Digital Technical Journal

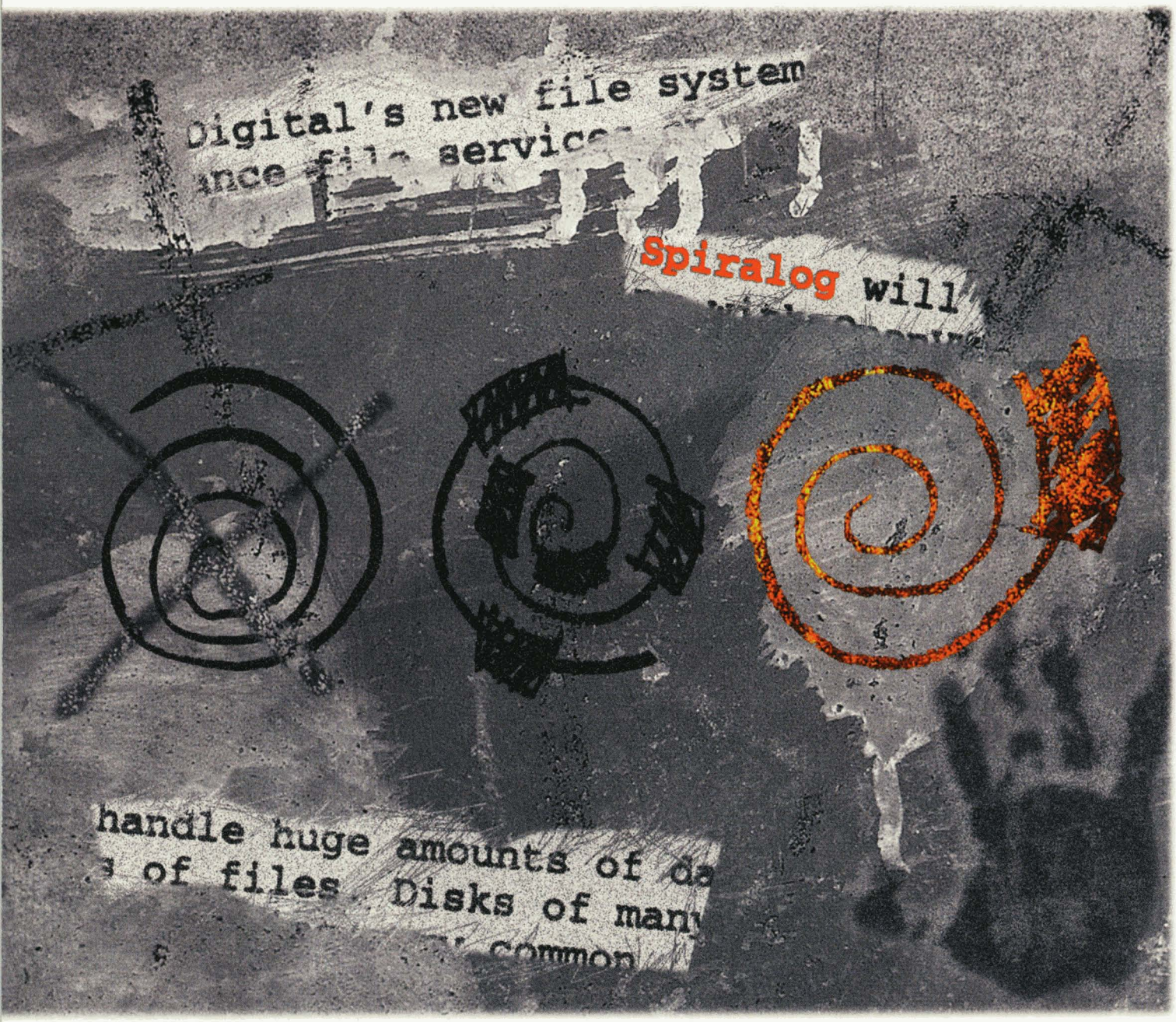
SPIRALOG LOG-STRUCTURED FILE SYSTEM

OPENVMS FOR 64-BIT ADDRESSABLE VIRTUAL MEMORY

HIGH-PERFORMANCE MESSAGE PASSING FOR CLUSTERS

SPEECH RECOGNITION SOFTWARE

digital™



Volume 8 Number 2
1996

Cover Design
Digital's new Spiralog file system, a featured topic in the issue, supports full 64-bit system capability and fast backup and is integrated with the OpenVMS 64-bit version 7.0 operating system. The cover graphic captures the inspired character of the Spiralog design effort and illustrates a concept taken from University of California research in which the whole disk is treated as a single, sequential log and all file system modifications are appended to the tail of the log.

The cover was designed by Lucinda O'Neill of Digital's Design Group using images from PhotoDisc, Inc., copyright 1996.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, MA 01460.

Subscriptions can be ordered by sending a check in U.S. funds (made payable to Digital Equipment Corporation) to the published-by address. General subscription rates are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Digital's customers may qualify for gift subscriptions and are encouraged to contact their account representatives.

Single copies and back issues are available for $16.00 (non-U.S. $18) each and can be ordered by sending the requested issue's volume and number and a check to the published-by address. See the Further Readings section in the back of this issue for a complete listing. Recent issues are also available on the Internet at **http://www.digital.com/info/dtj.**

Digital employees may order subscriptions through Readers Choice at URL http://webrc.das.dec.com or by entering VTX PROFILE at the system prompt.

Inquiries, address changes, and complimentary subscription orders can be sent to the *Digital Technical Journal* at the published-by address or the electronic mail address, dtj@digital.com. Inquiries can also be made by calling the *Journal* office at 508-486-2538.

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

ISSN 0898-901X

Documentation Number EC-N6992-18

Book production was done by Quantic Communications, Inc.

Contents

Editor's Introduction

This past spring when we surveyed *Journal* subscribers, readers took the time to comment on the particular value of the issues featuring Digital's 64-bit Alpha technology. The engineering described in those two issues continues, with ever higher levels of performance in Alpha microprocessors, servers, clusters, and systems software. This issue presents recent developments: a log-structured file system, called Spiralog; the OpenVMS operating system extended to take full advantage of 64-bit addressing; high-performance computing software for Alpha clusters; and speech recognition software for Alpha workstations.

Spiralog is a wholly new clusterwide file system integrated with the new 64-bit OpenVMS version 7.0 operating system and is designed for high data availability and high performance. The first of four papers about Spiralog is written by Jim Johnson and Bill Laing, who introduce log-structured file (LFS) concepts, the university research behind the design, and design innovations.

The advantages of LFS technology over conventional "update-in-place" technology are explained by Chris Whitaker, Stuart Bayley, and Rod Widdowson. In their paper about the file server design, they compare the Spiralog implementation of the LFS technology with others and describe the novel combination of the technology with a B-tree mapping mechanism to provide the system with needed data recovery guarantees.

A third paper about Spiralog, written by Russ Green, Alasdair Baird, and Chris Davies, addresses a critical customer requirement—fast, application-consistent, on-line backup. Exploiting the features of log-structured storage, designers were able to combine the flexibility of file-based backup and the high performance of physically oriented backup. Consistent copies of the file system are created while applications modify data.

The Spiralog integration into the OpenVMS file system required that existing applications be able to run unchanged. Mark Howell and Julian Palmer describe the integration of the write-back caching used in Spiralog into the write-through environment used in the existing Files-11 file system.

The importance of compatibility for existing 32-bit applications in a 64-bit environment is stressed again in the set of three papers about the latest step in the evolution of the OpenVMS operating system. Digital first ported the 32-bit OpenVMS operating system to the Alpha architecture in 1992. The extension of the system to exploit 64-bit virtual addressing is presented by Mike Harvey and Lenny Szubowicz. Their discussion includes the team's solution to significant scaling issues that involved a new approach to page-table residency.

The OpenVMS team anticipated that applications would mix 32- and 64-bit addresses, or pointers, in the new environment. Tom Benson, Karen Noel, and Rich Peterson explain why this mixing of pointer sizes is expected and the DEC C compiler solution they developed to support the practice. In a related discussion, Duane Smith's paper reviews new techniques the team used to analyze and modify the C run-time library interfaces that accommodate applications using 32-bit, 64-bit, or both address sizes.

Designed for scientific users, the parallel-programming tool next described does not run on the OpenVMS Alpha system but instead on UNIX clusters connected with MEMORY CHANNEL technology. Jim Lawton, John Brosnan, Morgan Doyle, Seosamh Ó Riordáin, and Tim Reddin review the challenges in designing the TruCluster MEMORY CHANNEL Software product, which is a message-passing system intended for builders of parallel software libraries and implementers of parallel compilers. The product reduces communications latency to less than 10 μs in shared memory systems.

Finally, Bernie Rozmovits presents the design of user interfaces for the Digital Speech Recognition Software (DSRS) product. Although DSRS is targeted for Digital's Alpha workstations running UNIX, the implementation issues examined and the team's efforts to ensure the product's ease-of-use can be generally applied to speech recognition product development.

Coming up are papers on a variety of topics, including the internet protocol, collaborative software for the internet, and high-performance servers. These topics reflect areas of interest *Journal* readers rated near the top in last spring's survey. Our sincere thanks go to everyone who responded to that survey.

Jane Blake

Jane C. Blake
Managing Editor

Foreword

Rich Marcello
Vice President, OpenVMS Systems Software Group

The papers you will read in this issue of the *Journal* describe how we in the OpenVMS engineering community set out to bring the OpenVMS operating system and our loyal customer base into the twenty-first century. The papers present both the development issues and the technical challenges faced by the engineers who delivered the OpenVMS operating system version 7.0 and the Spiralog file system, a new log-structured file system for OpenVMS.

We are extremely proud of the results of these efforts. In December 1995 at U.S. Fall DECUS (Digital Equipment Computer Users Society), Digital announced OpenVMS version 7.0 and the Spiralog file system as part of a first wave of product deliveries for the OpenVMS Windows NT Affinity Program. OpenVMS version 7.0 provides the "unlimited high end" on which our customers can build their distributed computing environments and move toward the next millennium.

The release of OpenVMS version 7.0 in January of this year represents the most significant engineering enhancement to the OpenVMS operating system since Digital released the VAXcluster system in 1983. OpenVMS version 7.0 extends the 32-bit architecture of OpenVMS to a 64-bit architecture, allowing OpenVMS Alpha users to fully exploit the 64-bit virtual address capacity of the Alpha architecture. As you will read in some of the papers in this issue, however, our design goal for OpenVMS version 7.0 went beyond just delivering 64-bit virtual address capability to OpenVMS users. It was essential to us that OpenVMS users be able to upgrade to version 7.0 with full compatibility for their existing 32-bit applications.

In addition to achieving the significant goals of 64-bit addressing and compatibility for 32-bit applications, version 7.0 includes very large memory (VLM), very large database (VLDB), fast I/O, fast path, and symmetric multiprocessing (SMP) enhancements. These new features recently combined with the power of the Alpha architecture to earn OpenVMS a world record for performance. In May of this year, OpenVMS version 7.0 on an AlphaServer 8400 system configured with eight processors and 8 gigabytes of memory, running Oracle's Rdb7 database and using the ACMS transaction processing monitor, set a new world record for TPC-C performance on a single SMP system. Audited performance was 14,227 tpmC at $269 per tpmC. Just this past August, the combination of OpenVMS version 7.0, Oracle's Rdb7 database, the ACMS monitor, and the AlphaServer 4100 system achieved world-record departmental server performance. The new world record was set on an AlphaServer 4100 5/400 system configured with four processors and 4 gigabytes of memory. In audited benchmarks, the performance results were 7,985 tpmC at $173 per tpmC.

Such outstanding results are achievable in a full 64-bit environment—hardware architecture, operating systems, and applications such as Oracle's Rdb database. No other vendor today can deliver this power.

In fact, Digital has two 64-bit operating systems with this power: the OpenVMS and the Digital UNIX operating systems.

As noted above, Digital introduced the OpenVMS operating system with support for full 64-bit virtual addressing at the same time it introduced the Spiralog file system, in December 1995. The Spiralog design is based on the Sprite log-structured file system from the University of California, Berkeley. With its use of this log-structured approach, Spiralog offers major new performance features, including fast, application-consistent, on-line backup. Further, it is fully compatible with customers' existing Files-11 file systems, and applications that run on Files-11 will run on Spiralog with no modification. To deliver all of the features we felt were essential to meet the needs of our loyal customer base, the Spiralog team examined and resolved a number of technical issues. The papers in this issue describe some of the challenges they faced, including the decision to design a Files-11 file system emulation.

The delivery of the OpenVMS version 7.0 operating system and the Spiralog file system are part of Digital's continued commitment to the OpenVMS customer base. These products represent the work of dedicated, talented engineering teams that have deployed state-of-the-art technology in products that will help our customers remain competitive for years to come.

In the OpenVMS group as elsewhere in Digital, we are committed to excellence in the development and delivery of business computing solutions. We will continue to maintain and enhance a product portfolio that meets our customers' need for true 24-hour by 365-day access to their data, full integration with Microsoft Windows NT environments, and the full complement of network solutions and application software for today and well into the next millennium.

James E. Johnson
William A. Laing

Overview of the Spiralog File System

The OpenVMS Alpha environment requires a file system that supports its full 64-bit capabilities. The Spiralog file system was developed to increase the capabilities of Digital's Files-11 file system for OpenVMS. It incorporates ideas from a log-structured file system and an ordered write-back model. The Spiralog file system provides improvements in data availability, scaling of the amount of storage easily managed, support for very large volume sizes, support for applications that are either write-operation or file-system-operation intensive, and support for heterogeneous file system client types. The Spiralog technology, which matches or exceeds the reliability and device independence of the Files-11 system, was then integrated into the OpenVMS operating system.

Digital's Spiralog product is a log-structured, clusterwide file system with integrated, on-line backup and restore capability and support for multiple file system personalities. It incorporates a number of recent ideas from the research community, including the log-structured file system (LFS) from the Sprite file system and the ordered write back from the Echo file system.[1,2]

The Spiralog file system is fully integrated into the OpenVMS operating system, providing compatibility with the current OpenVMS file system, Files-11. It supports a coherent, clusterwide write-behind cache and provides high-performance, on-line backup and per-file and per-volume restore functions.

In this paper, we first discuss the evolution of file systems and the requirements for many of the basic designs in the Spiralog file system. Next we describe the overall architecture of the Spiralog file system, identifying its major components and outlining their designs. Then we discuss the project's results: what worked well and what did not work so well. Finally, we present some conclusions and ideas for future work.

Some of the major components, i.e., the backup and restore facility, the LFS server, and OpenVMS integration, are described in greater detail in companion papers in this issue.[3–5]

The Evolution of File Systems

File systems have existed throughout much of the history of computing. The need for libraries or services that help to manage the collection of data on long-term storage devices was recognized many years ago. The early support libraries have evolved into the file systems of today. During their evolution, they have responded to the industry's improved hardware capabilities and to users' increased expectations. Hardware has continued to decrease in price and improve in its price/performance ratio. Consequently, ever larger amounts of data are stored and manipulated by users in ever more sophisticated ways. As more and more data are stored on-line, the need to access that data 24 hours a day, 365 days a year has also escalated.

Significant improvements to file systems have been made in the following areas:

- Directory structures to ease locating data
- Device independence of data access through the file system
- Accessibility of the data to users on other systems
- Availability of the data, despite either planned or unplanned service outages
- Reliability of the stored data and the performance of the data access

Requirements of the OpenVMS File System

Since 1977, the OpenVMS operating system has offered a stable, robust file system known as Files-11. This file system is considered to be very successful in the areas of reliability and device independence. Recent customer feedback, however, indicated that the areas of data availability, scaling of the amount of storage easily managed, support for very large volume sizes, and support for heterogeneous file system client types were in need of improvement.

The Spiralog project was initiated in response to customers' needs. We designed the Spiralog file system to match or somewhat exceed the Files-11 system in its reliability and device independence. The focus of the Spiralog project was on those areas that were due for improvement, notably:

- Data availability, especially during planned operations, such as backup.

  If the storage device needs to be taken off-line to perform a backup, even at a very high backup rate of 20 megabytes per second (MB/s), almost 14 hours are needed to back up 1 terabyte. This length of service outage is clearly unacceptable. More typical backup rates of 1 to 2 MB/s can take several days, which, of course, is not acceptable.
- Greatly increased scaling in total amount of on-line storage, without greatly increasing the cost to manage that storage.

  For example, 1 terabyte of disk storage currently costs approximately $250,000, which is well within the budget of many large computing centers. However, the cost in staff and time to manage such amounts of storage can be many times that of the storage.[6] The cost of storage continues to fall, while the cost of managing it continues to rise.
- Effective scaling as more processing and storage resources become available.

  For example, OpenVMS Cluster systems allow processing power and storage capacity to be added incrementally. It is crucial that the software support-ing the file system scale as the processing power, bandwidth to storage, and storage capacity increase.
- Improved performance for applications that are either write-operation or file-system-operation intensive.

  As file system caches in main memory have increased in capacity, data reads and file system read operations have become satisfied more and more from the cache. At the same time, many applications write large amounts of data or create and manipulate large numbers of files. The use of redundant arrays of inexpensive disks (RAID) storage has increased the available bandwidth for data writes and file system writes. Most file system operations, on the other hand, are small writes and are spread across the disk at random, often negating the benefits of RAID storage.
- Improved ability to transparently access the stored data across several dissimilar client types.

  Computing environments have become increasingly heterogeneous. Different client systems, such as the Windows or the UNIX operating system, store their files on and share their files with server systems such as the OpenVMS server. It has become necessary to support the syntax and semantics of several different file system personalities on a common file server.

These needs were central to many design decisions we made for the Spiralog file system.

The members of the Spiralog project evaluated much of the ongoing work in file systems, databases, and storage architectures. RAID storage makes high bandwidth available to disk storage, but it requires large writes to be effective. Databases have exploited logs and the grouping of writes together to minimize the number of disk I/Os and disk seeks required. Databases and transaction systems have also exploited the technique of copying the tail of the log to effect backups or data replication. The Sprite project at Berkeley had brought together a log-structured file system and RAID storage to good effect.[1]

By drawing from the above ideas, particularly the insight of how a log structure could support on-line, high-performance backup, we began our development effort. We designed and built a distributed file system that made extensive use of the processor and memory near the application and used log-structured storage in the server.

Spiralog File System Design

The main execution stack of the Spiralog file system consists of three distinct layers. Figure 1 shows the overall structure. At the top, nearest the user, is the file

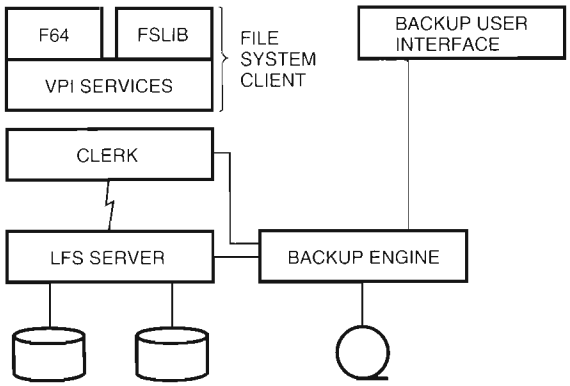


Figure 1
Spiralog Structure Overview

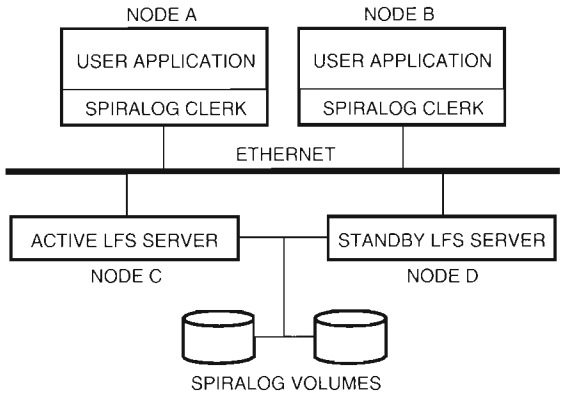


Figure 2
Spiralog Cluster Configuration

system client layer. It consists of a number of file system personalities and the underlying personality-independent services, which we call the VPI.

Two file system personalities dominate the Spiralog design. The F64 personality is an emulation of the Files-11 file system. The file system library (FSLIB) personality is an implementation of Microsoft's New Technology Advanced Server (NTAS) file services for use by the PATHWORKS for OpenVMS file server.

The next layer, present on all systems, is the clerk layer. It supports a distributed cache and ordered write back to the LFS server, giving single-system semantics in a cluster configuration.

The LFS server, the third layer, is present on all designated server systems. This component is responsible for maintaining the on-disk log structure; it includes the cleaner, and it is accessed by multiple clerks. Disks can be connected to more than one LFS server, but they are served only by one LFS server at a time. Transparent failover, from the point of view of the file system client layer, is achieved by cooperation between the clerks and the surviving LFS servers.

The backup engine is present on a system with an active LFS server. It uses the LFS server to access the on-disk data, and it interfaces to the clerk to ensure that the backup or restore operations are consistent with the clerk's cache.

Figure 2 shows a typical Spiralog cluster configuration. In this cluster, the clerks on nodes A and B are accessing the Spiralog volumes. Normally, they use the LFS server on node C to access their data. If node C should fail, the LFS server on node D would immediately provide access to the volumes. The clerks on nodes A and B would use the LFS server on node D, retrying all their outstanding operations. Neither user application would detect any failure. Once node C had recovered, it would become the standby LFS server.

File System Client Design

The file system client is responsible for the traditional file system functions. This layer provides files, directories, access arbitration, and file naming rules. It also provides the services that the user calls to access the file system.

VPI Services Layer The VPI layer provides an underlying primitive file system interface, based on the UNIX VFS switch. The VPI layer has two overall goals:

1. To support multiple file system personalities
2. To effectively scale to very large volumes of data and very large numbers of files

To meet the first goal, the VPI layer provides

- File names of 256 Unicode characters, with no reserved characters
- No restriction on directory depth
- Up to 255 sparse data streams per file, each with 64-bit addressing
- Attributes with 255 Unicode character names, containing values of up to 1,024 bytes
- Files and directories that are freely shared among file system personality modules

To meet the second goal, the VPI layer provides

- File identifiers stored as 64-bit integers
- Directories through a B-tree, rather than a simple linear structure, for $\log(n)$ file name lookup time

The VPI layer is only a base for file system personalities. Therefore it requires that such personalities are trusted components of the operating system. Moreover, it requires them to implement file access security (although there is a convention for storing access control list information) and to perform all necessary cleanup when a process or image terminates.

F64 File System Personality As previously stated, the Spiralog product includes two file system personalities, F64 and FSLIB. The F64 personality provides a service that emulates the Files-11 file system.[5] Its functions, services, available file attributes, and execution behaviors are similar to those in the Files-11 file system. Minor differences are isolated into areas that receive little use from most applications.

For instance, the Spiralog file system supports the various Files-11 queued I/O ($QIO) parameters for returning file attribute information, because they are used implicitly or explicitly by most user applications. On the other hand, the Files-11 method of reading the file header information directly through a file called INDEXF.SYS is not commonly used by applications and is not supported.

The F64 file system personality demonstrates that the VPI layer contains sufficient flexibility to support a complex file system interface. In a number of cases, however, several VPI calls are needed to implement a single, complex Files-11 operation. For instance, to do a file open operation, the F64 personality performs the tasks listed below. The items that end with (VPI) are tasks that use VPI service calls to complete.

- Access the file's parent directory (VPI)
- Read the directory's file attributes (VPI)
- Verify authorization to read the directory
- Loop, searching for the file name, by
  - Reading some directory entries (VPI)
  - Searching the directory buffer for the file name
  - Exiting the loop, if the match is found
- Access the target file (VPI)
- Read the file's attributes (VPI)
- Audit the file open attempt

FSLIB File System Personality The FSLIB file system personality is a specialized file system to support the PATHWORKS for OpenVMS file server. Its two major goals are to support the file names, attributes, and behaviors found in Microsoft's NTAS file access protocols, and to provide low run-time cost for processing NTAS file system requests.

The PATHWORKS server implements a file service for personal computer (PC) clients layered on top of the Files-11 file system services. When NTAS service behaviors or attributes do not match those of Files-11, the PATHWORKS server has to emulate them. This can lead to checking security access permissions twice, mapping file names, and emulating file attributes.

Many of these problems can be avoided if the VPI interface is used directly. For instance, because the FSLIB personality does not layer on top of a Files-11 personality, security access checks do not need to be performed twice. Furthermore, in a straightforward design, there is no need to map across different file naming or attribute rules. For reasons we describe later, in the VPI Results section, we chose not to pursue this design to its conclusion.

Clerk Design

The clerks are responsible for managing the caches, determining the order of writes out of the cache to the LFS server, and maintaining cache coherency within a cluster. The caches are write behind in a manner that preserves the order of dependent operations.

The clerk-server protocol controls the transfer of data to and from stable storage. Data can be sent as a multiblock atomic write, and operations that change multiple data items such as a file rename can be made atomically. If a server fails during a request, the clerk treats the request as if it were lost and retries the request.

The clerk-server protocol is idempotent. Idempotent operations can be applied repeatedly with no effects other than the desired one. Thus, after any number of server failures or server failovers, it is always safe to reissue an operation. Clerk-to-server write operations always leave the file system state consistent.

The clerk-clerk protocol protects the user data and file system metadata cached by the clerks. Cache coherency information, rather than data, is passed directly between clerks.

The file system caches are kept in the clerks. Multiple clerks can have copies of stabilized data, i.e., data that has been written to the server with the write acknowledged. Only one clerk can have unstabilized, volatile data. Data is exchanged between clerks by stabilizing it. When a clerk needs to write a block of data to the server from its cache, it uses a token interface that is layered on the clerk-clerk protocol.

The writes from the cache to the server are deferred as long as possible within the constraints of the cache protocol and the dependency guarantees.

Dirty data remains in the cache as long as 30 seconds. During that time, overwrites are combined within the constraints of the dependency guarantees. Furthermore, operations that are known to offset one another, such as freeing a file identifier and allocating a file identifier, are fully combined within the cache.

Eventually, some trigger causes the dirty data to be written to the server. At this point, several writes are grouped together. Write operations to adjacent, or overlapping, file locations are combined to form a smaller number of larger writes. The resulting write operations are then grouped into messages to the LFS server.

The clerks perform write behind for four reasons:

- To spread the I/O load over time
- To remove occluded data, which can result from repeated overwrites of a data block, from being transferred to the server

- To avoid writing data that is quickly deleted such as temporary files
- To combine multiple small writes into larger transfers

The clerks order dependent writes from the cache to the server; consequently, other clerks never see "impossible" states, and related writes never overtake each other. For instance, the deletion of a file cannot happen before a rename that was previously issued to the same file. Related data writes are caused by a partial overwrite, or an explicit linking of operations passed into the clerk by the VPI layer, or an implicit linking due to the clerk-clerk coherency protocol.

The ordering between writes is kept as a directed graph. As the clerks traverse these graphs, they issue the writes in order or collapse the graph when writes can be safely combined or eliminated.

LFS Server Design

The Spiralog file system uses a log-structured, on-disk format for storing data within a volume, yet presents a traditional, update-in-place file system to its users. Recently, log-structured file systems, such as Sprite, have been an area of active research.[1]

Within the LFS server, support is provided for the log-structured, on-disk format and for mapping that format to an update-in-place model. Specifically, this component is responsible for

- Mapping the incoming read and write operations from their simple address space to positions in an open-ended log
- Mapping the open-ended log onto a finite amount of disk space
- Reclaiming disk space by cleaning (garbage collecting) the obsolete (overwritten) sections of the log

Figure 3 shows the various mapping layers in the Spiralog file system, including those handled by the LFS server.

Incoming read and write operations are based on a single, large address space. Initially, the LFS server transforms the address ranges in the incoming operations into equivalent address ranges in an open-ended log. This log supports a very large, write-once address space.

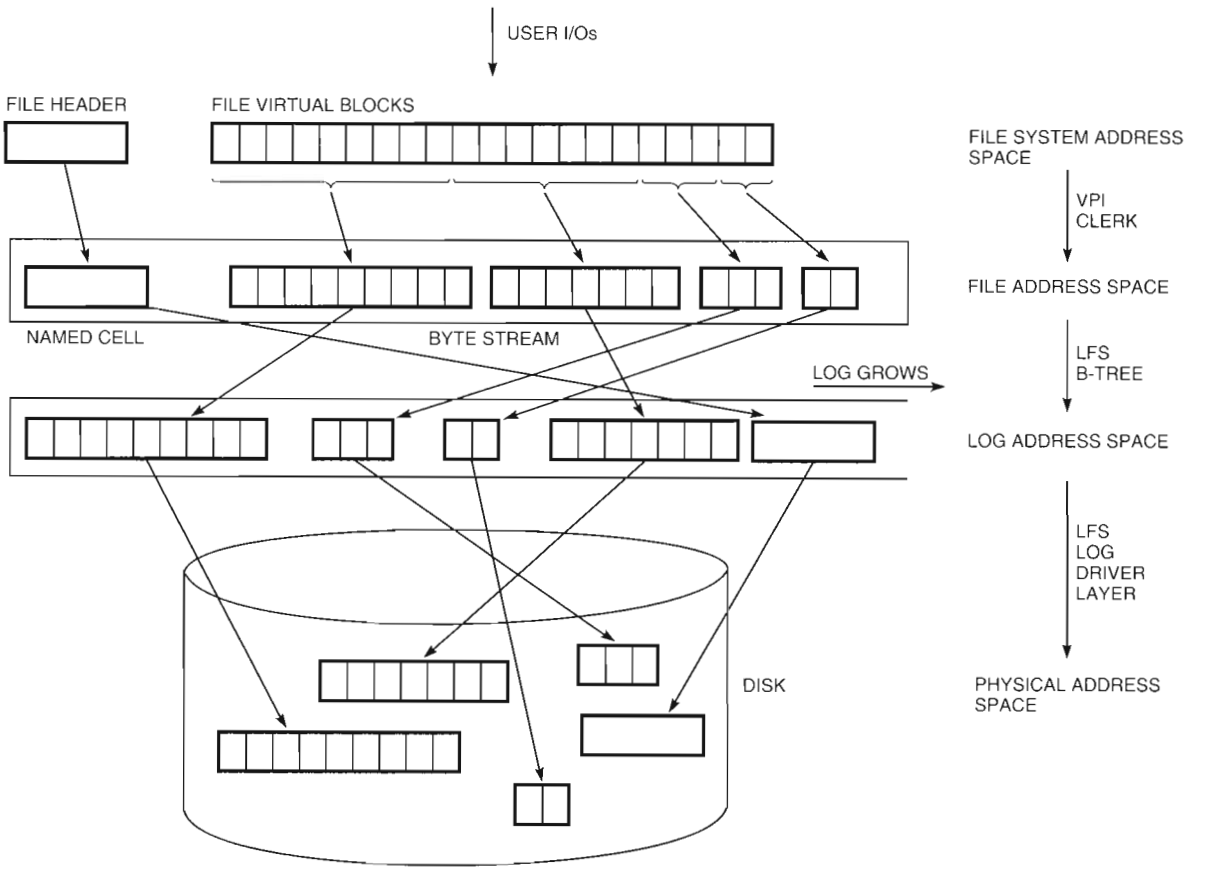


Figure 3
Spiralog Address Mapping

A read operation looks up its location in the open-ended log and proceeds. On the other hand, a write operation makes obsolete its current address range and appends its new value to the tail of the log.

In turn, locations in the open-ended log are then mapped into locations on the (finite-sized) disk. This additional mapping allows disk blocks to be reused once their original contents have become obsolete.

Physically, the log is divided into log segments, each of which is 256 kilobytes (KB) in length. The log segment is used as the transfer unit for the backup engine. It is also used by the cleaner for reclaiming obsolete log space.

More information about the LFS server can be found in this issue.[4]

On-line Backup Design

The design goals for the backup engine arose from higher storage management costs and greater data availability needs. Investigations with a number of customers revealed their requirements for a backup engine:

- Consistent save operations without stopping any applications or locking out data modifications
- Very fast save operations
- Both full and incremental save operations
- Restores of a full volume and of individual files

Our response to these needs influenced many decisions concerning the Spiralog file system design. The need for a high-performance, on-line backup led to a search for an on-disk structure that could support it. Again, we chose the log-structured design as the most suitable one.

A log-structured organization allows the backup facility to easily demarcate snapshots of the file system at any point in time, simply by marking a point in the log. Such a mark represents a version of the file system and prevents disk blocks that compose that version from being cleaned. In turn, this allows the backup to run against a low level of the file system, that of the logical log, and therefore to operate close to the spiral transfer rate of the underlying disk.

The difference between a partial, or incremental, and a full save operation is only the starting point in the log. An incremental save need not copy data back to the beginning of the log. Therefore, both incremental and full save operations transfer data at very high speed.

By implementing these features in the Spiralog file system, we fulfilled our customers' requirements for high-performance, on-line backup save operations. We also met their needs for per-file and per-volume restores and an ongoing need for simplicity and reduction in operating costs.

To provide per-file restore capabilities, the backup utility and the LFS server ensure that the appropriate file header information is stored during the save operation. The saved file system data, including file headers, log mapping information, and user data, are stored in a file known as a *saveset*. Each saveset, regardless of the number of tapes it requires, represents a single save operation.

To reduce the complexity of file restore operations, the Spiralog file system provides an off-line saveset merge feature. This allows the system manager to merge several savesets, either full or incremental, to form a new, single saveset. With this feature, system managers can have a workable backup save plan that never calls for an on-line full backup, thus further reducing the load on their production systems. Also, this feature can be used to ensure that file restore operations can be accomplished with a small, bounded set of savesets.

The Spiralog backup facility is described in detail in this issue.[3]

Project Results

The Spiralog file system contains a number of innovations in the areas of on-line backup, log-structured storage, clusterwide ordered write-behind caching, and multiple-file-system client support.

The use of log structuring as an on-disk format is very effective in supporting high-performance, on-line backup. The Spiralog file system retains the previously documented benefits of LFS, such as fast write performance that scales with the disk size and throughput that increases as large read caches are used to offset disk reads.[1]

It should also be noted that the Files-11 file system sets a high standard for data reliability and robustness. The Spiralog technology met this challenge very well: as a result of the idempotent protocol, the cluster failover design, and the recover capability of the log, we encountered few data reliability problems during development.

In any large, complex project, many technical decisions are necessary to convert research technology into a product. In this section, we discuss why certain decisions were made during the development of the Spiralog subsystems.

VPI Results

The VPI file system was generally successful in providing the underlying support necessary for different file system personalities. We found that it was possible to construct a set of primitive operations that could be used to build complex, user-level, file system operations.

By using these primitives, the Spiralog project members were able to successfully design two distinctly different personality modules. Neither was a functional superset of the other, and neither was layered on top of the other. However, there was an important second-order problem.

The FSLIB file system personality did not have a full mapping to the Files-11 file system. As a consequence, file management was rather difficult, because all the data management tools on the OpenVMS operating system assumed compliance with a Files-11, rather than a VPI, file system.

This problem led to the decision not to proceed with the original design for the FSLIB personality in version 1.0 of Spiralog. Instead, we developed an FSLIB file system personality that was fully compatible with the F64 personality, even when that compatibility forced us to accept an additional execution cost.

We also found an execution cost to the primitive VPI operations. Generally, there was little overhead for data read and write operations. However, for operations such as opening a file, searching for a file name, and deleting a file, we found too high an overhead from the number of calls into the VPI services and the resulting calls into the cache manager. We called this the "fan-out" problem: one high-level operation would turn into several VPI operations, each of which would turn into several cache manager calls. Table 1 gives the details of the fan-out problem.

We believe that it would be worthwhile to provide slightly more complex VPI services in order to combine calls that always appear in the same sequence.

Table 1
Call Fan-out by Level

Operation	F64 Calls	VPI Calls	Clerk Calls	Revised Clerk Calls
Create file	4	18	29	24
Open file	1	6	18	14
Read block	1	1	3	3
Write block	2	4	7	6
Close file	1	4	13	10

Clerk Results

The clerk met a number of our design goals. First, the use of idempotent operations allowed failover to standby LFS servers to occur with no loss of service to the file system clients, and with little additional complexity within the clerk.

Second, the ordered write behind proved to be effective at ordering dependent, metadata file system operations, thus supporting the ability to construct complex file system operations out of simpler elements.

Third, the clerk was able to manage large physical caches. It is very effective at making use of unused pages when the memory demand from the OpenVMS operating system is low, and at quickly shrinking the cache when memory demands increase. Although certain parameters can be used to limit the size of a clerk's cache, the caches are normally self-tuning.

Fourth, the clerks reduce the number of operations and messages sent to the LFS server, with a subsequent reduction to the number of messages and operations waiting to be processed. For the COPY command, the number of operations sent to the server was typically reduced by a factor of 3. By using transient files with lifetimes of fewer than 30 seconds, we saw a reduction of operations by a factor of 100 or more, as long as the temporary file fit into the clerk's cache.

In general, the code complexity and CPU path length within the clerk were greater than we had originally planned, and they will need further work. Two aspects of the services offered by the clerk compounded the cost in CPU path length. First, the clerk has a simple interface that supports reads and writes into a single, large address space only. This interface requires a number of clerk operations for a number of the VPI calls, further expanding the call fan-out issues. Second, a concurrency control model allows the clerk to unilaterally drop locks. This requires the VPI layer to revalidate its internal state with each call.

Either a change to the clerk and VPI service interfaces to support notification of lock invalidation, or a change to the concurrency control model to disallow locks that could be unilaterally invalidated, would reduce the number of calls made. We believe such changes would produce the results given in the last column of Table 1.

LFS Server Results

The LFS server provides a highly available, robust file system server. Under heavy write loads, it provides the ability to group together multiple requests and reduce the number of disk I/Os. In a cluster configuration, it supports failover to a standby server.

In normal operation, the cleaner was successful in minimizing overhead, typically adding only a few percent to the elapsed time. The cleaner operated in a lazy manner, cleaning only when there was an immediate shortage of space. The cleaner operations were further lessened by the tendency for normal file overwrites to free up recently filled log segments for reuse.

Although this produced a cleaner that operated with little overhead, it also brought about two unusual interactions with the backup facility. In the first place, the log often contains a number of obsolete areas that

are eligible for cleaning but have not yet been processed. These obsolete areas are also saved by the backup engine. Although they have no effect on the logical state of the log, they do require the backup engine to move more data to backup storage than might otherwise be necessary.

Second, the design initially prohibited the cleaner from running against a log with snapshots. Consequently, the cleaner was disabled during a save operation, which had the following effects: (1) The amount of available free space in the log was artificially depressed during a backup. (2) Once the backup was finished, the activated cleaner would discover that a great number of log segments were now eligible for cleaning. As a result, the cleaner underwent a sudden surge in cleaning activity soon after the backup had completed.

We addressed this problem by reducing the area of the log that was off-limits to the cleaner to only the part that the backup engine would read. This limited snapshot window allowed more segments to remain eligible for cleaning, thus greatly alleviating the shortage of cleanable space during the backup and eliminating the postbackup cleaning surge. For an 8-gigabyte time-sharing volume, this change typically reduced the period of high cleaner activity from 40 seconds to less than one-half of a second.

We have not yet experimented with different cleaner algorithms. More work needs to be done in this area to see if the cleaning efficiency, cost, and interactions with backup can be improved.

The current mapping transformation from the incoming operation address space to locations in the open-ended log is more expensive in CPU time than we would like. More work is needed to optimize the code path.

Finally, the LFS server is generally successful at providing the appearance of a traditional, update-in-place file system. However, as the unused space in a volume nears zero, the ability to behave with semantics that meet users' expectations in a log-structured file system proved more difficult than we had anticipated and required significant effort to correct.

The LFS server is described in much more detail in this issue.[4]

Backup Performance Results

We took a new approach to the backup design in the Spiralog system, resulting in a very fast and very low impact backup that can be used to create consistent copies of the file system while applications are actively modifying data. We achieved this degree of success without compromising such functionality as incremental backup or fast, selective restore.

The performance improvements of the Spiralog save operation are particularly noticeable with the large numbers of transient or active files that are typically found on user volumes or on mail server volumes. In the following tables, we compare the Spiralog and the file-based Files-11 backup operations on a DEC 3000 Model 500 workstation with a 260-MB volume, containing 21,682 files in 401 directories and a TZ877 tape.

Table 2 gives the results of two save operations, which are the average of five operations. Although its saveset size is somewhat larger, the Spiralog save operation completes nearly twice as fast as the Files-11 save operation.

Table 3 gives the results from restoring a single file to the target volume. In this case, the Spiralog file restore operation executes more than three times as fast as the Files-11 system.

The performance advantage of the Spiralog backup and restore facility increases further for large, multitape savesets. In these cases, the Spiralog system is able to omit tapes that are not needed for the file restore; the Files-11 system does not have this capability.

Observations and Conclusions

Overall, we believe that the significant innovation and real success of the Spiralog project was the integration of high-performance, on-line backup with the log-structured file system model. The Spiralog file system delivers an on-line backup engine that can run near device speeds, with little impact on concurrently running applications. Many file operations are significantly faster in elapsed time as a result of the reduction in I/Os due to the cache and the grouping of write operations. Although the code paths for a number of operations are longer than we had planned, their

Table 2
Performance Comparison of the Backup Save Operation

File System	Elapsed Time (Minutes:Seconds)	Saveset Size (MB)	Throughput (MB/s)
Spiralog	05:20	339	1.05
Files-11	10:14	297	0.48

Table 3
Performance Comparison of the Individual File Restore Operation

File System	Elapsed Time (Minutes:Seconds)
Spiralog	01:06
Files-11	03:35

length is mitigated by continuing improvements in processor performance.

We learned a great deal during the Spiralog project and made the following observations:

- Volume full semantics and fine-tuning the cleaner were more complex than we anticipated and will require future refinement.
- A heavily layered architecture extends the CPU path length and the fan-out of procedure calls. We focused too much attention on reducing I/Os and not enough attention on reducing the resource usage of some critical code paths.
- Although elegant, the memory abstraction for the interface to the cache was not as good a fit to file system operations as we had expected. Furthermore, a block abstraction for the data space would have been more suitable.

In summary, the project team delivered a new file system for the OpenVMS operating system. The Spiralog file system offers single-system semantics in a cluster, is compatible with the current OpenVMS file system, and supports on-line backup.

Future Work

During the Spiralog version 1.0 project, we pursued a number of new technologies and found four areas that warrant future work:

- Support is needed from storage and file-management tools for multiple, dissimilar file system personalities.
- The cleaner represents another area of ongoing innovation and complex dynamics. We believe a better understanding of these dynamics is needed, and design alternatives should be studied.
- The on-line backup engine, coupled with the log-structured file system technology, offers many areas for potential development. For instance, one area for investigation is continuous backup operation, either to a local backup device or to a remote replica.
- Finally, we do not believe the higher-than-expected code path length is intrinsic to the basic file system design. We expect to be working on this resource usage in the near future.

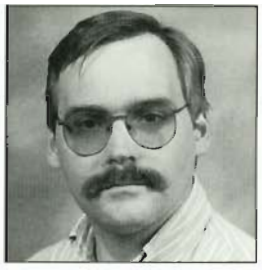
Acknowledgments

We would like to take this opportunity to thank the many individuals who contributed to the Spiralog project. Don Harbert and Rich Marcello, OpenVMS vice presidents, supported this work over the lifetime of the project. Dan Doherty and Jack Fallon, the OpenVMS managers in Livingston, Scotland, had day-to-day management responsibility. Cathy Foley kept the project moving toward the goal of shipping. Janis Horn and Clare Wells, the product managers who helped us understand our customers' needs, were eloquent in explaining our project and goal to others. Near the end of the project, Yehia Beyh and Paul Mosteika gave us valuable testing support, without which the product would certainly be less stable than it is today. Finally, and not least, we would like to acknowledge the members of the development team: Alasdair Baird, Stuart Bayley, Rob Burke, Ian Compton, Chris Davies, Stuart Deans, Alan Dewar, Campbell Fraser, Russ Green, Peter Hancock, Steve Hirst, Jim Hogg, Mark Howell, Mike Johnson, Robert Landau, Douglas McLaggan, Rudi Martin, Conor Morrison, Julian Palmer, Judy Parsons, Ian Pattison, Alan Paxton, Nancy Phan, Kevin Porter, Alan Potter, Russell Robles, Chris Whitaker, and Rod Widdowson.

References

1. M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log Structured File System," *ACM Transactions on Computer Systems,* vol. 10, no. 1 (February 1992): 26–52.
2. T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart, "A Coherent Distributed File Cache with Directory Write-behind," Digital Systems Research Center, Research Report 103 (June 1993).
3. R. Green, A. Baird, and J. Davies, "Designing a Fast, On-line Backup System for a Log-structured File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 32–45.
4. C. Whitaker, J. Bayley, and R. Widdowson, "Design of the Server for the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 15–31.
5. M. Howell and J. Palmer, "Integrating the Spiralog File System into the OpenVMS Operating System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 46–56.
6. R. Wrenn, "Why the Real Cost of Storage is More Than $1/MB," presented at U.S. DECUS Symposium, St. Louis, Mo., June 3–6, 1996.

Biographies



James E. Johnson
Jim Johnson, a consulting software engineer, has been working for Digital since 1984. He was a member of the OpenVMS Engineering Group, where he contributed in several areas, including RMS, transaction processing services, the port of OpenVMS to the Alpha architecture, file systems, and system management. Jim recently joined the Internet Software Business Unit and is working on the application of X.500 directory services. Jim holds two patents on transaction commit protocol optimizations and maintains a keen interest in this area.

William A. Laing
Bill Laing, a corporate consulting engineer, is the technical director of the Internet Software Business Unit. Bill joined Digital in 1981; he worked in the United States for five years before transferring to Europe. During his career at Digital, Bill has worked on VMS systems performance analysis, VAXcluster design and development, operating systems development, and transaction processing. He was the technical director of OpenVMS engineering, the technical director for engineering in Europe, and most recently was focusing on software in the Technology and Architecture Group of the Computer Systems Division. Prior to joining Digital, Bill held research and teaching posts in operating systems at the University of Edinburgh, where he worked on the EMAS operating system. He was also part of the start-up of European Silicon Structures (ES2), an ambitious pan-European company. He holds undergraduate and postgraduate degrees in computer science from the University of Edinburgh.

Design of the Server for the Spiralog File System

Christopher Whitaker
J. Stuart Bayley
Rod D. W. Widdowson

The Spiralog file system uses a log-structured, on-disk format inspired by the Sprite log-structured file system (LFS) from the University of California, Berkeley. Log-structured file systems promise a number of performance and functional benefits over conventional, update-in-place file systems, such as the Files-11 file system developed for the OpenVMS operating system or the FFS file system on the UNIX operating system. The Spiralog server combines log-structured technology with more traditional B-tree technology to provide a general server abstraction. The B-tree mapping mechanism uses write-ahead logging to give stability and recoverability guarantees. By combining write-ahead logging with a log-structured, on-disk format, the Spiralog server merges file system data and recovery log records into a single, sequential write stream.

The goal of the Spiralog file system project team was to produce a high-performance, highly available, and robust file system with a high-performance, on-line backup capability for the OpenVMS Alpha operating system. The server component of the Spiralog file system is responsible for reading data from and writing data to persistent storage. It must provide fast write performance, scalability, and rapid recovery from system failures. In addition, the server must allow an on-line backup utility to copy a consistent snapshot of the file system to another location, while allowing normal file system operations to continue in parallel.

In this paper, we describe the log-structured file system (LFS) technology and its particular implementation in the Spiralog file system. We also describe the novel way in which the Spiralog server maps the log to provide a rich address space in which files and directories are constructed. Finally, we review some of the opportunities and challenges presented by the design we chose.

Background

All file systems must trade off performance against availability in different ways to provide the throughput required during normal operations and to protect data from corruption during system failures. Traditionally, file systems fall into two categories, careful write and check on recovery.

- Careful writing policies are designed to provide a fail-safe mechanism for the file system structures in the event of a system failure; however, they suffer from the need to serialize several I/Os during file system operations.
- Some file systems forego the need to serialize file system updates. After a system failure, however, they require a complete disk scan to reconstruct a consistent file system. This requirement becomes a problem as disk sizes increase.

Modern file systems such as Cedar, Episode, Microsoft's New Technology File System (NTFS), and Digital's POLYCENTER Advanced File System use logging to overcome the problems inherent in these two approaches.[1,2] Logging file system metadata removes the need to serialize I/Os and allows a simple

and bounded mechanism for reconstructing the file system after a failure. Researchers at the University of California, Berkeley, took this process one stage further and treated the whole disk as a single, sequential log where all file system modifications are appended to the tail of the log.[3]

Log-structured file system technology is particularly appropriate to the Spiralog file system, because it is designed as a clusterwide file system. The server must support a large number of file system clerks, each of which may be reading and writing data to the disk. The clerks use large write-back caches to reduce the need to read data from the server. The caches also allow the clerks to buffer write requests destined for the server. A log-structured design allows multiple concurrent writes to be grouped together into large, sequential I/Os to the disk. This I/O pattern reduces disk head movement during writing and allows the size of the writes to be matched to characteristics of the underlying disk. This is particularly beneficial for storage devices with redundant arrays of inexpensive disks (RAID).[4]

The use of a log-structured, on-disk format greatly simplifies the implementation of an on-line backup capability. Here, the challenge is to provide a consistent snapshot of the file system that can be copied to the backup media while normal operations continue to modify the file system. Because an LFS appends all data to the tail of a log, all data writes within the log are temporally ordered. A complete snapshot of the file system corresponds to the contents of the sequential log up to the point in time that the snapshot was created. By extension, an incremental backup corresponds to the section of the sequential log created since the last backup was taken. The Spiralog backup utility uses these features to provide a fast, on-line, full and incremental backup scheme.[5]

We have taken a number of features from the existing log-structured file system implementations, in particular, the idea of dividing the log into fixed-sized segments as the basis for space allocation and cleaning.[6] Fundamentally, however, existing log-structured file systems have been built by using the main body of an existing file system and layering on top of an underlying, log-structured container.[3,7] This design has been taken to the logical extreme with the implementation of a log-structured disk.[8] For the Spiralog file system, we have chosen to use the sequential log capability provided by the log-structured, on-disk format throughout the file system. The Spiralog server combines log-structured technology with more traditional B-tree technology to provide a general server abstraction. The B-tree mapping mechanism uses write-ahead logging to give stability and recoverability guarantees.[9] By combining write-ahead logging with a log-structured on-disk format, the Spiralog server merges file system data and recovery log records into a single, sequential write stream.

The Spiralog file system differs from existing log-structured implementations in a number of other important ways, in particular, the mechanisms that we have chosen to use for the cleaner. In subsequent sections of this paper, we compare these differences with existing implementations where appropriate.

Spiralog File System Server Architecture

The Spiralog file system employs a client-server architecture. Each node in the cluster that mounts a Spiralog volume runs a file system clerk. The term clerk is used in this paper to distinguish the client component of the file system from clients of the file system as a whole. Clerks implement all the file functions associated with maintaining the file system state with the exception of persistent storage of file system and user data. This latter responsibility falls on the Spiralog server. There is exactly one server for each volume, which must run on a node that has a direct connection to the disk containing the volume. This distribution of function, where the majority of file system processing takes place on the clerk, is similar to that of the Echo file system.[10] The reasons for choosing this architecture are described in more detail in the paper "Overview of the Spiralog File System," elsewhere in this issue.[11]

Spiralog clerks build files and directories in a structured address space called the file address space. This address space is internal to the file system and is only loosely related to that perceived by clients of the file system. The server provides an interface that allows the clerks to persistently map to file space addresses. Internally, the server uses a logically infinite log structure, built on top of a physical disk, to store the file system data and the structures necessary to locate the data. Figure 1 shows the relationship between the clerks and the server and the relationships among the major components within the server.

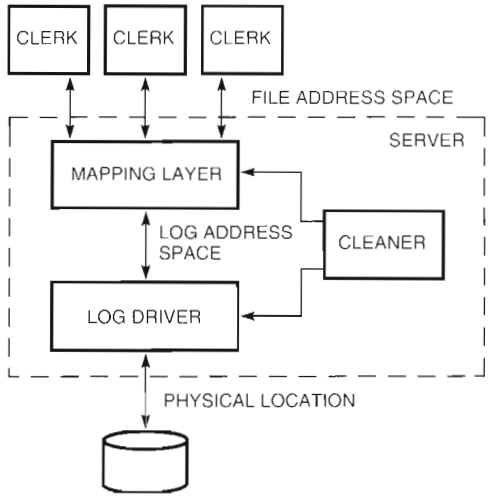


Figure 1
Server Architecture

The mapping layer is responsible for maintaining the mapping between the file address space used by the clerks to the address space of the log. The server directly supports the file address space so that it can make use of information about the relative performance sensitivity of parts of the address space that is implicit within its structure. Although this results in the mapping layer being relatively complex, it reduces the complexity of the clerks and aids performance. The mapping layer is the primary point of contact with the server. Here, read and write requests from clerks are received and translated into operations on the log address space.

The log driver (LD) creates the illusion of an infinite log on top of the physical disk. The LD transforms read and write requests from the mapping layer that are cast in terms of a location in the log address space into read and write requests to physical addresses on the underlying disk. Hiding the implementation of the log from the mapping layer allows the organization of the log to be altered transparently to the mapping layer. For example, parts of the log can be migrated to other physical devices without involving the mapping layer.

Although the log exported by the LD layer is conceptually infinite, disks have a finite size. The cleaner is responsible for garbage collecting or coalescing free space within the log.

Figure 2 shows the relationship between the various address spaces making up the Spiralog file system. In the next three sections, we examine each of the components of the server.

Mapping Layer

The mapping layer implements the mapping between the file address space used by the file system clerks and the log address space maintained by the LD. It exports an interface to the clerks that they use to read data from locations in the file address space, to write new data to the file address space, and to specify which previously written data is no longer required. The interface also allows clerks to group sets of dependent writes into units that succeed or fail as if they were a single write. In this section, we introduce the file address space and describe the data structure used to map it. Then we explain the method used to handle clerk requests to modify the address space.

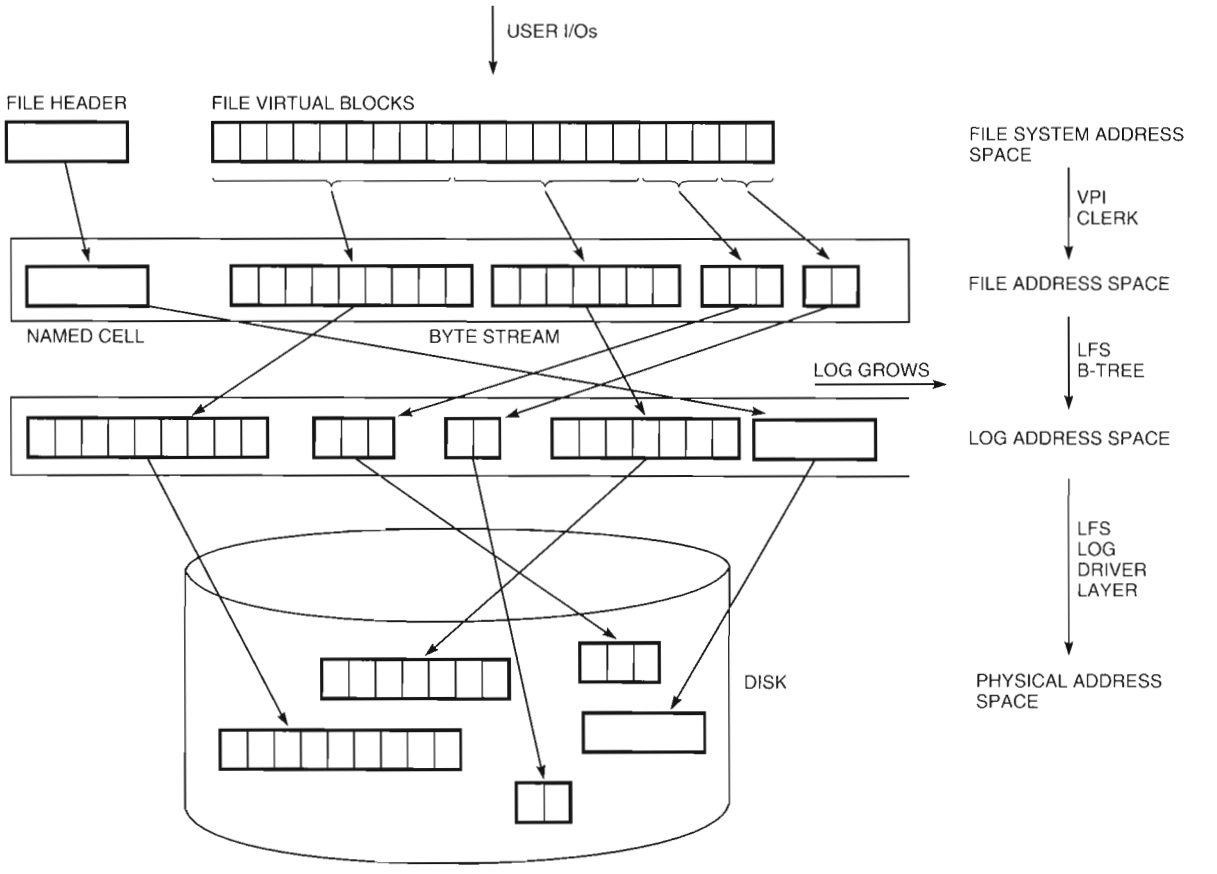


Figure 2
Address Translation

File Address Space

The file address space is a structured address space. At its highest level it is divided into objects, each of which has a numeric object identifier (OID). An object may have any number of named cells associated with it and up to $2^{16}-1$ streams. A named cell may contain a variable amount of data, but it is read and written as a single unit. A stream is a sequence of bytes that are addressed by their offset from the start of the stream, up to a maximum of $2^{64}-1$. Fundamentally, there are two forms of addresses defined by the file address space: Named addresses of the form

```
<OID, name>
```

specify an individual named cell within an object, and numeric addresses of the form

```
<OID, stream-id, stream-offset, length>
```

specify a sequence of *length* contiguous bytes in an individual stream belonging to an object.

The clerks use named cells and streams to build files and directories. In the Spiralog file system version 1.0, a file is represented by an object, a named cell containing its attributes, and a single stream that is used to store the file's data. A directory is represented by an object that contains a number of named cells. Each named cell represents a link in that directory and contains what a traditional file system refers to as a directory entry. Figure 3 shows how data files and directories are built from named cells and streams.

The mapping layer provides three principal operations for manipulating the file address space: read, write, and clear. The read operation allows a clerk to read the contents of a named cell, a contiguous range of bytes from a stream, or all the named cells for a particular object that fall into a specified search range. The write operation allows a clerk to write to a contiguous range of bytes in a stream or an individual named cell. The clear operation allows a clerk to remove a named cell or a number of bytes from an object.

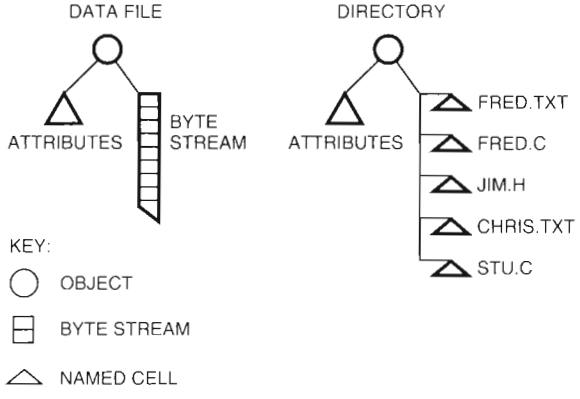


Figure 3
File System

Mapping the File Address Space

We looked at a variety of indexing structures for mapping the file address space onto the log address space.[1,12] We chose a derivative of the B-tree for the following reasons. For a uniform address space, B-trees provide predictable worst-case access times because the tree is balanced across all the keys it maps. A B-tree scales well as the number of keys mapped increases. In other words, as more keys are added, the B-tree grows in width and in depth. Deep B-trees carry an obvious performance penalty, particularly when the B-tree grows too large to be held in memory. As described above, directory entries, file attributes, and file data are all addresses, or keys, in the file address space. Treating these keys as equals and balancing the mapping B-tree across all these keys introduces the possibility that a single directory with many entries, or a file with many extents, may have an impact on the access times for all the files stored in the log.

To solve this problem, we limited the keys for an object to a single B-tree leaf node. With this restriction, several small files can be accommodated in a single leaf node. Files with a large number of extents (or large directories) are supported by allowing individual streams to be spawned into subtrees. The subtrees are balanced across the keys within the subtree. An object can never span more than a single leaf node of the main B-tree; therefore, nonleaf nodes of the main B-tree only need to contain OIDs. This allows the main B-tree to be very compact. Figure 4 shows the relationship between the main B-tree and its subtrees.

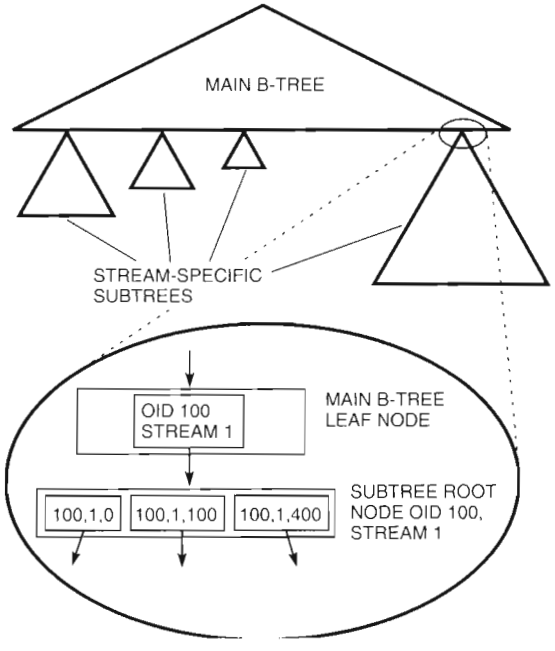


Figure 4
Mapping B-tree Structure

To reduce the time required to open a file, data for small extents and small named cells are stored directly in the leaf node that maps them. For larger extents (greater than one disk block in size in the current implementation), the data item is written into the log and a pointer to it is stored in the node. This pointer is an address in the log address space. Figure 5 illustrates how the B-tree maps a small file and a file with several large extents.

Processing Read Requests

The clerks submit read requests that may be for a sequence of bytes from a stream (reading a data from a file), a single named cell (reading a file's attributes), or a set of named cells (reading directory contents). To fulfill a given read request, the server must consult the B-tree to translate from the address in the file address space supplied by the clerk to the position in the log address space where the data is stored. The extents making up a stream are created when the file data is written. If an application writes 8 kilobytes (KB) of data in 1-KB chunks, the B-tree would contain 8 extents, one for each 1-KB write. The server may need to collect data from several different parts of the log address space to fulfill a single read request.

Read requests share access to the B-tree in much the same way as processes share access to the CPU of a multiprocessing computer system. Read requests arriving from clerks are placed in a first in first out (FIFO) work queue and are started in order of their arrival. All operations on the B-tree are performed by a single worker thread in each volume. This avoids the need for heavyweight locking on individual nodes in the B-tree, which significantly reduces the complexity of the tree manipulation algorithms and removes the potential for deadlocks on tree nodes. This reduction in complexity comes at the cost of the design not scaling with the number of processors in a symmetric multiprocessing (SMP) system. So far we have no evidence to show that this design decision represents a major performance limitation on the server.

The worker thread takes a request from the head of the work queue and traverses the B-tree until it reaches a leaf node that maps the address range of the read request. Upon reaching a leaf node, it may discover that the node contains

- Records that map part or all of the address of the read request to locations in the log, and/or
- Records that map part or all of the address of the read request to data stored directly in the node, and/or
- No records mapping part or all of the address of the read request

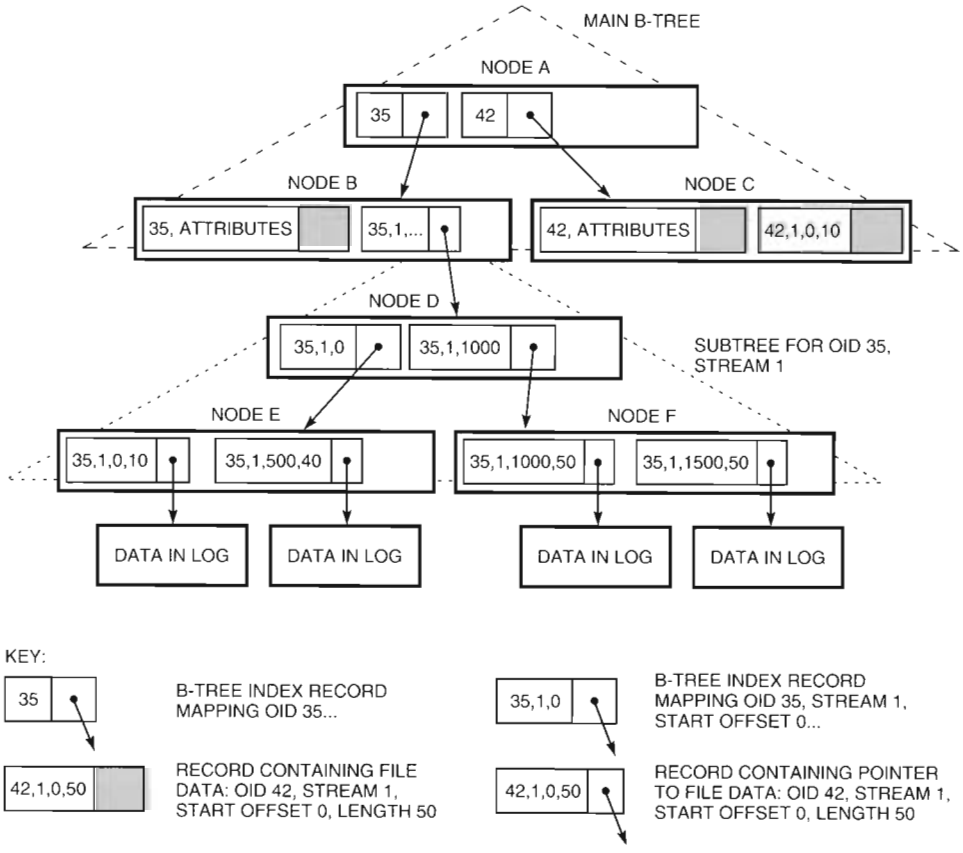


Figure 5
Mapping B-tree Detail

Data that is stored in the node is simply copied to the output buffer. When data is stored in the log, the worker thread issues requests to the LD to read the data into the output buffer. Once all the reads have been issued, the original request is placed on a pending queue until they complete; then the results are returned to the clerk. When no data is stored for all or part of the read request, the server zero-fills the corresponding part of the output buffer.

The process described above is complicated by the fact that the B-tree is itself stored in the log. The mapping layer contains a node cache that ensures that commonly referenced nodes are normally found in memory. When the worker thread needs to traverse through a tree node that is not in memory, it must arrange for the node to be read into the cache. The address of the node in the log is the value of the pointer to it from its parent node. The worker thread uses this to issue a request to the LD to read the node into a cache buffer. While the node read request is in progress, the original clerk operation is placed on a pending queue and the worker thread proceeds to the next request on the work queue. When the node is resident in memory, the pending read request is placed back on the work queue to be restarted. In this way, multiple read requests can be in progress at any given time.

Processing Write Requests

Write requests received by the server arrive in groups consisting of a number of data items corresponding to updates to noncontiguous addresses in the file address space. Each group must be written as a single failure atomic unit, which means that all the parts of the write request must be made stable or none of them must become stable. Such groups of writes are called wunners and are used by the clerk to encapsulate complex file system operations.[11]

Before the server can complete a wunner, that is, before an acknowledgment can be sent back to the clerk indicating that the wunner was successful, the server must make two guarantees:

1. All parts of the wunner are stably stored in the log so that the entire wunner is persistent in the event of a system failure.
2. All data items described by the wunner are visible to subsequent read requests.

The wunner is made persistent by writing each data item to the log. Each data item is tagged with a log record that identifies its corresponding file space address. This allows the data to be recovered in the event of a system failure. All individual writes are made as part of a single compound atomic operation (CAO). This method is provided by the LD layer to bracket a set of writes that must be recovered as an atomic unit. Once all the writes for the wunner have been issued to the log, the mapping layer instructs the LD layer to end (or commit) the CAO.

The wunner can be made visible to subsequent read operations by updating the B-tree to reflect the location of the new data. Unfortunately, this would cause writes to incur a significant latency since updating the B-tree involves traversing the B-tree and potentially reading B-tree nodes into memory from the log. Instead, the server completes a write operation before the B-tree is updated. By doing this, however, it must take additional steps to ensure that the data is visible to subsequent read requests.

Before completing the wunner, the mapping layer queues the B-tree updates resulting from writing the wunner to the same FIFO work queue as read requests. All items are queued atomically, that is, no other read or write operation can be interleaved with the individual wunner updates. In this way, the ordering between the writes making up the wunner and subsequent read or write operations is maintained. Work cannot begin on a subsequent read request until work has started on the B-tree updates ahead of it in the queue.

Once the B-tree updates have been queued to the server work queue and the mapping layer has been notified that the CAO for the writes has committed, both of the guarantees that the server gives on write completion hold. The data is persistent, and the writes are visible to subsequent operations; therefore, the server can send an acknowledgment back to the clerk.

Updating the B-tree

The worker thread processes a B-tree update request in much the same way as a read request. The update request traverses the B-tree until either it reaches the node that maps the appropriate part of the file address space, or it fails to find a node in memory.

Once the leaf node is reached, it is updated to point at the location of the data in the log (if the data is to be stored directly in the node, the data is copied into the node). The node is now dirty in memory and must be written to the log at some point. Rather than writing the node immediately, the mapping layer writes a log record describing the change, locks the node into the cache, and places a flush operation for the node to the mapping layer's flush queue. The flush operation describes the location of the node in the tree and records the need to write it to the log at some point in the future.

If, on its way to the leaf node, the write operation reaches a node that is not in memory, the worker thread arranges for it to be read from the log and the write operation is placed on a pending queue as with a read operation. Because the write has been acknowledged to the clerk, the new data must be visible to subsequent read operations even though the B-tree has not been updated fully. This is achieved by attaching an in-memory record of the update to the node that is

being read. If a read operation reaches the node with records of stalled updates, it must check whether any of these records contains data that should be returned. The record contains either a pointer to the data in the log or the actual data itself. If a read operation finds a record that can satisfy all or part of the request, the read request uses the information in the record to fetch the data. This preserves the guarantee that the clerk must see all data for which the write request has been acknowledged.

Once the node is read in from the log, the stalled updates are restarted. Each update removes its log record from the node and recommences traversing the B-tree from that point.

Writing B-tree Nodes to the Log

Writing nodes consumes bandwidth to the disk that might otherwise be used for writing or reading user data, so the server tries to avoid doing so until absolutely necessary. Two conditions make it necessary to begin writing nodes:

1. There are a large number of dirty nodes in the cache.
2. A checkpoint is in progress.

In the first condition, most of the memory available to the server has been given over to nodes that are locked in memory and waiting to be written to the log. Read and update operations begin to back up, waiting for available memory to store nodes. In the second condition, the LD has requested a checkpoint in order to bound recovery time (see the section Checkpointing later in this paper).

When either of these conditions occurs, the mapping layer switches into flush mode, during which it only writes nodes, until the condition is changed. In flush mode, the worker thread processes flush operations from the mapping layer's flush queue in depth order, that is, starting with the nodes furthest from the root of the B-tree. For each flush operation, it traverses the B-tree until it finds the target node and its parent. The target node is identified by the keys it maps and its level. The level of a node is its distance from the leaf of the B-tree (or subtree). Unlike its depth, which is its distance from the root of the B-tree, a node's level does not change as the B-tree grows and shrinks.

Once it has reached its destination, the flush operation writes out the target node and updates the parent with the new log address. The modifications made to the parent node by the flush operation are analogous to those made to a leaf node by an update operation. In this way, a modification to a leaf node eventually works its way to the root of the B-tree, causing each node in its path to be rewritten to the log over time. Writing dirty nodes only when necessary and then in deepest first order minimizes the number of nodes written to the log and increases the average number of changes that are reflected in each node written.

Log Driver

The log driver is responsible for creating the illusion of a semi-infinite sequential log on top of a physical disk. The entire history of the file system is recorded in the updates made to the log, but only those parts of the log that describe its current or live state need to be persistently stored on the disk. As files are overwritten or deleted, the parts of the log that contain the previous contents become obsolete.

Segments and the Segment Array

To make the management of free space more straightforward, the log is divided into sections called segments. In the Spiralog file system, segments are 256 KB. Segments in the log are identified by their segment identifier (SEGID). SEGIDs increase monotonically and are never reused. Segments in the log that contain live data are mapped to physical, segment-sized locations or slots on the disk that are identified by their segment number (SEGNUM) as shown in Figure 6. The mapping between SEGID and SEGNUM is maintained by the segment array. The segment array also tracks which parts of each mapped segment contain live data. This information is used by the cleaner.

The LD interface layer contains a segment switch that allows segments to be fetched from a location other than the disk.[13] The backup function on the Spiralog file system uses this mechanism to restore files contained in segments held on backup media. Figure 7 shows the LD layer.

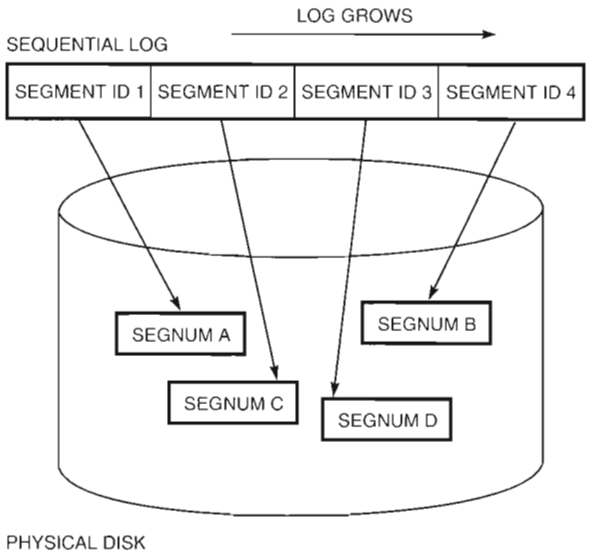


Figure 6
Mapping the Log onto the Disk

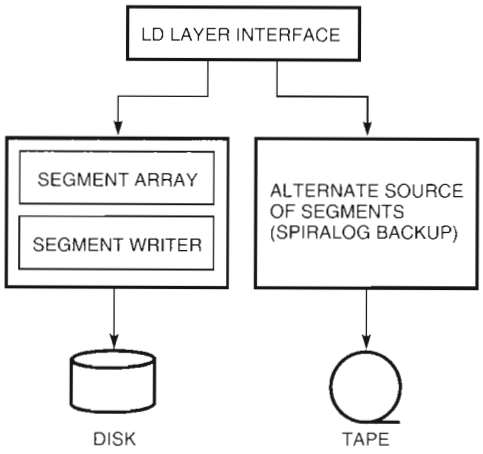


Figure 7
Subcomponents of the LD Layer

The Segment Writer

The segment writer is responsible for all I/Os to the log. It groups together writes it receives from the mapping layer into large, sequential I/Os where possible. This increases write throughput, but at the potential cost of increasing the latency of individual operations when the disk is lightly loaded.

As shown in Figure 8, the segment writer is responsible for the internal organization of segments written to the disk. Segments are divided into two sections, a data area and a much smaller commit record area. Writing a piece of data requires two operations to the segment at the tail of the log. First the data item is written to the data area of the segment. Once this I/O has completed successfully, a record describing that data is written to the commit record area. Only when the write to the commit record area is complete can the original request be considered stable.

The need for two writes to disk (potentially, with a rotational delay between) to commit a single data write is clearly a disadvantage. Normally, however, the segment writer receives a set of related writes from the mapping layer which are tagged as part of a single CAO. Since the mapping layer is interested in the completion of the whole CAO and not the writes within it, the segment writer is able to buffer additions to the commit records area in memory and then write them with a single I/O. Under a normal write load, this reduces the number of I/Os for a single data write to very close to one.

The boundary between the commit record area and the data area is fixed. Inevitably, this wastes space in either the commit record area or data area when the other fills. Choosing a size for the commit record area that minimizes this waste requires some care. After analysis of segments that had been subjected to a typical OpenVMS load, we chose 24 KB as the value for the commit record area.

This segment organization permits the segment writer to have complete control over the contents of the commit record area, which allows the segment writer to accomplish two important recovery tasks:

- Detect the end of the log
- Detect multiblock write failure

When physical segments are reused to extend the log, they are not scrubbed and their commit record areas contain stale (but comprehensible) records. The recovery manager must distinguish between records belonging to the current and the previous incarnation of the physical slot. To achieve this, the segment writer writes a sequence number into a specific byte in every block written to the commit record area. The original contents of the "stolen" bytes are stored within the record being written. The sequence number used for

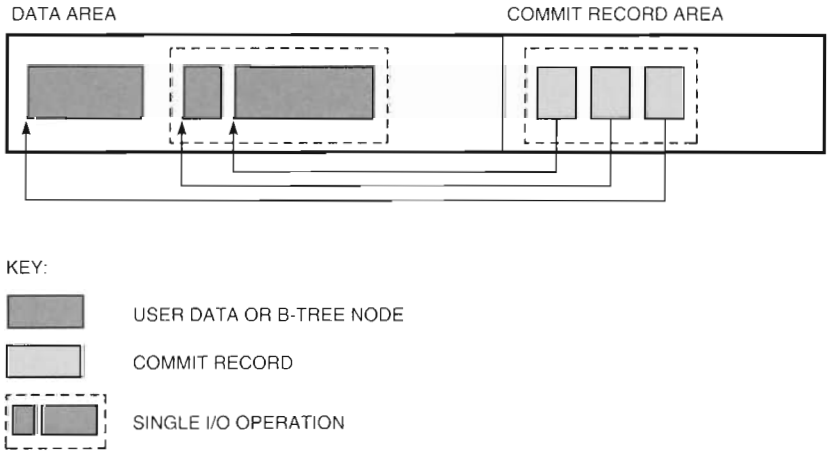


Figure 8
Organization of a Segment

a segment is an attribute of the physical slot that is assigned to it. The sequence number for a physical slot is incremented each time the slot is reused, allowing the recovery manager to detect blocks that do not belong to the segment stored in the physical slot. The cost of resubstituting the stolen bytes is incurred only during recovery and cleaning, because this is the only time that the commit record area is read.

In hindsight, the partitioning of segments into data and commit areas was probably a mistake. A layout that intermingles the data and commit records and that allows them to be written in one I/O would offer better latency at low throughput. If combined with careful writing, command tag queuing, and other optimizations becoming more prevalent in disk hardware and controllers, such an on-disk structure could offer significant improvements in latency and throughput.

Cleaner

The cleaner's job is to turn free space in segments in the log into empty, unassigned physical slots that can be used to extend the log. Areas of free space appear in segments when the corresponding data decays; that is, it is either deleted or replaced.

The cleaner rewrites the live data contained in partially full segments. Essentially, the cleaner forces the segments to decay completely. If the rate at which data is written to the log matches the rate at which it is deleted, segments eventually become empty of their own accord. When the log is full (fullness depends on the distribution of file longevity), it is necessary to proactively clean segments. As the cleaner continues to consume more of the disk bandwidth, performance can be expected to decline. Our design goal was that performance should be maintained up to a point at which the log is 85 percent full. Beyond this, it was acceptable for performance to degrade significantly.

Bytes Die Young

Recently written data is more likely to decay than old data.[14,15] Segments that were written a short time ago are likely to decay further, after which the cost of cleaning them will be less. In our design, the cleaner selects candidate segments that were written some time ago and are more likely to have undergone this initial decay.

Mixing data cleaned from older segments with data from the current stream of new writes is likely to produce a segment that will need to be cleaned again once the new data has undergone its initial decay. To avoid mixing cleaned data and data from the current write stream, the cleaner builds its output segments separately and then passes them to the LD to be threaded in at the tail of the log. This has two important benefits:

- The recovery information in the output segment is minimal, consisting only of the self-describing tags on the data. As a result, the cleaner is unlikely to waste space in the data area by virtue of having filled the commit record area.
- By constructing the output segment off-line, the cleaner has as much time as it needs to look for data chunks that best fill the segment.

Remapping the Output Segment

The data items contained in the cleaner's output segment receive new addresses. The cleaner informs the mapping layer of the change of location by submitting B-tree update operation for each piece of data it copied. The mapping layer handles this update operation in much the same way as it would a normal overwrite. This update does have one special property: the cleaner writes are conditional. In other words, the mapping layer will update the B-tree to point to the copy created by the cleaner as long as no change has been made to the data since the cleaner took its copy. This allows the cleaner to work asynchronously to file system activity and avoids any locking protocol between the cleaner and any other part of the Spiralog file system.

To avoid modifying the mapping layer directly, the cleaner does not copy B-tree nodes to its output segment. Instead, it requests the mapping layer to flush the nodes that occur in its input segments (i.e., rewrite them to the tail of the log). This also avoids wasting space in the cleaner output segment on nodes that map data in the cleaner's input segments. These nodes are guaranteed to decay as soon as the cleaner's B-tree updates are processed.

Figure 9 shows how the cleaner constructs an output segment from a number of input segments. The cleaner keeps selecting input segments until either the output segment is full, or there are no more input segments. Figure 9 also shows the set of operations that are generated by the cleaner. In this example, the output segment is filled with the contents of two full segments and part of a third segment. This will cause the third input segment to decay still further, and the remaining data and B-tree nodes will be cleaned when that segment is selected to create another output segment.

Cleaner Policies

A set of heuristics governs the cleaner's operation. One of our fundamental design decisions was to separate the cleaner policies from the mechanisms that implement them.

When to clean?

Our design explicitly avoids cleaning until it is required. This design appears to be a good match for

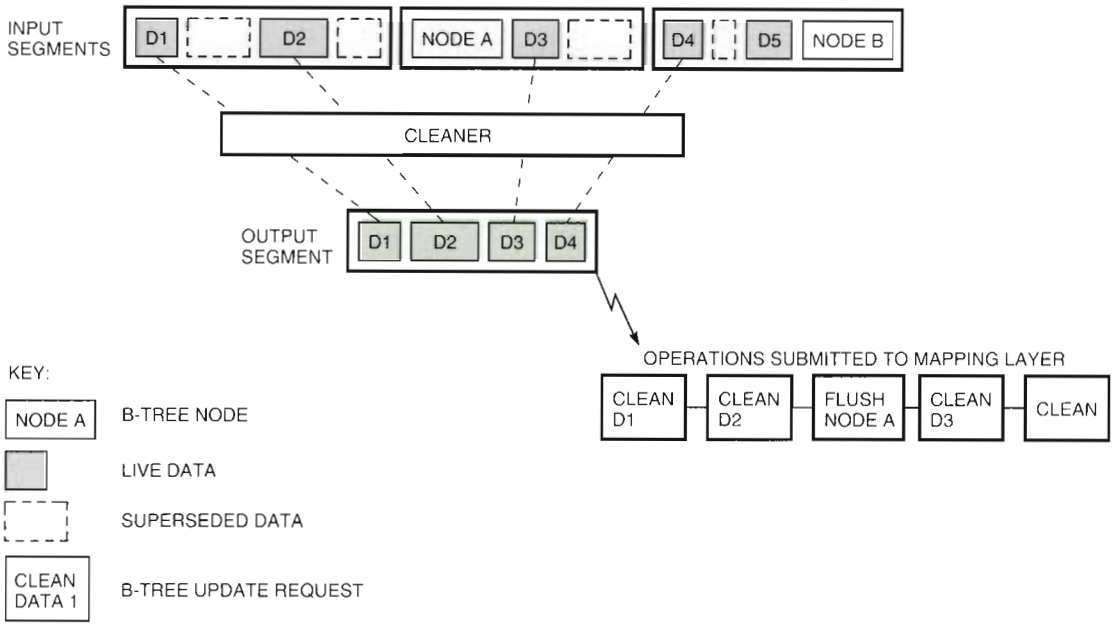


Figure 9
Cleaner Operation

a workload on the OpenVMS system. On our time-sharing system, the cleaner was entirely inactive for the first three months of 1996; although segments were used and reused repeatedly, they always decayed entirely to empty of their own accord. The trade-off in avoiding cleaning is that although performance is improved (no cleaner activity), the size of the full savesnaps created by backup is increased. This is because backup copies whole segments, regardless of how much live data they contain.

When the cleaner is not running, the live data in the volume tends to be distributed across a large number of partially full segments. To avoid this problem, we have added a control to allow the system manager to manually start and stop the cleaner. Forcing the cleaner to run before performing a full backup compacts the live data in the log and reduces the size of the savesnap.

In normal operation, the cleaner will start cleaning when the number of free segments available to extend the log falls below a fixed threshold (300 in the current implementation). In making this calculation, the cleaner takes into account the amount of space in the log that will be consumed by writing data currently held in the clerks' write-behind caches. Thus, accepting data into the cache causes the cleaner to "clear the way" for the subsequent write request from the clerk.

When the cleaner starts, it is possible that the amount of live data in the log is approaching the capacity of the underlying disk, so the cleaner may find nothing to do. It is more likely, however, that there will be free space it can reclaim. Because the cleaner works by forcing the data in its input segments to decay by rewriting, it is important that it begins work while free segments are available. Delaying the decision to start cleaning could result in the cleaner being unable to proceed.

A fixed number was chosen for the cleaning threshold rather than one based on the size of the disk. The size of the disk does not affect the urgency of cleaning at any particular point in time. A more effective indicator of urgency is the time taken for the disk to fill at the maximum rate of writing. Writing to the log at 10 MB per second will use 300 segments in about 8 seconds. With hindsight, we realize that a threshold based on a measurement of the speed of the disk might have been a more appropriate choice.

Input Segment Selection

The cleaner divides segments into four distinct groups:

1. Empty. These segments contain no live data and are available to the LD to extend the log.
2. Noncleanable. These segments are not candidates for cleaning for one of two reasons:
   - The segment contains information that would be required by the recovery manager in the event of a system failure. Segments in this group are always close to the tail of the log and therefore likely to undergo further decay, making them poor candidates for cleaning.
   - The segment is part of a snapshot.[5] The snapshot represents a reference to the segment, so it cannot be reused even though it may no longer contain live data.

3. Preferred noncleanable. These segments have recently experienced some natural decay. The supposition is that they may decay further in the near future and so are not good candidates for cleaning.
4. Cleanable. These segments have not decayed for some time. Their stability makes them good candidates for cleaning.

The transitions between the groups are illustrated in Figure 10. It should be noted that the cleaner itself does not have to execute to transfer segments into the empty state.

The cleaner's job is to fill output segments, not to empty input segments. Once it has been started, the cleaner works to entirely fill one segment. When that segment has been filled, it is threaded into the log; if appropriate, the cleaner will then repeat the process with a new output segment and a new set of input segments. The cleaner will commit a partially full output segment only under circumstances of extreme resource depletion.

The cleaner fills the output segment by copying chunks of data forward from segments taken from the cleanable group. The members of this group are held on a list sorted in order of emptiness. Thus, the first cleaner cycle will always cause the greatest number of segments to decay. As the output segment fills, the smallest chunk of data in the segment at the head of the cleanable list may be larger than the space left in the output segment. In this case, the cleaner performs a limited search down the cleanable list for segments containing a suitable chunk. The required information is kept in memory, so this is a reasonably cheap operation. As each input segment is processed, the cleaner temporarily removes it from the cleanable list. This allows the mapping layer to process the operations the cleaner submitted to it and thereby cause decay to occur before the cleaner again considers the segment as a candidate for cleaning. As the volume fills, the ratio between the number of segments in the cleanable and preferred noncleanable groups is adjusted so that the size of the preferred noncleanable group is reduced and segments are inserted into the cleanable list. If appropriate, a segment in the cleanable list that experiences decay will be moved to the preferred noncleanable list. The preferred noncleanable list is kept in order of least recently decayed. Hence, as it is emptied, the segments that are least likely to experience further decay are moved to the cleanable group.

Recovery

The goal of recovery of any file system is to rebuild the file system state after a system failure. This section describes how the server reconstructs state, both in memory and in the log. It then describes checkpointing, the mechanism by which the server bounds the amount of time it takes to recover the file system state.

Recovery Process

In normal operation, a single update to the server can be viewed as several stages:

1. The user data is written to the log. It is tagged with a self-identifying record that describes its position in the file address space. A B-tree update operation is generated that drives stage 2 of the update process.

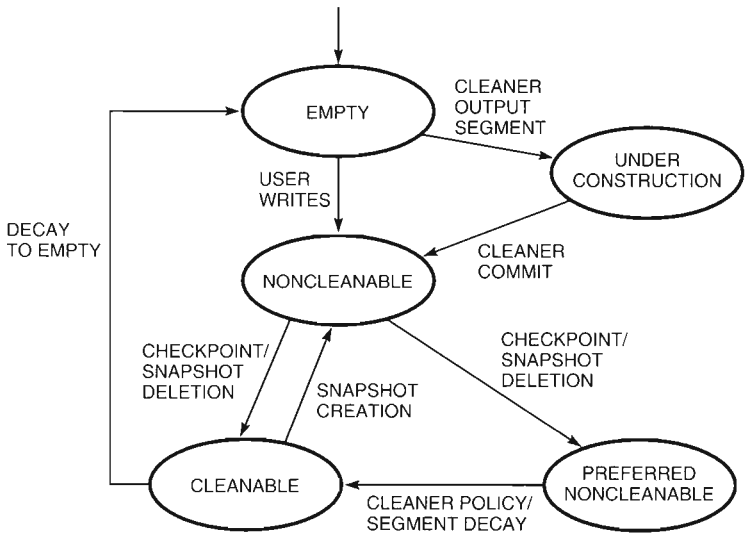


Figure 10
Segment States

2. The leaf nodes of the B-tree are modified in memory, and corresponding change records are written to the log to reflect the position of the new data. A flush operation is generated and queued and then starts stage 3.
3. The B-tree is written out level by level until the root node has been rewritten. As one node is written to the log, the parent of that node must be modified, and a corresponding change record is written to the log. As a parent node is changed, a further flush operation is generated for the parent node and so on up to the root node.

Stage 2 of this process, logging changes to the leaf nodes of the B-tree, is actually redundant. The self-identifying tags that are written with the user data are sufficient to act as change records for the leaf nodes of the B-tree. When we started to design the server, we chose a simple implementation based on physiological write-ahead logging.[9] As time progressed, we moved more toward operational logging.[9] The records written in stage 2 are a holdover from the earlier implementation, which we may remove in a future release of the Spiralog file system.

At each stage of the process, a change record is written to the log and an in-memory operation is generated to drive the update through the next stage. In effect, the change record describes the set of changes made to an in-memory copy of a node and an in-memory operation associated with that change.

Figure 11 shows the log and the in-memory work queue at each stage of a write request. The B-tree describing the file system state consists of three nodes: A, B, and C. A wunner, consisting of a single data write is accepted by the server. The write request requires that both leaf nodes A and B are modified. Stage 1 starts with an empty log and a write request for Data 1.

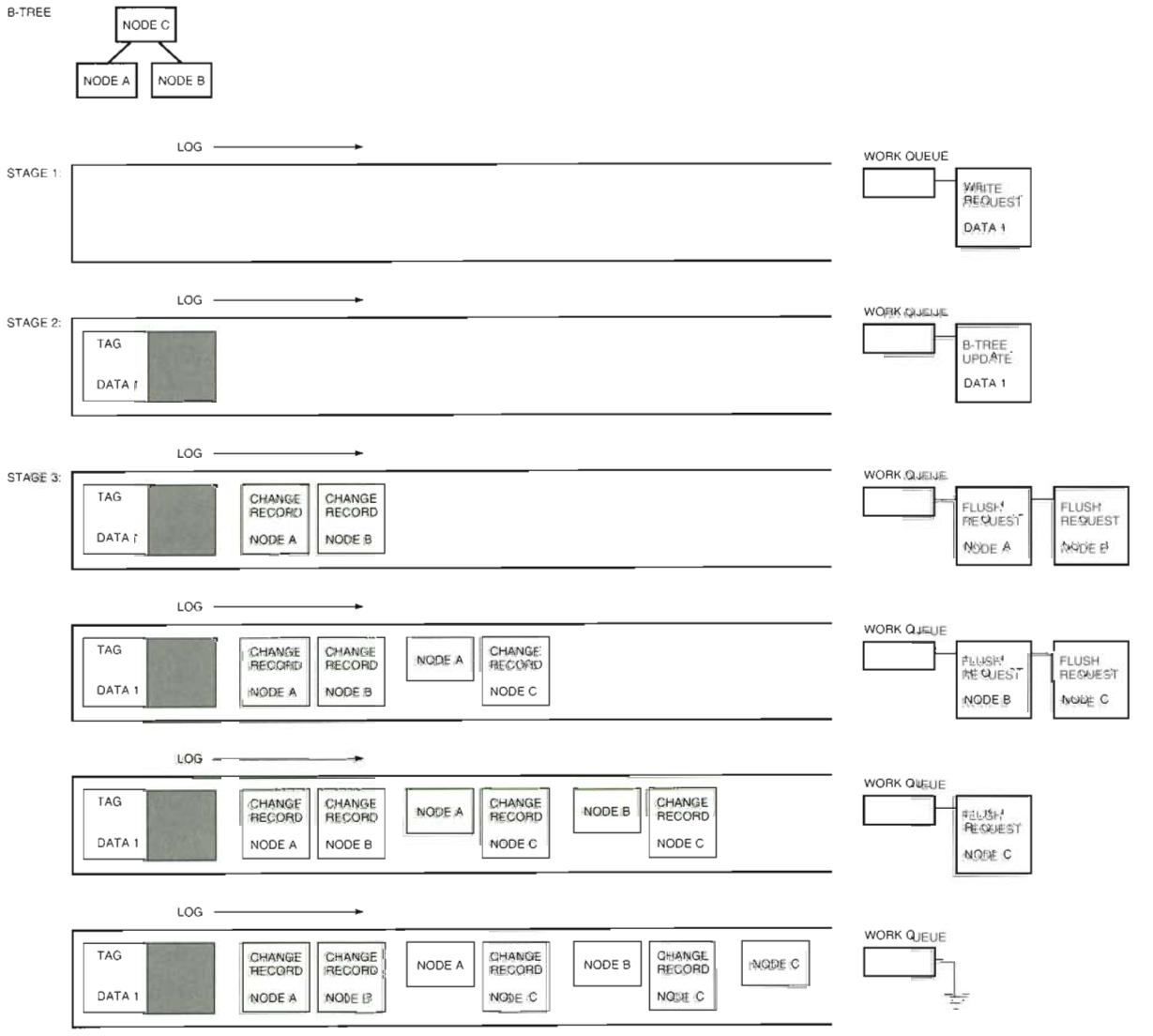


Figure 11
Stages of a Write Request

After a system failure, the server's goal is to reconstruct the file system state to the point of the last write that was written to the log at the time of the crash. This recovery process involves rebuilding, in memory, those B-tree nodes that were dirty and generating any operations that were outstanding when the system failed. The outstanding operations can be scheduled in the normal way to make the changes that they represent permanent, thus avoiding the need to recover them in the event of a future system failure. The recovery process itself does not write to the log.

The mapping layer work queue and the flush lists are rebuilt, and the nodes are fetched into memory by reading the sequential log from the recovery start position (see the section Checkpointing) to the end of the log in a single pass.

The B-tree update operations are regenerated using the self-identifying tag that was written with each piece of data. When the recovery process finds a node, a copy of the node is stored in memory. As log records for node changes are read, they are attached to the nodes in memory and a flush operation is generated for the node. If a log record is read for a node that has not yet been seen, the log record is attached to a placeholder node that is marked as not-yet-seen. The recovery process does not perform reads to fetch in nodes that are not part of the recovery scan. Changes to B-tree nodes are a consequence of operations that happened earlier in the log; therefore, a B-tree node log record has the effect of committing a prior modification. Recovery uses this fact to throw away update operations that have been committed; they no longer need to be applied.

Figure 12 shows a log with change records and B-tree nodes along with the in-memory state of the B-tree node cache and the operations that are regenerated. In this example, change record 1 for node A is superseded or committed by the new version of node A (node A′). The new copy of node C (node C′) supersedes change records 3 and 5. This example also shows the effect of finding a log record without seeing a copy of the node during recovery. The log record for node B is attached to an in-memory version of the node that is marked as not-yet-seen. The data record with self-identifying tag Data 1 generates a B-tree update record that is placed on the work queue for processing. As a final pass, the recovery process generates the set of flush operations that was outstanding when the system failed. The set of flush requests is defined as the set of nodes in the B-tree node cache that has log records attached when the recovery scan is complete. In this case, flush operations for nodes A′ and B are generated.

The server guarantees that a node is never written to the log with uncommitted changes, which means that we only need to log redo records.[9,16] In addition, when we see a node during the recovery scan, any log records that are attached to the previous version of the node in memory can be discarded.

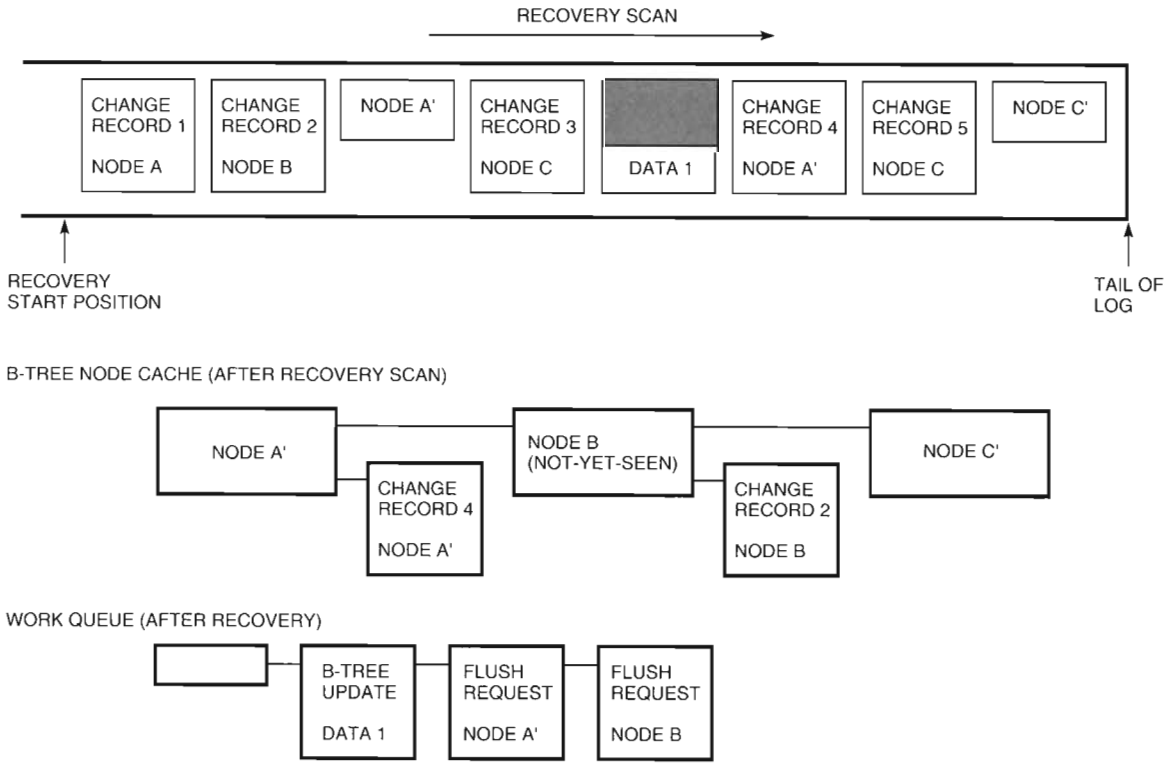


Figure 12
Recovering a Log

Operations generated during recovery are posted to the work queues as they would be in normal running. Normal operation is not allowed to begin until the recovery pass has completed; however, when recovery reaches the end of the log, the server is able to service operations from clerks. Thus new requests from the clerk can be serviced, potentially in parallel with the operations that were generated by the recovery process.

Log records are not applied to nodes during recovery for a number of reasons:

- Less processing time is needed to scan the log and therefore the server can start servicing new user requests sooner.
- Recovery will not have seen copies of all nodes for which it has log records. To apply the log records, the B-tree node must be read from the log. This would result in random read requests during the sequential scan of the log, and again would result in a longer period before user requests could be serviced.
- There may be a copy of the node later in the recovery scan. This would make the additional I/O operation redundant.

Checkpointing

As we have shown, recovering an LFS log is implemented by a single-pass sequential scan of all records in the log from the recovery start position to the tail of the log. This section defines a recovery start position and describes how it can be moved forward to reduce the amount of log that has to be scanned to recover the file system state.

To reconstruct the in-memory state when a system crashed, recovery must see something in the log that represents each operation or change of state that was represented in memory but not yet made stable. This means that at time t, the recovery start position is defined as a point in the log after which all operations that are not stably stored have a log record associated with them. Operations obtain the association by scanning the log sequentially from the beginning to the end. The recovery position then becomes the start of the log, which has two important problems:

1. In the worst case, it would be necessary to sequentially scan the entire log to perform recovery. For large disks, a sequential read of the entire log consumes a great deal of time.
2. Recovery must process every log record written between the recovery start position and the end of the log. As a consequence, segments between the start of recovery and the end of the log cannot be cleaned and reused.

To restrict the amount of time to recover the log and to allow segments to be released by cleaning, the recovery position must be moved forward from time to time, so that it is always close to the tail of the log.

Under any workload, a number of outstanding operations are at various stages of completion. In other words, there is no point in the log when all activity has ceased. To overcome this problem, we use a fuzzy checkpoint scheme.[9] In version 1.0 of the Spiralog file system, the server initiates a new checkpoint when 20 MB of data has been written since the previous checkpoint started. The process cannot yet move the recovery position forward in the log to the start of the new checkpoint, because some outstanding operations may have priority. The mapping layer keeps track of the operations that were started before the checkpoint was initiated. When the last of these operations has moved to the next stage (as defined by the recovery process), the mapping layer declares that the checkpoint is complete. Only then can the recovery position be moved forward to the point in the log where the checkpoint was started.

With this scheme, the server does not need to write all the nodes in all paths in the B-tree between a dirty node and the root node. All that is required in practice is to write those nodes that have flush operations queued for them at the time that the checkpoint is started. Flushing these nodes causes change records to be written for their parent nodes after the start of the checkpoint. As the recovery scan proceeds from the start of the last completed checkpoint, it is able to regenerate the flush operation on the parent nodes from these change records.

We chose to base the checkpoint interval on the amount of data written to the log rather than on the amount of time to recover the log. We felt that this would be an accurate measure of how long it would take to recover a particular log. In operation, we find this works well on logs that experience a reasonable write load; however, for logs that predominantly service read requests, the recovery time tends toward the limit. In these cases, it may be more appropriate to add timer-based checkpoints.

Managing Free Space

A traditional, update-in-place file system overwrites superseded data by writing to the same physical location on disk. If, for example, a single block is continually overwritten by a file system client, no extra disk space is required to store the block. In contrast, a log-structured file system appends all modifications to the file system to the tail of the log. Every update to a single block requires log space, not only for the data, but also for the log records and B-tree nodes required to make the B-tree consistent. Although old copies of the data and B-tree nodes are marked as no longer live, this free space is not immediately available for reuse; it must be reclaimed by the cleaner. The goal is to ensure that there is sufficient space in the log to write the

parts of the B-tree that are needed to make the file system structures consistent. This means that we can never have dirty B-tree nodes in memory that cannot be flushed to the log.

The server must carefully manage the amount of free space in the log. It must provide two guarantees:

1. A write will be accepted by the server only if there is sufficient free space in the log to hold the data and rewrite the mapping B-tree to describe it. This guarantee must hold regardless of how much space the cleaner may subsequently reclaim.
2. At the higher levels of the file system, if an I/O operation is accepted, even if that operation is stored in the write-behind cache, the data will be written to the log. This guarantee holds except in the event of a system failure.

The server provides these guarantees using the same mechanism. As shown in Figure 13, the free space and the reserved space in the log are modeled using an escrow function.[17]

The total number of blocks that contain live, valid data is maintained as the used space. When a write operation is received, the server calculates the amount of space in the log that is required to complete the write and update the B-tree, based on the size of the write and the current topology of the B-tree. The calculation is generous because the B-tree is a dynamic structure and the outcome of a single update has unpredictable effects on it. Each clerk reserves space for dirty data that it has stored in the write-behind cache using the same mechanism.

To accept an operation and provide the required guarantees, the server checks the current state of the escrow function. If the guaranteed free space is sufficient, the server accepts the operation. As operations proceed, reserved space is converted to used space as writes are performed. A single write operation may affect several leaf nodes. As it becomes clear how the B-tree is changing, we can convert any unrequired reserved space back to guaranteed free space.

If the cost of an operation exceeds the free space irrespective of how the reserved space is converted, the operation cannot be guaranteed to complete; therefore it is rejected. On the other hand, if the cost of the operation is greater than the guaranteed free space (yet it may fit in the log, depending on the outcome of the outstanding operations), the server enters a "maybe" state. For the server to leave the maybe state and return definitive results, the escrow cost function must be collapsed. This removes any uncertainty by decreasing the reserved space figure, potentially to zero. Operations and unused clerk reservations are drained so that reserved space is converted to either used space or guaranteed free space.

This mechanism provides a fuzzy measure of how much space is available in the log. When it is clear that operations can succeed, they are allowed to continue. If success is doubtful, the operation is held until a definitive yes or no result can be determined. This scheme of free space management is similar to the method described in reference 7.

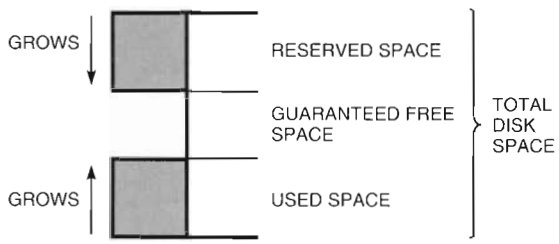


Figure 13
Modeling Free Space

Future Directions

This section outlines some of the possibilities for future implementations of the Spiralog file system.

Hierarchical Storage Management

The Spiralog server distinguishes between the logical position of a segment in the log and its physical location on the media by means of the segment array. This mapping can be extended to cover a hierarchy of devices with differing access characteristics, opening up the possibility of transparent data shelving. Since the unit of migration is the segment, even large, sparsely used files can benefit. Segments containing sections of the file not held on the primary media can be retrieved from slower storage as required. This is identical to the virtual memory paging concept.

For applications that require a complete history of the file system, segments can be saved to archive media before being recycled by the cleaner. In principle, this would make it possible to reconstruct the state of the file system at any time.

Disk Mirroring (RAID 1) Improvements

When a mirrored set of disks is forcefully dismounted with outstanding updates, such as when a system crashes, rebuilding a consistent disk state can be an expensive operation. A complete scan of the members may be necessary because I/Os may have been outstanding to any part of the mirrored set.

Because the data on an LFS disk is temporally ordered, making the members consistent following a failure is much more straightforward. In effect, an LFS allows the equivalent of the minimerge functionality provided by Volume Shadowing for OpenVMS, without the need for hardware support such as I/O controller logging of operations.[18]

Compression

Adding file compression to an update-in-place file system presents a particular problem: what to do when a data item is overwritten with a new version that does not compress to the same size. Since all updates take place at the tail of the log, an LFS avoids this problem entirely. In addition, the amount of space consumed by a data item is determined by its size and is not influenced by the cluster size of the disk. For this reason, an LFS does not need to employ file compaction to make efficient use of large disks or RAID sets.[19]

Future Improvements

The existing implementation can be improved in a number of areas, many of which involve resource consumption. The B-tree mapping mechanism, although general and flexible, has high CPU overheads and requires complex recovery algorithms. The segment layout needs to be revisited to remove the need for serialized I/Os when committing write operations and thus further reduce the write latency.

For the Spiralog file system version 1.0, we chose to keep complete information about live data and data that was no longer valid for every segment in the log. This mechanism allows us to reduce the overhead of the cleaner; however, it does so at the expense of memory and disk space and consequently does not scale well to multi-terabyte disks.

A Final Word

Log structuring is a relatively new and exciting technology. Building Digital's first product using this technology has been both a considerable challenge and a great deal of fun. Our experience during the construction of the Spiralog product has led us to believe that LFS technology has an important role to play in the future of file systems and storage management.

Acknowledgments

We would like to take this opportunity to acknowledge the contributions of the many individuals who helped during the design of the Spiralog server. Alan Paxton was responsible for initial investigations into LFS technology and laid the foundation for our understanding. Mike Johnson made a significant contribution to the cleaner design and was a key member of the team that built the final server. We are very grateful to colleagues who reviewed the design at various stages, in particular, Bill Laing, Dave Thiel, Andy Goldstein, and Dave Lomet. Finally, we would like to thank Jim Johnson and Cathy Foley for their continued loyalty, enthusiasm, and direction during what has been a long and sometimes hard journey.

References

1. D. Gifford, R. Needham, and M. Schroeder, "The Cedar File System," *Communications of the ACM,* vol. 31, no. 3 (March 1988).
2. S. Chutanai, O. Anderson, M. Kazar, and B. Leverett, "The Episode File System," *Proceedings of the Winter 1992 USENIX Technical Conference* (January 1992).
3. M. Rosenblum, "The Design and Implementation of a Log-Structured File System," Report No. UCB/CSD 92/696, University of California, Berkeley (June 1992).
4. J. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: The Case for Log-Structured File Systems," *Operating Systems Review* (January 1989).
5. R. Green, A. Baird, and J. Davies, "Designing a Fast, On-line Backup System for a Log-structured File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 32–45.
6. J. Ousterhout et al., "A Comparison of Logging and Clustering," Computer Science Department, University of California, Berkeley (March 1994).
7. M. Seltzer, K. Bostic, M. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the Winter 1993 USENIX Technical Conference* (January 1993).
8. M. Wiebren de Jounge, F. Kaashoek, and W.-C. Hsieh, "The Logical Disk: A New Approach to Improving File Systems," *ACM SIGOPS '93* (December 1993).
9. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques* (San Mateo, Calif.: Morgan Kaufman Publishers, 1993), ISBN 1-55860-190-2.
10. A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, "The Echo Distributed File System," Digital Systems Research Center, Research Report 111 (September 1993).
11. J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 5–14.
12. A. Sweeney et al., "Scalability in the XFS File System," *Proceedings of the Winter 1996 USENIX Technical Conference* (January 1996).
13. J. Kohl, "Highlight: Using a Log-structured File System for Tertiary Storage Management," USENIX Association Conference Proceedings (January 1993).
14. M. Baker et al., "Measurements of a Distributed File System," Symposium on Operating System Principles (SOSP) 13 (October 1991).
15. J. Ousterhout et al., "A Trace-driven Analysis of the UNIX 4.2 BSD File System," Symposium on Operating System Principles (SOSP) 10 (December 1985).

16. D. Lomet and B. Salzberg, "Concurrency and Recovery for Index Trees," Digital Cambridge Research Laboratory, Technical Report (August 1991).

17. P. O'Neil, "The Escrow Transactional Model," *ACM Transactions on Distributed Systems,* vol. 11 (December 1986).

18. *Volume Shadowing for OpenVMS AXP Version 6.1* (Maynard, Mass.: Digital Equipment Corp., 1994).

19. M. Burrows et al., "On-line Data Compression in a Log-structured File System," Digital Systems Research Center, Research Report 85 (April 1992).

Biographies

Christopher Whitaker
Chris Whitaker joined Digital in 1988 after receiving a B.Sc. Eng. (honours, 1st-class) in computer science from the Imperial College of Science and Technology, University of London. He is a principal software engineer with the OpenVMS File System Development Group located near Edinburgh, Scotland. Chris was the team leader for the LFS server component of the Spiralog file system. Prior to this, Chris worked on the distributed transaction management services (DECdtm) for OpenVMS and the port of the OpenVMS record management services (RMS and RMS journaling) to Alpha.

J. Stuart Bayley
Stuart Bayley is a member of the OpenVMS File System Development Group, located near Edinburgh, Scotland. He joined Digital in 1990 and prior to becoming a member of the Spiralog LFS server team, worked on OpenVMS DECdtm services and the OpenVMS XQP file system. Stuart graduated from King's College, University of London, with a B.Sc. (honours) in physics in 1986.

Rod D. W. Widdowson
Rod Widdowson received a B.Sc. (1984) and a Ph.D. (1987) in computer science from Edinburgh University. He joined Digital in 1990 and is a principal software engineer with the OpenVMS File System Development Group located near Edinburgh, Scotland. Rod worked on the implementation of LFS and cluster distribution components of the Spiralog file system. Prior to this, Rod worked on the port of the OpenVMS XQP file system to Alpha. Rod is a charter member of the British Computer Society.

Designing a Fast, On-line Backup System for a Log-structured File System

Russell J. Green
Alasdair C. Baird
J. Christopher Davies

The Spiralog file system for the OpenVMS operating system incorporates a new technical approach to backing up data. The fast, low-impact backup can be used to create consistent copies of the file system while applications are actively modifying data. The Spiralog backup uses the log-structured file system to solve the backup problem. The physical on-disk structure allows data to be saved at near-maximum device throughput with little processing of data. The backup system achieves this level of performance without compromising functionality such as incremental backup or fast, selective restore.

Most computer users want to be able to recover data lost through user error, software or media failure, or site disaster but are unwilling to devote system resources or downtime to make backup copies of the data. Furthermore, with the rapid growth in the use of data storage and the tendency to move systems toward complete utilization (i.e., 24-hour by 7-day operation), the practice of taking the system off line to back up data is no longer feasible.

The Spiralog file system, an optional component of the OpenVMS Alpha operating system, incorporates a new approach to the backup process (called simply backup), resulting in a number of substantial customer benefits. By exploiting the features of log-structured storage, the backup system combines the advantages of two different traditional approaches to performing backup: the flexibility of file-based backup and the high performance of physically oriented backup.

The design goal for the Spiralog backup system was to provide customers with a fast, application-consistent, on-line backup. In this paper, we explain the features of the Spiralog file system that helped achieve this goal and outline the design of the major backup functions, namely volume save, volume restore, file restore, and incremental management. We then present some performance results arrived at using Spiralog version 1.1. The paper concludes with a discussion of other design approaches and areas for future work.

Background

File system data may be lost for many reasons, including

- User error—A user may mistakenly delete data.
- Software failure—An application may execute incorrectly.
- Media failure—The computing equipment may malfunction because of poor design, old age, etc.
- Site disaster—Computing facilities may experience failures in, for example, the electrical supply or cooling systems. Also, environmental catastrophes such as electrical storms and floods may damage facilities.

The ability to save backup copies of all or part of a file system's information in a form that allows it to be restored is essential to most customers who use computing resources. To understand the backup capability needed in the Spiralog file system, we spoke to a number of customers—five directly and several hundred through public forums. Each ran a different type of system in a distinct environment, ranging from research and development to finance on OpenVMS and other systems. Our survey revealed the following set of customer requirements for the Spiralog backup system:

1. Backup copies of data must be consistent with respect to the applications that use the data.
2. Data must be continuously available to applications. Downtime for the purpose of backup is unacceptable. An application must copy all data of interest as it exists at an instant in time; however, the application should also be allowed to modify the data during the copying process. Performing backup in such a way as to satisfy these constraints is often called hot backup or on-line backup. Figure 1 illustrates how data inconsistency can occur during an on-line backup.
3. The backup operations, particularly the save operation, must be fast. That is, copying data from the system or restoring data to the system must be accomplished in the time available.
4. The backup system must allow an incremental backup operation, i.e., an operation that captures only the changes made to data since the last backup.

The Spiralog backup team set out to design and implement a backup system that would meet the four customer requirements. The following section discusses the features of the implementation of a log-structured file system (LFS) that allowed us to use a new approach to performing backup. Note that throughout this paper we use *disk* to describe the physical media used to store data and *volume* to describe the abstraction of the disk as presented by the Spiralog file system.

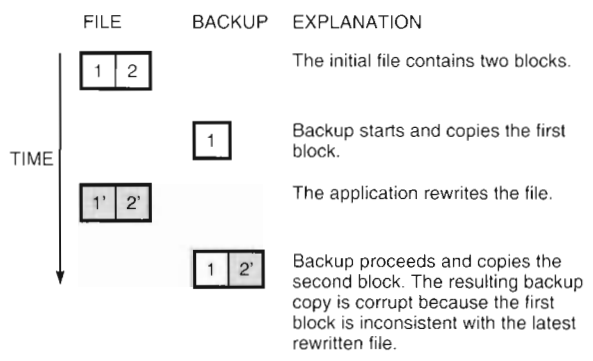


Figure 1
Example of an On-line Backup That Results in Inconsistent Data

Spiralog Features

The Spiralog file system is an implementation of a log-structured file system. An LFS is characterized by the use of disk storage as a sequential, never-ending repository of data. We generally refer to this organization of data as a log. Johnson and Laing describe in detail the design of the Spiralog implementation of an LFS and how files are maintained in this implementation.[1] Some features unique to a log-structured file system are of particular interest in the design of a backup system.[2–4] These features are

- Segments, where a segment is the fundamental unit of storage
- The no-overwrite nature of the system
- The temporal ordering of on-disk data structures
- The means by which files are constructed

This section of the paper discusses the relevance of these features; a later section explains how these features are exploited in the backup design.

Segments

In this paper, the term segment refers to a logical entity that is uniquely identified and never overwritten. This definition is distinct from the physical storage of a segment. The only physical feature of interest to backup with regard to segments is that they are efficient to read in their entirety.

Using log-structured storage in a file system allows efficient writing irrespective of the write patterns or load to the file system. All write operations are grouped in segment-sized chunks. The segment size is chosen to be sufficiently large that the time required to read or write the segment is significantly greater than the time required to access the segment, i.e., the time required for a head seek and rotational delay on a magnetic disk. All data (except the LFS homeblock and checkpoint information used to locate the end of the data log) is stored in segments, and all segments are known to the file system. From a backup point of view, this means that the entire contents of a volume can be copied by reading the segments. The segments are large enough to allow efficient reading, resulting in a near-maximum transfer rate of the device.

No Overwrite

In a log-structured file system, in which the segments are never overwritten, all data is written to new, empty segments. Each new segment is given a segment identifier (segid) allocated in a monotonically increasing

manner. At any point in time, the entire contents and state of a volume can be described in terms of a (*checkpoint position, segment list*) pair. At the physical level, a volume consists of a list of segments and a position within a segment that defines the end of the log. Rosenblum describes the concept of time travel, where an old state of the file system can be revisited by creating and maintaining a snapshot of the file system for future access.[3] Allowing time travel in this way requires maintaining an old checkpoint and disabling the reuse of disk space by the cleaner. The cleaner is a mechanism used to reclaim disk space occupied by obsolete data in a log, i.e., disk space no longer referenced in the file system. The contents of a snapshot are independent of operations undertaken on the live version of the file system. Modifying or deleting a file affects only the live version of the file system (see Figure 2). Because of the no-overwrite nature of the LFS, previously written data remains unchanged.

Other mechanisms specific to a particular backup algorithm have been developed to achieve on-line consistency.[5] The snapshot model as described above allows a more general solution with respect to multiple concurrent backups and the choice of the save algorithm.

A read-only version of the file system at an instant in time is precisely what is required for application consistency in on-line backup. This snapshot approach to attaining consistency in on-line backup has been used in other systems.[6,7] As explained in the following sections, the Spiralog file system combines the snapshot technique with features of log-structured storage to obtain both on-line backup consistency and performance benefits for backup.

Temporal Ordering

As mentioned earlier, all data, i.e., user data and file system metadata (data that describes the user data in the file system), is stored in segments and there is no overwrite of segments. All on-disk data structures that refer to physical placement of data use pointers, namely (*segid, offset*) pairs, to describe the location of the data. Each (*segid, offset*) pair specifies the segment and where within that segment the data is stored. Together, these imply the following two properties of data structures, which are key features of an LFS:

1. On-disk structure pointers, namely (*segid, offset*) pairs, are relatively time ordered. Specifically, data stored at ($s2$, $o2$) was written more recently than data stored at ($s1$, $o1$) if and only if $s2$ is greater than $s1$ or $s2$ equals $s1$ and $o2$ is greater than $o1$. Thus, new data would appear to the right in the data structure depicted in Figure 3.
2. Any data structure that uses on-disk pointers stored within the segments (the mapping data structure implementing the LFS index) must be time ordered; that is, all pointers must refer to data written prior to the pointer. Referring again to Figure 3, only data structures that point to the left are valid.

These properties of on-disk data structures are of interest when designing backup systems. Such data structures can be traversed so that segments are read in reverse time order. To understand this concept, consider the root of some on-disk data structure. This root must have been written after any of the data to which it refers (property 2). A data item that the root references must have been written before the root and so must have been stored in a segment with a segid less than or equal to that of the segment in which the root is stored (property 1). A similar inductive argument can be used to show that any on-disk data structure can be traversed using a single pass of segments in increasing segment age, i.e., decreasing segid. This is of particular interest when considering how to recover selective pieces of data (e.g., individual files) from an on-disk structure that has been stored in such a way that only sequential access is viable. The storage of the segments that compose a volume on tape as part of a backup is an example of such an on-disk data structure.

File Construction

Whitaker, Bayley, and Widdowson describe the persistent address space as exported by the Spiralog LFS.[8] Essentially, the interface presented by the log-structured server is that of a memory (various read and write operations) indexed using a file identifier and an address range. The entire contents of a file, regardless of type or size, are defined by the file identifier and all possible addresses built using that identifier.

This means of file construction is important when considering how to restore the contents of a file. All

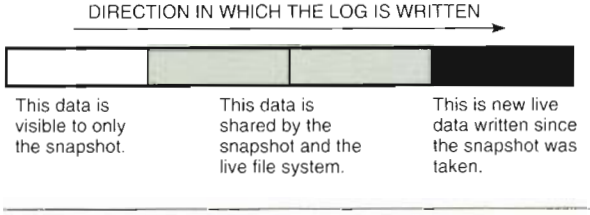


Figure 2
Data Accessible to the Snapshot and to the Live File System

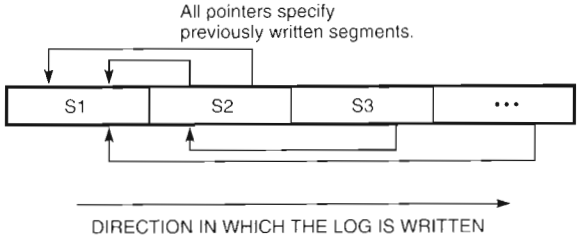


Figure 3
A Valid Data Structure in the Log

data contained in a file defined by a file identifier can be recovered, independent of how the file was created, without any knowledge of the file system structure. Consequently, together with the temporal ordering of data in an LFS, files can be recovered using an ordered linear scan of the segments of a volume, provided the on-disk data structures are traversed correctly. This mechanism allows efficient file restore from a sequence of segments. In particular, a set of files can be restored in a single pass of a saved volume stored on tape.

Existing Approaches to Backup

The design of the Spiralog backup attempts to combine the advantages of file-based backup tools such as Files-11 backup, UNIX tar, and Windows NT backup, and physical backup tools such as UNIX dd, Files-11 backup/PHYSICAL, and HSC backup (a controller-based backup for OpenVMS volumes).[9]

File-based Backup

A file-based backup system has two main advantages: (1) the system can explicitly name files to be saved, and (2) the system can restore individual files. In this paper, the file or structure that contains the output data of a backup save operation is called a saveset. Individual file restore is achieved by scanning a saveset for the file and then recreating the file using the saved contents. Incremental file-based backup usually entails keeping a record of when the last backup was made (either on a per-file basis or on a per-volume basis) and copying only those files and directories that have been created or modified since a previous backup time.

The penalty associated with these features of a file-based backup system is that of save performance. In effect, the backup system performs a considerable amount of work to lay out data in the saveset to allow simple restore. All files are segregated to a much greater extent than they are in the file system on-disk structure. The limiting factor in the performance of a file-based save operation is the rate at which data can be read from the source disk. Although there are some ways to improve performance, in the case of a volume that has a large number of files, read performance is always costly. Figure 4 illustrates the layouts of three different types of savesets.

Physical Backup

In contrast to the file-based approach to backup, a physical backup system copies the actual blocks of data on the source disk to a saveset. The backup system is able to read the disk optimally, which allows an implementation to achieve data throughput near the disk's maximum transfer rate. Physical backups typically allow neither individual file restore nor incremental backup. The overhead required to include sufficient information for these features usually erodes the performance benefits offered by the physical copy. In addition, a physical backup usually requires that the entire volume be saved regardless of how much of the volume is used to store data.

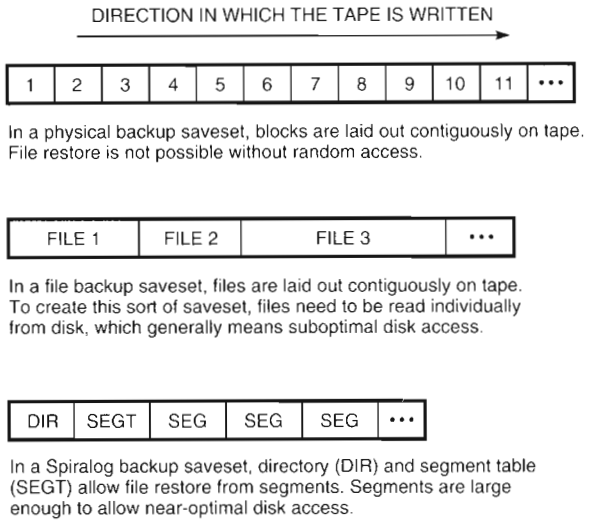


Figure 4
Layouts of Three Different Types of Saveset

How Spiralog Backup Exploits the LFS

Spiralog backup uses the snapshot mechanism to achieve on-line consistency for backup. This section describes how Spiralog attains high-performance backup with respect to the various save and restore operations.

Volume Save Operation

The save operation of Spiralog creates a snapshot and then physically copies it to a tape or disk structure called a savesnap. (This term is chosen to be different from saveset to emphasize that it holds a consistent snapshot of the data.) This physical copy operation allows high-performance data transfer with minimal processing.[10] In addition, the temporal ordering of data stored by Spiralog means that this physical copy operation can also be an incremental operation.

The savesnap is a file that contains, among other information, a list of segments exactly as they exist in the log. The structure of the savesnap allows the efficient implementation of volume restore and file restore (see Figure 5 and Figure 6).

The steps of a full save operation are as follows:

1. Create a snapshot and mount it. This mounted snapshot looks like a separate, read-only file system. Read information about the snapshot.

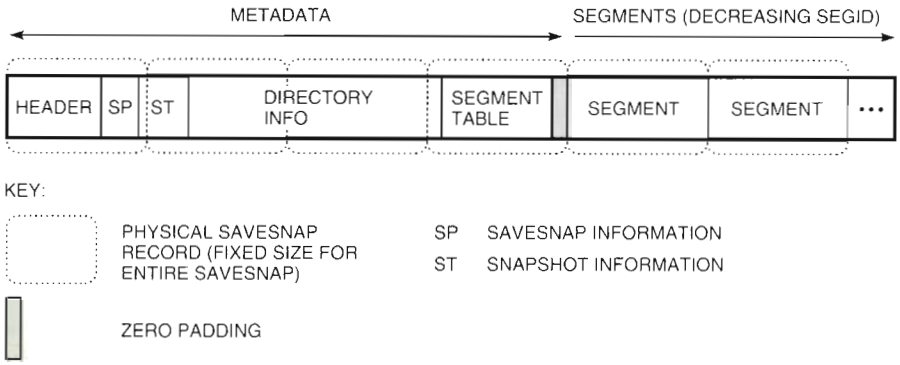


Figure 5
Savesnap Structure

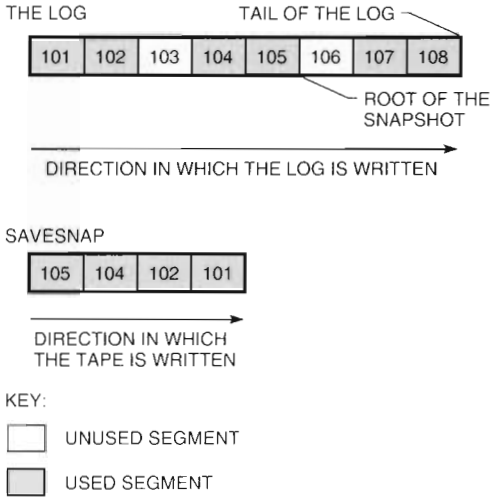


Figure 6
Correspondence between Segments on Disk and in the Savesnap

2. Write the header to the savesnap, including snapshot information such as the checkpoint position.
3. Copy the contents of the file system directories to the savesnap.
4. Write the list of segids that compose the snapshot to the savesnap as a segment table in decreasing segid order.
5. Copy these segments in decreasing segid order from the volume to the savesnap (see Figure 6).
6. Dismount and delete the snapshot, leaving only the contents of the live volume accessible. The effect of deleting the snapshot is to release all the space used to store segments that contain only snapshot data. All segments that contain data in the live volume are left intact.

The Spiralog backup system is primarily physical. The system copies the volume (snapshot) data in segments that are large enough to allow efficient disk reading, regardless of the number of files in the volume. To save a volume, the Spiralog backup system has to read all the directories in the volume and then all the segments. In comparison, a file-based backup system must read all the directories and then all the files. On volumes with large file populations, file-based backup performance suffers greatly as a result of the number of read operations required to save the volume. Our measurements showed that the directory-gathering phase of our copy operation was insignificant in relation to the data transfer during the segment copy phase.

Incremental Save Operation

The incremental save operation in Spiralog is very different from that in a file-based backup. We use the temporal ordering feature of the LFS to capture only the changes in a volume's data as part of the incremental save. The temporal ordering provides a simple way of determining the relative age of data. To be precise, data in the segment with segid $s2$ must have been written after data in the segment with segid $s1$ if and only if $s2$ is greater than $s1$.

Consider the lifetime of a volume as an endless sequence of segments. A backup copy of a volume at any time is a copy of all segments that contain data accessible in that volume. Segments in the volume's history that are not included in the backup copy are those that no longer contain any useful data or those that have been cleaned. An incremental backup contains the sequence of segments containing accessible data written since a previous backup.

This is different from an incremental save operation in a file-based backup scheme. The Spiralog incremental save operation copies only the data written since the last backup. In comparison, a file-based backup

incremental save comprises entire files that contain new or modified data. For example, consider an incremental save of a volume in which a large database file has had only one record updated in place since a full backup. Spiralog's incremental save copies the segments written since the last full backup that contain the modified record with other updated file system index data. A file-based backup copies the entire database file.

The following steps for the incremental save operation augment the six process steps previously described for the save operation. Note that steps 3a, 4a, and 5a follow steps 3, 4, and 5, respectively.

3a. Write dependent savesnap information. This is a list of the savesnaps required to complete the chain of segments that constitutes the entire snapshot contents. The savesnap information includes a unique savesnap identifier (*volume id, segment id, segment offset*). This is the checkpoint position of the snapshot and is unique across volumes.

4a. Determine the segment range to be stored in this savesnap. This range is calculated by reading the segment range of the last backup from a file stored on the source volume.

5a. Record the minimum segid stored in this savesnap with the segment table. The segment table contains the segids of all segments in the saved snapshot. The incremental savesnap contains segments identified by a subset of these segids. The segid of the last segment stored in the savesnap is recorded as the minimum segid held in the savesnap.

7. Record on the source volume the segment range stored in the savesnap.

The implementation provides an interface that allows the user to specify the maximum number of savesnaps required for a restore operation. This feature is similar to specifying the levels in the UNIX dump utility, where a level 0 save is a full backup (it requires no other savesnaps for a restore), and a level 1 save is an incremental backup since the full backup (it requires one additional savesnap for a restore, namely the full backup).

Figure 7 shows the savesnaps produced from full and incremental save operations. Note that the most recently written segment may appear in two different savesnaps that supposedly contain disjoint data. For example, segment 4, the youngest segment in Monday's savesnap, appears in the savesnaps made on both Monday and Wednesday. The youngest segment is not guaranteed to be full at the time of a snapshot creation, and therefore a later savesnap may contain data that was not in the first savesnap. Consequently, incremental savesnaps recapture the oldest segment in their segment range.

Note that with this design a slowly changing file can be spread across many incremental savesnaps. Restoring such a file accordingly may require access to many savesnaps. The file restore section shows that the design of file restore allows efficient tape traversal for these files.

Volume Restore Operation

The Spiralog backup volume restore operation takes a set of savesnaps and copies the segments that make up a snapshot onto a disk. Together, this set of segments and the location of the snapshot checkpoint define a volume. The steps involved in a volume restore from a full savesnap are

1. Open the savesnap, and read the snapshot checkpoint position from the savesnap header.
2. Initialize the target disk to be a Spiralog volume.
3. Copy all segments from the savesnap to the target disk. Note that the segments written to the target disk do not depend in any way on the target disk geometry. This means that the target disk may be completely different from the source

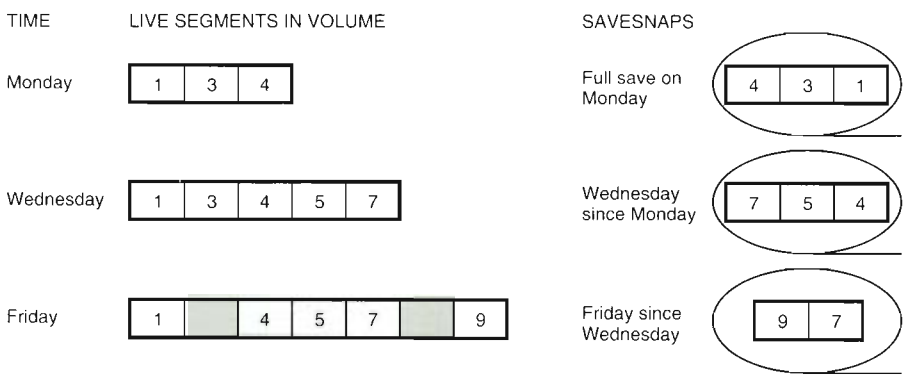


Figure 7
Snapshot Contents in Incremental Savesnaps

disk from which the savesnap was made, providing the target container is large enough to hold the restored segments.

4. Backup declares the volume restore as complete (no more segments will be written to the volume). Backup tells the file system how to mount the volume by supplying the snapshot checkpoint location.

A Spiralog restore operation treats an incremental savesnap and all the preceding savesnaps upon which it depends as a single savesnap. For savesnaps other than the most recent savesnap (the base savesnap), the snapshot information and directory information are ignored. The sole purpose of these savesnaps is to provide segments to the base savesnap.

To restore a volume from a set of incremental savesnaps, the Spiralog backup system performs steps 1 and 2 using the base savesnap. In step 3, the restore copies all the segments in the snapshot defined by the base savesnap to the target disk. (Note that there is a one-to-one correspondence between snapshots and savesnaps.) The savesnaps are processed in reverse chronological order. The contents of the segment table in the base savesnap define the list of segments in the snapshot to be restored. Although the volume restore operation copies all the segments in the base savesnap, not all segments in the savesnaps processed may be required. Savesnaps are included in the restore process if they contain some segments that are needed. Such savesnaps may also contain segments that were cleaned before the base savesnap was created.

The structure of the savesnap allows the efficient location and copying of specific segments. The segment table in the savesnap describes exactly which segments are stored in the savesnap. Since the segments are of a fixed size, it is easy to calculate the position within the savesnap where a particular segment is stored, provided the segment table is available and the position of the first segment is known. This will always be the case by the time the segment table has been read because the segments immediately follow this table.

Most savesnaps are stored on tape. This storage medium lends itself to the indexing just described. In particular, modern tape drives such as the Digital Linear Tape (DLT) series provide fast, relative tape positioning that allows tape-based savesnaps to be selectively read more quickly than with a sequential scan.[11] Similarly, on random-access media such as disks, a particular segment can be read without strict sequential scanning of data.

The volume restore operation is therefore a physical operation. The segments can be read and written efficiently (even in the case of incremental savesnaps from sequential media), resulting in a high-performance recovery from volume failure or site disaster.

File Restore Operation

The purpose of a file restore operation is to provide a fast and efficient way to retrieve a small number of files from a savesnap without performing a full volume restore. Typically, file restore is used to recover files that have been inadvertently deleted. To achieve high-performance file restore, we imposed the following requirements on the design:

- A file restore session must process as few savesnaps as possible; it should skip savesnaps that do not contain data needed by the session.
- When processing a savesnap, the file restore must scan the savesnap linearly, in a single pass.

The process of restoring files can be broken down into three steps: (1) discover the file identifiers for all the files to be restored; (2) use the file identifiers to locate the file data in the saved segments, and then read that data; and (3) place the newly recovered data back into the current Spiralog file system.

Discovering the File Identifiers The user supplies the names of the files to be restored. The mapping between the file names and the file identifiers associated with these names is stored in the segments, but this information cannot be discovered simply by inspecting the contents of the saved segments. A corollary of the temporal ordering of the segments within a savesnap is that hierarchical information, such as nested directories, tends to be presented in precisely the wrong order for scanning in a single pass. To overcome this problem, the save operation writes the complete directory tree to the savesnap before copying any segments to the savesnap. This tree maps file names to identifiers for every file and directory in the savesnap. The file restore session constructs a partial tree of the names of the files to be restored. The partial tree is then matched, in a single pass, against the complete tree stored in the savesnap. This process produces the required file identifiers.

Locating and Reading the File Data After discovering the file identifiers, the file restore session reads the list of segments present in the savesnap; this list comes after the directory tree and before any saved segments. The file restore then switches focus to discover precisely which segments contain the file data that correspond to the file identifiers.

The first segment read from the savesnap contains the tail of the log. The log provides a mapping between file identifiers and locations of data within segments. The tail of the log contains the root of the map.

We developed a simple interface for the file restore to use to navigate the map. Essentially, this interface permits the retrieval of all mapping information

relevant to a particular file identifier that is held within a given segment. The mapping information returned through this interface describes either mapping information held elsewhere or real file data. One characteristic of the log is that anything to which such mapping information points must occur earlier in the log, that is, in a subsequent saved segment. Recall property 2 of the LFS on-disk data structures. Consequently, the file restore session will progress through the savesnaps in the desired linear fashion provided that requests are presented to the interface in the correct order. The correct order is determined by the allocation of segids. Since segids increase monotonically over time, it is necessary only to ensure that requests are presented in a decreasing segid order.

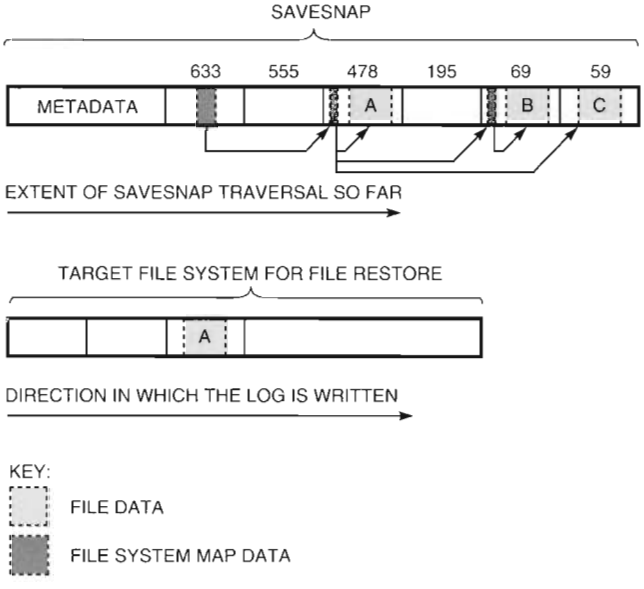
The file restore interface operates on an object called a context. The context is a tuple that contains a location in the log, namely (*segid, offset*), and a type field. When supplied with a file identifier and a context, the core function of the interface inspects the segment determined by the context and returns the set of contexts that enumerate all available mapping information for the file identifier held at the location given by the initial context.

The type of context returned indicates one of the following situations:

- The location contains real file data.
- The location given by the context holds more mapping information. In this case, the core function can be applied repeatedly to determine the precise location of the file's data.

A work list of contexts in decreasing segid order drives the file restore process. The procedure for retrieving the data for a single file identifier is as follows. At the outset of the file restore operation, the work list holds a single context that identifies the root of the map (the tail of the log). As items are taken from the head of the list, the file restore must perform one of two actions. If the context is a pointer to real file data, then the file restore reads the data at that location. If the context holds the location of mapping information, then the core function must be applied to enumerate all possible further mapping information held there. The file restore operation places all returned contexts in the work list in the correct order prior to picking the next work item. This simple procedure, which is illustrated in Figure 8, continues until the work list is empty and all the file's data has been read.

To cope with more than one file, the file restore operation extends this procedure by converting the work list so that it associates a particular file identifier



The shaded areas represent the file data to be restored and the file system metadata that needs to be accessed to retrieve that data. The restore session has thus far processed segment 478. Part A of the file has been recovered into the target file system. Parts B and C are still to come. After processing segment 478, the file restore visits the next known parts of the log, segments 69 and 59. Items that describe metadata in segment 69 and data in segment 59 will be on the work list. The next segment that the file restore will read is segment 69, so the session can skip the intervening segment (segment 195).

Figure 8
File Restore Session in Progress

with each context. File restore initializes the work list to hold a pointer to the root of the map (the tail of the log) for each file identifier to be restored. The effect is to interleave requests to read more than one file while maintaining the correct segid ordering.

A further subtlety occurs when the context at the head of the work list is found to refer to a segment outside the current savesnap. The ordering imposed on the work list implies that all subsequent items of work must also be outside the current savesnap. This follows from the temporal ordering properties of LFS on-disk structures and the way in which incremental savesnaps are defined. When this situation occurs, the work list is saved. When the next savesnap is ready for processing, the file restore session can be restarted using the saved work list as the starting point.

During this step, the file restore writes the pieces of files to the target volume as they are read from the savesnap. Since the file restore process allocates file identifiers on a per-volume basis, restore must allocate new file identifiers in the target volume to accept the data being read from the source savesnap.

The new file identifiers are hidden from users during the file restore until the file restore process has finished since the files are not complete and may be missing vital parts such as access permissions. Rather than allow access to these partial files, the file restore hides the new file identifiers until all the data is present, at which time the final stage of the file restore can take place.

Making the Recovered Files Available to the User In the third step of the process, the file restore operation makes the newly recovered files accessible. At the beginning of the step, the files exist only as bits of data associated with new file identifiers—the files do not yet have names. The names that are now bound to these file identifiers come from the partial directory tree that was originally used to match against the directory tree in the savesnap. This final step restores the original names and contents to all the files that were originally requested. The files retain the new file identifiers that were allocated during the file restore process.

Management of Incremental Saves

One design goal for the Spiralog backup was to reduce the cost of storage management. The design includes the means of performing an incremental volume save that copies only data written since the previous backup. To implement a backup strategy that never requires more than one full backup but allows restores using a finite number of savesnaps, we designed and implemented the savesnap merge function.

Savesnap merge operates similarly to volume restore, but instead of copying segments to a disk as in a volume restore, savesnap merge copies segments to a new savesnap. As shown in Figure 9, the effect of merging a base savesnap and all the incremental savesnaps upon which it depends is to produce a full savesnap. This savesnap is precisely the one that would have been created had the base savesnap been specified as a full savesnap instead of an incremental savesnap. Spiralog merge copies the savesnap information and the directory information stored in the base savesnap to the merged savesnap before it copies the segment table and the segments.

Savesnap merge provides a practical way of managing very large data volumes. The merge operation can be used to limit the number of savesnaps required to restore a snapshot, even if full backups are never taken. Merge is independent of the source volume and can be undertaken on a different system to allow further system management flexibility.

Summary of Spiralog Backup Features

A summary of the features and performance provided by the Spiralog backup system appears in Table 3 at the end of the Results section. For comparison, the table also contains corresponding information for the file-based and physical approaches to backup.

Results

We measured volume save and individual file restore performance on both the Spiralog backup system and the backup system for Files-11, the original OpenVMS file system. The hardware configuration consisted of a DEC 3000 Model 500 and a single RZ25 source disk each for Spiralog and Files-11 volumes, respectively. The target device for the backup was a TZ877 tape. The system was running under the OpenVMS version 7.0 operating system and Spiralog version 1.1. The volumes were populated with file distributions that reflected typical user accounts in our development environment. Each volume contained 260 megabytes (MB) of user data, which included a total of 21,682 files in 401 directories.

Volume Save Performance

For both the Spiralog backup and the Files-11 backup, we saved the source volume to a freshly initialized tape on an otherwise idle system. We measured the elapsed time of the save operation and recorded the size of the output savesnap or saveset. We averaged the results over five iterations of the benchmark. Table 1 presents these measurements and the resulting throughput.

The throughput represents the average rate in megabytes per second (MB/s) of writing to tape over the duration of a save operation. In the case of Spiralog, tape throughput varies greatly with the

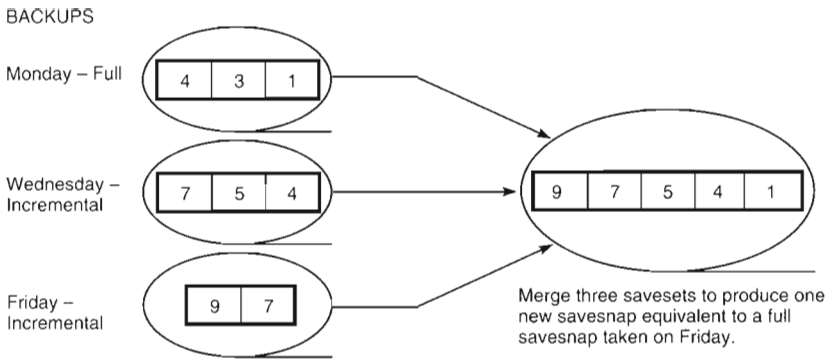


Figure 9
Merging Savesnaps

Table 1
Performance Comparison of the Spiralog and Files-11 Backup Save Operations

Backup System	Elapsed Time (Minutes:seconds)	Savesnap or Saveset Size (Megabytes)	Throughput (Megabytes/second)
Spiralog save	05:20	339	1.05
Files-11 backup	10:14	297	0.48

phases of the save operation. During the directory scan phase (typically up to 20 percent of the total elapsed save time), the only tape output is a compact representation of the volume directory graph. In comparison, the segment writing phase is usually bound by the tape throughput rate. In this configuration, the tape is the throughput bottleneck; its maximum raw data throughput is 1.25 MB/s (uncompressed).[11]

Overall, the Spiralog volume save operation is nearly twice as fast as the Files-11 backup volume save operation in this type of computing environment. Note that the Spiralog savesnap is larger than the corresponding Files-11 saveset. The Spiralog savesnap is less efficient at holding user data than the packed per-file representation of the Files-11 saveset. In many cases, though, the higher performance of the Spiralog save operation more than outweighs this inefficiency, particularly when it is taken into account that the Spiralog save operation can be performed on-line.

File Restore Performance

To determine file restore performance, we measured how long it took to restore a single file from the savesets created in the save benchmark tests. The hardware and software configurations were identical to those used for the save measurements. We deleted a single 3-kilobyte (KB) file from the source volume and then restored the file. We repeated this operation nine times, each time measuring the time it took to restore the file. Table 2 shows the results.

Table 2
Performance Comparison of the Spiralog and Files-11 Individual File Restore Operations

Backup System	Elapsed Time (Minutes:seconds)
Spiralog file restore	01:06
Files-11 backup	03:35

The Spiralog backup system achieves such good performance for file restore by using its knowledge of the way the segments are laid out on tape. The file restore process needs to read only those segments required to restore the file; the restore skips the intervening segments using tape skip commands. In the example presented in Figure 8, the restore can skip segments 555 and 195. In contrast, a file-based backup such as Files-11 usually does not have accurate indexing information to minimize tape I/O. Spiralog's tape-skipping benefit is particularly noticeable when restoring small numbers of files from very large savesnaps; however, as shown in Table 2, even with small savesets, individual file restore using Spiralog backup is three times as fast as using Files-11.

Table 3 presents a comparison of the save performance and features of the Spiralog, file-based, and physical backup systems.

Table 3
Comparison of Spiralog, File-based, and Physical Backup Systems

	Spiralog Backup System	File-based Backup System	Physical Backup System
Save performance (the number of I/Os required to save the the source volume)	The number of I/Os is O(number of segments that contain live data) plus O(number of directories)	The number of I/Os is O(number of files) I/Os to read the file data plus O(number of directories) I /Os	The number of I/Os is O(size of the disk)
File restore	Yes	Yes	No
Volume restore	Yes, fast	Yes	Yes, fast but limited to disks of the same size
Incremental save	Yes, physical	Yes, entire files that have changed	No

Note that this table uses "big oh" notation to bound a value. O(*n*), which is pronounced "order of *n*," means that the value represented is no greater than C*n* for some constant C, regardless of the value of *n*. Informally, this means that O(*n*) can be thought of as some constant multiple of *n*.

Other Approaches and Future Work

This section outlines some other design options we considered for the Spiralog backup system. Our approach offers further possibilities in a number of areas. We describe some of the opportunities available.

Backup and the Cleaner

The benefits of the write performance gains in an LFS are attained at the cost of having to clean segments.[8] An opportunity appears to exist in combining the cleaner and backup functions to reduce the amount of work done by either or both of these components; however, the aims of backup and the cleaner are quite different. Backup needs to read all segments written since a specific time (in the case of a full backup, since the birth of the volume). The cleaner needs to defragment the free space on the volume. This is done most efficiently by relocating data held in certain segments. These segments are those that are sufficiently empty to be worth scavenging for free space. The data in these segments should also be stable in the sense that the data is unlikely to be deleted or outdated immediately after relocation.

The only real benefit that can be exacted by looking at these functions together is to clean some segments while performing backup. For example, once a segment has been read to copy to a savesnap, it can be cleaned. This approach is probably not a good one because it reduces system performance in the following ways: additional processing required in cleaning removes CPU and memory resources available to applications, and the cleaner generates write operations that reduce the backup read rate.

There are two other areas in which backup and the cleaner mechanism interact that warrant further investigation.

1. The save operation copies segments in their entirety. That is, the operation copies both "stale" (old) data and live data to a savesnap. The cost of extra storage media for this extraneous data is traded off against the performance penalty in trying to copy only live data. It appears that the file system should run the cleaner vigorously prior to a backup to minimize the stale data copied.
2. Incremental savesnaps contain cleaned data. This means that an incremental savesnap contains a copy of data that already exists in one of the savesnaps on which it depends. This is an apparent waste of effort and storage space.

It is best to undertake a full backup after a thorough cleaning of the volume. A single strategy for incremental backups is less easy to define. On one hand, the size of an incremental backup is increased if much cleaning is performed before the backup. On the other hand, restore operations from a large incremental backup (particularly selective file restores) are likely to be more efficient. The larger the incremental backup, the more data it contains. Consequently, the chance of restoring a single file from just the base savesnap increases with the size of the incremental backup. Studying the interactions between the backup and the cleaner may offer some insight into how to improve either or both of these components.

A continuous backup system can take copies of segments from disk using policies similar to the cleaner. This is explored in Kohl's paper. [12]

Separating the Backup Save Operation into a Snapshot and a Copy

The design of the save operation involves the creation of a snapshot followed by the fast copy of the snapshot to some separate storage. The Spiralog version 1.1 implementation of the save operation combines these steps. A snapshot can exist only during a backup save operation.

System administrators and applications have significantly more flexibility if the split in these two functions of backup is visible. The ability to create snapshots that can be mounted to look like read-only versions of a file system may eliminate the need for the large number of backups performed today. Indeed, some file systems offer this feature.[6,7] The additional advantage that Spiralog offers is to allow the very efficient copying of individual snapshots to off-line media.

Improving the Consistency and Availability of On-line Backup

There are a number of ways to improve application consistency and availability using the Spiralog backup design. In addition, some of these features further reduce storage management costs.

Intervolume Snapshot Creation Spiralog allows a practical way of creating and managing large volumes, but there will be times when applications require data consistency for backup across volumes. A coordinated snapshot across volumes would provide this.

Application Involvement The Spiralog version 1.1 implementation does not address application involvement in the creation of a snapshot. A snapshot's contents are precisely the volume's contents that are on disk at the time of snapshot creation. This means that applications accessing the volume have to commit independently to the file system data they require to be part of the snapshot.

There is an emerging trend to design system-level interfaces that allow better application interaction with the file system. For example, the Windows NT operating system provides the oplock and NtNotifyChangeDirectory interfaces to advise an interested application of changes to files and directories. Similarly, an interface could allow applications to register an interest with the file system for notification of an impending snapshot creation. The application would then be able to commit the data it needs as part of a backup and continue, thus improving application consistency and availability and reducing work for system administrators.

Minimizing Disk Reads

The Spiralog file restore retrieves the data that constitutes a number of files in a single pass of segments read in a specific order. This design was important to allow the efficient restore of files from sequential media.

More generally, this way of traversing the file system allows specific, known parts of a set of files to be obtained by reading the segments that contain part of this data only once. This technique is also interesting for random-access media storage of volumes because it describes an algorithm for minimizing the number of disk reads to get this data. Possible applications of this technique are numerous and are particularly interesting in the context of data management of very large volumes.

For example, suppose an application is required to monitor an attribute (e.g., the time of last access) of all files on a massive volume. Suppose also that the volume is too big to allow the application to trawl the file system daily for this information; this process takes too long. If the application maintains a database of the information, it needs only to gather the changes that have happened to this data on a daily basis. Therefore, the application could obtain this information by traversing only those segments written since the last time it updated its database and locating the relevant data within those segments. Our mechanism for restoring files provides exactly this capability. An investigation of how applications might best use this technique could lead to the design of an interface that the file system could use for fast scanning of data.

Conclusions

File systems use backup to protect against data loss. A significant portion of the cost associated with managing storage is directly related to the backup function.[13–17] Log-structured data storage provides some features that reduce the costs associated with backup.

The Spiralog log-structured file system version 1.1 for the OpenVMS Alpha operating system includes a new, high-performance, on-line backup system. The approach that Spiralog takes to obtain data consistency for on-line backup is similar to the snapshot approach used in Network Appliance Corporation's FAServer, the Digital UNIX Advanced File System, and other systems.[6,7] The feature unique to the Spiralog backup system is its use of the physical attributes of log-structured storage to obtain high-performance saving and restoring of data to and from tape. In particular, the gain in save performance is the result of a restore strategy that can efficiently retrieve data from a sequence of segments stored on tape as they are on disk. This design leads to a minimum of processing and discrete I/O operations. The restore operation uses improvements in tape hardware to reduce processing and I/O bandwidth consumption; the operation uses tape record skipping within savesnaps for fast

data indexing. The Spiralog backup implementation provides an on-line backup save operation with significantly improved performance over existing offerings. Performance of individual file restore is also improved.

Acknowledgments

We would like to thank the following people whose efforts were vital in bringing the Spiralog backup system to fruition: Nancy Phan, who helped us develop the product and worked relentlessly to get it right; Judy Parsons, who helped us clarify, describe, and document our work; Clare Wells, who helped us focus on the real customer problems; Alan Paxton, who was involved in the early design ideas and later specification of some of the implementation; and, finally, Cathy Foley, our engineering manager, who supported us throughout the project.

References

1. J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 5–14.
2. M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems,* vol. 10, no. 1 (February 1992): 26–52.
3. M. Rosenblum, "The Design and Implementation of a Log-Structured File System," Report No. UCB/CSD 92/696 (Berkeley, Calif.: University of California, Berkeley, 1992).
4. M. Seltzer, K. Bostock, M. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the USENIX Winter 1993 Technical Conference,* San Diego, Calif. (January 1993).
5. K. Walls, "File Backup System for Producing a Backup Copy of a File Which May Be Updated during Backup," U.S. Patent No. 5,163,148.
6. D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the USENIX Winter 1994 Technical Conference,* San Francisco, Calif. (January 1994).
7. S. Chutani, O. Anderson, M. Kazar, and B. Leverett, "The Episode File System," *Proceedings of the USENIX Winter 1992 Technical Conference,* San Francisco, Calif. (January 1992).
8. C. Whitaker, J. Bayley, and R. Widdowson, "Design of the Server for the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 15–31.
9. *OpenVMS System Management Utilities Reference Manual: A–L,* Order No. AA-PV5PC-TK (Maynard, Mass.: Digital Equipment Corporation, 1995).
10. L. Drizis, "A Method for Fast Tape Backups and Restores," *Software—Practice and Experience,* vol. 23, no. 7 (July 1993): 813–815.
11. "Digital Linear Tape Meets Critical Need for Data Backup," Quantum Technical Information Paper, http://www.quantum.com/products/whitepapers/dlttips.html (Milpitas, Calif.: Quantum Corporation, 1996).
12. J. Kohl, C. Staelin, and M. Stonebraker, "HighLight: Using a Log-structured File System for Tertiary Storage Management," *Proceedings of the USENIX Winter 1993 Technical Conference* (Winter 1993).
13. R. Mason, "The Storage Management Market Part 1: Preliminary 1994 Market Sizing," IDC No. 9538 (Framingham, Mass.: International Data Corporation, December 1994).
14. I. Stenmark, "Implementation Guidelines for Client/Server Backup" (Stamford, Conn.: Gartner Group, March 14, 1994).
15. I. Stenmark, "Market Size: Network and Systems Management Software" (Stamford, Conn.: Gartner Group, June 30, 1995).
16. I. Stenmark, "Client/Server Backup—Leaders and Challengers" (Stamford, Conn.: Gartner Group, May 9, 1994).
17. R. Wrenn, "Why the Real Cost of Storage is More Than $1/MB," presented at the U.S. DECUS Symposium, St. Louis, Mo., June 3–6, 1996.

Biographies

Russell J. Green
Russell Green is a principal software engineer in Digital's OpenVMS Engineering group in Livingston, Scotland. He was responsible for the design and delivery of the backup component of the Spiralog file system for the OpenVMS operating system. Currently, Russ is the technical leader of Spiralog follow-on work. Prior to joining Digital in 1991, he was a staff member in the computer science department at the University of Edinburgh. Russ received a B.Sc. (Honours, 1st class, 1983) in engineering from the University of Cape Town and an M.Sc. (1986) in engineering from the University of Edinburgh. He holds two patents and has filed a patent application for his Spiralog backup system work.

Alasdair C. Baird
Alasdair Baird joined Digital in 1988 to work for the ULTRIX Engineering group in Reading, U.K. He is a senior software engineer and has been a member of Digital's OpenVMS Engineering group since 1991. He worked on the design of the Spiralog file system and then contributed to the Spiralog backup system, particularly the file restore component. Currently, he is involved in Spiralog development work. Alasdair received a B.Sc. (Honours, 1988) in computer science from the University of Edinburgh.

J. Christopher Davies
Software engineer Chris Davies has worked for Digital Equipment Corporation in Livingston, Scotland, since September 1991. As a member of the Spiralog team, he initially designed and implemented the Spiralog on-line backup system. In subsequent work, he improved the performance of the file system. Chris is currently working on further Spiralog development. Prior to joining Digital, Chris was employed by NRG Surveys as a software engineer while earning his degree. He holds a B.Sc. (Honours, 1991) in artificial intelligence and computer science from the University of Edinburgh. He is coauthor of a filed patent application for the Spiralog backup system.

Mark A. Howell
Julian M. Palmer

Integrating the Spiralog File System into the OpenVMS Operating System

Digital's Spiralog file system is a log-structured file system that makes extensive use of write-back caching. Its technology is substantially different from that of the traditional OpenVMS file system, known as Files-11. The integration of the Spiralog file system into the OpenVMS environment had to ensure that existing applications ran unchanged and at the same time had to expose the benefits of the new file system. Application compatibility was attained through an emulation of the existing Files-11 file system interface. The Spiralog file system provides an ordered write-behind cache that allows applications to control write order through the barrier primitive. This form of caching gives the benefits of write-back caching and protects data integrity.

The Spiralog file system is based on a log-structuring method that offers fast writes and a fast, on-line backup capability.[1–3] The integration of the Spiralog file system into the OpenVMS operating system presented many challenges. Its programming interface and its extensive use of write-back caching were substantially different from those of the existing OpenVMS file system, known as Files-11.

To encourage use of the Spiralog file system, we had to ensure that existing applications ran unchanged in the OpenVMS environment. A file system emulation layer provided the necessary compatibility by mapping the Files-11 file system interface onto the Spiralog file system. Before we could build the emulation layer, we needed to understand how these applications used the file system interface. The approach taken to understanding application requirements led to a file system emulation layer that exceeded the original compatibility expectations.

The first part of this paper deals with the approach to integrating a new file system into the OpenVMS environment and preserving application compatibility. It describes the various levels at which the file system could have been integrated and the decision to emulate the low-level file system interface. Techniques such as tracing, source code scanning, and functional analysis of the Files-11 file system helped determine which features should be supported by the emulation.

The Spiralog file system uses extensive write-back caching to gain performance over the write-through cache on the Files-11 file system. Applications have relied on the ordering of writes implied by write-through caching to maintain on-disk consistency in the event of system failures. The lack of ordering guarantees prevented the implementation of such careful write policies in write-back environments. The Spiralog file system uses a write-behind cache (introduced in the Echo file system) to allow applications to take advantage of write-back caching performance while preserving careful write policies.[4] This feature is unique in a commercial file system. The second part of this paper describes the difficulties of integrating write-back caching into a write-through environment and how a write-behind cache addressed these problems.

Providing a Compatible File System Interface

Application compatibility can be described in two ways: compatibility at the file system interface and compatibility of the on-disk structure. Since only specialized applications use knowledge of the on-disk structure and maintaining compatibility at the interface level is a feature of the OpenVMS system, the Spiralog file system preserves compatibility at the file system interface level only. In the section Files-11 and the Spiralog File System On-disk Structures, we give an overview of the major on-disk differences between the two file systems.

The level of interface compatibility would have a large impact on how well users adopted the Spiralog file system. If data and applications could be moved to a Spiralog volume and run unchanged, the file system would be better accepted. The goal for the Spiralog file system was to achieve 100 percent interface compatibility for the majority of existing applications. The implementation of a log-structured file system, however, meant that certain features and operations of the Files-11 file system could not be supported.

The OpenVMS operating system provides a number of file system interfaces that are called by applications. This section describes how we chose the most compatible file system interface. The OpenVMS operating system directly supports a system-level call interface (QIO) to the file system, which is an extremely complex interface.[5] The QIO interface is very specific to the OpenVMS system and is difficult to map directly onto a modern file system interface. This interface is used infrequently by applications but is used extensively by OpenVMS utilities.

OpenVMS File System Environment

This section gives an overview of the general OpenVMS file system environment, and the existing OpenVMS and the new Spiralog file system interfaces. To emulate the Files-11 file system, it was important to understand the way it is used by applications in the OpenVMS environment. A brief description of the Files-11 and the Spiralog file system interfaces gives an indication of the problems in mapping one interface onto the other. These problems are discussed later in the section Compatibility Problems.

In the OpenVMS environment, applications interact with the file system through various interfaces, ranging from high-level language interfaces to direct file system calls. Figure 1 shows the organization of interfaces within the OpenVMS environment, including both the Spiralog and the Files-11 file systems.

The following briefly describes the levels of interface to the file system.

- High-level language (HLL) libraries. HLL libraries provide file system functions for high-level languages such as the Standard C library and FORTRAN I/O functions.
- OpenVMS language-specific libraries. These libraries offer OpenVMS-specific file system functions at a high level. For example, lib$create_dir( ) creates a new directory with specific OpenVMS security attributes such as ownership.
- Record Management Services. The OpenVMS Record Management Services (RMS) are a set of complex routines that form part of the OpenVMS kernel. These routines are primarily used to access structured data within a file. However, there are also routines at the file level, for example, open, close, delete, and rename. The RMS parsing routines for file search and open give the OpenVMS operating system a consistent syntax for file names. These routines also provide file name parsing operations for higher level libraries. RMS calls to the file system are treated in the same way as direct application calls to the file system.

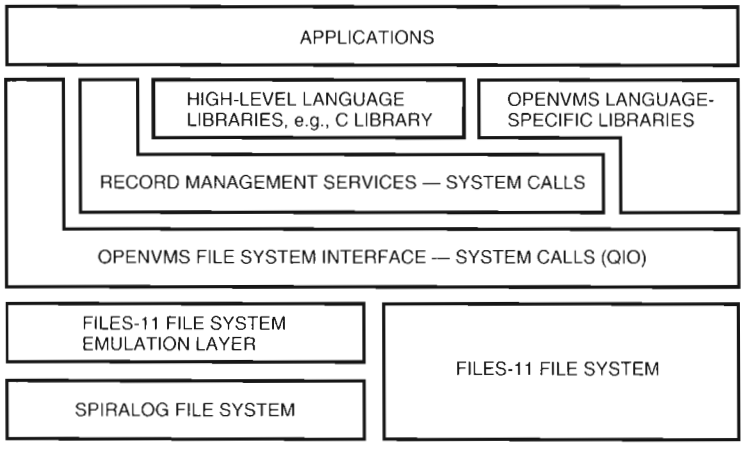


Figure 1
The OpenVMS File System Environment

- Files-11 file system interface. The OpenVMS operating system has traditionally provided the Files-11 file system for applications. It provides a low-level file system interface so that applications can request file system operations from the kernel.

  Each file system call can be composed of multiple subcalls. These subcalls can be combined in numerous permutations to form a complex file system operation. The number of permutations of calls and subcalls makes the file system interface extremely difficult to understand and use.

- File system emulation layer. This layer provides a compatible interface between the Spiralog file system and existing applications. Calls to export the new features available in the Spiralog file system are also included in this layer. An important new feature, the write-behind cache, is described in the section Overview of Caching.

- The Spiralog file system interface. The Spiralog file system provides a generic file system interface. This interface was designed to provide a superset of the features that are typically available in file systems used in the UNIX operating system. File system emulation layers, such as the one written for Files-11, could also be written for many different file systems.[6] Features that could not be provided generically, for example, the implementation of security policies, are implemented in the file system emulation layer.

  The Spiralog file system's interface is based on the Virtual File System (VFS), which provides a file system interface similar to those found on UNIX systems.[7] Functions available are at a higher level than the Files-11 file system interface. For example, an atomic rename function is provided.

Files-11 and the Spiralog File System

On-disk Structures

A major difference between the Files-11 and the Spiralog file systems is the way data is laid out on the disk. The Files-11 system is a conventional, update-in-place file system.[8] Here, space is reserved for file data, and updates to that data are written back to the same location on the disk. Given this knowledge, applications could place data on Files-11 volumes to take advantage of the disk's geometry. For example, the Files-11 file system allows applications to place files on cylinder boundaries to reduce seek times.

The Spiralog file system is a log-structured file system (LFS). The entire volume is treated as a continuous log with updates to files being appended to the tail of the log. In effect, files do not have a fixed home location on a volume. Updates to files, or cleaner activity, will change the location of data on a volume. Applications do not have to be concerned where their data is placed on the disk; LFS provides this mapping.

With the advent of modern disks in the last decade, the exact placement of data has become much less critical. Modern disks frequently return geometry information that does not reflect the exact geometry of the disk. This nullifies any advantage that exact placement on the disk offers to applications. Fortunately, with the Files-11 file system, the use of exact file placement is considered a hint to the file system and can be safely ignored.

Interface Decision

Many features of the Spiralog file system and the Files-11 file system are not directly compatible. To enable existing applications to use the Spiralog file system, a suitable file system interface had to be selected and emulated. The file system emulation layer would need to hook into an existing kernel-level file system interface to provide existing applications with access to the Spiralog file system.

Analysis of existing applications showed that the majority of file system calls came through the RMS interface. This provides a functionally simpler interface onto the lower level Files-11 interface. Most applications on the OpenVMS operating system use the RMS interface, either directly or through HLL libraries, to access the file system.

Few applications make direct calls to the low-level Files-11 interface. Calls to this interface are typically made by RMS and OpenVMS utilities that provide a simplified interface to the file system. RMS supports file access routines, and OpenVMS utilities support modification of file metadata, for example, security information. Although few in number, those applications that do call the Files-11 file system directly are significant ones. If the only interface supported was RMS, then these utilities, such as SET FILE and OpenVMS Backup, would need significant modification. This class of utilities represents a large number of the OpenVMS utilities that maintain the file system.

To provide support for the widest range of applications, we selected the low-level Files-11 interface for use by the file system emulation layer. By selecting this interface, we decreased the amount of work needed for its emulation. However, this gain was offset by the increased complexity in the interface emulation.

Problems caused by this interface selection are described in the next section.

Interface Compatibility

Once the file system interface was selected, choices had to be made about the level of support provided by the emulation layer. Due to the nature of the log-structured file system, described in the section Files-11 and the Spiralog File System On-disk Structures, full compatibility of all features in the emulation layer was not possible. This section discusses some of the decisions made concerning interface compatibility.

An initial decision was made to support documented low-level Files-11 calls through the emulation layer as often as possible. This would enable all well-behaved applications to run unchanged on the Spiralog file system. Examples of well-behaved applications are those that make use of HLL library calls. The following categories of access to the file system would not be supported:

- Those directly accessing the disk without going through the file system
- Those making use of specific on-disk structure information
- Those making use of undocumented file system features

A very small number of applications fell into these categories. Examples of applications that make use of on-disk structure knowledge are the OpenVMS boot code, disk structure analyzers, and disk defragmenters.

The majority of OpenVMS applications make file system calls through the RMS interface. Using file system call-tracing techniques, described in the section Investigation Techniques, a full set of file system calls made by RMS could be constructed. After analysis of this trace data, it was clear that RMS used a small set of well-structured calls to the low-level file system interface. Further, detailed analysis of these calls showed that all RMS operations could be fully emulated on the Spiralog file system.

The support of OpenVMS file system utilities that made direct calls to the low-level Files-11 interface was important if we were to minimize the amount of code change required in the OpenVMS code base. Analysis of these utilities showed that the majority of them could be supported through the emulation layer.

Very few applications made use of features of the Files-11 file system that could not be emulated. This enabled a high number of applications to run unchanged on the Spiralog file system.

Compatibility Problems

This section describes some of the compatibility problems that we encountered in developing the emulation layer and how we resolved them.

When considering the compatibility of the Spiralog file system with the Files-11 file system, we placed the features of the file system into three categories: supported, ignored, and not supported. Table 1 gives examples and descriptions of these categories. A feature was recategorized only if it could be supported but was not used, or if it could not be easily supported but was used by a wide range of applications.

The majority of OpenVMS applications make supported file system calls. These applications will run as intended on the Spiralog file system. Few applications make calls that could be safely ignored. These applications would run successfully but could not make use of these features. Very few applications made calls that were not supported. Unfortunately, some of these applications were very important to the success of the Spiralog file system, for example, system management utilities that were optimized for the Files-11 system.

Analysis of applications that made unsupported calls showed the following categories of use:

- Those that accessed the file header—a structure used to store a file's attributes. This method was used to return multiple file attributes in one call. The supported mechanism involved an individual call for each attribute.

  This was solved by returning an emulated file header to applications that contained the majority of information interesting to applications.
- Those reading directory files. This method was used to perform fast directory scans. The supported mechanism involved a file system call for each name.

  This was solved by providing a bulk directory reading interface call. This call was similar to the getdirentries( ) call on the UNIX system and was

Table 1
Categorization of File System Features

Category	Examples	Notes
Supported. The operation requested was completed, and a success status was returned.	Requests to create a file or open a file.	Most calls made by applications belong in the supported category.
Ignored. The operation requested was ignored, and a success status was returned.	A request to place a file in a specific position on the disk to improve performance.	This type of feature is incompatible with a log-structured file system. It is very infrequently used and not available through HLL libraries. It could be safely ignored.
Not supported. The operation requested was ignored, and a failure status was returned.	A request to directly read the on-disk structure.	This type of request is specific to the Files-11 file system and could be allowed to fail because the application would not work on the Spiralog file system. It is used only by a few specialized applications.

straightforward to replace in applications that directly read directories.

The OpenVMS Backup utility was an example of a system management utility that directly read directory files. The backup utility was changed to use the directory reading call on Spiralog volumes.

- Those accessing reserved files. The existing file system stores all its metadata in normal files that can be read by applications. These files are called reserved files and are created when a volume is initialized.

  No reserved files are created on a Spiralog volume, with the exception of the master file directory (MFD). Applications that read reserved files make specific use of on-disk structure information and are not supported with the Spiralog file system. The MFD is used as the root directory and performs directory traversals. This file was virtually emulated. It appears in directory listings of a Spiralog volume and can be used to start a directory traversal, but it does not exist on the volume as a real file.

Investigation Techniques

This section describes the approach taken to investigate the interface and compatibility problems described above. Results from these investigations were used to determine which features of the Files-11 file system needed to be provided to produce a high level of compatibility.

The investigation focused on understanding how applications called the file system and the semantics of the calls. A number of techniques were used in lieu of design documentation for applications and the Files-11 file system. These techniques were also used to avoid the direct examination of source code.

The following techniques were used to understand application calls to the file system:

- Tracing file system operations

  Tracing file system operations provided a large amount of data for applications. A modified Files-11 file system was constructed that logged all file operations on a volume. A full set of regression tests were then run for the 25 Digital and third-party products most often layered on the Files-11 file system. The data was then reduced to determine the type of file system calls made by the layered products. Analysis of log data showed that most layered products made file system calls through HLL libraries or the RMS interface. This technique is useful where source code is not available, but full code path coverage is available to construct a full picture of calls and arguments.

- Surveying application maintainers on file system use

  Surveying application maintainers was a potentially useful technique for alerting the other maintainers about the impact of the Spiralog file system. More than 2,000 surveys were sent out, but fewer than 25 useful results were returned. Sadly, most application maintainers were not aware of how their product used the file system.

- Automated application source code searching

  Automated source code searching quickly checks a large amount of source code. This technique was most useful when analyzing file system calls made by the OpenVMS operating system or utilities. However, this does not work well when applications make dynamic calls to the file system at run time.

The following techniques were used to understand the semantics of file system calls:

- Functional analysis of the Files-11 file system

  Functional analysis of the Files-11 file system was one of the most useful techniques adopted. It avoided the need to reverse-engineer the Files-11 file system. Whenever possible, the Files-11 file system was treated as a black box, and its function was inferred from interface documentation and application calls. This technique avoided duplicating defects in the interface and enabled the design of the emulation layer to be derived from function, rather than the existing implementation of the Files-11 system.

- Test programs to determine call semantics

  Test programs were used extensively to isolate specific application calls to the file system. Individual calls could be analyzed to determine how they worked with the Files-11 file system and with the emulation layer. This technique formed the basis for an extensive file system interface regression test suite without requiring the complete application.

Level of Compatibility Achieved

The level of file system compatibility with applications far exceeded our initial expectations. Table 2 summarizes the results of the regression tests used to verify compatibility.

Table 2 illustrates that applications that use the C or the FORTRAN language or the RMS interface to access the file system can be expected to work unchanged. Verification with the top 25 Digital layered products and third-party products shows that all products that do not make specific use of Files-11 on-disk features run with the Spiralog file system. With the version 1.0 release of the Spiralog file system, there are no known compatibility issues.

Providing New Caching Features

The Spiralog file system uses ordered write-back caching to provide performance benefits for applications.

Table 2
Verification of Compatibility

Test Suite	Number of Tests	Result
RMS regression tests	~500	All passed.
OpenVMS regression tests	~100	All passed.
Files-11 compatibility tests	~100	All passed.
C2 security test suite	~50 discrete tests	All passed, giving the Spiralog file system the same potential security rating as the Files-11 system.
C language tests	~2,000	All passed.
FORTRAN language tests	~100	All passed.

Write-back caching provides very different semantics to the model of write-through caching used on the Files-11 file system. The goal of the Spiralog project members was to provide write-back caching in a way that was compatible with existing OpenVMS applications.

This section compares write-through and write-back caching and shows how some important OpenVMS applications rely on write-through semantics to protect data from system failure. It describes the ordered write-back cache as introduced in the Echo file system and explains how this model of caching (known as write-behind caching) is particularly suited to the environment of OpenVMS Cluster systems and the Spiralog log-structured file system.

Overview of Caching

During the last few years, CPU performance improvements have continued to outpace performance improvements for disks. As a result, the I/O bottleneck has worsened rather than improved. One of the most successful techniques used to alleviate this problem is caching. Caching means holding a copy of data that has been recently read from, or written to, the disk in memory, giving applications access to that data at memory speeds rather than at disk speeds.

Write-through and write-back caching are two different models frequently used in file systems.

- Write-through caching. In a write-through cache, data read from the disk is stored in the in-memory cache. When data is written, a copy is placed in the cache, but the write request does not return until the data is on the disk. Write-through caches improve the performance of read requests but not write requests.
- Write-back caching. A write-back cache improves the performance of both read and write requests. Reads are handled exactly as in a write-through cache. This time though, a write request returns as soon as the data has been copied to the cache; some time later, the data is written to the disk. This method allows both read and write requests to operate at main memory speeds. The cache can also amalgamate write requests that supersede one another. By deferring and amalgamating write requests, a write-back cache can issue many fewer write requests to the disk, using less disk bandwidth and smoothing the write pattern over time.

Figure 2 shows the write-through and write-back caching models. The Spiralog file system makes extensive use of caching, providing both write-through and write-back models. The use of write-back caching allows the Spiralog file system to amalgamate writes, thus conserving disk bandwidth. This is especially important in an OpenVMS Cluster system where disk bandwidth is shared by several computers. The Spiralog file system attempts to amalgamate not just data writes but also file system operations. For example, many compilers create temporary files that are deleted at the end of the compilation. With write-back caching, it is possible that this type of file may be created and deleted without ever being written to the disk.

There are two disadvantages of write-back caching: (1) if the system fails, any write requests that have not been written to the disk are lost, and (2) once in the cache, any ordering of the write requests is lost. The data may be written from the cache to the disk in a completely different order than the order in which the application issued the write requests. To preserve data integrity, some applications rely on write ordering and the use of careful write techniques. (Careful writing is discussed further in the section below.) The Spiralog file system preserves data integrity by providing an ordered write-back cache known as a write-behind cache.

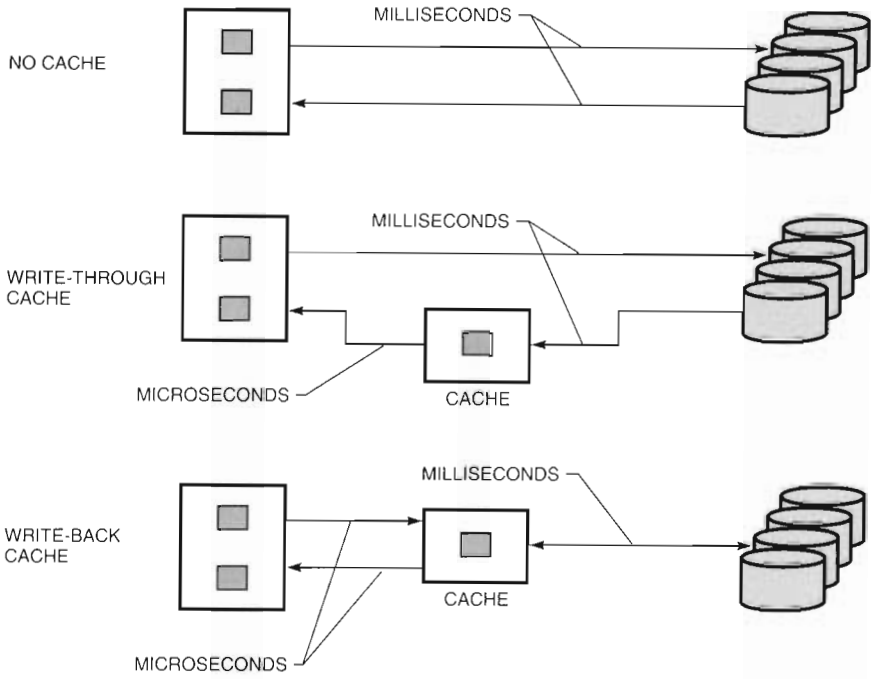


Figure 2
Caching Models

Caching is more important to the Spiralog file system than it is to conventional file systems. Log-structured file systems have inherently worse read performance than conventional, update-in-place file systems, due to the need to locate the data in the log. As described in another paper in this *Journal*, locating data in the log requires more disk I/Os than an update-in-place file system.[2] The Spiralog file system uses large read caches to offset this extra read cost.

Careful Writing

The Files-11 file system provides write-through semantics. Key OpenVMS applications such as transaction processing and the OpenVMS Record Management Services (RMS) have come to rely on the implicit ordering of write-through. They use a technique known as careful writing to prevent data corruption following a system failure.

Careful writing allows an application to ensure that the data on the disk is never in an inconsistent or invalid state. This guarantee avoids situations in which an application has to scan and possibly rebuild the data on the disk after a system failure. Recovery to a consistent state after a system failure is often a very complex and time-consuming task. By ensuring that the disk can never be inconsistent, careful writing removes the need for this form of recovery.

Careful writing is used in situations in which an update requires several blocks on the disk to be written. Most disks guarantee atomic update of only a single disk block. The occurrence of a system failure while several blocks are being updated could leave the blocks partially updated and inconsistent. Careful writing avoids this risk by defining the order in which the blocks should be updated on the disk. If the blocks are written in this order, the data will always be consistent.

For example, the file shown in Figure 3 represents a persistent data structure. At the start of the file is an index block, I, that points to two data blocks within the file, A and B. The application wishes to update the data (A, B) to the new data (A′, B′). For the file to be valid, the index must point to a consistent set of data blocks. So, the index must point either to (A, B) or to (A′, B′). It cannot point to a mixture such as (A′, B). Since the disk can guarantee to write only a single block atomically, the application cannot simply write (A′, B′) on top of (A, B) because that involves writing two blocks. Should the system fail during the updates, doing so could leave the data in an invalid state.

To solve this problem, the application writes the new data to the file in a specific order. First, it writes the new data (A′, B′) to a new section of the file, waiting until the data is written to the disk. Once (A′, B′) are known to be on the disk, it atomically updates the index block to point to the new data. The old blocks (A, B) are now obsolete, and the space they consume can be reused. During the update, the file is never in an inconsistent state.

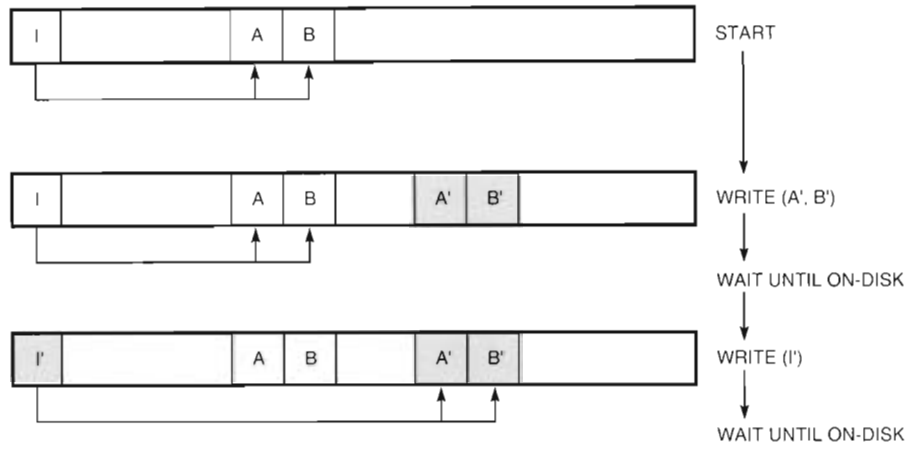


Figure 3
Example of a Careful Write

Write-behind Caching

A careful write policy relies totally on being able to control the order of writes to the disk. This cannot be achieved on a write-back cache because the write-back method does not preserve the order of write requests. Reordering writes in a write-back cache would risk corrupting the data that applications using careful writing were seeking to protect. This is unfortunate because the performance benefits of deferring the write to the disk are compatible with a careful write policy. Careful writing does not need to know when the data is written to the disk, only the order it is written.

To allow these applications to gain the performance of the write-back cache but still protect their data on disk, the Spiralog file system uses a variation on write-back caching known as write-behind caching. Introduced in the Echo file system, write-behind caching is essentially write-back caching with ordering guarantees.[3] The cache allows the application to specify which writes must be ordered and the order in which they must be written to the disk.

This is achieved by providing the barrier primitive to applications. Barrier defines an order or dependency between write operations. For example, consider the diagram in Figure 4: Here, writes are represented as a time-ordered queue, with later writes being added to the tail. In the example, the application issues the writes in the order 1,2,3,4. Without a barrier, the cache could write the data to the disk in any order (for example, 1,3,4,2). If a barrier is placed in the write queue, it specifies to the cache that all writes prior to the barrier must be written to the disk before (or atomically with) any write requests after it. In the example, if a barrier is placed after the second write, the cache file system guarantees that writes 1 and 2 will be written to the disk before writes 3 and 4. Writes 1 and 2 may still be written in any order, as could writes 3 and 4, but 3 and 4 will be written after 1 and 2.

A careful write policy can easily be implemented on a write-behind cache. As shown in Figure 5, the application would use barriers to control the write ordering. Two barriers are required. The first (B1) comes after the writes of the new data (A′, B′). The second (B2) is placed after the index update I′. B1 is required to ensure that the new data is on the disk before the index block is updated. B2 ensures that the index block is updated before any subsequent write requests.

The use of barriers avoids the need to wait for I/Os to reach the disk, improving CPU utilization. In addition, the Spiralog file system allows amalgamation of superseding writes between barriers, reducing the number of requests being written to the disk.

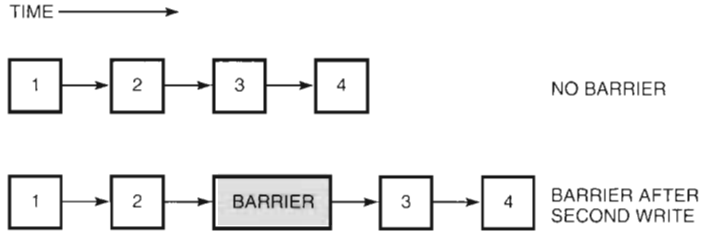


Figure 4
Barrier Insertion in Write Queue

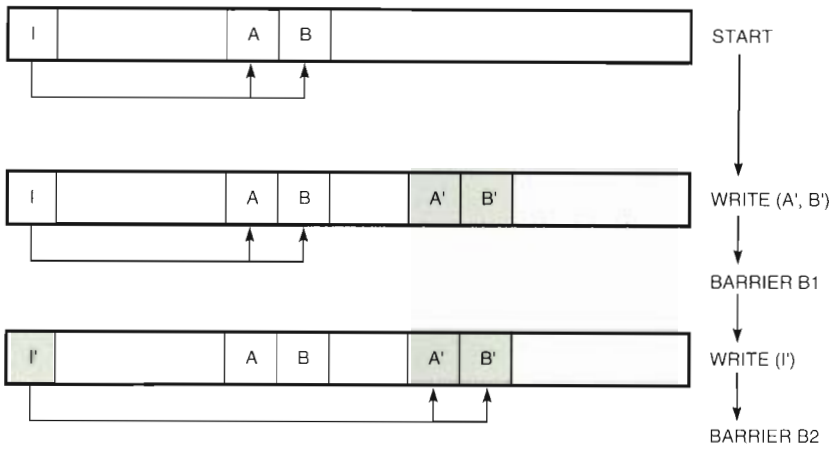


Figure 5
Example of a Careful Write Using Barrier

Internally, the Spiralog file system allows barriers to be placed between any two write operations, even if they are to different files. The Spiralog file system uses this to build its own careful write policy for all changes to files, including metadata changes. This guarantees that the file system is always consistent and gives write-back performance on changes to file metadata as well as data. One major advantage is that the Spiralog file system does not require a disk repair utility such as the UNIX system's fsck to rebuild the file system following a system failure.

Barriers are used internally in several places to preserve the order of updates to the file system metadata. For example, when a file is extended, the allocation of new blocks must be written to the disk before any subsequent data writes to the newly allocated region. A barrier is placed immediately after the write request to update the file length.

Barriers are also used during complex file operations such as a file create. These complex operations frequently update shared resources such as parent directories. The barriers prevent updates to these shared objects, avoiding the risk of corruption due to the updates being reordered by the cache.

At the application level, the Spiralog file system provides the barrier function only within a file. It is not possible to order writes between files. This was sufficient to allow RMS (described in the section OpenVMS File System Environment) to exploit the performance of write-behind caching on most of its file organizations. RMS was enhanced to use barriers in its own careful write policy, which ensures the consistency of complex file organizations, such as indexed files, even when they are subject to write-behind caching. Since the majority of OpenVMS applications access the file system through RMS, gaining write-behind caching on all RMS file organizations provides a significant performance benefit to applications.

Internally, the Spiralog file system supports barriers between files. The decision to support barriers within a file was made to limit the complexity of interface changes, in the belief that a cross-file barrier was of little use to RMS. In retrospect, this proved to be wrong. Some key RMS file organizations use secondary files to hold journal records for the main application file. These file organizations cannot express the order in which updates to the two files should reach the disk, and so are precluded from using write-behind caching.

Application-level Caching Policies

The main problem with the barrier primitive is its requirement that the application express the dependencies to the file system. Although this is unavoidable, it means that the application has to change if it wishes to safely exploit write-behind caching. Clearly, many applications were not going to make these changes. In addition, some applications have on-disk consistency requirements that tie them to a write-through environment.

The file system emulation layer provides additional support for these types of applications by exposing three caching policies to applications. The policies are stored as permanent attributes of the file. By default, when the file is opened by the file system, the permanent caching policy is used on all write requests.

The three policies are described as follows:

1. Write-through caching policy. This policy provides applications with the standard write-through behavior provided by the Files-11 file system. Each write request is flushed to the disk before the application request returns. If an application needs to know what data is on the disk at all times, it should use write-through caching.
2. Write-behind caching policy. A pure write-behind cache provides the highest level of performance. Dirty data is not flushed to the disk when the file is

closed. The semantics of full write-behind caching are best suited to applications that can easily regenerate lost data at any time. Temporary files from a compiler are a good example. Should the system fail, the compilation can be restarted without any loss of data.

3. Flush-on-close caching policy. The flush-on-close policy provides a restricted level of write-behind caching for applications. Here, all updates to the file are treated as write behind, but when the file is closed, all changes are forced to the disk. This gives the performance of write-behind but, in addition, provides a known point when the data is on the disk. This form of caching is particularly suitable for applications that can easily re-create data in the event of a system crash but need to know that data is on the disk at a specific time. For example, a mail store-and-forward system receiving an incoming message must know the data is on the disk when it acknowledges receipt of the message to the forwarder. Once the acknowledgment is sent, the message has been formally passed on, and the forwarder may delete its copy. In this example, the data need not be on the disk until that acknowledgment is sent, because that is the point at which the message receipt is committed. Should the system fail before the acknowledgment is sent, all dirty data in the cache would be lost. In that event, the sender can easily re-create the data by sending the message again.

Figure 6 shows the results of a performance comparison of the three caching policies. The test was run on a dual-CPU DEC 7000 Alpha system with 384 megabytes of memory on a RAID-5 disk. The test repeated the following sequence for the different file sizes.

1. Create and open a file of the required size and set its caching policy.
2. Write data to the whole file in 1,024-byte I/Os.
3. Close the file.
4. Delete the file.

With small files, the number of file operations (create, close, delete) dominates. The leftmost side of the graph therefore shows the time per operation for file operations. With time, the files increase in size, and the data I/Os become prevalent. Hence, the rightmost side of Figure 6 is displaying the time per operation for data I/Os.

Figure 6 clearly shows that an ordered write-behind cache provides the highest performance of the three caching models. For file operations, the write-behind cache is almost 30 percent faster than the write-through cache. Data operations are approximately three times faster than the corresponding operation with write-through caching.

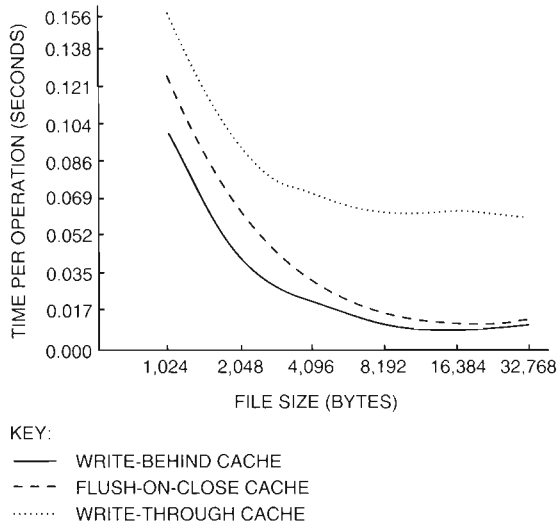


Figure 6
Performance Comparison of Caching Policies

Summary and Conclusions

The task of integrating a log-structured file system into the OpenVMS environment was a significant challenge for the Spiralog project members. Our approach of carefully determining the interface to emulate and the level of compatibility was important to ensure that the majority of applications worked unchanged.

We have shown that an existing update-in-place file system can be replaced by a log-structured file system. Initial effort in the analysis of application usage furnished information on interface compatibility. Most file system operations can be provided through a file system emulation layer. Where necessary, new interfaces were provided for applications to replace their direct knowledge of the Files-11 file system.

File system operation tracing and functional analysis of the Files-11 file system proved to be the most useful techniques to establish interface compatibility. Application compatibility far exceeds the level expected when the project was started. A majority of people use the Spiralog file system volumes without noticing any change in their application's behavior.

Careful write policies rely on the order of updates to the disk. Since write-back caches reorder write requests, applications using careful writing have been unable to take advantage of the significant improvements in write performance given by write-back caching. The Spiralog file system solves this problem by providing ordered write-back caching, known as write-behind. The write-behind cache allows applications to control the order of writes to the disk through a primitive called barrier.

Using barriers, applications can build careful write policies on top of a write-behind cache, gaining all the performance of write-back caching without risking

data integrity. A write-behind cache also allows the file system itself to gain write-back performance on all file system operations. Since many file system operations are themselves quickly superseded, using write-behind caching prevents many file system operations from ever reaching the disk. Barriers also allow the file system to protect the on-disk file system consistency by implementing its own careful write policy, avoiding the need for disk repair utilities.

The barrier primitive provided a way to get write-through semantics within a file for those applications relying on careful write policies. Changing RMS to use the barrier primitive allowed the Spiralog file system to support write-behind caching as the default policy on all file types in the OpenVMS environment.

Acknowledgments

The development of the Spiralog file system involved the help and support of many individuals. We would like to acknowledge Ian Pattison, in particular, who developed the Spiralog cache. We also want to thank Cathy Foley and Jim Johnson for their help throughout the project, and Karen Howell, Morag Currie, and all those who helped with this paper. Finally, we are very grateful to Andy Goldstein, Stu Davidson, and Tom Speer for their help and advice with the Spiralog integration work.

References

1. J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical Journal*, vol. 8, no. 2 (1996, this issue): 5–14.

2. C. Whitaker, S. Bayley, and R. Widdowson, "Design of the Server for the Spiralog File System," *Digital Technical Journal*, vol. 8, no. 2 (1996, this issue): 15–31.

3. R. Green, A. Baird, and J. Davies, "Designing a Fast, On-line Backup System for a Log-structured File System," *Digital Technical Journal*, vol. 8, no. 2 (1996, this issue): 32–45.

4. A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, "The Echo Distributed File System," Digital Systems Research Center, Research Report 111 (September 1993).

5. *OpenVMS I/O User's Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1988).

6. R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures* (Newton, Mass.: Digital Press, 1994).

7. S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of Summer USENIX Conference*, Atlanta, Ga. (1986): 238–247.

8. K. McCoy, *VMS File System Internals* (Burlington, Mass.: Digital Press, 1990).

Biographies

Mark A. Howell
Mark Howell is an engineering manager in the OpenVMS Engineering Group in Livingston, U.K. Mark was the project leader for Spiralog and wrote some of the product code. He is now managing the follow-on releases to Spiralog version 1.0. In previous projects, Mark contributed to Digital's DECdtm distributed transaction manager, DECdfs distributed file system, and the Alpha port of OpenVMS. Prior to joining Digital, Mark worked on flight simulators and flight software for British Aerospace. Mark received a B.Sc. (honours) in marine biology and biochemistry from Bangor University, Wales. He is one of the rare people who still like interactive fiction (the stuff you have to type, instead of the stuff you point a mouse at.)

Julian M. Palmer
A senior software engineer, Julian Palmer is a member of the OpenVMS Engineering Group in Livingston, Scotland. He is currently working on file system caching for OpenVMS. Prior to his work in file systems, Julian contributed to OpenVMS interprocess communication. Julian joined Digital in 1989 after completing his B.Sc. (honours) in computer science from Edinburgh University.

Michael S. Harvey
Leonard S. Szubowicz

Extending OpenVMS for 64-bit Addressable Virtual Memory

The OpenVMS operating system recently extended its 32-bit virtual address space to exploit the Alpha processor's 64-bit virtual addressing capacity while ensuring binary compatibility for 32-bit nonprivileged programs. This 64-bit technology is now available both to OpenVMS users and to the operating system itself. Extending the virtual address space is a fundamental evolutionary step for the OpenVMS operating system, which has existed within the bounds of a 32-bit address space for nearly 20 years. We chose an asymmetric division of virtual address extension that allocates the majority of the address space to applications by minimizing the address space devoted to the kernel. Significant scaling issues arose with respect to the kernel that dictated a different approach to page table residency within the OpenVMS address space. The paper discusses key scaling issues, their solutions, and the resulting layout of the 64-bit virtual address space.

The OpenVMS Alpha operating system initially supported a 32-bit virtual address space that maximized compatibility for OpenVMS VAX users as they ported their applications from the VAX platform to the Alpha platform. Providing access to the 64-bit virtual memory capability defined by the Alpha architecture was always a goal for the OpenVMS operating system. An early consideration was the eventual use of this technology to enable a transition from a purely 32-bit-oriented context to a purely 64-bit-oriented native context. OpenVMS designers recognized that such a fundamental transition for the operating system, along with a 32-bit VAX compatibility mode support environment, would take a long time to implement and could seriously jeopardize the migration of applications from the VAX platform to the Alpha platform. A phased approach was called for, by which the operating system could evolve over time, allowing for quicker time-to-market for significant features and better, more timely support for binary compatibility.

In 1989, a strategy emerged that defined two fundamental phases of OpenVMS Alpha development. Phase 1 would deliver the OpenVMS Alpha operating system initially with a virtual address space that faithfully replicated address space as it was defined by the VAX architecture. This familiar 32-bit environment would ease the migration of applications from the VAX platform to the Alpha platform and would ease the port of the operating system itself. Phase 1, the OpenVMS Alpha version 1.0 product, was delivered in 1992.[1]

For Phase 2, the OpenVMS operating system would successfully exploit the 64-bit virtual address capacity of the Alpha architecture, laying the groundwork for further evolution of the OpenVMS system. In 1989, strategists predicted that Phase 2 could be delivered approximately three years after Phase 1. As planned, Phase 2 culminated in 1995 with the delivery of OpenVMS Alpha version 7.0, the first version of the OpenVMS operating system to support 64-bit virtual addressing.

This paper discusses how the OpenVMS Alpha Operating System Development group extended the OpenVMS virtual address space to 64 bits. Topics covered include compatibility for existing applications, the options for extending the address space, the

strategy for page table residency, and the final layout of the OpenVMS 64-bit virtual address space. In implementing support for 64-bit virtual addresses, designers maximized privileged code compatibility; the paper presents some key measures taken to this end and provides a privileged code example. A discussion of the immediate use of 64-bit addressing by the OpenVMS kernel and a summary of the work accomplished conclude the paper.

Compatibility Constraints

Growing the virtual address space from a 32-bit to a 64-bit capacity was subject to one overarching consideration: compatibility. Specifically, any existing nonprivileged program that could execute prior to the introduction of 64-bit addressing support, even in binary form, must continue to run correctly and unmodified under a version of the OpenVMS operating system that supports a 64-bit virtual address space.

In this context, a nonprivileged program is one that is coded only to stable interfaces that are not allowed to change from one release of the operating system to another. In contrast, a privileged program is defined as one that must be linked against the OpenVMS kernel to resolve references to internal interfaces and data structures that may change as the kernel evolves.

The compatibility constraint dictates that the following characteristics of the 32-bit virtual address space environment, upon which a nonprivileged program may depend, must continue to appear unchanged.[2]

- The lower-addressed half (2 gigabytes [GB]) of virtual address space is defined to be private to a given process. This process-private space is further divided into two 1-GB spaces that grow toward each other.
  1. The lower 1-GB space is referred to as P0 space. This space is called the program region, where user programs typically reside while running.
  2. The higher 1-GB space is referred to as P1 space. This space is called the control region and contains the stacks for a given process, process-permanent code, and various process-specific control cells.
- The higher-addressed half (2 GB) of virtual address space is defined to be shared by all processes. This shared space is where the OpenVMS operating system kernel resides. Although the VAX architecture divides this space into a pair of separately named 1-GB regions (S0 space and S1 space), the OpenVMS Alpha operating system makes no material distinction between the two regions and refers to them collectively as S0/S1 space.

Figure 1 illustrates the 32-bit virtual address space layout as implemented by the OpenVMS Alpha operating system prior to version 7.0.[1] An interesting mechanism can be seen in the Alpha implementation of this address space. The Alpha architecture defines 32-bit load operations such that values (possibly pointers) are sign extended from bit 31 as they are loaded into registers.[3] This facilitates address calculations with results that are 64-bit, sign-extended forms of the original 32-bit pointer values. For all P0 or P1 space addresses, the upper 32 bits of a given pointer in a register will be written with zeros. For all S0/S1 space addresses, the upper 32 bits of a given pointer in a register will be written with ones. Hence, on the Alpha platform, the 32-bit virtual address space actually exists as the lowest 2 GB and highest 2 GB of the entire 64-bit virtual address space. From the perspective of a program using only 32-bit pointers, these regions appear to be contiguous, exactly as they appeared on the VAX platform.

Superset Address Space Options

We considered the following three general options for extending the address space beyond the current 32-bit limits. The degree to which each option would relieve the address space pressure being felt by applications and the OpenVMS kernel itself varied significantly, as did the cost of implementing each option.

1. Extension of shared space
2. Extension of process-private space
3. Extension of both shared space and process-private space

The first option considered was to extend the virtual address boundaries for shared space only. Process-private space would remain limited to its current size of 2 GB. If processes needed access to a huge amount of virtual memory, the memory would have to have been created in shared space where, by definition, all processes would have access to it. This option's chief advantage was that no changes were required in the complex memory management code that specifically supports process-private space. Choosing this option would have minimized the time-to-market for delivering some degree of virtual address extension, however limited it would be. Avoiding any impact to process-private space was also its chief disadvantage. By failing to extend process-private space, this option proved to be generally unappealing to our customers. In addition, it was viewed as a makeshift solution that we would be unable to discard once process-private space was extended at a future time.

The second option was to extend process-private space only. This option would have delivered the highly desirable 64-bit capacity to processes but would not have extended shared space beyond its current 32-bit boundaries. The option presumed to reduce the degree of change in the kernel, hence maximizing

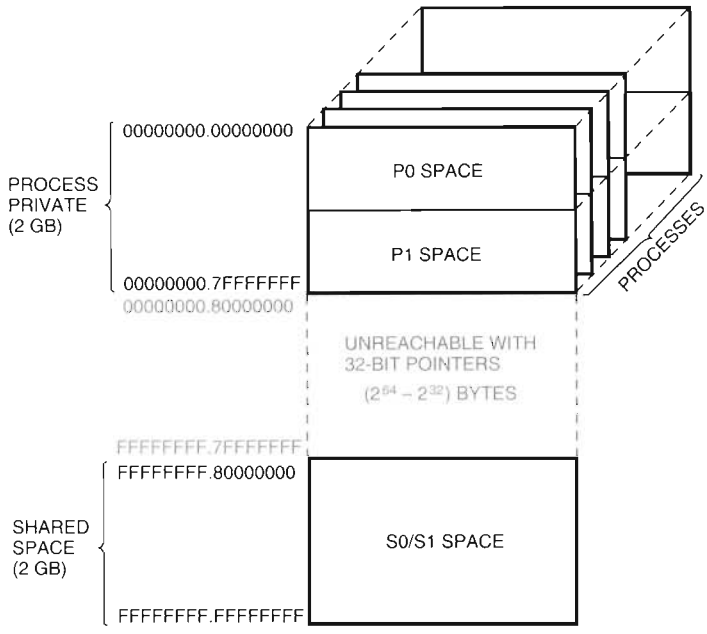


Figure 1
OpenVMS Alpha 32-bit Virtual Address Space

privileged code compatibility and ensuring faster time-to-market. However, analysis of this option showed that there were enough significant portions of the kernel requiring change that, in practice, very little additional privileged code compatibility, such as for drivers, would be achievable. Also, this option did not address certain important problems that are specific to shared space, such as limitations on the kernel's capacity to manage ever-larger, very large memory (VLM) systems in the future.

We decided to pursue the option of a flat, superset 64-bit virtual address space that provided extensions for both the shared and the process-private portions of the space that a given process could reference. The new, extended process-private space, named P2 space, is adjacent to P1 space and extends toward higher virtual addresses.[4,5] The new, extended shared space, named S2 space, is adjacent to S0/S1 space and extends toward lower virtual addresses. P2 and S2 spaces grow toward each other.

A remaining design problem was to decide where P2 and S2 would meet in the address space layout. A simple approach would split the 64-bit address space exactly in half, symmetrically scaling up the design of the 32-bit address space already in place. (The address space is split in this way by the Digital UNIX operating system.[3]) This solution is easy to explain because, on the one hand, it extends the 32-bit convention that the most significant address bit can be treated as a sign bit, indicating whether an address is private or shared. On the other hand, it allocates fully one-half the available virtual address space to the operating system kernel, whether or not this space is needed in its entirety.

The pressure to grow the address space generally stems from applications rather than from the operating system itself. In response, we implemented the 64-bit address space with a boundary that floats between the process-private and shared portions. The operating system configures at bootstrap only as much virtual address space as it needs (never more than 50 percent of the whole). At this point, the boundary becomes fixed for all processes, with the majority of the address space available for process-private use.

A floating boundary maximizes the virtual address space that is available to applications; however, using the sign bit to distinguish between process-private pointers and shared-space pointers continues to work only for 32-bit pointers. The location of the floating boundary must be used to distinguish between 64-bit process-private and shared pointers. We believed that this was a minor trade-off in return for realizing twice as much process-private address space as would otherwise have been achieved.

Page Table Residency

While pursuing the 64-bit virtual address space layout, we grappled with the issue of where the page tables that map the address space would reside within that address space. This section discusses the page table structure that supports the OpenVMS operating system, the residency issue, and the method we chose to resolve this issue.

Virtual Address–to–Physical Address Translation

The Alpha architecture allows an implementation to choose one of the following four page sizes: 8 kilobytes (KB), 16 KB, 32 KB, or 64 KB.[3] The architecture also defines a multilevel, hierarchical page table structure for virtual address-to-physical address (VA–to–PA) translations. All OpenVMS Alpha platforms have implemented a page size of 8 KB and three levels in this page table structure. Although throughout this paper we assume a page size of 8 KB and three levels in the page table hierarchy, no loss of generality is incurred by this assumption.

Figure 2 illustrates the VA–to–PA translation sequence using the multilevel page table structure.

1. The page table base register (PTBR) is a per-process pointer to the highest level (L1) of that process' page table structure. At the highest level is one 8-KB page (L1PT) that contains 1,024 page table entries (PTEs) of 8 bytes each. Each PTE at the highest page table level (that is, each L1PTE) maps a page table page at the next lower level in the translation hierarchy (the L2PTs).
2. The Segment 1 bit field of a given virtual address is an index into the L1PT that selects a particular L1PTE, hence selecting a specific L2PT for the next stage of the translation.
3. The Segment 2 bit field of the virtual address then indexes into that L2PT to select an L2PTE, hence selecting a specific L3PT for the next stage of the translation.
4. The Segment 3 bit field of the virtual address then indexes into that L3PT to select an L3PTE, hence selecting a specific 8-KB code or data page.
5. The byte-within-page bit field of the virtual address then selects a specific byte address in that page.

An Alpha implementation may increase the page size and/or number of levels in the page table hierarchy, thus mapping greater amounts of virtual space up to the full 64-bit amount. The assumed combination of 8-KB page size and three levels of page table allows the system to map up to 8 terabytes (TB) (i.e., 1,024 × 1,024 × 1,024 × 8 KB = 8 TB) of virtual memory for a single process.

To map the entire 8-TB address space available to a single process requires up to 8 GB of PTEs (i.e., 1,024 × 1,024 × 1,024 × 8 bytes = 8 GB). This fact alone presents a serious sizing issue for the OpenVMS operating system. The 32-bit page table residency model that the OpenVMS operating system ported from the VAX platform to the Alpha platform does not have the capacity to support such large page tables.

Page Tables: 32-bit Residency Model

We stated earlier that materializing a 32-bit virtual address space as it was defined by the VAX architecture would ease the effort to port the OpenVMS operating

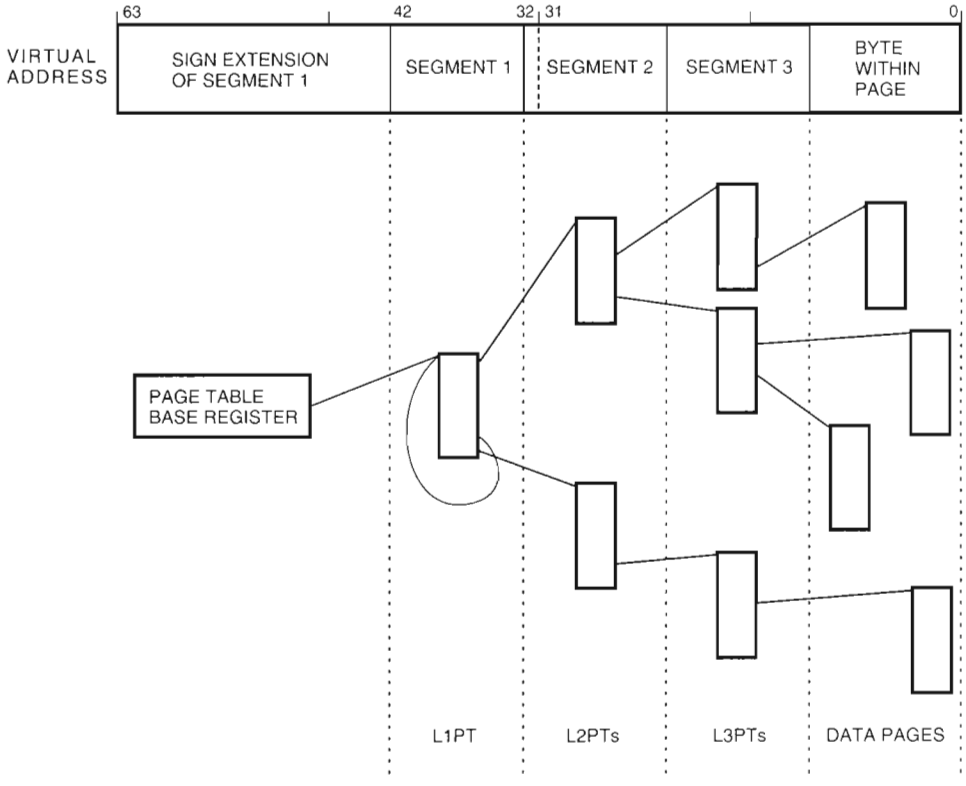


Figure 2
Virtual Address–to–Physical Address Translation

system from the VAX platform to the Alpha platform. A concrete example of this relates to page table residency in virtual memory.

The VAX architecture defines, for a given process, a P0 page table and a P1 page table that map that process' P0 and P1 spaces, respectively.[2] The architecture specifies that these page tables are to be located in S0/S1 shared virtual address space. Thus, the page tables in virtual memory are accessible regardless of which process context is currently active on the system.

The OpenVMS VAX operating system places a given process' P0 and P1 page tables, along with other per-process data, in a fixed-size data structure called a balance slot. An array of such slots exists within S0/S1 space with each memory-resident process being assigned to one of these slots.

This page table residency design was ported from the VAX platform to the Alpha platform.[1] The L3PTs needed to map P0 and P1 spaces and the one L2PT needed to map those L3PTs are all mapped into a balance slot in S0/S1 space. (To conserve virtual memory, the process' L1PT is not mapped into S0/S1 space.) The net effect is illustrated in Figure 3.

The VAX architecture defines a separate, physically resident system page table (SPT) that maps S0/S1 space. The SPT was explicitly mapped into S0/S1 space by the OpenVMS operating system on both the VAX and the Alpha platforms.

Only 2 megabytes (MB) of level 3 PT space is required to map all of a given process' P0 and P1 spaces. This balance slot design reasonably accommodates a large number of processes, all of whose P0 and P1 page tables simultaneously reside within those balance slots in S0/S1 shared space.

This design cannot scale to support a 64-bit virtual address space. Measured in terms of gigabytes per process, the page tables required to map such an enormous address space are too big for the balance slots, which are constrained to exist inside the 2-GB S0/S1 space. The designers had to find another approach for page table residency.

Self-mapping the Page Tables

Recall from earlier discussion that on today's Alpha implementations, the page size is 8 KB, three levels of translation exist within the hierarchical page table structure, and each page table page contains 1,024 PTEs. Each L1PTE maps 8 GB of virtual memory. Eight gigabytes of PT space allows all 8 TB of virtual memory that this implementation can materialize to be mapped.

An elegant approach to mapping a process' page tables into virtual memory is to self-map them. A single PTE in the highest-level page table page is set to map that page table page. That is, the selected L1PTE contains the page frame number of the level 1 page table page that contains that L1PTE.

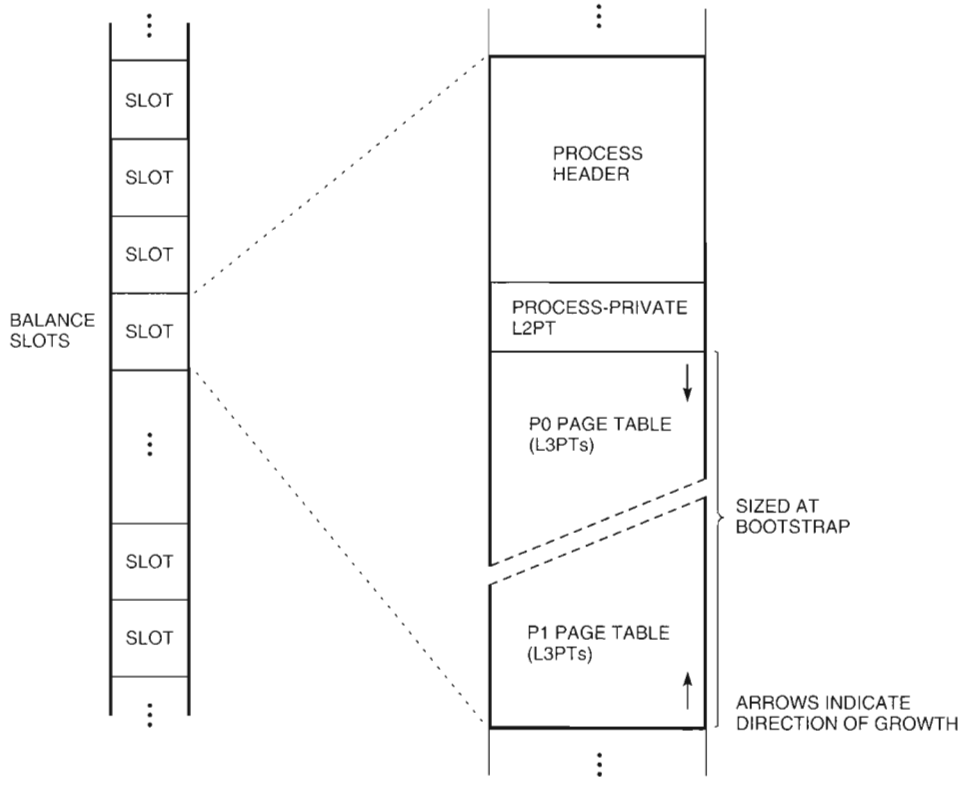


Figure 3
32-bit Page Tables in S0/S1 Space (Prior to OpenVMS Alpha Version 7.0)

The effect of this self-mapping on the VA–to–PA translation sequence (shown in Figure 2) is subtle but important.

- For those virtual addresses with a Segment 1 bit field value that selects the self-mapper L1PTE, step 2 of the VA–to–PA translation sequence reselects the L1PT as the effective L2PT (L2PT′) for the next stage of the translation.
- Step 3 indexes into L2PT′ (the L1PT) using the Segment 2 bit field value to select an L3PT′.
- Step 4 indexes into L3PT′ (an L2PT) using the Segment 3 bit field value to select a specific data page.
- Step 5 indexes into that data page (an L3PT) using the byte-within-page bit field of the virtual address to select a specific byte address within that page.

When step 5 of the VA–to–PA translation sequence is finished, the final page being accessed is itself one of the level 3 page table pages, not a page that is mapped by a level 3 page table page. The self-map operation places the entire 8-GB page table structure at the end of the VA–to–PA translation sequence for a specific 8-GB portion of the process' address space. This virtual space that contains all of a process' potential page tables is called page table space (PT space).[6]

Figure 4 depicts the effect of self-mapping the page tables. On the left is the highest-level page table page containing a fixed number of PTEs. On the right is the virtual address space that is mapped by that page table page. The mapped address space consists of a collection of identically sized, contiguous address range sections, each one mapped by a PTE in the corresponding position in the highest-level page table page. (For clarity, lower levels of the page table structure are omitted from the figure.)

Notice that L1PTE #1022 in Figure 4 was chosen to map the high-level page table page that contains that PTE. (The reason for this particular choice will be explained in the next section. Theoretically, any one

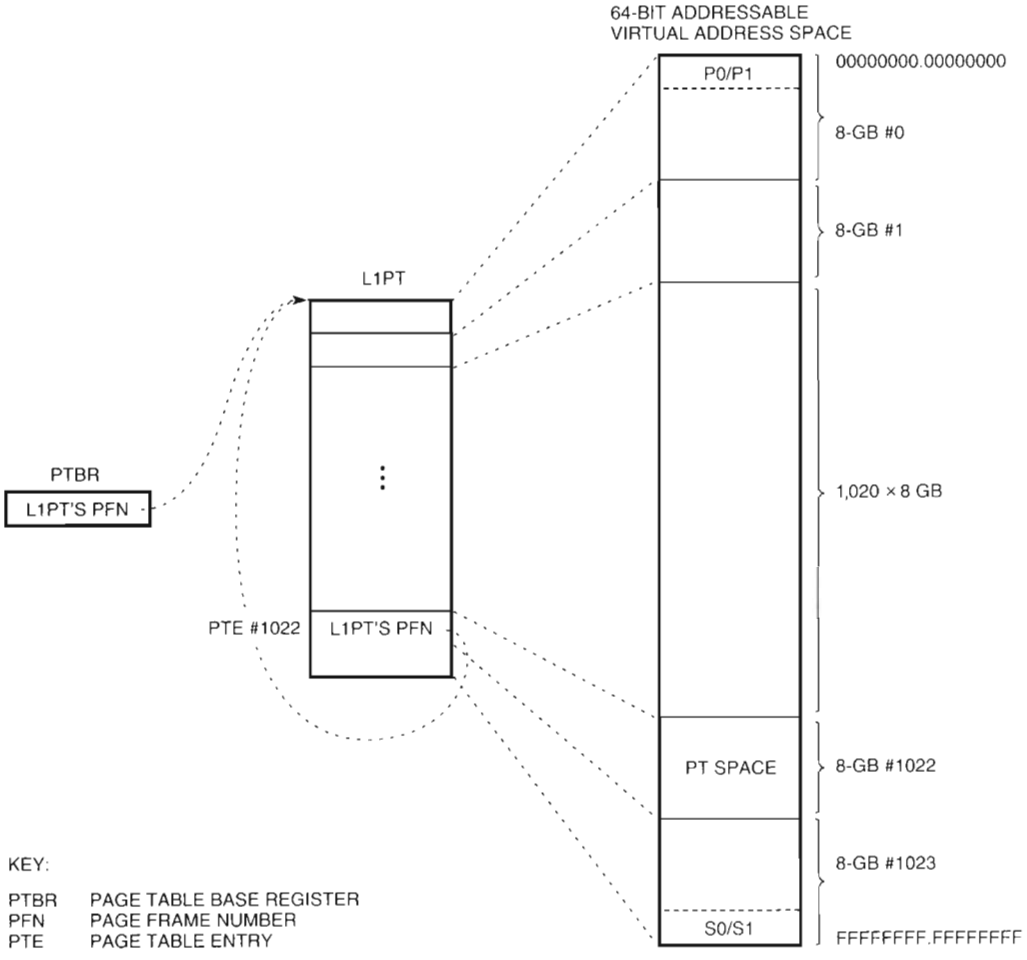


Figure 4
Effect of Page Table Self-map

of the L1PTEs could have been chosen as the self-mapper.) The section of virtual memory mapped by the chosen L1PTE contains the entire set of page tables needed to map the available address space of a given process. This section of virtual memory is PT space, which is depicted on the right side of Figure 4 in the 1,022d 8-GB section in the materialized virtual address space.

The base address for this PT space incorporates the index of the chosen self-mapper L1PTE (1,022 = 3FE(16)) as follows (see Figure 2):

Segment 1 bit field = 3FE
Segment 2 bit field = 0
Segment 3 bit field = 0
Byte within page = 0,

which result in

VA = FFFFFFFC.00000000
(also known as PT_Base).

Figure 5 illustrates the exact contents of PT space for a given process. One can observe the positional effect of choosing a particular high-level PTE to self-map the page tables even within PT space. In Figure 4, the choice of PTE for self-mapping not only places PT space as a whole in the 1,022d 8-GB section in virtual memory but also means that

- The 1,022d grouping of the lowest-level page tables (L3PTs) within PT space is actually the collection of next-higher-level PTs (L2PTs) that map the other groupings of L3PTs, beginning at

  Segment 1 bit field = 3FE
  Segment 2 bit field = 3FE
  Segment 3 bit field = 0
  Byte within page = 0,

  which result in

  VA = FFFFFFFD.FF000000
  (also known as L2_Base).
- Within that block of L2PTs, the 1,022d L2PT is actually the next-higher-level page table that maps the L2PTs, namely, the L1PT. The L1PT begins at

  Segment 1 bit field = 3FE
  Segment 2 bit field = 3FE
  Segment 3 bit field = 3FE
  Byte within page = 0,

  which result in

  VA = FFFFFFFD.FF7FC000
  (also known as L1_Base).
- Within that L1PT, the 1,022d PTE is the one used for self-mapping these page tables. The address of the self-mapper L1PTE is

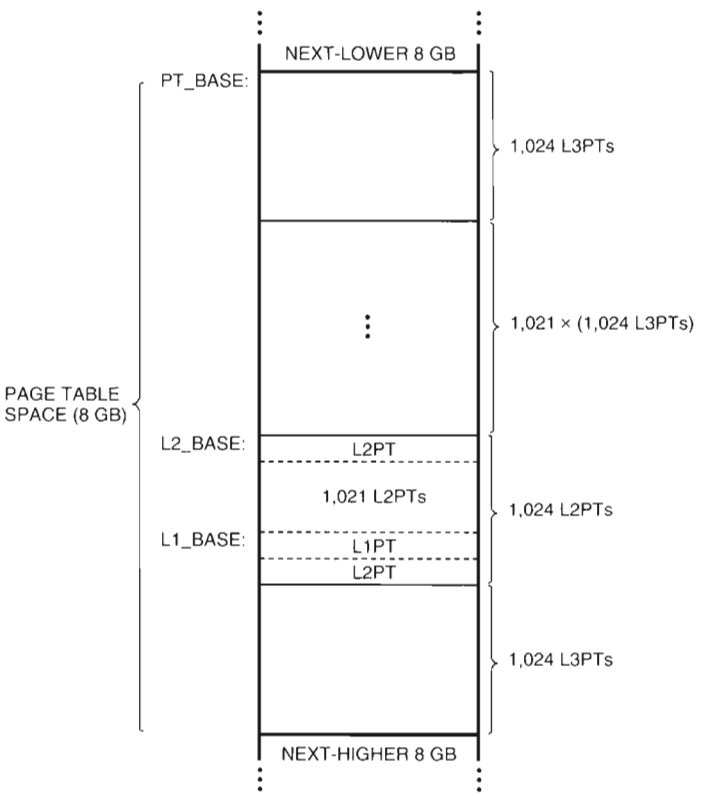


Figure 5
Page Table Space

Segment 1 bit field = 3FE
Segment 2 bit field = 3FE
Segment 3 bit field = 3FE
Byte within page = 3FE × 8

which result in

VA = FFFFFFFD.FF7FDFF0.

This positional correspondence within PT space is preserved should a different high-level PTE be chosen for self-mapping the page tables.

The properties inherent in this self-mapped page table are compelling.

- The amount of virtual memory reserved is exactly the amount required for mapping the page tables, regardless of page size or page table depth. Consider the segment-numbered bit fields of a given virtual address from Figure 2. Concatenated, these bit fields constitute the virtual page number (VPN) portion of a given virtual address.

  The total size of the PT space needed to map every VPN is the number of possible VPNs times 8 bytes, the size of a PTE. The total size of the address space mapped by that PT space is the number of possible VPNs times the page size. Factoring out the VPN multiplier, the difference between these is the page size divided by 8, which is exactly the size of the Segment 1 bit field in the virtual address. Hence, all the space mapped by a single PTE at the highest level of page table is exactly the size required for mapping all the PTEs that could ever be needed to map the process' address space.
- The mapping of PT space involves simply choosing one of the highest-level PTEs and forcing it to self-map.
- No additional system tuning or coding is required to accommodate a more widely implemented virtual address width in PT space. By definition of the self-map effect, the exact amount of virtual address space required will be available, no more and no less.
- It is easy to locate a given PTE. The address of a PTE becomes an efficient function of the address that the PTE maps. The function first clears the byte-within-page bit field of the subject virtual address and then shifts the remaining virtual address bits such that the Segments 1, 2, and 3 bit field values (Figure 2) now reside in the corresponding next-lower bit field positions. The function then writes (and sign extends if necessary) the vacated Segment 1 field with the index of the self-mapper PTE. The result is the address of the PTE that maps the original virtual address. Note that this algorithm also works for addresses within PT space, including that of the self-mapper PTE itself.
- Process page table residency in virtual memory is achieved without imposing on the capacity of shared space. That is, there is no longer a need to map the process page tables into shared space. Such a mapping would be redundant and wasteful.

OpenVMS 64-bit Virtual Address Space

With this page table residency strategy in hand, it became possible to finalize a 64-bit virtual address layout for the OpenVMS operating system. A self-mapper PTE had to be chosen. Consider again the highest level of page table in a given process' page table structure (Figure 4). The first PTE in that page table maps a section of virtual memory that includes P0 and P1 spaces. This PTE was therefore unavailable for use as a self-mapper. The last PTE in that page table maps a section of virtual memory that includes S0/S1 space. This PTE was also unavailable for self-mapping purposes.

All the intervening high-level PTEs were potential choices for self-mapping the page tables. To maximize the size of process-private space, the correct choice is the next-lower PTE than the one that maps the lowest address in shared space.

This choice is implemented as a boot-time algorithm. Bootstrap code first determines the size required for OpenVMS shared space, calculating the corresponding number of high-level PTEs. A sufficient number of PTEs to map that shared space are allocated later from the high-order end of a given process' highest-level page table page. Then the next-lower PTE is allocated for self-mapping that process' page tables. All remaining lower-ordered PTEs are left available for mapping process-private space. In practice, nearly all the PTEs are available, which means that on today's systems, almost 8 TB of process-private virtual memory is available to a given OpenVMS process.

Figure 6 presents the final 64-bit OpenVMS virtual address space layout. The portion with the lower addresses is entirely process-private. The higher-addressed portion is shared by all process address spaces. PT space is a region of virtual memory that lies between the P2 and S2 spaces for any given process and at the same virtual address for all processes.

Note that PT space itself consists of a process-private and a shared portion. Again, consider Figure 5. The highest-level page table page, L1PT, is process-private. It is pointed to by the PTBR. (When a process' context is loaded, or made active, the process' PTBR value is loaded from the process' hardware-privileged context block into the PTBR register, thereby making current the page table structure pointed to by that PTBR and the process-private address space that it maps.)

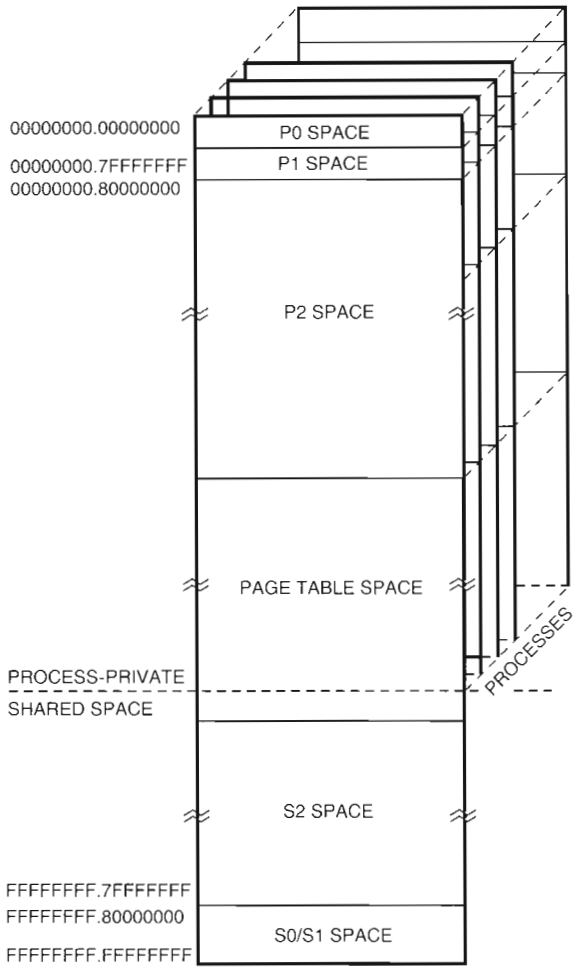


Figure 6
OpenVMS Alpha 64-bit Virtual Address Space

All higher-addressed page tables in PT space are used to map shared space and are themselves shared. They are also adjacent to the shared space that they map. All page tables in PT space that reside at addresses lower than that of the L1PT are used to map process-private space. These page tables are process-private and are adjacent to the process-private space that they map. Hence, the end of the L1PT marks a universal boundary between the process-private portion and the shared portion of the entire virtual address space, serving to separate even the PTEs that map those portions. In Figure 6, the line passing through PT space illustrates this boundary.

A direct consequence of this design is that the process page tables have been privatized. That is, the portion of PT space that is process-private is currently active in virtual memory only when the owning process itself is currently active on the processor. Fortunately, the majority of page table references occur while executing in the context of the owning process. Such references actually are enhanced by the privatization of the process page tables because the mapping function of a virtual address to its PTE is now more efficient.

Privatization does raise a hurdle for certain privileged code that previously could access a process' page tables when executing outside the context of the owning process. With the page tables resident in shared space, such references could be made regardless of which process is currently active. With privatized page tables, additional access support is needed, as presented in the next section.

A final commentary is warranted for the separately maintained system page table. The self-mapped page table approach to supplying page table residency in virtual memory includes the PTEs for any virtual

addresses, whether they are process-private or shared. The shared portion of PT space could serve now as the sole location for shared-space PTEs. Being redundant, the original SPT is eminently discardable; however, discarding the SPT would create a massive compatibility problem for device drivers with their many 32-bit SPT references. This area is one in which an opportunity exists to preserve a significant degree of privileged code compatibility.

Key Measures Taken to Maximize Privileged Code Compatibility

To implement 64-bit virtual address space support, we altered central sections of the OpenVMS Alpha kernel and many of its key data structures. We expected that such changes would require compensating or corresponding source changes in surrounding privileged components within the kernel, in device drivers, and in privileged layered products.

For example, the previous discussion seems to indicate that any privileged component that reads or writes PTEs would now need to use 64-bit-wide pointers instead of 32-bit pointers. Similarly, all system fork threads and interrupt service routines could no longer count on direct access to process-private PTEs without regard to which process happens to be current at the moment.

A number of factors exacerbated the impact of such changes. Since the OpenVMS Alpha operating system originated from the OpenVMS VAX operating system, significant portions of the OpenVMS Alpha operating system and its device drivers are still written in MACRO-32 code, a compiled language on the Alpha platform.[1] Because MACRO-32 is an assembly-level style of programming language, we could not simply change the definitions and declarations of various types and rely on recompilation to handle the move from 32-bit to 64-bit pointers. Finally, there are well over 3,000 references to PTEs from MACRO-32 code modules in the OpenVMS Alpha source pool.

We were thus faced with the prospect of visiting and potentially altering each of these 3,000 references. Moreover, we would need to follow the register lifetimes that resulted from each of these references to ensure that all address calculations and memory references were done using 64-bit operations. We expected that this process would be time-consuming and error prone and that it would have a significant negative impact on our completion date.

Once OpenVMS Alpha version 7.0 was available to users, those with device drivers and privileged code of their own would need to go through a similar effort. This would further delay wide use of the release. For all these reasons, we were well motivated to minimize the impact on privileged code. The next four sections discuss techniques that we used to overcome these obstacles.

Resolving the SPT Problem

A significant number of the PTE references in privileged code are to PTEs within the SPT. Device drivers often double-map the user's I/O buffer into S0/S1 space by allocating and appropriately initializing system page table entries (SPTEs). Another situation in which a driver manipulates SPTEs is in the substitution of a system buffer for a poorly aligned or noncontiguous user I/O buffer that prevents the buffer from being directly used with a particular device. Such code relies heavily on the system data cell MMG$GL_SPTBASE, which points to the SPT.

The new page table design completely obviates the need for a separate SPT. Given an 8-KB page size and 8 bytes per PTE, the entire 2-GB S0/S1 virtual address space range can be mapped by 2 MB of PTEs within PT space. Because S0/S1 resides at the highest addressable end of the 64-bit virtual address space, it is mapped by the highest 2 MB of PT space. The arcs on the left in Figure 7 illustrate this mapping. The PTEs in PT space that map S0/S1 are fully shared by all processes, but they must be referenced with 64-bit addresses.

This incompatibility is completely hidden by the creation of a 2-MB "SPT window" over the 2 MB in PT space (level 3 PTEs) that maps S0/S1 space. The SPT window is positioned at the highest addressable end of S0/S1 space. Therefore, an access through the SPT window only requires a 32-bit S0/S1 address and can obtain any of the PTEs in PT space that map S0/S1 space. The arcs on the right in Figure 7 illustrate this access path.

The SPT window is set up at system initialization time and consumes only the 2 KB of PTEs that are needed to map 2 MB. The system data cell MMG$GL_SPTBASE now points to the base of the SPT window, and all existing references to that data cell continue to function correctly without change.

Providing Cross-process PTE Access for Direct I/O

The self-mapping of the page tables is an elegant solution to the page table residency problem imposed by the preceding design. However, the self-mapped page tables present significant challenges of their own to the I/O subsystem and to many device drivers.

Typically, OpenVMS device drivers for mass storage, network, and other high-performance devices perform direct memory access (DMA) and what OpenVMS calls "direct I/O." These device drivers lock down into physical memory the virtual pages that contain the requester's I/O buffer. The I/O transfer is performed directly to those pages, after which the buffer pages are unlocked, hence the term "direct I/O."

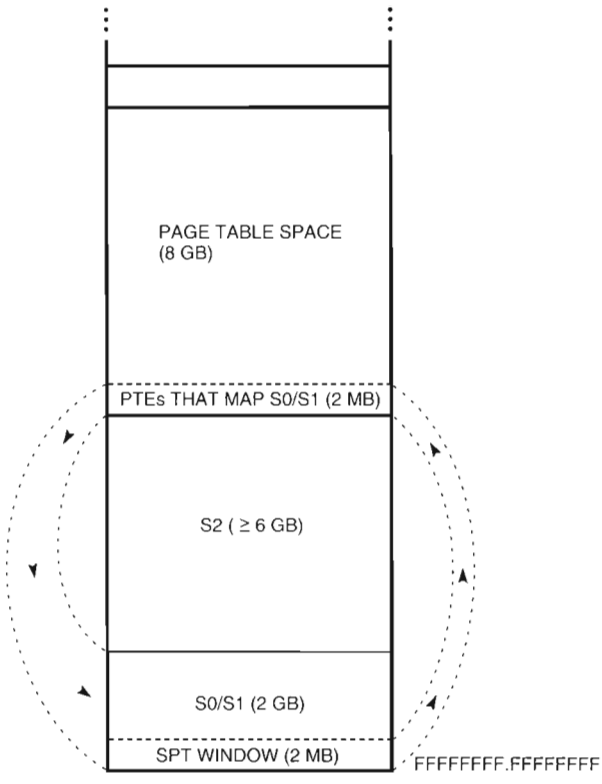


Figure 7
System Page Table Window

The virtual address of the buffer is not adequate for device drivers because much of the driver code runs in system context and not in the process context of the requester. Similarly, a process-specific virtual address is meaningless to most DMA devices, which typically can deal only with the physical addresses of the virtual pages spanned by the buffer.

For these reasons, when the I/O buffer is locked into memory, the OpenVMS I/O subsystem converts the virtual address of the requester's buffer into (1) the address of the PTE that maps the start of the buffer and (2) the byte offset within that page to the first byte of the buffer.

Once the virtual address of the I/O buffer is converted to a PTE address, all references to that buffer are made using the PTE address. This remains the case even if this I/O request and I/O buffer are handed off from one driver to another. For example, the I/O request may be passed from the shadowing virtual disk driver to the small computer systems interface (SCSI) disk class driver to a port driver for a specific SCSI host adapter. Each of these drivers will rely solely on the PTE address and the byte offset and not on the virtual address of the I/O buffer.

Therefore, the number of virtual address bits the requester originally used to specify the address of the I/O buffer is irrelevant. What really matters is the number of address bits that the driver must use to reference a PTE.

These PTE addresses were always within the page tables within the balance set slots in shared S0/S1 space. With the introduction of the self-mapped page tables, a 64-bit address is required for accessing any PTE in PT space. Furthermore, the desired PTE is not accessible using this 64-bit address when the driver is no longer executing in the context of the original requester process. This is called a cross-process PTE access problem.

In most cases, this access problem is solved for direct I/O by copying the PTEs that map the I/O buffer when the I/O buffer is locked into physical memory. The PTEs in PT space are accessible at that point because the requester process context is required in order to lock the buffer. The PTEs are copied into the kernel's heap storage and the 64-bit PT space address is replaced by the address of the PTE copies. Because the kernel's heap storage remains in S0/S1 space, the replacement address is a 32-bit address that is shared by all processes on the system.

This copy approach works because drivers do not need to modify the actual PTEs. Typically, this arrangement works well because the associated PTEs

can fit into dedicated space within the I/O request packet data structure used by the OpenVMS operating system, and there is no measurable increase in CPU overhead to copy those PTEs.

If the I/O buffer is so large that its associated PTEs cannot fit within the I/O request packet, a separate kernel heap storage packet is allocated to hold the PTEs. If the I/O buffer is so large that the cost of copying all the PTEs is noticeable, a direct access path is created as follows:

- The L3PTEs that map the I/O buffer are locked into physical memory.
- Address space within S0/S1 space is allocated and mapped over the L3PTEs that were just locked down.

This establishes a 32-bit addressable shared-space window over the L3PTEs that map the I/O buffer.

The essential point is that one of these methods is selected and employed until the I/O is completed and the buffer is unlocked. Each method provides a 32-bit PTE address that the rest of the I/O subsystem can use transparently, as it has been accustomed to doing, without requiring numerous, complex source changes.

Use of Self-identifying Structures

To accommodate 64-bit user virtual addresses, a number of kernel data structures had to be expanded and changed. For example, asynchronous system trap (AST) control blocks, buffered I/O packets, and timer queue entries all contain various user-provided addresses and parameters that can now be 64-bit addresses. These structures are often embedded in other structures such that a change in one has a ripple effect to a set of other structures.

If these structures changed unconditionally, many scattered source changes would have been required. Yet, at the same time, each of these structures had consumers who had no immediate need for the 64-bit addressing–related capabilities.

Instead of simply changing each of these structures, we defined a new 64-bit-capable variant that can coexist with its traditional 32-bit counterpart. The 64-bit variant's structures are "self-identifying" because they can readily be distinguished from their 32-bit counterparts by examining a particular field within the structure itself. Typically, the 32-bit and 64-bit variants can be intermixed freely within queues and only a limited set of routines need to be aware of the variant types.

Thus, for example, components that do not need 64-bit ASTs can continue to build 32-bit AST control blocks and queue them with the SCH$QAST routine. Similarly, 64-bit AST control blocks can be queued with the same SCH$QAST routine because the AST delivery code was enhanced to support either type of AST control block.

The use of self-identifying structures is also a technique that was employed to compatibly enhance public user-mode interfaces to library routines and the OpenVMS kernel. This topic is discussed in greater detail in "The OpenVMS Mixed Pointer Size Environment."[8]

Limiting the Scope of Kernel Changes

Another key tactic that allowed us to minimize the required source code changes to the OpenVMS kernel came from the realization that full support of 64-bit virtual addressing for all processes does not imply or require exclusive use of 64-bit pointers within the kernel. The portions of the kernel that handled user addresses would for the most part need to handle 64-bit addresses; however, most kernel data structures could remain within the 32-bit addressable S0/S1 space without any limit on user functionality. For example, the kernel heap storage is still located in S0/S1 space and continues to be 32-bit addressable. The Record Management Services (RMS) supports data transfers to and from 64-bit addressable user buffers, but RMS continues to use 32-bit-wide pointers for its internal control structures. We therefore focused our effort on the parts of the kernel that could benefit from internal use of 64-bit addresses (see the section Immediate Use of 64-bit Addressing by the OpenVMS Kernel for examples) and that needed to change to support 64-bit user virtual addresses.

Privileged Code Example—The Swapper

The OpenVMS working set swapper provides an interesting example of how the 64-bit changes within the kernel may impact privileged code.

Only a subset of a process' virtual pages is mapped to physical memory at any given point in time. The OpenVMS operating system occasionally swaps this working set of pages out of memory to secondary storage as a consequence of managing the pool of available physical memory. The entity responsible for this activity is a privileged process called the working set swapper or swapper, for short. Since it is responsible for transferring the working set of a process into and out of memory when necessary, the swapper must have intimate knowledge of the virtual address space of a process including that process' page tables.

Consider the earlier discussion in the section OpenVMS 64-bit Virtual Address Space about how the process' page tables have been privatized as a way to efficiently provide page table residency in virtual memory. A consequence of this design is that while the swapper process is active, the page tables of the process being swapped are not available in virtual memory. Yet, the swapper requires access to those page tables to

do its job. This is an instance of the cross-process PTE access problem mentioned earlier.

The swapper is unable to directly access the page tables of the process being swapped because the swapper's own page tables are currently active in virtual memory. We solved this access problem by revising the swapper to temporarily "adopt" the page tables of the process being swapped. The swapper accomplishes this by temporarily changing its PTBR contents to point to the page table structure for the process being swapped instead of to the swapper's own page table structure. This change forces the PT space of the process being swapped to become active in virtual memory and therefore available to the swapper as it prepares the process to be swapped. Note that the swapper can make this temporary change because the swapper resides in shared space. The swapper does not vanish from virtual memory as the PTBR value is changed. Once the process has been prepared for swapping, the swapper restores its own PTBR value, thus relinquishing access to the target process' PT space contents.

Thus, it can be seen how privileged code with intimate knowledge of OpenVMS memory management mechanisms can be affected by the changes to support 64-bit virtual memory. Also evident is that the alterations needed to accommodate the 64-bit changes are relatively straightforward. Although the swapper has a higher-than-normal awareness of memory management internal workings, extending the swapper to accommodate the 64-bit changes was not particularly difficult.

Immediate Use of 64-bit Addressing by the OpenVMS Kernel

Page table residency was certainly the most pressing issue we faced with regard to the OpenVMS kernel as it evolved from a 32-bit to a 64-bit-capable operating system. Once implemented, 64-bit virtual addressing could be harnessed as an enabling technology for solving a number of other problems as well. This section briefly discusses some prominent examples that serve to illustrate how immediately useful 64-bit addressing became to the OpenVMS kernel.

Page Frame Number Database and Very Large Memory

The OpenVMS Alpha operating system maintains a database for managing individual, physical page frames of memory, i.e., page frame numbers. This database is stored in S0/S1 space. The size of this database grows linearly as the size of the physical memory grows.

Future Alpha systems may include larger memory configurations as memory technology continues to evolve. The corresponding growth of the page frame number database for such systems could consume an unacceptably large portion of S0/S1 space, which has a maximum size of 2 GB. This design effectively restricts the maximum amount of physical memory that the OpenVMS operating system would be able to support in the future.

We chose to remove this potential restriction by relocating the page frame number database from S0/S1 to 64-bit addressable S2 space. There it can grow to support any physical memory size being considered for years to come.

Global Page Table

The OpenVMS operating system maintains a data structure in S0/S1 space called the global page table (GPT). This pseudo–page table maps memory objects called global sections. Multiple processes may map portions of their respective process-private address spaces to these global sections to achieve protected shared memory access for whatever applications they may be running.

With the advent of P2 space, one can easily anticipate a need for orders-of-magnitude-greater global section usage. This usage directly increases the size of the GPT, potentially reaching the point where the GPT consumes an unacceptably large portion of S0/S1 space. We chose to forestall this problem by relocating the GPT from S0/S1 to S2 space. This move allows the configuration of a GPT that is much larger than any that could ever be configured in S0/S1 space.

Summary

Although providing 64-bit support was a significant amount of work, the design of the OpenVMS operating system was readily scalable such that it could be achieved practically. First, we established a goal of strict binary compatibility for nonprivileged applications. We then designed a superset virtual address space that extended both process-private and shared spaces while preserving the 32-bit visible address space to ensure compatibility. To maximize the available space for process-private use, we chose an asymmetric style of address space layout. We privatized the process page tables, thereby eliminating their residency in shared space. The few page table accesses that occurred from outside the context of the owning process, which no longer worked after the privatization of the page tables, were addressed in various ways. A variety of ripple effects stemming from this design were readily solved within the kernel.

Solutions to other scaling problems related to the kernel were immediately possible with the advent of 64-bit virtual address space. Already mentioned was the complete removal of the process page tables from shared space. We also removed the global page table

and the page frame number database from 32-bit addressable to 64-bit addressable shared space. The immediate net effect of these changes was significantly more room in S0/S1 space for configuring more kernel heap storage, more balance slots to be assigned to greater numbers of memory resident processes, etc. We further anticipate use of 64-bit addressable shared space to realize additional benefits of VLM, such as for caching massive amounts of file system data.

Providing 64-bit addressing capacity was a logical, evolutionary step for the OpenVMS operating system. Growing numbers of customers are demanding the additional virtual memory to help solve their problems in new ways and to achieve higher performance. This has been especially fruitful for database applications, with substantial performance improvements already proved possible by the use of 64-bit addressing on the Digital UNIX operating system. Similar results are expected on the OpenVMS system. With terabytes of virtual memory and many gigabytes of physical memory available, entire databases may be loaded into memory at once. Much of the I/O that otherwise would be necessary to access the database can be eliminated, thus allowing an application to improve performance by orders of magnitude, for example, to reduce query time from eight hours to five minutes. Such performance gains were difficult to achieve while the OpenVMS operating system was constrained to a 32-bit environment. With the advent of 64-bit addressing, OpenVMS users now have a powerful enabling technology available to solve their problems.

Acknowledgments

The work described in this paper was done by members of the OpenVMS Alpha Operating System Development group. Numerous contributors put in many long hours to ensure a well-considered design and a high-quality implementation. The authors particularly wish to acknowledge the following major contributors to this effort: Tom Benson, Richard Bishop, Walter Blaschuk, Nitin Karkhanis, Andy Kuehnel, Karen Noel, Phil Norwich, Margie Sherlock, Dave Wall, and Elinor Woods. Thanks also to members of the Alpha languages community who provided extended programming support for a 64-bit environment; to Wayne Cardoza, who helped shape the earliest notions of what could be accomplished; to Beverly Schultz, who provided strong, early encouragement for pursuing this project; and to Ron Higgins and Steve Noyes, for their spirited and unflagging support to the very end.

The following reviewers also deserve thanks for the invaluable comments they provided in helping to prepare this paper: Tom Benson, Cathy Foley, Clair Grant, Russ Green, Mark Howell, Karen Noel, Margie Sherlock, and Rod Widdowson.

References and Notes

1. N. Kronenberg, T. Benson, W. Cardoza, R. Jagannathan, and B. Thomas, "Porting OpenVMS from VAX to Alpha AXP," *Digital Technical Journal,* vol. 4, no. 4 (1992): 111–120.

2. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford, Mass.: Digital Press, 1987).

3. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual,* 2d ed. (Newton, Mass.: Digital Press, 1995).

4. Although an OpenVMS process may refer to P0 or P1 space using either 32-bit or 64-bit pointers, references to P2 space require 64-bit pointers. Applications may very well execute with mixed pointer sizes. (See reference 8 and D. Smith, "Adding 64-bit Pointer Support to a 32-bit Run-time Library," *Digital Technical Journal,* vol. 8, no. 2 [1996, this issue]: 83–95.) There is no notion of an application executing in either a 32-bit mode or a 64-bit mode.

5. Superset system services and language support were added to facilitate the manipulation of 64-bit addressable P2 space.[8]

6. This mechanism has been in place since OpenVMS Alpha version 1.0 to support virtual PTE fetches by the translation buffer miss handler in PALcode. (PALcode is the operating system–specific privileged architecture library that provides control over interrupts, exceptions, context switching, etc.[3]) In effect, this means that the OpenVMS page tables already existed in two virtual locations, namely, S0/S1 space and PT space.

7. The SPT window is more precisely only an S0/S1 PTE window. The PTEs that map S2 space are referenced using 64-bit pointers to their natural locations in PT space and are not accessible through the use of this SPT window. However, because S2 PTEs did not exist prior to the introduction of S2 space, this limitation is of no consequence to contexts that are otherwise restricted to S0/S1 space.

8. T. Benson, K. Noel, and R. Peterson, "The OpenVMS Mixed Pointer Size Environment," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 72–82.

General References

R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures, Version 1.5* (Newton, Mass.: Digital Press, 1994).

OpenVMS Alpha Guide to 64-Bit Addressing (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBCA-TE, December 1995).

OpenVMS Alpha Guide to Upgrading Privileged-Code Applications (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBGA-TE, December 1995).

Biographies

Michael S. Harvey
Michael Harvey joined Digital in 1978 after receiving his B.S.C.S. from the University of Vermont. In 1984, as a member of the OpenVMS Engineering group, he participated in new processor support for VAX multiprocessor systems and helped develop OpenVMS symmetric multiprocessing (SMP) support for these systems. He received a patent for this work. Mike was an original member of the RISCy-VAX task force, which conceived and developed the Alpha architecture. Mike led the project that ported the OpenVMS Executive from the VAX to the Alpha platform and subsequently led the project that designed and implemented 64-bit virtual addressing support in OpenVMS. This effort led to a number of patent applications. As a consulting software engineer, Mike is currently working in the area of infrastructure that supports the Windows NT/OpenVMS Affinity initiative.

Leonard S. Szubowicz
Leonard Szubowicz is a consulting software engineer in Digital's OpenVMS Engineering group. Currently the technical leader for the OpenVMS I/O engineering team, he joined Digital Software Services in 1983. As a member of the OpenVMS 64-bit virtual addressing project team, Lenny had primary responsibility for I/O and driver support. Prior to that, he was the architect and project leader for the OpenVMS high-level language device driver project, contributed to the port of the OpenVMS operating system to the Alpha platform, and was project leader for RMS Journaling. Lenny is a coauthor of *Writing OpenVMS Alpha Device Drivers in C*, which was recently published by Digital Press.

The OpenVMS Mixed Pointer Size Environment

Thomas R. Benson
Karen L. Noel
Richard E. Peterson

A central goal in the implementation of 64-bit addressing on the OpenVMS operating system was to provide upward-compatible support for applications that use the existing 32-bit address space. Another guiding principle was that mixed pointer sizes are likely to be the rule rather than the exception for applications that use 64-bit address space. These factors drove several key design decisions in the OpenVMS Calling Standard and programming interfaces, the DEC C language support, and the system services support. For example, self-identifying 64-bit descriptors were designed to ease development when mixed pointer sizes are used. DEC C support makes it easy to mix pointer sizes and to recompile for uniform 32- or 64-bit pointer sizes. OpenVMS system services remain fully upward compatible, with new services defined only where required or to enhance the usability of the huge 64-bit address space. This paper describes the approaches taken to support the mixed pointer size environment in these areas. The issues and rationale behind these OpenVMS and DEC C solutions are presented to encourage others who provide library interfaces to use a consistent programming interface approach.

Support for 64-bit virtual addressing on the OpenVMS Alpha operating system, version 7.0, has vastly increased the amount of virtual address space available for application use.[1] At the same time, fully compatible support for applications that use only 32-bit addresses (also called *pointers*) has been preserved.

An application that mixes 32-bit and 64-bit pointer sizes operates in a *mixed pointer size environment*. Mixed pointer size applications were the design center for the initial implementation of 64-bit support in the OpenVMS operating system. This paper discusses the reasons why mixing pointer sizes is expected to be a common practice and describes the design of operating system and language features that are provided to ease programming in this mixed pointer size environment.

Reasons for Mixed Pointer Sizes

To use 64-bit address space, some simple applications need only be recompiled for a uniform 64-bit pointer size. For example, self-contained DEC C applications that rely on only the C run-time library, without using system services or other libraries, can take this approach. Real-world applications are seldom this clean-cut, however. In more complex applications, where 64-bit address space is likely to be needed, mixes of languages, dependencies on system interfaces and other libraries, and reliance on third-party packages or libraries are common. These practices all lead to the mixed pointer size environment in which applications continue to use some 32-bit addresses while taking advantage of 64-bit virtual address space.

Applications that are likely to take advantage of 64-bit memory are those in which the declaration and management of a large data set can be logically separated from the rest of the program. This separation does not need to be at the source file level. It can be at a program flow level, indicating which internal and external interfaces will be given 64-bit addresses to work with.

The following sections explore the reasons for mixing pointer sizes.

OpenVMS and Language Support

Implementation choices that Digital made for this first release of the OpenVMS operating system that supports 64-bit virtual addressing will probably encourage mixed pointer size programming. These choices were driven largely by the need for absolute upward compatibility for existing programs and the goal of supporting large, dynamic data sets as the primary application for 64-bit addressing.

Dynamic Data Only OpenVMS services support dynamic allocation of 64-bit address space. This mechanism most closely resembles the malloc and free functions for allocating and deallocating dynamic storage in the C programming language. Allocation of this type differs from static and stack storage in that explicit source statements are required to manage it. For static and stack storage, the system is allocating the memory on behalf of the application at image activation time. (Of course, the allocation may be extended during execution in the case of stack storage.) This allocation continues to be from 32-bit addressable space.

Two special cases of static allocation are worth mentioning. Linkage sections, which are program sections that contain routine linkage information, and code sections, which contain the executable instructions, do not differ substantially from preinitialized static storage. As a result, these sections also reside only in 32-bit addressable memory.

Upward-compatibility Constraints The OpenVMS Alpha operating system is cautious to avoid using 64-bit memory freely where it may prevent upward compatibility for 32-bit applications. For example, the linkage section might seem to be a natural candidate for the OpenVMS system to allocate automatically in 64-bit memory. This allocation would essentially free more 32-bit addressable memory for application use; however, even if this were done only for applications relinked for new versions of the OpenVMS operating system, there is no guarantee that all object code treats linkage section addresses as 64 bits in width. A simple example is storing the address of a routine in a structure. Since a routine's address is the address of its procedure descriptor in the linkage section, moving the linkage section to 64-bit memory would cause code that stores this address in a 32-bit cell to fail.

Allocating the user stack in 64-bit space also appears to be a good opportunity to easily increase the amount of memory available to an application. Stack addresses are often more visible to application code than linkage section addresses are. For instance, a routine can easily allocate a local variable using temporary storage on the stack and pass the address of the variable to another routine. If the stack is moved to 64-bit space, this address quietly becomes a 64-bit address. If the called routine is not 64-bit capable, attempts to use the address will fail.

Focus on Services Required for Large Data Sets Not all system services could be changed to support 64-bit addresses (i.e., *promoted*) in time for the first version of the OpenVMS operating system to support 64-bit addressing. With the mixed-pointer model in mind, we focused on those services that were likely to be required for large data sets. For example, to allow I/O directly to and from high memory, it was essential that the I/O queuing service, SYS$QIO, accept a 64-bit buffer address. Conversely, the SYS$TRNLNM service for translating a logical name did not need to be modified to accept 64-bit addresses. Its arguments include a logical name, a table name, and a vector that contains requests for information about the name. These are small data elements that are unlikely to require 64-bit addressing on their own. Of course, they may be part of some larger structure that resides in 64-bit space. In this case, they can easily be copied to or from 32-bit addressable memory.

System services are discussed further in the section OpenVMS System Services. The 32-bit address restriction on certain system services again emphasizes the importance of being able to logically separate large data set support from the rest of an application.

Limited Language Support Another interface point that requires care when using 64-bit addressing is at calls between modules written in different programming languages. The OpenVMS Calling Standard traditionally makes it easy to mix languages in an application, but DEC C is the only high-level language to fully support 64-bit addresses in the first 64-bit-capable version of the OpenVMS operating system.[2]

The use of 64-bit addresses in mixed-language applications is possible, and data that contains 64-bit addresses may even be shared; however, references that actually use the data pointed to by these addresses need to be limited to DEC C code or assembly language. Mixed high-level language applications are certain to be mixed pointer size applications in this version of the operating system.

Support for 32-bit Libraries

Many applications rely on library packages to provide some aspect of their functionality. Typical examples include user interface packages, graphics libraries, and database utilities. Third-party libraries may or may not support 64-bit addresses. Applications that use these libraries will probably mix 32-bit and 64-bit pointer sizes and will therefore require an operating system that supports mixed pointer sizes.

Implications of Full 64-bit Conversion

For some applications, it may be desirable to mix pointer sizes to avoid the side effects of universal 64-bit address conversion. The approach of recompiling everything with 64-bit address widths is sometimes called "throwing the switch." An obvious implication of throwing the switch is that all pointer data doubles in size. For complex linked data structures, this can be a significant overall increase in size. Increasing the pointer size may also reveal hidden dependencies on pointer size being the same as integer size. If code accesses a cell as both a 32-bit integer and a 32-bit pointer, the code will no longer work if the pointer is enlarged. Thus, universally increasing the pointer size may force changes to code that would otherwise continue to work.

There is a more compelling reason for not throwing the switch for code that is part of a shared library. Library packages must not return 64-bit addresses to users of the library unless the calling code is definitely 64-bit capable. If the library developer throws the switch when building a library written in DEC C, all memory returned by the malloc function will be in 64-bit address space. This can be a problem if the address is blindly returned to a library caller. If a library is to work in a mixed pointer size environment, and it sometimes returns pointers to memory it has allocated, it needs to use mixed pointer sizes internally.

Programming Interface Issues

The coexistence of 32-bit and 64-bit pointers raised several design questions for operating system and language support, particularly in the area of routine interfaces. When an application or library is being modified to use 64-bit address space, argument passing may be the most exposed area. In this section, we describe how mixed pointer size support affects argument-passing mechanisms and the design decisions made to ease the coexistence of mixed pointer sizes.

Argument List Width

Even before the introduction of 64-bit addressing, the OpenVMS Calling Standard defined argument list elements to be 64 bits in width. When passing a 32-bit address (that is, when passing an item in 32-bit space by reference), compilers sign extend the 32-bit value into the 64-bit argument location.[1] Passing 64-bit addresses as values works transparently without changing the calling standard, assuming, of course, that the called routine expects to receive 64-bit addresses. Passing 32-bit addresses as values to routines that expect 64-bit addresses works properly because the values have been sign extended to a 64-bit width.

Pointers by Reference

Passing the addresses of pointers requires special care when mixing pointer sizes. If the caller passes a 32-bit address by reference, and the called routine reads it as a 64-bit address from memory, the upper 32 bits will be incorrect. Similarly, if the address of a 64-bit address is passed, and the called routine reads only 32 bits from memory, it will fail when that address is used.

This is the simplest case in which support of 64-bit addresses may require a programming interface change for 64-bit callers. A single entry point that receives a pointer by reference cannot tell which size pointer it has received. Some possible solutions include a new alternate entry point for 64-bit-capable callers or a new parameter indicating the size of the address.

Pointers Embedded in Structures

Pointers passed by reference are a special case of the more general problem of passing structures that contain pointers. Again, the caller and called routine must agree on the size of the pointers contained in the structure. This case offers an option that may not require a new programming interface, however. If the structure is self-identifying, the routine may be able to tell which form of the structure it has received and dispatch to appropriate code for the corresponding pointer length.

Function Return Values

Function return values are also defined to be 64 bits in width, so no calling standard change was required to support 64-bit pointer returns. It is important that a 64-bit address not be returned blindly, though, unless it is known that the caller is 64-bit capable. Typically, this is a problem for library support routines rather than for those within an application. A library routine should return a 64-bit address only if the routine has been specifically developed for a 64-bit environment or if it can tell with certainty, based on input parameters received, that the caller is 64-bit capable.

Calling Standard Issues

The OpenVMS Calling Standard defines register usage conventions, argument list locations, data structures, and standard practices for making procedure calls that operate correctly in a multilanguage and multithreaded environment. As mentioned earlier, this standard already defined argument list elements to be 64 bits in width; however, some key data structures defined by the standard were based on 32-bit pointer sizes. The goal of upward compatibility for existing code complicated the job of extending the standard. The following sections describe how the structures were ultimately changed and illustrate some approaches to supporting mixed pointer sizes when shared structures contain pointers.

Descriptors Descriptors are structures defined by the calling standard to specify an argument's type, length, and address, along with other type or

structure-specific information. Typically, descriptors are used only for character strings, arrays, and complex data types such as packed decimal.

Descriptor types are by definition self-identifying by virtue of the type and class fields they contain. An obvious choice, therefore, for extending descriptors to handle 64-bit addresses would be to add new type constants for 64-bit data elements and extend the structure beyond the type fields to accommodate larger addresses and sizes. In practice, however, the address and length fields from descriptors are frequently used without accessing the type fields, particularly when a character string descriptor is expected.

As a result, a solution was sought that would yield a predictable failure, rather than incorrect results or data corruption, when a 64-bit descriptor is received by a routine that expects only the 32-bit form. The final design includes a separate 64-bit descriptor layout that contains two special fields at the same offsets as the length and address fields in the 32-bit descriptor. These fields are called MBO (must be one) and MBMO (must be minus one), respectively. The simplest versions of the 32-bit and 64-bit descriptors are illustrated in Figure 1.

If a routine that expects a 32-bit descriptor receives a 64-bit descriptor, it will find the value 1 in the length field. This nonzero value ensures that the address will need to be read. Otherwise, the descriptor could be treated as describing a null value, and the address would be ignored. In the address field, a 32-bit reader will find the value −1. When the reader attempts to reference this address, an access violation occurs, because the OpenVMS operating system guarantees this address to be inaccessible. This combination of values ensures that an access will also fail if the length is added to the address first, in an attempt to read the last byte of data.

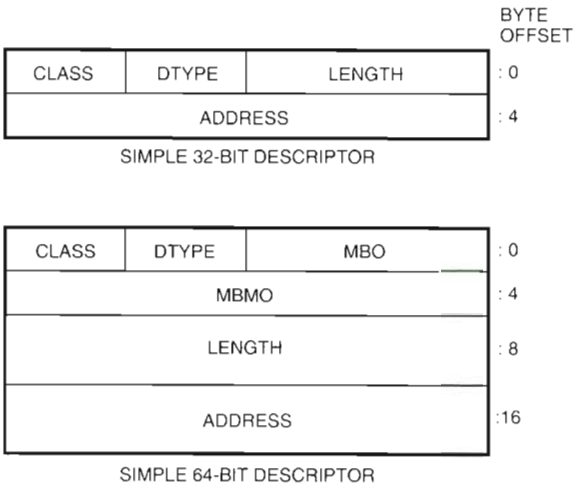


Figure 1
Simplest Versions of the 32-bit and 64-bit Descriptors

To distinguish the descriptor forms, a new routine must check the MBO and MBMO fields for the expected 64-bit descriptor values. In the OpenVMS operating system, many routines now accept either descriptor form.

Signal Arrays The signal array is a user-visible structure that is passed to condition handlers when an exception occurs. The array contains message codes, arguments specific to the conditions, and control data. Because the arguments may include one or more virtual addresses, a new format was necessary to accommodate 64-bit addresses.

The signal array could not simply be promoted to contain 64-bit addresses, because handlers in existing code often make assumptions about its format. The mechanism array, a related structure containing a snapshot of register contents, was already 64 bits in width.

The solution was to leave the original form of the signal array unchanged and create a 64-bit counterpart. The items passed to a condition handler, the 32-bit signal array address, and a 64-bit mechanism array address are the same. The mechanism array now contains a pointer to the 64-bit version of the signal array. This allows existing code to work without change, while new handlers that may require access to 64-bit addresses in exceptions can obtain the 64-bit array address from the mechanism array. Some additional work was needed in OpenVMS exception handling to keep these two arrays synchronized, because handlers are allowed to change their contents.

Sign-extension Checking

As described earlier, 32-bit addresses passed as routine arguments are sign extended into 64-bit argument locations. A safeguard that can be used in 32-bit routines that are not extended to fully support 64-bit addresses is referred to as sign-extension checking of the argument addresses. This checking consists of simply reading the low 32 bits of the argument, sign extending this value to a 64-bit width, and comparing the result to the full 64 bits of the argument. If the bits differ, the address is not one that can be represented in 32 bits. The routine can then return an error status of some kind, rather than failing in some unpredictable way. Sign-extension checking is a useful tool for ensuring robust interfaces in the mixed pointer size environment.

DEC C Language Support for Mixed Pointer Sizes

To support application programming in the mixed pointer size environment, some design work was required in the DEC C compiler. This section describes the rationale behind the final design.

It was clear that the compiler would have to provide a way for 32-bit and 64-bit pointers to coexist in the same regions of code. At the same time, customers and

internal users initially favored a simple command line switch when polled on potential compiler support for 64-bit address space. (At least one C compiler that supports 64-bit addressing, MIPSpro C, does so only through command line switches for setting pointer sizes.[3]) The motivation for using switches was to limit the source changes needed to take advantage of the additional address space, especially when portability to other platforms is desired. For cases in which mixing pointer sizes was unavoidable, something more flexible than a switch was needed.

Why Not __near and __far?

The most common suggestion for controlling individual pointer declarations was to adopt the __near and __far type qualifier syntax used in the PC environment in its transition from 16-bit to 32-bit addressing.[4] While this idea has merit in that it has already been used elsewhere in C compilers and is familiar to PC software developers, we rejected this approach for the following reasons:

- The syntax is not standard.
- The syntax requires source code edits at each declaration to be affected.
- The syntax has become largely obsolete even in the PC domain with the acceptance of the flat 32-bit address space model offered by modern 386-minimum PC compilers and the Win32 programming interface.
- Because of the vast difference in scale in choosing between 16-bit or 32-bit pointers on a PC as compared to choosing between 32-bit or 64-bit pointers on an Alpha system, there would be no porting benefit in using the same keywords. No existing source code base would be able to port to the OpenVMS mixed pointer size environment more easily because of the presence of __near and __far qualifiers.

Pragma Support

The Digital UNIX C compiler had previously defined pragma preprocessing directives to control pointer sizes for slightly different reasons than those described for the OpenVMS system.[5] By default, the Digital UNIX operating system offers a pure 64-bit addressing model. In some circumstances, however, it is desirable to be able to represent pointers in 32 bits to match externally imposed data layouts or, more rarely, to reduce the amount of memory used in representing pointer values. The Digital UNIX pointer_size pragmas work in conjunction with command line options and linker/loader features that limit memory use and map memory such that pointer values accessible to the C program can always be represented in 32 bits.

Since compatibility with the Digital UNIX compiler would have greater value if it met the needs of the OpenVMS platform, we evaluated the pragma-based approach and decided to adopt it, propagating any necessary changes back to the UNIX platform to maintain compatibility. The decision to use pragmas to control pointer size addressed the major deficiencies of the __near and __far approach. In particular, the pragma directive is specified by ISO/ANSI C in such a way that using it does not compromise portability as the use of additional keywords can, because unrecognized pragmas are ignored. Furthermore, pragmas can easily be specified to apply to a range of source code rather than to an individual declaration. A number of DEC C pragmas, including the pointer size controls implemented on the UNIX system, provide the ability to save and restore the state of the pragma. This makes them convenient and safe to use to modify the pointer size within a particular region of code without disturbing the surrounding region. The state may easily be saved before changing it at the beginning of the region and then restored at the end.

Command Line Interaction

Pragmas fit in with the initial desire of prospective users to have a simple command line switch to indicate 64 bits. As with several other pragmas, we defined a command line qualifier (/pointer_size) to specify the initial state of the pragma before any instances are encountered in the text. Unlike other pragmas, though, we also use the same command line qualifier to enable or disable the action of the pragmas altogether. In this way, a default compilation of source code modified for 64-bit support behaves the same way that it would on a system that did not offer 64-bit support. That is, the pragmas are effectively ignored, with only an informational message produced.

This behavior was adopted for consistency with the Digital UNIX behavior and also to aid in the process of adding optional 64-bit support to existing portable 32-bit source code that might be compiled for an older system or with an older compiler. In this model, a compilation of new source code using an old command line produces behavior that is equivalent to the behavior produced using an older compiler or a compiler on another platform. With one notable exception, building an application that actually uses 64-bit addressing requires changing the command line.

The exception to the rule that existing 32-bit build procedures do not create 64-bit dependencies is a second form of the pragma, named required_pointer_size. This form contrasts with the form pointer_size in that it is always active regardless of command line qualifiers; otherwise, required_pointer_size and pointer_size are identical. The intent of this second pragma is to support writing source code that specifies or interfaces to services or libraries that can only work correctly with 64-bit pointers. An example of this code might be a header file that contains declarations for both 64-bit and 32-bit memory management services; the services

must always be defined to accept and return the appropriate pointer size, regardless of the command line qualifier used in the compilation.

Pragma Usage

The use of pragmas to control pointer sizes within a range of source code fits well with the model of starting with a working 32-bit application and extending it to exploit 64-bit addressing with minimal source code edits. Programming interface and data structure declarations are typically packaged together in header files, and the primary manipulators of those data structures are often implemented together in modules.

One good approach for extending a 32-bit application would be to start with an initial analysis of memory usage measurements. The purpose of this analysis would be to produce a rough partitioning of routines and data structures into two categories: "32-bit sufficient" and "64-bit desirable." Next, 64-bit pointer pragmas could be used to enclose just the header files and source modules that correspond to the routines and data structures in the 64-bit-desirable category. After recompilation, the next step would be to respond to compiler diagnostics for pointer-type mismatches by adding pragma regions to mark sections of the 64-bit files as 32-bit and parts of the 32-bit files as 64-bit and to carefully add type casts, where necessary. This operation is likely to iterate until the compilation is clean and a debugging cycle has shown correctness. The end result is an application that takes advantage of the increased address space for the data structures that will benefit from it.

A common approach to minimizing the spread of pragmas throughout a program is to limit them to typedefs in header files. Then, subsequent uses of the defined type do not require the pragma. A simple example appears in Figure 2.

This example defines a type called char_ptr64, which may be used to declare 64-bit pointers to character data without the use of pragmas. Of course, individual pointers within structure types may also be set to 64-bit or 32-bit sizes.

Secondary Effects

With the decision made to use pragmas and the basic semantics of how the pragmas take effect established by the Digital UNIX implementation, we needed to consider additional requirements and issues that might be specific to the OpenVMS implementation. Two major differences between the platforms are

1. On the Digital UNIX system, the linker/loader options used with mixed pointer size compilations ensure that any address value obtained by the program can be represented using 32 bits, whereas on the OpenVMS system, any program using 64-bit pointers in C will almost certainly encounter address values that cannot be represented in 32 bits.
2. On the Digital UNIX system, the scope of the use of mixed pointer sizes was expected to be quite small and not likely to grow much over time, whereas on the OpenVMS system, the scope is expected to be somewhat larger at first and grow significantly over time.

These two differences emphasized the need for effective compile-time diagnostics, debugging aids, environmental support, and clear documentation.

Diagnostics As an aid to finding bugs resulting from improper mixing of pointer sizes, the DEC C compiler provides two kinds of diagnostics. Compile-time warnings are issued for assignments from long pointers to short pointers because of the possibility of data loss. In addition, users may enable run-time checking for pointer truncation through a command line qualifier. This option causes the compiler to generate code on each conversion from a long to a short pointer, which will signal a range-check error if data truncation occurs.

Run-time checking is particularly useful in code that sometimes employs type casting to use long pointers in short pointer contexts. Since this action prevents a compile-time warning about using a long pointer where a short pointer is expected, a run-time check may be the only way to discover a coding error. The run-time check qualifier provides options distinguishing this case from checking on general assignments and parameter passing, allowing users to select for which classes of pointer-size mixing the compiler should generate checking code. Run-time checking is also available for parameters received by a routine. This allows detection of 64-bit addresses passed to routines that expect 32-bit parameters even when the caller is separately compiled or written in a different programming language. For performance reasons, it is usually desirable to remove all run-time checking once a program is debugged.

```
#pragma required_pointer_size save    /* Save the previous pointer size */
#pragma required_pointer_size 64      /* Set pointer size to 64 bits */
typedef char * char_ptr64;            /* Define a 64-bit char pointer */
#pragma required_pointer_size restore /* Restore the pointer size */
```

Figure 2
Sample Header File Code That Limits Pragmas to Defined Types

Allocation Function Mapping The command line qualifier setting the default pointer size has an additional effect that simplifies the use of 64-bit address space. If an explicit pointer size is specified on the command line, the malloc function is mapped to a routine specific to the address space for that size. For example, _malloc64 is used for malloc when the default pointer size is 64 bits. This allows allocation of 64-bit address space without additional source changes. The source code may also call the size-specific versions of run-time routines explicitly, when compiled for mixed pointer sizes. These size-specific functions are available, however, only when the /pointer_size command line qualifier is used. See "Adding 64-bit Pointer Support to a 32-bit Run-time Library" in this issue for a discussion of other effects of 64-bit addressing on the C run-time library.[6]

Header File Semantics The treatment of pointer_size pragmas in and around header files (i.e., any source included by the #include preprocessing directive) deserves special mention. Programs typically include both private definition files and public or system-specific header files. In the latter case, it may not be desirable for definitions within the header files to be affected by the pointer_size pragmas or command line currently in effect. To prevent these definitions from being affected, the DEC C compiler searches for special prologue and epilogue header files when a #include directive is processed. These files may be used to establish a particular state for environmental pragmas, such as pointer_size, for all header files in the directory. This eliminates the need to modify either the individual header files or the source code that includes them.

The compiler creates a predefined macro called __INITIAL_POINTER_SIZE to indicate the initial pointer size as specified on the command line. This may be of particular use in header files to determine what pointer size should be used, if mixed pointer size support is desirable. Conditional compilation based on this macro's definition state can be used to set or override pointer size or to detect compilation by an older compiler lacking pointer-size support. If its value is zero, no /pointer_size qualifier was specified, which means that pointer_size pragmas do not take effect. If its value is 32 or 64, pointer_size pragmas do take effect, so it can be assumed that mixed pointer sizes are in use.

Code Example

In the simple code example shown in Figure 3, suppose that the routine proc1 is part of a library that has been only partially promoted to use 64-bit addresses. This function may receive either a 32-bit address or a 64-bit address in the *argument_ptr* parameter. To demonstrate the use of the new DEC C features, proc1 has been modified to copy this character string parameter from 64-bit space to 32-bit space when necessary, so that routines that proc1 subsequently calls need to deal with only 32-bit addresses.

The __INITIAL_POINTER_SIZE macro is used to determine if pointer_size pragmas will be effective and, hence, whether *argument_ptr* might be 64 bits in width. If it might be a 64-bit pointer, whose actual width is unknown in this example, the pointer's value is copied to a 32-bit-wide pointer. The pointer_size pragma is used to change the current pointer size to 32 bits to declare the temporary pointer. Next, the two pointer values are compared to determine if the original pointer fits in 32 bits. If the pointer does not fit, temporary storage in 32-bit addressable space is allocated, and the argument is copied there. Note that the example uses _malloc32 rather than malloc, because malloc would allocate 64-bit address space if the initial pointer size was 64 bits. At the end of the routine, the temporary space is freed, if necessary.

A type cast is used in the assignment from *argument_ptr* to *temp_short_ptr*, even though both variables are of type char *. Without this type cast, if *argument_ptr* is a 64-bit-wide pointer, the DEC C compiler would report a warning message because of the potential data loss when assigning from a 64-bit to a 32-bit pointer.

For other examples of pointer_size pragmas and the use of the __INITIAL_POINTER_SIZE macro, see Duane Smith's paper on 64-bit pointer support in run-time libraries.[6]

OpenVMS System Services

The OpenVMS operating system provides a suite of services that perform a variety of basic operating system functions.[7] Design work was required to maximize the utility of these routines in the new mixed pointer size environment. Issues that needed to be addressed included the following, which are discussed in subsequent sections:

- Several services pass pointers by reference and, hence, required an interface change.
- Because of resource constraints, not all system services could be promoted to handle 64-bit addresses in the first version of the 64-bit-capable OpenVMS operating system.
- Since the services provide mixed levels of support, it is important to indicate those that support 64-bit addresses and those that do not.
- Certain new services seemed desirable to improve the usability of 64-bit address space.

Services That Are 64-bit Friendly

Services that can be promoted to support 64-bit addresses without any interface change are called 64-bit friendly. If a service receives an address by reference, the service is typically not 64-bit friendly, and a separate

```
void proc1(char * argument_ptr)
{
#if __INITIAL_POINTER_SIZE != 0
   #pragma pointer_size save
   #pragma pointer_size 32
   char * temp_short_ptr;
   temp_short_ptr = (char *)argument_ptr;
   if (temp_short_ptr != argument_ptr) {
       temp_short_ptr = _malloc32(strlen(argument_ptr) + 1);
       strcpy(temp_short_ptr,argument_ptr);
       argument_ptr = temp_short_ptr;
   }
   else {
      temp_short_ptr = 0;
   }
   #pragma pointer_size restore
#endif

/*
    The actual body of proc1 is omitted. Assume that it calls
    routines that operate on the data pointed to by argument_ptr
    and that the routines are not yet prepared to handle 64-bit
    addresses.
*/

#if __INITIAL_POINTER_SIZE != 0
if (temp_short_ptr != 0)
    free(temp_short_ptr);
#endif
}
```

Figure 3
Code Example of Pointer_size Pragmas and the __INITIAL_POINTER_SIZE Macro

entry point is required to support 64-bit addresses. A single routine cannot distinguish whether the address at the specified location is 32 bits or 64 bits in width.

If a service does not receive or return an address by reference, the service is usually 64-bit friendly. Even descriptor arguments present no problem, because the 32-and 64-bit versions can be distinguished at run time. The majority of services fall into this category.

The services that are not 64-bit friendly include the entire suite of memory management system services, since they access address ranges passed by reference. Other such services include those that receive a 32-bit vector as an argument, which may include the address of a pointer as an element. A good example from this group is SYS$FAOL, which accepts a 32-bit vector argument for formatted output. For all these services, new interfaces were designed to accommodate 64-bit callers.

Promotion of Services

The OpenVMS project team explored the idea of promoting all system services to support 64-bit addresses. Since the majority of OpenVMS system service routines are written in the MACRO-32 assembly language or the Bliss-32 programming language, the internals of the routines could not be promoted to handle 64-bit addresses without modifications. We could not take advantage of the throw-the-switch approach, and we did not want to because many pointers used internally in the OpenVMS operating system remain at 32 bits.

We considered using 64-bit jacket routines to copy 64-bit arguments to the stack in 32-bit space, which would then call the 32-bit internal routine to perform the requested function. However, this approach would fail for context arguments such as asynchronous system trap (AST) routine parameters, where the address of the argument is stored for subsequent use. This approach would also prevent services from operating on any true 64-bit addresses. It was clear that at least some routines would have to be modified internally.

The idea of using jacket routines was ultimately rejected for several reasons. First, the jackets would need to be custom written to ensure correct parameter semantics. There could not be a "common jacket" that could have saved time and lowered risk. Second, there would be an undesirable performance impact for 64-bit callers. Third, we decided that having a complete 64-bit system service suite was not essential for usable 64-bit support. We could define a subset that would meet the needs of 64-bit address space users, while lowering our risk and implementation costs.

The services selected for 64-bit support fall into four categories.

1. Memory management services.
2. Performance-critical services. This group includes services that are typically sensitive to the addition of

even a few cycles of execution time. Requiring that a 64-bit address user do any additional work, such as copying data to 32-bit space, is undesirable. An example of this type of service is SYS$ENQ, which is used for queuing lock requests.

3. Design center services. The primary design center for 64-bit support was database applications. Database architects and consultants were polled to determine which services were most needed by their products. Many of these services, for example SYS$QIO for queuing I/O requests, were also in the performance-critical set.
4. Other useful basic services. This set includes services to ease the transition to 64 bits with minimal change to program structure. For example, the SYS$CMKRNL service accepts a routine address and a vector of 32-bit arguments and invokes the routine in kernel mode, passing those arguments. Without a new 64-bit version of SYS$CMKRNL, a caller could not pass a 64-bit address to the kernel mode routine without changing the form of the argument block, such as passing a structure that SYS$CMKRNL would not interpret as a vector.

Several steps were taken to ease programming to this subset implementation.

- For all 64-bit services, *all* pointer arguments may be in 64-bit space. Extending only individual arguments for some services would have been confusing and difficult to document.
- The 64-bit-capable system services are clearly listed in the OpenVMS documentation, and the documentation for individual services clearly calls out their capabilities.[7,8]
- For C programmers, the header file that defines function prototypes for system services (STARLET.H) defines the expected pointer size for service arguments. This file can be used for compile-time type checking for correct argument pointer sizes.
- A strict naming convention has been adhered to for 64-bit services. If a routine was 64-bit friendly, i.e., it required no interface change, its name was not changed. If a new entry point was required because, for example, an address is passed by reference, a "_64" suffix was added to the name to identify the new entry point.
- Sign-extension checking is performed in routines that do not accept 64-bit addresses.

Centralized Sign-extension Checking

For services that have not been promoted to accept arguments in 64-bit space, centralized sign-extension checking takes place. As described in the section Sign-extension Checking, such checking prevents errors that occur when a 64-bit address is erroneously passed to a routine that uses only 32-bit addresses. This centralized checking is part of the system service dispatcher, which returns the error status SS$_ARG_GTR_32_BITS when the error is discovered. Performing the checking at this common point minimized the implementation effort, while protecting sensitive inner mode services. No changes were necessary to the modules that contain the 32-bit service code. The internal vector of services contains a flag for each service indicating whether checking should be done.

Naturally, it is best for mixed-size errors to be discovered at compile time. The DEC C compiler issues a warning message when a 64-bit pointer is used as a parameter to a routine whose function prototype specifies that the parameter should be a 32-bit pointer. Run-time sign-extension checking works for any language, though, including MACRO-32.

This support can also be used to allow a run-time decision to be made to copy data from 64-bit space to 32-bit space. For example, a routine could call a system service, passing along an address that it had received as a parameter. If the service returns SS$_ARG_GTR_32_BITS, the calling routine can then copy the argument to the stack and retry the service. In this way, the overhead of copying can be avoided if copying is unnecessary. When the system service is promoted to handle 64-bit addresses in a future version of the OpenVMS operating system, no change will be needed in this caller; the data copying code will never be invoked. This approach may be appropriate for a run-time library that needs to be fully 64-bit capable today on OpenVMS Alpha version 7.0, if that library will not be rereleased for a future version of the OpenVMS operating system.

Memory Management System Services

The OpenVMS memory management system services are not 64-bit friendly because they pass 32-bit input and output address arguments by reference. This set of services includes SYS$EXPREG (expand program/control region), SYS$MGBLSC (map global section), SYS$CRMPSC (create and map section), and SYS$PURGWS (purge working set), among others.

The guiding principle in promoting these services was that the new 64-bit services had to perform the same functions as their 32-bit counterparts but not necessarily with an identical interface. Since 32-bit addresses can be expressed as 64-bit addresses with sign-extension bits in the upper 32 bits, it made sense to accommodate 32-bit addresses in the 64-bit interfaces, making the new services a superset of the 32-bit forms. For example, the SYS$CRMPSC service was split into multiple 64-bit-capable services, because it handles a variety of types of sections. The new services can operate on either 32-bit or 64-bit addresses and have simpler

interfaces than the 32-bit-only SYS$CRMPSC. The original SYS$CRMPSC is still present so that existing code may function without change.

Some new feature requests were considered as part of the 64-bit effort, but, to maintain the focus of the release, these features were not implemented. The 64-bit memory management services were designed to more easily accommodate new features in the future. For example, the new services check the argument count for both too many and too few supplied arguments. In this way, new optional arguments can be added later to the end of the list without jeopardizing backward compatibility.

Virtual Regions

One new feature that was added to the suite of 64-bit memory management services is support for new entities called virtual regions. A virtual region is an address range that is reserved by a program for future dynamic allocation requests. The region is similar in concept to the program region (P0) and the control region (P1), which have long existed on the OpenVMS operating system.[9] A virtual region differs from the program and control regions in that it may be defined by the user by calling a system service and may exist within P0, P1, or the new 64-bit addressable process-private space, P2.[1] When a virtual region is created, a handle is returned that is subsequently used to identify the region in memory management requests.

Address space within virtual regions is allocated in the same manner as in the default P0, P1, and P2 regions, with allocation defined to expand space toward either ascending or descending addresses. As in the default regions, allocation is in multiples of pages. The OpenVMS operating system keeps track of the first free virtual address within the region. A region can be created such that address space is created automatically when a virtual reference is made within the region, just as the control region in P1 space expands automatically to accommodate user stack expansion. When a virtual region is created within P0, P1, or P2, the remainder of that containing region decreases in size so that it does not overlap with the virtual region.

Virtual regions were added to the OpenVMS Alpha operating system along with the 64-bit addressing capability so that the huge expanse of 64-bit address space could be more easily managed. If a subsystem requires a large portion of virtually contiguous address space, the space can be reserved within P2 with little overhead. Other subsystems within the application cannot inadvertently interfere with the contiguity of this address space. They may create their own regions or create address space within one of the default regions.

Another advantage of using virtual regions is that they are the most efficient way to manage sparse address space within the 64-bit P2 space. Furthermore, no quotas are charged for the creation of a virtual region. The internal storage for the description of the region comes from process address space, which is the only resource used.

Summary

This paper presents the reasons behind the new OpenVMS mixed pointer size environment and the support added to allow programming within this environment. The discussion touches on some of the new support designed to simplify the use of the 64-bit address space.

The approaches discussed yielded full upward compatibility for 32-bit applications, while allowing other applications access to the huge 64-bit address space for data sets that require it. Promotion of all pointers to 64-bit width is not required to use 64-bit space; the mixed pointer size environment was considered paramount in all design decisions. A case study of adding 64-bit support to the C run-time library also appears in this issue of the *Journal*.[6]

Acknowledgments

The authors wish to thank the other members of the 64-bit Alpha-L Team who helped shape many of the ideas presented in this paper: Mark Arsenault, Gary Barton, Barbara Benton, Ron Brender, Ken Cowan, Mark Davis, Mike Harvey, Lon Hilde, Duane Smith, Cheryl Stocks, Lenny Szubowicz, and Ed Vogel.

References

1. M. Harvey and L. Szubowicz, "Extending OpenVMS for 64-bit Addressable Virtual Memory," *Digital Technical Journal*, vol. 8, no. 2 (1996, this issue): 57–71.

2. *OpenVMS Calling Standard* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBBA-TE, 1995).

3. *MIPSpro 64-Bit Porting and Transition Guide*, Document No. 007-2391-002 (Mountain View, Calif.: Silicon Graphics, Inc., 1996).

4. *C Language Reference for MS-DOS and Windows Operating Systems* (Redmond, Wash.: Microsoft Corporation, 1991) and "Declarations and Types," chap. 3, and "Expressions and Assignments," chap. 4, in *Microsoft C/C++ Version 7.0* (Redmond, Wash.: Microsoft Corporation, 1991).

5. *Digital UNIX Programmer's Guide* (Maynard, Mass.: Digital Equipment Corporation, 1996).

6. D. Smith, "Adding 64-bit Pointer Support to a 32-bit Run-time Library," *Digital Technical Journal*, vol. 8, no. 2 (1996, this issue): 83–95.

7. *OpenVMS System Services Reference Manual: A-GETMSG* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBMA-TE, 1995) and *OpenVMS System Services Reference Manual: GETQUI-Z* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBN-TE, 1995).

8. *OpenVMS Alpha Guide to 64-Bit Addressing* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBCA-TE, 1995).

9. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford, Mass.: Digital Press, 1987).

Biographies

Thomas R. Benson
A consulting engineer in the OpenVMS Engineering Group, Tom Benson is one of the developers of 64-bit addressing support. Tom joined Digital's VAX Basic project in 1979 after receiving B.S. and M.S. degrees in computer science from Syracuse University. After working on an optimizing compiler shell used by several VAX compilers, he joined the VMS Group where he led the VMS DECwindows FileView and Session Manager projects, and brought the Xlib graphics library to the VMS operating system. Tom holds three patents on the design of the VAX MACRO-32 compiler for Alpha and recently applied for two patents related to 64-bit addressing work.

Karen L. Noel
A principal engineer in the OpenVMS Engineering Group, Karen Noel is one of the developers of 64-bit addressing support. After receiving a B.S. in computer science from Cornell University in 1985, Karen joined Digital's RSX Development Group. In 1990, she joined the VMS Group and ported several parts of the VMS kernel from the VAX platform to the Alpha platform. As one of the principal designers of OpenVMS Alpha 64-bit addressing support, she has recently applied for six software patents.

Richard E. Peterson
Rich Peterson joined Digital's DEC C/C++ team in 1992. He was the project leader for the development of the C and C++ compilers that joined the Microsoft front ends to the GEM back end. These compilers were used to build and deliver the first release of the Windows NT operating system on the Alpha platform and later were used in Visual C++ for Alpha. A principal software engineer in the Core Technologies Group, Rich is currently the project leader for DEC C on the Digital UNIX and OpenVMS platforms. Prior to joining Digital, Rich worked at Intermetrics on a number of compiler projects, including HAL/S for the Space Shuttle and Ada for IBM/370 and MIL-STD 1750A. Rich also worked at COMPASS, where he was the project leader for the Thinking Machines Fortran compiler and Digital's initial MPP compiler effort. He received a B.S. in English from the California Institute of Technology and has applied for one patent on Alpha OpenVMS 64-bit compiler work.

Duane A. Smith

Adding 64-bit Pointer Support to a 32-bit Run-time Library

A key component of delivering 64-bit addressing on the OpenVMS Alpha operating system, version 7.0, is an enhanced C run-time library that allows application programmers to allocate and utilize 64-bit virtual memory from their C programs. This C run-time library includes modified programming interfaces and additional new interfaces yet ensures upward compatibility for existing applications. The same run-time library supports applications that use only 32-bit addresses, only 64-bit addresses, or a combination of both. Source code changes are not required to utilize 64-bit addresses, although recompilation is necessary. The new techniques used to analyze and modify the interfaces are not specific to the C run-time library and can serve as a guide for engineers who are enhancing their programming interfaces to support 64-bit pointers.

The OpenVMS Alpha operating system, version 7.0, has extended the address space accessible to applications beyond the traditional 32-bit address space. This new address space is referred to as 64-bit virtual memory and requires a 64-bit pointer for memory access.[1] The operating system has an additional set of new memory allocation routines that allows programs to allocate and release 64-bit memory. In OpenVMS Alpha version 7.0, this set of routines is the only mechanism available to acquire 64-bit memory.

For application programs to take advantage of these new OpenVMS programming interfaces, high-level programming languages such as C had to support 64-bit pointers. Both the C compiler and the C run-time library required changes to provide this support. The compiler needed to understand both 32-bit and 64-bit pointers, and the run-time library needed to accept and return such pointers.

The compiler has a new qualifier called /pointer_size, which sets the default pointer size for the compilation to either 32 bits or 64 bits. Also added to the compiler are pragmas (directives) that can be used within the source code to change the active pointer size. An application program is not required to compile each module using the same /pointer_size qualifier; some modules may use 32-bit pointers while others use 64-bit pointers. Benson, Noel, and Peterson describe these compiler enhancements.[2] The *DEC C User's Guide for OpenVMS Systems* documents the qualifier and the pragmas.[3]

The C run-time library added 64-bit pointer support either by modifying the existing interface to a function or by adding a second interface to the same function. Public header files define the C run-time library interfaces. These header files contain the publicly accessible function prototypes and structure definitions. The library documentation and header files are shipped with the C compiler; the C run-time library ships with the operating system.

This paper discusses all phases of the enhancements to the C run-time library, from project concepts through the analysis, the design, and finally the implementation. The *DEC C Runtime Library Reference Manual for OpenVMS Systems* contains user documentation regarding the changes.[4]

Starting the Project

We devoted the initial two months of the project to understanding the overall OpenVMS presentation of 64-bit addresses and deciding how to present 64-bit enhancements to customers. Representatives from OpenVMS engineering, the compiler team, the run-time library team, and the OpenVMS Calling Standard team met weekly with the goal of converging on the deliverables for OpenVMS Alpha version 7.0.

The project team was committed to preserving both source code compatibility and the upward compatibility aspects of shareable images on the OpenVMS operating system. Early discussions with application developers reinforced our belief that the OpenVMS system must allow applications to use 32-bit and 64-bit pointers within the same application. The team also agreed that for a mixed-pointer application to work effectively, a single run-time library would need to support both 32-bit and 64-bit pointers; however, there was no known precedent for designing such a library.

One implication of the decision to design a run-time library that supported 32-bit and 64-bit pointers was that the library could never return an unsolicited 64-bit pointer. Returning a 64-bit pointer to an application that was expecting a 32-bit pointer would result in the loss of one half of an address. Although typically this error would cause a hardware exception, the resulting address could be a valid address. Storing to or reading from such an address could result in incorrect behavior that would be difficult to detect.

The *OpenVMS Calling Standard* specifies that arguments passed to a function be 64-bit values.[5] If a 32-bit address is used, it is always sign extended to form a 64-bit address that can be used by the Alpha hardware. The C run-time library team exploited this fact when creating the 64-bit interface to the library.

The team also agreed that using 64-bit pointers should be as simple as possible; the simplest mode would allow the application to compile using the qualifier /pointer_size=64 without making source code changes. The design of 64-bit support must appear as a logical extension to the C programming environment in use today. Furthermore, applications written to conform strictly to the ANSI standard must be able to use 64-bit pointers while remaining conformant. For example, allocating 64-bit virtual memory would be an extension to the standard C memory management functions malloc, calloc, realloc, and free.

This paper shows that each of the C run-time library interfaces examined falls into one of the following four categories (listed in order of added complexity to library users):

1. Not affected by the size of a pointer
2. Enhanced to accept both pointer sizes
3. Duplicated to have a 64-bit-specific interface
4. Restricted from using 64-bit pointers

One last point to come from the meetings was that many of the C run-time library interfaces are implemented by calling other OpenVMS images. For example, the Curses Screen Management interfaces make calls to the OpenVMS Screen Management (SMG) facility. It is important that the C run-time library defines the interfaces to support 64-bit addresses without looking at the implementation of the function. Consistency and completeness of the interface are more important than the complexity of the implementation. In the SMG example, if the C run-time library needs to make a copy of a string prior to passing the string to the SMG facility, this is what will be implemented.

Analyzing the Interfaces

The process of analyzing the interfaces began by creating a document that listed all the header files and the definitions in these files. A total of 50 header files that contained approximately 50 structures and 500 prototypes needed to be analyzed. Each structure or prototype had to be examined to see if a change in pointer size would affect the interface. Keep in mind that we analyzed only the interfaces; we did not examine the underlying implementation changes that would be required.

Analyzing the Structures

It is necessary to distinguish between a structure, which may contain pointers, and a pointer to the structure itself. For example, the div_t structure contains two integer fields. Although the size of the pointer to div_t does not affect the contents of the structure, the entire structure may be allocated in 32-bit or 64-bit virtual memory. Functions that accept a pointer to such a structure may need to be modified to accommodate the 64-bit case. The div_t structure is

```
typedef struct {
  int quot, rem;
} div_t;
```

Many structures used in the C run-time library interfaces are allocated by the run-time library in response to a function call. For example, a call to the fopen function returns the following pointer to the FILE structure:

```
FILE *fopen(const char *filename,
            const char *mode);
```

The C run-time library always allocates FILE structures in 32-bit virtual memory and returns a 32-bit pointer to the calling program. This important concept can dramatically impact the use of 64-bit pointers

in structures. If a FILE pointer is always a 32-bit pointer, structures that contain only FILE pointers are not affected by the choice of pointer size. We use this information when we look at the layout of structures and examine function prototypes that accept pointers to structures.

In this paper, structures that are always allocated in 32-bit virtual memory are referred to as structures bound to low memory. After determining which structures are bound to low memory, we examine the layout of each structure to decide if the structure is affected by pointer size. We keep in mind that pointer size does not affect a structure that is bound to low memory.

For upward compatibility, the analysis must always consider existing software that depends on the layout of the structure. In the case of public header file analysis, such dependence will probably always be present. An application may have executable code that, for example, fetches 4 bytes beginning at byte 12 of the structure and dereferences those 4 bytes as the address of a string.

For these public structures, the analysis must weigh the impact of forcing these structures to be 32-bit pointers. If the decision is made to allocate two different structure types, each function that either returns or is passed such a structure must have a pointer-size-specific implementation. The case analysis and further details appear in the section Pointer to Pointer-size-sensitive Structures.

Analyzing the Function Prototypes

Analyzing functions only requires looking at the function prototypes. To determine the effect of pointer size on a function, we look at each parameter and return value that involves a pointer. This section describes the types of situations that are affected by pointer size, from the simplest type to the most complex. Note that when a program passes an array of any type to a function, the array is passed as a pointer and must be considered.

Making 64-bit-friendly Parameters As mentioned in the section Starting the Project, the *OpenVMS Calling Standard* specifies that a 32-bit address is sign extended to a 64-bit address when passed as an argument to a function. This implies that existing programs that pass addresses as parameters are already sign extending those 32-bit addresses to be passed as 64-bit quantities. Each 32-bit address can, therefore, be expressed as a 64-bit address in which the top 32 bits are zero.

This sign-extending scheme allows the run-time library to have a single implementation that can be used by both 32-bit and 64-bit calling programs. This implementation would be modified to accept only 64-bit addresses. An implementation that supports parameters of either pointer size is referred to as being 64-bit friendly. The function strlen is an example of a 64-bit-friendly function.

```
size_t strlen(const char *string);
```

The *string* parameter is the only part of the strlen function that the pointer size affects. To support 64-bit addressing, the strlen function had to be modified to accept a 64-bit pointer.

Parameters Bound to Low Memory In structures bound to low memory, the addresses that the programs pass are always 32-bit addresses. One explanation is that the structures are managed by the run-time library, and the only method of creating, destroying, or obtaining the addresses of these structures is by calling a library routine. Given that a single library services both 32-bit and 64-bit calling programs, the library does not change the structures based on command qualifiers, nor does it allocate the structures in 64-bit virtual memory. For user convenience, the C run-time library implemented these pointers as 32-bit return values but 64-bit-friendly parameters.

The reason for this design became apparent while testing the 64-bit interfaces to the library. Consider the following code fragment, which exists in many applications:

```
FILE *fp;
char buffer[100];
fp = fopen("the_file", "r");
fread(array, sizeof(buffer), 1, fp);
```

The C run-time library always allocates a FILE structure in 32-bit virtual memory. When the previous code fragment is compiled using /pointer_size=64, *fp* is declared as a 64-bit pointer to a FILE structure, because using this qualifier specifies the default pointer size to be used. When the fopen function returns the 32-bit pointer, the return value is sign extended into the 64-bit FILE pointer. If the fourth parameter of the fread function had been declared as a 32-bit FILE pointer, the compiler would report an error when the 64-bit FILE pointer *fp* was passed as an argument. This example explains why the C run-time library declares structures bound to low memory as 32-bit return values but 64-bit parameters.

Parameters Restricted to Low Memory Structures restricted to low memory are similar to those bound to low memory except that the user allocates the structures and can allocate the structures in high memory. The C run-time library cannot support the allocation of such structures in 64-bit virtual memory.

An example of a parameter being restricted to a low memory address is the buffer being passed as the parameter to the function setbuf. The parameter defines this buffer to be used for I/O operations. The user expects to see this buffer change as I/O operations are performed on the file. If the run-time library made a copy of this buffer, the changes would appear in the copy and not in the original buffer that the user supplied. When the C run-time library begins to use the 64-bit OpenVMS Record Management Services (RMS) interface, this low-memory restriction will be removed.

In most cases, the run-time library is able to hide the fact that the 32-bit RMS interface is not able to interpret a 64-bit virtual memory address. Consider the *filename* parameter to the fopen function. If the parameter is a 64-bit virtual memory address, the run-time library copies this parameter to 32-bit virtual memory and passes the address of the copy to RMS. Neither the user nor RMS is aware that this copy has been made. The library may copy the data if and only if such a copy operation does not change functionality or significantly degrade performance.

Size-independent Structure Pointers Many functions receive the address of a structure whose layout is not affected by pointer size. The simplest address in this category is that of an array of integers. This array may be in either 32-bit or 64-bit virtual memory, but only one interface is needed to determine the layout of the structure. If the structure layout is independent of pointer size, then pointer-size-specific entry points are not required for this parameter. The developer would still make the parameter 64-bit friendly so that the user would have the freedom to make the allocation that is best for the application.

Pointer to Pointer Parameters It is common practice for a function to be passed a pointer to a pointer. If the pointer being pointed to is not bound or restricted to a 32-bit address, then two implementations of the function are necessary.

To understand why some functions require two implementations, consider the following strtod function:

```
double strtod(const char *string,
              char **endptr);
```

The strtod function converts a string to a floating-point double-precision number. The second parameter to this function, *endptr*, is a pointer to a memory location into which the address of the first unrecognized character is to be placed. The caller of this function has allocated either 4 or 8 bytes to store this address. Without pointer-size-specific entry points, the function has no way of determining how many bytes to write. Writing 4 bytes may truncate a pointer; writing 8 bytes may overwrite 4 bytes of user data that follows the pointer. The strtod function, therefore, has two implementations. The first expects *endptr* to be the address of a 32-bit pointer, and the second expects *endptr* to be the address of a 64-bit pointer.

Pointer to Pointer-size-sensitive Structures Many functions receive the address of a structure. If the analysis reveals that the layout of this structure is dependent upon pointer size, the functions that receive or return this structure must have pointer-size-specific entry points.

Note that the layout of the structure is separate from whether the structure is allocated in low memory or in high memory. The 32-bit-specific entry point is needed to understand the layout of the structure, but the parameter should allow this structure to be allocated in high memory.

Functions that receive the address of an array of addresses are treated in the same way, assuming that the addresses in the array are neither bound nor restricted to low memory. The function being called needs to know if the array contains 32-bit addresses or 64-bit addresses. Unlike the address of the array, the individual members of the array are not sign extended to 64-bit values.

Separate implementations are necessary only to determine the layout of what is being pointed to. The 32-bit interface handles pointers to structures containing 32-bit addresses, and the 64-bit interface handles pointers to structures containing 64-bit addresses.

Functions That Return Pointers Many functions return pointers as the value of the function. These pointers are either pointer-size specific or they are not affected by the pointer size. Similar to its specifications for 64-bit-friendly parameters, the *OpenVMS Calling Standard* indicates that return values on the OpenVMS Alpha operating system are always sign extended to 64-bit values and returned in register zero (R0).

To make an address parameter 64-bit friendly, a function allows a 64-bit address to be passed, thus enabling both 32-bit and 64-bit calling programs to use a single interface. Conversely, if a function returns a 64-bit address to a 32-bit calling program, the address is safely returned in R0 but is truncated when moved from R0 into the user's data area. A 64-bit-friendly address return value is always 32 bits. When moved from R0 into the calling program's variable, it is sign extended when the calling program is using 64-bit addresses.

If the return value of a function can be a 64-bit address, this function must have pointer-size-specific entry points. If the function returns the address of a

structure that is bound to low memory, such as a FILE or WINDOW pointer, the return value does not force separate entry points.

Certain functions, such as malloc, allocate memory on behalf of the calling program and return the address of that memory as the value of the function. These functions have two implementations: the 32-bit interface always allocates 32-bit virtual memory, and the 64-bit interface always allocates 64-bit virtual memory.

Many string and memory functions have return values that are relative to a parameter passed to the same routine. These addresses may be returned as high memory addresses if and only if the parameter is a high memory address.

The following is the function prototype for strcat, which is found in the header file <string.h>:

```
char *strcat(char *s1, const char *s2);
```

The strcat function appends the string pointed to by *s2* to the string pointed to by *s1*. The return value is the address of the latest string *s1*.

In this case, the size of the pointer in the return value is always the same as the size of the pointer passed as the first parameter. The C programming language has no way to reflect this. Since the function may return a 64-bit pointer, the strcat function must have two entry points.

As discussed earlier, the pointer size used for parameter *s2* is not related to the returned pointer size. The C run-time library made this *s2* argument 64-bit friendly by declaring it a 64-bit pointer. This declaration allows the application programmer to concatenate a string in high memory to one in low memory without altering the source code. The following strcat function statement shows this declaration:

```
char *strcat(char *s1, __char_ptr64 s2);
```

The data type *__char_ptr64* is a 64-bit character pointer whose definition and use will be explained later in this paper.

High-level Design

The /pointer_size qualifier is available in those versions of the C compiler that support 64-bit pointers. The compiler has a predefined macro named __INITIAL_POINTER_SIZE whose value is based on the use of the /pointer_size qualifier. The macro accepts the following values:

- 0, which indicates that the /pointer_size qualifier is not used or is not available
- 32, which indicates that the /pointer_size qualifier is used and has a value of 32
- 64, which indicates that the /pointer_size qualifier is used and has a value of 64

The C run-time library header files conditionally compile based on the value of this predefined macro. A zero value indicates to the header files that the computing environment is purely 32-bit. The pointer-size-specific function prototypes are not defined. The user must use the /pointer_size qualifier to access 64-bit functionality. The choice of 32 or 64 determines the default pointer size.

The header files define two distinct types of declarations: those that have a single implementation and those that have pointer-size-specific implementations. The addresses passed or returned from functions that have a single implementation are either bound to low memory, restricted to low memory, or widened to accept a 64-bit pointer.

Those functions that have pointer-size-specific entry points have three function prototypes defined. Using malloc as an example, prototypes are created for the functions malloc, _malloc32, and _malloc64. The latter two prototypes are the pointer-size-specific prototypes and are defined only when the /pointer_size qualifier is used. The malloc prototype defaults to calling _malloc32 when the default pointer size is 32 bits. The malloc prototype defaults to calling _malloc64 when the default pointer size is 64 bits. Application programmers who mix pointer types use the /pointer_size qualifier to establish the default pointer size but can then use the _malloc32 and _malloc64 explicitly to achieve nondefault behavior.

In addition to being enhanced to support 64-bit pointers, the C compiler has the added capability of detecting incorrect mixed-pointer usage. It is the function prototype found in the header files that tells the compiler exactly what pointer size is permitted or expected in a call. Proper use of the header files helps prevent pointer truncation.

The actual functions called in the C run-time library are either decc$malloc or decc$_malloc64, depending on the pointer size. The C run-time library does not contain an entry point called decc$_malloc32. This naming scheme was selected so that applications that link on older systems always get the 32-bit interface.

The C compiler has always looked at a table within the C run-time library shareable image for assistance in name prefixing. Using this table, the compiler knows to change calls to the malloc function into calls to the decc$malloc function and not to change calls to xyz, which is not a C run-time library function, into calls to decc$xyz.

The C run-time library and the C compiler have added new information to the table that tells the compiler which functions have pointer-size-specific entry points. When the compiler sees a call to the function _xyz32, it looks it up in the name table. If the name of the function is found, the compiler then looks at

whether the function is the 32-bit-specific entry point. If it is, the compiler forms the prefixed name by adding "decc$" to the beginning of the name but also removes the "_" and the "32." Consequently, the function name _malloc32 becomes decc$malloc, but the function name _xyz32 does not change.

Implementation

To illustrate changes that needed to be made to the header files, we invented a single header file called <header.h>. This file, which is shown in Figure 1, illustrates the classes of problems faced by a developer who is adding support for 64-bit pointers. The functions defined in this header file are actual C run-time library functions.

Preparing the Header File

The first pass through <header.h> resulted in a number of changes in terms of formatting, commenting, and 64-bit support. Realizing that many modifications would be made to the header files, we considered readability a major goal for this release of these files.

The initial header files assumed a uniform pointer size of 32 bits for the OpenVMS operating system. During the first pass through <header.h>, we added pointer-size pragmas to ensure that the file saved the user's pointer size, set the pointer size to 32 bits, and then restored the user's pointer size at the end of the header.

Next we formatted <header.h> to show the various categories that the structures and functions fall into. The categories and the result of the first pass through <header.h> can be seen in Figure 2. For example, the function rand had no pointers in the function prototype and was immediately moved to the section "Functions that support 64-bit pointers."

Organizing <header.h> in this way gave us an accurate reading of how many more functions needed 64-bit support. If any of the sections became empty, we did not remove the section. This approach worked well because while some engineers were doing 64-bit work, others were adding new functions. Any new functions added to a header file after the 64-bit work was done would be placed in the section "Functions that need 64-bit support." Prior to shipping the header files, we removed the empty sections.

Preparing the Source Code

After several false starts, we settled on a design for modifying the source code for 64-bit support. The expected starting design was to modify the source code by adding pointer_size pragmas and compile the source modules using the /pointer_size qualifier. Some modules would use /pointer_size=32; others would use /pointer_size=64. The major drawback to this approach was that looking at a variable declared as a pointer requires an understanding of the context in which that variable appears. No longer would "char *" be simply a character pointer. It could be a 32-bit or a 64-bit character pointer, and the implementer needed to know which one.

The design on which we decided overcomes the readability problem. By default, source files are not compiled with the /pointer_size qualifier. This means that no pointer-size manipulation occurs when including the header files. The readability of the source code is improved in that the implementers can see which pointers are 32-bit pointers and which are 64-bit pointers.

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

int    execv(const char *, char *[]);
void   free(void *);
void  *malloc(size_t);
int    rand(void);
char  *strcat(char *, const char *);
char  *strerror(int);
size_t strlen(const char *);

#endif /* __HEADER_LOADED */
```

Figure 1
Original Header File <header.h>

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

/*
** Ensure that we begin with 32-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  if (__VMS_VER < 70000000)
#     error "Pointer size added in OpenVMS V7.0 for Alpha"
#  endif
#  pragma __pointer_size __save
#  pragma __pointer_size 32
#endif

/*
** STRUCTURES NOT AFFECTED BY POINTERS
*/
#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

/*
** FUNCTIONS THAT NEED 64-BIT SUPPORT
*/
int    execv(const char *, char *[]);
void   free(void *);
void  *malloc(size_t);
char  *strcat(char *, const char *);
char  *strerror(int);
size_t strlen(const char *);

/*
** Create 32-bit header file typedefs.
*/

/*
** Create 64-bit header file typedefs.
*/

/*
** FUNCTIONS RESTRICTED FROM 64 BITS
*/

/*
** Change default to 64-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size 64
#endif

/*
** FUNCTIONS THAT SUPPORT 64-BIT POINTERS
*/
int rand(void);

/*
** Restore the user's pointer context.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore
#endif

#endif /* __HEADER_LOADED */
```

Figure 2
First Pass through <header.h>

We created a C run-time library private header file called <wide_types.src>. This header file has the appropriate pragmas to define 64-bit pointer types used within the C run-time library, as shown in Figure 3.

This header file also contains the definitions of macros used in the implementations of the functions. Figure 4 shows the macros declared in <wide_types.src>.

Once a module includes the file <wide_types.src>, the compilation of that module changes to add the qualifier /pointer_size=32. This change improves the readability of the code because "char *" is read as a 32-bit character pointer, whereas 64-bit pointers use typedefs whose names begin with "__wide." The name of the new typedef is __wide_char_ptr, which is read as a 64-bit character pointer.

The C run-time library design also requires that the implementation of a function include all header files that define the function. This ensures that the implementation matches the header files as they are modified to support 64-bit pointers. For functions defined in multiple header files, this ensures that header files do not contradict each other.

```
/*
** This include file defines all 32-bit and 64-bit data types used in
** the implementation of 64-bit addresses in the C run-time library.
**
** Those modules that are compiled with a 64-bit-capable compiler
** are required to enable pointer size with /POINTER_SIZE=32.
*/
#ifdef __INITIAL_POINTER_SIZE
#  if (__INITIAL_POINTER_SIZE != 32)
#     error "This module must be compiled /pointer_size=32"
#  endif
#endif

/*
** All interfaces that require 64-bit pointers must use one of
** the following definitions. When this header file is used on
** platforms not supporting 64-bit pointers, these definitions
** will define 32-bit pointers.
*/
#ifdef __INITIAL_POINTER_SIZE
#  pragma __pointer_size __save
#  pragma __pointer_size 64
#endif

typedef char *__wide_char_ptr;
typedef const char *__wide_const_char_ptr;

typedef int *__wide_int_ptr;
typedef const int *__wide_const_int_ptr;

typedef char **__wide_char_ptr_ptr;
typedef const char **__wide_const_char_ptr_ptr;

typedef void *__wide_void_ptr;
typedef const void *__wide_const_void_ptr;

#include <curses.h>
typedef WINDOW *__wide_WINDOW_ptr;

#include <string.h>
typedef size_t *__wide_size_t_ptr;

/*
** Restore pointer size.
*/
#ifdef __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore
#endif
```

Figure 3
Typedefs from <wide_types.src>

```
/*
** Define macros that are used to determine pointer size and
** macros that will copy from high memory onto the stack.
*/
#ifdef __INITIAL_POINTER_SIZE

#  include <builtins.h>

#  define C$$IS_SHORT_ADDR(addr) \
     ((((__int64)(addr)<<32)>>32) == (unsigned __int64)addr)

#  define C$$SHORT_ADDR_OF_STRING(addr) \
     (C$$IS_SHORT_ADDR(addr) ? (char *) (addr) \
     :(char *) strcpy(__ALLOCA(strlen(addr) + 1), (addr)))

#  define C$$SHORT_ADDR_OF_STRUCT(addr) \
     (C$$IS_SHORT_ADDR(addr) ? (void *) (addr) \
     :(void *) memcpy(__ALLOCA(sizeof(* addr)), (addr), sizeof(*addr)))

#  define C$$SHORT_ADDR_OF_MEMORY(addr, len) \
     (C$$IS_SHORT_ADDR(addr) ? (void *) (addr) \
     :(void *) memcpy(__ALLOCA(len), (addr), len))

#else

#  define C$$IS_SHORT_ADDR(addr) (1)
#  define C$$SHORT_ADDR_OF_STRING(addr) (addr)
#  define C$$SHORT_ADDR_OF_STRUCT(addr) (addr)
#  define C$$SHORT_ADDR_OF_MEMORY(addr, len) (addr)

#endif
```

Figure 4
Macros from <wide_types.src>

Implementing the strerror Return Pointer

The function strerror always returns a 32-bit pointer. The memory is allocated by the C run-time library for both 32-bit and 64-bit calling programs. As shown in Figure 5, we moved the function strerror into the section "Functions that support 64-bit pointers" of <header.h> to show that there are no restrictions on the use of this function.

The "Create 32-bit header file typedefs" section of <header.h> is in the 32-bit pointer section, where the bound-to-low-memory data structures are declared. The function returns a pointer to a character string. We, therefore, added typedefs for __char_ptr32 and __const_char_ptr32 while in a 32-bit pointer context. These declarations are protected with the definition of __CHAR_PTR32 to allow multiple header files to use the same naming convention. Declarations of the const form of the typedef are always made in the same conditional code since they usually are needed and using the same condition removes the need for a different protecting name.

The strerror function could have been implemented in <header.h> by placing the function in the 32-bit section, but that would have implied that the 32-bit pointer was a restriction that could be removed later. The pointer is not a restriction, and the strerror function fully supports 64-bit pointers.

The private header file typedefs are always declared starting with two underscores and ending in either "_ptr32" or "_ptr64." These typedefs are created only when the header file needs to be in a particular pointer-size mode while referring to a pointer of the other size. The return value of strerror is modified to use the typedef __char_ptr32.

Including the header file, which declares strerror, allows the compiler to verify that the arguments, return values, and pointer sizes are correct.

Widening the strlen Argument

The function strlen accepts a constant character pointer and returns an unsigned integer (*size_t*). Implementing full 64-bit support in strlen means changing the parameter to a 64-bit constant character pointer. If an application passes a 32-bit pointer to the strlen function, the compiler-generated code sign extends the pointer. The required header file modification is to simply move strlen from the section "Functions that need 64-bit support" to the section "Functions that support 64-bit pointers."

The steps necessary for the source code to support 64-bit addressing are as follows:

1. Ensure that the module includes header files that declare strlen.

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

/*
** Ensure that we begin with 32-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  if (__VMS_VER < 70000000)
#     error "Pointer size added in OpenVMS V7.0 for Alpha"
#  endif
#  pragma __pointer_size __save
#  pragma __pointer_size 32
#endif

/*
** STRUCTURES NOT AFFECTED BY POINTERS
*/
#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

/*
** FUNCTIONS THAT NEED 64-BIT SUPPORT
*/

/*
** Create 32-bit header file typedefs.
*/
#ifndef __CHAR_PTR32
#  define __CHAR_PTR32 1
   typedef char *__char_ptr32;
   typedef const char *__const_char_ptr32;
#endif

/*
** Create 64-bit header file typedefs.
*/
#ifndef __CHAR_PTR64
#  define __CHAR_PTR64 1
#  pragma __pointer_size 64
   typedef char *__char_ptr64;
   typedef const char *__const_char_ptr64;
#  pragma __pointer_size 32
#endif

/*
** FUNCTIONS RESTRICTED FROM 64 BITS
*/
int execv(__const_char_ptr64, char *[]);

/*
** Change default to 64-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size 64
#endif

/*
** The following functions have interfaces of XXX, _XXX32,
** and _XXX64.
**
** The function strcat has two interfaces because the return
** argument is a pointer that is relative to the first arguments.
**
** The malloc function returns either a 32-bit or a 64-bit
** memory address.
*/
#if __INITIAL_POINTER_SIZE == 32
# pragma __pointer_size 32
#endif
```

Figure 5
Final Form of <header.h>

```
void *malloc(size_t __size);
char *strcat(char *__s1, __const_char_ptr64 __s2);

#if __INITIAL_POINTER_SIZE == 32
#  pragma __pointer_size 64
#endif

#if __INITIAL_POINTER_SIZE && __VMS_VER >= 70000000
#  pragma __pointer_size 32
   void *_malloc32(size_t);
   char *_strcat32(char *__s1, __const_char_ptr64 __s2);
#  pragma __pointer_size 64
   void *_malloc64(size_t);
   char *_strcat64(char *__s1, const char *__s2);

#endif

/*
** FUNCTIONS THAT SUPPORT 64-BIT POINTERS
*/
void   free(void *__ptr);
int    rand(void);
size_t strlen(const char *__s);

__char_ptr32 strerror(int __errnum);

/*
** Restore the user's pointer context.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore

#endif

#endif /* __HEADER_LOADED */
```

Figure 5
Continued

2. Add the following line of code to the top of the module: `#include <wide_types.src>`.
3. Change the declaration of the function to accept a __wide_const_char_ptr parameter instead of the previous const char * parameter.
4. Visually follow this argument through the code, looking for assignment statements. This particular function would be a simple loop. If local variables store this pointer, they must also be declared as __wide_const_char_ptr.
5. Compile the source code using the directive /warn=enable=maylosedata to have the compiler help detect pointer truncation.
6. Add a new test to the test system to exercise 64-bit pointers.

Restricting execv from High Memory

Examination of the execv function prototype showed that this function receives two arguments. The first argument is a pointer to the name of the file to start. The second argument represents the argv array that is to be passed to the child process. This array of pointers to null terminated strings ends with a NULL pointer.

Initially, the execv function was to have had two implementations. The parameters passed to the execv function are used as the parameters to the main function of the child process being started. Because no assumptions could be made about that child process (in terms of support for 64-bit pointers), these parameters are restricted to low memory addresses.

To illustrate that the argv passing was a restriction, we place that prototype into the section "Functions restricted from 64 bits" of <header.h>. The first argument, the name of the file, did not need to have this restriction. The section "Create 64-bit header file typedefs" was enhanced to add the definition of __const_char_ptr64, which allows the prototypes to define a 64-bit pointer to constant characters while in either 32-bit or 64-bit context.

Returning a Relative Pointer in strcat

The strcat function returns a pointer relative to its first argument. We looked at this function and determined that it required two entry points. In addition, we widened the second parameter, which is the address of the string to concatenate to the second, to allow the application to concatenate a 64-bit string to a 32-bit string without source code changes.

Figure 5 shows the changes made to support functions that have pointer-size-specific entry points. The prototypes of functions XXX, _XXX32, and _XXX64 begin in 64-bit pointer-size mode. Since the unmodified function name (strcat, XXX) is to be in the pointer size specified by the /pointer_size qualifier, the pointer size is changed from 64 bits to 32 bits if and only if the user has specified /pointer_size=32. At this point, we are not certain of the pointer size in effect. We know only that the size is the same as the size of the qualifier. The second argument to strcat uses the __const_char_ptr64 typedef in case we are in 32-bit pointer mode. Notice the declaration of _strcat64 does not use this typedef because we are guaranteed to be in 64-bit pointer context. Figure 6 shows the implementation of both the 32-bit and the 64-bit strcat functions.

The 64-bit malloc Function

The implementation of multiple entry points was discussed and demonstrated in the strcat implementation. Although multiple entry points are typically added to avoid truncating pointers, functions such as memory allocation routines have newly defined behavior.

The functions decc$malloc and decc$_malloc64 use new support provided by the OpenVMS Alpha operating system for allocating, extending, and freeing 64-bit virtual memory. The C run-time library utilizes this new functionality through the LIBRTL entry points. The LIBRTL group added new entry points for each of the existing memory management functions. The LIBRTL includes an additional second entry point for the free function. Since our implementation of the free function simply widens the pointer, we end up with a single, C run-time library function that must choose which LIBRTL function to call.

```
int free(__wide_void_ptr ptr) {
   if (!(C$$IS_SHORT_ADDR(ptr)))
       return(c$$_free64(ptr));
   else return(c$$_free32((void *) ptr);
}
```

Concluding Remarks

The project took approximately seven person-months to complete. The work involved two months to determine what we wanted to do, one month to figure out how we were going to do it, and four person-months to modify, document, and test the software.

During the initial two months, the technical leaders met on a weekly basis and discussed the overall approach to adding 64-bit pointers to the OpenVMS environment. Since I was the technical lead for the C run-time library project, this initial phase occupied between 25 and 50 percent of my time.

The one month of detailed analysis and design consumed more than 90 percent of my time and resulted in a detailed document of approximately 100 pages. The document covered each of the 50 header files and 500 function interfaces. The functions were grouped by type, based on the amount of work required to support 64-bit pointers.

The first month of implementation occupied nearly all of my time, as I made several false starts. Once I worked out the final implementation technique, I completed at least two of each type of work. As coding deadlines approached, I taught two other engineers on my team how to add 64-bit pointer support, pointing out those functions already completed for reference. They came up to speed within one week. Together, we completed the work during the final month of the project.

```
#include <string.h>
#include <wide_types.src>

/*
** STRCAT/_STRCAT64
**
** The 'strcat' function concatenates 's2', including the
** terminating null character, to the end of 's1'.
*/

__wide_char_ptr _strcat64(__wide_char_ptr s1, __wide_const_char_ptr s2)
{
  (void) _memcpy64((s1 + strlen(s1)), s2, (strlen(s2) + 1));
  return(s1);
}

char *_strcat32(char *s1, __wide_const_char_ptr s2) {
  (void) memcpy((s1 + strlen(s1)), s2, (strlen(s2) + 1));
  return(s1);
```

Figure 6
Implementation of 32-bit and 64-bit strcat Functions

Acknowledgments

The author would like to acknowledge the others who contributed to the success of the C run-time library project. The engineers who helped with various aspects of the analysis, design, and implementation were Sandra Whitman, Brian McCarthy, Greg Tarsa, Marc Noel, Boris Gubenko, and Ken Cowan. Our writer, John Paolillo, worked countless hours documenting the changes we made to the library.

References

1. M. Harvey and L. Szubowicz, "Extending OpenVMS for 64-bit Addressable Virtual Memory," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 57–71.
2. T. Benson, K. Noel, and R. Peterson, "The OpenVMS Mixed Pointer Size Environment," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 72–82.
3. *DEC C User's Guide for OpenVMS Systems* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PUNZE-TK, 1995).
4. *DEC C Runtime Library Reference Manual for OpenVMS Systems* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PUNEE-TK, 1995).
5. *OpenVMS Calling Standard* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBBA-TE, 1995).

Biography

Duane A. Smith
As a consulting software engineer, Duane Smith is currently architect and project leader of the C run-time library for the OpenVMS VAX and Alpha platforms. He joined Digital in 1981 and has worked on a variety of projects, including the A-to-Z Database Manager and the Language-Sensitive Editor. Duane received his B.S. in engineering from the University of Connecticut in 1981 and his M.S. in software engineering from Wang Institute of Graduate Studies in 1987. He pursued his master's degree through Digital's Graduate Engineering Education Program (GEEP). Duane holds one U.S. patent issued for the DECwindows Structured Visual Navigation (SVN) widget.

Building a High-performance Message-passing System for MEMORY CHANNEL Clusters

James V. Lawton
John J. Brosnan
Morgan P. Doyle
Seosamh D. Ó Riordáin
Timothy G. Reddin

The new MEMORY CHANNEL for PCI cluster interconnect technology developed by Digital (based on technology from Encore Computer Corporation) dramatically reduces the overhead involved in intermachine communication. Digital has designed a software system, the TruCluster MEMORY CHANNEL Software version 1.4 product, that provides fast user-level access to the MEMORY CHANNEL network and can be used to implement a form of distributed shared memory. Using this product, Digital has built a low-level message-passing system that reduces the communications latency in a MEMORY CHANNEL cluster to less than 10 microseconds. This system can, in turn, be used to easily build the communications libraries that programmers use to parallelize scientific codes. Digital has demonstrated the successful use of this message-passing system by developing implementations of two of the most popular of these libraries, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).

During the last few years, significant research and development has been undertaken in both academia and industry in an effort to reduce the cost of high-performance computing (HPC). The method most frequently used was to build parallel systems out of clusters of commodity workstations or servers that could be used as a virtual supercomputer.[1] The motivation for this work was the tremendous gains that have been achieved in reduced instruction set computer (RISC) microprocessor performance during the last decade. Indeed, processor performance in today's workstations and servers often exceeds that of the individual processors in a tightly coupled supercomputer. However, traditional local area network (LAN) performance has not kept pace with microprocessor performance. LANs, such as fiber distributed data interface (FDDI), offer reasonable bandwidth, since communication is generally carried out by means of traditional protocol stacks such as the user datagram protocol/internet protocol (UDP/IP) or the transmission control protocol/internet protocol (TCP/IP), but software overhead is a major factor in message-transfer time.[2] This software overhead is not reduced by building faster LAN network hardware. Rather, a new approach is needed—one that bypasses the protocol stack while preserving sequencing, error detection, and protection.

Much current research is devoted to reducing this communications overhead using specialized hardware and software. To this end, Digital has been working to make commercial Alpha clusters, descended from the original VAXcluster technology, available to scientific and technical users.[3,4] This cluster technology uses available commodity hardware and software to implement a high-performance communications subsystem.[5] The hardware interconnect that supports clustered operation is Encore Computer Corporation's patented MEMORY CHANNEL technology.[6] This interconnect provides a mechanism that allows the virtual address space of a process to be mapped so that a store instruction in one system is directly reflected in the physical memory of another system. We have developed software application programming interfaces (APIs) that provide user-level applications with this capability in a controlled and protected manner.

Data may then be transferred between the machines using simple memory read and write operations, with no software overhead, essentially utilizing the full performance of the hardware. This approach is similar to the one used in the Princeton SHRIMP project, where this process is described as Virtual Memory-Mapped Communication (VMMC).[7–10]

Figure 1 shows the relationship between the various components of our message-passing system. The first phase of our work involved designing a programming library and associated kernel components to provide protected, unprivileged access to the MEMORY CHANNEL network. Our objective in creating this library was to provide a facility much like the standard System V interprocess communication (IPC) shared memory functions available in UNIX implementations. Programmers could use the library to set up operations over the MEMORY CHANNEL interconnect, but they would not need to use the library functions for data transfer. In this way, performance could be maximized. This product, the TruCluster MEMORY CHANNEL Software, provides programmers with a simple, high-performance mechanism for building parallel systems.

TruCluster MEMORY CHANNEL Software delivers the performance available from the MEMORY CHANNEL network directly to user applications but requires a programming style that is different from that required for shared memory. This different programming style is necessary because of the different access characteristics between local memory and memory on a remote node connected through a MEMORY CHANNEL network. To make programming with the MEMORY CHANNEL technology relatively simple while continuing to deliver the hardware performance, we built a library of primitive communications functions. This system, called Universal Message Passing (UMP), hides the details of MEMORY CHANNEL operations from the programmer and operates seamlessly over two transports (initially): shared memory and the MEMORY CHANNEL interconnect. This allows seamless growth from a symmetric multiprocessor (SMP) to a full MEMORY CHANNEL cluster. Development can be done on a workstation, while production work is done on the cluster. The UMP layer was designed from the beginning with performance considerations in mind, particularly with respect to minimizing the overhead involved in sending small messages.

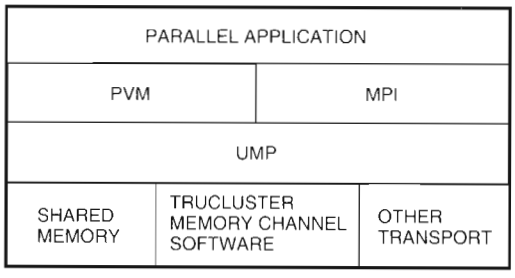


Figure 1
Message-passing System Architecture

Two distributed memory models are predominantly used in high-performance computing today:

1. Data parallel, which is used in High Performance Fortran (HPF).[11] With this model, the programmer uses parallel language constructs to indicate to the compiler how to distribute data and what operations should be performed on it. The problem is assumed to be regular so that the compiler can use one of a number of data distribution algorithms.
2. Message passing, which is used in Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).[12–15] In this approach, all messaging is performed explicitly, so the application programmer determines the data distribution algorithm, making this approach more suitable for irregular problems.

It is not yet clear whether one of these approaches will predominate in the future or if both will continue to coexist. Digital has been working to provide competitive solutions for both approaches using MEMORY CHANNEL clusters. Digital's HPF work has been described in a previous issue of the *Journal*.[16,17] This paper is primarily concerned with message passing.

Building on the UMP layer, we constructed implementations of two common message-passing systems. The first, PVM, is a de facto standard for programmers who want to parallelize large scientific and technical applications. In addition to messaging functions, PVM also provides process control functions. The second, MPI, represents the efforts of a large group of academic and industrial users who are working together to specify a standard API for message passing. At this time, MPI does not provide any process control facilities. The performance of these PVM and MPI systems on MEMORY CHANNEL clusters exceeds that of the public-domain implementations.

MEMORY CHANNEL Overview

Encore's MEMORY CHANNEL technology is a high-performance network that implements a form of clusterwide shared virtual memory. In Digital's first implementation of this technology, it is a shared, 100-megabyte-per-second (MB/s) bus that provides a write-only path from a page of virtual address space on one node to a page of physical memory on another node (or multiple other nodes). The MEMORY CHANNEL network outperforms any traditional LAN technology that uses a bus topology. For example, a peak bandwidth of between 35 MB/s and 70 MB/s is possible with the current 32-bit peripheral component interconnect (PCI) MEMORY CHANNEL adapters,

depending on the bandwidth of the I/O subsystem into which the adapter is plugged. Although the current MEMORY CHANNEL network is a shared bus, the plan for the next generation is to utilize a switched technology that will increase the aggregate bandwidth of the network beyond that of currently available switched LAN technologies. The latency (time to send a minimum-length message one way between two processes) is less than 5 microseconds (μs). The MEMORY CHANNEL network provides a communications medium with a low bit-error rate, on the order of 10^{-16}. The probability of undetected errors occurring is so small (on the order of the undetected error rate of CPUs and memory subsystems) that it is essentially negligible. A MEMORY CHANNEL cluster consists of one or more PCI MEMORY CHANNEL adapters on each node and a hub connecting up to eight nodes.

The MEMORY CHANNEL cluster supports a 512-MB global address space into which each adapter, under operating system control, can map regions of local virtual address space.[18] Figure 2 illustrates the MEMORY CHANNEL operation. Figure 2a shows transmission, and Figure 2b shows reception. A page table entry (PTE) is an entry in the system virtual-to-physical map that translates the virtual address of a page to the corresponding physical address. The MEMORY CHANNEL adapter contains a page control table (PCT) that indicates for each page of MEMORY CHANNEL global address space if that page is mapped locally and whether it is mapped for transmission or reception. Thus, to map a page of local virtual memory for transmission, all that is required is to

- Set up an entry in the system virtual-to-physical map to point to a page in the MEMORY CHANNEL adapter's PCI I/O address space window, which is directly mapped to the page in MEMORY CHANNEL space
- Enable the corresponding page entry in the PCT for transmission

Any write to the mapped virtual page will then result in a corresponding write to the MEMORY CHANNEL network.

To complete the circuit, the page of MEMORY CHANNEL space must be mapped to virtual memory on another node. This is accomplished on the other node by

- Making a page of physical memory nonpageable (wired)
- Creating a virtual region whose PTE points to the wired page
- Setting up the I/O direct memory access (DMA) scatter/gather map to point to the physical page
- Enabling the appropriate entry in the adapter's PCT for reception

Thus, when a MEMORY CHANNEL network packet is received that corresponds to the page that is mapped for reception, the data is transferred directly to the appropriate page of physical memory by the system's DMA engine. In addition, any cache lines that refer to the updated page are invalidated.

Subsequently, any writes to the mapped page of virtual memory on the first node result in corresponding writes to physical memory on the second node. This means that when a region in MEMORY CHANNEL space has been allocated and attached to a process, writes to that region are just simple stores to a process virtual address. The virtual address translates to a physical address that is mapped for transmission. Reads from that region are simply loads from a process virtual address, so the operating system is not involved in data transfer, with consequent reduction in overhead.

To use the MEMORY CHANNEL hardware, the operating system must provide certain basic services. Digital's cluster software includes a set of low-level primitives that can be used in the UNIX kernel. The functionality that these services provide includes

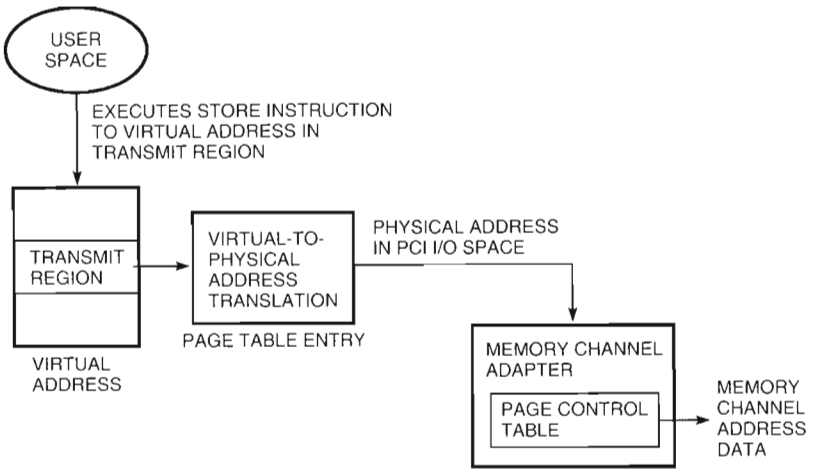
- Allocating and deallocating regions of MEMORY CHANNEL space for transmission or reception
- Allocating and deallocating cluster spinlocks
- Providing the capability to be notified when a page has been written (i.e., a notification channel)

TruCluster MEMORY CHANNEL Software

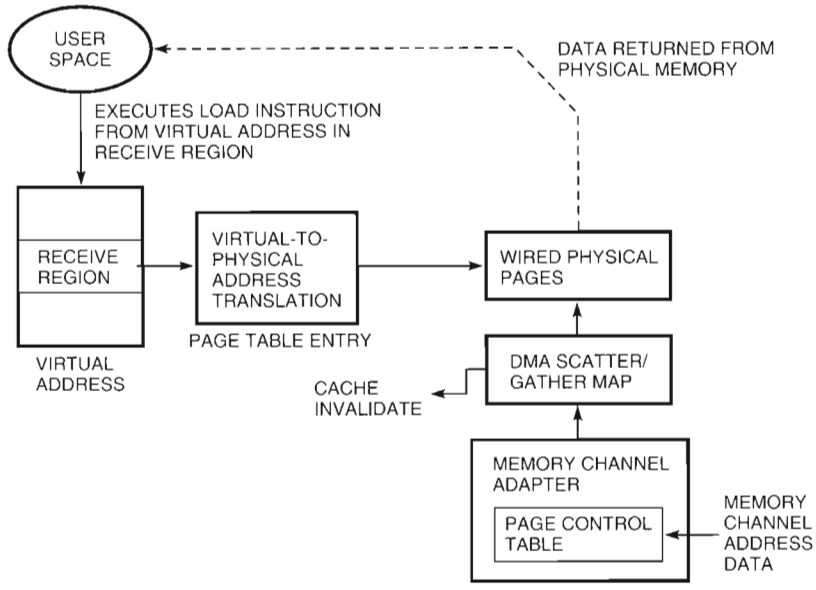
We designed the TruCluster MEMORY CHANNEL Software product to provide user-level access to the kernel functions that control the MEMORY CHANNEL hardware. The target audience for this technology is parallel software library builders and parallel compiler implementers. As shown in Figure 3, the product consists of two components layered on top of the kernel MEMORY CHANNEL functions:

1. A kernel subsystem that interfaces to the low-level kernel functions
2. A user-level API library

There were two choices in developing the product: provide simple user-level access to the basic functionality or build a more sophisticated system (e.g., a distributed shared memory [DSM] system). We chose to make a subset of the functionality of the operating system kernel primitives available to applications for two reasons. First, we did not initially know the degree of functionality required to provide generic user-level access to the MEMORY CHANNEL network for the long term. Second, the original purpose of the work was to give scientific and technical customers, rather than commercial cluster users, early access to the MEMORY CHANNEL network. As a result, the functionality we built into the product is



(a) Transmission



(b) Reception

Figure 2
MEMORY CHANNEL Operation

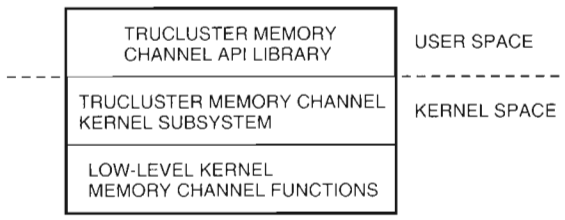


Figure 3
TruCluster MEMORY CHANNEL Software Architecture

a set of simple building blocks that are analogous to the System V IPC facility in most UNIX implementations. The advantage is that while a very simple interface is provided initially, the interface can later be extended as required, without losing compatibility with applications based on the initial implementation. Table 1 details the MEMORY CHANNEL API library functions that the product provides. An important feature to note is that when a MEMORY CHANNEL region is allocated using TruCluster MEMORY CHANNEL Software, a key is specified that uniquely identifies this region in the cluster. Other processes anywhere in the cluster can attach to the same region using the same key; the collection of keys provides a clusterwide namespace.

The MEMORY CHANNEL API library communicates with the kernel subsystem using kmodcall, a simple generic system call used to manage kernel subsystems. The library function constructs a command block containing the type of command (i.e.,

Table 1
TruCluster MEMORY CHANNEL API Library Functions

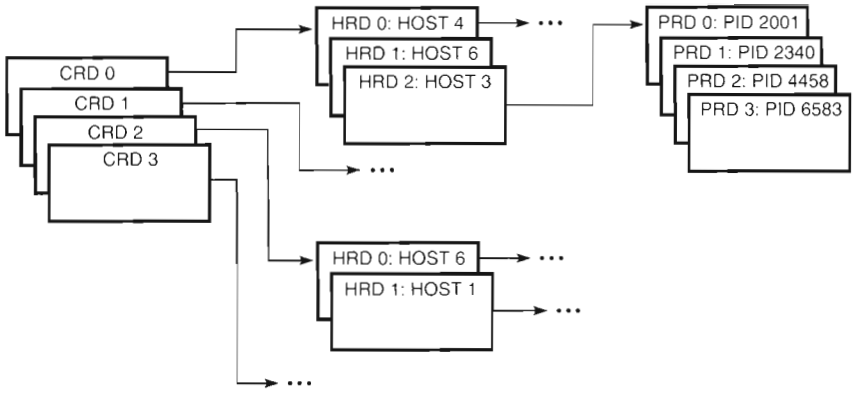
Function Name	Description
imc_asalloc	Allocates a region of MEMORY CHANNEL address space of a specified size and permissions and with a user-supplied key; the ability to specify a key allows other cluster processes to rendezvous at the same region. The function returns to the user a clusterwide ID for this region.
Imc_asattach	Attaches an allocated MEMORY CHANNEL region to a process virtual address space. A region can be attached for transmission or reception, and in shared or exclusive mode. The user can also request that the page be attached in loopback mode, i.e., any writes will be reflected back to the current node so that if an appropriate reception mapping is in effect, the result of the writes can be seen locally. The virtual address of the mapped region is assigned by the kernel and returned to the user.
Imc_asdetach	Detaches an allocated MEMORY CHANNEL region from a process virtual address space.
imc_asdealloc	Deallocates a region of MEMORY CHANNEL address space with a specified ID.
imc_lkalloc	Allocates a set of clusterwide spinlocks. The user can specify a key and the required permissions. Normally, if a spinlock set exists, then this function just returns the ID of that lock set; otherwise it creates the set. If the user specifies that creation is to be exclusive, then failure will result if the spinlock set exists already. In addition, by specifying the IMC_CREATOR flag, the first spinlock in the set will be acquired. These two features prevent the occurrence of races in the allocation of spinlock sets across the cluster.
imc_lkacquire	Acquires (locks) a spinlock in a specified spinlock set.
imc_lkrelease	Releases (unlocks) a spinlock in a specified spinlock set.
imc_lkdealloc	Deallocates a set of spinlocks.
imc_rderrcnt	Reads the clusterwide MEMORY CHANNEL error count and returns the value to the user. This value is not guaranteed to be up-to-date for all nodes in the cluster. It can be used to construct an application-specific error-detection scheme.
imc_ckerrcnt	Checks for outstanding MEMORY CHANNEL errors, i.e., errors that have not yet been reflected in the clusterwide MEMORY CHANNEL error count returned by imc_rderrcnt. This function checks each node in the cluster for any outstanding errors and updates the global error count accordingly.
imc_kill	Sends a UNIX signal to a specified process on another node in the cluster.
imc_gethosts	Returns the number of nodes currently in the cluster and their host names.

which library function has been called) and any parameters and sends it to the kernel subsystem using kmodcall. The kernel subsystem has a matching function for each of the library calls. When a command block is received, it is parsed and the appropriate function is called to service the request. All security and resource checks are performed inside the kernel.

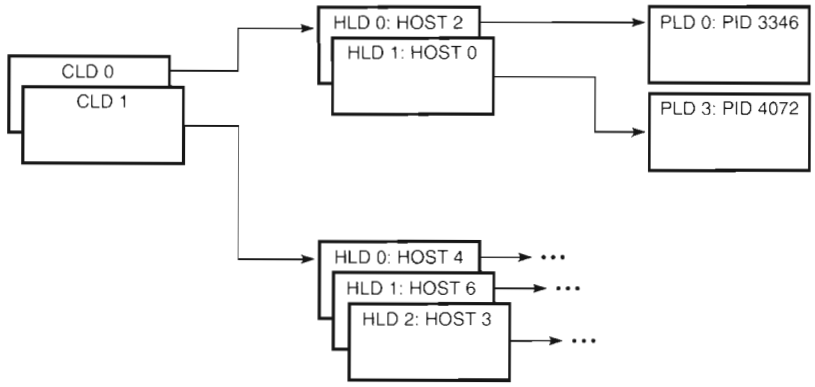
Figure 4 shows some of the data structures that the kernel services use. A clusterwide region of MEMORY CHANNEL space is allocated to store these management structures. This region contains a control structure and six linked lists of descriptors. The control structure manages MEMORY CHANNEL resources allocated using TruCluster MEMORY CHANNEL Software. Each region of MEMORY CHANNEL address space and each set of MEMORY CHANNEL spinlocks allocated using the product have a corresponding descriptor in the kernel data structure.

For each region of MEMORY CHANNEL address space allocated in the cluster, there is a cluster region descriptor (CRD) that contains information describing the region, including its clusterwide region identification number (ID), its size, key, permissions, creation time, and the UNIX user ID (UID) and group ID (GID) of the creating process. For an individual CRD, there is a host region descriptor (HRD) for each node that has the region mapped. This HRD contains the cluster ID of the node and other node-specific information. Finally, for a specific HRD, there is a process region descriptor (PRD) for each process on that node that is using the region. The PRD contains the UNIX process ID (PID) of the process that created the region and any process-specific information, such as virtual addresses.

Similarly, for each set of spinlocks allocated on the cluster there is a cluster lock descriptor (CLD) that contains information describing the spinlock set, including its clusterwide lock ID, the number of spinlocks in the set, the key, permissions, creation time, and the UID and GID of the creating process. For an individual CLD, there is a host lock descriptor (HLD) for each node that is using the spinlock set. The HLD contains the cluster ID of the node and other node-specific information about the spinlock set. For a specific HLD, there is a process lock descriptor (PLD) for each process on that node that is using the spinlock



(a) Regions



(b) Spinlocks

KEY:

CLD	CLUSTER LOCK DESCRIPTOR
CRD	CLUSTER REGION DESCRIPTOR
HLD	HOST LOCK DESCRIPTOR
HRD	HOST REGION DESCRIPTOR
PLD	PROCESS LOCK DESCRIPTOR
PRD	PROCESS REGION DESCRIPTOR

Figure 4
TruCluster MEMORY CHANNEL Kernel Data Structures

set. The PLD contains the PID of the process that created the spinlock set and any process-specific information about the spinlock set.

All these cluster data structures have pointers that cannot be updated atomically. In our implementation, they actually consist of two copies (old and new) and a toggle that indicates which of the two copies is valid. The toggle is switched from an old copy to a new copy only when the new copy is known to be consistent, so that failure of a cluster member while modifying the structures can be tolerated.

Figure 4a illustrates a hypothetical situation in which four regions of MEMORY CHANNEL space have been allocated on the cluster. The first region, with descriptor CRD 0, is mapped on three nodes: host 4, host 6, and host 3. The diagram also shows four processes on host 3 with the region mapped and lists the PID of each process. Figure 4b shows a similar situation for spinlocks. Two sets of spinlocks have been allocated. The first, with descriptor CLD 0, is mapped on two nodes of the cluster: host 2 and host 0. One process on each of these nodes is currently using the spinlock set.

Command Relay

The command relay is a kernel-level framework that enables the execution of a generic service routine on another node within the cluster. It functions as a simple kernel remote procedure call (RPC) mechanism based on fixed unidirectional message locations (mailboxes) and MEMORY CHANNEL notification channels to awaken the server kernel thread. Figure 5 shows the major components of the command relay and illustrates its operation between two hosts in a cluster. A client kernel thread on one host invoking a service and the corresponding server kernel thread on another cluster host communicate data using a defined bidirectional command/response block, known as a parameter structure. The client and server routines

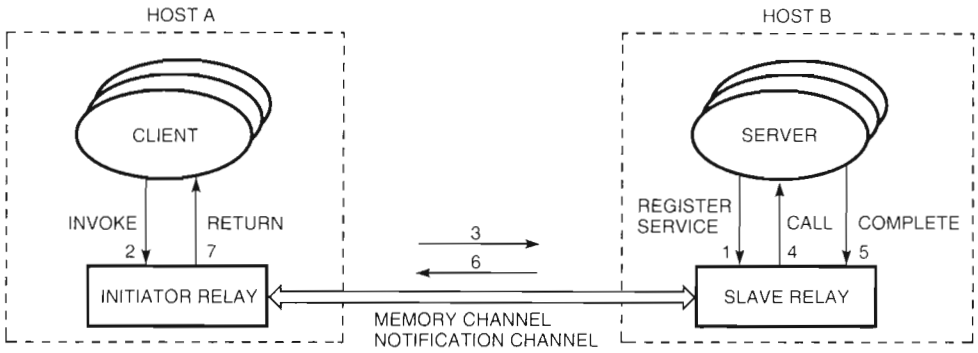


Figure 5
Command Relay Operation

must conform to this interface and must be reliable, i.e., they must always return to the caller. The server can call any kernel function. Server routines are registered (step 1 in Figure 5) using a clusterwide service ID. A kernel thread invoking a remote service passes a packed parameter structure to the command relay, together with a destination node ID and a service ID (step 2). This command relay then adds process credentials and builds a service protocol data unit (SPDU). Using a MEMORY CHANNEL notification channel, it signals the remote node and passes the SPDU by means of a mailbox in MEMORY CHANNEL space (step 3). The server parses the SPDU and calls the requested service function, passing it the parameter structure (step 4). When the service function completes (step 5), its return status and any data values are packed into an SPDU and placed into the mailbox, and the initiating relay is signaled (step 6). The initiator then unpacks the data from the SPDU and returns the appropriate status and values to the client kernel thread (step 7).

All calls to the command relay are synchronous and serialized. The invoking kernel thread blocks until the server returns. Requests to the command relay subsystem are treated on a first-come first-served basis, and calls to a busy relay block until the relay becomes free. Relays are automatically created between all nodes in the cluster.

The command relay mechanism makes it possible to send a UNIX signal to a process on another node within the MEMORY CHANNEL cluster. The imc_kill library function uses the command relay to invoke the registered kernel server routine for cluster signals on the remote node, which, in turn, calls the kernel kill function directly with the PID supplied.

Initial Coherency

When a process on a cluster member maps a region of MEMORY CHANNEL address space for both reception and transmission, any writes to the transmit region by that process are reflected as changes to the corresponding receive region. If another process on another cluster node subsequently maps the same region for reception, the contents of its receive region are indeterminate; i.e., the two processes do not have a coherent view of that region. This situation is known as the initial coherency problem. For an application developer, this problem makes it difficult to treat MEMORY CHANNEL address space as another form of shared memory. Applications can overcome this difficulty by using some form of start-up synchronization. However, all developers would have to implement these solutions separately. To increase the usability of TruCluster MEMORY CHANNEL Software, the design team decided to build in the ability to request coherent allocation of MEMORY CHANNEL address space across the cluster. Developers can specify this as an option in the call to imc_asalloc. As a result, a process can attach a MEMORY CHANNEL region for reception following any updates and still share a common view of the region with other processes in the cluster.

A special process, called the mapper, is used to provide the virtual address space to hold the coherent user space mappings. When the kernel subsystem receives a request for coherent allocation, it allocates the MEMORY CHANNEL region as normal and then maps the region for reception into the virtual address space of the mapper process. The command relay mechanism then causes all the other nodes in the cluster to allocate the same region and map it for reception into the address space of the mapper process on each node. Since multiple user-level processes on a node that attach a particular region for reception share the same physical memory, all updates to the region are seen by late-joining processes on any node in the cluster. If the requesting process exits, the region will still be allocated to the mapper, so that another allocation of the same region on that node will result in a coherent picture of that region. The region is fully deallocated (i.e., from all the mapper processes) when the last application process allocating the region either exits or explicitly deallocates the region.

Given the usefulness of coherent allocations, it may seem unusual that we made this feature an option rather than the default. There are several reasons for this. With coherent allocations, the associated physical memory becomes nonpageable on all nodes within the cluster, and, as such, it consumes physical resources. In addition, every outbound write to such a region results in an inbound write to the physical memory of each node in the cluster. For some application designs, it may be more desirable to create a region that is written by one node and only read by other nodes. Also, automatically reflecting all writes back to a node, as is done for coherent regions, consumes twice as much bandwidth on the PCI bus.

Late Join and Failure Resilience

To provide an operational environment in which nodes can join or leave the cluster at any time, the kernel subsystem needs to overcome a number of problems resulting from late join and node failure. In fact, the kernel subsystem is subject to the same difficulties of initial coherency as application-level processes. To manage user space allocations, late-joining nodes require a coherent view of the cluster data structures. Moreover, failure of an existing node can result in out-of-date or, even worse, corrupt data structures in the subsystem's control region. To contain the failure, corrupt data structures must be repaired.

Low-level kernel routines detect cluster membership change and wake up a management service thread on each node that performs operations local to that node. The first management service thread to acquire a specific spinlock is elected to manage clusterwide updates.

In the case of late join, the management service thread updates local state to reflect the new configuration. The thread that has been designated to manage clusterwide updates is responsible for providing the late-joining node with an up-to-date copy of the cluster data structures. When triggered by the new node, the thread retransmits the contents of the data structures so that the late-joining node has a fully up-to-date view of allocations and resource usage.

When a node fails, the thread elected to manage clusterwide updates must examine the entire management data structure and repair it appropriately. Repair is necessary when the failing node that is in the process of updating the global data structures has left these clusterwide updates in an unstable state. Repair is possible because all updates to global data structures use two copies of the structure (old and new, as described previously), which means that the structures can be reset easily to a stable state. If the failed node was not actively updating the data structures at the time of the failure, the management thread simply removes all resources allocated to the failed node.

Error Management

The MEMORY CHANNEL hardware provides a very low error rate, ordering guarantees, and an ability to detect remote error situations quickly, making it possible to construct simple error detection and recovery protocols. A kernel interrupt service routine detects cluster errors and updates an error counter that reflects the clusterwide error count. A low-level kernel routine returns the value of this counter. Due to timing considerations, it is not possible to guarantee that this count will be up-to-date with respect to possible errors on remote nodes. A low-level kernel routine that efficiently reads the error status of remote MEMORY CHANNEL adapters and detects unprocessed errors is provided. This routine uses a hardware feature, known as an ACK page, that is specifically designed to facilitate error detection. A write to such a page results in the error status of each MEMORY CHANNEL adapter being written to successive locations of the corresponding reception mapped region.

During development, we built simple interfaces to access these low-level routines, thereby allowing message-passing libraries to build in error management. Because the method of getting into and out of the kernel is a generic one, the overhead is high—approximately 30 μs. This compares poorly with the raw latency for short messages, which is less than 5 μs. To provide suitable performance, we reimplemented the functions to execute totally in user space. As a result, when an application reads the error count for the first time (using imc_rderrcnt), the kernel value of the error count is mapped for read-only access into the virtual address space of the process. Subsequent reads of the error count are then simply reads of a memory location. Similarly, when an application calls the check error service (using imc_ckerrcnt) for the first time, ACK pages are transparently mapped into the virtual address space of the process, and the error detection is performed at hardware speeds directly from user space. This has been measured at less than 5 μs.

The following sequence can be used to guarantee detection of intervening errors by the transmitter:

1. Save the error count.
2. Write the message.
3. Check the error count (using imc_ckerrcnt).

If the transmitter writes the saved error count at the end of the message, the message receiver can determine if any intervening errors have occurred by simply comparing the error count in the message with the current value using imc_rderrcnt. This is possible because of the sequencing guarantees built into the MEMORY CHANNEL network. Using imc_rderrcnt and imc_ckerrcnt, the programmer can build an appropriate error detection and/or recovery scheme that meets the performance requirements of the application.

Performance

The performance of TruCluster MEMORY CHANNEL Software on a pair of AlphaServer 4100 5/300 machines is presented in Table 2. These measurements were made using version 1.5 MEMORY CHANNEL adapters. The bandwidth (64 MB/s) and latency (2.9 μs) achieved using this system are essentially that of the hardware, since no system overhead is involved. The times required to perform the error-checking functions indicate that the overhead of calling imc_rderrcnt is much less than that of imc_ckerrcnt. This is because the latter has to synchronize with all other members of the cluster. Protocols that rely on receiver-only error detection (using imc_rderrcnt) will therefore have a lower overhead.

Programming with TruCluster MEMORY CHANNEL Software

The MEMORY CHANNEL network imposes some unique restrictions on the programmer. Since the network requires separate transmit and receive regions, any read-write memory location that is to be visible clusterwide must have two addresses: a read address and a write address. Attempts to read from a write address typically cause a segmentation violation. MEMORY CHANNEL address space can be used like shared memory. Unlike shared memory, though, its latency is visible to the programmer, who must consider latency effects when writing to a clusterwide location.

As an example of programming with TruCluster MEMORY CHANNEL Software, Figure 6 shows a simple program that implements a global counter, performs some work, and then decrements the global counter and exits. For the purposes of this example, assume that multiple copies of the program are run concurrently on different machines in a cluster. Such operation requires synchronization to ensure safe access to shared data in MEMORY CHANNEL space. The example program first allocates MEMORY CHANNEL regions for transmission and reception and attaches them to process virtual addresses. Next, a set of spinlocks is created (unless it already exists). The first copy of the program to create the spinlock set acquires the first lock in the set and initializes the global region, whereupon it releases the spinlock and continues. All other copies of the program wait in imc_lkacquire until the spinlock is released by the first copy. Each copy in turn acquires the lock itself, increments the process counter, and releases the lock. The copies then perform some work in parallel. When each program has finished its portion of the work, it decrements the global process counter (using the spinlock to control access again). Finally, the spinlock set and shared regions are deallocated. Several examples of code illustrating these topics are contained in the *TruCluster MEMORY CHANNEL Software Programmer's Manual*.[10] We have found that implementing a simple message-passing layer on top of TruCluster MEMORY CHANNEL Software is a more effective solution than programming directly with MEMORY CHANNEL regions, as described in the next section.

Table 2
TruCluster MEMORY CHANNEL Software Performance

Sustained bandwidth	64 MB/s
Latency	2.9 μs
Read error count (imc_rderrcnt)	<1 μs
Check error count (imc_ckerrcnt)	<5 μs

Several features described above were not initially present in the TruCluster MEMORY CHANNEL Software product. As a result of our experience implementing UMP and the higher PVM and MPI layers, we added the following features:

- Initial coherency
- Command relay
- Cluster signals
- User-level error checking

Universal Message Passing

The Universal Message Passing (UMP) library is designed to provide a foundation for implementing efficient message-passing systems on the MEMORY CHANNEL network. From the outset, we were aware that there would be a demand for PVM and MPI implementations and that other implementations might follow. We felt that it would be easier to construct high-performance message-passing systems if we provided a thin layer that could efficiently handle the restrictions that the MEMORY CHANNEL network imposes.

The goals in developing UMP were to

- Simplify the construction of message-passing systems utilizing the MEMORY CHANNEL network by hiding the details of the underlying communications transport (initially, shared memory or MEMORY CHANNEL).
- Optimize performance and exploit the low latency of the MEMORY CHANNEL network; the initial goal for latency over the MEMORY CHANNEL network using PVM was to achieve less than 30 μs.
- Ease the development of parallel message-passing libraries by providing a simple set of message-passing functions.
- Perform only basic communications; any more complex operations (e.g., process control) would be performed by a higher layer.
- Act as a convergence center for possible future interconnects.

```
extern long asm(const char *, ...);
#pragma intrinsic(asm)
#define mb() asm("mb")

#include <sys/types.h>
#include <sys/imc.h>

main ()
{
  int status, i, locks=4, temp, errors;
  imc_asid_t region_id;                                  /* MC region ID */
  imc_lkid_t lock_id;                                 /* MC spinlock set ID */
  typedef struct {                                  /* Shared data structure */
    volatile int processes;
    volatile int pattern[2047];
  } shared_region;
  shared_region *region_read, *region_write;
  caddr_t read_ptr = 0, write_ptr = 0;

  /* Allocate a region of coherent MC address space and attach to */
  /* process VA */
  imc_asalloc(123, 8192, IMC_URW, IMC_COHERENT, &region_id);
  imc_asattach(region_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK, &write_ptr);
  imc_asattach(region_id, IMC_RECEIVE, IMC_SHARED, 0, &read_ptr);

  region_read = (shared_region *)write_ptr;
  region_write = (shared_region *)read_ptr;

  /* Allocate a set of spinlocks and atomically acquire the first lock */
  status = imc_lkalloc(456, &locks, IMC_LKU, IMC_CREATOR, &lock_id);
  errors = imc_rderrcnt();
  if (status == IMC_SUCCESS) {
     do {
       region_write->processes = 0;        /* Initialize the global region */
       for (i=0; i<2047; i++)
         region_write->pattern[i] = i;
       i--;
       mb();
     } while (imc_ckerrcnt(&errors) || region_read->pattern[i] != i) ;
    imc_lkrelease(lock_id, 0);
  } else if (status == IMC_EXISTS) {
    imc_lkalloc(456, &locks, IMC_LKU, 0, &lock_id);
    imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT);
    temp = region_read->processes + 1;    /* Increment the process counter */
    errors = imc_rderrcnt();
    do {
      region_write->processes = temp;
      mb();
    } while (imc_ckerrcnt(&errors) || region_read->processes != temp) ;
    imc_lkrelease(lock_id, 0);
  }

       .
       .  (Body of program goes here)
       .

  /* clean up */
  imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT);
  temp = region_read->processes - 1;      /* Decrement the process counter */
  errors = imc_rderrcnt();
  do {
    region_write->processes = temp;
    mb();
  } while (imc_ckerrcnt(&errors) || region_read->processes != temp) ;

  imc_lkrelease(lock_id, 0);
  imc_lkdealloc(lock_id);                          /* Deallocate spinlock set */
  imc_asdetach(region_id);                            /* Detach shared region */
  imc_asdealloc(region_id);                    /* Deallocate MC address space */
}
```

Figure 6
Programming with TruCluster MEMORY CHANNEL Software

These goals placed some important constraints on the architecture of UMP, particularly with regard to performance. This meant that design decisions had to be constantly evaluated in terms of their performance impact. The initial design decision was to use a dedicated point-to-point circular buffer between every pair of processes. These buffers use producer and consumer indexes to control the reading and writing of buffer contents. The indexes can be modified only by the consumer and producer tasks and allow fully lockless operation of the buffers. Removing lock requirements eliminates not only the software costs associated with lock manipulation (in the initial implementation of TruCluster MEMORY CHANNEL Software, acquiring and releasing an uncontested spinlock takes approximately 130 μs and 120 μs, respectively) but also the impact on processor performance associated with Load-locked/Store-conditional instruction sequences.

Although this buffering style eliminates lock manipulation costs, it results in an exponential demand for storage and can limit scalability. If there are N processes communicating using this method, that implies N^2 buffers are required for full mesh communication. MEMORY CHANNEL address space is a relatively scarce resource that needs to be carefully husbanded. To manage the demand on cluster resources as fairly as possible, we decided to do the following:

- Allocate buffers sparsely, i.e., as required up to some default limit. Full N^2 allocation would still be possible if the user increased the number of buffers.
- Make the size of the buffers configurable.
- Use lock-controlled single-writer, multiple-reader buffers to handle both the overflow from the N^2 buffer and fast multicast. One of these buffers, called outbufs, would be assigned to each process using UMP upon initialization.

Note that while the channel buffers are logically point-to-point, they may be implemented physically as either point-to-point or broadcast. For example, in the first version of UMP, we used broadcast MEMORY CHANNEL mappings for the sake of simplicity. We are currently modifying UMP to use point-to-point MEMORY CHANNEL mappings, both to increase available bandwidth and to exploit a switched MEMORY CHANNEL network.

Figure 7 shows several tasks communicating in a cluster and illustrates how the two types of UMP buffers are used. Task 1 and task 2 are executing on node 1, while task 3 is executing on node 2. In the diagram, the channel buffers are located under the task in whose virtual address space they reside to indicate visually that they reside in the virtual address space of the destination task. In the figure, task 1 communicates

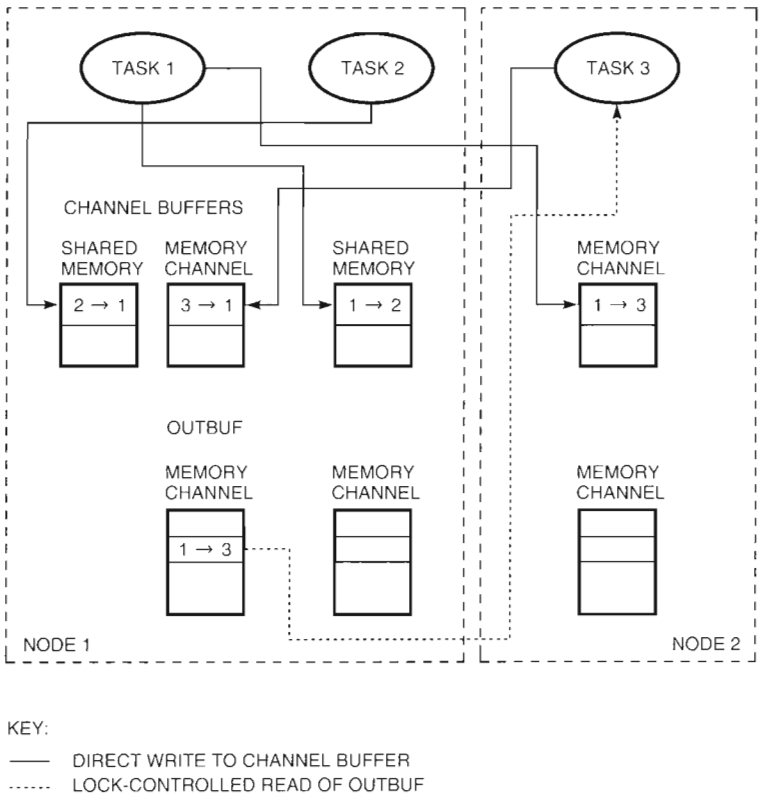


Figure 7
Cluster Communication Using UMP

with task 2 using UMP channel buffers in shared memory, shown as 1→2 and 2→1. Task 1 and task 3 communicate using UMP channel buffers in MEMORY CHANNEL space, shown as 1→3 and 3→1. Task 3 is reading a message from task 1 using an outbuf. The outbuf can be written only by task 1 but is mapped for transmission to all other cluster members. On node 2, the same region is mapped for reception. Access to each outbuf is controlled by a unique cluster spinlock.

Our rationale for taking this approach is that a short software path is more appropriate for small messages because overhead dominates message transfer time, whereas the overhead of lock manipulation is a small component of message transfer time for large messages. We felt that this approach helped to control the use of cluster resources and maintained the lowest possible latency for short messages yet still accommodated large messages. Note that outbufs are still fixed-size buffers but are generally configured to be much larger than the N^2 buffers.

This approach worked for PVM because its message transfer semantics make it acceptable to fail a message send request due to buffer space restrictions (e.g., if both the N^2 buffer and the outbuf are full). When we analyzed the requirements for MPI, however, we found that this approach was not possible. For this reason, we changed the design to use only the N^2 buffers. Instead of writing the message as a single operation, the message is streamed through the buffer in a series of fragments. Not only does this approach support arbitrarily large messages, but it also improves message bandwidth by allowing (and, for messages exceeding the available buffer capacity, requiring) the overlapped writing and reading of the message. Deadlock is avoided by using a background thread to write the message. Since overflow is now handled using the streaming N^2 buffers, outbufs were not necessary to achieve the required level of performance for large messages and were not implemented. Outbufs are retained in the design to provide fast multicast messaging, even though in the current implementation they are not yet supported.

Achieving the performance goals set for UMP was not easy. In addition to the buffer architecture described earlier, several other techniques were used.

- No syscalls were allowed anywhere in the UMP messaging functions, so UMP runs completely in user space.
- Calls to library routines and any expensive arithmetic operations were minimized.
- Global state was cached in local memory wherever possible.
- Careful attention was paid to data alignment issues, and all transfers are multiples of 32-bit data.

At the programmer's level, UMP operation is based on duplex point-to-point links called channels, which correspond to the N^2 buffers already described. A channel is a pair of unidirectional buffers used to provide two-way communication between a pair of process endpoints anywhere in the cluster. UMP provides functions to open a channel between a pair of tasks. While the resources are allocated by the first task to open the channel, the connection is not complete until the second task also opens the same channel. Once a channel has been opened by both sides, UMP functions can be used to send and receive messages on that channel. It is possible to direct UMP to use shared memory or MEMORY CHANNEL address space for the channel buffers, depending on the relative location of the associated processes. In addition, UMP provides a function to wait on any event (e.g., arrival of a message, creation or deletion of a channel). In total, UMP provides a dozen functions, which are listed in Table 3. Most of the functions relate to initialization, shutdown, and miscellaneous operations. Three functions establish the channel connection, and three functions perform all message communications.

UMP channels provide guaranteed error detection but not recovery. Through the use of TruCluster MEMORY CHANNEL Software error-checking routines, we were able to provide efficient error detection in UMP. We decided to let the higher layers implement error recovery. As a result, designers of higher layers can control the performance penalty they incur by specifying their own error recovery mechanisms, or, since reliability is high, can adapt a fail-on-error strategy.

Performance

UMP avoids any calls to the kernel and any copying of data across the kernel boundary. Messages are written directly into the reception buffer of the destination channel. Data is copied once from the user's buffer to physical memory on the destination node by the sending process. The receiving process then copies the data from local physical memory to the destination user's buffer. By comparison, the number of copies involved in a similar operation over a LAN using sockets is greater. In this case, the data has to be copied into the kernel, where the network driver uses DMA to copy it again into the memory of the network adapter. At this point the data is transmitted onto the LAN.

The first version of UMP used one large shared region of MEMORY CHANNEL space to contain its channel buffers and a broadcast mapping to transmit this simultaneously to all nodes in the cluster. This version of UMP also used loopback to reflect transmissions back to the corresponding receive region on the sending node, which resulted in a loss of available bandwidth. Using our AlphaServer 2100 4/190 development machines, we measured

Table 3
UMP API Functions

Function Name	Description
ump_init	Initializes UMP and allocates the necessary resources.
ump_exit	Shuts down UMP and deallocates any resources used by the calling process.
ump_open	Opens a duplex channel between two endpoints over a given transport (shared memory or MEMORY CHANNEL). Channel endpoints are identified by user-supplied, 64-bit integer handles.
ump_close	Closes a specified UMP channel, deallocating all resources assigned to that channel as necessary.
ump_listen	Registers an endpoint for a channel over a specified transport. This can be used by a server process to wait on connections from clients with unknown handles. This function returns immediately, but the channel is created only when another task opens the channel. This can be detected using ump_wait.
ump_wait	Waits for a UMP event to occur, either on one specified channel to this task or on all channels to this task.
ump_read	Reads a message from a specified channel.
ump_write	Writes a message to a specified channel. This function is blocking, i.e., it does not return until the complete message has been written to the channel.
ump_nbread	Starts reading a message from a channel, i.e., it returns as soon as a specified amount of the message has been received, but not necessarily all the message.
ump_nbwrite	Starts writing a message to a specified channel, i.e., it returns as soon as the write has started. A background thread will continue writing the message until it is completely transmitted.
ump_mcast	Writes a message to a specified list of channels.
ump_info	Returns UMP configuration and status information.

- Latency: 11 μs (MEMORY CHANNEL), 4 μs (shared memory)
- Bandwidth: 16 MB/s (MEMORY CHANNEL), 30 MB/s (shared memory)

To increase bandwidth, we modified UMP to use transmit-only regions for its channel buffers, thus eliminating loopback. The performance measured for the revised UMP using the same machines was

- Latency: 9 μs (MEMORY CHANNEL), 3 μs (shared memory)
- Bandwidth: 23 MB/s (MEMORY CHANNEL), 32 MB/s (shared memory)

Figure 8 shows the message transfer time and Figure 9 shows the bandwidth for various message sizes for the revised version of UMP using both blocking and nonblocking writes over shared memory and the MEMORY CHANNEL network. Using newer AlphaServer 4100 5/300 machines, which have a faster I/O subsystem than the older machines, and version 1.5 MEMORY CHANNEL adapters, the measured latency is 5.8 μs (MEMORY CHANNEL), 2 μs (shared memory). The peak bandwidth achieved is 61 MB/s (MEMORY CHANNEL), 75 MB/s (shared memory). In the nonblocking cases, the buffer size used was 256 kilobytes (KB) for shared memory and 32 KB for MEMORY CHANNEL. Further work is under way to improve the performance using shared memory as the transport. This work is aimed at eliminating the high-end falloff in bandwidth in the blocking case and the notch when the message size exceeds the buffer size in the nonblocking case. Note that these effects are not displayed in the MEMORY CHANNEL results.

Message-passing Libraries

Message-passing libraries provide the programmer with a set of facilities to build parallel applications. Typically, these services include the ability to send and receive a variety of data types to and from other peer processes in a variety of modes, as well as collective operations that span a set of peer processes. Other facilities may be provided in addition to the basic set, e.g., PVM provides functions for managing PVM processes (spawning, killing, signaling, etc.), whereas MPI (at least in its first revision, MPI-1) does not. PVM is probably the most widely used message-passing system. It has been available for approximately five years, and implementations are available for a wide variety of platforms. MPI is an emerging standard for message passing that is growing rapidly in popularity; many new applications are being written for it.

Parallel Virtual Machine

Parallel Virtual Machine (PVM) is supported on a wide variety of platforms, including supercomputers and networks of workstations (NOWs). PVM uses a variety of underlying communications methods: shared memory on multiprocessors, various native message-passing systems on massively parallel processors (MPPs), and UDP/IP or TCP/IP on NOWs. The large software overhead in the IP stacks makes it difficult to provide high-performance communications for

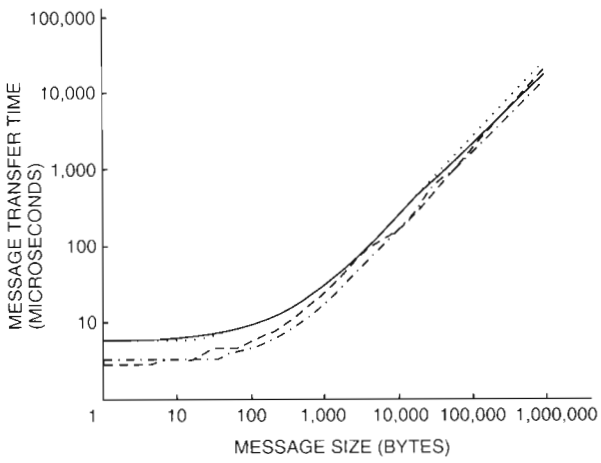


Figure 8
UMP Communications Performance: Message Transfer Time

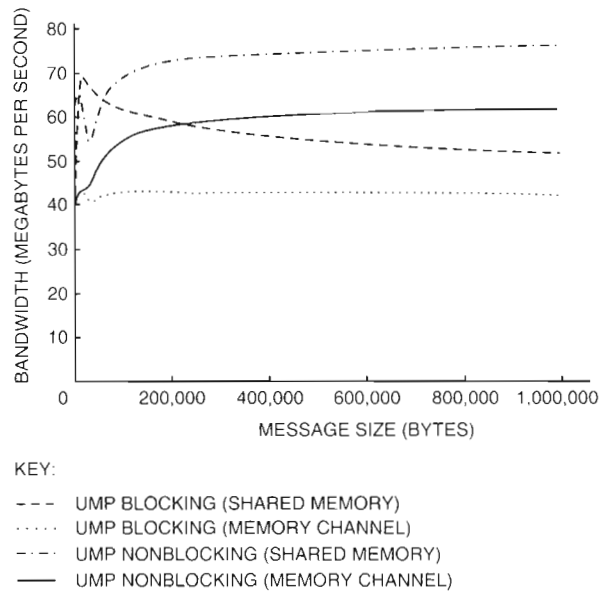


Figure 9
UMP Communications Performance: Bandwidth

PVM when using networks like Ethernet or FDDI. The high cost of communications for these systems means that only the more coarse-grained parallel applications have demonstrated performance improvements as a result of parallelization using PVM. Using the MEMORY CHANNEL cluster technology described earlier, we have implemented an optimized PVM that offers low latency and high-bandwidth communications. The PVM library and daemon use UMP to provide seamless communications over the MEMORY CHANNEL cluster.

When we began to develop PVM for MEMORY CHANNEL clusters, we had one overriding goal: to use the hardware performance the MEMORY CHANNEL interconnect offers to provide a PVM with industry-leading communications performance, specifically with regard to latency. Initially, we set a target latency for PVM of less than 15 μs using shared memory and less than 30 μs using the MEMORY CHANNEL transport.

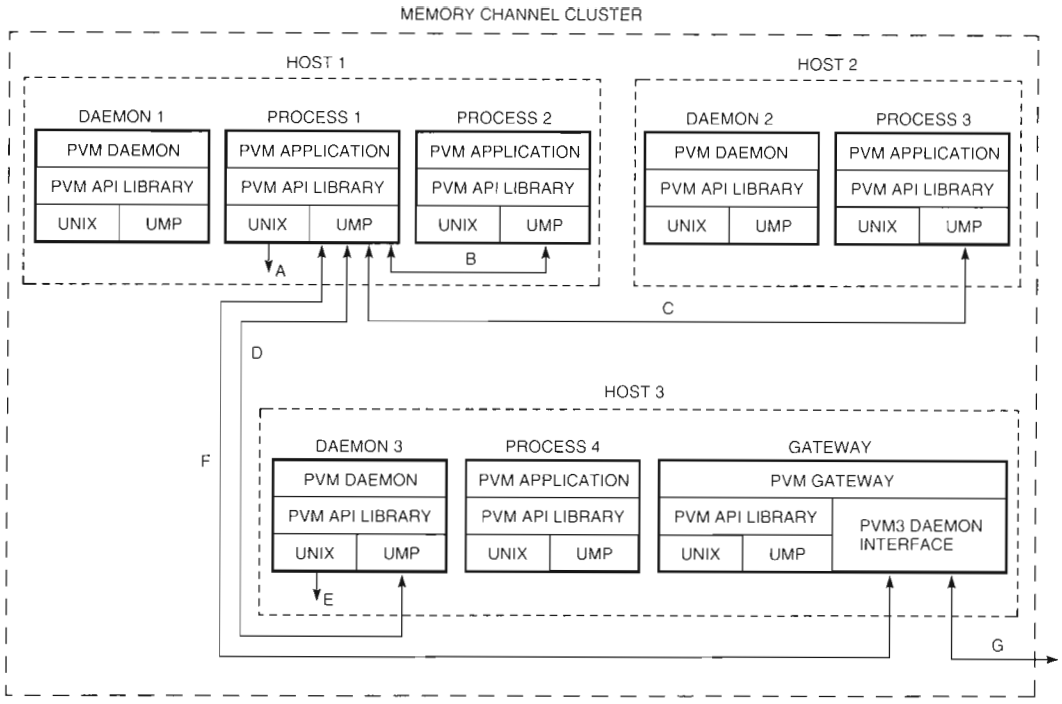
Our first task was to build a prototype using the public-domain PVM implementation. We used an early prototype of the MEMORY CHANNEL system jointly developed by Digital and Encore. The prototype had a hardware latency of 4 μs. We modified the shared-memory version of PVM to use the prototype hardware and achieved a PVM latency of 60 μs. Profiling and straightforward code analysis revealed that most of the overhead was caused by

- PVM's support for heterogeneity (i.e., external data representation [XDR] encoding)
- Messages being copied multiple times inside PVM
- A large number of function calls in the critical communications path
- Inefficient coding of the low-level data copy routines

Since we wanted to achieve the maximum possible performance available from the hardware, we decided to reimplement the PVM library, eliminating support for heterogeneity from the communications functions of PVM and focusing on maximum performance inside a Digital cluster.[20] Heterogeneity would then be supported by using a special PVM gateway process.

The overall architecture of the Digital PVM implementation is shown in Figure 10. To maximize performance, we decided that, wherever possible, an operation should be executed in-line rather than be requested from a remote task or daemon. This contrasts with PVM's traditional approach of relaying such requests to the PVM daemon for service. For example, when a PVM task starts, often it first calls pvm_mytid to request a unique task identifier (TID). Previously, this would have involved sending a message to a PVM daemon, which would then allocate a TID to the process and return another message. In our design, we could use global data structures in MEMORY CHANNEL space (e.g., the list of all PVM tasks and associated data). Now, for example, pvm_mytid simply involves acquiring a cluster lock on a global table, getting the new TID, and releasing the lock—all executed in-line by the calling process rather than a daemon. Executing PVM services in-line with the requesting process increases multiprocessing capability and eliminates daemon bottlenecks and associated delays.

We reimplemented the PVM library with the emphasis on performance rather than heterogeneity, although we plan to eventually allow interoperation with heterogeneous implementations of PVM using a special



KEY:

A A PVM application on host 1 performs local control functions using UNIX signals.

B A PVM application on host 1 communicates with another PVM task on the same host using UMP (via shared memory).

C A PVM application on host 1 communicates with another PVM task on a different host in the cluster (host 2) using UMP (via MEMORY CHANNEL).

D A PVM application on host 1 requires a control function (e.g., a signal) to be executed on another host in the cluster (host 3); it sends a request to a PVM daemon on host 3.

E The PVM daemon on host 3 executes the control function.

F A PVM application on host 1 sends a message to a PVM task on a host outside the MEMORY CHANNEL cluster; the message is routed to the PVM gateway task on host 3.

G The PVM gateway translates the cluster message into a form compatible with the external PVM implementation and forwards the message to the external task via IP sockets.

Figure 10
Digital PVM Architecture

gateway daemon. The PVM API library is a complete rewrite of the standard PVM version 3.3 API, with which full functional compatibility is maintained. Emphasis has been placed on optimizing the performance of the most frequently used code paths. In addition, all data structures and data transfers have been optimized for the Alpha architecture. As stated earlier, the amount of message passing between tasks and the local daemon has been minimized by performing most operations in-line and communicating with the daemon only when absolutely necessary. Intermediate buffers are used for copying data between the user buffers. This is necessary because of the semantics of PVM, which allow operations on buffer contents before and after a message has been sent. The one exception to this is pvm_psend; in this case, data is copied directly since the user is not allowed to modify the send buffer.

The purpose of our PVM daemon is different from that of the daemon in the standard PVM package. Our daemon is designed to relay commands between different nodes in the PVM cluster. It exists solely to perform remote execution of those commands that cannot be performed in-line by UNIX calls in the PVM API library or by directly manipulating global data structures. Commands to be executed on a remote node are sent to the daemon on that node, which then executes the command directly. Note that this removes a level of indirection that exists in standard PVM. Daemon-to-daemon communications are minimized. Since there is no master daemon, the PVM cluster has no single point of failure. All daemons are equal. When not in use, the daemon sleeps, being awakened as required by a signal from the calling task. For a local task, UNIX signals are used. If the task is on another node in the cluster, then MEMORY CHANNEL cluster signals are used. As a result, the daemon uses minimal cluster resources.

The PVM group or collective functions operate on a group of PVM tasks. For example: pvm_barrier synchronizes multiple PVM processes; pvm_bcast sends a message to all members of a particular group; pvm_scatter distributes an array to the members of a group; pvm_gather reassembles the array from the

contributions of each of the group members, etc. The group functions are implemented separately from the other PVM messaging functions. They use a separate global structure (the group table) to manage PVM group data. Access to the group table is controlled by locks. Unlike other PVM implementations, there is no PVM group server, since all group operations can manipulate the group table directly.

Performance

Table 4 compares the communications latency achieved by various PVM implementations. As the table indicates, the latency between two machines with Digital PVM over a MEMORY CHANNEL transport is much less than the latency of the public-domain PVM implementation over shared memory, which validates our approach of removing support for heterogeneity from the critical performance paths. Figure 11 shows the message transfer time and Figure 12 shows the bandwidth for Digital PVM over shared memory and MEMORY CHANNEL transports for various message sizes. Two AlphaServer 4100 5/300 machines were used for these measurements. The peak bandwidth reached by Digital PVM is about 66 MB/s (shared memory) and 43 MB/s (MEMORY CHANNEL). By comparison, PVM 3.3.10 achieves a bandwidth of 24 MB/s (shared memory) and 3 MB/s (FDDI LAN). A version of PVM developed at Digital's Systems Research Center (SRC) using a specially adapted asynchronous transfer mode (ATM) driver achieved a latency of approximately 60 μs and a bandwidth of approximately 16 MB/s using the AN2 ATM LAN.[21] The performance results for PVM latency over the MEMORY CHANNEL transport given in Reference 6 were obtained using an earlier version of Digital PVM. Since those results were measured, latency has been halved, mostly due to improvements in UMP performance.

Figure 13 compares the performance of an unmodified PVM application using the public-domain PVM 3.3.7 implementation and Digital PVM version 1.0. The application is a parallel molecular modeling program. The bar chart shows the elapsed time for a variety of configurations. The application ran for 220 seconds on 2 two-processor SMP machines connected with FDDI. By replacing FDDI with a MEMORY CHANNEL network and PVM 3.3.7 with Digital PVM, we were able to speed up performance by a factor of approximately 3.4 for the same number of processors: the run time dropped from 220 seconds to 65 seconds. For comparison, we also ran the program on a four-processor SMP; the application completed in 64.5 seconds. This time was just marginally faster than the MEMORY CHANNEL configuration for the same number of processors, demonstrating that Digital PVM scales well from SMP to the MEMORY CHANNEL cluster. Finally, 2 four-processor SMP machines connected in a two-node MEMORY CHANNEL cluster ran the program in 38 seconds, demonstrating a speedup of 1.7.

Message Passing Interface

Message Passing Interface (MPI) is a message-passing standard developed by a large group of industrial and academic users. The standard contains a substantial number of functions (more than 120) and offers the same wide range of facilities that many earlier message-passing APIs provided. In fact, many parallel applications can be written using only six of the functions, but a correct implementation must provide the complete set. Argonne National Laboratory (ANL) has produced a reference implementation called MPICH.[22] This is a robust, clean implementation of the complete MPI-1 function set. In addition, it has isolated transport-specific components behind an abstract device interface (ADI).[23] The abstract device implements the communications-related functions and is further layered on what is called the channel device. The public domain version comes with channel implementations for a number of interconnects including shared memory, TCP/IP, and other proprietary interfaces. This version also includes a template for building a channel device, called the channel interface.[24] To build a channel device, the programmer must supply five functions:

1. Indicate if a control message is available on a control channel
2. Get a control message from a control channel
3. Send a control message to a control channel

Table 4
PVM Latency Comparison

PVM Implementation	Transport	Platform	Latency
PVM 3.3.9	Sockets FDDI	DEC 3000/800	400 μs
PVM 3.3.9	Shared Memory	AlphaServer 2100 4/233	60 μs
Digital PVM V1.0	MEMORY CHANNEL 1.0	AlphaServer 2100 4/233	11 μs
Digital PVM V1.0	MEMORY CHANNEL 1.5	AlphaServer 4100 5/300	8 μs
Digital PVM V1.0	Shared Memory	AlphaServer 2100 4/233	5 μs
Digital PVM V1.0	Shared Memory	AlphaServer 4100 5/300	4 μs
Digital PVM V1.0	Shared Memory	AlphaServer 8400 5/350	3 μs

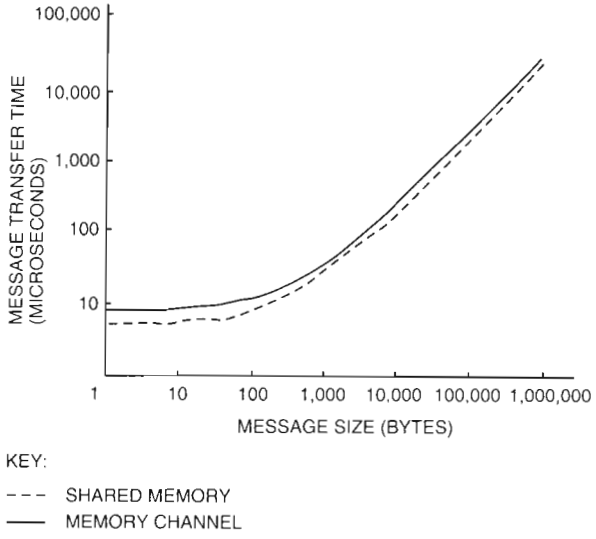


Figure 11
Digital PVM Communications Performance: Message Transfer Time

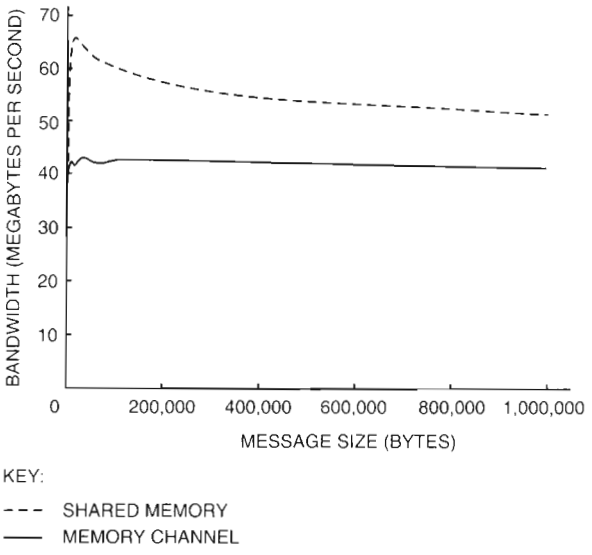


Figure 12
Digital PVM Communications Performance: Bandwidth

4. Receive data from a data channel
5. Send data to a data channel

These functions can all be implemented using the UMP functions ump_read, ump_write, and ump_wait described earlier. In addition, hooks are added to the channel initialization and shutdown code to call ump_init and ump_exit. This approach leaves the portable MPICH API library unchanged and attempts to deliver optimum performance. MPICH implements all its operations, point-to-point and collective, on the basic point-to-point services that the ADI provides.

Working with the Edinburgh Parallel Computing Centre (EPCC), we produced an early functional MPI prototype by building a channel device on UMP, as

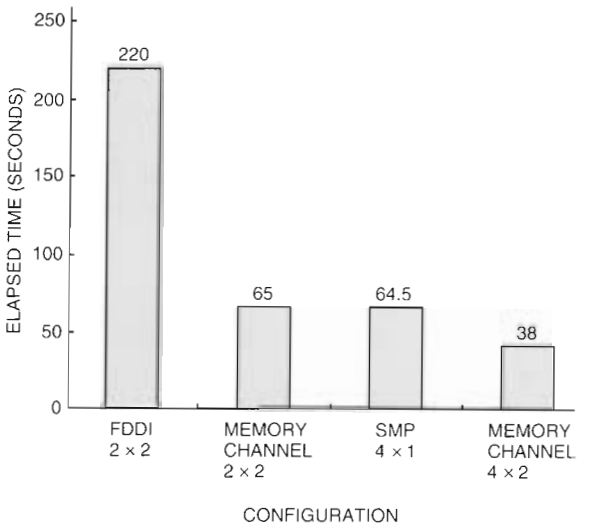


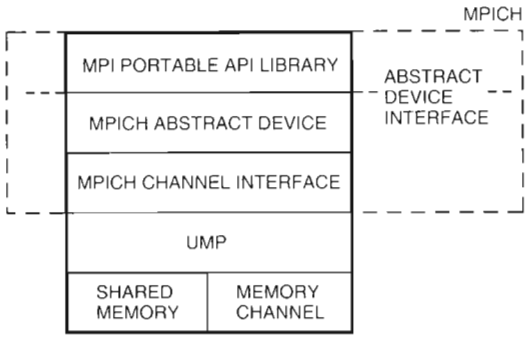
Figure 13
PVM Application Performance

shown in Figure 14a. This implementation demonstrated latencies of 12.5 μs (shared memory) and 29 μs (MEMORY CHANNEL), respectable performance for such a quick port of MPI for clusters. Furthermore, since this implementation uses UMP, it works transparently on shared memory and MEMORY CHANNEL. ADI channels typically support only one interconnect; multiple ADIs are not yet supported by MPICH. Unlike PVM, the semantics of MPI allow operation without an intermediate buffer, so that UMP buffers can be used directly.

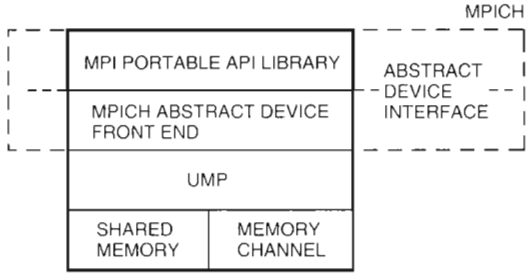
To further improve the performance of MPI on clusters, we eliminated the MPICH channel device and interfaced UMP directly to the ADI, as shown in Figure 14b. The abstract device incurs some performance penalty in its support for the channel device. In the UMP implementation, this is unnecessary as UMP already performs the function of hiding details of the transport mechanism. This implementation demonstrated latencies of 9.5 μs (shared memory) and 16 μs (MEMORY CHANNEL), using an Alpha cluster consisting of two AlphaServer 2100 4/233 machines connected by a MEMORY CHANNEL network.

Performance

Table 5 compares the communications latency achieved by MPICH and the Digital MPI implementation, using an Alpha cluster. Results are shown for both AlphaServer 2100 4/190 and AlphaServer 4100 5/300 machines connected by a MEMORY CHANNEL network. Figure 15 shows the message transfer time and Figure 16 shows the bandwidth of Digital MPI over shared memory and MEMORY CHANNEL transports for a variety of message sizes. A pair of AlphaServer 4100 5/300 machines were used for these measurements. Digital MPI reaches a peak bandwidth of about 64 MB/s using shared memory and 61 MB/s



(a) Initial Prototype



(b) Version 1.0 Implementation

Figure 14
Digital MPI Architecture

using MEMORY CHANNEL. By comparison, the unmodified MPICH achieves a peak bandwidth of 24 MB/s using shared memory and 5.5 MB/s using TCP/IP over an FDDI LAN.

Figure 17 shows the speedup demonstrated by an MPI application. The application is the Accelerated Strategic Computing Initiative (ASCI) benchmark SPPM, which solves a three-dimensional gas dynamics problem on a uniform Cartesian mesh.[25,26] The same code was run using both Digital MPI and MPICH using TCP/IP. The hardware configuration was a two-node MEMORY CHANNEL cluster of AlphaServer 8400 5/350 machines, each with six CPUs. Digital MPI used shared memory and MEMORY CHANNEL transports, whereas MPICH used the Ethernet LAN connecting the machines. The maximum speedup obtained using Digital MPI was approximately 7, whereas for MPICH the maximum speedup was approximately 1.6.

Future Work

We intend to continue refining the components described in this paper. The major change envisioned regarding the TruCluster MEMORY CHANNEL Software product is the addition of user-space spinlocks, which should significantly reduce the cost of acquiring a spinlock. We intend to increase the performance of UMP by making more efficient use of MEMORY CHANNEL in a number of ways: striping large messages over multiple adapters, supporting next-generation adapters, and using point-to-point mappings with a MEMORY CHANNEL switch. In addition, we plan to add outbufs to increase multicast message-passing performance. PVM enhancements planned include the addition of the gateway daemon to allow interoperation with other PVM implementations on external platforms. PVM will also be modified to use the UMP nonblocking write facility for arbitrarily large messages so that its performance matches that of MPI. Since the semantics of PVM force the use of an intermediate buffer, performance when using shared memory will be improved by passing pointers to a lock-controlled buffer for messages whose transfer time would exceed the overhead associated with a lock. We will continue to improve MPI performance by optimizing the UMP ADI for the MPICH implementation.

Summary

We have built a high-performance communications infrastructure for scientific applications that utilizes a new network technology to bypass the software overhead that limits the applicability of traditional networks. The performance of this system has been proven to be on a par with that of current supercomputer technology and has been achieved using commodity technology developed for Digital's commercial cluster products. The paper demonstrates the suitability of the MEMORY CHANNEL technology as a communications medium for scalable system development.

Table 5
MPI Latency Comparison

MPI Implementation	Transport	Platform	Latency
MPICH 1.0.10	Sockets FDDI	DEC 3000/800	350 μs
MPICH 1.0.10	Shared Memory	AlphaServer 2100 4/233	30 μs
Digital MPI V1.0	MEMORY CHANNEL 1.0	AlphaServer 2100 4/233	16 μs
Digital MPI V1.0	MEMORY CHANNEL 1.5	AlphaServer 4100 5/300	6.9 μs
Digital MPI V1.0	Shared Memory	AlphaServer 2100 4/233	9.5 μs
Digital MPI V1.0	Shared Memory	AlphaServer 4100 5/300	5.2 μs

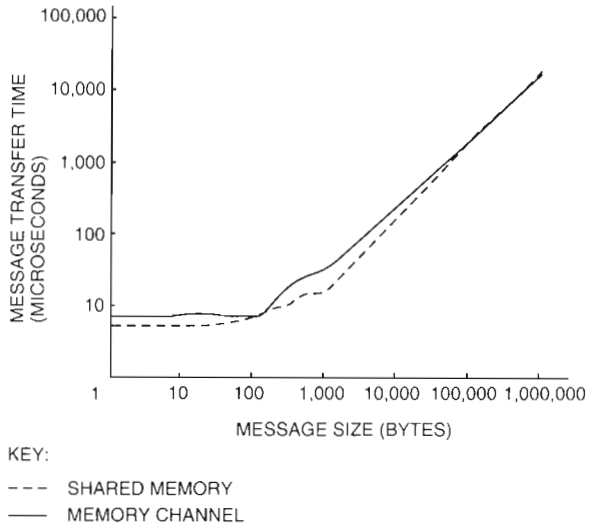


Figure 15
MPI Communications Performance: Message Transfer Time

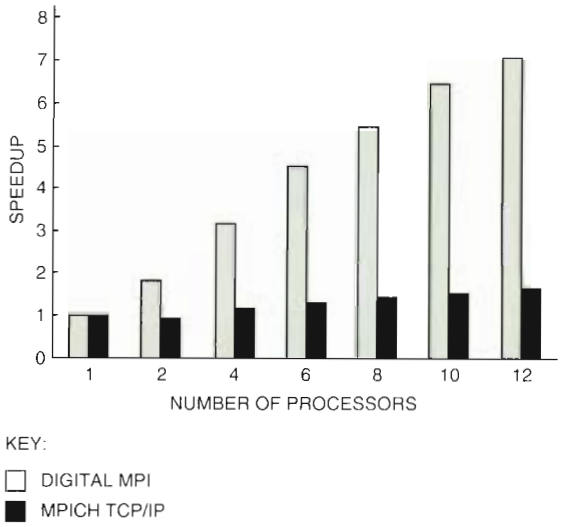


Figure 17
MPI Application Speedup

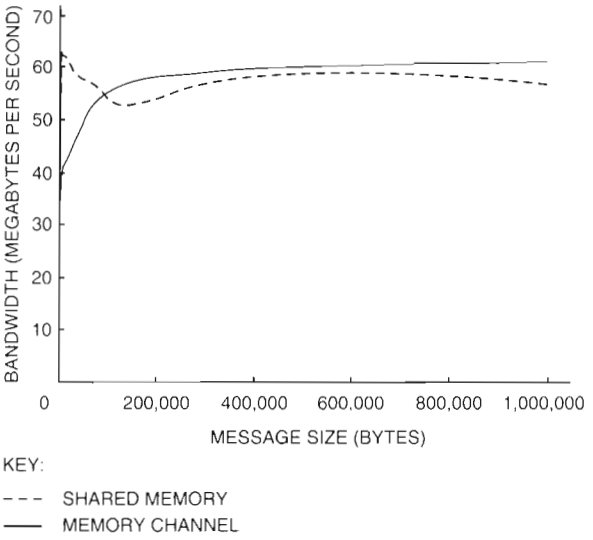


Figure 16
MPI Communications Performance: Bandwidth

Acknowledgments

The authors would like to acknowledge the following people for their contributions to this project: Gavan Duffy, whose testing made the TruCluster MEMORY CHANNEL Software a much more robust product; Liam Kelleher and Garret Taylor, who contributed some of the Digital PVM functionality; Wayne Cardoza and Brian Stevens of UNIX Engineering, who provided early access to and ongoing support of kernel MEMORY CHANNEL software; Rick Gillett and Mike Collins, who provided early MEMORY CHANNEL hardware; Richard Kaufmann, who gave us encouragement and support; and Lyndon Clarke and Kenneth Cameron at Edinburgh Parallel Computing Centre (EPCC), who modified MPICH to use UMP for Digital MPI.

References and Note

1. T. Anderson, D. Culler, and D. Patterson, "A Case for NOW (Network of Workstations)," *Proceedings of the Hot Interconnects II Symposium*, Palo Alto, Calif. (August 1994).

2. K. Keeton, T. Anderson, and D. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Proceedings of the Hot Interconnects III Symposium*, Palo Alto, Calif. (August 1995).

3. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, Order No. EY-L520E-DP, 1992).

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

5. W. Cardoza, F. Glover, and W. Snaman, Jr., "Design of the TruCluster Multicomputer System for the Digital UNIX Environment," *Digital Technical Journal*, vol. 8, no. 1 (1996): 5–17.

6. R. Gillett, "MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect," *IEEE Micro* (February 1996):12–18.

7. M. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proceedings of the Twenty-first Annual International Symposium on Computer Architecture* (April 1994): 142–153.

8. M. Blumrich et al., "Two Virtual Memory Mapped Network Interface Designs," *Proceedings of the Hot Interconnects II Symposium*, Palo Alto, Calif. (August 1994): 134–142.

9. L. Iftode et al., "Improving Release-Consistent Shared Virtual Memory using Automatic Update," *Proceedings of the Second International Symposium on High-Performance Computer Architecture* (February 1996).

10. C. Dubnicki et al., "Software Support for Virtual Memory-Mapped Communication," *Proceedings of the Tenth International Parallel Processing Symposium* (April 1996).

11. High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 1.0, *Scientific Programming*, vol. 2, no. 1 (1993).

12. A. Geist et al., *PVM 3 User's Guide and Reference Manual*, ORNL/TM-12187 (Oak Ridge, Tenn.: Oak Ridge National Laboratory, May 1994). Also available on-line at http://www.netlib.org/pvm3/ug.ps.

13. A. Geist et al., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing* (Cambridge, Mass.: The MIT Press, 1994). Also available on-line at http://www.netlib.org/pvm3/book/pvm-book.html.

14. MPI Forum, "MPI: A Message Passing Interface Standard," *International Journal of Supercomputer Applications*, vol. 8, no. 3/4 (1994). Version 1.1 of this document is available on-line at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

15. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface* (Cambridge, Mass.: The MIT Press, 1994).

16. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal*, vol. 7, no. 3 (1995): 5–23.

17. E. Benson et al., "Design of Digital's Parallel Software Environment," *Digital Technical Journal*, vol. 7, no. 3, (1995): 24–38.

18. In the first implementations, the PCI MEMORY CHANNEL network adapter places a limit of 128 MB on the amount of MEMORY CHANNEL space that can be allocated.

19. *TruCluster MEMORY CHANNEL Software Programmer's Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QTN4A-TE, 1996).

20. J. Brosnan, J. Lawton, and T. Reddin, "A High-Performance PVM for Alpha Clusters," *Proceedings of the Second European PVM Users' Group Meeting*, Lyons, France (September 1995).

21. M. Hausner, M. Burrows, and C. Thekkath, "Efficient Implementation of PVM on the AN2 ATM Network," *Proceedings of High-Performance Computing and Networking* (May 1995).

22. W. Gropp and N. Doss, "MPICH Model MPI Implementation Reference Manual," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, June 1995).

23. W. Gropp and E. Lusk, "MPICH ADI Implementation Reference Manual," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, October 1994).

24. W. Gropp and E. Lusk, "MPICH Working Note: Creating a New MPICH Device using the Channel Interface," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, June 1995).

25. Accelerated Strategic Computing Initiative (ASCI), RFP Statement of Work C6939RFP6-3X, Los Alamos National Laboratory (LANL) (February 12, 1996). This document is also available on-line at http://www.llnl.gov/asci_rfp/asci-sow.html.

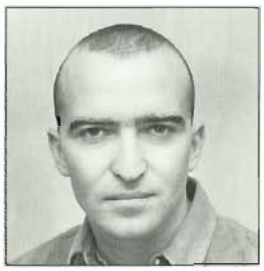
26. The ASCI SPPM Benchmark Code is available from Lawrence Livermore National Laboratory at http://www.llnl.gov/asci_benchmarks/asci/limited /ppm/asci_sppm.html.

Biographies

James V. Lawton

Jim Lawton joined Digital in 1986 and is a principal engineer in the Technical Computing Group. In his current position, he contributed to the design of Digital PVM and the UMP library and was responsible for implementing UMP and adding support for collective operations to Digital PVM. Before that, he worked on the characterization and optimization of customer scientific/technical benchmark codes and on various hardware and software design projects. Prior to coming to Digital, Jim contributed to the design of analog and digital motion control systems and sensors at the Inland Motor Division of Kollmorgen Corporation. Jim received a B.E. in electrical engineering (1982) and an M.Eng.Sc. (1985) from University College Cork, Ireland, where he wrote his thesis on the design of an electronic control system for variable reluctance motors. In addition to receiving the Hewlett-Packard (Ireland) Award for Innovation (1982), Jim holds one patent and has published several papers. He is a member of IEEE and ACM.

John J. Brosnan
John Brosnan is currently a principal engineer in the Technical Computing Group where he is project leader for Digital PVM. In prior positions at Digital, he was project leader for the High Performance Fortran test suite and a significant contributor to a variety of publishing technology products. John joined Digital after receiving his B.Eng. in electronic engineering in 1986 from the University of Limerick, Ireland. He received his M.Eng. in computer systems in 1994, also from the University of Limerick.



Morgan P. Doyle
In 1994, Morgan Doyle came to Digital to work on the High Performance Fortran test suite. Presently, he is an engineer in the Technical Computing Group. Early on, he contributed significantly to the design and development of the TruCluster MEMORY CHANNEL Software, and he is now responsible for its development. Morgan received his B.A.I. and B.A. in electronic engineering (1991) and his M.Sc. (1993) from Trinity College Dublin, Ireland.

Seosamh D. Ó Riordáin
Seosamh Ó Riordáin is an engineer in the Technical Computing Group where he is currently working on Digital MPI and on enhancements to the UMP library. Previously, he contributed to the design and implementation of the TruCluster MEMORY CHANNEL Software. Seosamh joined Digital after receiving his B.Sc. (1991) and M.Sc. (1993) in computer science from the University of Limerick, Ireland.

Timothy G. Reddin
A principal engineer in the Technical Computing Group, Timothy Reddin currently leads the team responsible for the TruCluster MEMORY CHANNEL Software, the UMP library, Digital PVM, and Digital MPI. Prior to coming to Digital in 1994, Tim worked for eight years as a systems designer at ICL High Performance Systems in the United Kingdom. He was responsible for the I/O architecture of the ICL Goldrush parallel database server, for which he holds two patents, and the design of an I/O and communications controller. Tim also worked at Raytheon on the data communications subsystem for the NEXRAD distributed real-time Doppler weather radar subsystem. Prior to that, he developed the software architecture for an integrated executive workstation while working at CPT Limited. After receiving his B.Sc. (with distinction, 1976) in computer science and mathematics from University College Dublin, Ireland, Tim joined the staff of University College Cork, where he was a systems programmer. Tim is a member of the British Computer Society and is a Chartered Engineer.

Bernard A. Rozmovits

The Design of User Interfaces for Digital Speech Recognition Software

Digital Speech Recognition Software (DSRS) adds a new mode of interaction between people and computers—speech. DSRS is a command and control application integrated with the UNIX desktop environment. It accepts user commands spoken into a microphone and converts them into keystrokes. The project goal for DSRS was to provide an easy-to-learn and easy-to-use computer–user interface that would be a powerful productivity tool. Making DSRS simple and natural to use was a challenging engineering problem in user interface design. Also challenging was the development of the part of the interface that communicates with the desktop and applications. DSRS designers had to solve timing-induced problems associated with entering keystrokes into applications at a rate much higher than that at which people type. The DSRS project clarifies the need to continue the development of improved speech integration with applications as speech recognition and text-to-speech technologies become a standard part of the modern desktop computer.

In the 1960s and early 1970s, people controlled computers using toggle switches, punched cards, and punched paper tape. In the 1970s, the common control mechanism was the keyboard on teletypes and on video terminals. In the 1980s, with the advent of graphical user interfaces, people found that a new mode of interaction with the computer was useful. The concept of a pointer—the mouse—evolved. Its popularity grew such that the mouse is now a standard component of every modern computer. In the 1990s, the time is right to add yet another mode of interaction with the computer. As compute power grows each year, the boundary of the man–machine interface can move from interaction that is native to the computer toward communication that is natural to humans, that is, speech recognition.

DSRS Product Overview

Very simply, DSRS is an application that provides speech macros. The user speaks a command, phrase, or sentence (that is, an utterance), and DSRS performs some actions. The action might be to launch an application, for example, in response to the command "bring up calendar"; or to type something, for example, in response to "edit to-do list," to invoke emacs \files\projectA\todo.txt. DSRS not only houses the speech macro capability but also provides a user interface, a speech recognition engine, and interfaces to the X Window System.

Following is a high-level description of how the software functions. Commands are spoken into a microphone, and the audio is captured and digitized. The first step in the processing is the speech analysis system, which provides a spectral representation of the characteristics of the time-varying speech signal. Next is the feature-detection stage. Here, the spectral measurements are converted to a set of features that describe the broad acoustic properties of the different phonetic units.[1] These representations of the speech signal are then segmented and identified as phonetic sequences. The speech recognition engine accepts these phonetic sequences and returns word matches and confidence values for each match. These data are used to determine if each match is acceptable. If a

match is acceptable, DSRS retrieves keystrokes associated with each utterance, and the keystrokes are then sent into the system's keyboard buffer or to the appropriate application. For instances of continuous speech recognition, a sentence is recognized and keystrokes are concatenated to represent the utterance. For example, for the utterance "five two times seven three four equals," the keys "52 * 734 =" would be delivered to the calculator application.

Although this concept seems simple, its implementation raised significant system integration issues and directly affected the user interface design, which was critical to the product's success. This paper specifically addresses the user interface and integration issues and concludes with a discussion of future directions for speech recognition products.

Project Objective

The objective of the DSRS project was to provide a useful but limited tool to users of Digital's Alpha workstations running the UNIX operating system. DSRS would be designed as a low-cost, speech recognition application and would be provided at no cost to workstation users for a finite period of time.

When the project began in 1994, a number of command and control speech recognition products for PCs already existed. These programs were aimed at end users and performed useful tasks "out of the box," that is, immediately upon start-up. They all came with built-in vocabulary for common applications and gave users the ability to add their own vocabulary.

On UNIX systems, however, speech recognition products existed only in the form of programmable recognizers, such as BBN Hark software. Our objective was to build a speech recognition product for the UNIX workstation that had the characteristics of the PC recognizers, that is, one that would be functional immediately upon start-up and would allow the nonprogrammer end user to customize the product's vocabulary.

We studied several speech recognition products, including Talk→To Next from Dragon Systems, Inc., VoiceAssist from Creative Labs, Voice Pilot from Microsoft, and Listen from Verbex. We decided to provide users with the following features as the most desirable in a command and control speech recognition product:

- Intuitive, easy-to-use interface
- Speaker-independent models that would eliminate the need for extensive training
- Speaker-adaptive capability to improve accuracy of words
- Continuous speech recognition capability
- Prompts for active vocabulary
- Minimum use of screen area
- User control over the user interface configuration
- Simple mechanism to modify and create new vocabulary
- Integration with the X Window System
- Support for out-of-the-box desktop applications provided with the UNIX operating system
- Support for vi and emacs editors, and for C programming

The DSRS Architecture

DSRS comprises several major components which are outlined below and illustrated in Figure 1. Of these components, three are licensed from Dragon Systems, Inc.: the front-end processor, the recognizer engine, and the speaker-independent speech models.

Dragon Systems, Inc. was chosen as the provider of the speech recognition engine based on the accuracy of their technology, their products and expertise in other local languages, and their long-term commitment to speech recognition.

Data acquisition consists of the microphone, audio card, and the multimedia services application programming interface (API) that provides support for the sound card.

The *front-end processor* analyzes a stream of digitized data and differentiates between silence, noise, and speech; it then extracts a set of computed features from the speech signals.

The *recognizer*, or speech recognition engine, accepts the computed representation of the speech in the form of feature packets which drive the Hidden Markov Models to recognize utterances. Hidden Markov Models are basically state machines that transition from a beginning state to a number of internal states and then to a final state based on input data and probabilities.[2] Each transition carries two sets of probabilities: a transition probability, which provides the probability of this transition being taken, and an output probability density function (PDF), which is the conditional probability of emitting each output symbol from a finite alphabet given that a transition is taken.[3] The PDFs are adapted when the model is "trained," that is, customized, by the individual user.

The *finite state grammar* is a state machine that contains a representation of the vocabulary supported by DSRS. Each state contains words, phrases, or sentences; their associated actions; and the information needed to transition to the next state. The current state is used to control the Active words.

The *speech models* are a set of utterance models used by the recognizer. DSRS provides vocabulary and speaker-independent models for the applications supported by DSRS. Users who wish to include their own

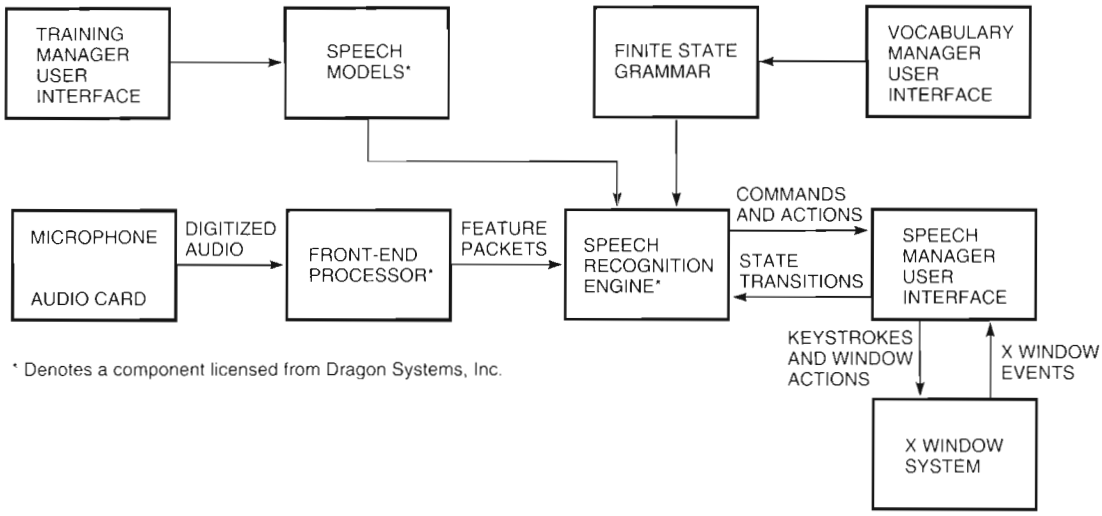


Figure 1
DSRS Architectural Block Diagram

words can create models using the Vocabulary Manager user interface.

The *Speech Manager* is the main user-interface component. The Speech Manager window provides visual feedback to users. It also keeps track of the current window in focus and acts as the agent to control focus in response to users' speech commands.

The *Vocabulary Manager* user-interface window displays the current hierarchy of the finite state grammar file. The Vocabulary Manager allows the user to customize using the functions for addition, deletion, and modification of words or macros. Also in this window, the command-utterance to keystroke translations are displayed, created, or modified.

In the *Training Manager* user interface, the user may train newly created words or phrases in the user vocabulary files and retrain, or adapt, the product-supplied, independent vocabulary.

The DSRS Implementation

As the design team gained experience with the DSRS prototypes, we refined user procedures and interfaces. This section describes the key functions the team developed to ensure the user-friendliness of the product, including the first-time setup, the Speech Manager, the Training Manager, the Vocabulary Manager, and the finite state grammar.

First-time Setup

DSRS requires a setup process when used for the first time. The user must create user-specific files and settings. The user begins by selecting the microphone and by testing and adjusting the microphone input volume to usable settings. The user is then prompted to speak a few words, which are presented on the screen. DSRS uses the speech data to choose the speaker-independent model that most closely matches the speaker's voice. There are models for lower- and higher-pitched voices. The software copies the selected model to the user's home directory; the model is then modified when the user makes changes to the provided models and vocabulary. After setup is complete, the next step is the Training Manager which presents the user with a list of 20 words to train; when this step is completed, DSRS is ready for use. The Training Manager is described in more detail later in this section.

The procedure above was developed to take a new user through the entire setup process without the need to refer to any documentation. Once the user files are created, DSRS bypasses these steps and comes up ready to work. A notable change that we made to the setup was instigated by our own use of the software. We found that inconsistent microphone volume settings were a frequent problem. When systems were rebooted, volume settings were reset to default values. Consequently, we created an initialization file that records the volume settings as well as all user-definable characteristics of the graphical user interface.

Speech Manager

Once DSRS is ready and in its idle state, it presents the user with the Speech Manager window, an example of which is shown in Figure 2. The Speech Manager provides the following critical controls:

- Microphone on/off switch.
- A VU (volume units) meter that gives real-time feedback to the audio signal being heard. A VU meter is a visual feedback device commonly used on devices such as tape decks. Users are generally very comfortable using them.

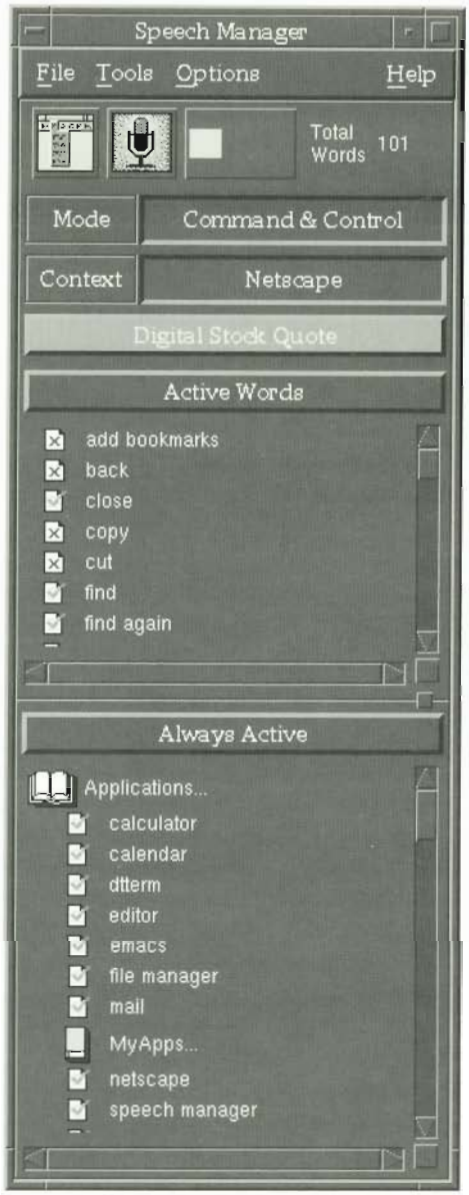


Figure 2
DSRS Speech Manager Window

- Two user-controllable panes that display the Always Active and Active vocabulary sets. The Always Active vocabulary words are recognized regardless of the current application in focus. The Active vocabulary words are specific to the application in focus and change dynamically as the current application changes. The vocabularies are designed in this way so that a user can speak commands both within an application context and in order to switch contexts.
- Three small frames that provide status information to the user.
  - The Mode frame indicates the current state of the Speech Manager: command and control or sleeping.
  - The Context frame displays the class name of the application currently in focus. This context also determines the current state of the Active word list.
  - The history frame displays the word, phrase, or sentence last heard by the recognizer. The history frame is set up as a button. When pressed, it drops down to reveal the last 20 recognized utterances.
- A menu that provides access to the management of user files, the Vocabulary Manager, the Training Manager, and various user-configurable options.

Training Manager

The Training Manager adapts the speaker-independent speech models to the user's speech patterns and creates new models for added words. Our study of PC-based speech recognizers led us to the conclusion that the design of a training interface is critical to obtain good results. For example, the training component of one PC-based recognizer we examined did not provide clear feedback to the user when an utterance had been processed, thus causing the user confusion about when to speak. This confusion lead to training errors and frustration. Another recognizer did not allow the user to pause while training, a major inconvenience for the user who, for example, needed to clear his throat or speak to someone.

We developed the following list of design characteristics for a good training user interface.

- Strong, clear indications that utterances are processed. We added a series of boxes that are checked off as each utterance is processed and a VU meter that shows the system is picking up audio signals.
- Reduced amount of eye movement needed for the training to proceed smoothly and quickly. We placed visual feedback objects in positions that allow users to focus their eyes on a limited area of the screen and not have to look back and forth across the screen at each utterance.
- A glimpse of upcoming words. A list of words is displayed on the user interface and moves as words are processed.
- A progress indicator. Text is displayed and updated as each word is processed, indicating progress, for example, Word 4 of 21.
- Option to pause, resume, and restart training.
- Large, bold font display of the word to be spoken and a small prompt, "Please continue," displayed when the system is waiting for input.
- Automatic addition of repeated utterances that are "bad" or do not match the expected word.
- Control over the number of repetitions.

As the example in Figure 3 shows, the Training Manager presents a word from a list of words to be trained. The word to be spoken is presented in a large,

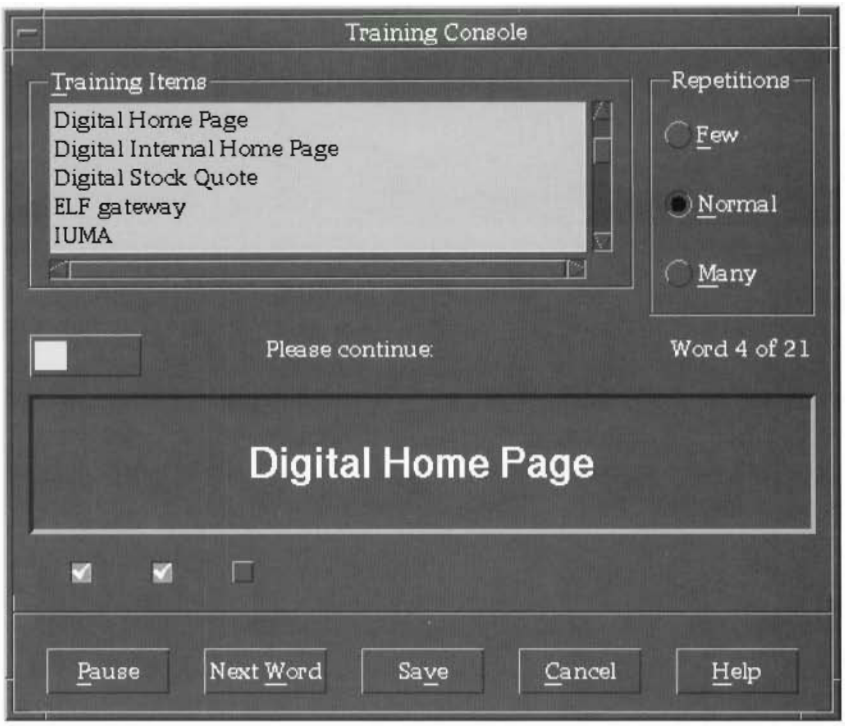


Figure 3
Training Manager Window

bold font to differentiate it from the other elements in the window. To train the words, the user repeats an utterance from one to six times. The user must speak at the proper times to make training a smooth and efficient process. DSRS manages the process by prompting the speaker with visual cues. Right below the word is a set of boxes that represent the repetitions. The boxes are checked off as utterances are processed, providing positive visual feedback to the speaker. When one word is complete, the next word to be trained is displayed and the process is repeated. When all the words in the list are trained, the user saves the files, and DSRS returns to the Speech Manager and its active mode with the microphone turned off.

Vocabulary Manager

The Vocabulary Manager, an example of which is shown in Figure 4, enables users to modify speech macros by changing the keystrokes stored for each command and by adding new commands to existing applications. Users can also add speech support for entirely new applications. The vocabularies are represented graphically as hierarchies of application vocabularies, groups of words, and individual words. The Vocabulary Manager provides an interface that allows manipulation of this database of words without resorting to text editors. The Always Active vocabularies are accessible here and are manipulated in the same manner as the application-specific vocabularies. With the Vocabulary Manager, the user may import and export vocabularies or parts of vocabularies in order to share commands and thus enable speech recognition in applications not supported by default in DSRS.

Finite State Grammar

The finite state grammar (FSG) is a state machine with all the vocabulary required to transition between states and conditions. The FSG has two distinct sets of vocabulary, which have already been mentioned: the Always Active, or global vocabulary, and the Active, or context-specific, vocabulary.

In creating the FSG, we found that we needed special functions for interaction with the windowing system and representations for all keyboard keys. While creating these special functions, we designed the interaction for maximum convenience. For example, when a user speaks the phrase "go to calculator" or "switch to calculator" or simply "calculator," the meaning is readily interpreted by the software. For the user's convenience, these phrases trigger the following conditional actions.

- If a window of class "calculator" is present on the system, then set focus to it. This is done regardless of its state; the window may be in an icon state, hidden, or on another work space such as may be found in the Common Desktop Environment (CDE).
- If the window does not exist, then create one by launching the application.

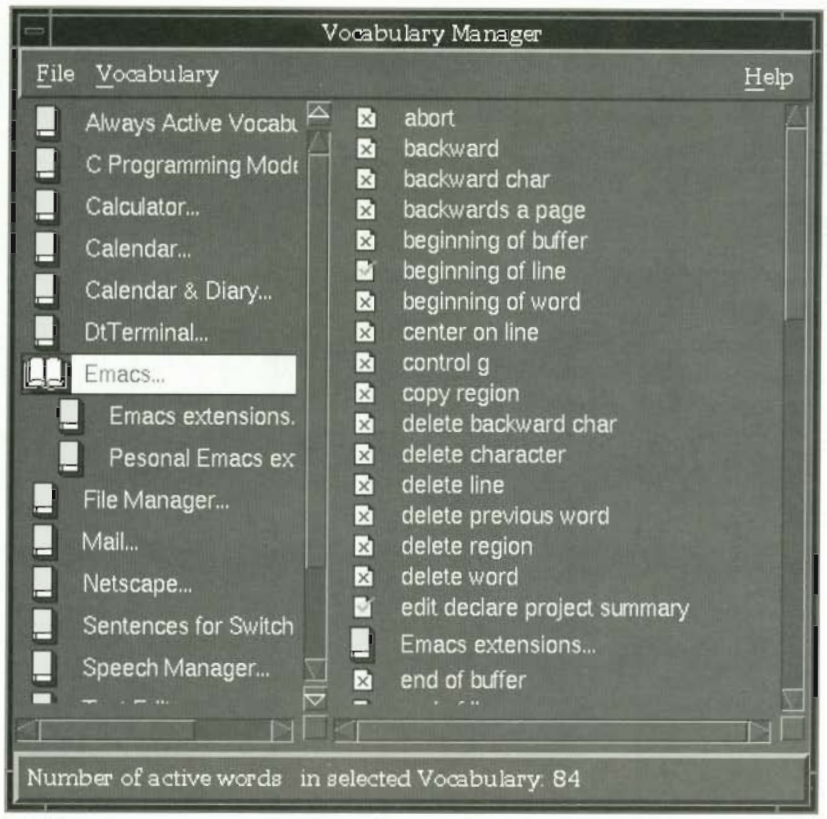


Figure 4
Vocabulary Manager Window

The simple logic of this special function enhances user productivity. Often workstation and PC screens are littered with windows or applications icons and icon boxes through which the user must search. Speech control eliminates the steps between the user thinking "I want the calculator" and the application being presented in focus, ready to be used. The DSRS team created a function called FocusOrLaunch, which implements the behavior described above. The function is encoded into the FSG continuous-switching-mode sentences in the Always Active vocabulary associated with the spoken commands "switch to <application name>," "go to <application name>," and just plain "<application name>."

Applications like calculator and calendar are not likely to be needed in multiple instances. However, applications such as terminal emulator windows are. DSRS defines the specific phrase "bring up <application name>" to explicitly launch a new instance of the application; that is, the phrase "bring up <application name>" is tied to a function named Launch.

The phrases "next <application name>" and "previous <application name>" were chosen for navigating between instances of the same application. DSRS remembers the previous state of the application. For instance, if the calendar application is minimized when the user says "switch to calendar," the calendar window is restored. When the user says "switch to emacs," the calendar is returned to its former state. In this case, it is minimized.

DSRS also adds speech control to the common window controls such as minimize, maximize, and close. These functions operate on whatever window is currently in focus.

Another convenient command is "Speech Manager go to sleep." When the user speaks this command, DSRS transitions into a special standby state. In this state, termed "sleeping," the recognizer is still listening but will return to command and control mode only when the command "Speech Manager wake up" is spoken. The "go to sleep" command puts DSRS into a standby state, allowing normal conversation to take place without words being recognized as commands and causing unwanted events to occur.

Version 1.1 of DSRS adds even more functions, such as the "microphone off" command, which goes a step beyond "go to sleep." With "microphone off," the input audio section is completely released and DSRS will no longer listen until the microphone is manually turned back on. This function allows the

user to launch an audio-based application that will record, such as a teleconferencing session. Version 1.1 also includes a function that allows the user to play a "wave," or digitized audio clip. Audio cues may thus be played as part of speech macros. The "say" command invokes DECtalk Text-to-Speech functionality so that audio events can be spoken.[4]

Since speech recognition is a statistical process and prone to errors, the design team deemed "confirm" an important function to protect user data and prevent unwanted actions. The "confirm" function protects certain sensitive actions, such as exiting an editor, with a confirmation dialog box. Simply adding the "confirm" syntax within a speech macro causes the dialog box "are you sure?" to appear. The vocabulary is switched to respond to only yes and no so that a higher reliability can be achieved. If the user says no or presses the no button, the computer returns to its previous state. If the user says yes, the action following the "confirm" function is executed.

Another concept encoded in the FSG for user convenience is menu flattening. Menu displays are hierarchical because the number of menu entries that can be shown on the screen at one time is limited. A good example is the File menu. When the user clicks the mouse button on File, a drop-down menu appears containing actions such as Open file, Save file, Save file as ..., Print, and Exit. However, hierarchical menus do not really represent the way people normally think about actions; for example, when the user thinks "exit," he or she must then take the steps file and exit. With speech recognition, the computer can take the interim steps. The FSG in DSRS was built to handle two cases: (1) The user says "file" and "exit," and (2) the user says only "exit" and DSRS performs the file and exit sequence transparently. This second mode connects the actions more closely with the user's thought processes and does not force a sequence of actions in order for tasks to be performed. The menu-flattening feature of DSRS was encoded into the FSG file. While the example given may seem trivial, the concept is an important one and can be used to flatten many levels of menus. For instance, users take several steps to change the font or type size on a region of highlighted text in a word processing program. The following could conceivably be invoked as a speech macro: "Change to Helvetica Bold Italic 24 points."

Integrating Speech Recognition in Applications

As described in the section Overview, DSRS feeds keystrokes to applications. Therefore, the preferred application development method for allowing access to functions—one that will allow integration of speech recognition—is accelerator keys. Typically, accelerator keys are in the form of CTRL + <key> bindings that allow direct access to a function, regardless of menu hierarchies. It should be noted that this lack of hierarchy limits the number of directly accessible functions.

A second method for integrating speech within an application is through menu mnemonics. Mnemonics are the keyboard equivalents signified in application menus by an underlined letter. The first mnemonic is invoked by a combination of the ALT key and the underlined letter, which can be followed by another underlined letter. For example, pressing ALT + f invokes the file menu item; pressing x immediately thereafter invokes the "exit" entry for the application.

Integrating speech recognition becomes difficult when application functions are not accessible through the keyboard. Applications designed to allow access to functions only by means of the mouse cannot be speech enabled as DSRS is currently implemented. Although DSRS can send mouse clicks into the system, consistently locating the mouse pointer on applications is difficult. The next sections further illustrate the issues that stemmed from these integration issues as we implemented and tested DSRS.

Client–Server Protocols

Applications such as emacs and Netscape Navigator have protocols that allow other processes to send commands to them. For example, a file name or a universal resource locator (URL) may be sent via the command line. DSRS exploits this facility in Netscape Navigator to allow Web surfing by voice. For example, in the Netscape context, the speech macro "Digital home page" would translate to the following command issued to a window: netscape-remote openURL("http://www.digital.com"). Although this command string seems a bit awkward, the result is that the actions being taken are all transparent to the user and they work very well.

Problems Encountered in Implementation

Unlike the applications discussed in this paper, some applications are not developed with good programming practices. Neither are the keyboard interfaces well-tested. We encountered the following types of problems when using the keyboard as the main input mechanism.

- Applications had multiple menu mnemonics mapped to the same key sequence. This approach could not work even if the keyboard were used directly.
- Application functions controlled by graphic buttons were accessible only by mouse.
- Keyboard mapping was incomplete, that is, mnemonics were only partially implemented.

In the implementation of DSRS, we encountered one unexpected problem. When a nested menu mnemonic was invoked, the second character was lost. The sequence of events was as follows:

- A spoken word was recognized, and keystrokes were sent to the keyboard buffer.
- The first character, ALT + <key>, acted normally and caused a pop-up menu to display.
- The menu remained on display, and the last key was lost.

We determined that the second keystroke was being delivered to the application before the pop-up menu was displayed. Therefore, at the time the key was pressed, it did not yet have meaning to the application. It is apparent that such applications are written for a human reaction–based paradigm. DSRS, on the other hand, is typing on behalf of the user at computer speeds and is not waiting for the pop-up menu to display before entering the next key.

To overcome this problem, we developed a synchronizing function. Normally the Vocabulary Manager notation to send an ALT + f followed by an x would be ALT + f x. This new synchronizing function was designated as sALT + f x. The synchronizing function sends the ALT + f and then monitors events for a map-notify message indicating that the pop-up menu has been written to the screen. The character following ALT + f is then sent, in this case, the x. The synchronizing function also has a watchdog timer to prevent a hang in the event a map-notify message. This method is included in the final product.

Guidelines for Writing Speech-friendly Applications

Several guidelines for enabling speech recognition in applications became apparent as we gained experience using DSRS. Coincidentally, a guideline recently published by Microsoft Corporation documents some of the very same points.[5]

- Provide keyboard access to all features.
- Provide access keys for all menu items and controls.
- Fully document the keyboard user interface.
- Whenever possible, use accelerator keys; they are more reliable than using menu mnemonics. Mnemonics can be overloaded or non-functional if the menu is not active.
- Client–server protocols can work well for enabling speech recognition; document fully.
- Do not depend on human reaction times for displayed windows or on slow typing rates.
- Provide user-friendly titles for all windows, even if the title is not visible.
- Avoid triggering actions or messages by mouse pointer location.
- Give dialog boxes consistent keyboard access; for instance, boxes should close when the ESC key is pressed. The dialog box responses yes and no should correspond to the y and n keys.

Application developers who wish to design a speech interface directly into their applications now have this option. Several speech APIs are available. Microsoft offers the Speech Software Development Kit, and the Speech Recognition API Committee, chaired by Novell, offers SRAPI. Computer–human speech interaction is the subject of ongoing research. Much of the government-sponsored research is now being commercialized. Several groups, such as ACM CHI,[6] have been and continue to study speech-only interfaces. They are discovering that "translating a graphical interface into speech is not likely to produce an effective interface. The design of the Speech User Interface must be a separate effort that involves studying the human–human conversations in the application domain."[6]

Future Directions for Speech Recognition

In addition to uncovering points for developers to build speech-enabled applications, we also gained a perspective on how speech recognition may develop in the future. A brief overview of these insights is presented in this section.

Integrating speech and audio output—The addition of a two-way interface of speech and audio that gives users feedback will move the user interface to a new level.

Telephone access—Telephone access can make workstations more valuable communications devices by connecting users to information such as e-mail messages and appointment calendars. The telephone can extend the reach of our desktop computers.[6]

Dictation—Discrete dictation products with capabilities of 60,000 words are commercially available now; in the not-too-distant future, continuous-recognition dictation products will become viable. A command and control recognizer that can be seamlessly switched to dictation mode is a very powerful tool.

Speech recognition integrated with natural language processing—The field of natural language processing deals with the extraction of semantic information contained in a sentence. Machine understanding of natural language is an obvious next step. Users will be able to speak in a less restricted fashion and still have their desired actions carried out.

A new paradigm for applications—A new class of applications needs to be created, one that is modeled more on human thought processes and natural language expression than on the functional partitioning

in today's applications. A user agent or secretary program that could process common requests delivered entirely by speech is not out of reach even with the technology available today, for example:

User: What time is it?
Computer: It is now 1:30 p.m.

User: Do I have any meetings today?
Computer: Staff meeting is ten o'clock to twelve o'clock in the corner conference room.

Computer: Mike Jones is calling on the phone. Would you like to answer or transfer the call to voice mail?
User: Answer it.

User: Do I have any new mail?
Computer: Yes, two messages. One is from Paul Jones, the other from your boss.
User: Read message two.

User: What is the price of Digital stock?
Computer: Digital stock is at $72 1/2, up 1/4.

The example above shows the user agent providing information and interacting with e-mail, telephone, stock quote, and calendar programs. As we move into the future, the computer–user interface should move closer to the interaction model humans use to communicate with each other. Speech recognition and text-to-speech software help in a significant way to move in this direction.[6]

Performance

DSRS word recognition, which is the primary performance measure, is as good as comparable command and control recognizers found on PCs. Training troublesome and acoustically similar words improves the performance. The vocabulary, because of the targets chosen, that is, UNIX commands, does have acoustic collisions, for example, escape and Netscape. Further, we had to use the vocabularies supporting the UNIX shell commands, and commands such as vi can be pronounced in different ways, for example, vee-eye or vie. The shell commands are also full of very short utterances that tend to result in higher error rates.

On the slower, first-generation Alpha workstations, DSRS has noticeable delays, on the order of a few hundred milliseconds. However, on the newer and faster Alpha workstations, DSRS responds within human perceptual limits, less than 100 milliseconds.

Another interesting phenomenon associated with the speed of the workstation is the improvement DSRS makes in user productivity. On a slow machine, the speech interface has little impact if the application is slow in performing its tasks. In other words, the time it takes to perform a certain task is not greatly affected unless the human input of commands is a significant portion of that time. However on a fast machine, the application performs tasks as quickly as the commands are spoken, and the productivity enhancement, therefore, is great.

Summary and Conclusions

The DSRS team accomplished its objective of developing a low-cost speech recognition product. DSRS for Digital UNIX is being shipped with all Alpha workstations at no additional cost. Integration with the X Window System was successful.

With reference to the focus of this paper—developing the user-friendly interface—we found through feedback from our user base that most first-time users perform useful work using DSRS without consulting the documentation. The first-time setup design that provides instructions and feedback to users was successful. The list of Active and Always Active words and phrases is a helpful reference for new users until they learn the "language" they can use to communicate with their applications.

Adding vocabulary for new applications is a bit more challenging because some "reverse engineering" may be required on a particular application. One needs to know the class name of each of the windows and then map the keystrokes for each of the functions to speech macros. Although this procedure is documented in the manual, it can be challenging for users.

Prototypes of DSRS control for sophisticated menu-driven applications, such as mechanical computer-aided design, show excellent promise for enhancing user productivity. For example, with computer-aided design or drafting software, users can focus their eyes on the drawing target on the screen while they are speaking menu functions.

Speech recognition is an evolutionary step in the overall computer–user interface. It is not a replacement for the keyboard and mouse and should be used to complement these devices. Speech recognition works as an interface because it allows a more direct connection between the human thought processes and the applications.

Speech recognition coupled with natural language processing, text-to-speech, and a new generation of applications will make computers more accessible to people by making them easier to use and understand.

Acknowledgments

Thanks go to the dedicated team of engineers who developed this product: Krishna Mangipudi, Darrell Stam, Alex Doohovskoy, Bill Hallahan, and Bill Scarborough, and to Dragon Systems, Inc. for being a cooperative business and engineering partner.

References

1. L. Rabiner and B. Juang, *Fundamentals of Speech Recognition* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1993): 45–46.
2. C. Schmandt, *Voice Communication with Computers: Conversational Systems* (New York, N.Y.: Van Nostrand Reinhold, 1994): 144–145.
3. K.F. Lee, *Large-Vocabulary Speaker-Independent Continuous Speech Recognition: The SPHINX System* (Pittsburgh, Pa.: Carnegie-Mellon University Computer Science Department, April 1988).
4. W. Hallahan, "DECtalk Software: Text-to-Speech Technology and Implementation," *Digital Technical Journal,* vol. 7, no. 4 (1995): 5–19.
5. G. Lowney, *The Microsoft Windows Guidelines for Accessible Software Design* (Redmond, Wash.: Microsoft Development Library, 1995): 3–4.
6. N. Yankelovich, G. Levow, and M. Marx, "Designing SpeechActs: Issues in Speech User Interfaces," *Proceedings of ACM Conference on Computer–Human Interaction (CHI) '95: Human Factors in Computing Systems: Mosaic of Creativity,* Denver, Colo. (May 1995): 369–376.

Biography

Bernard A. Rozmovits

During his tenure at Digital, Bernie Rozmovits has worked on both sides of computer engineering—hardware and software. Currently he manages Speech Services in Digital's Light and Sound Software Group, which developed the user interfaces for Digital's Speech Recognition Software and also developed the DECtalk software product. Prior to joining this software effort, he focused on hardware engineering in the Computer Special Systems Group and was the architect for voice-processing platforms in the Image, Voice and Video Group. Bernie received a Diplôme D'Étude Collègiale (DEC) from Dawson College, Montreal, Quebec, Canada, in 1974. He holds a patent entitled "Data Format For Packets Of Information," U.S. Patent No. 5,317,719.

Further Readings

The *Digital Technical Journal* is a refereed, quarterly publication of papers that explore the foundations of Digital's products and technologies. *Journal* content is selected by the Journal Advisory Board, and papers are written by Digital's engineers and engineering partners. Engineers who would like to contribute a paper to the *Journal* should contact the Managing Editor, Jane Blake, at Jane.Blake@ljo.dec.com.

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

Digital UNIX Clusters/Object Modification Tools/ eXcursion for Windows Operating Systems/ Network Directory Services
Vol. 8, No. 1, 1996, EY-U025E-TJ

Audio and Video Technologies/UNIX Available Servers/ Real-time Debugging Tools
Vol. 7, No. 4, 1995, EY-U002E-TJ

High Performance Fortran in Parallel Environments/ Sequoia 2000 Research
Vol. 7, No. 3, 1995, EY-T838E-TJ
(Available only on the Internet)

Graphical Software Development/Systems Engineering
Vol. 7, No. 2, 1995, EY-U001E-TJ

Database Integration/Alpha Servers & Workstations/ Alpha 21164 CPU
Vol. 7, No. 1, 1995, EY-T135E-TJ
(Available only on the Internet)

RAID Array Controllers/Workflow Models/ PC LAN and System Management Tools
Vol. 6, No. 4, Fall 1994, EY-T118E-TJ

AlphaServer Multiprocessing Systems/ DEC OSF/1 Symmetric Multiprocessing/ Scientific Computing Optimization for Alpha
Vol. 6, No. 3, Summer 1994, EY-S799E-TJ

Alpha AXP Partners—Cray, Raytheon, Kubota/ DECchip 21071/21072 PCI Chip Sets/ DLT2000 Tape Drive
Vol. 6, No. 2, Spring 1994, EY-F947E-TJ

High-performance Networking/ OpenVMS AXP System Software/ Alpha AXP PC Hardware
Vol. 6, No. 1, Winter 1994, EY-Q011E-TJ

Software Process and Quality
Vol. 5, No. 4, Fall 1993, EY-P920E-DP

Product Internationalization
Vol. 5, No. 3, Summer 1993, EY-P986E-DP

Multimedia/Application Control
Vol. 5, No. 2, Spring 1993, EY-P963E-DP

DECnet Open Networking
Vol. 5, No. 1, Winter 1993, EY-M770E-DP

Alpha AXP Architecture and Systems
Vol. 4, No. 4, Special Issue 1992, EY-J886E-DP

NVAX-microprocessor VAX Systems
Vol. 4, No. 3, Summer 1992, EY-J884E-DP

Semiconductor Technologies
Vol. 4, No. 2, Spring 1992, EY-L521E-DP

PATHWORKS: PC Integration Software
Vol. 4, No. 1, Winter 1992, EY-J825E-DP

Image Processing, Video Terminals, and Printer Technologies
Vol. 3, No. 4, Fall 1991, EY-H889E-DP

Availability in VAXcluster Systems/ Network Performance and Adapters
Vol. 3, No. 3, Summer 1991, EY-H890E-DP

Fiber Distributed Data Interface
Vol. 3, No. 2, Spring 1991, EY-H876E-DP

Transaction Processing, Databases, and Fault-tolerant Systems
Vol. 3, No. 1, Winter 1991, EY-F588E-DP

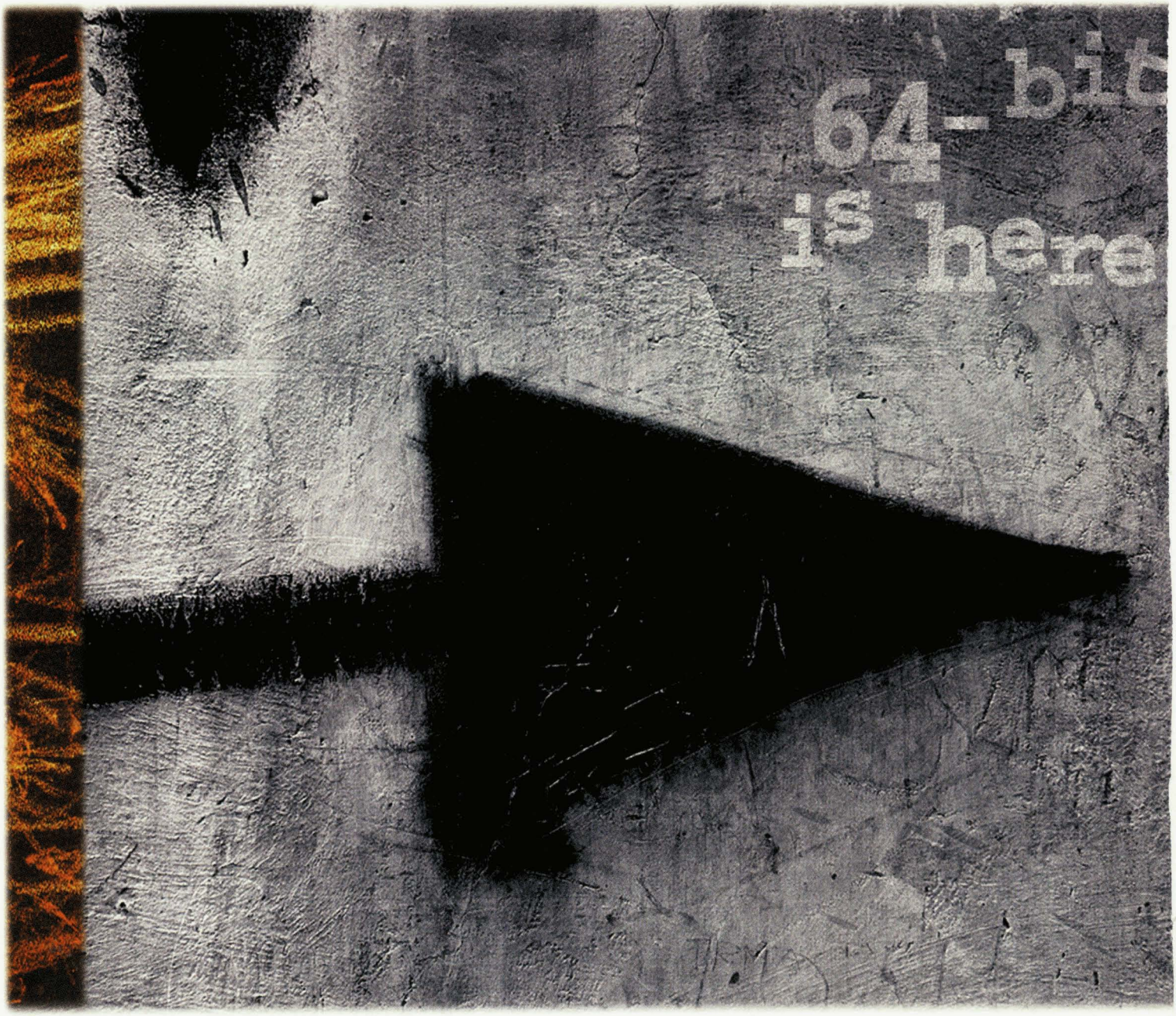
VAX 9000 Series
Vol. 2, No. 4, Fall 1990, EY-E762E-DP

DECwindows Program
Vol. 2, No. 3, Summer 1990, EY-E756E-DP

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990, EY-C197E-DP

Compound Document Architecture
Vol. 2, No. 1, Winter 1990, EY-C196E-DP

digital™



ISSN 0898-901X

Printed in U.S.A. EC-N6992-18/96 9 14 20.0