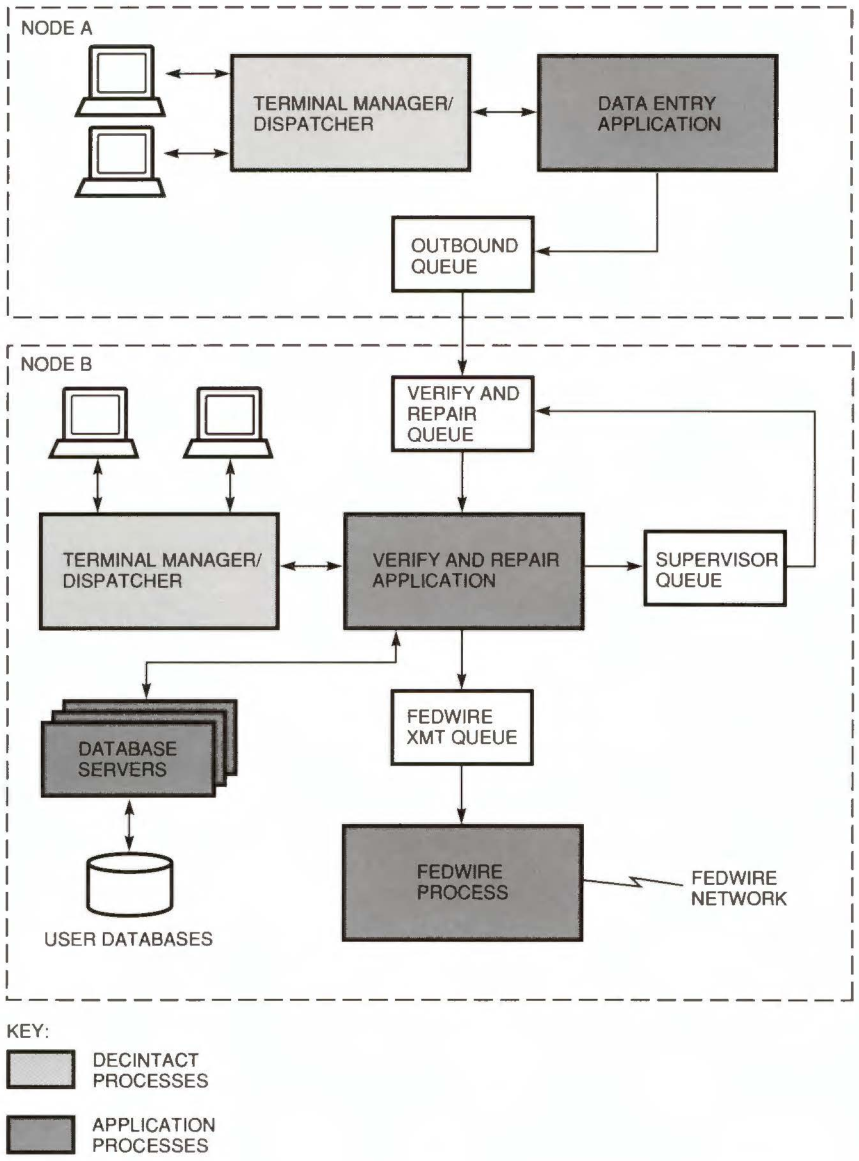


Figure 6 Elements of a DECintact Electronics Funds Transfer

As illustrated in Figure 6, the terminal manager controls terminals for the Data Entry and Verify and Repair applications. Clerks enter data from user-generated documents on-line as complete messages. Verification and repair clerks receive these messages as work items from the verify and repair queue through the Verify and Repair application. The result of verification is either a validated message, which is ultimately sent to a destination bank, or an unverifiable message, which is routed to the supervisor queue for special handling. After special handling, the message rejoins the processing flow by returning to the verify and repair queue. After validation, the messages are inserted in the Fedwire Xmt queue and sent over the network to the Federal Reserve System. The Fedwire Process application controls the physical interface to the communication line and implements the Fedwire protocol. The validated messages are also used to update a local database by means of database server programs.

The Fedwire Xmt queue could be defined as a queue set, which would permit the Fedwire Process application to remove items from the queue by a number of algorithms that bias the transfer amount by queue and item priority. Similarly, this queue set could be passively reprioritized near the close of the business day. In other words, the DECintact system administrator could use the DECintact queue utility near the end of the day to change queue-wide priorities and ensure that items with a higher priority level in the queue set would be sent over the Fedwire first, without changing any application code.

Application Management

Typically, transaction processing applications are crucial to the business running the applications. If the applications cannot perform their functions reliably or securely, business activity may have to cease altogether or be curtailed, as in the case of an inventory control application or electronic funds processing application. Therefore, the applications require additional controls to ensure that the applications and the access by users to the applications are limited to exactly what is needed for the business.

ACMS Application Management

Of the many features and tools for monitoring and controlling the system offered in the ACMS monitor, three areas are most often used.

- Controlling and restricting terminal user environments
- Controlling and restricting the application
- Ability to dynamically make changes to the application without stopping work

In addition to using the VMS user authorization file (VMS SYSUAF), the ACMS monitor provides utilities to define which users and terminals have access to the ACMS system. Controlled terminals are terminals defined by one of these utilities to be owned by the ACMS monitor. These terminals are allocated by the ACMS monitor when the ACMS system is started. When a user presses the Return key, the ACMS monitor displays its login prompt. Unless the user has login access, the VMS system cannot be accessed. The user's access is restricted to only those ACMS functions that the user is permitted to invoke. This restriction prevents a user from damaging the integrity of data on the system. The ACMS monitor also allows access support for terminals that are automatically logged in to the ACMS system, such as a terminal on a shop floor. Such access is useful for unprivileged users who are not accustomed to computers. They can enter data without understanding the process for logging in to the system.

For application control, the ACMS monitor uses a protected directory, ACMS$DIRECTORY, to store the application definition files. The application authorization utility (AAU) ensures that special authorization is required for a user to make changes to an application.

In the ACMS monitor, the application is a single point of control. The ACMS/START APPLICATION and ACMS/STOP APPLICATION commands cause the execution controller for the application to be created and deleted. An operator can control the times when an application is accessible. For example, an application can be controlled to run only on Fridays or only between certain hours. The control of access times can also be used to restrict access while changes or repairs are made to the application. This type of access control is difficult to achieve with only the VMS system because the VMS system does not provide these capabilities.

The execution controller does access-control list checking that is specified for each task. This mechanism can restrict user access by function. For example, a user could have the privilege to make a particular update to a database but not have access to read or make changes to any other parts of that database. The execution controller achieves a much finer level of control than do the mechanisms of the VMS system or the database system.

DECintact Application Management

The DECintact monitor controls access to the whole system and to individual tasks by means of a security subsystem. The subsystem adds transaction-processing-specific features to basic VMS security.

- User security profiles specify the DECintact user name and password (DECintact users are not required to have an entry in the VMS SYSUAF file); levels of security entitlement; inclusive and exclusive hours of permissible sign-on; menu entries authorized for the user. Only one user under a given DECintact user name can be signed on to the DECintact system at any one time on any one node.
- Dedicated terminal security profiles are used, in conjunction with user security profiles, to provide geographic entitlement.
- CAPTIVE and INITIAL_MENU user attributes restrict users to a specific menu level of functions and prevent the user from accessing outer levels.
- User-specific menus are menu entries for which an explicit authorization has been granted in the user profile and are the only menu items visible on the menu presented to terminal users. The DECintact monitor does include an exception for users who have an auditor privilege. Auditors can see all menu functions but must be specifically authorized to execute any single function.
- The subsystem provides the ability to dynamically enable or disable specific menu functions.
- Password revalidation is an attribute that can be associated with a menu function. If set, the user must reenter the DECintact user name and password before being allowed to access the function.

The DECintact monitor supports both controlled or dedicated terminals and terminals assigned LAT terminal server application ports, as does the ACMS monitor. These terminals are owned by, and allocated to, the DECintact system. When a user types any character at these terminals, a DECintact sign-on screen is displayed, and the user is prevented from logging in to the VMS system.

Geographic entitlement limits certain DECintact terminal-based functions to certain terminals or even to certain users on certain terminals. The three elements in geographic entitlement are as follows:

- The user security profile enables a function to be accessed by a certain user.
- The terminal security profile enables a function to be accessed at a certain terminal.
- A GEOG attribute is associated with a menu entry in the terminal manager/dispatcher's menu database. This attribute, when associated with a function, demands that there be an applicable terminal security profile before the function can be accessed.

Normally, if a function is enabled in a user profile, the user can access the function without further checks. If the GEOG attribute is associated with the function, however, that function must be enabled in the user profile and in the terminal profile before it can be accessed.

Geographic entitlement is frequently a requirement in financial environments which have specific and rigid security protocols. For example, a bank officer may be authorized to execute certain sensitive functions available only at dedicated terminals when the officer is signed-in at the home office. The same officer may be authorized to execute only a subset of less sensitive functions when signed-in from a branch office. Such sensitive functions can be protected by requiring that the user profile and the dedicated terminal profile enable the function.

Applications and resources are controlled within the context of a DECintact copy's run-time and management environment. Multiple copies can be established on the same VMS system. Different groups of users can maintain a certain level of autonomy (e.g., separate applications and data files), but all users can also share some or all functions and resources of a given DECintact version. A typical example of this concept, that is, the ability to create multiple DECintact copies for isolation and partitioning, is the common practice of establishing development, acceptance testing, and production DECintact environments. Managing applications and resources within a development environment, for example, can differ from managing applications and resources within a production environment with a different system manager.

Access to menu functions is controlled by the INTACT MANAGE DISABLE/ENABLE command. This command removes or restores specified functions dynamically from all menus in the DECintact copy and disables or enables their selection by subsequent users. (Current accessors of the specified function are allowed to complete the function.) The execution of single- and multithreaded applications or DECintact system components can be shut down by the INTACT MANAGE SHUTDOWN command. This command issues a mailbox request to the application or component, which then initiates an orderly shutdown. Access to the system by inclusive and exclusive time of day is controlled on a per-user basis through the DECintact security subsystem. In addition to these commands and functions, the queuing subsystem is managed by means of a queue management utility. This utility creates and deletes queues and queue sets, modifies queue and queue set attributes, and performs all other functions necessary for managing the DECintact queuing subsystem.

In general, the DECintact monitor's security and application control focuses on the front end by concentrating access checking at the point of system sign-in and menu generation. The ACMS system concentrates more on the back-end parts of the system by means of VMS access control lists (ACL) on specified tasks. The ACMS approach is built on VMS security and system access (the SYSUAF file) and reflects an environment in which the VMS system and the transaction processing security functions are typically performed by the same system management agency. The DECintact monitor's system access is handled more independently of the VMS system and reflects an environment in which transaction-processing-specific security functions may be performed by a different department from those of the general VMS security system.

Conclusion

The ACMS and DECintact transaction processing monitors provide a unified set of transaction-processing-specific services to the application environment. A large functional overlap exists between the services each monitor provides. Where the functions provided by each monitor are identical or similar (e.g., client/server database access and support for DECforms), the factors that distinguish one from the other are primarily a result of the use of 4GL and 3GL application programming styles and interfaces. Where notable functional differences remain (as in each product's respective queuing or security systems), the differences are primarily ones of emphasis rather than functional incompatibility. The set of common features shared by both monitors has been growing with the latest releases of the ACMS and DECintact monitors. This external convergence has been fostered and made possible by an internal convergence, which is based on sharing the underlying code that supports the common features of each monitor. As more common features are introduced and enhanced in the DECtp system, the investment in applications built on either monitor can be protected and the distinctive programming styles of both can be preserved.

Reference

1. P. A. Bernstein, W. T. Emberton, and V. Trehan, "DECdta — Digital's Distributed Transaction Processing Architecture," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 10-17.

William A. Laing
James E. Johnson
Robert V. Landau

Transaction Management Support in the VMS Operating System Kernel

Distributed transaction management support is an enhancement to the VMS operating system. This support provides services in the VMS operating system for atomic transactions that may span multiple resource managers, such as those for flat files, network databases, and relational databases. These transactions may also be distributed across multiple nodes in a network, independent of the communications mechanisms used by either the application programs or the resource managers. The Digital distributed transaction manager (DECdtm) services implement an optimized variant of the two-phase commit protocol to ensure transaction atomicity. Additionally, these services take advantage of the unique VAXcluster capabilities to greatly reduce the potential for blocking that occurs with the traditional two-phase commit protocol. These features, now part of the VMS operating system, are readily available to multiple resource managers and to many applications outside the traditional transaction processing monitor environment.

Businesses are becoming critically dependent on the availability and integrity of data stored on computer systems. As these businesses expand and merge, they acquire ever greater amounts of on-line data, often on disparate computer systems and often in disparate databases. The Digital distributed transaction manager (DECdtm) services described in this paper address the problem of integrating data from multiple computer systems and multiple databases while maintaining data integrity under transaction control.

The DECdtm services are a set of transaction processing features embedded in the VMS operating system. These services support distributed atomic transactions and implement an optimized variant of the well-known, two-phase commit protocol.

Design Goals

Our overall design goal was to provide base services on which higher layers of software could be built. This software would support reliable and robust applications, while maintaining data integrity.

Many researchers report that an atomic transaction is a very powerful abstraction for building robust applications that consistently update data.[1,2] Supporting such an abstraction makes it possible both to respond to partial failures and to maintain data consistency. Moreover, a simplifying abstraction is crucial when one is faced with the complexity of a distributed system.

With increasingly reliable hardware and the influx of more general-purpose, fault-tolerant systems, the focus on reliability has shifted from hardware to software.[3] Recent discussions indicate that the key requirements for building systems with a 100-year mean time between failures may be (1) software-fault containment, using processes, and (2) software-fault masking, using process checkpointing and transactions.[4]

It was clear that we could use transactions as a pervasive technique to increase application availability and data consistency. Further, we saw that this technique had merit in a general-purpose operating system that supports transaction processing, as well as timesharing, office automation, and technical computing.

The design of DECdtm services also reflects several other Digital and VMS design strategies:

- Pervasive availability and reliability. As organizations become increasingly dependent on their information systems, the need for all applications to be universally available and highly reliable increases. Features that ensure application availability and data integrity, such as journaling and two-phase commit, must be available to *all* applications, and not limited to those traditionally thought of as "transaction processing."

- Operating environment consistency. Embedding features in the operating system that are required by a broad range of utilities ensures consistency in two areas: first, in the functionality across all layered software products, and, second, in the interface for developers. For instance, if several distributed database products require the two-phase commit protocol, incorporating the protocol into the underlying system allows programmers to focus on providing "value-added" features for their products instead of re-creating a common routine or protocol.
- Flexibility and interoperability. Our vision includes making DECdtm interfaces available to any developer or customer, allowing a broad range of software products to take advantage of the VMS environment. Future DECdtm services are also being designed to conform to de facto and international standards for transaction processing, thereby ensuring that VMS applications can interoperate with applications on other vendors' systems.

Transaction Manager — Some Definitions

To grasp the concept of transaction manager, some basic terms must first be understood:

- Resource manager. A software entity that controls both the access and recovery of a resource. For example, a database manager serves as the resource manager for a database.
- Transaction. The execution of a set of operations with the properties of atomicity, serializability, and durability on recoverable resources.
- Atomicity. Either all the operations of a transaction complete, or the transaction has no effect at all.
- Serializability. All operations that executed for the transaction must appear to execute serially, with respect to every other transaction.
- Durability. The effects of operations that executed on behalf of the transaction are resilient to failures.

A transaction manager supports the transaction abstraction by providing the following services:

- Demarcation operations to start, commit, and abort a transaction
- Execution operations for resource managers to declare themselves part of a transaction and for transaction branch managers to declare the distribution of a transaction
- Two-phase commit operations for resource managers and other transaction managers to change the transaction state (to either "preparing" or "committing") or to acknowledge receipt of a request to change state

Benefits of Embedding Transaction Semantics in the Kernel

Several benefits are achieved by embedding transaction semantics in the kernel of the VMS operating system. Briefly, these benefits include consistency, interoperability, and flexibility. Embedding transaction semantics in the kernel makes a set of services available to different environments and products in a consistent manner. As a consequence, interoperability between products is encouraged, as well as investment in the development of "value-added" features. The inherent flexibility allows a programmer to choose a transaction processing monitor, such as VAX ACMS, and to access multiple databases anywhere in the network. The programmer may also write an application that reads a VAX DBMS CODASYL database, updates an Rdb/VMS relational database, and writes report records to a sequential VAX RMS file — all in a single transaction. Because all database and transaction processing products use DECdtm services, a failure at any point in the transaction causes all updates to be backed out and the files to be restored to their original state.

Two-phase Commit Protocol

DECdtm services use an optimized variant of the technique referred to as two-phase commit. The technique is a member of the class of protocols known as Atomic Commit Protocols. This class guarantees two outcomes: first, a single yes or no decision is reached among a distributed set of participants; and, second, this decision is consistently propagated to all participants, regardless of subsequent machine or communications failures. This guarantee is used in transaction processing to help achieve the atomicity property of a transaction.

The basic two-phase commit protocol is straightforward and well known. It has been the subject of considerable research and technical literature for several years.[5,6,7,8,9] The following section describes in detail this general two-phase commit protocol for those who wish to have more information on the subject.

The Basic Two-phase Commit Protocol

The two-phase commit protocol occurs between two types of participants: one coordinator and one or more subordinates. The coordinator must arrive at a yes or no decision (typically called the "commit decision") and propagate that decision to all subordinates, regardless of any ensuing failures. Conversely, the subordinates must maintain certain guarantees (as described below) and must defer to the coordinator for the result of the commit decision. As the name suggests, two-phase commit occurs in two distinct phases, which the coordinator drives.

In the first phase, called the prepare phase, the coordinator issues "requests to prepare" to all subordinates. The subordinates then vote, either a "yes vote" or a "veto." Implicit in a "yes vote" is the guarantee that the subordinate will neither commit nor abort the transaction (decide yes or no) without an explicit order from the coordinator. This guarantee must be maintained despite any subsequent failures and usually requires the subordinate to place sufficient data on disk (prior to the "yes vote") to ensure that the operations can be either completed or backed out.

The second phase, called the commit phase, begins after the coordinator receives all expected votes. Based on the subordinate votes, the coordinator decides to commit if there are no "veto" votes; otherwise, it decides to abort. The coordinator propagates the decision to all subordinates as either an "order to commit" or an "order to abort."

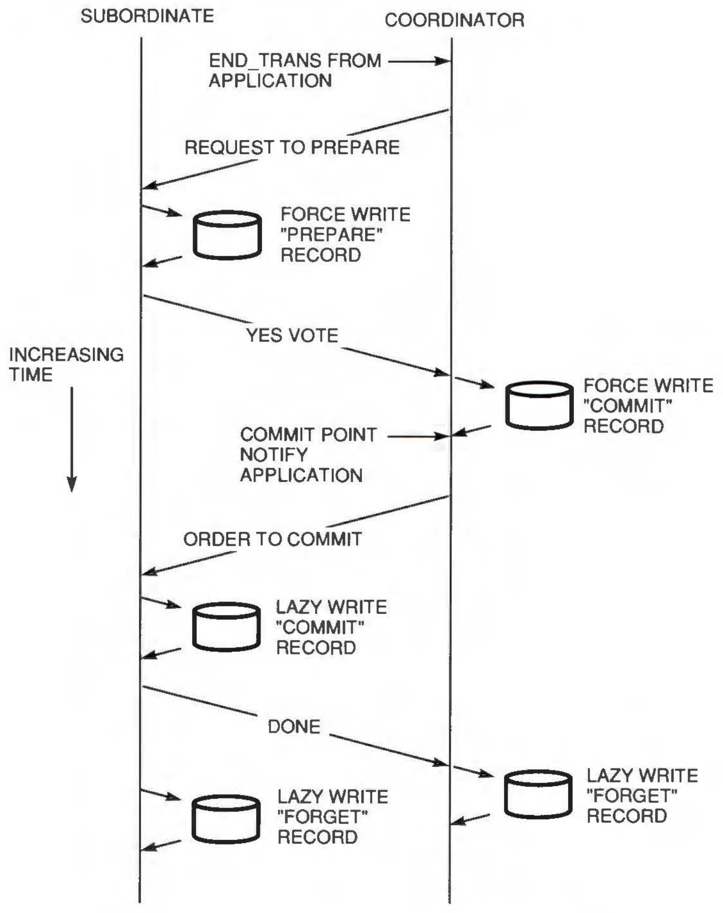

SUBORDINATE
COORDINATOR
END_TRANS FROM APPLICATION
REQUEST TO PREPARE
FORCE WRITE "PREPARE" RECORD
YES VOTE
INCREASING TIME
FORCE WRITE "COMMIT" RECORD
COMMIT POINT NOTIFY APPLICATION
ORDER TO COMMIT
LAZY WRITE "COMMIT" RECORD
DONE
LAZY WRITE "FORGET" RECORD
LAZY WRITE "FORGET" RECORD

Figure 1 Simple Two-phase Commit Time Line

Because the coordinator's decision must survive failures, a record of the decision is usually stored on disk before the orders are sent to the subordinates. When the subordinates complete processing, they send an acknowledgment back to the coordinator that they are "done." This allows the coordinator to reclaim disk storage from completed transactions. Figure 1 shows a time line of the two-phase commit sequence.

A subordinate node may also function as a superior (intermediate) node to follow-on subordinates. In such cases, there is a tree-structured relationship between the coordinator and the full set of subordinates. Intermediate nodes must propagate the messages down the tree and collect responses back up the tree. Figure 2 shows a time line for a two-phase commit sequence with an intermediate node.

Most of us have had direct contact with the two-phase commit protocol. It occurs in many activities. Consider the typical wedding ceremony as presented below, which is actually a very precise two-phase commit.

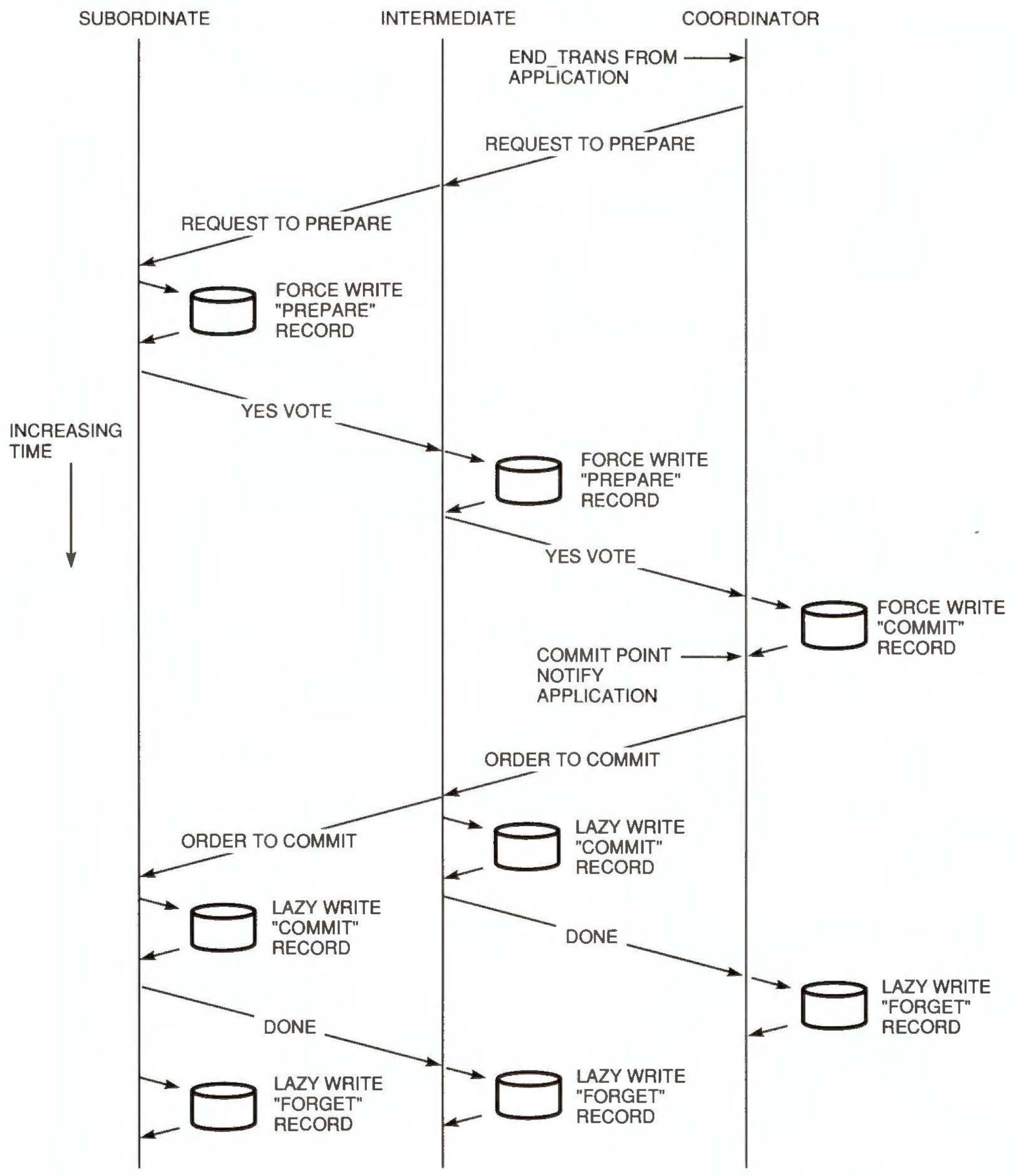

SUBORDINATE
INTERMEDIATE
COORDINATOR
END_TRANS FROM APPLICATION
REQUEST TO PREPARE
REQUEST TO PREPARE
FORCE WRITE "PREPARE" RECORD
YES VOTE
INCREASING TIME
FORCE WRITE "PREPARE" RECORD
YES VOTE
FORCE WRITE "COMMIT" RECORD
COMMIT POINT NOTIFY APPLICATION
ORDER TO COMMIT
ORDER TO COMMIT
LAZY WRITE "COMMIT" RECORD
LAZY WRITE "COMMIT" RECORD
DONE
LAZY WRITE "FORGET" RECORD
DONE
LAZY WRITE "FORGET" RECORD
LAZY WRITE "FORGET" RECORD

Figure 2 Three-node Two-phase Commit Time Line

Official:	Will you, Mary, take John...?
Bride:	I will.
Official:	Will you, John, take Mary...?
Groom:	I will.
Official:	I now pronounce you man and wife.

The above dialog can be viewed as a two-phase commit:

Coordinator:	Request to Prepare?
Participant 1:	Yes Vote.
Coordinator:	Request to Prepare?
Participant 2:	Yes Vote.
Coordinator:	Commit Decision.
	Order to Commit.

The basic two-phase commit protocol is straightforward, survives failures, and produces a single, consistent yes or no decision. However, this protocol is rarely used in commercial products. Optimizations are often applied to minimize message exchanges and physical disk writes. These optimizations are important particularly to the transaction processing market because the market is very performance sensitive, and two-phase commit occurs after the application is complete. Thus, two-phase commit is reasonably considered an added overhead cost. We have endeavored to reduce the cost in a number of ways, resulting in low overhead and a scalable protocol embodied in the DECdtm services. Some of the optimizations are described later in another section.

Components of the DECdtm Services

The DECdtm services were developed as three separate components: a transaction manager, a log manager, and a communication manager. Together, these components provide support for distributed transaction management. The transaction manager is the central component. The log manager services enable the transaction manager to store data on nonvolatile storage. The communication manager provides a location-independent interprocess communication service used by the transaction and log managers. Figure 3 shows the relationships among these components.

The Digital Distributed Transaction Manager

As the central component of the DECdtm services, the transaction manager is responsible for the application interface to the DECdtm services. This section presents the system services the transaction manager comprises.

The transaction coordinator is the core of the transaction manager. It implements the transaction state machine and knows which resource managers and subordinate transaction managers are involved in a transaction. The coordinator also controls what is written to nonvolatile storage and manages the volatile list of active transactions.

The user services are routines that implement the START_TRANSACTION, END_TRANSACTION, and ABORT_TRANSACTION transaction system services.

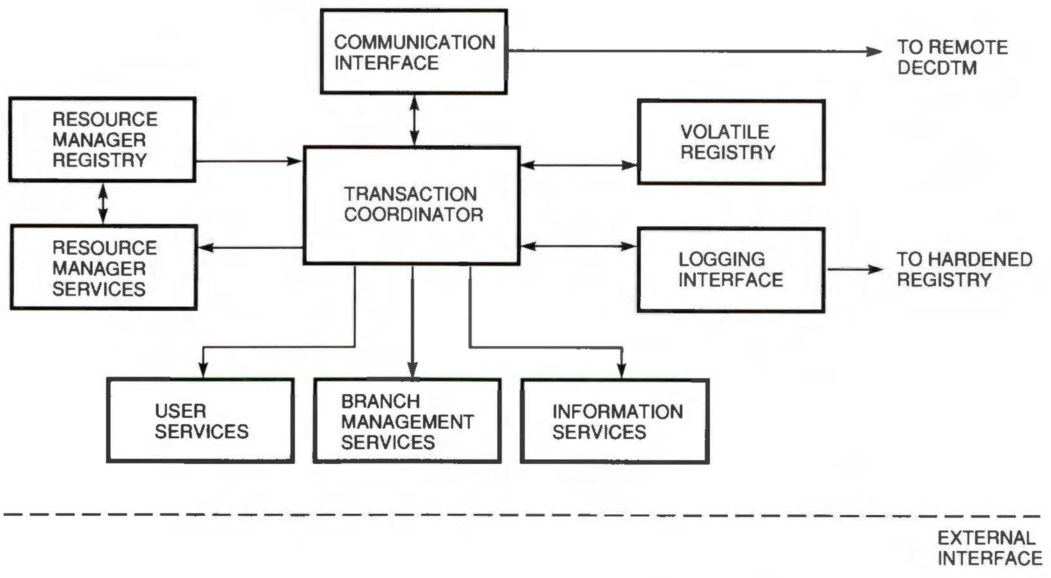


Figure 3 Components of the DECdtm Services

They validate user parameters, dispense a transaction identifier, pass state transition requests to the transaction coordinator, and return information about the transaction outcome.

The branch management services support the creation and demarcation of branches in the distributed transaction tree. New branches are constructed when subordinate application programs are invoked in a distributed environment. The services are called on to attach an application program to the transaction, to demarcate the work done in that application as part of the transaction, and finally to return information about the transaction outcome.

The resource manager services are routines that provide the interface between the DECdtm services and the cooperating resource managers. This interface allows resource managers to declare themselves to the transaction manager and to register their involvement in the "voting" stage of the two-phase commit process of a specific transaction.

Finally, the information services routines are the interface that allows resource managers to query and update transaction information stored by DECdtm services. This information is stored in either the volatile-active transaction list or the nonvolatile transaction log. Resource managers may resolve and possibly modify the state of "in-doubt" transactions through these services.

The Log Manager

The log manager provides the transaction manager with an interface for storing sufficient information in nonvolatile storage to ensure that the outcome of a transaction can be consistently resolved. This interface is available to operating system components. The log manager also supports the creation, deletion, and general management of the transaction logs used by the transaction manager. An additional utility enables operators to examine transaction logs and, in extreme cases, makes it possible to change the state of any transaction.

The Communication Manager

The communication manager provides a command/response message-passing facility to the transaction manager and the log manager. The interface is specifically designed to offer high-performance, low-latency services to operating system components. The command/response, connection-oriented, message-passing system allows clients to exchange messages. The clients may reside on the same node, within the same cluster, or within a homogeneous VMS wide area network. The communication manager also provides highly optimized local (that is, intranode) and intracluster transports. In addition, this service component multiplexes communication links across a single, cached DECnet virtual circuit to improve the performance of creating and destroying wide area links.

Transaction Processing Model

Digital's transaction processing model entails the cooperation of several distinct elements for correct execution of a distributed transaction. These elements are (1) the application programmer, (2) the resource managers, (3) the integration of the DECdtm services into the VMS operating system, (4) transaction trees, and (5) vote-gathering and the final outcome.

Application Programmer

The application programmer must bracket a series of operations with START_TRANSACTION and END_TRANSACTION calls. This bracketing demarcates the unit of work that the system is to treat as a single atomic unit. The application programmer may call the DECdtm services to create the branches of the distributed transaction tree.

Resource Managers

Resource managers, such as VAX RMS, VAX Rdb/VMS, and VAX DBMS, that access recoverable resources during a transaction inform the DECdtm services of their involvement in the transaction. The resource managers can then participate in the voting phase and react appropriately to the decision on the final outcome of the transaction. Resource managers must also provide recovery mechanisms to restore resources they manage to a transaction-consistent state in the event of a failure.

Integration in the Operating System

The DECdtm services are a basic component of the VMS operating system. These services are responsible for maintaining the overall state of the distributed transaction and for ensuring that sufficient information is recorded on stable storage. Such information is essential in the event of a failure so that resource managers can obtain a consistent view of the outcome of transactions.

Each VMS node in a network normally contains one transaction manager object. This object maintains a list of participants in transactions that are active on the node. This list consists of resource managers local to the node and the transaction manager objects located on other nodes.

Transaction Trees

The node on which the transaction originated (that is, the node on which the START_TRANSACTION service was called) may be viewed as the "root" of a distributed transaction tree. The transaction manager object on this node is usually responsible for coordinating the transaction commit phase of the transaction. The transaction tree grows as applications call on the branch management services of the transaction manager object.

The transaction identifier dispensed by the START_TRANSACTION service is an input parameter to the branch services. This parameter identifies two concerns for the local transaction manager object: (1) to which transaction tree the new branch should be added, and (2) which transaction manager object is the immediate superior in the tree.

Resource managers join specific branches in a transaction tree by calling the resource manager services of the local transaction manager object.

Vote-gathering and the Final Outcome

When the "commit" phase of the transaction is entered (triggered by an application call to END_TRANSACTION), each transaction manager object involved in the transaction must gather the "votes" of the locally registered resource managers and the subordinate transaction manager objects. The results are forwarded to the coordinating transaction manager object.

The coordinating transaction manager object eventually informs the locally registered resource managers and the subordinate transaction manager objects of the final outcome of the transaction. The subordinate transaction manager objects, in turn, propagate this information to locally registered resource managers as well as to any subordinate transaction manager objects.

Protocol Optimizations

The DECdtm services use several previously published optimizations and extend those optimizations with a number that are unique to VAXcluster systems. In this section we present these general optimizations, a discussion of VAXcluster considerations, and two VAXcluster-specific optimizations.

General Optimizations

The following sections describe some previously published optimizations.

Presumed Abort DECdtm services use the "presumed abort" optimization.[8,9] This optimization states that, if no information can be found for a transaction by the coordinator, the transaction aborts. This removes the need to write an abort decision to disk and to subsequently acknowledge the order to abort. In addition, subordinates that do not modify any data during the transaction (that is, they are "read only"), avoid writing information to disk or participating in the commit phase.

Lazy Commit Log Write The DECdtm services can act as intermediate nodes in a distributed transaction. In this mode, they write a "prepare" record prior to responding with a "yes vote." They also write a "commit" record upon receipt of an order to commit. This latter record is written so that the coordinator need not be asked about the commit decision should the intermediate node fail. This refinement isolates the intermediate node's recovery from communication failures between it and the coordinator.

Performance is enhanced when the DECdtm services write the commit record on an intermediate node in a "nonurgent" or "lazy" manner.[10] The lazy write buffers the information and waits for an urgent request to trigger the group commit timer to write the data to disk. Typically, this operation avoids a disk write at the intermediate node. The increase in the length of time before the commit record is written is negligible.

One-phase Commit A key consideration in the design of the DECdtm services was to incur minimal impact on the performance of Digital's database products. We exploited two attributes to achieve this goal. First, all current users are limited to non-distributed transactions (those that involve only a single subordinate). Second, the two-phase commit protocol requires that all subordinates respond with a "yes vote" to commit the transaction. This allows a highly optimized path for single subordinate transactions. Such transactions require no writes to disk by the DECdtm services and execute in one phase. The subordinate is told that it is the only voting party in the transaction and, if it is willing to respond with a "yes vote," it should proceed and perform its order to commit processing.

VAXcluster Considerations

The optimizations listed above (and others not described here) provide the DECdtm services with a competitive two-phase commit protocol. VAXcluster technology, though, offers other untapped potential. VAXcluster systems offer several unique features, in particular, the guarantee against partitioning, the distributed lock manager, and the ability to share disk access between CPUs.[11]

Within a VAXcluster system, use of these unique features allows the DECdtm services to avoid a blocked condition which occurs during the short period of time when a subordinate node responds with a "yes vote" and communication with its coordinator is lost. Normally, the subordinate is unable to proceed with that transaction's commit until communications have been restored.

Outside a VAXcluster system, the DECdtm services would indeed be blocked. If, however, the subordinate and its coordinator are in the same VAXcluster system, this will not occur. If communication is lost, a subordinate node knows, as a result of the guarantee against partitioning, that its coordinator has failed.

Because a subordinate node can access the transaction log of the failed coordinator, it may immediately "host" its failed coordinator's recovery. Communications to the hosted coordinator are quickly restored, and the subordinate node is able to complete the transaction commit.

VAXcluster-specific Optimizations

Once the blocking potential was removed from intra-VAXcluster transactions, several additional protocol optimizations became practical. The optimizations described in this section are dynamically enabled if the subordinate and its coordinator are both in the same VAXcluster system.

Early Prepare Log Write As mentioned earlier, an intermediate node must write a "prepare" record prior to responding with a "yes vote." The presence of this record in an intermediate node's log indicates that the node must get the outcome of the transaction from the coordinator and, thus, it is subject to blocking. Therefore, the prepare record is typically written after all the expected votes are returned, which adds to commit-time latency.

The DECdtm services are free from blocking concerns within a VAXcluster system; the vast majority of transactions do commit. This factor prompted an optimization that writes a prepare record while simultaneously collecting the subordinate votes. This reduces commit-time latency.

No Commit Log Write The lazy commit log write optimization described above causes the intermediate node's commit record to be written and, thus, minimizes the potential for blocking should the intermediate node fail. Note that this is not a concern for the intra-VAXcluster case. Therefore, no commit record is written at the intermediate node.

Performance Evaluation

Table 1 describes the message and log write costs of the DECdtm services protocol and compares it to the basic two-phase commit protocol, as well as to the standard presumed abort variant previously described.[8,9]

Table 1 Logging and Message Cost by Two-phase Commit (2PC) Protocol Variant

	Coordinator		Intermediate	
Coordinator	Log Write	Message	Log Write	Message
Basic 2PC:	2, 1 forced	2N	2, 2 forced	2
Presumed Abort:	2, 1 forced	2N	2, 2 forced	2
(RO intermediate)	2, 1 forced	1N	0	1
Normal DECdtm:	2, 1 forced	2N	2, 1 forced	2
(RO intermediate)	2, 1 forced	1N	0	1
Intracluster:	2, 1 forced	2N	1, 1 forced*	2
(RO intermediate)	2, 1 forced	1N	0	1
DECdtm 1PC:	0	1	–	–

Notes:

Log writes are total writes, forced. The table entry 2,1 forced means that there are two total log writes, one of which is forced. A forced write must complete before the protocol makes a transition to the next state.

RO means Read Only.

Where a message is listed as xN, N represents the number of intermediates that fit that category.

* In this instance, forced means that the log write is initiated optimistically; thus, it has lower latency.

Ease-of-use Evaluation

A primary goal in providing transaction processing primitives within the VMS kernel was to supply many disparate applications with a straightforward interface to distributed transaction management. This contrasts with most commercially available systems, where distributed transaction management functionality is available only from a transaction processing monitor. This latter form restricts the functionality to applications written to execute under the control of the transaction processing monitor, and it effectively precludes other applications from making use of the technology.

From the outset of development, we endeavored to provide an interface that was suitable for as many applications as possible. We made early versions of the DECdtm services available within Digital to decrease the "time to market" for software products that wished to exploit distributed transaction processing technology. As of July 1990, at least seven Digital software products have been modified to use the DECdtm services. These products are VAX Rdb/VMS, VAX DBMS, VAX RMS Journaling, VAX ACMS, DECintact, VAX RALLY, and VAX SQL.

In general, the modifications to these products have been relatively minor, as might be inferred from the short time it took to make the required changes. Based on this experience, we expect third-party software vendors to rapidly take advantage of the DECdtm services as they become available as part of the standard VMS operating system.

To incorporate the DECdtm services into a recoverable resource manager, the existing internal transaction management module with calls to the DECdtm services must be replaced. The resource manager must also be modified to correctly respond to the prepare and commit callbacks by the DECdtm services. Further, the recovery logic of the resource manager must be modified to obtain from the DECdtm services the state of "in doubt" transactions.

Example of DECdtm Usage

The model and pseudocode shown in Figures 4a and b illustrate the use of DECdtm services in a simple example of a distributed transaction. The transaction spans two nodes, NODE_A and NODE_B, in a VMS network. During the course of the transaction, recoverable resources managed by resource managers, RM_A and RM_B, are modified. Two "application" programs, APPL_A and APPL_B, that run on NODE_A and NODE_B, respectively, make normal procedural calls to RM_A and RM_B. APPL_A

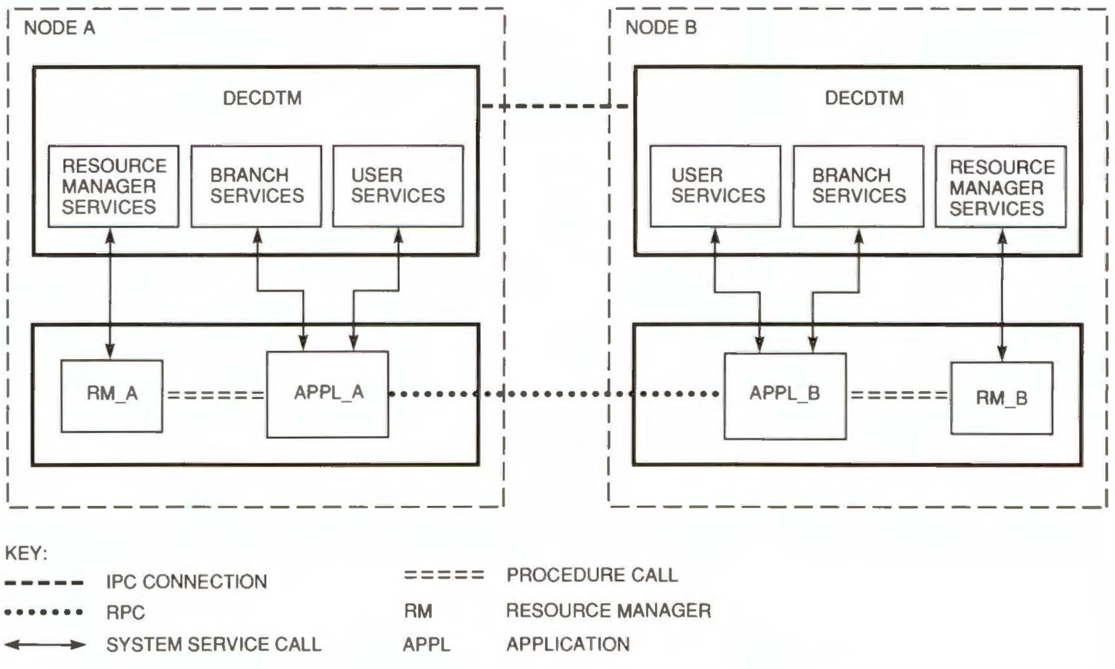


Figure 4a Model Illustrating the Use of DECdtm Services

```
PROGRAM APPL_A
    .
    .
    .
    ! Establish communications with remote application
    !
    IPC_LINK (node="NODE_B", application="APPL_B", link=link_id);

    ! Exchange transaction manager names
    !
    LIB$GETJPI (JPI$_COMMIT_DOMAIN,,,my_cd);
    IPC_TRANSCEIVE (link=link_id, send_data=my_cd,
                    receive_data=your_cd);
    .
    .
    .
    ! Start a transaction
    !
    $START_TRANSW (iosb=status, tid=tid);

    ! Make a procedural call to RM_A to perform an operation
    !
    RM_A (tid, requested_operation);

    ! Now create a transaction branch for the remote application
    !
    $ADD_BRANCHW (iosb=status, tid=tid, branch=bid,
                  cd_name=your_cd);

    ! Ask APPL_B to do something as part of this transaction
    !
    IPC_TRANSCEIVE (link=link_id, send_data=(tid, bid, data),
                    receive_data=status);

    ! And end the transaction
    !
    $END_TRANSW (iosb=status, tid=tid);


PROGRAM APPL_B (link_id)

    ! Exchange transaction manager names
    !
    IPC_RECEIVE (link=link_id, data=sup_cd);
    LIB$GETJPI (JPI$_COMMIT_DOMAIN,,,my_cd);
    IPC_REPLY (link=link_id, data=my_cd);

    ! Now we execute transaction requests
    !
    loop;
       IPC_RECEIVE (link=link_id, data=(tid, bid, data));
       ! Start the transaction branch created by APPL_A.
       !
       $START_BRANCHW (iosb=status, tid=tid, branch=bid,
                       cd_name=sup_cd);

       ! Make a procedural call to RM_B to perform an operation
       !
       RM_B (tid, requested_operation);

       ! Tell APPL_A we are done
       !
       IPC_REPLY (link=link_id, data=SS$_NORMAL);

       ! Declare that we are finished for this transaction and
       ! wait for it to complete
       !
       $READY_TO_COMMITW (iosb=status, tid=tid);
    end_loop;
```

```
ROUTINE RM_A (tid, requested_operation)

    ! If this is the first operation, register with DECdtm services as a
    ! resource manager. As part of the registration we declare an event
    ! routine that will be called during the voting process.
    !
    if first time we've been called then
        $DECLARE_RMW (iosb=status, name="RM_A", evtrtn=RM_A_EVENT,
                      rm_id=rm_handle);

    ! Inform DECdtm services of our interest in this transaction
    !
    if tid has not previously been seen then
        $JOIN_RMW (iosb=status, rm_id=rm_handle, tid=tid,
                   part_id=participant);

    ! Perform the requested operation
    !
    DO_OPERATION (requested_operation);
    RETURN


ROUTINE RM_A_EVENT (event_block)

    ! Select action from the DECdtm services event type
    !
    CASE event_block.DDTM$L_OPTYPE FROM ... TO ...
       .
       .
       .

       ! Do "request to prepare" processing
       !
       [DDTM$K_PREPARE]:
           DO_PREPARE_ACTIVITY (result=status, tid=event_type.DDTM$A_TID);

       ! Do "order to commit" processing
       !
       [DDTM$K_COMMIT]:
           DO_COMMIT_ACTIVITY (result=status, tid=event_type.DDTM$A_TID);

       ! Do "order to abort" processing
       !
       [DDTM$K_ABORT]:
           DO_ABORT_ACTIVITY (result=status, tid=event_type.DDTM$A_TID);

       .
       .
       .
    ESAC;

    ! Inform the DECdtm services of the final status of our event
    ! processing.
    !
    $FINISH_RMOPW (iosb=iosb, part_id=event_type.DDTM$L_PART_ID,
                   retsts=status);
    RETURN
```

Figure 4b Pseudocode Illustrating the Use of DECdtm Services

and APPL_B use an interprocess communication mechanism to communicate information across the network. The DECdtm service calls are prefixed with a dollar sign ($).

The code for the resource managers, RM_A and RM_B, is identical with respect to calls for the DECdtm services. The resource manager routine, ROUTINE RM_A_EVENT, is invoked by the DECdtm services during transaction state transitions.

Conclusions

The addition of a distributed transaction manager to the kernel of the general-purpose VMS operating system makes distributed transactions available to a wide spectrum of applications. This design and implementation was accomplished with comparative ease and with quality performance. In addition to utilizing the most commonly described optimizations of the two-phase commit protocol, we have used optimizations that exploit some of the unique benefits of the VAXcluster system.

Acknowledgments

We wish to gratefully acknowledge the contributions of all the transaction processing architects involved, and in particular Vijay Trehan, for delivering to us an understandable and implementable architecture. We also extend our thanks to Phil Bernstein for his encouragement and advice, and to our initial users, Bill Wright, Peter Spiro, and Lenny Szubowicz, for their persistence and good nature.

Finally, and most importantly, we would like to thank all the DECdtm development engineers and the others who helped ship the product: Stuart Bayley, Cathy Foley, Mike Grossmith, Tom Harding, Tony Hasler, Mark Howell, Dave Marsh, Julian Palmer, Kevin Playford, and Chris Whitaker.

References

1. R. Haskin, Y. Malachi, W. Sawdon, and G. Chan, "Recovery Management in Quicksilver," *ACM Transactions on Computer Systems*, vol. 6, no. 1 (February 1988).

2. A. Spector et al., *Camelot: A Distributed Transaction Facility for Mach and the Internet — An Interim Report* (Pittsburgh: Carnegie Mellon University, Department of Computer Science, June 1987).

3. W. Bruckert, C. Alonso, and J. Melvin, "Verification of the First Fault-tolerant VAX System," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 79-85.

4. J. Gray, "A Census of Tandem System Availability between 1985 and 1990," Tandem Technical Report 90.1, part no. 33579 (January 1990).

5. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Reading, MA: Addison-Wesley, 1987).

6. J. Gray, "Notes on Database Operating Systems," In *Operating Systems: An Advanced Course* (Berlin: Springer-Verlag, 1978).

7. B. Lampson, "Atomic Transactions," In *Distributed Systems—Architecture and Implementation: An Advanced Course*, edited by G. Goos and J. Hartmanis (Berlin: Springer-Verlag, 1981).

8. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Transactions on Computer Systems*, vol. 11, no. 4 (December 1986).

9. C. Mohan and B. Lindsay, "Efficient Commit Protocol for the Tree of Processes Model of Distributed Transactions," *Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, August 1983).

10. D. Duchamp, "Analysis of Transaction Management Performance," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (Special issue), vol. 23, no. 5 (December 1989): 177-190.

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

Walter H. Kohler
Yun-Ping Hsu
Thomas K. Rogers
Wael H. Bahaa-El-Din

Performance Evaluation of Transaction Processing Systems

Performance and price/performance are important attributes to consider when evaluating a transaction processing system. Two major approaches to performance evaluation are measurement and modeling. TPC Benchmark A is an industry standard benchmark for measuring a transaction processing system's performance and price/performance. Digital has implemented TPC Benchmark A in a distributed transaction processing environment. Benchmark measurements were performed on the VAX 9000 Model 210 and the VAX 4000 Model 300 systems. Further, a comprehensive analytical model was developed and customized to model the performance behavior of TPC Benchmark A on Digital's transaction processing platforms. This model was validated using measurement results and has proven to be an accurate performance prediction tool.

Transaction processing systems are complex in nature and are usually characterized by a large number of interactive terminals and users, a large volume of on-line data and storage devices, and a high volume of concurrent and shared database accesses. Transaction processing systems require layers of software components and hardware devices to work in concert. Performance and price/performance are two important attributes for customers to consider when selecting transaction processing systems. Performance is important because transaction processing systems are frequently used to operate the customer's business or handle mission-critical tasks. Therefore, a certain level of throughput and response time guarantee are required from the systems during normal operation. Price/performance is the total system and maintenance cost in dollars, normalized by the performance metric.

The performance of a transaction processing system is often measured by its throughput in transactions per second (TPS) that satisfies a response time constraint. For example, 90 percent of the transactions must have a response time that is less than 2 seconds. This throughput, qualified by the associated response time constraint, is called the maximum qualified throughput (MQTh). In a transaction processing environment, the most meaningful response time definition is the end-to-end response time, i.e., the response time observed by a user at a terminal. The end-to-end response time represents the time required by all components that compose the transaction processing system.

The two major approaches used for evaluating transaction processing system performance are measurement and modeling. The measurement approach is the most realistic way of evaluating the performance of a system. Performance measurement results from standard benchmarks have been the most accepted form of performance assessment of transaction processing systems. However, due to the complexity of transaction processing systems, such measurements are usually very expensive, very time-consuming, and difficult to perform.

Modeling uses simulation or analytical modeling techniques. Compared to the measurement approach, modeling makes it easier to produce results and requires less computing resources. Performance models are also flexible. Models can be used to answer "what-if" types of questions and to provide insights into the complex performance behavior of transaction processing systems, which is difficult (if not impossible) to observe in the measurement environment. Performance models are widely used in research and engineering communities to provide valuable analysis of design alternatives, architecture evaluation, and capacity planning. Simplifying assumptions are usually made in the modeling approach. Therefore, performance models require validation, through detailed simulation or measurement, before predictions from the models are accepted.

This paper presents Digital's benchmark measurement and modeling approaches to transaction processing system performance evaluation. The paper includes an overview of the current industry standard transaction processing benchmark, the TPC Benchmark A, and a description of Digital's implementation of the benchmark, including the distinguishing features of the implementation and the benchmark methodology. The performance measurement results that were achieved by using the TPC Benchmark A are also presented. Finally, a multilevel analytical model of the performance behavior of transaction processing systems with response time constraints is presented and validated against measurement results.

TPC Benchmark A—An Overview

The TPC Benchmark A simulates a simple banking environment and exercises key components of the system under test (SUT) by using a simple, update-intensive transaction type. The benchmark is intended to simulate a class of transaction processing application environments, not the entire range of transaction processing environments. Nevertheless, the single transaction type specified by the TPC Benchmark A standard provides a simple and repeatable unit of work.

The benchmark can be run in either a local area network (LAN) or a wide area network (WAN) configuration. The related throughput metrics are tpsA-Local and tpsA-Wide, respectively. The benchmark specification defines the general application requirements, database design and scaling rules, testing and pricing guidelines, full disclosure report requirements, and an audit checklist.[1] The following sections provide an overview of the benchmark.

Application Environment

The TPC Benchmark A workload is patterned after a simplified banking application. In this model, the bank contains one or more branches. Each branch has 10 tellers and 100,000 customer accounts. A transaction occurs when a teller enters a deposit or a withdrawal for a customer against an account at a branch location. Each teller enters transactions at an average rate of one every 10 seconds. Figure 1 illustrates this simplified banking environment.

Transaction Logic

The transaction logic of the TPC Benchmark A workload can be described in terms of the bank environment shown in Figure 1. A teller deposits in or withdraws money from an account, updates the current cash position of the teller and branch, and makes an entry of the transaction in a history file. The pseudocode shown in Figure 2 represents the transaction.

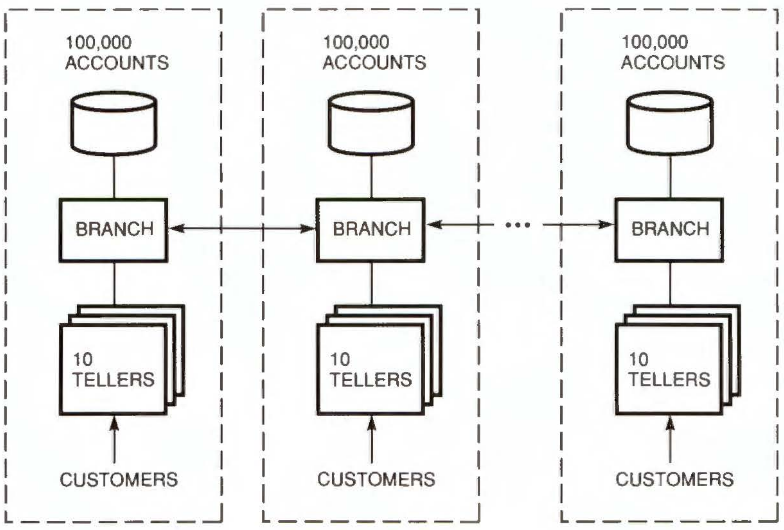


Figure 1 TPC Benchmark A Banking Environment

```
Read 100 bytes including Bid, Tid, Aid, Delta from terminal
BEGIN TRANSACTION
  Update Account where Account_ID = Aid:
    Read Account_Balance from Account
    Set Account_Balance = Account_Balance + Delta
    Write Account_Balance to Account
  Write to History:
    Aid, Tid, Bid, Delta, Time_Stamp
  Update Teller where Teller_ID = Tid:
    Set Teller_Balance = Teller_Balance + Delta
    Write Teller_Balance to Teller
  Update Branch where Branch_ID = Bid:
    Set Branch_Balance = Branch_Balance + Delta
    Write Branch_Balance to Branch
COMMIT TRANSACTION
Write 200 bytes including Aid, Tid, Delta, Account_Balance
to terminal
```

Figure 2 TPC Benchmark A Transaction Pseudocode

Terminal Communication

For each transaction, the originating terminal is required to transmit data to, and receive data from, the system under test. The data sent to the system under test must consist of at least 100 alphanumeric data bytes, organized as at least four distinct fields: Account_ID, Teller_ID, Branch_ID, and Delta. The Branch_ID identifies the branch where the teller is located. The Delta is the amount to be credited to, or debited from, the specified account. The data received from the system under test consists of at least 200 data bytes, organized as the above four input fields and the Account_Balance that results from the successful commit operation of the transaction.

Implementation Constraints

The TPC Benchmark A imposes several conditions on the test environment.

- The transaction processing system must support atomicity, consistency, isolation, and durability (ACID) properties during the test.
- The tested system must preserve the effects of committed transactions and ensure database consistency after recovering from
  - The failure of a single durable medium that contains datatbase or recovery log data
  - The crash and reboot of the system
  - The loss of all or part of memory
- Eighty-five percent of the accounts processed by a teller must belong to the home branch (the one to which the teller belongs). Fifteen percent of the accounts processed by a teller must be owned by a remote branch (one to which the teller does not belong). Accounts must be uniformly distributed and randomly selected.

Database Design

The database consists of four individual files/tables: Branch, Teller, Account, and History, as defined in Table 1. The overall size of the database is determined by the throughput capacity of the system. Ten tellers, each entering transactions at an average rate of one transaction every 10 seconds, generate what is defined as a one-TPS load. Therefore, each teller contributes one-tenth (1/10) TPS. The history area must be large enough to store the history records generated during 90 eight-hour days of operation at the published system TPS capacity. For a system that has a processing capacity of x TPS, the database is sized as shown in Table 2.

For example, to process 20 TPS, a system must use a database that includes 20 branch records, 200 teller records, and 2,000,000 account records. Because each teller uses a terminal, the price of the system must include 200 terminals. A test that results in a higher TPS rate is invalid unless the size of the database and the number of terminals are increased proportionately.

Table 1 Database Entities

Record	Bytes	Fields Required	Description
Branch	100	Branch_ID	Identifies the branch across the range of branches
		Branch_Balance	Contains the branch's current cash balance
Teller	100	Teller_ID	Identifies the teller across the range of tellers
		Branch_ID	Identifies the branch where the teller is located
		Teller_Balance	Contains the teller's current cash balance
Account	100	Account_ID	Identifies the customer account uniquely for the entire database
		Branch_ID	Identifies the branch where the account is held
		Account_Balance	Contains the account's current cash balance
History	50	Account_ID	Identifies the account updated by the transaction
		Teller_ID	Identifies the teller involved in the transaction
		Branch_ID	Identifies the branch associated with the teller
		Amount	Contains the amount of credit or debit (delta) specified by the transaction
		Time_Stamp	Contains the date and time taken between the BEGIN TRANSACTION and COMMIT TRANSACTION statements

Table 2 Database Sizing

Number of Records	Record Type
$1 \times x$	Branch records
$10 \times x$	Teller records
$100{,}000 \times x$	Account records
$2{,}592{,}000 \times x$	History records

Benchmark Metrics

TPC Benchmark A uses two basic metrics:

- Transactions per second (TPS) — throughput in TPS, subject to a response time constraint, i.e., the MQTh, is measured while the system is in a sustainable steady-state condition.
- Price per TPS (K$/TPS) — the purchase price and five-year maintenance costs associated with one TPS.

Transactions per Second To guarantee that the tested system provides fast response to on-line users, the TPC Benchmark A imposes a specific response time constraint on the benchmark. Ninety percent of all transactions must have a response time of less than two seconds. The TPC Benchmark A standard defines transaction response time as the time interval between the transmission from the terminal of the first byte of the input message to the system under test to the arrival at the terminal of the last byte of the output message from the system under test.

The reported TPS is the total number of committed transactions that both started and completed during an interval of steady-state performance, divided by the elapsed time of the interval. The steady-state measurement interval must be at least 15 minutes, and 90 percent of the transactions must have a response time of less than 2 seconds.

Price per TPS The K$/TPS price/performance metric measures the total system price in thousands of dollars, normalized by the TPS rating of the system. The priced system includes all the components that a customer requires to achieve the reported performance level and is defined by the TPC Benchmark A standard as the

- Price of the system under test, including all hardware, software, and maintenance for five years.
- Price of the terminals and network components, and their maintenance for five years.
- Price of on-line storage for 90 days of history records at the published TPS rate, which amounts to 2,592,000 records per TPS. A storage medium is considered to be on-line if any record can be accessed randomly within one second.
- Price of additional products required for the operation, administration, or maintenance of the priced systems.
- Price of products required for application development.

All hardware and software used in the tested configuration must be announced and generally available to customers.

TPC Benchmark A Implementation

Digital's implementation of the TPC Benchmark A goes beyond the minimum requirements of the TPC Benchmark A standard and uses Digital's distributed approach to transaction processing.[2] For example, Digital's TPC Benchmark A implementation includes forms management and transaction processing monitor software that are required in most real transaction processing environments but are not required by the benchmark. The following sections provide an overview of Digital's approach and implementation.

Transaction Processing Software Environment

The three basic functions of a general-purpose transaction processing system are the user interface (forms processing), applications management, and database management. Digital has developed a distributed transaction architecture (DECdta) to define how the major functions are partitioned and supported by components that fit together to form a complete transaction processing system. Table 3 shows the software components in a typical Digital transaction processing environment.

Table 3 Transaction Processing Software Components

Component	Example
Operating system	VMS
Communications	LAT, DECnet
Database	VAX Rdb/VMS
TP monitor	VAX ACMS, DECintact
Forms	DECforms
Application	COBOL

Distributed Transaction Processing Approach

Digital transaction processing systems can be distributed by placing one or more of the basic system functions (i.e., user interface, application manager, database manager) on separate computers. In the simplest form of a distributed transaction processing system, the user interface component runs on a front-end processor, and the application and database components run on a back-end processor. The configuration allows terminal and forms management to be performed at a remote location, whereas the application is processed at a central location. The Digital transaction processing software components are separable because their clearly defined interfaces can be layered transparently onto a network. How these components may be partitioned in the Digital distributed transaction processing environment is illustrated in Figure 3.

TPC Benchmark A Test Environment

The Digital TPC Benchmark A tests are implemented in a distributed transaction processing environment using the transaction processing

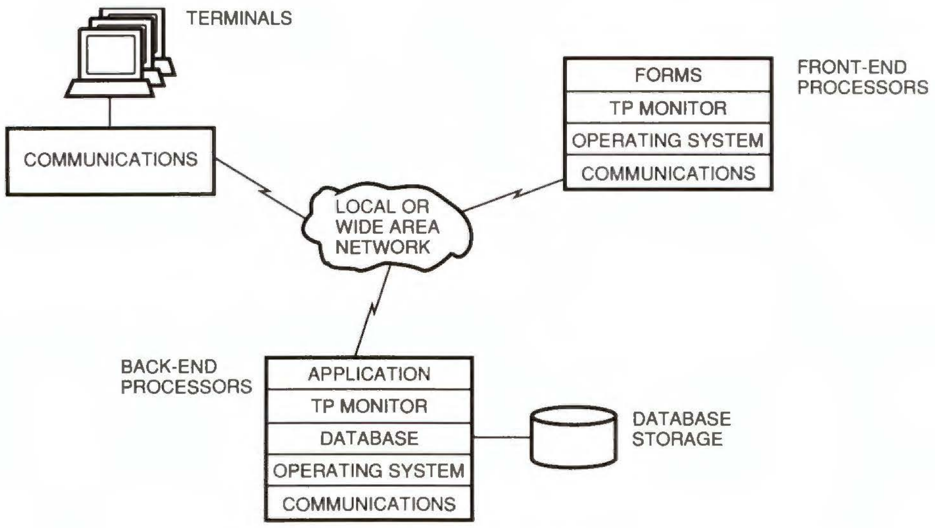


Figure 3 Distributed Transaction Processing Environment

software components shown in Figure 3. The user interface component runs on one or more front-end processors, whereas the application and database components run on one or more back-end processors. Transactions are entered from teller terminals, which communicate with the front-end processors. The front-end processors then communicate with the back-end processors to invoke the application servers and perform database operations. The communications can take place over either a local area or a wide area network. However, to simplify testing, the TPC Benchmark A standard allows sponsors to use remote terminal emulators (RTEs) rather than real terminals. Therefore, the TPC Benchmark A tests base performance and price/performance results on two distinctly configured systems, the target system and the test system.

The target system is the configuration of hardware and software components that customers can use to perform transaction processing. With the Digital distributed transaction processing approach, user terminals initiate transactions and communicate with the front-end processors. Front-end processors communicate with a back-end processor using the DECnet protocol.

The test system is the configuration of components used in the lab to measure the performance of the target system. The test system uses RTEs, rather than user terminals, to generate the workload and measure response time. (Note: In previously published reports, based on Digital's DebitCredit benchmark, the RTE emulated front-end processors. In the TPC Benchmark A standard, the RTE emulates only the user terminals.) The RTE component

- Emulates the behavior of terminal users according to the benchmark specification (e.g., think time, transaction parameters)
- Emulates terminal devices (e.g., conversion and multiplexing into the local area transport [LAT] protocol used by the DECserver terminal servers)
- Records transaction messages and response times (e.g., the starting and ending times of individual transactions from each emulated terminal device)

Figure 4 depicts the test system configuration in the LAN environment with one back-end processor, multiple front-end processors, and multiple remote terminal emulators.

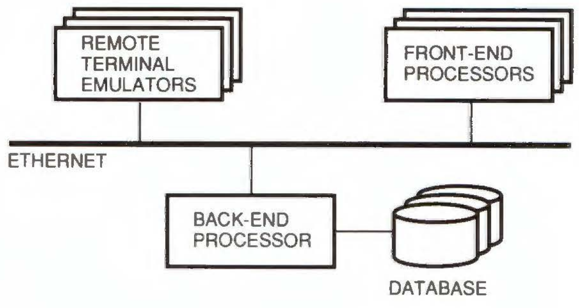


Figure 4 Test System Configuration

TPC Benchmark A Results

We now present the results of two TPC Benchmark A tests based on audited benchmark experiments performed on the VAX 9000 Model 210 and the VAX 4000 Model 300 systems.[3,4] These two systems are representative of Digital's large and small transaction processing platforms. The benchmark was implemented using the VAX ACMS transaction processing monitor, the VAX Rdb/VMS relational database management system, and the DECforms forms management system on the VMS operating system. Tables 4 and 5 show the back-end system configurations for the VAX 9000 Model 210 and the VAX 4000 Model 300 systems, respectively. Table 6 shows the system configuration of the front-end systems.

Measurement Results

The maximum qualified throughput and response time results for the TPC Benchmark A are summarized in Table 7 for the VAX 9000 Model 210 and the VAX 4000 Model 300 systems. Both configurations have sufficient main memory and disk drives such

Table 4 VAX 9000 Model 210 Back-end System Configuration

Component	Product	Quantity
Processor	VAX 9000 Model 210	1
Memory		256 MB
Tape drive	TA81	1
Disk controller	KDM70	2
Disks	RA92	16
Operating system	VMS 5.4	1
Communications	DECnet-VMS Phase IV	1
TP monitor	VAX ACMS V3.1	1
Dictionary	VAX CDD/Plus V4.1	1
Application	VAX COBOL V4.2	1
Database system	VAX Rdb/VMS V4.0	1
Forms management	DECforms V1.2	1

Table 5 VAX 4000 Model 300 Back-end System Configuration

Component	Product	Quantity
Processor	VAX 4000 Model 300	1
Memory		64 MB
Tape drive	TK70	1
Disk controller	DSSI	3
Disks	RF31	18
Operating system	VMS 5.4	1
Communications	DECnet-VMS Phase IV	1
TP monitor	VAX ACMS V3.1	1
Dictionary	VAX CDD/Plus V4.1	1
Application	VAX COBOL V4.2	1
Database system	VAX Rdb/VMS V4.0	1
Forms management	DECforms V1.2	1

that the processors are effectively utilized with no other bottleneck. Both systems achieved well over 90 percent CPU utilization at the maximum qualified throughput under the response time constraint. In addition to the throughput and response time, the TPC Benchmark A specification requires that several other data points and graphs be reported. We demonstrate these data and graphs by using the VAX 9000 Model 210 TPC Benchmark A results.

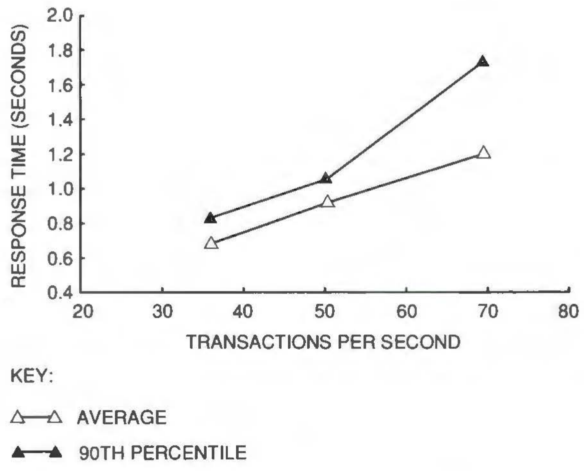


Figure 5 VAX 9000 Response Time in Relationship to Transactions per Second

- Response Time in Relationship to TPS. Figure 5 shows the ninetieth percentile and average response times at 100 percent and approximately 80 percent and 50 percent of the maximum qualified throughput. The mean transaction response time still grows linearly with the transaction rate up to the 70 TPS level, but the ninetieth percentile response time curve has started to rise quickly due to the high CPU utilization and random arrival of transactions.
- Response Time Frequency Distribution. Figure 6 is a graphical representation of the transaction

Table 6 Front-end Run-time System Configuration

Component	Product	Quantity
Processor	VAXserver 3100 Model 10	10 for VAX 9000 back-end 3 for VAX 4000 back-end
Memory		16 MB for VAX 9000 back-end 12 MB for VAX 4000 back-end
Disks	RZ23 (104 MB)	16
Operating system	VMS 5.3 VMS 5.4	1 for VAX 9000 back-end 1 for VAX 4000 back-end
Communications	DECnet-VMS Phase IV	1
TP monitor	VAX ACMS V3.1	1
Forms management	DECforms V1.2	1

Table 7 Maximum Qualified Throughput

		Response Time (seconds)		
System	TPS (tpsA-Local)	Average	90 percent	Maximum
VAX 9000 Model 210	69.4	1.20	1.74	5.82
VAX 4000 Model 300	21.6	1.39	1.99	4.81

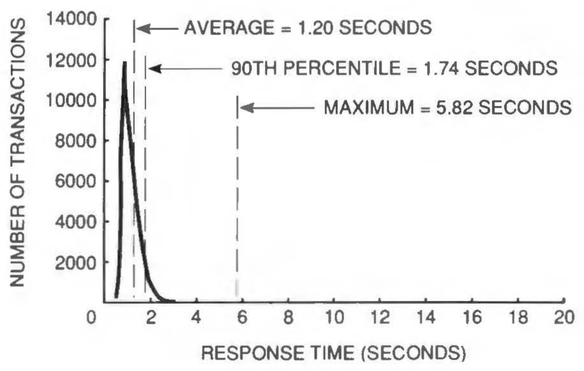


Figure 6 VAX 9000 Response Time Frequency Distribution

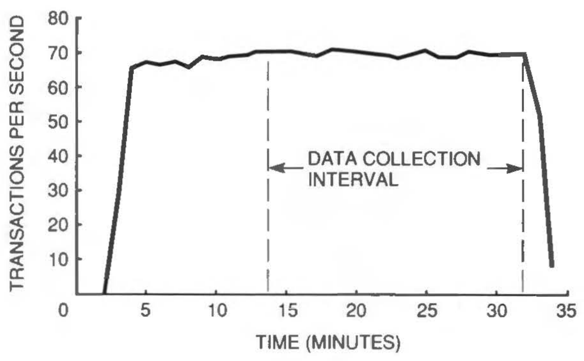


Figure 7 VAX 9000 Transactions per Second over Time

response time distribution. The average, ninetieth percentile, and maximum transaction response times are also marked on the graph.

- Transactions per Second over Time. The results shown in Figure 7 demonstrate the sustainable maximum qualified throughput. The one-minute running average transaction throughputs during the warm-up and data collection periods of the experiment are plotted on the graph. This graph shows that the throughput was steady during the period of data collection.

- Average Response Time over Time. The results shown in Figure 8 demonstrate the sustainable average response time in the experiment. The one-minute running average transaction response times during the warm-up and data collection periods of the experiment are plotted on the graph. This graph shows that the mean response time was steady during the period of data collection.

Comprehensive Analytical Model

Modeling techniques can be used as a supplement or an alternative to the measurement approach. The performance behavior of complex transaction processing systems can be characterized by a set of parameters, a set of performance metrics, and the relationships among them. These parameters can be used to describe the different resources available in the system, the database operations of transactions, and the workload that the transaction processing system undergoes. To completely represent such a system, the size of the parameter set would be too huge to manage. An analytical model simplifies, through abstraction, the complex behavior of a system into a manageable set of parameters and policies. Such a model, after proper validation, can be a powerful tool for many types of analysis, as well as a performance prediction tool. Results can be obtained quickly for any combination of parameters.

A comprehensive analytical model of the performance behavior of transaction processing systems with a response time constraint was developed and validated against measurement results. This model is hierarchical and flexible for extension. The following sections describe the basic construction of the model and the customization made to model the execution of TPC Benchmark A on Digital's transaction processing systems. The model can also be used to study different transaction processing workloads in addition to the TPC Benchmark A.

Response Time Components

The main metric used in the model is the maximum qualified throughput under a response time constraint. The response time constraint is in the

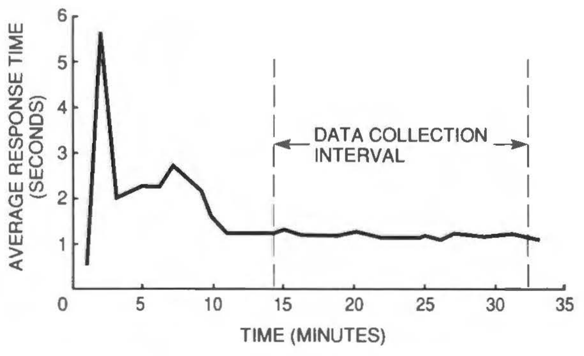


Figure 8 VAX 9000 Average Response Time over Time

form of "*x* percent of transaction response times are less than *y* seconds."

To evaluate throughput under such response time constraint, the distribution of transaction response times is determined by first decomposing the transaction response time into nonoverlapping and independent components. The distribution of each component is then evaluated. Finally, the overall transaction response time distribution is derived from the mathematical convolution of the component response time distributions.

The logical flow of a transaction in a front-end and back-end distributed transaction processing system that is used to implement TPC Benchmark A is depicted in Figure 9. The response time of a transaction consists of three basic components: front-end processing, back-end processing, and communication delays.

- Front-end processing usually includes terminal I/O processing, forms/presentation services, and communication with the back-end systems. In the benchmark experiments, no disk I/O activity was involved during the front-end processing.
- Back-end processing includes the execution of application, database access, concurrency control, and transaction commit processing. The back-end processing usually involves a high degree of concurrency and many disk I/O activities.
- Communication delays primarily include the communications between the user terminal and the front-end node, and the front-end and back-end interactions.

(Note: These response time components do not overlap with each other.)

Within the back-end system, the transaction response time is further decomposed into two additional components, CPU delays and non-CPU, nonoverlapping delays. CPU delays include both the CPU service and the CPU waiting times of transactions. Non-CPU, nonoverlapping delays include:

- Logging delays, which include the time for transaction log writes and commit protocol delays
- Database I/O delays, which include both waiting and service times for accessing storage devices
- Other delays, which include delays that result from concurrency control (e.g., waiting for locks) and waiting for messages

Two-level Approach

The model is configured in a two-level hierarchy, a high level and a detailed level. The use of a hierarchy allows a complex and detailed model that considers many components and involves many parameters to be constructed easily. Because of the hierarchical approach, the model also provides flexibility for modifications and extensions, and validation of separate submodels.

The high-level model assumes the decomposition of transaction response times, as described in the Response Time Components section, and models the behavior of the transaction processing system by an open queuing system, as shown in Figure 10. The queuing system consists of servers and delay centers, which are connected in a queuing network with the following assumptions:

- The front-end processing does not involve any disk I/O operation, and the load on the front-end systems is equally balanced.

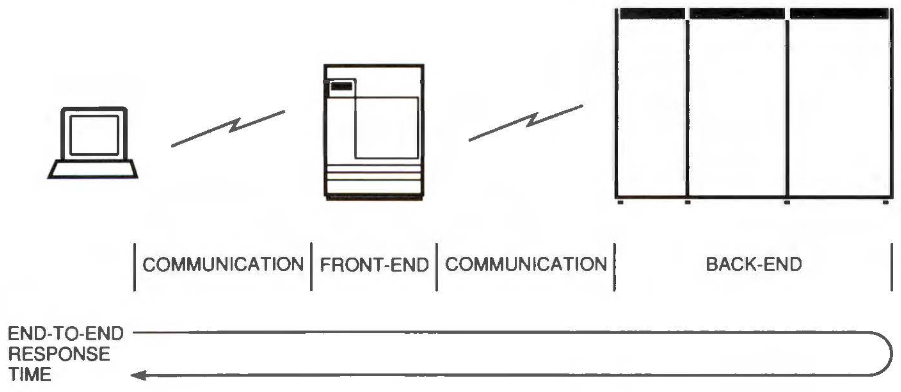


Figure 9 Response Time Components

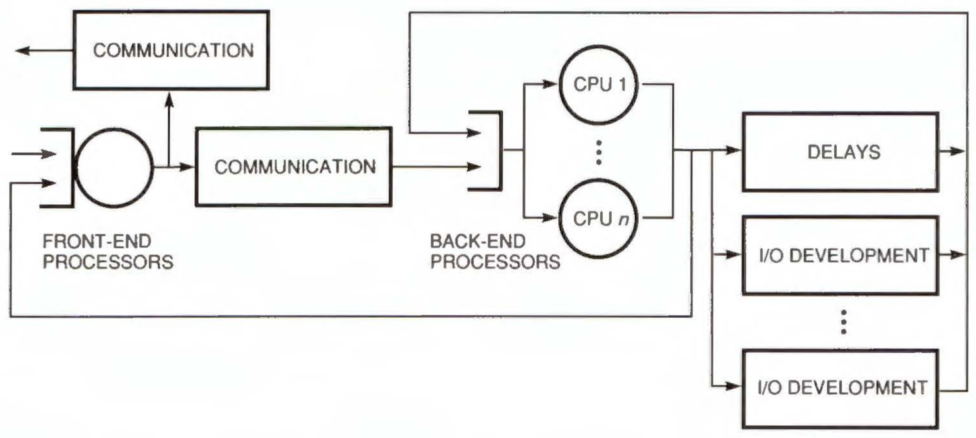


Figure 10 High-level Queuing Model for a Transaction Processing System

- The back-end is a shared-memory multiprocessor system with symmetrical loads on all processors (or it can be simply a uniprocessor).
- No intratransaction parallelism exists within individual transaction execution.
- No mutual dependency exists between transaction response time components.
- Transaction arrivals to the processors have a Poisson distribution.

These assumptions correspond to Digital's TPC Benchmark A testing methodology and implementation.

The front-end CPU is modeled as an M/M/1 queuing center, and the back-end CPU is modeled as an M/M/m queuing center. The transactions' CPU times on the front-end and back-end systems are assumed to be exponentially distributed (coefficient of variation equal to 1) due to the single type of transaction in the benchmark. (Note: An approximation of M/G/m can be used to consider a coefficient of variation other than 1 for the back-end transaction CPU service time, especially in the multiprocessor case when the bus is highly utilized.) Database I/O, logging I/O, and other delays are modeled as delay centers, with appropriate delay distributions. For the model of the TPC Benchmark A workload, the database I/O, journaling I/O, and other communication and synchronization delays are combined into one delay center, called the LOD delay center, which is represented by a 2-Erlang distribution. The major input parameters for this high-level model are the

- Number of front-end systems and the front-end CPU service time per transaction
- Number of CPUs in the back-end system and the back-end CPU service time per transaction
- Sum of the back-end database I/O response time, journaling I/O response time, and other delay times (i.e., the mean for the LOD delay center's 2-Erlang distribution)
- Response time constraint (in the form of x percentile less than y seconds)

The main result from the high-level model is the MQTh. This high-level model presents a global picture of the performance behavior and manifests the relationship between the most important parameters of the transaction processing system and MQTh.

Some of the input parameters in the high-level model are dynamic. The CPU service time of a transaction may vary with the throughput or number of processors, and the database I/O or other delays may also depend on the throughput. A good example of a dynamic model is a tightly coupled multiprocessor system, with one bus interconnecting the processors and with a shared common memory (e.g., a VAX 6000 Model 440 system). Such a system would run a single copy of the symmetrical multiprocessing operating system (e.g., the VMS system). The average CPU service time of transactions is affected by both hardware and software factors, such as

- Hardware contention that results from conflicting accesses to the shared bus and main memory and that causes processor speed degradation and longer CPU service time.
- Processor synchronization overhead that results from the serialization of accesses to shared data

structures. Many operating systems use spin-locks as the mechanism for processor-level synchronization, and the processor spins (i.e., busy-waits) in the case of a conflict. In the model, the busy-wait overhead is considered to be part of the transaction code path, and such contention elongates the transaction CPU service time.

Four detailed-level submodels are used to account for the dynamic behavior of these parameters: CPU-cache-bus-memory, busy-wait, I/O group, and LOD.

The CPU-cache-bus-memory submodel consists of many low-level parameters associated with the workload, processor, cache, bus, and memory components of multiprocessor systems. It models these components by using a mixed queuing network model that consists of both open and closed chains, as shown in Figure 11. The most important output from this submodel is the average number of CPU clock cycles per instruction.

The busy-wait submodel models the spin-lock contention that is associated with the two major VMS spin-locks, called SCHED and IOLOCK8. This submodel divides the state of a processor into several nonoverlapping states and uses probability analysis to derive busy-wait time. The I/O grouping submodel models the group commit and group write mechanisms of the VAX Rdb/VMS relational database management system. This submodel affects the path length of transaction because of the amortization of disk I/O processing among grouped transactions. The LOD submodel considers the disk I/O times and the lock contention of certain critical resources in the VAX Rdb/VMS system.

Integrating the Two Levels of the Model

The two levels of the model are integrated by using an iterative procedure outlined in Figure 12. It starts at the detailed-level submodels, with initial values for the MQTh, the transaction path length, the busy-wait overhead, and the CPU utilization.

By applying the initialized parameters to the submodels, the values of these parameters are refined and input to the high-level model. The output parameters from the high-level model are then fed back to the detailed-level submodels, and this iterative process continues until the MQTh converges. In most cases, convergence is reached within a few iterations.

Model Predictions

The back-end portion of the model was validated against measurement results from numerous DebitCredit benchmarks (Digital's precursor of the TPC Benchmark A) on many VAX computers with the VMS operating system, running VAX ACMS and VAX Rdb/VMS software.[5] With sufficient detailed parameters available (such as transaction instruction count, instruction cycle time, bus/memory access time, cache hit ratio), the model correctly estimated the MQTh and many intermediate results for several multiprocessor VAX systems. The model was then extended to include the front-end systems. In this section, we discuss applying this complete end-to-end model to the TPC Benchmark A on two VAX platforms, the VAX 9000 Model 210 and the VAX 4000 Model 300 systems, and then compare the results. The benchmark environment and implementation are described in the TPC Benchmark A Implementation section of this paper.

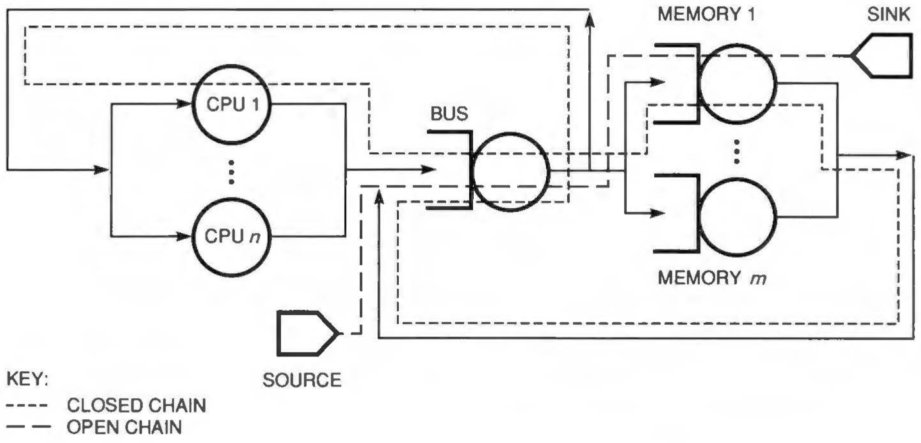


Figure 11 CPU-cache-bus-memory Submodel

```
INITIALIZE:
    TxnPL,MQTh,BusyWaitPL,CpuUtilization;
LOD-submodel(input:MQTh;output:LOD)
REPEAT
    I/O-Grouping-submodel(input:MQTh;output:DioPerTxn,TxnPL);
    REPEAT
        REPEAT
            BusyWait-submodel(input:TxnPL,BusyWaitPL,CpuUtilization,
                DioPerTxn;output:BusyWaitPL);
        UNTIL(BusyWaitPL converges);
        CPU-Cache-Bus-Memory-submodel(input:TxnPL,BusyWaitPL;
            output:CpuUtilization,AvgCpuSvcTime);
    UNTIL(CpuUtilization converges);
    REPEAT
        MQTh-model(input:AvgCpuSvcTime,LOD;output:MQTh,CpuUtilization);
        LOD-submodel(input:MQTh;output:LOD);
    UNTIL(MQTh converges);
UNTIL(MQTh converges);
```

Figure 12 The Iterative Procedure to Integrating Submodels

Because both the VAX 9000 Model 210 and the VAX 4000 Model 300 systems are uniprocessor systems, there is no other processor contending for the processor-memory interconnect and memory subsystems. Such contention effects can therefore be ignored when modeling a uniprocessor system. The transaction processing performance prediction for the VAX 9000 Model 210 system is a successful example of the application of our analytical model.

We needed an accurate estimate of TPC Benchmark A performance on the VAX 9000 Model 210 system before a VAX 9000 system was actually available for testing. The high-level (MQTh) model was used with estimated values for the input parameters, LOD and transaction CPU service time. The estimated LOD was based on previous measurement observations from the VAX 6000 systems. The other parameter, back-end transaction CPU service time, was derived from the

- Timing information of the VAX 9000 CPU
- Memory access time and cache miss penalty of the VAX 9000 CPU
- Prediction of cache hit ratio of the VAX 9000 system under the TPC Benchmark A workload
- Transaction path length of the TPC Benchmark A implementation
- Instruction profile of the TPC Benchmark A implementation

The high-level model predicted a range of MQTh, with a high end of 70 TPS and with a strong probability that the high-end performance was achievable.

Additional predictions were made later, when an early prototype version of the VAX 9000 Model 210 system was available for testing. A variant of the DebitCredit benchmark, much smaller in scale and easier to run, was performed on the prototype system, with the emphasis on measuring the CPU performance in a transaction processing environment. The result was used to extrapolate the CPU service time of the TPC Benchmark A transactions on the VAX 9000 Model 210 system and to refine the early estimate. The results of these modifications supported the previous high-end estimate of performance of 70 TPS and refined the low-end performance to be 62 TPS. The final, audited TPC Benchmark A measurement result of the VAX 9000 Model 210 system showed 69.4 TPS, which closely matches the prediction. Table 8 compares the results from benchmark measurement and the analytical model outputs.

Table 8 Measurement Compared to Model Predictions

System	Measured MQTh	Modeled MQTh
VAX 9000 Model 210	69.4	70.0
VAX 4000 Model 300	21.5	20.8

The VAX 4000 Model 300 TPC Benchmark A results were also used as a validation case. VAX 4000 Model 300 systems use the same CMOS chip as the VAX 6000 Model 400 series and the same 28-nanosecond (ns) CPU cycle time. However, in the VAX 4000 series, the CPU-memory interconnect is not the XMI bus but a direct primary memory interconnect. This direct memory interconnect results in fast main memory access. The processor, cache, and main memory subsystems are otherwise the same as in the VAX 6000 Model 400 systems. Therefore, the detailed-level model and associated parameters for the VAX 6000 Model 410 system can be used by ignoring the bus access time. The TPC Benchmark A measurement results are within 7 percent of the model prediction, which means that our assumption on the memory access time is acceptable.

Conclusion

Performance is one of the most important attributes in evaluating a transaction processing system. However, because of the complex nature of transaction processing systems, a universal assessment of transaction processing system performance is impossible. The performance of a transaction processing system is workload dependent, configuration dependent, and implementation dependent. A standard benchmark, like TPC Benchmark A, is a step toward a fair comparison of transaction processing performance by different vendors. But it is only one transaction processing benchmark that represents a limited class of applications. When evaluating transaction processing systems performance, a good understanding of the targeted application environment and requirements is essential before using any available benchmark result. Additional benchmarks that represent a broader range of commercial applications are expected to be standardized by the Transaction Processing Performance Council (TPC) in the coming years.

Performance modeling is an attractive alternative to benchmark measurement because it is less expensive to perform and results can be compiled more quickly. Modeling provides more insight into the behavior of system components that are treated as black boxes in most measurement experiments. Modeling helps system designers to better understand performance issues and to discover existing or potential performance problems. Modeling also provides solutions for improving performance by modeling different tuning or design alternatives. The analytical model presented in this paper was validated and used extensively in many engineering performance studies. The model also helped the benchmark process to size the hardware during preparation (e.g., the number of RTE and front-end systems needed, the size of the database) and to provide an MQTh goal as a sanity check and a tuning aid. The model could be extended to represent additional distributed configurations, such as shared-disk and "shared-nothing" back-end transaction processing systems, and could be applied to additional transaction processing workloads.

Acknowledgments

The Digital TPC Benchmark A implementation and measurements are the result of work by many individuals within Digital. The authors would like especially to thank Jim McKenzie, Martha Ryan, Hwan Shen, and Bob Tanski for their work in the TPC Benchmark A measurement experiments; and Per Gyllstrom and Rabah Mediouni for their contributions to the analytical model and validation.

References

1. *Transaction Processing Performance Council, TPC Benchmark A Standard Specification* (Menlo Park, CA: Waterside Associates, November 1989).

2. *Transaction Processing Systems Handbook* (Maynard: Digital Equipment Corporation, Order No. EC-H0650-57, 1990).

3. *TPC Benchmark: A Report for the VAX 9000 Model 210 System* (Maynard: Digital Equipment Corporation, Order No. EC-N0302-57, 1990).

4. *TPC Benchmark: A Report for the VAX 4000 Model 300 System* (Maynard: Digital Equipment Corporation, Order No. EC-N0301-57, 1990).

5. L. Wright, W. Kohler, and W. Zahavi, "The Digital DebitCredit Benchmark: Methodology and Results," *Proceedings of the International Conference on Management and Performance Evaluation of Computer Systems* (December 1989): 84–92.

William Z. Zahavi
Frances A. Habib
Kenneth J. Omahen

Tools and Techniques for Preliminary Sizing of Transaction Processing Applications

Sizing transaction processing systems correctly is a difficult task. By nature, transaction processing applications are not predefined and can vary from the simple to the complex. Sizing during the analysis and design stages of the application development cycle is particularly difficult. It is impossible to measure the resource requirements of an application which is not yet written or fully implemented. To make sizing easier and more accurate in these stages, a sizing methodology was developed that uses measurements from systems on which industry-standard benchmarks have been run and employs standard systems analysis techniques for acquiring sizing information. These metrics are then used to predict future transaction resource usage.

The transaction processing marketplace is dominated by commercial applications that support businesses. These applications contribute substantially to the success or failure of a business, based on the level of performance the application provides. In transaction processing, poor application performance can translate directly into lost revenues.

The risk of implementing a transaction processing application that performs poorly can be minimized by estimating the proper system size in the early stages of application development. Sizing estimation includes configuring the correct processor and proper number of disk drives and controllers, given the characteristics of the application.

The sizing of transaction processing systems is a difficult activity. Unlike traditional applications such as mail, transaction processing applications are not predefined. Each customer's requirement is different and can vary from simple to complex. Therefore, Digital chose to develop a sizing methodology that specifically meets the unique requirements of transaction processing customers. The goal of this effort was to develop sizing tools and techniques that would help marketing groups and design consultants in recommending configurations that meet the needs of Digital's customers. Digital's methodology evolved over time, as experience was gained in dealing with the real-world problems of transaction processing system sizing.

The development of Digital's transaction processing sizing methodology was guided by several principles. The first principle is that the methodology should rely heavily upon measurements of Digital systems running industry-standard transaction processing benchmarks. These benchmarks provide valuable data that quantifies the performance characteristics of different hardware and software configurations.

The second principle is that systems analysis methodologies should be used to provide a framework for acquiring sizing information. In particular, a multilevel view of a customer's business is adopted. This approach recognizes that a manager's view of the business functions performed by an organization is different from a computer analyst's view of the transaction processing activity. The sizing methodology should accommodate both these views.

The third principle is that the sizing methodology must employ tools and techniques appropriate to the current stage of the customer's application design cycle. Early in the effort to develop a sizing methodology, it was found that a distinction must be made between preliminary sizing and sizing during later stages of the application development cycle. Preliminary sizing occurs during the analysis and design stages of the application development cycle. Therefore, no application software exists which can be measured. Application software does exist in later stages of the application development cycle, and its measurement provides valuable input for more precise sizing activities.

For example, if a customer is in the analysis or design stages of the application development cycle, it is unlikely that estimates can be obtained for such quantities as paging rates or memory usage. However, if the application is fully implemented, then tools such as the VAXcluster Performance Advisor (VPA) and the DECcp capacity planning products can be used for sizing. These tools provide facilities for measuring and analyzing data from a running system and for using the data as input to queuing models.

The term sizing, as used in this paper, refers to preliminary sizing. The paper presents the metrics and algebra used in the sizing process for DECtp applications. It also describes the individual tools developed as part of Digital's transaction processing sizing effort.

Sizing

The purpose of sizing tools is twofold. First, sizing tools are used to select the appropriate system components and to estimate the performance level of the system in terms of device utilization and user response times. Second, sizing tools bridge the gap between business specialists and computer specialists. This bridge translates the business units into functions that are performed on the system and, ultimately, into units of work that can be quantified and measured in terms of system resources.

In the sections that follow, a number of important elements of the sizing methodology are described. The first of these elements is the platform on which the transaction processing system will be implemented. It is assumed that the customer will supply general preferences for the software and hardware configuration as part of the platform information. The Levels of Business Metrics section details the multilevel approach used to describe the work performed by the business. The Sizing Metrics and Sizing Formulas sections describe the algorithms that use platform and business metric information to perform transaction processing system sizing.

Platforms

The term platform is used in transaction processing sizing methodology to encompass general customer preferences for the hardware and software upon which the transaction processing application will run.

The hardware platform specifies the desired topology or processing style. For example, processing style includes a centralized configuration and a front-end and back-end configuration as valid alternatives. The hardware platform may also include specific hardware components within the processing style. (In this paper, the term processor refers to the overall processing unit, which may be composed of multiple CPUs.)

The software platform identifies the set of layered products to be used by the transaction processing application, with each software product identified by its name and version number. In the transaction processing environment, a software platform is composed of the transaction processing monitor, forms manager, database management system, application language, and operating system.

Different combinations of software platforms may be configured, depending on the hardware platform used. A centralized configuration contains all the software components on the same system. A distributed system is comprised of a front-end processor and a back-end processor; different software platforms may exist on each processor.

Levels of Business Metrics

The term business metrics refers collectively to the various ways to measure the work associated with a customer's business. In this section, various levels of business metrics are identified and the relationship between metrics at different levels is described.[1] As mentioned earlier, the levels correspond to the multilevel view of business operation typically used for systems analysis. The organization or personnel most interested in a metric in relation to its business operation is noted in the discussion of each metric.

The decomposition of the business application requirements into components that can be counted and quantified in terms of resource usage requires that a set of metrics be defined. These metrics reflect the business activity and the system load. The business metrics are the foundation for the development of several transaction processing sizing tools and for a consistent algebra that connects the business units with the computer units.

The business metrics are natural forecasting units, business functions, transactions, and the number of I/Os per transaction. The relationship among these levels is shown in Figure 1. In general, a business may have one or more natural forecasting units. Each natural forecasting unit may drive one or more business functions. A business function may

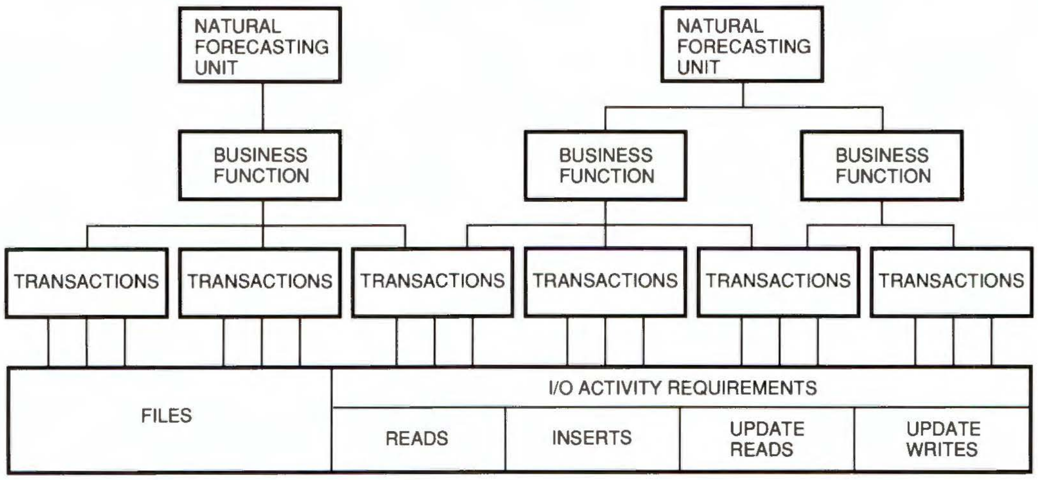


Figure 1 Levels of Business Activity Characterization

have multiple transactions, and a single transaction may be activated by different business functions. Every transaction issues a variety of I/O operations to one or more files, which may be physically located on zero, one, or more disks. This section discusses the business metrics but does not discuss the physical distribution of I/Os across disks, which is an implementation-specific item.

A natural forecasting unit is a macrolevel indicator of business volume. (It is also called a key volume indicator.) A business generally uses a volume indicator to measure the level of success of the business. The volume is often measured in time intervals that reflect the business cycle, such as weekly, monthly, or quarterly. For example, if business volume indicators were "number of ticket sales per quarter," or "monthly production of widgets," then the corresponding natural forecasting units would be "ticket sales" and "widgets." Natural forecasting units are used by high-level executives to track the health of the overall business.

Business functions are a logical unit of work performed on behalf of a natural forecasting unit. For example, within an airline reservation system, a common business function might be "selling airline tickets." This business function may consist of multiple interactions with the computer (e.g., flight inquiry, customer credit check). The completion of the sale terminates the business function, and "airline ticket" acts as a natural forecasting unit for the enterprise selling the tickets. The measurement metric for business functions is the number of business function occurrences per hour. Business functions may be used by middle-level managers to track the activity of their departments.

A transaction is an atomic unit of work for an application, and transaction response time is the primary performance measure seen by a user. Each of the interactions mentioned in the above business function is a transaction. The measurement metric for a transaction is the number of transaction occurrences per business function. Transactions may be used by low-level managers to track the activity of their groups.

The bulk of commercial applications involves the maintaining and moving of information. This information is data that is often stored on permanent storage devices such as rotational disks, solid state disks, or tapes. An I/O operation is the process by which a transaction accesses that data. The measurement metric for the I/O profile is the number of I/O operations per transaction. I/O operations by each transaction are important to programmers or system analysts.

In addition to issuing I/Os, each transaction requires a certain amount of CPU time to handle forms processing. (Forms processing time is not illustrated in Figure 1.) The measurement metric for forms processing time is the expected number of fields. The number of input and output fields per form are important metrics for users of a transaction processing application or programmer/system analysts.

By collecting information about a transaction processing application at various levels, high-level volume indicators are mapped to low-level units of I/O activity. This mapping is fundamental to the transaction processing sizing methodology.

Performance goals play a particularly important role in the sizing of transaction processing systems.[2] The major categories of performance goals commonly encountered in the transaction processing marketplace are bounds for

- Device utilization(s)
- Average response time for transactions
- Response time quantiles for transactions

For example, a customer might specify a required processor utilization of less than 70 percent. Such a constraint reflects the fact that system response time typically rises dramatically at higher processor utilizations. A common performance goal for response time is to use a transaction's average response time and response time quantiles. For example, the proposed system should have an average response time of x seconds, with 95 percent of all responses completing in less than or equal to y seconds, where x is less than y. Transaction response times are crucial for businesses. Poor response times translate directly into decreased productivity and lost revenues.

When a customer generates a formal Request For Proposal (RFP), the performance goals for the transaction processing system typically are specified in detail. The specification of goals makes it easier to define the performance bounds. For customers who supply only general performance goals, it is assumed that the performance goal takes the form of bounds for device utilizations.

Overall response time consists of incremental contributions by each major component of the overall system:

- Front-end processor
- Back-end processor
- Communications network
- Disk subsystem

A main objective in this approach to sizing was to identify and use specific metrics that could be easily counted for each major component. For instance, the number of fields per form could be a metric used for sizing front-end processors because that number is specific and easily counted. As the path of a transaction is followed through the overall system, the units of work appropriate for each component become clear. These units become the metrics for sizing that particular component. The focus of this paper is on processor sizing with bounds on processor utilization. Processors generally constitute the major expense in any proposed system solution. Mistakes in processor sizing are very expensive to fix, both in terms of customer satisfaction and cost.

Sizing Metrics

Transaction processing applications permit a large number of users to share access to a common database crucial to the business and usually residing on disk memory. In an interactive transaction processing environment, transactions generally involve some number of disk I/O operations, although the number is relatively small compared to those generated by batch transaction processing applications. CPU processing also is generally small and consists primarily of overhead for layered transaction processing software products. Although these numbers are small, they did influence the sizing methodology in several ways.

Ratings for relative processor capacity in a transaction processing environment were developed to reflect the ability of a processor to support disk I/O activity (as observed in benchmark tests). In addition, empirical studies of transaction processing applications showed that, for purposes of preliminary sizing, the number of disk I/Os generated by a transaction provides a good prediction of the required amount of CPU processing.[3] Numerous industry-standard benchmark tests for product positioning were run on Digital's processors. These processors were configured as back-end processors in a distributed configuration with different software platforms.

The base workload for this benchmark testing is currently the Transaction Processing Performance Council's TPC Benchmark A (TPC-A, formerly the DebitCredit benchmark).[4,5,6] The most complete set of benchmark testing was run under Digital's VAX ACMS transaction processing monitor and VAX Rdb/VMS relational database. Therefore, results from this software platform on all Digital processors were used to compute the first sizing metric called the base load factor.

The base load factor is a high-level metric that incorporates the contribution by all layered software products on the back-end processor to the total CPU time per I/O operation. Load factors are computed by dividing the total CPU utilization by the number of achieved disk I/O operations per second. (The CPU utilization is normalized in the event that the processor is a Symmetrical Multiprocessing [SMP] system, to ensure that its value falls within the range of 0 to 100 percent.) The calculation of load factor yields the total CPU time, in centiseconds (hundredths of seconds), required to support an application's single physical I/O operation.

The base load factors give the CPU time per I/O required to run the base workload, TPC-A, on any Digital processor in a back-end configuration using the ACMS/Rdb. The CPU time per I/O can be estimated for any workload. This generalized metric is called the application load factor.

To relate the base load factors to workloads other than the base, an additional metric was defined called the intensity factor. The metric calculation for the intensity factor is the application load factor divided by the base load factor. The value in using intensity factors is that, once estimated (or calculated for running applications), intensity factors can be used to characterize any application in a way that can be applied across all processor types to estimate processor requirements.

Intensity factors vary based on the software platform used. If a software platform other than a combined VAX ACMS and VAX Rdb/VMS platform is selected, the estimate of the intensity factor must be adjusted to reflect the resource usage characteristics of the selected DECtp software platform.

To estimate an appropriate intensity factor for a nonexistent application, judgment and experience with similar applications are required. However, measured cases from a range of DECtp applications shows relatively little variation in intensity factors. Guidelines to help determine intensity factors are included in the documentation for Digital's internally developed transaction processing sizing tools.

The work required by any transaction processing application is composed of two parts: the application/database and the forms management. This division of work corresponds to what occurs in a distributed configuration, where the forms processing is off-loaded to one or more front-end processors. Load factors and intensity factors are metrics that were developed to size the application/database. To estimate the amount of CPU time required for forms management, a forms-specific metric is required. For a first-cut approximation, the expected number of (input) fields is used as the sizing metric. This number is obtained easily from the business-level description of the application.

Sizing Formulas

This section describes the underlying algebra developed for processor selection. Different formulas to estimate the CPU time required for both the application/database and forms management were developed. These formulas are used separately for sizing back-end and front-end processors in a distributed configuration. The individual contributions of the formulas are combined for sizing a centralized configuration.

The application/database is the work that takes place on the back-end processor of a distributed configuration. It is a function of physical disk accesses. To determine the minimal CPU time required to handle this load, processor utilization is used as the performance goal, setting up an inequality that is solved to obtain a corresponding load factor. The resulting load factor is then compared to the table of base load factors to obtain a recommendation for a processor type. To reinforce this dependence of load factors on processor types, load factor x refers to the associated processor type x in the following calculations.

One method for estimating the average CPU time per transaction is to multiply the number of I/Os per transaction by the load factor x and the intensity factor. This yields CPU time per transaction, expressed in centiseconds per transaction. By multiplying this product by the transactions per second rate, an expression for processor utilization is derived. Thus processor utilization (expressed as a percentage scaled between 0 and 100 percent) is the number of transactions per second, times the number of I/Os per transaction, times load factor x, times the intensity factor.

The performance goal is a CPU utilization that is less than the utilization specified by the customer. Therefore, the calculation used to derive the load factor is the utilization percentage provided by the customer, divided by the number of transactions per second, times the number of I/Os per transaction, times the intensity factor.

Once computed, the load factor is compared to those values in the base load factor table. The base load factor equal to or less than the computed value is selected, and its corresponding processor type, x, is returned as the minimal processor required to handle this workload.

The four input parameters that need to be estimated for inclusion in this inequality are

- Processor utilization performance goal (traditionally set at around 70 percent, but may be set higher for Digital's newer, faster processors)
- Target transactions per second (which may be derived from Digital's multilevel mapping of business metrics)

- I/Os per transaction (estimated from application description and database expertise)
- Intensity factor (estimated from experience with similar applications)

Note: Response time performance goals do not appear in this formula. This sizing formula deals strictly with ensuring adequate processor capacity. However, these performance parameters (including the CPU service time per transaction) are fed into an analytic queuing solver embedded in some of the transaction processing sizing tools, which produces estimates of response times.

Forms processing is the work that occurs either on the front-end processor of a distributed configuration or in a centralized configuration. It is not a function of physical disk accesses; rather, forms processing is CPU intensive. To estimate the CPU time (in seconds) required for forms processing, the following simple linear equation is used:

$$y = c(a + bz)$$

where y equals the CPU time for forms processing; a equals the CPU time per form per transaction instance, depending on the forms manager used; b equals the CPU time per field per transaction instance, depending on the forms manager used; z equals the expected number of fields; and c equals the scaling ratio, depending on the processor type. This equation was developed by feeding the results of controlled forms testing into a linear regression model to estimate the CPU cost per form and per field (i.e., a and b). The multiplicative term, c, is used to eliminate the dependence of factors a and b on the hardware platform used to run these tests.

Sizing Tools

Several sizing tools were constructed by using the above formulas as starting points. These tools differ in the range of required inputs and outputs, and in the expected technical sophistication of the user.

The first tool developed is for quick, first-approximation processor sizing. Currently embodied as a DECalc spreadsheet, with one screen for processor selection and one for transactions-per-second sensitivity analysis, it can handle back-end, front-end, or centralized sizing. The first screen shows the range of processors required, given the target processor utilization, target transactions per second, expected number of fields, and the possible intensity factors and number of I/Os per transaction. (Because the estimation of these last two inputs generally involves the most uncertainty, the spreadsheet allows the user to input a range of values for each.) The second screen turns the analysis around, showing the resulting transaction-per-second ranges that can be supported by the processor type selected by the user, given the target processor utilization, expected number of fields, and possible intensity factors and number of I/Os per transaction.

The basic sizing formula addresses issues that deal specifically with capacity but not with performance. To predict behavior such as response times and queue lengths, modeling techniques that employ analytic solvers or simulators are needed. A second tool embeds an analytic queuing solver within itself to produce performance estimates. This tool is an automated system (i.e., a DECtp application) that requests information from the user according to the multilevel workload characterization methodology. This starts from general business-level information and proceeds to request successively more detailed information about the application. The tool also contains a knowledge base of Digital's product characteristics (e.g., processor and disk) and measured DECtp applications. The user can search through the measured cases to find a similar case, which could then be used to provide a starting point for estimating key application parameters. The built-in product characteristics shield the user from the numeric details of the sizing algorithms.

A third tool is a spin-off from the second tool. This tool is a standalone analytic queuing solver with a simple textual interface. The tool is intended for the sophisticated user and assumes that the user has completed the level of analysis required to be able to supply the necessary technical input parameters. No automatic table lookups are provided. However, for a completely characterized application, this tool gives the sophisticated user a quick means to obtain performance estimates and run sensitivity analyses. The complete DECtp software platform necessary to run the second tool is not required for this tool.

Data Collection

To use the sizing tools fully, certain data must be available, which allows measured workloads to be used to establish the basic metrics. Guidance in sizing unmeasured transaction processing applications is highly dependent on developing a knowledge base of real-world transaction processing application descriptions and measurements. The kinds of data that need to be stored within the knowledge base require the data collection tools to gather information consistent with the transaction processing sizing algebra.

For each transaction type and for the aggregate of all the transaction types, the following information is necessary to perform transaction processing system sizing:

- CPU time per disk I/O
- Disk I/O operations per transaction
- Transaction rates
- Logical-to-physical disk I/O ratio

The CPU to I/O ratio can be derived from Digital's existing instrumentation products, such as the VAX Software Performance Monitor (SPM) and VAXcluster Performance Advisor (VPA) products.[7] Both products can record and store data that reflects CPU usage levels and physical disk I/O rates.

The DECtrace product collects event-driven data. It can collect resource items from layered software products, including VAX ACMS monitor, the VAX Rdb/VMS and DBMS database systems, and if instrumented, from the application program itself. As an event collector, the DECtrace product can be used to track the rate at which events occur.

The methods for determining the logical-to-physical disk I/O ratio per transaction remain open for continuing study. Physical disk I/O operations are issued based on logical commands from the application. The find, update, or fetch commands from an SQL program translate into from zero to many thousands of physical disk I/O operations, depending upon where and how data is stored. Characteristics that affect this ratio include the length of the data tables, number of index keys, and access methods used to reach individual data items (i.e., sequential, random).

Few tools currently available can provide data on physical I/O operations for workloads in the design stage. A knowledge base that stores the logical-to-physical disk I/O activity ratio is the best method available at this time for predicting that value. The knowledge base in the second sizing tool is beginning to be populated with application descriptions that include this type of information. It is anticipated that, as this tool becomes widely used in the field, many more application descriptions will be stored in the knowledge base. Pooling individual application experiences into one central repository will create a valuable source of knowledge that may be utilized to provide better information for future sizing exercises.

Acknowledgments

The authors would like to acknowledge our colleagues in the Transaction Processing Systems Performance Group whose efforts led to the development of these sizing tools, either through product characterization, system support, objective critique, or actual tool development. In particular, we would like to acknowledge the contributions made by Jim Bouhana to the development of the sizing methodology and tools.

References

1. W. Zahavi and J. Bouhana, "Business-Level Description of Transaction Processing Applications," *CMG '88 Proceedings* (1988): 720-726.

2. K. Omahen, "Practical Strategies for Configuring Balanced Transaction Processing Systems," *IEEE COMPCON Spring '89 Proceedings* (1989): 554-559.

3. W. Zahavi, "A First Approximation Sizing Technique —The I/O Operation as a Metric of CPU Power," *CMG '90 Conference Proceedings* (forthcoming December 10-14, 1990).

4. "TPC BENCHMARK A — Standard Specification," (Transaction Processing Performance Council, November 1989).

5. "A Measure of Transaction Processing Power," *Datamation,* vol. 31, no. 7 (April 1, 1985): 112-118.

6. L. Wright, W. Kohler, and W. Zahavi, "The Digital DebitCredit Benchmark: Methodology and Results," *CMG '89 Conference Proceedings* (1989): 84-92.

7. F. Habib, Y. Hsu, and K. Omahen, "Software Measurement Tools for VAX/VMS Systems," *CMG Transactions* (Summer 1988): 47-78.

Ananth Raghavan
T. K. Rengarajan

Database Availability for Transaction Processing

A transaction processing system relies on its database management system to supply high availability. Digital offers a network-based product, the VAX DBMS system, and a relational data-based product, the VAX Rdb/VMS database system, for its transaction processing systems. These database systems have several strategies to survive failures, disk head crashes, revectored bad blocks, database corruptions, memory corruptions, and memory overwrites by faulty application programs. They use base hardware technologies and also employ novel software techniques, such as parallel transaction recovery, recovery on surviving nodes of a VAXcluster system, restore and roll-forward operations on areas of the database, on-line backup, verification and repair utilities, and executive mode protection of trusted database management system code.

Modern businesses store critical data in database management systems. Much of the daily activity of business includes manipulation of data in the database. As businesses extend their operations worldwide, their databases are shared among office locations in different parts of the world. Consequently, these businesses require transaction processing systems to be available for use at all times. This requirement translates directly to a goal of perfect availability for database management systems.

VAX DBMS and VAX Rdb/VMS database systems are based on network and relational data models, respectively. Both systems use a kernel of code that is largely responsible for providing high availability. This layer of code is maintained by the KODA group. KODA is the physical subsystem for VAX DBMS and VAX Rdb/VMS database systems. It is responsible for all I/O, buffer management, concurrency control, transaction consistency, locking, journaling, and access methods.

In this paper, we define database availability, and describe downtime situations and how such situations can be resolved. We then discuss the mechanisms that have been implemented to provide minimal loss of availability.

Database Availability

The unit of work in transaction processing systems is a transaction. We therefore define database availability as the ability to execute transactions. One way the database management system provides high availability is by guaranteeing the properties of transactions: atomicity, serializability, and durability.[1] For example, if a transaction that has made updates to the database is aborted, other transactions must not be allowed to see these updates; the updates made by the aborted transaction must be removed from the database before other transactions may use that data. Yet, data that has not been accessed by the aborted transaction must continue to be available to other transactions.

Downtime is the term used to refer to periods when the database is unavailable. Downtime is caused by either an unexpected failure (unexpected downtime) or scheduled maintenance on the database (scheduled downtime). Such classifications of downtime are useful. Unexpected downtime is caused by factors that are beyond the control of the transaction processing system. For example, a disk failure is quite possible at any time during normal processing of transactions. However, scheduled downtime is entirely within the control of the database administrator. High availability demands that we eliminate scheduled downtime and ensure fast system recovery from unexpected failures.

The layers of the software and hardware services which compose a transaction processing system are dependent on one another for high availability. The dependency among these services is illustrated in Figure 1. Each service depends on the

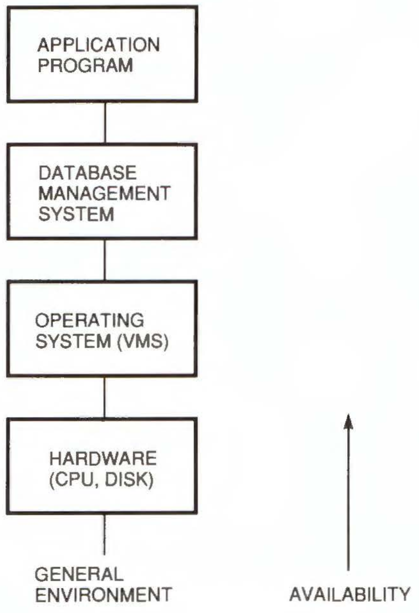


Figure 1 Layers of Availability in Transaction Processing Systems

availability of the service in the lower layers. Errors and failures can occur in any layer, but may not be detected immediately. For example, in the case of a database management system, the effects of a database corruption may not be apparent until long after the corruption (error) has occurred. Hence it is difficult to deal with such errors. On the other hand, failures are noticed immediately. Failures usually make the system unavailable and are the cause of unexpected downtime.

Each layer can provide only as much availability as the immediate lower layer. Hence we can also express the perfect-availability goal of a database management system as the goal of matching the availability of the immediately lower layer, which in our case is the operating system.

At the outset, it is clear that a database management system layered on top of an operating system and hence only as available as the underlying operating system. However, a database management system is in general not as available as the underlying layer because of the need to guarantee the properties of transactions.

Unexpected Downtime

In this section we discuss the causes of unexpected downtime and the techniques that minimize downtime.

A database monitor must be started on a node before a user's process running on that node can access a database. The monitor oversees all database activity on the node. It allows processes to attach to and detach from databases and detects failures. On detecting a failure, the monitor starts a process to recover the transactions that did not complete because of the failure. Note that this database monitor is different from the TP monitor.[2]

Application Program Exceptions

Although transaction processing systems are based on the client/server architecture, Digital's database systems are process based. The privileged database management system code is packaged in a shareable library and linked with the application programs. Therefore, bugs in the applications have a good chance of affecting the consistency of the database. Such bugs in applications are one type of failure that can make the database unavailable.

The VAX DBMS and VAX Rdb/VMS systems guard against this class of failure by executing the database management system code in the VAX executive mode. Since application programs execute in user mode, they do not have access to data structures used by the database management system. When a faulty application program attempts such an access, the VMS operating system detects it and generates an exception. This exception then forces an image rundown of the application program.

In general, when an image rundown is initiated, Digital's database management products use the condition-handling facility of VMS to abort the transaction. Condition handling of image rundown is performed at two levels. Two condition handlers are established, one in user mode and the other in kernel mode. The user mode exit handler is usually invoked, which rolls back the current transaction and unbinds it from the database. In this case, the rest of the users on the system are not affected at all. The database remains available. The execution of the user mode exit handler is, however, not guaranteed by the VMS operating system. Under some abnormal circumstances, the user mode exit handlers may not be executed at all. In such circumstances, the kernel mode exit handler is invoked by the VMS system. This handler resides in the database monitor. The monitor starts a database recovery (DBR) process. It is the responsibility of the DBR process to roll back the effects of the aborted transaction. To do this, the DBR process first establishes a database freeze. This freeze prevents other processes from acquiring locks that were held by the aborted transaction and hence see and update uncommitted data. (The VMS lock manager releases all locks held by a process when that process dies.) The DBR process then proceeds to roll back the aborted transaction.

Code Corruptions

It is important to prevent coding mistakes within the DBMS from irretrievably corrupting the database. To protect the database management system from coding mistakes, internal data structure consistency is examined at different points in the code. If any inconsistency is found, a bug-check utility is called that dumps the internal database format to a file. The utility then raises an exception that is handled by the monitor, and the DBR process is started as described above.

To deal with corruptions to the database that are undetected with this mechanism, an explicit utility is provided that verifies the structural consistency of the database. This verify utility may be executed on-line, while users are still accessing the database. Such verification may also be executed by a database administrator (DBA) in response to a bug-check dump. Once such a corruption is detected, an on-line utility provides the ability to repair the database.

In general, corruption in databases causes unexpected downtime. Digital provides the means of detecting such corruption on-line and repairing them on-line through recovery utilities.

Process Failure

In the VMS system, a process failure is always preceded by image rundown of the current image running as part of the process. Therefore, a process failure is detected by the database monitor, which then starts a DBR process to handle recovery.

Node Failure

Among the many mechanisms Digital provides for availability is node failover within a cluster. When a node fails, another node on the cluster detects the failure and rolls back the lost transactions from the failed node. Thus the failure of one node does not cause transactions on other active nodes of the cluster to come to a halt (except for the time the DBR process enforces a freeze). It is the database monitor that detects node failure and starts a recovery process for every lost transaction on the failed node. The database becomes available as soon as recovery is complete for all the users on the failed node.

Power Failure

Power failure is a hardware failure. As soon as power is restored, the VMS system boots. When a process attaches to the database, a number of messages are passed between the process that is attaching and the monitor. If the database is corrupt (because of power failure), the monitor is so informed by the attaching process, and again the monitor starts recovery processes to return the database to a consistent state. The database becomes available as soon as recovery is complete for all such failed users.

As described above, recovery is always accomplished by the monitor process starting DBR processes to do the recovery. The only differences in the case of process, node, or cluster failure is the mechanism by which the monitor is informed of the failure.

Disk Head Crash

Some failures can result in the loss or corruption of the data on the stable storage device (disk). Digital has a mechanism for bringing the database back to a consistent state in such cases.

A disk head crash is a failure of hardware that is usually characterized by the inability to read from or write to the disk. Hence database storage areas residing on that disk are unavailable and possibly irretrievable. A disk head crash automatically aborts transactions that need to read from or write to that disk. In addition, recovery of these aborted transactions is not possible since the recovery processes need access to the same disk. In this case, the database is shut down and access is denied until the storage areas on the failed disk are brought on-line. Areas are restored from backups and then rolled forward until consistent with the rest of the database. The after image journal (AIJ) files are used to roll the areas forward. As soon as all the areas on the failed disk have been restored onto a good disk and rolled forward, the database becomes available.

Bad Disk Blocks

Bad blocks are hardware errors that often are not detected when they happen. The bad blocks are revectored, and the next time the disk block is read, an error is reported. Bad blocks simply mean that the contents of a disk block are lost forever. The database administrator detects the problem only when a database application fails to fetch data on the revectored block. Such an error may cause a certain transaction or a set of transactions to fail, no matter how many attempts are made to execute the transactions. This failure constitutes reduced availability; parts of the database are unavailable to transactions. Exactly how much of the database remains available depends on which blocks were revectored.

The mechanism provided to reduce the possible downtime is early detection. Digital's database systems provide a verification utility that can be executed while users are running transactions. The verification utility checks the structural consistency of the database. Once a bad block is detected by such a utility, that area of the database may be restored and rolled forward. These two operations make the whole database temporarily unavailable; however, the bad block is corrected, and future downtime is avoided. The downtime caused by the bad block may be traded off against the downtime needed to restore and roll forward.

Site Failure

A site failure occurs when neither the computers nor the disks are available. A site failure is usually caused by a natural disaster such as an earthquake. The best recourse for recovery is archival storage. Digital provides mechanisms to back up the database and AIJ files to tape. These tapes must then be stored at a site away from the site at which the database resides. Should a disaster happen, these backup tapes can be used to restore the database. However, the recovery may not be complete. It cannot restore the effects of those committed transactions that were not backed up to tape.

After a disaster, the database can be restored and rolled forward to the state of the completion of the last AIJ that was backed up to tape. Any transactions that committed after the last AIJ was backed up cannot be recovered at the alternate site. Such transaction losses can be minimized by frequently backing up the AIJ files.

Memory Errors

Memory errors are quite infrequent, and when they happen, they usually are not detected. If the error happens to a data record, it may never be detected by any utility, but may be seen as incorrect data by the user. If the verification utility is run on-line, it may also detect the errors. Again, the database may only be partially available, as in the case of bad blocks. However, it is possible to repair the database while users are still accessing the database. Digital's database management products provide explicit repair facilities for this purpose. The loss of availability during repair is not worse than the loss due to the memory error itself.

As explained previously, the database monitor plays an important part in ensuring database consistency and availability. Most unexpected failure scenarios are detected by the monitor, which then starts recovery processes. In addition, some failures might require the use of backup files to restore the database.

Scheduled Downtime

Most database systems have scheduled maintenance operations that require a database shutdown. Database backup for media recovery and verification to check structural consistency are examples of operations that may require scheduled downtime. In this section we describe ways to perform many of these operations while the database is executing transactions.

Backup

Digital's database systems allow two types of transactions: update and "snapshot." The ability to back up data on-line depends on the snapshot transaction capability of the database.

Database backup is a standard way of recovering from media failures. Digital's database systems provide the ability to do transaction consistent backups of data on-line while users continue to change the database.

The general scheme for snapshot transactions is as follows. The update transactions of the database preserve the previous versions of the database records in the snapshot file. All versions of a database record are chained. Only the current version of the record is in the database area. The older versions are kept in the snapshot area. The versions of the records are tagged with the transaction numbers (TSNs). When a snapshot transaction (for example, a database backup) needs to read a database record, it traverses the chain for that database record and then uses the appropriate version of the record.

There are two modes of database operation with respect to snapshot activity. In one mode, all update transactions write snapshot copies of any records they update. In the deferred snapshot mode, the updates cause snapshot copies to be written only if a snapshot transaction is active and requires old versions of a record. In this mode, a snapshot transaction cannot start until all currently active update transactions (which are not writing snapshot records) have completed; that is, the snapshot transaction must wait for a quiet point in time. If there are either active or pending snapshot transactions when an update transaction starts, the update transaction must write snapshot copies.

Here we see a trade-off between update transactions and snapshot transactions. The database is completely available to snapshot transactions if all update transactions always write snapshot copies. On the other hand, if the deferred snapshot mode is enabled, update transactions need not write snapshot copies if a snapshot transaction in not active. This approach obviously results in some loss of availability to snapshot transactions.

Verification

Database corruption can also result in downtime. Although database corruption is not probable, it is possible. Any database system that supports critical data must provide facilities to ensure the consistency of the database. Digital's database management systems provide verification utilities that scan the database to check the structural consistency of the database. These utilities may also be executed on-line through the use of snapshot transactions.

AIJ Backup

The backup and the AIJ log are the two mechanisms that provide media recovery for Digital's database management products. The AIJ file is continuously written to by all user processes updating the database. We need to provide some ability to back up the AIJ file since it monotonically increases in size and eventually fills up the disk. Digital's database systems offer the ability to back up the AIJ file to tape (or another device) on-line. The only restriction is that a quiet point must be established for a short period during which the backup operation takes place. A quiet point is defined as a point when the database is quiescent, i.e., there are no active transactions.

On-line Schema Changes

Digital's database management systems allow users to change metadata on-line, while users are still accessing the database. Although this may be standard for relational database management systems, it is not standard for network databases. The VAX DBMS system provides a utility called the database restructuring utility (DRU) to allow for on-line schema modifications.

Acknowledgments

Many engineers have contributed to the development of the algorithms described in this paper. We have chosen not to enumerate all such contributions. However, we would like to recognize the contributions of Peter Spiro, Ashok Joshi, Jeff Arnold, and Rick Anderson who, together with the authors, are members of the KODA team.

References

1. P. Bernstein, W. Emberton, and V. Trehan, "DECdta — Digital's Distributed Transaction Processing Architecture," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 10-17.

2. T. Speer and M. Storm, "Digital's Transaction Processing Monitors," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991, this issue): 18-32.

Peter M. Spiro
Ashok M. Joshi
T. K. Rengarajan

Designing an Optimized Transaction Commit Protocol

Digital's database products, VAX Rdb/VMS and VAX DBMS, share the same database kernel called KODA. KODA uses a grouping mechanism to commit many concurrent transactions together. This feature enables high transaction rates in a transaction processing (TP) environment. Since group commit processing affects the maximum throughput of the transaction processing system, the KODA group designed and implemented several grouping algorithms and studied their performance characteristics. Preliminary results indicate that it is possible to achieve up to a 66 percent improvement in transaction throughput by using more efficient grouping designs.

Digital has two general-purpose database products, Rdb/VMS software, which supports the relational data model, and VAX DBMS software, which supports the CODASYL (Conference on Data Systems Languages) data model. Both products layer on top of a database kernel called KODA. In addition to other database services, KODA provides the transaction capabilities and commit processing for these two products.

In this paper, we address some of the issues relevant to efficient commit processing. We begin by explaining the importance of commit processing in achieving high transaction throughput. Next, we describe in detail the current algorithm for group commit used in KODA. We then describe and contrast several new designs for performing a group commit. Following these discussions, we present our experimental results. And, finally, we discuss the possible direction of future work and some conclusions. No attempt is made to present formal analysis or exhaustive empirical results for commit processing; rather, the focus is on an intuitive understanding of the concepts and trade-offs, along with some empirical results that support our conclusions.

Commit Processing

To follow a discussion of commit processing, two basic terms must first be understood. We begin this section by defining a transaction and the "moment of commit."

A transaction is the execution of one or more statements that access data managed by a database system. Generally, database management systems guarantee that the effects of a transaction are atomic, that is, either all updates performed within the context of the transaction are recorded in the database, or no updates are reflected in the database.

The point at which a transaction's effects become durable is known as the "moment of commit." This concept is important because it allows database recovery to proceed in a predictable manner after a transaction failure. If a transaction terminates abnormally before it reaches the moment of commit, then it aborts. As a result, the database system performs transaction recovery, which removes all effects of the transaction. However, if the transaction has passed the moment of commit, recovery processing ensures that all changes made by the transaction are permanent.

Transaction Profile

For the purpose of analysis, it is useful to divide a transaction processed by KODA into four phases: the transaction start phase, the data manipulation phase, the logging phase, and the commit processing phase. Figure 1 illustrates the phases of a transaction in time sequence. The first three phases are collectively referred to as "the average transaction's CPU cost (excluding the cost of commit)" and the last phase (commit) as "the cost of writing a group commit buffer."[1]

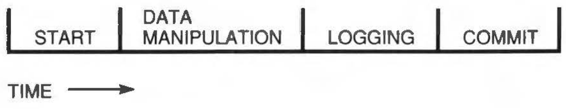


Figure 1 Phases in the Execution of a Transaction

The transaction start phase involves acquiring a transaction identifier and setting up control data structures. This phase usually incurs a fixed overhead.

The data manipulation phase involves executing the actions dictated by an application program. Obviously, the time spent in this phase and the amount of processing required depend on the nature of the application.

At some point a request is made to complete the transaction. Accordingly in KODA, the transaction enters the logging phase which involves updating the database with the changes and writing the undo/redo information to disk. The amount of work done in the logging phase is usually small and constant (less than one I/O) for transaction processing.

Finally, the transaction enters the commit processing phase. In KODA, this phase involves writing commit information to disk, thereby ensuring that the transaction's effects are recorded in the database and now visible to other users.

For some transactions, the data manipulation phase is very expensive, possibly requiring a large number of I/Os and a great deal of CPU time. For example, if 500 employees in a company were to get a 10 percent salary increase, a transaction would have to fetch and modify every employee/salary record in the company database. The commit processing phase, in this example, represents 0.2 percent of the transaction duration. Thus, for this class of transaction, commit processing is a small fraction of the overall cost. Figure 2 illustrates the profile of a transaction modifying 500 records.

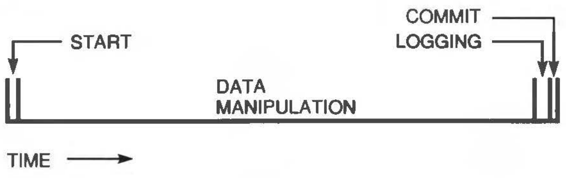


Figure 2 Profile of a Transaction Modifying 500 Records

In contrast, for transaction processing applications such as hotel reservation systems, banking applications, stock market transactions, or the telephone system, the data manipulation phase is usually short (requiring few I/Os). Instead, the logging and commit phases comprise the bulk of the work and must be optimized to allow high transaction throughput. The transaction profile for a transaction modifying one record is shown in Figure 3. Note that the commit processing phase represents 36 percent of the transaction duration, in this example.

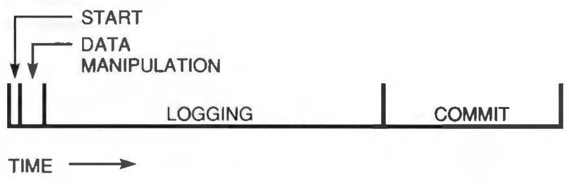


Figure 3 Profile of a Transaction Modifying One Record

Group Commit

Generally, database systems must force write information to disk in order to commit transactions. In the event of a failure, this operation permits recovery processing to determine which failed transactions were active at the time of their termination and which ones had reached their moment of commit. This information is often in the form of lists of transaction identifiers, called commit lists.

Many database systems perform an optimized version of commit processing where commit information for a group of transactions is written to disk in one I/O operation, thereby, amortizing the cost of the I/O across multiple transactions. So, rather than having each transaction write its own commit list to disk, one transaction writes to disk a commit list containing the commit information for a number of other transactions. This technique is referred to in the literature as "group commit."[2]

Group commit processing is essential for achieving high throughput. If every transaction that reached the commit stage had to actually perform an I/O to the same disk to flush its own commit information, the throughput of the database system would be limited to the I/O rate of the disk. A magnetic disk is capable of performing 30 I/O operations per second. Consequently, in the absence of group commit, the throughput of the system is limited to 30 transactions per second (TPS). Group commit is essential to breaking this performance barrier.

There are several variations of the basic algorithms for grouping multiple commit lists into a single I/O. The specific group commit algorithm chosen can significantly influence the throughput and response times of transaction processing. One study reports throughput gains of as much as 25 percent by selecting an optimal group commit algorithm.[1]

At high transaction throughput (hundreds of transactions per second), efficient commit processing provides a significant performance advantage. There is little information in the database literature about the efficiency of different methods of performing a group commit. Therefore, we analyzed several grouping designs and evaluated their performance benefits.

Factors Affecting Group Commit

Before proceeding to a description of the experiments, it is useful to have a better understanding of the factors affecting the behavior of the group commit mechanism. This section discusses the group size, the use of timers to stall transactions, and the relationship between these two factors.

Group Size An important factor affecting group commit is the number of transactions that participate in the group commit. There must be several transactions in the group in order to benefit from I/O amortization. At the same time, transactions should not be required to wait too long for the group to build up to a large size, as this factor would adversely affect throughput.

It is interesting to note that the incremental advantage of adding one more transaction to a group decreases as the group size increases. The incremental savings is equal to $1/(G \times (G + 1))$, where G is the initial group size. For example, if the group consists of 2 transactions, each of them does one-half a write. If the group size increases to 3, the incremental savings in writes will be (1/2 - 1/3), or 1/6 per transaction. If we do the same calculation for a group size incremented from 10 to 11, the savings will be (1/10 - 1/11), or 1/110 of a write per transaction.

In general, if G represents the group size, and I represents the number of I/Os per second for the disk, the maximum transaction commit rate is $I \times G$ TPS. For example, if the group size is 45 and the rate is 30 I/Os per second to disk, the maximum transaction commit rate is 30×45, or 1350 TPS. Note that a grouping of only 10 will restrict the maximum TPS to 300 TPS, regardless of how powerful the computer is. Therefore, the group size directly affects the maximum transaction throughput of the transaction processing system.

Use of Timers to Stall Transactions One of the mechanisms to increase the size of the commit group is the use of timers.[1,2] Timers are used to stall the transactions for a short period of time (on the order of tens of milliseconds) during commit processing. During the stall, more transactions enter the commit processing phase and so the group size becomes larger. The stalls provided by the timers have the advantage of increasing the group size, and the disadvantage of increasing the response time.

Trade-offs This section discusses the trade-offs between the size of the group and the use of timers to stall transactions. Consider a system where there are 50 active database programs, each repeatedly processing transactions against a database. Assume that on average each transaction takes between 0.4 and 0.5 seconds. Thus, at peak performance, the database system can commit approximately 100 transactions every second, each program actually completing two transactions in the one-second time interval. Also, assume that the transactions arrive at the commit point in a steady stream at different times.

If transaction commit is stalled for 0.2 seconds to allow the commit group to build up, the group then consists of about 20 transactions (0.2 seconds $\times$ 100 TPS). In this case, each transaction only incurs a small delay at commit time, averaging 0.10 seconds, and the database system should be able to approach its peak throughput of 100 TPS. However, if the mechanism delays commit processing for one second, an entirely different behavior sequence occurs. Since the transactions complete in approximately 0.5 seconds, they accumulate at the commit stall and are forced to wait until the one-second stall completes. The group size then consists of 50 transactions, thereby maximizing the I/O amortization. However, throughput is also limited to 50 TPS, since a group commit is occurring only once per second.

Thus, it is necessary to balance response time and the size of the commit group. The longer the stall, the larger the group size; the larger the group size, the better the I/O amortization that is achieved. However, if the stall time is too long, it is possible to limit transaction throughput because of wasted CPU cycles.

Motivation for Our Work

The concept of using commit timers is discussed in great detail by Reuter.[1] However, there are significant differences between his group commit scheme and our scheme. These differences prompted the work we present in this paper.

In Reuter's scheme, the timer expiration triggers the group commit for everyone. In our scheme, no single process is in charge of commit processing based on a timer. Our commit processing is performed by one of the processes desiring to write a commit record. Our designs involve coordination between the processes in order to elect the group committer (a process).

Reuter's analysis to determine the optimum value of the timer based on system load assumes that the total transaction duration, the time taken for commit processing, and the time taken for performing the other phases are the same for all transactions. In contrast, we do not make that assumption. Our designs strive to adapt to the execution of many different transaction types under different system loads. Because of the complexity introduced by allowing variations in transaction classes, we do not attempt to calculate the optimal timer values as does Reuter.

Cooperative Commit Processing

In this section, we present the stages in performing the group commit with cooperating processes, and we describe, in detail, the grouping design currently used in KODA, the Commit-Lock Design.

Group Committer

Assume that a number of transactions have completed all data manipulation and logging activity and are ready to execute the commit processing phase. To group the commit requests, the following steps must be performed in KODA:

1. Each transaction must make its commit information available to the group committer.
2. One of the processes must be selected as the "group committer."
3. The other members of the group need to be informed that their commit work will be completed by the group committer. These processes must wait until the commit information is written to disk by the group committer.
4. Once the group committer has written the commit information to stable storage, it must inform the other members that commit processing is completed.

Commit-Lock Design

The Commit-Lock Design uses a VMS lock to generate groups of committing transactions; the lock is also used to choose the group committer.

Once a process completes all its updates and wants to commit its transaction, the procedure is as follows. Each transaction must first declare its intent to join a group commit. In KODA, each process uses the interlocked queue instructions of the VAX system running VMS software to enqueue a block of commit information, known as a commit packet, onto a globally accessible commit queue. The commit queue and the commit packets are located in a shared, writeable global section.

Each process then issues a lock request for the commit lock. At this point, a number of other processes are assumed to be going through the same sequence; that is, they are posting their commit packets and making lock requests for the commit lock. One of these processes is granted the commit lock. For the time being, assume the process that currently acquires the lock acts as the group committer.

The group committer, first, counts the number of entries on the commit queue, providing the number of transactions that will be part of the group commit. Because of the VAX interlocked queue instructions, scanning to obtain a count and concurrent queue operations by other processes can proceed simultaneously. The group committer uses the information in each commit packet to format the commit block which will be written to disk. In KODA, the commit block is used as a commit list, recording which transactions have committed and which ones are active. In order to commit for a transaction, the group committer must mark each current transaction as completed. In addition, as an optimization, the group committer assigns a new transaction identifier for each process's next transaction. Figure 4 illustrates a commit block ready to be flushed to disk.

Once the commit block is modified, the group committer writes it to disk in one atomic I/O. This is the moment of commit for all transactions in the group. Thus, all transactions that were active and took part in this group commit are now stably marked as committed. In addition, as explained above, these transactions now have new transaction identifiers. Next, the group committer sets a commit flag in each commit packet for all recently committed transactions, removes all commit packets from the commit queue, and, finally, releases the commit lock. Figure 5 illustrates a committed

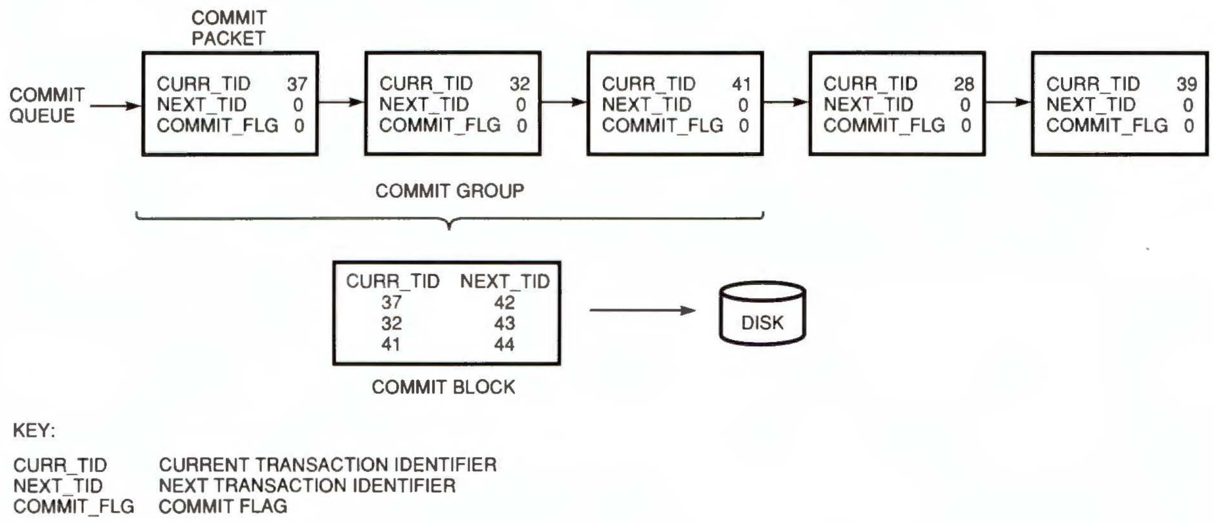


Figure 4 Commit Block Ready to be Flushed to Disk

group with new transaction identifiers and with commit flags set.

At this point, the remaining processes that were part of the group commit are, in turn, granted the commit lock. Because their commit flags are already set, these processes realize they do not need to perform a commit and, thus, release the commit lock and proceed to the next transaction. After all these committed processes release the commit lock, a process that did not take part in the group commit acquires the lock, notices it has not been committed, and, therefore, initiates the next group commit.

There are several interesting points about using the VMS lock as the grouping mechanism. Even though all the transactions are effectively committed after the commit block I/O has completed, the transactions are still forced to proceed serially; that is, each process is granted the lock, notices that it is committed, and then releases the lock.

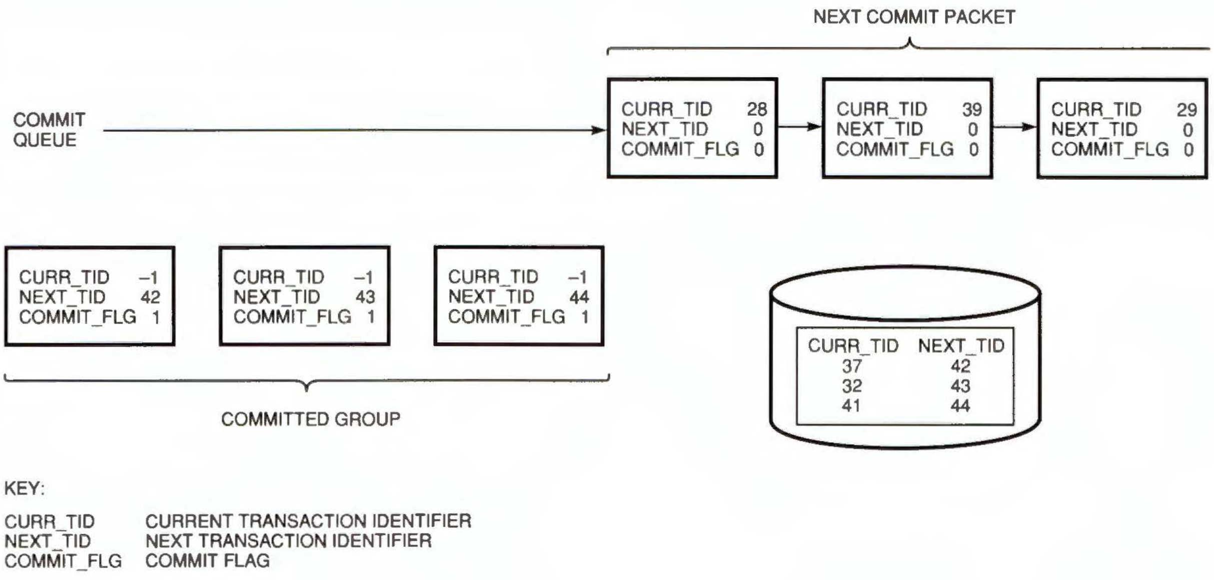


Figure 5 Committed Group

So there is a serial procession of lock enqueues/dequeues before the next group can start.

This serial procession can be made more concurrent by, first, requesting the lock in a shared mode, hoping that all processes committed are granted the lock in unison. However, in practice, some processes that are granted the lock are not committed. These processes must then request the lock in an exclusive mode. If this lock request is mastered on a different node in a VAXcluster system, the lock enqueue/dequeues are very expensive.

Also, there is no explicit stall time built into the algorithm. The latency associated with the lock enqueue/dequeue requests allows the commit queue to build up. This stall is entirely dependent on the contention for the lock, which in turn depends on the throughput.

Group Commit Mechanisms — Our New Designs

To improve on the transaction throughput provided by the Commit-Lock Design, we developed three different grouping designs, and we compared their performances at high throughput. Note that the basic paradigm of group commit for all these designs is described in the Group Committer section. Our designs are as follows.

Commit-Stall Design

In the Commit-Stall Design, the use of the commit lock as the grouping mechanism is eliminated. Instead, a process inserts its commit packet onto the commit queue and, then, checks to see if it is the first process on the queue. If so, the process acts as the group committer. If not, the process schedules its own wake-up call, then sleeps. Upon waking, the process checks to see if it has been committed. If so, the process proceeds to its next transaction. If not, the process again checks to see if it is first on the commit queue. The algorithm then repeats, as described above.

This method attempts to eliminate the serial wake-up behavior displayed by using the commit lock. Also, the duration for which each process stalls can be varied per transaction to allow explicit control of the group size. Note that if the stall time is too small, a process may wake up and stall many times before it is committed.

Willing-to-Wait Design

As we have seen before, a delay in the commit sequence is a convenient means of converting a response time advantage into a throughput gain. If we increase the stall time, the transaction duration increases, which is undesirable. At the same time, the grouping size for group commit increases, which is desirable. The challenge is to determine the optimal stall time. Reuter presented an analytical way of determining the optimal stall time for a system with transactions of the same type.[1]

Ideally, we would like to devise a flexible scheme that makes the trade-off we have just described in real time and determines the optimum commit stall time dynamically. However, we cannot determine the optimum stall time automatically, because the database management system cannot judge which is more important to the user in a general customer situation — the transaction response time or the throughput.

The Willing-to-Wait Design provides a user parameter called WTW time. This parameter represents the amount of time the user is willing to wait for the transaction to complete, given this wait will benefit the complete system by increasing throughput. WTW time may be specified by the user for each transaction. Given such a user specification, it is easy to calculate the commit stall to increase the group size. This stall equals the WTW time minus the time taken by the transaction thus far, but only if the transaction has not already exceeded the WTW time. For example, if a transaction comes to commit processing in 0.5 second and the WTW time is 2.0 seconds, the stall time is then 1.5 seconds. In addition, we can make a further improvement by reducing the stall time by the amount of time needed for group commit processing. This delta time is constant, on the order of 50 milliseconds (one I/O plus some computation).

The WTW parameter gives the user control over how much of the response time advantage (if any) may be used by the system to improve transaction throughput. The choice of an abnormally high value of WTW by one process only affects its own transaction response time; it does not have any adverse effect on the total throughput of the system. A low value of WTW would cause small commit groups, which in turn would limit the throughput. However, this can be avoided by administrative controls on the database that specify a minimum WTW time.

Hiber Design

The Hiber Design is similar to the Commit-Stall Design, but, instead of each process scheduling its own wake-up call, the group committer wakes up all processes in the committed group. In addition, the group committer must wake up the process that will be the next group committer.

Note, this design exhibits a serial wake-up behavior like the Commit-Lock Design, however, the mechanism is less costly than the VMS lock used by the Commit-Lock Design. In the Hiber Design, if a process is not the group committer, it simply sleeps; it does not schedule its own wake-up call. Therefore, each process is guaranteed to sleep and wake up at most once per commit, in contrast to the Commit-Stall Design. Another interesting characteristic of the Hiber Design is that the group committer can choose to either wake up the next group committer immediately, or it can actually schedule the wake-up call after a delay. Such a delay allows the next group size to become larger.

Experiments

We implemented and tested the Commit-Lock, the Commit-Stall, and the Willing-to-Wait designs in KODA. The objectives of our experiments were

- To find out which design would yield the maximum throughput under response time constraints
- To understand the performance characteristics of the designs

In the following sections, we present the details of our experiments, the results we obtained, and some observations.

Details of the Experiments

The hardware used for all of the following tests was a VAX 6340 with four processors, each rated at 3.6 VAX units of performance (VUP). The total possible CPU utilization was 400 percent and the total processing power of the computer was 14.4 VUPs. As the commit processing becomes more significant in a transaction (in relation to the other phases), the impact of the grouping mechanism on the transaction throughput increases. Therefore, in order to accentuate the performance differences between the various designs, we performed our experiments using a transaction that involved no database activity except to follow the commit sequence. So, for all practical purposes, the TPS data presented in this paper can be interpreted as "commit sequences per second." Also, note that our system imposed an upper limit of 50 on the grouping size.

Results

Using the Commit-Lock Design, transaction processing bottlenecked at 300 TPS. Performance greatly improved with the Commit-Stall Design; the maximum throughput was 464 TPS. The Willing-to-Wait Design provided the highest throughput, 500 TPS. Using this last design, it was possible to achieve up to a 66 percent improvement over the less-efficient Commit-Lock Design. Although both timer schemes, i.e., the Commit-Stall and Willing-to-Wait designs, needed tuning to set the parameters and the Commit-Lock Design did not, we observed that the maximum throughput obtained using timers is much better than that obtained with the lock. These results were similar to those of Reuter.

For our Willing-to-Wait Design, the minimum transaction duration is the WTW time. Therefore, the maximum TPS, the number of servers, and the WTW stall time, measured in milliseconds, are related by the formula: number of servers $\times$ 1000/WTW = maximum TPS. For example, our maximum TPS for the WTW design was obtained with 50 servers and 90 milliseconds WTW time. Using the formula, $50 \times 1000/90 = 555$. The actual TPS achieved was 500, which is 90 percent of the maximum TPS. This ratio is also a measure of the effectiveness of the experiment.

During our experiments, the maximum group size observed was 45 (with the Willing-to-Wait Design). This is close to the system-imposed limit of 50 and, so, we may be able to get better grouping with higher limits on the size of the group.

Observations

In the Commit-Stall and the Willing-to-Wait designs, given a constant stall, if the number of servers is increased, the TPS increases and then decreases. The rate of decrease is slower than the rate of increase. The TPS decrease is due to CPU overloading. The TPS increase is due to more servers trying to execute transactions and better CPU utilization. Figure 6 illustrates how TPS varies with the number of servers, given a constant stall WTW time.

Again, in the stalling designs, for a constant number of servers, if the stall is increased, the TPS increases and then decreases. The TPS increase is due to better grouping and the decrease is due to CPU underutilization. Figures 7 and 8 show the effects on TPS when you vary the commit-stall time or the WTW time, while keeping the number of servers constant.

To maximize TPS with the Commit-Stall Design, the following "mountain-climbing" algorithm was useful. This algorithm is based on the previous two observations. Start with a reasonable value of the stall and the number of servers, such that the CPU is underutilized. Then increase the number of servers. CPU utilization and the TPS increase.

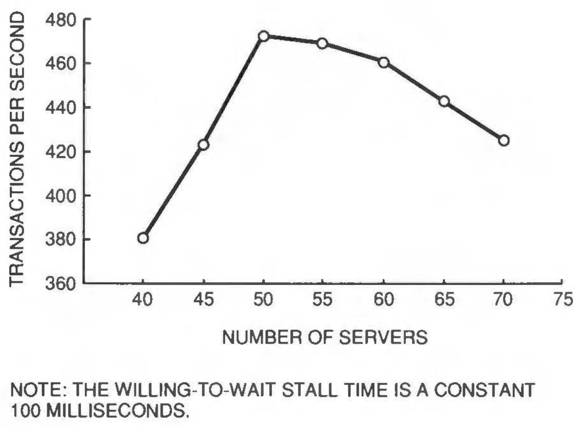


Figure 6 *Transactions per Second in Relationship to the Number of Servers, Given a Constant Willing-to-Wait Time*

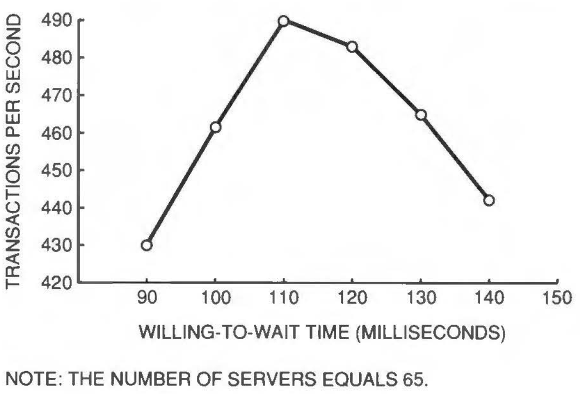


Figure 8 *Transactions per Second in Relationship to the WTW Time, Given a Constant Number of Servers*

Continue until the CPU is overloaded; then, increase the stall time. CPU utilization decreases, but the TPS increases due to the larger group size.

This algorithm demonstrates that increasing the number of servers and the stall by small amounts at a time increases the TPS, but only up to a limit. After this point, the TPS drops. When close to the limit, the two factors may be varied alternately in order to find the true maximum. Table 1 shows the performance measurements of the Commit-Stall Design. Comments are included in the table to highlight the performance behavior the data supports.

The same mountain-climbing algorithm is modified slightly to obtain the maximum TPS with the Willing-to-Wait Design. The performance measurements of this design are presented in Table 2. As we have seen before, the maximum TPS with this design is inversely proportional to the WTW time, while CPU is not fully utilized. The first four rows of Table 2 illustrate this behavior. The rest of the table follows the same pattern as Table 1.

The Willing-to-Wait Design performs slightly better than the Commit-Stall Design by adjusting to the variations in the speed at which different servers arrive at the commit point. Such variations are compensated for by the variable stalls in the Willing-to-Wait Design. Therefore, if the variation is high and the commit sequence is a significant portion of the transaction, we expect the Willing-to-Wait Design to perform much better than the Commit-Stall Design.

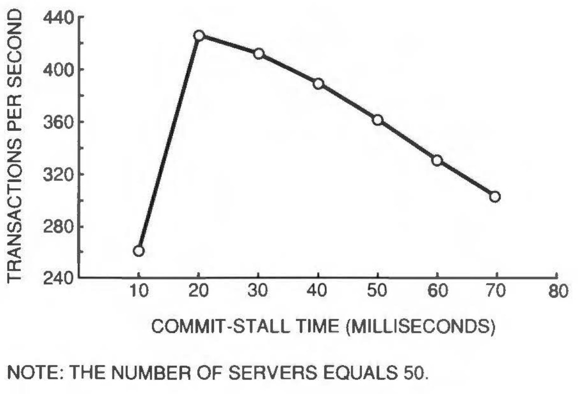


Figure 7 *Transactions per Second in Relationship to the Commit-Stall Time, Given a Constant Number of Servers*

Future Work

There is scope for more interesting work to further optimize commit processing in the KODA database kernel. First, we would like to perform experiments on the Hiber Design and compare it to the other designs. Next, we would like to explore ways of combining the Hiber Design with either of the two timer designs, Commit-Stall or Willing-to-Wait. This may be the best design of all the above, with a good mixture of automatic stall, low overhead, and explicit control over the total stall time. In addition, we would like to investigate the use of timers to ease system management. For example, a system administrator may increase the stalls for all transactions on the system in order to ease CPU contention, thereby increasing the overall effectiveness of the system.

Table 1 Commit-Stall Design Performance Data

Number of Servers	Commit Stall (Milliseconds)	CPU Utilization (Percent)*	TPS	Comments
50	20	360	425	Starting point
55	20	375	454	Increased number of servers, therefore, higher TPS
60	20	378	457	Increased number of servers, therefore, CPU saturated
60	30	340	461	Increased stall, therefore, CPU less utilized
65	30	350	464	Increased number of servers, maximum TPS
70	30	360	456	"Over-the-hill" situation, same strategy of further increasing the number of servers does not increase TPS
70	40	330	451	No benefit from increasing number of servers and stall
65	40	329	448	No benefit from just increasing stall

* Four processors were used in the experiments. Thus, the total possible CPU utilization is 400 percent.

Table 2 Willing-to-Wait Performance Data

Number of Servers	Willing-to-Wait Stall (Milliseconds)	CPU Utilization (Percent)*	TPS	Comments
45	100	285	426	Starting point, CPU not saturated
45	90	295	466	Decreased stall to load CPU, CPU still not saturated
45	80	344	498	Decreased stall again
45	70	363	471	Further decreased stall, CPU almost saturated
50	80	372	485	Increased number of servers, CPU more saturated
50	90	340	500	Increased stall to lower CPU usage, maximum TPS
55	90	349	465	"Over-the-hill"situation, same strategy of further increasing number of servers does not increase TPS
50	100	324	468	No benefit from just increasing stall

* Four processors were used in the experiments. Thus, the total possible CPU utilization is 400 percent.

Conclusions

We have presented the concept of group commit processing as well as a general analysis of various options available, some trade-offs involved, and some performance results indicating areas for possible improvement. It is clear that the choice of the algorithm can significantly influence performance at high transaction throughput. We are optimistic that with some further investigation an optimal commit sequence can be incorporated into Rdb/VMS and VAX DBMS with considerable gains in transaction processing performance.

Acknowledgments

We wish to acknowledge the help provided by Rabah Mediouni in performing the experiments discussed in this paper. We would like to thank Phil Bernstein and Dave Lomet for their careful reviews of this paper. Also, we want to thank the other KODA group members for their contributions during informal discussions. Finally, we would like to acknowledge the efforts of Steve Klein who designed the original KODA group commit mechanism.

References

1. P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group Commit Timers and High Volume Transaction Processing Systems," *High Performance Transaction Systems, Proceedings of the 2nd International Workshop* (September 1987).

2. D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *Database Engineering* (June 1985).

William F. Bruckert
Carlos Alonso
James M. Melvin

Verification of the First Fault-tolerant VAX System

The fault-tolerant character of the VAXft 3000 system required that plans be made early in the development stages for the verification and test of the system. To ensure proper test coverage of the fault-tolerant features, engineers built fault-insertion points directly into the system hardware. During the verification process, test engineers used hardware and software fault insertion in directed and random test forms. A four-phase verification strategy was devised to ensure that the VAXft system hardware and software was fully tested for error recovery that is transparent to applications on the system.

The VAXft 3000 system provides transparent fault tolerance for applications that run on the system. Because the 3000 includes fault-tolerant features, verification of the system was unlike that ordinarily conducted on VAX systems. To facilitate system test, the verification strategy outlined a four-phase approach which would require hardware to be built into the system specifically for test purposes.

This paper presents a brief overview of the VAXft system architecture and then describes the methods used to verify the system's fault tolerance.

VAXft 3000 Architectural Overview

The VAXft fault-tolerant system is designed to recover from any single point of hardware failure. Fault tolerance is provided transparently for all applications running under the VMS operating system. This section reviews the implementation of the system to provide background for the main discussion of the verification process.

The system comprises two duplicate systems, called zones. Each zone is a fully functional computer with enough elements to run an operating system. These two zones, referred to as zone A and zone B, are shown in Figure 1, which illustrates the duplication of the system components. The two independent zones are connected by duplicate cross-link cables. The cabinet of each zone also includes a battery, a power regulator, cooling fans, and an AC power input. Each zone's hardware has sufficient error checking to detect all single faults within that zone.

Figure 2 is a block diagram of a single zone with one I/O adapter. Note the portions of the zone labeled dual-rail and single-rail. The dual-rail portions of the system have two independent sets of hardware performing the same operations. Correct operation is verified by comparison. The fault-detection mechanism for the single-rail I/O modules combines checking codes and communication protocols.

The system performs I/O operations by sending and receiving message packets. The packets are exchanged between the CPU and various servers, including disks, Ethernet, and synchronous lines. These message packets are formed and interpreted in the dual-rail portion of the system. They are protected in the single-rail portion of the machine by check codes which are generated and checked in the dual-rail portion of the machine. Corrupted packets can be retransmitted through the same or alternate paths.

In the normal mode of fault-tolerant operation, both zones execute the same instruction at the same time. The four processors (two in each zone) appear to the operating system as a single logical CPU. The hardware supplies the detection and recovery facilities for faults detected in the CPU and memory portions of the system. A defective CPU module and its memory are automatically removed from service by the hardware, and the remaining CPU continues processing.

Error handling for the I/O interconnections is managed differently. The paths to and from I/O adapters are duplicated for checking purposes. If a fault is detected, the hardware retries the operation. If the retry is successful, the error is logged, and operation continues without software assistance.

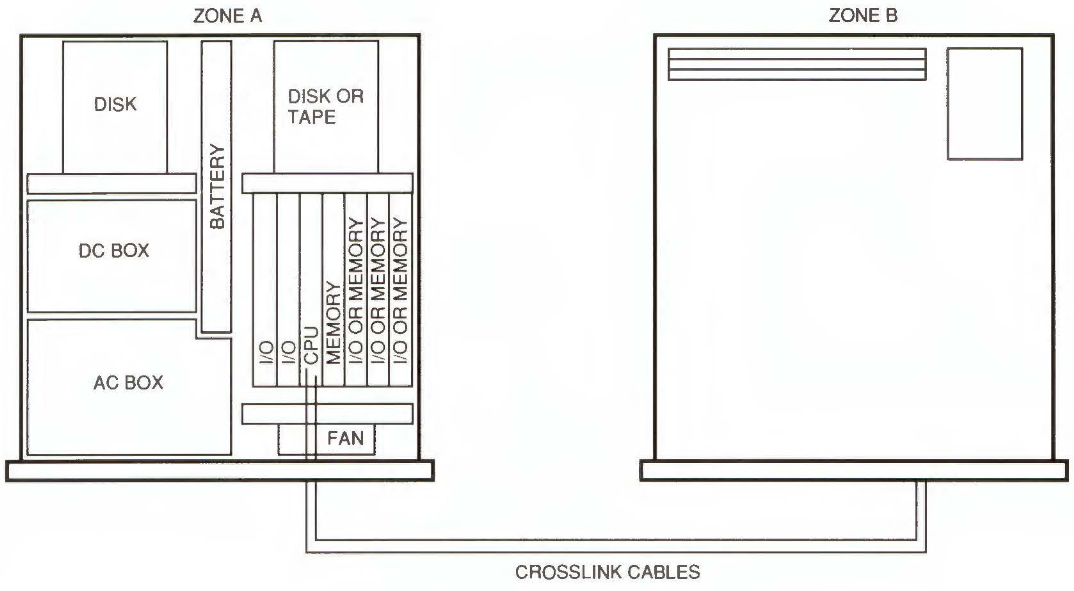


Figure 1 A Dual-zone VAXft System

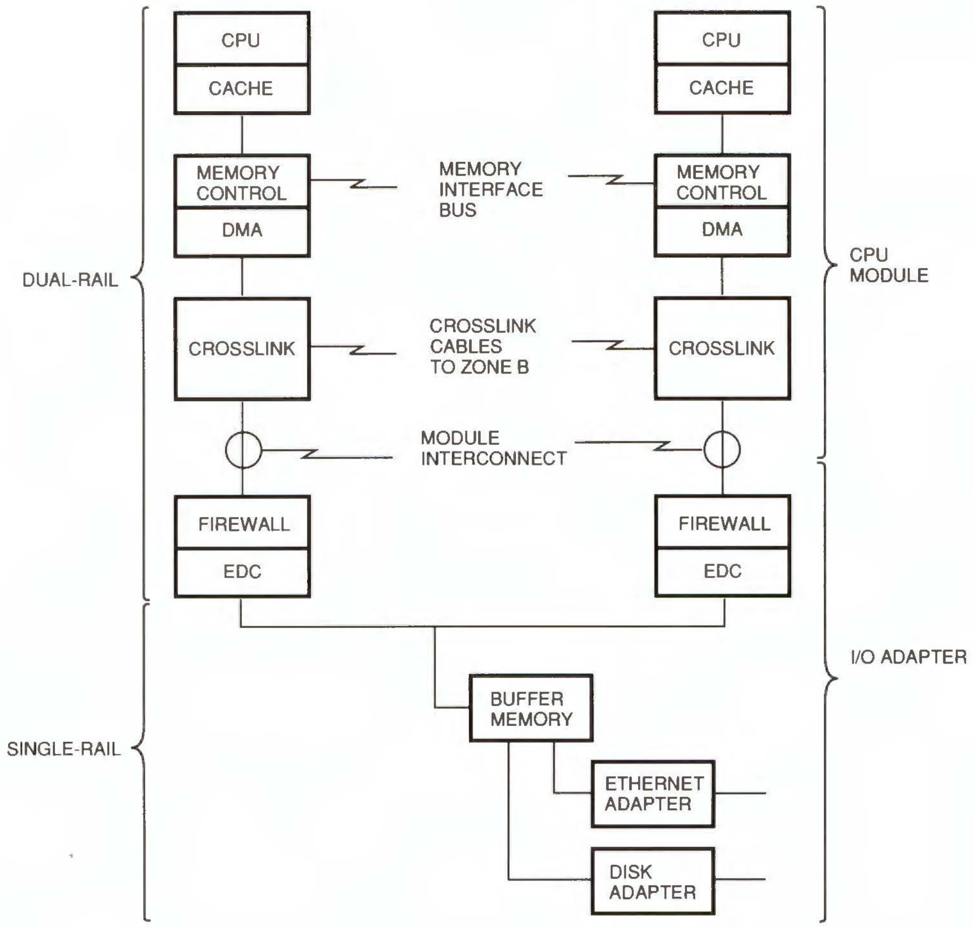


Figure 2 Single-zone Structure of a VAXft 3000 System

If the retry is unsuccessful, the Fault-tolerant System Services (FTSS) software performs error recovery. FTSS is a layered software product that is utilized with every VAXft 3000 system. It provides the software necessary to complete system error recovery. For system recovery from a failed I/O device, an alternate path or device is used. All recoverable faults have an associated maximum threshold value. If this threshold is exceeded, FTSS performs appropriate device reconfiguration.

Verification of a Fault-tolerant VAX System

This section entails a discussion of the types of system tests and the fault-insertion techniques used to ensure the correct operation of the VAXft system. In addition, the four-phase verification strategy and the procedures involved in each phase are reviewed.

There are two types of system tests: directed and random. Directed tests, which test specific hardware or software features, are used most frequently in computer system verification and follow a strict test sequence. Complex systems, however, cannot be completely verified in a directed fashion.[1] As a case in point, an operating system running on a processor has innumerable states. Directed tests verify functional operation under a particular set of conditions. They may not, however, be used to verify that same functionality under all possible system conditions.

In comparison, random testing allows multiple test processes to interact in a pseudo-random or random fashion. In random testing, test coverage is increased with additional run-time. Thus, once the proper test processes are in place, the need to develop additional tests in order to increase coverage is eliminated. This type of testing also reduces the effects of the biases of the engineers generating the tests. While directed testing can provide only a limited level of coverage, this coverage level can be well understood. Random testing offers a potentially unbounded level of coverage; however, quantifying this coverage is difficult if not impossible.

To achieve the proper level of verification, the VAXft verification utilized a balance of directed and random testing. Directed testing was used to achieve a certain base level of functionality, and random testing was used to expand the level of coverage.

To permit testing of system fault tolerance in a practical amount of time, some form of fault insertion is required. The reliability of components used in computer systems has been improving, and more importantly, the number of components used to implement any function has been dramatically decreasing. These factors have produced a corresponding reduction in system failure rates. Given the high reliability of today's machines, it is not practical from a verification standpoint to verify a system by letting it run until failures occur.

Conceptually, faults can be inserted in two ways. First, memory locations and registers can be corrupted to mimic the results of gate-level faults (software fault insertion). Second, gate-level faults may be inserted directly into the hardware (hardware fault insertion). There are advantages to both techniques. One advantage of software-implemented fault insertion is that no embedded hardware support is required.[2] The advantage of hardware fault insertion, on the other hand, is that faults are more representative of actual hardware failures and can reveal unanticipated side effects from a gate-level failure. To utilize hardware fault insertion, either a mechanism must be designed into the system, or an external insertion device must be developed once the hardware is available. Given the physical feature size of the components used today, it is virtually impossible to achieve adequate fault-insertion coverage through an external fault-insertion mechanism.

The error detection and recovery mechanism determines which fault insertion technique is suitable for each component. Some examples illustrate this point. For the lockstep portion of the VAXft 3000 CPUs, software fault insertion is not suitable because the lockstep functionality prevents corruption of memory or registers when faults occur. Therefore, hardware faults cannot be mimicked by modifying memory contents. However, the software fault-insertion technique was suitable to test the I/O adapters since the system handles faults in the adapters by detecting the corruption of data. Hardware fault insertion was not suitable because the I/O adapters were implemented with standard components that did not support hardware fault insertion.

Because the verification strategy for the 3000 was considered a fundamental part of the system development effort, fault insertion points were built directly into the system hardware. The amount of logic necessary to implement fault insertion is relatively small. The goals of the fault-insertion hardware were to

- Eliminate any corruption of the environment under test that could result from fault insertion. For example, if a certain type of system write operation is required to insert a fault, then every test case will be done on a system that is in a "post-fault-insertion" state.

- Enable the user to distribute faults randomly across the system.
- Allow insertion of faults during system operation.
- Enable testing of transient and solid faults.

The fault-insertion points are accessed through a separate serial interface bus isolated from the operating hardware. This separate interface ensures that the environment under test is unbiased by fault insertion.

Even with hardware support for fault insertion, only a small number of fault-insertion points can be implemented relative to the total number possible. Where the number of fault-insertion points is small, the selection of the fault-insertion points is important to achieve a random distribution. Fault-insertion points were designed into most of the custom chips in the VAXft system. When the designers were choosing the fault-insertion points, a single bit of a data path was considered sufficient for data path coverage. Since a significant portion of the chip area is consumed by data paths, a high level of coverage of each chip was achieved with relatively few fault-insertion points. The remaining fault-insertion points could then be applied to the control logic. Coverage of this logic was important because control logic faults result in error modes that are more unpredictable than data path failures.

The effect that a given fault has on the system depends on the current system operation and when in that operation the fault was inserted. In the 3000, for example, a failure of bit 3 in a data path will have significantly different behavior depending upon whether the data bit was incorrect during the address transmission portion of a cycle or during the succeeding data portion. Therefore, the timing of the fault insertion was pseudo-random. The choice of pseudo-random insertion was based on the fact that the fault-insertion hardware operated asynchronously to the system under test. This meant that faults could be inserted at any time, without correlation to the activity of the system under test.

Faults may be transient or solid in nature. For design purposes, a solid fault was defined as a failure that will be present on retry of an operation. A transient fault was defined as a fault that will not be present on retry of the operation. Transient faults do not require the removal of the device that experienced the fault; solid faults do require device removal. Since the system reacts differently to transient and hard faults, both types of faults had to be verified in the VAXft system. Therefore, it was required that the fault-insertion hardware be capable of inserting solid or transient faults. Solid faults were inserted by continually applying the fault-insertion signal. Transient faults were inserted by applying the fault-insertion signal only until the machine detected an error.

As noted earlier, the verification strategy utilized both hardware and software fault insertion. The hardware fault-insertion mechanisms allowed faults to be inserted into any system environment, including diagnostics, exercisers, and the VMS operating system. As such, it was used for initial verification as well as regression testing of the system. The verification strategy for the VAXft 3000 system involved a multiphase effort. Each of the following four verification phases built upon the previous phase:

1. Hardware verification under simulation
2. Hardware verification with system exerciser and fault insertion
3. System software verification with fault insertion
4. System application verification with fault insertion

Figure 3 shows the functional layers of the VAXft 3000 system in relation to the verification phases. The numbered brackets to the right of the diagram correlate to the testing coverage of each layer. For example, the system software verification, phase 3, verified the VMS system, Fault-tolerant System Services (FTSS), and the hardware platform.

The following sections briefly describe the four phases of the VAXft verification.

Hardware Verification under Simulation

Functional design verification using software simulation is inherently slow in a design as large as the VAXft 3000 system. To use resources most efficiently, a verification effort must incorporate a number of different modeling levels, which means trading off detail to achieve other goals such as speed.[3]

VAXft 3000 simulation occurred at two levels: the module level and the system level. Module-level simulation verified the base functionality of each module. Once this verification was complete, a system-level model was produced to validate the intermodule functionality. The system-level model

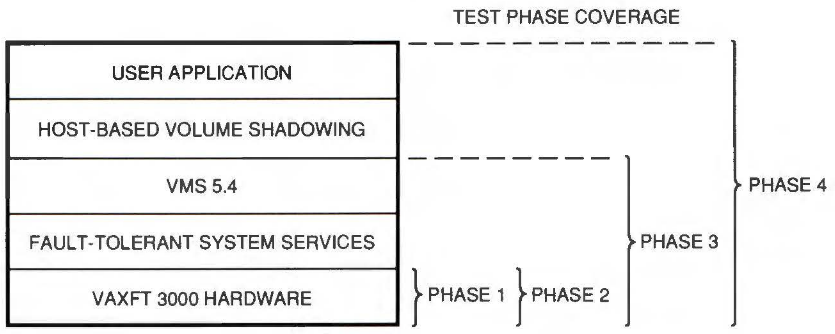


Figure 3 Functional Layers of the VAXft 3000 System in Relation to the Verification Phases

consisted of a full dual-rail, dual-zone system with an I/O adapter in each zone. At the final stage, full system testing was performed.

More than 500 directed error test cases were developed for gate-level system simulation. For each test, the test environment was set up on a fully operational system model, and then the fault was inserted. A simulation controller was developed to coordinate the system operations in the simulation environment. The simulation controller provided the following control over the testing:

- Initialization of all memory elements and certain system registers to reduce test time
- Setup of all memory data buffers to be used in testing
- Automated test execution
- Automated checking of test results
- Log of test results

For each test case, the test environment was selected from the following: memory testing, I/O register access, direct memory access (DMA) traffic, and interrupt cycles. In any given test case, any number of the previous tests could be run. These environments could be run with or without faults inserted. In addition, each environment consisted of multiple test cases. In an error handling test case, the proper system environment required for the test was set, and then the fault was inserted into the system. The logic simulator used was designed to verify logic design. When an illegal logic condition was detected, it produced an error response. When a fault insertion resulted in an illegal logic condition, the simulator responded by invalidating the test. Because of this, a great deal of time was spent to ensure that faults were inserted in a way that would not generate illegal conditions. Each test case was considered successful only when the system error registers contained the correct data and the system had the ability to continue operation after the fault.

Hardware Verification with System Exerciser and Fault Insertion

After the prototypes were available, the verification effort shifted from simulation to fault insertion on the hardware. The goal was to insert faults using an exerciser that induced stressful, reproducible hardware activity and that allowed us to analyze and debug the fault easily.

Exerciser test cases were developed to stress the various hardware functions. The tests were designed to create maximum interrupt and data transfer activity between the CPU and the I/O adapters. These functions could be tested individually or simultaneously. The exerciser scheduler provided a degree of randomness such that the interaction of functions was representative of a real operating system. The fault-insertion hardware was used to achieve a random distribution of fault cases across the system.

Because it was possible to insert initial faults while specific functions were performed, a great degree of reproducibility was achieved that aided debug efforts. Once the full suite of tests worked correctly, fault insertion was performed while the system continually switched between all functions. This testing was more representative of actual faults in customer environments, but was less reproducible.

As previously mentioned, the hardware fault-insertion tool allowed the insertion of both transient and solid failures. The VAXft 3000 hardware recovers from transient failures and utilizes software recovery for hard failures. Since the goal of phase 2 testing was to verify the hardware, the focus was on transient fault insertion. Two criteria for each error case determined the success of the test. First and foremost, the system must continue to run and to produce correct results. Second, the error data that the system captures must be correct based on the fault that was inserted. Correct error data is important because it is used to identify the failing component both for software recovery and for servicing.

Although the simulation environment of phase 1 was substantially slower than phase 2, it provided the designers with more information. Therefore when problems were discovered on the prototypes used in phase 2, the failing case was transferred to the simulator for further debugging. The hardware verification also validated the models and test procedures used in the simulation environment.

System Software Verification with Fault Insertion

In parallel with hardware verification, the VAXft 3000 system software error handling capabilities were tested. This phase represented the next higher level of testing. The goal was to verify the VAX functionality of the 3000 system as well as the software recovery mechanisms.

Digital has produced various test packages to verify VAX functionality. Since the VAXft 3000 system incorporates a VAX chip set used in the VAX 6000 series, it was possible to use several standard test packages that had been used to verify that system.[1]

Fault-tolerant verification, however, was not addressed by any of the existing test packages. Therefore, additional tests were developed by combining the existing functional test suite with the hardware fault-insertion tool and software fault-insertion routines. Test cases used included cache failure, clock failure, memory failure, interconnect failures, and disk failures. These failures were applied to the system during various system operations. In addition, servicing errors were also tested by removing cables and modules while the system was running. The completion criteria for tests included the following:

- Detection of the fault
- Isolation of the failed hardware
- Continuation of the test processes without interruption

System Application Verification with Fault Insertion

The goals for the final phase of the VAXft 3000 verification were to run an application with fault insertion and to demonstrate that any system fault recovery action had no effect on the process integrity and data integrity of the application. The application used in the testing was based on the standard DebitCredit banking benchmark and was implemented using the DECintact layered product. The bank has 10 branches, 100 tellers, and 3,600 customer accounts (10 tellers and 360 accounts per branch). Traffic on the system was simulated using terminal emulation process (VAXRTE) scripts representing bank teller activity. The transaction rate was initially one transaction per second (TPS) and was varied up to the maximum TPS rate to stress the system load.

The general test process can be described as follows:

1. Started application execution. The terminal emulation processes emulating the bank tellers were started and continued until the system was operating at the desired TPS rating.
2. Invoked fault insertion. A fault was selected at random from a table of hardware and software faults. The terminal emulation process submitted stimuli to the application before, during, and after fault insertion.
3. Stopped terminal emulation process. The application was run until a quiescent state was reached.
4. Performed result validation. The process integrity and data integrity of the application were validated.

All the meaningful events were logged and time-stamped during the experiments. Process integrity was proved by verifying continuity of transaction processing through failures. The time stamps on the transaction executions and the system error logs allowed these two independent processes to be correlated.

The proof of data integrity consisted of using the following consistency rules for transactions:

1. The sum of the account balances is equal to the sum of the teller balances, which is equal to the sum of the branch balances.
2. For each branch, the sum of the teller balances is equal to the branch balance.

3. For each transaction processed, a new record must be added to the history file.

Application verification under fault insertion served as the final level of fault-tolerant validation. Whereas the previous phases ensured that the various components required for fault tolerance operated properly, the system application verification demonstrated that these components could operate together to provide a fully fault-tolerant system.

Conclusions

The process of verifying fault tolerance requires a strong architectural test plan. This plan must be developed early in the design cycle because hardware support for testing may be required. The verification plan must demonstrate cognizance of the capabilities and limitations at each phase of the development cycle. For example, the speed of simulation prohibits verification of software error recovery in a simulation environment. Also, when a system is implemented with VLSI technology, the ability to physically insert faults into the system by means of an external mechanical mechanism may not be adequate to properly verify the correct system error recovery. These and other issues must be addressed before the chips are fabricated or adequate error recovery verification may not be possible. Inadequate error recovery verification directly increases the risk of real, unrecoverable faults resulting in system outages.

The verification plan for the VAXft 3000 system consisted of the following phases and objectives:

1. Hardware simulation with fault insertion verified error detection, hardware recovery, and error data capture.

2. System exerciser with fault insertion enhanced the coverage of the hardware simulation effort.

3. System software with fault insertion verified software error recovery and reporting.

4. System software verification with fault insertion verified the transparency of the system error recovery to the application running on the system.

The test of any fault tolerant system is to survive a *real* fault while running a customer application. Removing a module from a machine may be an impressive test, but machines fail as a result of modules falling out of the backplane. The initial test of the VAXft 3000 system showed that the system would survive most of the faults introduced. Tests also revealed problems that would have resulted in system outages if left uncorrected. System enhancements were made in the areas of system recovery actions and repair call out. Whereas some of the problems were simple coding errors, others were errors in carefully reviewed and documented algorithms. Simply put, the collective wisdom of the designers was not always sufficient to reach the degree of accuracy desired for this fault-tolerant system.

As the VAXft product family evolves, performance and functional enhancements will be available. The test processes described in this paper will remain in use, so that every future release of software will be better than the previous one. The combination of hardware and software fault insertion, coupled with physical system disruption allows testing to occur at such a greatly accelerated rate, that all testing performed will be repeated for every new release.

References

1. J. Croll, L. Camilli, and A. Vaccaro, "Test and Qualification of the VAX 6000 Model 400 System," *Digital Technical Journal,* vol. 2, no. 2 (Spring 1990): 73-83.

2. J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault Injection Experiments Using FIAT (Fault Injection-based Automated Testing," *IEEE Transactions on Computers,* vol. 39, no. 4 (April 1990).

3. R. Calcagni and W. Sherwood, "VAX 6000 Model 400 CPU Chip Set Functional Design Verification," *Digital Technical Journal,* vol. 2, no. 2 (Spring 1990): 64-72.

Further Readings

The Digital Technical Journal *publishes papers that explore the technological foundations of Digital's major products. Each Journal focuses on at least one product area and presents a compilation of papers written by the engineers who developed the product. The content for the Journal is selected by the Journal Advisory Board.*

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

VAX 9000 Series
Vol. 2, No. 4, Fall 1990
The technologies and processes used to build Digital's first mainframe computer, including papers on the architecture, microarchitecture, chip set, vector processor, and power system, as well as CAD and test methodologies

DECwindows Program
Vol. 2, No. 3, Summer 1990
An overview and descriptions of the enhancements Digital's engineers have made to MIT's X Window System in such areas as the server, toolkit, interface language, and graphics, as well as contributions made to related industry standards

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990
The highly expandable and configurable midrange family of VAX systems that includes a vector processor, a high-performance scalar processor, and advances in chip design and physical technology

Compound Document Architecture
Vol. 2, No. 1, Winter 1990
The CDA family of architectures and services that support the creation, interchange, and processing of compound documents in a heterogeneous network environment

Distributed Systems
Vol. 1, No. 9, June 1989
Products that allow system resource sharing throughout a network, the methods and tools to evaluate product and system performance

Storage Technology
Vol. 1, No. 8, February 1989
Engineering technologies used in the design, manufacture, and maintenance of Digital's storage and information management products

CVAX-based Systems
Vol. 1, No. 7, August 1988
CVAX chip set design and multiprocessing architecture of the mid-range VAX 6200 family of systems and the MicroVAX 3500/3600 systems

Software Productivity Tools
Vol. 1, No. 6, February 1988
Tools that assist programmers in the development of high-quality, reliable software

VAXcluster Systems
Vol. 1, No. 5, September 1987
System communication architecture, design and implementation of a distributed lock manager, and performance measurements

VAX 8800 Family
Vol. 1, No. 4, February 1987
The microarchitecture, internal boxes, VAXBI bus, and VMS support for the VAX 8800 high-end multiprocessor, simulation, and CAD methodology

Networking Products
Vol. 1, No. 3, September 1986
The Digital Network Architecture (DNA), network performance, LANbridge 100, DECnet-ULTRIX and DECnet-DOS, monitor design

MicroVAX II System
Vol. 1, No. 2, March 1986
The implementation of the microprocessor and floating point chips, CAD suite, MicroVAX workstation, disk controllers, and TK50 tape drive

VAX 8600 Processor
Vol. 1, No. 1, August 1985
The system design with pipelined architecture, the I-box, F-box, packaging considerations, signal integrity, and design for reliability

Subscriptions to the *Digital Technical Journal* are available on a yearly, prepaid basis. The subscription rate is $40.00 per year (four issues). Requests should be sent to Cathy Phillips, Digital Equipment Corporation, MLO1-3/B68, 146 Main Street, Maynard, MA 01754, U.S.A. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* can be ordered from Digital Press at a cost of $16.00 per copy.

Technical Papers and Books by Digital Authors

P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Reading, MA: Addison-Wesley, 1987).

P. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," *Proceedings 1990 ACM SIGMOD Conference on Management of Data* (May 1990).

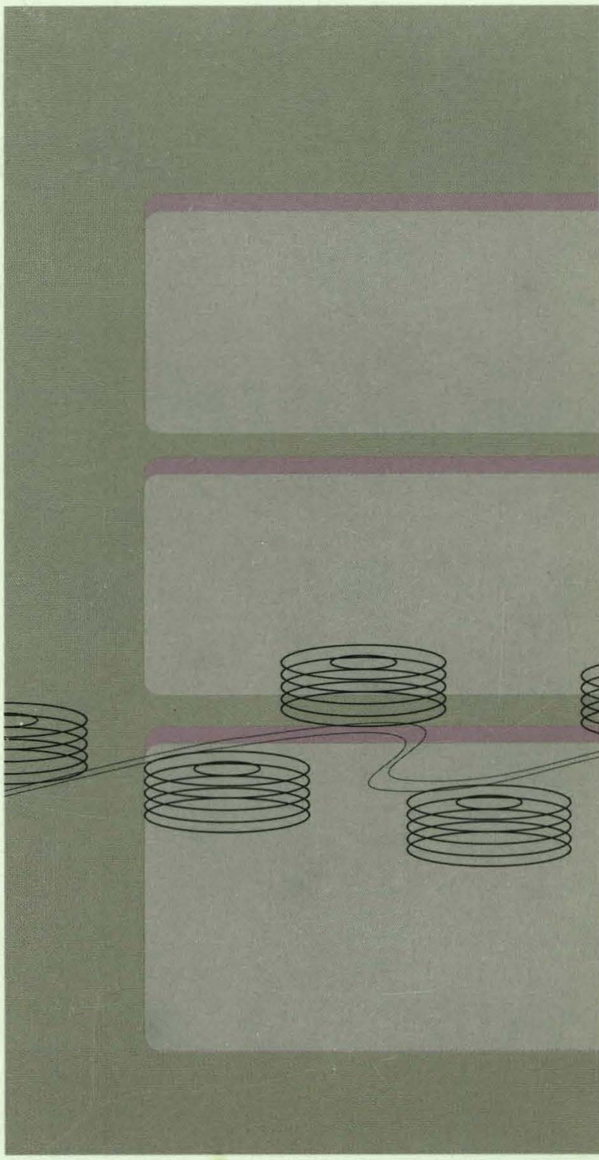
T.K. Rengarajan, P. Spiro, and W. Wright, "High Availability Mechanisms of VAX DBMS Software," *Digital Technical Journal*, vol. 1, no. 8 (February 1989): 88-98.

K. Morse, "The VMS/MicroVMS Merge," *DEC Professional Magazine*, vol. 7, no. 5 (May 1988).

K. Morse and R. Gamache, "VAX/SMP," *DEC Professional Magazine*, vol. 7, no. 4 (April 1988).

K. Morse, "Shrinking VMS," *Datamation* (July 15, 1984).

L. Frampton, J. Schriesheim, and M. Rountree, "Planning for Distributed Processing," *Auerbach Report on Communications* (1989).



ISSN 0898-901X