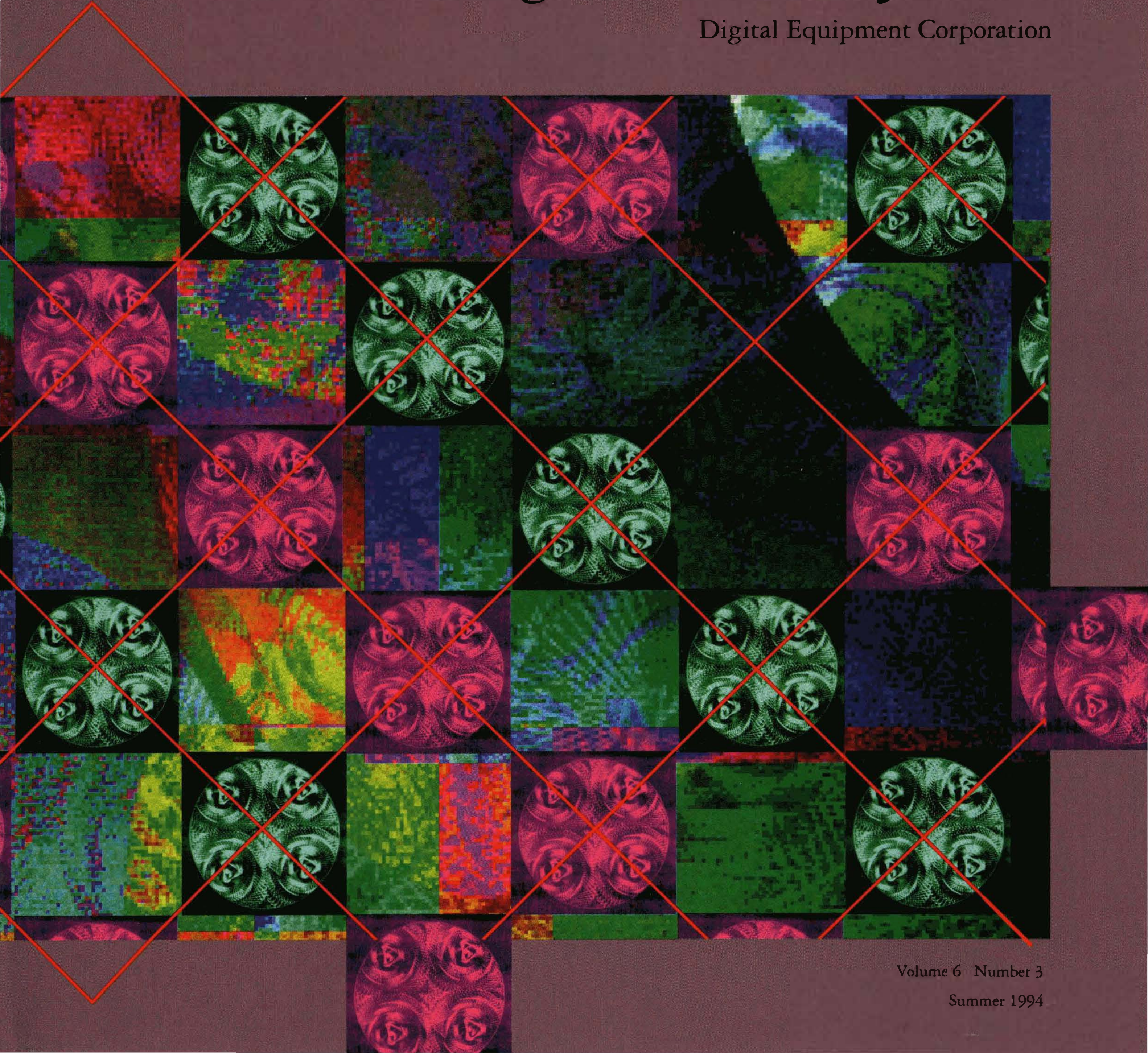
- *AlphaServer Multiprocessing Systems*
- *DEC OSF/1 Symmetric Multiprocessing*
- *Scientific Computing Optimizations for Alpha*

Digital Technical Journal

Digital Equipment Corporation



Volume 6 Number 3

Summer 1994

Editorial
Jane C. Blake, Managing Editor
Helen L. Patterson, Editor
Kathleen M. Stetson, Editor

Circulation
Catherine M. Phillips, Administrator
Dorothea B. Cassady, Secretary

Production
Terri Autieri, Production Editor
Anne S. Katzeff, Typographer
Peter R. Woodbury, Illustrator

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to dtj@digital.com. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are also available on the Internet at http://www.digital.com/info/DTJ/home.html. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital employees may order subscriptions through Readers Choice by entering VTX PROFILE at the system prompt.

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

ISSN 0898-901X

Documentation Number EY-S799E-TJ

Book production was done by Quantic Communications, Inc.

Cover Design

The cover design captures two major concepts in this issue—symmetry and parallelism. At the hardware level, the AlphaServer multiprocessing systems provide symmetrical access to hardware system resources. As processors are added to the multiprocessing system, the DEC OSF/1 operating system provides the parallelism that allows applications to take advantage of the added processor power. The KAP preprocessor also provides parallelism, specifically for programs running on symmetric multiprocessing systems. In each case, symmetry and parallelism are among the keys to achieving designs that offer the highest levels of performance.

The cover was designed by Joe Pozerycki, Jr., of Digital's Design Group.

Contents

Editor's Introduction

Jane C. Blake
Managing Editor

Designs that capitalize on Digital's 64-bit Alpha RISC processors or that enhance the performance of scientific applications are the subjects of papers in this issue. Featured topics include the well-received AlphaServer multiprocessing systems, the DEC OSF/1 symmetric multiprocessing operating system, a high-performance math library, and a preprocessor program developed by Kuck & Associates, Inc.

To develop a price/performance leader for the server market, designers of the AlphaServer 2100 and 2000 multiprocessing systems had to make decisions that were at once creative, pragmatic, and timely. Fidelma Hayes, an engineering manager for the Server Group, presents an overview of these high-performance servers that incorporate Alpha RISC technology and PC-style I/O subsystems, and support three operating systems—Microsoft's Windows NT, DEC OSF/1, and OpenVMS. Because of the engineering team's persistent focus on performance, cost, and time-to-market, all these goals for the AlphaServer systems were surpassed.

Introducing two PC buses in the AlphaServer multiprocessing system was an important factor in market success and an interesting engineering challenge. Andy Russo discusses the benefits of a dual-level I/O structure that contains both the widely used EISA bus and the newer high-performance PCI bus that connects to a 128-bit multiprocessing system bus. He describes several innovative techniques that promote efficiency in the hierarchical bus structure, the advantages offered by the selection of bus bridges (one custom ASIC and one standard chip set), and the I/O interrupt scheme that combines familiar technology with custom support logic.

The next paper presents the significant software work done to ensure high performance and reliability as CPUs are added to the 2100 and 2000 multiprocessing systems. Jeff Denham, Paula Long, and Jim Woodward first review the foundations of DEC OSF/1 version 3.0, Digital's implementation of UNIX for the AlphaServer multiprocessing systems. They then examine issues that arise when moving an operating system from a uniprocessor to a shared-memory SMP platform, in particular, the design team's efforts in lock-based synchronization and algorithm modifications aimed at parallelism within the operating system kernel.

The total impact of 64-bit RISC systems and operating system support for shared memory SMP platforms is demonstrated by meeting the demands of scientific and technical applications. A tool for accelerating application performance on all Alpha systems is the DXML Extended Math Library. Chandrika Kamath, Roy Ho, and Dwight Manley briefly discuss the role of mathematical libraries and then present an overview of DXML components, which include both public domain BLAS and LAPACK libraries and Digital proprietary software. Using example routines, they explain optimization techniques that effectively exploit the memory hierarchy and provide substantial performance improvements.

Another tool for optimizing scientific application performance is KAP, a preprocessor to parallelize DEC Fortran and DEC C programs. As authors Bob Kuhn, Bruce Leasure, and Sanjiv Shah from Kuck & Associates describe it, the KAP product is a super-optimizer, performing optimizations at the source code level that go beyond those performed by the compilers. Their paper reviews adaptations to KAP for SMP systems and the key design aspects, such as data dependence analysis and the selection of loops to parallelize from among many in a program.

The editors thank Andrei Shishov, Mid-range AlphaServers Program Manager, for his help in developing this issue of the *Journal*.

Jane Blake

Biographies

Jeffrey M. Denham A principal software engineer in the UNIX Software Group, Jeffrey Denham is a contributor to the DEC OSF/1 version 3.0 symmetric multiprocessing effort. Prior to this, he helped add POSIX.1b features to the DEC OSF/1 operating system and worked on the VAXELN real-time kernel. Jeff came to Digital in 1986 from Raytheon Corporation. He holds a B.A. (1979) from Hiram College, an M.A. (1980) from Tufts University, both in English, and an M.S. (1985) in Technical Communication from Rensselaer Polytechnic Institute.

Fidelma M. Hayes As an engineering manager in the Server Group, Fidelma Hayes led the development of the AlphaServer 2100 and AlphaServer 2000 systems. Prior to this work, she led the design of the DECsystem 5100. She has contributed as a member of the development team for several projects, including the DECsystem 5800 CPU, the PRISM system design, and the MicroVAX 3100. Fidelma joined Digital in 1984 after receiving a bachelor's degree in electrical engineering from University College Cork, Ireland. She is currently working toward a master's degree in computer science at Boston University.

Roy Ho As a principal software engineer in Digital's High Performance Computing Group, Roy Ho developed the signal-processing routines used in DXML. Prior to this work, he was a member of the High Performance Computing Technology Group. There he designed the clock distribution system for the VAX fault-tolerant system and the delay estimation software package for the VAX 9000 system boards. Roy has B.S. (1985) and M.S. (1987) degrees in electrical engineering from Rensselaer Polytechnic Institute. He joined Digital in 1987.

Chandrika Kamath Chandrika Kamath is a member of the Applied Computational Mathematics Group. She has designed and implemented the sparse linear solver packages that are included in DXML. She has also optimized customer benchmarks for Alpha systems. Chandrika holds a Bachelor of Technology in electrical engineering (1981) from the Indian Institute of Technology, an M.S. in computer science (1984) and a Ph.D. in computer science (1986), both from the University of Illinois at Urbana-Champaign. She has published several papers on numerical algorithms for parallel computers.

Robert H. Kuhn Robert Kuhn joined Kuck & Associates as the Director of Products in 1992. His functions are to formulate technical business strategy and to manage product deliveries. From 1987 to 1992, he worked at Alliant Computer Systems, where he managed compiler development and application software for parallel processing. Bob received his Ph.D. in computer science from the University of Illinois at Champaign-Urbana in 1980. He is the author of several technical publications and has participated in organizing various technical conferences.

Bruce Leasure Bruce Leasure, one of three founders of Kuck & Associates in 1979, serves as Vice President of Technology and is the chief scientist for the company. As a charter member and executive director of the Parallel Computing Forum (PCF), a standards-setting consortium, he was a leader in efforts to standardize basic forms of parallelism. The PCF subsequently became the ANSI X3H5 committee for Parallel Program Constructs for High-level Languages, which he chaired. Bruce received B.S. and M.S. degrees in computer science from the University of Illinois at Champaign-Urbana.

Paula Long Since joining Digital in 1986, Paula Long has contributed to various operating system projects. Presently a principal software engineer with the UNIX Software Group, she leads the development of symmetric multiprocessing (SMP) capabilities for the DEC OSF/1 operating system. In previous positions, she led the DEC OSF/1 real-time and DECwindows on VAXELN projects. Paula received a B.S.C.S. from Westfield State College in 1983.

Dwight P. Manley Dwight Manley is a consulting software engineer in the Applied Computational Mathematics Group. He joined the DXML Group in 1989 and continues to support and enhance the DXML and KAPF products. Since joining Digital in 1979, he has worked on system measurement and modeling projects and was responsible for all performance modeling of the VAX 9000 CPU design. He is listed as a coinventor on 11 patents and as a coauthor of a paper on matrix computation theory. Dwight has a B.S. in mathematics from the University of Massachusetts and an M.S. in operations research from Northeastern University.

Andrew P. Russo Andy Russo is a principal hardware engineer in the Alpha Server Group. Since joining Digital in 1983, Andy has been a project leader for several internal organizations, including the Mid-range I/O Options Group, the Fault Tolerant Group, and the Alpha Server Group. While at Digital, Andy has contributed to the architecture and design of high-performance ASICs and modules to provide a variety of end-product requirements. Andy holds several patents and has authored two papers. He received a B.S. in computer engineering from Boston University.

Sanjiv M. Shah Sanjiv Shah received a B.S. in computer science and mathematics (1986) and an M.S. in computer science and engineering (1988) from the University of Michigan. In 1988, he joined Kuck & Associates' KAP development group as a research programmer. He has since been involved in researching and developing the KAP Fortran and C products and managing the KAP development group. Currently, Sanjiv leads the research and development for parallel KAP performance.

James A. Woodward Principal software engineer James Woodward is a member of the UNIX Software Group. He is responsible for DEC OSF/1 symmetric multiprocessing (SMP) processor scheduling and base kernel support. In previous work, Jim led the ULTRIX SMP project and the VAX 8200, VAX 8800, and VAX 6000 ULTRIX operating system ports. He also wrote microcode for the VAX 8200 systems as a member of the Semiconductor Engineering Group. Jim joined Digital in 1981 after receiving a B.S.E.E. from the University of Michigan.

Foreword

Steve Holmes
Engineering Group Manager, Server Platform Development, and Director, Office Server Product Line

The engineering developments described in this issue represent the second of many planned generations of products that will be designed to fulfill Digital's Alpha vision. That vision is (a) to make Alpha systems open, and (b) to deliver a rich set of Alpha system products that lead the market both in performance and price/performance. It is heartening to see the vision being realized. It is yet more heartening to see it unfolding simultaneously with appreciable improvements in Digital's business practices. These combined events have already resulted in substantial market acceptance of Digital's AlphaServer products.

The particular set of papers in this issue is fortuitous in that it demonstrates the large number of individuals and range of engineering skills required to bring about an industry phenomenon such as Alpha. Included are papers focused on the AlphaServer multiprocessing systems, on the symmetric multiprocessing implementation of the DEC OSF/1 operating system, on the optimization of mathematical subroutine libraries for the Alpha architecture, and on the KAP preprocessor. If one can imagine these technical efforts multiplied manyfold, the scope of the Alpha undertaking will emerge.

The first generation of products based on the Alpha architecture was introduced in 1992. The AlphaServer 2100 system and DEC OSF/1 SMP operating system, introduced in mid-1994, together represent the beginning of the second-generation Alpha server products. The overarching development goal was to give our present and future customers a compelling reason to buy. The resultant direction was to provide very low cost multiprocessing system capability with industry-standard open I/O buses, in this case PCI and EISA. To capitalize on these attributes and to ensure that a complete solution was delivered, the engineering teams maintained a customer-focused perspective. It is this perspective that has enabled the AlphaServer 2100 to achieve rapid market acceptance.

Truly, though, the most significant achievement for the present round of Alpha server products is this: a whole new standard of price/performance for the industry has been reached. Computing that in the past could have been performed only with very expensive high-end machines or extensive distributed networks is now performed by affordable AlphaServer systems.

This price/performance breakthrough augments Digital's strong capabilities.

- A truly open environment that supports UNIX and Windows NT operating systems on Alpha systems
- The ongoing strength of the world's best full-featured commercial operating system, the OpenVMS system
- A world-class and worldwide service and delivery organization
- An extensive and growing network of channels
- Overall, Digital's renewed and meaningful commitment to be responsive to the demands and needs of the markets

This is a very exciting and productive time in Digital's history.

If this were the end of the story, there would be much of which to be proud. In fact, there is more to come across the range of AlphaGeneration products, including workstations, PCs, clustering, operating systems, and networking. In the server area specifically, the recently announced AlphaServer 2000 increases the price/performance lead of the 2100 system. Processor and cache upgrades have increased the absolute performance of the family. Just around the corner are similar advances for other members of Digital's server products. A little further away are significant enhancements in our clustering capabilities and in our server management tools.

All these developments are of direct and measurable benefit to our customers. All are guided by what the markets are telling us they want. The trend and pace of these enhancements will allow Digital to continue to deliver on the promise of the Alpha vision.

Performance measurements, for example, SPECmark data and transaction-per-second tests, and competitive comparisons support the statements above. However, the case is made most convincingly by the early acceptance and rapid ramp up of AlphaServer 2100 system purchases by our customers. In the highly competitive server arena, success is being demonstrated daily.

I would like to take this opportunity to offer a very enthusiastic thank-you to all whose work is represented in the accompanying technical papers, most especially to the AlphaServer 2100 development team whose work I have had the privilege to observe since the team's formation. The hard work and dedication of everyone is recognized, appreciated, and needed for the future.

This foreword will conclude in favor of the substantive papers that detail the technical contributions made by the authors and their colleagues. It is my expectation that readers of this issue of the *Digital Technical Journal* will gain useful technical insights. It is my hope that they will also see, as I do, that the future of Digital computing is bright.

Fidelma M. Hayes

Design of the AlphaServer Multiprocessor Server Systems

Digital's AlphaServer multiprocessor systems are high-performance servers that combine multiprocessing technology with PC-style I/O subsystems. The system architecture allows four processing nodes, four memory nodes (up to a maximum of 2 GB), and two I/O nodes. All nodes communicate through a system bus. The system bus was designed to support multiple generations of Alpha processor technology. The architecture can be implemented in different ways, depending on the size of the system packaging.

The AlphaServer 2100 (large pedestal) and the AlphaServer 2000 (small pedestal) servers from Digital combine multiprocessing Alpha technology with an I/O subsystem traditionally associated with personal computers (PCs). The I/O subsystem in the AlphaServer systems is based on the Peripheral Component Interconnect (PCI) and the Extended Industry Standard Architecture (EISA) buses. All AlphaServer products, including the AlphaServer 2100 cabinet version, share common technology and support at least three generations of the Alpha processor. In addition, the servers support three operating systems: Microsoft's Windows NT version 3.5, and Digital's DEC OSF/1 version 3.0 (and higher) and OpenVMS version 6.1 (and higher).

The AlphaServer systems are designed to be general-purpose servers for PC local area network (LAN) and database applications. All models of the system use a common multiprocessing bus interconnect that supports different numbers of nodes, depending on the system configuration. The systems share a common CPU, memory, and I/O architecture. The number of CPUs, the amount of memory, the number of I/O slots, and the amount of internal storage vary depending on the mechanical packaging. The flexibility of the architecture allows the quick development of new and enhanced systems.

This paper discusses the transformation of a set of requirements into high-performance, cost-effective product implementations. The following section describes the evolution of the AlphaServer design from an advanced development project into a design project. The paper then describes the CPU module, the multiprocessor system bus, and the memory module. Subsequent sections discuss module and silicon technology and the high-availability features incorporated into the design. The paper ends with a performance summary and conclusions about the project.

Concept Development

The engineering investigations of a client-server system originated from a business need that Digital perceived when it introduced the first systems to incorporate the Alpha technology in late 1992. Among Digital's first products in the server market were the DEC 4000 high-performance departmental system, the DEC 3000 deskside workstation/server, and the EISA-based Alpha PC. The lack of an explicitly identified, general-purpose system for the mid-range system market generated many requests from Digital's MicroVAX II system customers. Requests from these customers propelled the AlphaServer product development effort.

From the beginning of the project, two major constraints were evident: The schedule required a product by mid-1994, and the budget was limited. Accordingly, the product team was required to leverage other developments or to find newer, less costly ways of achieving the product goals. Work on the AlphaServer systems started as a joint effort between an advanced development team and a business planning team. The business team developed market profiles and a list of features without which the system would not be competitive. The business team followed a market-driven pricing model. The profit expected from the system dictated the product cost for the system. This cost is referred to as "transfer cost." The business team's cost requirement was critical: if it could not be met, the project would be canceled. Furthermore, the entry-level system was required to

1. Support at least two CPUs, with performance for a single CPU to yield 120 SPECmarks and 100+ transactions per second (TPS) on the TPC-A benchmark.
2. Support at least 1 gigabyte (GB) of memory.
3. Support multiple I/O buses with at least six option slots supported on the base system.
4. Provide high-availability features such as redundant power supplies, redundant array of inexpensive disks (RAID), "warm swap" of drives, and clustering.
5. Provide a number of critical system connectivity options, including Ethernet, fiber distributed data interface (FDDI), and synchronous controllers.
6. Support the Windows NT, the DEC OSF/1, and the OpenVMS operating systems.

Given these criteria, the engineering team decided to base the development of the new server on concepts taken from two Digital products and combine them with the enclosures, power supplies, and options commonly associated with PCs. The DEC 4000 server is a multiprocessor system with a Futurebus+ I/O subsystem; it provided the basis for the multiprocessor bus design.[1] The DECpc 150 PC is a uniprocessor system with an EISA I/O subsystem; it provided a model for designing an I/O subsystem capable of running the Windows NT operating system. The engineering team chose PC-style peripherals because of their low cost.

A strategic decision was made to incorporate the emerging PCI bus into the product in addition to the EISA bus. Major PC vendors had expressed high interest in its development, and they believed the PCI bus would gain acceptance by the PC community. The PCI bus provides a high-performance, low-cost I/O channel that allows connections to many options such as small computer systems interface (SCSI) adapters and other common PC peripherals.

After the initial design had been completed, changing market and competitive environments imposed additional requirements on the design team.

1. The initial transfer cost goal was reduced by approximately 13 percent.
2. Support for a maximum of four processor modules was necessary.

To meet these new requirements, the design team had to modify the system design during the product development phase.

System Overview

The base architecture developed for Digital's AlphaServer multiprocessor systems allows four processing nodes, four memory nodes (up to a maximum of 2 GB), and two I/O nodes. All nodes communicate through a system bus. The system bus was designed to support multiple generations of Alpha processor technology. The architecture can be implemented in different ways, depending on the size of the system packaging. It is flexible enough to meet a variety of market needs. Two implementations of the architecture are the AlphaServer 2100 and the AlphaServer 2000 products. Figure 1 is a block diagram of the AlphaServer 2100 implementation of the architecture.

In the AlphaServer 2100 large pedestal server, the system bus supports eight nodes. It is implemented on a backplane that has seven slots. The seven slots can be configured to support up to four processors. Due to the number of slots available, the server supports only 1 GB of memory when four processors are installed. It supports the full 2 GB of memory with three processors or less. The eighth node, which is the system bus-to-PCI bridge, is resident on the backplane. This provides a 32-bit PCI bus that operates at 33 megahertz (MHz). It is referred to as the primary PCI bus on the system.

A second I/O bridge can be installed in one of the system bus slots. This option, which will be available in 1995, will provide a 64-bit PCI bus for the system. A 64-bit PCI is an extension of a 32-bit PCI bus with a wider data bus. It operates at 33 MHz and is completely interoperable with the 32-bit PCI specification.[2] Options designed for the 32-bit PCI bus will also work in a 64-bit PCI slot.

EISA slots are supported through a bridge on the primary PCI bus on the system. Only one EISA bus can be supported in the system since many of the addresses used by EISA options are fixed.[3] Support of a single EISA bus is not perceived as an issue given the migration from the EISA bus to the much higher performing PCI bus. The maximum supported bandwidth on an EISA bus is 33 megabytes per second (MB/s) versus the maximum bandwidth on a 32-bit PCI bus of 132 MB/s. The EISA bus is used in the system for support of older adapters that have not migrated to PCI.

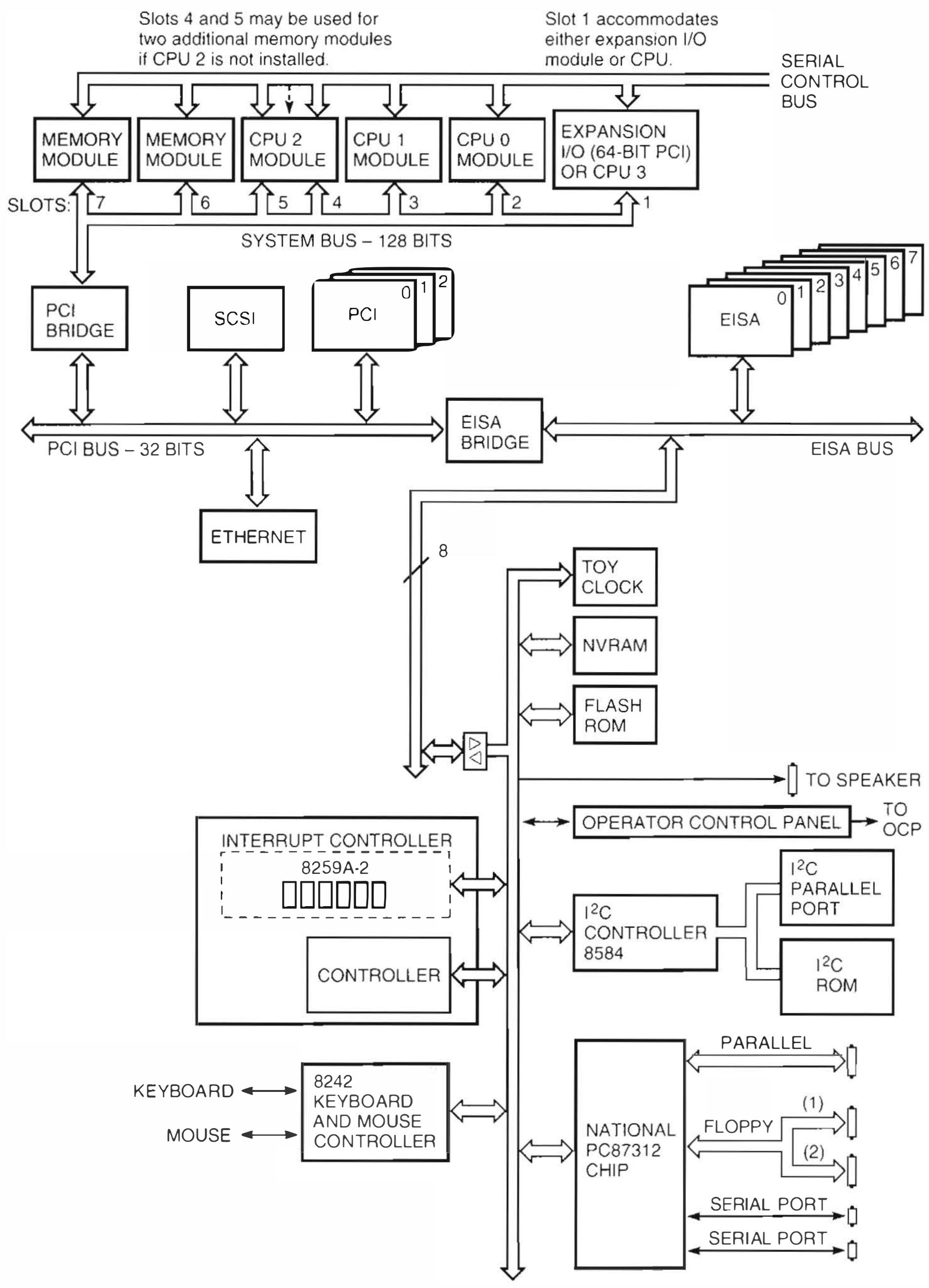


Figure 1 Block Diagram of the AlphaServer 2100 System Architecture

The AlphaServer 2000 small pedestal system supports five nodes on the system bus. The backplane provides four system bus slots, allowing a maximum configuration of two processor modules and two memory modules. The system bus-to-PCI bridge resides on the backplane and is the fifth node. A system bus slot can also be used to support the optional second I/O bridge.

The AlphaServer 2100 cabinet system is a rackmountable version of the large pedestal AlphaServer 2100 system. The rackmountable unit provides a highly available configuration of the pedestal system. It incorporates two separate backplanes. One backplane supports eight system bus nodes that are implemented as seven system bus slots. The eighth node (the system bus-to-PCI bridge) resides on the backplane. The second backplane provides the I/O slots. The number and configuration of I/O slots are identical to the AlphaServer 2100 pedestal system. The rackmount unit provides minimal storage capacity. Additional storage is supported in the cabinet version through StorageWorks shelves. These storage shelves can be powered independently of the base system unit, providing a highly available configuration.

Table 1 gives the specifications for the AlphaServer 2100 and the AlphaServer 2000 pedestal systems. Information on the cabinet version is not included because its characteristics are similar to the AlphaServer 2100 large pedestal version. All multiprocessing members of the AlphaServer family use the same processor and memory modules and differ only in system packaging and backplane implementations. This illustrates the flexibility of the architecture developed for the system and decreases the development time for new models.

CPU Module

The CPU module contains an Alpha processor, a secondary cache, and bus interface application specific integrated circuits (ASICs). As previously mentioned, the system architecture allows multiple processor generations. Multiple variations of the processor module are available for the system, but different variations cannot be used in the same system. Software has timing loops that depend on the speed of the processor and cannot guarantee

Table 1 AlphaServer System Specifications

Specifications	Large Pedestal AlphaServer 2100 System	Small Pedestal AlphaServer 2000 System	Comments
Height, inches	27.6	23.8	
Width, inches	16.9	16.9	
Depth, inches	31.9	25.6	
Maximum DC power output, watts per supply	600	400	Two possible per system in either redundant or current shared mode
Number of system slots	7	4	
Number of processors supported	4	2	
Minimum memory	32 MB	32 MB	
Maximum memory	2 GB	640 MB	
Embedded I/O controllers supported	1	1	
Optional I/O controllers supported	1	1	
32-bit PCI slots	3	3	
64-bit PCI slots (on separate I/O controller module)*	2	2	
EISA slots	8	7	
Serial ports	2	2	
Parallel port	1	1	
Ethernet ports (AUI and 10Base-T)	1	Not integral to system	Up to 18 total network ports supported on system via PCI and EISA options
SCSI II controller	1	1	
Removable media bays	3	2	
Internal warm-swap drive slots	16	8	

* Future option

synchronization between processors of different speeds. The CPU modules provide a range of performance and cost options for the system owner.

The cost-focused processor module uses the Alpha 21064 processor operating at 190 MHz. This chip was designed with Digital's fourth-generation complementary metal-oxide semiconductor (CMOS) technology. It has separate on-chip caches for instruction and data. The instruction cache holds 8 kilobytes (KB) of memory, and the data cache holds 8 KB. The 1-MB second-level data cache is implemented in 15-nanosecond (ns) static random-access memory (SRAM) devices. It is a write-back, direct-mapped cache. The access time to the second-level cache is a multiple of the CPU clock cycle. The use of 15-ns SRAMs resulted in a read-and-write cycle time of 26.3 ns to the second-level cache. This is a five-times multiple of the CPU cycle time. The additional 11.3 ns is needed for round-trip etch delay and address buffer delay. The use of 12-ns SRAMs was considered, but the read-and-write cycle time would have to decrease to 21 ns to improve performance. The reduction of 3 ns was not sufficient to meet the timing requirements of the module; therefore, the less costly 15-ns SRAMs were used.

Higher performance processor modules are also available for the system. These modules are based on the Alpha 21064A processor, which was designed using fifth-generation CMOS technology. The Alpha 21064A processor module operates at 275 MHz. The processor has separate on-chip instruction and data caches. The 16-KB instruction cache is direct mapped, and the 16-KB data cache is a 2-way, set-associative cache. The backup cache holds 4 MB of memory. The combination of higher processor speed, larger internal on-chip caches, and a large second-level cache reduces the number of accesses to main memory and processes data at a higher rate. As a result, the performance of the system is increased by approximately 20 percent.

Multiprocessor System Bus

The technology developed for the system bus in the DEC 4000 departmental server provided the basis for the multiprocessor bus designed for the AlphaServer system.[1] The system bus in the DEC 4000 product has the following features:

1. The 128-bit multiplexed address and data bus operates at a 24-ns cycle time. The bus runs synchronously.
2. The bus supports two CPU nodes, four memory nodes, and a single I/O node.
3. The bus supports addressing for block transfer only. A block is 32 bytes of data.
4. I/O is treated as either primary or secondary. Primary I/O refers to devices that could respond without stalling the system bus. This designation is restricted mainly to control and status registers (CSRs) that exist on system bus nodes, e.g., the I/O bridge.
5. All I/O on remote buses is referred to as secondary I/O and is accessed via a mailbox protocol. Mailboxes were invented to hide slow accesses to CSRs on remote I/O buses.

A CSR read could potentially take 1 to 10 microseconds, which is very slow relative to the processor cycle time. The bus is "nonpended," which means it would stall during a slow access. When a bus stalls, all accesses to CPUs and memories have to wait until the CSR access is complete. This could cause data to back up and potentially overflow. To avoid this state, either the system bus or the software device driver has to be pended.

A mailbox is a software mechanism that accomplishes "device driver pending." The processor builds a structure in main memory called the mailbox data structure. It describes the operation to be performed, e.g., CSR read of a byte. The processor then writes a pointer to this structure into a mailbox pointer register. The I/O node on the system bus reads the mailbox data structure, performs the operation specified, and returns status and any data to the structure in memory. The processor then retrieves the data from this structure and the transaction is complete. In this way, the mailbox protocol allows software pending of CSR reads; it also allows the software to pass byte information that is not available from the Alpha 21064A processor.[4,5]

Changes to the System Bus

Although the DEC 4000 system bus provided many features desirable in a multiprocessor interconnect, it did not meet the system requirements defined during the concept phase of the AlphaServer project. Two major hurdles existed. One was the lack of support for four CPUs and multiple I/O nodes. A second, more important issue was the incompatibility of the mailbox I/O structure with the Windows NT operating system.

The initial port of the Windows NT operating system to the DECpc 150 PC assumed direct-mapped I/O. With direct mapping the I/O is physically mapped into the processor's memory map, and all reads/writes to I/O space are handled as uncached memory accesses. Clearly, this was incompatible with the nonpended bus, which assumes the use of mailboxes. Consequently, the designers studied the advantages and disadvantages of using mailboxes to determine if they should be supported in the Windows NT operating system. They found that the software overhead of manipulating the mailbox structure made CSR accesses approximately three times slower than direct accesses by the hardware. Thus the CPU performing the I/O access waits longer to complete. For this reason, the designers chose not to use mailboxes.

The designers also had to ensure that the system bus would be available for use by other processors while the I/O transaction was completing. To satisfy this requirement, they added a retry mechanism to the system bus. The retry support was very simple and was layered on top of existing bus signals. A retry condition exists when the CPU initiates a cycle to the I/O that cannot be completed in one system bus transaction by the I/O bridge. The CPU involved in the transaction is notified of the retry condition. The CPU then "backs off" the multiprocessor bus and generates that transaction some period of time later. Other processor modules can access memory during the slow I/O transaction. The retry procedure continues until the I/O bridge has the requested data. At that stage, the data is returned to the requesting CPU.

Byte Addressing Byte granularity had been handled in the mailbox data structure. After the direct-mapped I/O scheme was adopted, the designers had to overcome the lack of byte addressability in the Alpha architecture. Therefore, the designers participated in a collaborative effort across Digital to define a mechanism for adding byte addressability in the Alpha architecture. The new scheme required the use of the four lower available Alpha Ad:[08:05] address bits to encode byte masks and lower order address bits for the PCI and EISA buses. For more details, see the paper on the AlphaServer 2100 I/O subsystem in this issue.[6]

The designers required a redefinition of the address map. All I/O devices are now memory mapped. The Alpha 21064A processor has a 34-bit address field that yields an address space of 16 GB. This 16-GB address region may be subdivided into 4-GB quadrants. Each quadrant can be individually marked as cacheable or noncacheable memory. The DEC 4000 system architecture split the 16-GB region in half: 8 GB was allocated as cacheable memory space and the remaining 8 GB as noncacheable space. Memory-mapped I/O devices are mapped into noncacheable space. The decision to support multiple I/O buses in the new systems together with the decision to memory map all I/O (i.e., no mailbox accesses) yielded a noncacheable memory requirement in excess of the 8 GB allocated in the DEC 4000 system. Therefore the designers of the AlphaServer systems changed the address map and allocated a single quadrant (4 GB) of memory as cacheable space and the remaining 12 GB as noncacheable. These 12 GB are used to memory map the I/O.

Arbitration The bus used in the DEC 4000 system supports two CPU nodes and a single I/O node. To achieve the AlphaServer product goals of multiple I/O bridges and multiple CPU nodes, the designers changed the address map to accommodate CSR space for these extra nodes and designed a new arbiter for the system. The arbiter includes enhanced functionality to increase the performance of future generations of processors. Some key features of the arbiter are listed below.

1. The arbiter is implemented as a separate chip on all processor modules. The logic was partitioned into a separate chip to accommodate a flexible architecture and to allow additional arbitrating nodes in the future. As many as four arbiters can exist in the system. Only one arbiter is enabled in the system. It is on the processor installed in slot 2 of the system bus.
2. I/O node arbitration is interleaved with CPU node arbitration. The arbitration is round robin and leads to an ordering scheme of CPU 0, I/O, CPU 1, I/O, CPU 2, I/O, CPU 3, I/O. This scheme attempts to minimize I/O latency by ensuring many arbitration slots for I/O devices. Processors still have more than adequate access to the system bus due to the nature of I/O traffic (generally bursts of data in short periods of time). On an idle bus, the arbiter reverts to a first-come, first-served scheme.
3. The arbiter implements an exclusive access cycle. This allows an arbitrating node to retain the use of the system bus for consecutive cycles. This cycle is used by the I/O bridge in response to a PCI lock cycle. A PCI lock cycle may be generated by a device that requires an atomic operation, which may take multiple transactions to complete. For example, the AlphaServer 2100 and AlphaServer

2000 systems use a PCI-to-EISA bridge chip set (Intel 82430 chip set).[7] This chip set requests a lock cycle on PCI when an EISA device requires an atomic read-modify-write operation.

The use of atomic read-modify-write operations is common in older I/O adapter designs. The I/O bridge on the system bus requests an exclusive access cycle from the arbiter. When it is granted, all buffers in the path to memory are flushed and the device has exclusive use of the PCI and the system bus until its transaction is completed. The use of this mode is not recommended for new adapter designs due to the unfair nature of its tenure on the system bus. It was implemented in the AlphaServer product design to support older EISA devices.

Memory Module

Main memory is accessed over the system bus either by processors (after missing in their on-board caches) or by I/O nodes performing direct memory access (DMA) transactions. They are called commanders.

The memory controller incorporates a number of performance-enhancing features that reduce latency in accessing the dynamic RAM (DRAM) array. One concept used is called a stream buffer. Stream buffers reduce the read latency to main memory. Reads to main memory normally require 9 to 10 cycles on the system bus, depending on the speed of DRAMs in the array. The use of stream buffers reduces this time to 7 cycles. The stream buffers provide a facility to load data fetched from the DRAM array prior to the receipt of a read request for that data.

A stream is detected by monitoring the read addresses from each commander on the system bus. The logic simply keeps a record of the memory addresses of the previous eight read transactions from each commander and compares each subsequent read address to see if the new address is contiguous to any of the recorded addresses. If a new address is determined to be contiguous to any of the previous eight addresses, a new stream is declared. As a result, one of the stream buffers is allocated to a new stream.

A stream buffer is implemented as a four-deep, first-in, first-out (FIFO) buffer. Each entry in the FIFO buffer is 32 bytes, which is equivalent to the system bus line size. Each memory module contains four stream buffers that can be allocated to different commanders. A least recently used (LRU) algorithm is used to allocate stream buffers. When a new stream is detected, or an existing stream is empty, the stream buffer fills from the DRAM array by using successive addresses from the head of the stream. After a buffer has been allocated and some amount of data has been placed in the FIFO buffer, "hit" logic compares incoming read addresses from the system bus to the stream address. If a comparison of these two addresses is successful, read data is delivered from the memory module without incurring the latency of accessing the DRAM array.

An invalidation scheme is used to ensure that the stream buffers stay coherent. Write cycle addresses are checked to see if they coincide with a stream buffer address. If the write address is equal to any address currently in the stream buffer, that entire stream buffer is declared invalid. Once it is invalidated, it can be reallocated to the next detected stream.

Writes to main memory complete on the system bus in six cycles, which is achieved using write buffers in the memory controller. The write transactions are essentially "dump and run." The total write buffering available in each memory module is 64 bytes, which is large enough to ensure that the system bus never has to stall during a write transaction.

The implementation of the memory module differs from the AlphaServer 2100 to the AlphaServer 2000 system. Both memory modules contain the same memory controller ASICs, but the implementation of the DRAM array is different. Due to space constraints on the AlphaServer 2100, the DRAM array was implemented as a flat, two-sided surface-mount module. On the AlphaServer 2000, single in-line memory modules (SIMMs) were used for the DRAM array. Memory module capacities vary from 32 MB to 512 MB. The AlphaServer 2100 system provides four system bus slots that can be populated with memory modules. The maximum supported configuration is 2 GB with four memory modules. This limits the maximum system configuration to three processors since one of the processor slots must be used as a memory slot. The AlphaServer 2000 system provides two system bus slots that can be populated with memory. The maximum memory supported in this system is 640 MB. This configuration consists of one 512-MB module and one 128-MB module. The maximum memory constraint is dictated by the power and cooling available within this system package. The AlphaServer 2000 still supports two processor modules when configured with maximum memory. Figure 2 shows a block diagram of the AlphaServer 2000 memory module.

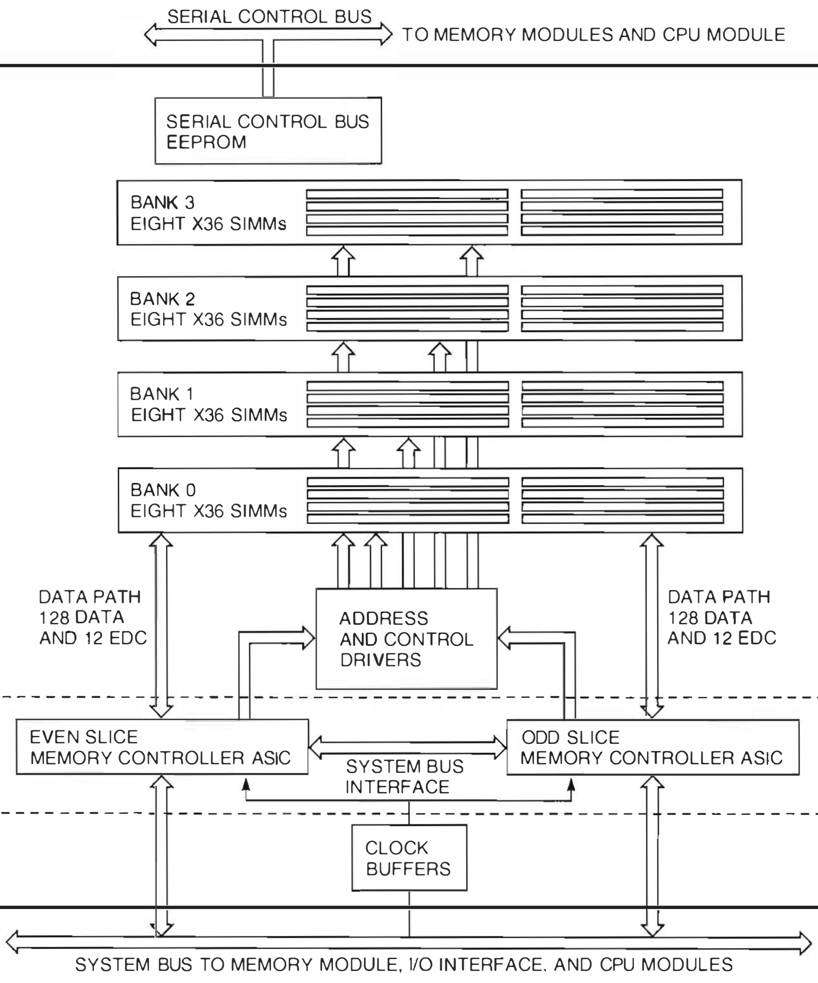

SERIAL CONTROL BUS
TO MEMORY MODULES AND CPU MODULE
SERIAL CONTROL BUS EEPROM
BANK 3 EIGHT X36 SIMMs
BANK 2 EIGHT X36 SIMMs
BANK 1 EIGHT X36 SIMMs
BANK 0 EIGHT X36 SIMMs
DATA PATH 128 DATA AND 12 EDC
ADDRESS AND CONTROL DRIVERS
DATA PATH 128 DATA AND 12 EDC
EVEN SLICE MEMORY CONTROLLER ASIC
SYSTEM BUS INTERFACE
ODD SLICE MEMORY CONTROLLER ASIC
CLOCK BUFFERS
SYSTEM BUS TO MEMORY MODULE, I/O INTERFACE, AND CPU MODULES

Figure 2 Block Diagram of the AlphaServer 2000 Memory Module

Technology Choices

This section briefly discusses some of the decisions and trade-offs made concerning module and silicon technology used in the systems.

Module Technology

The designers partitioned the logic into modules for two reasons: (1) Removable processor and memory modules allow for installation of additional memory and processors and (2) They also allow for easy upgrade to faster processor speeds. Since modularity adds cost to a system, the designers decided that the I/O subsystem (EISA and PCI logic) should reside on the backplane. They deviated from this strategy for the AlphaServer 2100 system design because the PCI-to-EISA bridge was a new, unfamiliar design. Fixing any problems with this chip set or any of the supporting logic would have required a backplane upgrade, which is a time-consuming effort. For this reason, the engineers chose to build an I/O module for the AlphaServer 2100 system that contained the PCI-to-EISA bridge; associated control logic; controllers for mouse, keyboard, printer, and floppy drive; and the integral Ethernet and SCSI controllers. This module was eliminated in the AlphaServer 2000 system due to the design stability of the I/O module.

The Metral connector specified by the Futurebus+ specification was chosen for the system bus implementation on the DEC 4000 product. This choice was consistent with the design of the DEC 4000 server, which is a Futurebus+ system. Cost studies undertaken during the initial design of the AlphaServer 2100 system showed that the cost per pin of the Metral connector was high and added a significant cost to the system. The team decided to investigate the use of either the PCI or the EISA connector for the system bus, since both connectors are used widely in the system. The PCI connector is actually a variant of the MicroChannel Architecture (MCA) connector used in microchannel systems. SPICE simulations showed that it performed better than the Metral connector on the Futurebus+.[8] The team chose a 240-pin version of the connector for implementation because it met the system requirements and had a low cost.

Due to the choice of the MCA connector, the board thickness was limited to a maximum of 0.062 inches. An 8-layer layup was chosen for the module technology. The processor modules had a requirement for both a 5.0-V supply and a 3.0-V supply. The designers chose a split plane to distribute the power rather than two separate power planes for each voltage. Routing high-speed signals across the split was minimized to reduce any emissions that might arise from using a split plane. Testing later validated this approach as emissions from this area were minimal.

Silicon Technology

The system partitioning required the design of four ASICs. These were the CPU bus interface ASIC, the memory bus interface ASIC, the system arbiter, and the system bus-to-PCI bridge. The DEC 4000 implementation of the Futurebus+ used an externally supplied gate-array process that was customized to meet the performance needs of the bus and the performance goals of the first Alpha systems. Gate-array costs are determined by the number of chips that are produced on the chosen gate-array process. The volume of chips produced by the gate-array process for the DEC 4000 system was low because the process was specially adjusted for that system application. As a result, the volume of chips was directly proportional to the volume of the DEC 4000 systems built. Therefore, the cost per component produced by this process was relatively high.

If they had used this customized gate-array process, the designers of the AlphaServer product could not have met their cost goals. They needed a more generic process that could produce chips that many system vendors could use. This would ensure that the line utilization was high and that the cost per component was low. Therefore, they changed the technology to one that is standard in the industry. Gate-array process technology had evolved since the DEC 4000 design, and a standard technology that was capable of meeting the system timing requirements was available. Extensive SPICE simulations verified the process capability. ASICs that were implemented with this process had no difficulty meeting the bus timing.[8]

Another interesting feature of the analog design on the AlphaServer 2100 system involves the support of 11 loads on the PCI. The PCI specification recommends 10 loads as the "cookbook" design.[2] The system requirement on the AlphaServer 2100 was to support three PCI slots, the integral PCI-Ethernet chip, the NCR810 (PCI-to-fast-SCSI controller), and the PCI-to-EISA bridge. Each PCI connector has been modeled to be equivalent to two electrical loads. Taking account of the system bus-to-PCI bridge and the additional load contributed by the I/O module connector yielded a PCI bus with 11 electrical loads. Extensive SPICE simulations of the bus and careful routing to ensure a short bus guaranteed that the new design would meet the electrical specifications of the PCI bus.[8]

System Start-up

The design team incorporated many availability features into the AlphaServer 2100 and AlphaServer 2000 servers. These included support of "hot-swap" storage devices that can be removed or installed while the system is operating, error correction code (ECC)-protected memory, redundant power supplies, and CPU recovery. Perhaps the most interesting part of the design for availability was the emphasis on ensuring that the system had enough built-in recovery and redundancy to allow it to remain in a usable or diagnosable state. Large systems sometimes have complicated paths in which to access the initial start-up code, and a system failure in that path can leave the owner with no visible failure indication. Moreover, in a multiprocessor system with more than one CPU installed, it is highly desirable to initialize the resident firmware and the operating system even if all CPUs are not in working order. The AlphaServer 2100 and 2000 systems employ two schemes to help achieve this goal.

The start-up code for the AlphaServer 2100 and AlphaServer 2000 systems is located in flash read-only memory (ROM), which resides on a peripheral bus behind the PCI-to-EISA bridge. In starting up a multiprocessing operating system, only one processor is designated to access the start-up code and initialize the operating system. This is referred to as the primary processor. Accessing the start-up code requires the processor, system bus, memory, and most of the I/O subsystem to be functional.

The AlphaServer systems have a number of features that help make the start-up process more robust. Each processor module contains a separate maintenance processor implemented as a simple microcontroller that connects to a serial bus on the system. The serial bus is a two-wire bus that has a data line and a clock line. On power-up the processor module performs a number of diagnostic tests and logs the results in an electrically erasable programmable read-only memory (EEPROM) on the module. This EEPROM resides on the serial bus. If a CPU fails one of its power-up tests or if it has an error logged in its EEPROM, then it is not allowed to be the primary processor. Assume that four CPUs are installed in the system; if only CPU 0 fails, then CPU 1 is the primary processor. If CPU 0 and CPU 1 fail, then CPU 2 is the primary processor. If CPU 0, CPU 1, and CPU 2 fail, then CPU 3 is the primary processor. If all four CPUs fail, then CPU 0 is the primary processor. If any one of the CPUs fails, a message is displayed on the operator control panel to inform the user that there is a problem. Any secondary CPU that has failed is disabled and will not be seen by the firmware console or the operating system. The primary processor then uses the system bus to access the start-up code in the flash ROM.

The flash ROM may contain incorrect data. The flash ROMs on many systems have a program update, and errors from a power spike or surge can be introduced into the ROM code during the update procedure. User error is another common way to introduce data error; for example, a user can accidentally press a key while the update program is running. Flash ROMs can also fail from intrinsic manufacturing faults such as current leakage, which will eventually convert a stored "1" into a stored "0," thus corrupting the program stored in the flash ROMs. Many techniques in the industry partially solve the problem of corrupted flash ROM data. One well-known technique uses a checksum and reports an error to the user if the data is not correct. Another technique provides a second set of flash ROMs and a switch that the user manipulates to transfer control to the new set in the event of a failure. The designers studied many previously used methods, but rejected them since they required intervention by the user.

In the AlphaServer 2100 and the AlphaServer 2000 system design, the design team implemented a scheme that did not require user intervention in the event of flash ROM corruption. The system has 1 MB of flash ROM of which the first 512 KB contain the system initialization code. This code is loaded into main memory, and many data integrity tests are performed. These include single and multiple bit parity checks, various data correction code checking, and a checksum calculation. The processor detects an error if the checksum calculation fails, i.e., if the calculated value is not equal to the stored value. The processor then writes a value to a register on the I/O module, which automatically changes the address pointing to the flash ROM to a second bank of flash ROM. This combination of hardware and software support provides a way for the AlphaServer 2100 system user to overcome any flash ROM corruption.

Design Considerations for the AlphaServer 2000 System

The design of the AlphaServer 2000 small pedestal system followed the AlphaServer 2100 system. Market pressures dictated the need for a smaller system with a lower entry-level cost. The introduction of the smaller server was scheduled to coincide with the release of the Windows NT version 3.5 operating system.

An examination of the AlphaServer 2100 development schedule revealed the following interesting points: (1) System power on occurred nine months after the team was formed; (2) Initial system shipments occurred eight months later; (3) The eight-month time period was spent mainly in porting and qualifying operating system software.

Based on these facts, the system designers believed that the key to reducing the time-to-market of the AlphaServer 2000 system was to eliminate the dependency on synchronizing the design schedule with an operating system release. Consequently, the new system could not require any software changes at the operating system level. Any changes would have to be transparent to software. To achieve this, the designers took advantage of a new feature in the DEC OSF/1 and the OpenVMS operating systems called dynamic system recognition (DSR).

A DSR machine is defined as a machine that requires no new software development. Operating systems, however, require licensing; this information is dependent upon the system model number. There are two components to building a DSR machine.

1. A programmer's view of the machine must be a subset of an already supported machine. In the case of the AlphaServer 2000, the designers

decided to make it a subset of the AlphaServer 2100. A clear understanding of how the operating systems initialized the AlphaServer 2100 system was critical to understanding what changes could be made. A joint team of hardware and software engineers examined various pieces of the code to identify the areas of the system design that could be changed. Investigations revealed that the system bus configuration code for the AlphaServer 2100 is somewhat generic. It assumes a maximum of eight nodes, which is the AlphaServer 2100 implementation. The I/O node to the primary PCI bus is expected to be present. The presence of additional processors and memories is detected by reading the CSR space of each module. A module that is present gives a positive acknowledgment. The design team could therefore reduce the number of system bus slots from seven to four. This had no effect on the software since nonexistent slots would merely be recognized as modules not installed in the system.

The physical packaging of the AlphaServer 2000 also dictated that the number of I/O slots be reduced from 11 (8 EISA and 3 PCI) to 10. Given the industry trend toward PCI, the desirable mix would have been 6 EISA slots and 4 PCI slots. The PCI bus configuration code searched for as many as 32 PCI slots, which is the number allowed by the PCI specification.[2] After careful consideration, the designers determined that the addition of another PCI slot would involve a change in interrupt tables to accommodate the additional interrupts and vectors required by the additional slot. Therefore, the team decided to implement 3 PCI and 7 EISA slots.

2. The other component to building a DSR machine is to provide the system model number to the operating system so that licensing information can be determined. The system resident code that runs at start-up is referred to as the console. The console and the operating systems communicate via a data structure known as the hardware parameter block (HWRPB). The HWRPB is used to communicate the model number to the operating system, which uses this number to provide the correct licensing information.

The AlphaServer 2000 system was completed in approximately nine months. Qualification was not dependent on the operating system schedules. By building a DSR machine, the design team met the project's time-to-market requirements.

Performance Summary

Table 2 summarizes the performance of the systems described in this paper. The numbers are heavily influenced by the processor speed, cache, memory, and I/O subsystems. The systems exceeded the performance goals specified at the beginning of the project. In some cases the important benchmarks that had been relevant in the industry changed during the course of system development. In the transaction processing measurement, for example, the TPC-A benchmark was superseded by the TPC-C benchmark.

The AlphaServer 2100 server was the price-performance leader in the industry at the time of its introduction in April 1994. Successive improvements in processor and I/O subsystems should help the AlphaServer 2100 and 2000 products maintain that position in the industry.

Table 2 System Performance

	AlphaServer 2100 4/275	AlphaServer 2000 4/200
SPECint92*	200.1	131.8
SPECfp92*	291.1	161.0
AIM III†		
Number of AIMs	227.5	177.5
User loads	1941.2	1516.0
Estimated TPS‡	850	660

Notes:

* Single-processor system only

† Dual-processor system only

‡ TPS is an abbreviation for transactions per second. These numbers are estimated for a quad-processor system using OpenVMS version 6.1 running Rdb.

Conclusions

The design team exceeded all the product requirements set at the beginning of the AlphaServer project. The transfer cost of the final product was 10 percent better than the goal. The reduced cost was achieved despite the erratic price levels for DRAMs, which were much higher in 1994 than predicted in late 1992.

Separate cost targets were established for each portion of the system, and each design engineer was responsible for meeting a particular goal.

Constant cost reviews ensured that variances could be quickly addressed. The requirement to run three operating systems quickly expanded the size and scope of the project. The operating system developers became an integral part of the design team. Multiple reviews and open communication between the hardware development team and the software groups were essential to managing this work. The hardware team performed system-level testing on all three operating systems. This proved invaluable in tracking down bugs quickly and resolving them in either hardware or software.

The project team delivered the expected performance and functionality on schedule. Development time was allocated for new power and packaging subsystems (using third-party design companies), new modules, new ASICs, new system firmware, and porting of three operating systems. To attain the schedule, development tasks were frozen at the beginning of the project. The tasks were also categorized into three classes: mandatory, nonessential, and disposable. Consequently, engineers were able to make trade-offs when required and maintain the integrity of the product. Another key factor to meeting the schedule was the use of knowledge and technology developed for previous products. This yielded many benefits: less design time, fewer resources required, known simulation environment, and less time to a working prototype.

Acknowledgments

Many people contributed to the success of this project. They spent long weekends and nights working to a schedule and a set of requirements that many thought were unachievable. The inspired dedication of team members made this project a reality. Although not complete, the following list credits those who were the most deeply involved: Vince Asbridge, Kate Baumgartner, Rachael Berman, Jack Boucher, John Bridge, Bob Brower, Harold Buckingham, Don Caley, Steve Campbell, Dave Carlson, Mike Chautin, Marco Ciaffi, Doug Field, Rich Freiss, Nitin Godiwala, Judy Gold, Paul Goodwin, Tom Hunt, Paul Jacobi, Steve Jenness, Jeff Kerrigan, Will Kerchner, Jeff Metzger, John Nerl, Mike O'Neill, Kevin Peterson, Bryan Porter, Ali Rafiemeyer, Lee Ridlon, Andy Russo, Stephen Shirron, Andrei Shishov, Gene Smith, Kurt Thaller, Frank Touserkani, Vicky Triolo, and Ralph Ware. Special thanks also to our manufacturing team in Massachusetts, Canada, and Scotland.

References and Note

1. B. Maskas, S. Shirron, and N. Warchol, "Design and Performance of the DEC 4000 AXP Departmental Server Computing Systems," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 82-99.

2. *PCI Local Bus Specification, Revision 2.0* (Hillsboro, OR: PCI Special Interest Group, Order No. 281446-001, April 1993).

3. E. Solari, *ISA and EISA, Theory and Operation* (San Diego, CA: Annabooks, 1992).

4. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, Order No. EY-L520E-DP, 1992).

5. *DECchip 21064 Microprocessor Hardware Reference Manual* (Maynard, MA: Digital Equipment Corporation, Order No. EC-N0079-72, 1992).

6. A. Russo, "The AlphaServer 2100 I/O Subsystem," *Digital Technical Journal,* vol. 6, no. 3 (Summer 1994, this issue): 20-28.

7. *82420/82430 PCIset ISA and EISA Bridges* (Santa Clara, CA: Intel Corporation, 1993).

8. SPICE is a general-purpose circuit simulator program developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Andrew P. Russo

The AlphaServer 2100 I/O Subsystem

The AlphaServer 2100 I/O subsystem contains a dual-level I/O structure that includes the high-powered PCI local bus and the widely used EISA bus. The PCI bus is connected to the server's multiprocessing system bus through the custom-designed bridge chip. The EISA bus supports eight general-purpose EISA/ISA connectors, providing connections to plug-in, industry-standard options. Data rate isolation, disconnected transaction, and data buffer management techniques were used to ensure bus efficiency in the I/O subsystem. Innovative engineering designs accomplished the task of combining Alpha CPUs and standard-system I/O devices.

Digital's AlphaServer 2100 server combines Alpha multiprocessing technology with an I/O subsystem typically associated with personal computers (PCs).[1] The I/O subsystem on the AlphaServer 2100 system contains a two-level hierarchical bus structure consisting of a high-performance primary I/O bus connected to a secondary, lower performance I/O bus. The primary I/O bus is a 32-bit peripheral component interconnect (PCI) local bus (or simply, PCI bus).[2] The PCI bus is connected to the AlphaServer 2100 system's multiprocessing system bus through a custom application specific integrated circuit (ASIC) bridge chip (referred to as the T2 bridge chip). The secondary I/O bus is a 32-bit Extended Industry Standard Architecture (EISA) bus connected to the PCI bus through a bridge chip set provided by Intel Corporation.[3] Figure 1 shows the I/O subsystem designed for the AlphaServer 2100 product. The I/O subsystem demonstrated sufficient flexibility to become the I/O interface for the small pedestal AlphaServer 2000 product and the rackmountable version of the AlphaServer 2100 server.

This paper discusses the dual-level bus hierarchy and the several I/O advantages it provides. The design considerations of the I/O subsystem for the AlphaServer 2100 server are examined in the sections that follow.

I/O Support for EISA and PCI Buses

The EISA bus enables the AlphaServer 2100 system to support a wide range of existing EISA or Industry Standard Architecture (ISA) I/O peripherals.[4] The EISA bus can sustain data rates up to a theoretical limit of 33 megabytes per second (MB/s) at a clock rate of 8.25 megahertz (MHz). In the current configuration for the AlphaServer 2100 product, the EISA bus supports eight general-purpose EISA/ISA connectors, and the EISA bridge chip set provides connections to various low-speed, system-standard I/O devices such as keyboard, mouse, and time-of-year (TOY) clock. For most system configurations, the AlphaServer 2100 system's EISA bus provides enough data bandwidth to meet all data throughput requirements. In light of the new requirements for faster data rates, however, the EISA bus will soon begin to run out of bus bandwidth.

To provide for more bandwidth, the AlphaServer 2100 system also contains a PCI bus as its primary bus. With data rates four times that of the EISA bus, the PCI bus provides a direct migration path from the EISA bus. The 32-bit PCI bus can sustain data rates up to a theoretical limit of 132 MB/s at a clock rate of 33 MHz. In the AlphaServer 2100 system configuration, the PCI bus provides connections to three general-purpose 32-bit PCI connectors, an Ethernet device, a SCSI device, the PCI-to-EISA bridge chip, and the T2 bridge chip.

A close examination of the bus structure reveals that the AlphaServer 2100 system actually contains a three-level, hierarchical bus structure. In addition to the PCI and EISA buses, the AlphaServer 2100 system includes a 128-bit multiprocessing system bus, as shown in Figure 1. Each bus is designed to adhere to its own bus interface protocols at different data rates. The system bus is 128 bits per 24 nanoseconds (ns); the PCI bus is 32 bits per 30 ns; and the EISA bus is 32 bits per 120 ns. Each bus is required

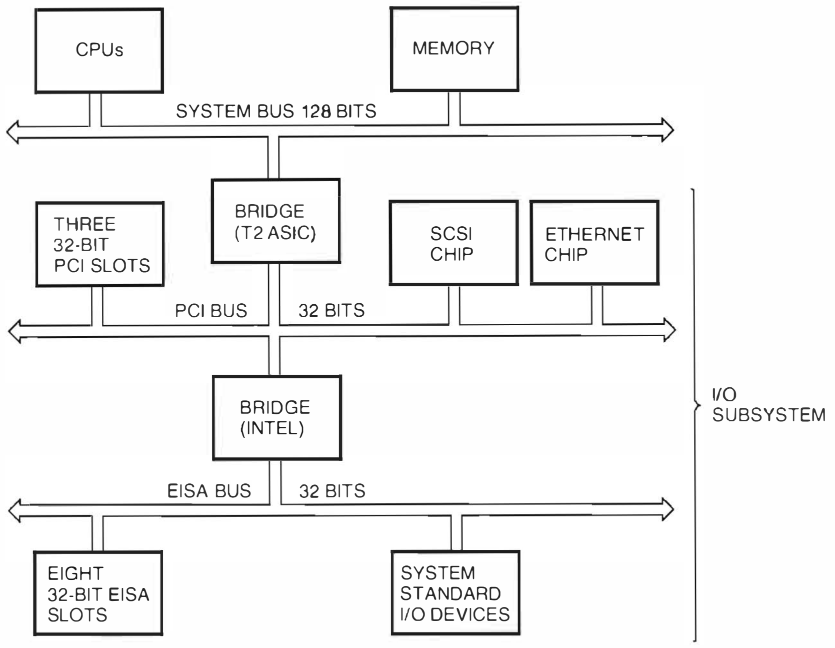


Figure 1 I/O Subsystem for the AlphaServer 2100 System

to provide a particular function to the system and is positioned in the bus hierarchy to maximize that efficiency. For example, the system bus is positioned close to the CPUs and memory to maximize CPU memory access time, and the lower performance I/O devices are placed on the EISA bus because their timing requirements are less critical. To maintain maximum bus efficiency on all three buses, it is critical that each bus be able to perform its various functions autonomously of each other. In other words, a slower performing bus should not affect the efficiency of a high-performance bus. The section below discusses a few techniques that we designed into the I/O subsystem to enable the buses to work together efficiently.

Using the Bus Hierarchy Efficiently

This section discusses the data rate isolation, disconnected transaction, data buffer management, and data bursting techniques used to ensure bus efficiency in the I/O subsystem.

Data Rate Isolation

The three-level bus hierarchy promotes data rate isolation and concurrency for simultaneous operations on all three buses. The design of the bus bridges helps to enable each bus to work independently: it provides bus interfaces with extensive data buffering that function at the same data rates as the interfacing bus. For example, the T2 bridge chip contains both a system bus interface and a PCI bus interface that run synchronously to their respective buses but are totally asynchronous to each other. The data buffers inside the T2 bridge chip act as a domain connector from one bus time zone to the other and help to isolate the data rates of the two buses.

Disconnected Transactions

Whenever possible, the bridges promote the use of disconnected (or pended) protocols to move data across the buses. Disconnected protocols decrease the interdependencies between the different buses. For example, when a CPU residing on the system bus needs to move data to the PCI bus, the CPU does so by sending its data onto the system bus. Here the T2 bridge chip (see Figure 2) stores the data into its internal data buffers at the system bus data rate. The T2 bridge chip provides enough buffering to store an entire CPU transaction. From the CPU's perspective, the transaction is completed as soon as the T2 bridge chip accepts its data. At that point, the T2 bridge chip must forward the data to the PCI bus, independent of the CPU. In this way, the CPU

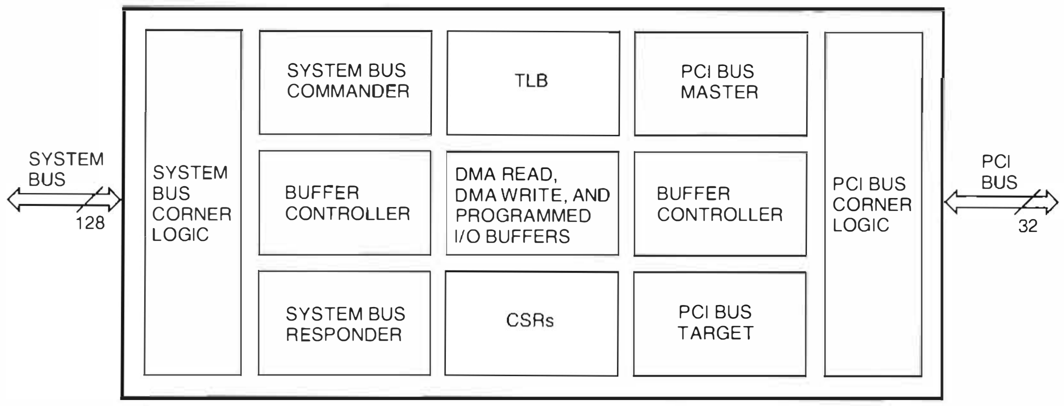


Figure 2 Block Diagram of the T2 Bridge Chip

is not required to waste bus bandwidth by waiting for the transfer to complete to its final destination on the PCI bus.

The T2 bridge chip implements disconnected transactions for all CPU-to-PCI transactions and most PCI-to-memory transactions. In a similar fashion, the PCI-to-EISA bridge implements disconnected transactions between the PCI bus and the EISA bus.

Data Buffer Management

In addition to containing temporary data buffering to store data on its journey from bus to bus, each bridge chip utilizes buffer management to allocate and deallocate its internal data buffers from one incoming data stream to another. In this way, a single ASIC bridge design can efficiently service multiple data streams with a relatively small amount of data buffering and without impacting bus performance.

The T2 bridge chip contains 160 bytes of temporary data buffering divided across the three specific bus transactions it performs. These three transactions are (1) direct memory access (DMA) writes from PCI to memory (system bus), (2) DMA reads from memory (system bus) to PCI, and (3) programmed I/O (system bus) reads/writes by a CPU from/to the PCI. The T2 bridge chip's data buffering is organized into five 32-byte buffers. Two 32-byte buffers each are allocated to the DMA write and DMA read functions, and one 32-byte buffer is allocated to the programmed I/O function. Each of the three transaction functions contains its own buffer management logic to determine the best use of its available data buffering. Buffer management is especially valuable in situations in which a PCI device is reading data from memory on the system bus. To maintain an even flow of data from bus to bus, the buffer management inside the T2 bridge chip attempts to prefetch more read data from memory while it is moving data onto the PCI.

Buffer management helps the bridges service bus transactions in a way that promotes continuous data flow that, in turn, promotes bus efficiency.

Burst Transactions

Using a bus efficiently also means utilizing as much of the bus bandwidth as possible for "useful" data movement. Useful data movement is defined as that section of time when only the actual data is moving on the bus, devoid of address or protocol cycles. Maximizing useful data movement can be accomplished by sending many data beats (data per cycle) per single transfer time. Sending multiple data beats per single transfer is referred to as a "burst transaction."

All three buses have the ability to perform burst transactions. The system bus can burst as much as 32 bytes of data per transaction, and the PCI and EISA buses can burst continuously as required.

Data bursting promotes bus efficiency and very high data rates. Each bus bridge in the server is required to support data bursting.

The Bus Bridges

In the previous section, we discussed certain design techniques used to promote efficiency within the server's hierarchical bus structure. The section that follows describes the bus bridges in more detail, emphasizing a few interesting features.

The T2 Bridge Chip

The T2 bridge chip is a specially designed ASIC that provides bridge functionality between the server's multiprocessing system bus and the primary PCI bus. (See Figures 1 and 2.) The T2 ASIC is a 5.0-volt chip designed in complementary metal-oxide semiconductor (CMOS) technology. It is packaged in a 299-pin ceramic pin grid array (CPGA).

As stated earlier, the T2 bridge chip contains a 128-bit system bus interface running at 24 ns and a 32-bit PCI interface running at 30 ns. By using these two interfaces and data buffering, the T2 bridge chip translates bus protocols in both directions and moves data on both buses, thereby providing the logical system bus-to-PCI interface (bridge). In addition to the previously mentioned bridge features, the T2 bridge chip integrates system functions such as parity protection, error reporting, and CPU-to-PCI address and data mapping, which is discussed later in the section Connecting the Alpha CPU to the PCI and EISA Buses.

The T2 bridge chip contains a sophisticated DMA controller capable of servicing three separate PCI masters simultaneously. The DMA controller supports different-size data bursting (e.g., single, multiple, or continuous) and two kinds of DMA transfers, direct mapped and scatter/gather mapped. Both DMA mappings allow the T2 bridge chip to transfer large amounts of data between the PCI bus and the system bus, independent of the CPU.

Direct-mapped DMAs use the address generated by the PCI to access the system bus memory directly. Scatter/gather-mapped DMAs use the address generated by the PCI to access a table of page frame numbers (PFNs) in the system bus memory. By using the PFNs from the table, the T2 bridge chip generates a new address to access the data. To enhance the performance of scatter/gather-mapped DMAs, the T2 bridge chip contains a translation look-aside buffer (TLB) that contains eight of the most recently used PFNs from the table. By storing the PFNs in the TLB, the T2 bridge chip does not have to access the table in system bus memory every time it requires a new PFN. The TLB improves scatter/gather-mapped DMA performance and conserves bus bandwidth. Each entry in the TLB can be individually invalidated as required by software.

The T2 bridge chip also contains a single I/O data mover that enables a CPU on the system bus to initiate data transfers with a device on the PCI bus. The I/O data mover supports accesses to all the valid PCI address spaces, including PCI I/O space, PCI memory space, and PCI configuration space. The T2 bridge chip supports two I/O transaction types when accessing PCI memory space: sparse-type data transfers and dense-type data transfers. Sparse-type transfers are low-performance operations consisting of 8-, 16-, 24-, 32-, and 64-bit data transactions. Dense-type transfers are high-performance operations consisting of 32-bit through 32-byte data transactions. Dense-type transfers are especially useful when accessing I/O devices with large data buffers, such as video graphics adapter (VGA) controllers. A single PCI device mapped into PCI memory space can be accessed with either sparse-type operations, dense-type operations, or both.

In addition to accessing the PCI, a CPU can access various T2 bridge chip internal control/status registers (CSRs) for setup and status purposes. For maximum flexibility, all the T2 bridge chip's functions are CSR programmable, allowing for a variety of optional features. All CPU I/O transfers, other than those to T2 bridge chip CSRs, are forwarded to the PCI bus.

Intel PCI-to-EISA Bridge Chip Set

The Intel PCI-to-EISA bridge chip set provides the bridge between the PCI bus and the EISA bus.[3] It integrates many of the common I/O functions found in today's EISA-based PCs. The chip set incorporates the logic for a PCI interface running at a clock rate of 30 ns and an EISA interface running at a clock rate of 120 ns. The chip set contains a DMA controller that supports direct- and scatter/gather-mapped data transfers, with a sufficient amount of data buffering to isolate the PCI bus from the EISA bus. The chip set also includes PCI and EISA arbiters and various other support control logic that provide decode for peripheral devices such as the flash read-only memories (ROMs) containing the basic I/O system (BIOS) code, real-time clock, keyboard/mouse controller, floppy controller, two serial ports, one parallel port, and hard disk drive. In the AlphaServer 2100 system, the PCI-to-EISA bridge chip set resides on the standard I/O module, which is discussed later in this paper.

Connecting the Alpha CPU to the PCI and EISA Buses

In the next section, we discuss several interesting design challenges that we encountered as we attempted to connect PC-oriented bus structures to a high-powered multiprocessing Alpha chassis.

Address and Data Mapping

When a CPU initiates a data transfer to a device on the PCI bus, the T2 bridge chip must first determine the location (address) and amount of data (mask) information for the requested transaction and then generate the appropriate PCI bus cycle. This issue is not straightforward because the PCI and EISA buses both support data transfers down to the byte granularity, but the Alpha CPU and the system bus provide masking granularity only down to 32 bits of data.

To generate less than 32-bit addresses and byte-masked data transactions on the PCI bus, the T2 bridge chip needed to implement a special decoding scheme that converts an Alpha CPU-to-I/O transaction, as it appears on the system bus, to a correctly sized PCI transaction. Tables 1 and 2 give the low-order Alpha address bits and Alpha 32-bit mask fields and show how they are encoded to generate the appropriate PCI address and data masks. By using this encoding scheme, the Alpha CPU can perform read and write transactions to a PCI device mapped in either PCI I/O, PCI memory, or PCI configuration space with sparse-type transfers. (Sparse-type transfer sizes have 8-, 16-, 24-, 32-, or 64-bit data granularity.)

Another mapping problem exists when a PCI device wants to move a byte of data (or anything smaller than 32 bytes of data) into the system bus

Table 1 CPU-to-PCI Read Size Encoding

Transaction Size	EV_Addr[6:5]	EV_Addr[4:3]	Instructions	PCI Byte Enables (L)	PCI_AD[1:0]	Data Returned to Processor, EV_Data[127:0]
8 bits	00	00	LDL	1110	00	OW_0:[D7:D0]
	01	00	LDL	1101	01	OW_0:[D15:D8]
	10	00	LDL	1011	10	OW_0:[D23:D16]
	11	00	LDL	0111	11	OW_0:[D31:D24]
16 bits	00	01	LDL	1100	00	OW_0:[D79:D64]
	01	01	LDL	1001	01	OW_0:[D87:D72]
	10	01	LDL	0011	10	OW_0:[D95:D80]
24 bits	00	10	LDL	1000	00	OW_1:[D23:D0]
	01	10	LDL	0001	01	OW_1:[D31:D8]
32 bits	00	11	LDL	0000	00	OW_1:[D95:D64]
64 bits	11	11	LDQ	0000 0000	00	OW_1:[D95:D64] OW_1:[D127:D96]

Table 2 CPU-to-PCI Write Size Encoding

Transaction Size	EV_Addr[6:5]	EV_Addr[4:3]	EV_Mask[7:0] (H)	Instructions	PCI Byte Enables (L)	PCI_AD[1:0]	Data Returned to Processor, EV_Data[127:0]
8 bits	00	00	00000001	LDL	1110	00	OW_0:[D7:D0]
	01	00	00000001	LDL	1101	01	OW_0:[D15:D8]
	10	00	00000001	LDL	1011	10	OW_0:[D23:D16]
	11	00	00000001	LDL	0111	11	OW_0:[D31:D24]
16 bits	00	01	00000100	LDL	1100	00	OW_0:[D79:D64]
	01	01	00000100	LDL	1001	01	OW_0:[D87:D72]
	10	01	00000100	LDL	0011	10	OW_0:[D95:D80]
24 bits	00	10	00010000	LDL	1000	00	OW_1:[D23:D0]
	01	10	00010000	LDL	0001	01	OW_1:[D31:D8]
32 bits	00	11	01000000	LDL	0000	00	OW_1:[D95:D64]
64 bits	11	11	11000000	LDQ	0000 0000	00	OW_1:[D95:D64] OW_1:[D127:D96]

memory. Neither the system bus nor its memory supports byte granularity data transfers. Therefore, the T2 bridge chip must perform a read-modify-write operation to move less than 32 bytes of data into the system bus memory. During the read-modify-write operation, the T2 bridge chip first reads a full 32 bytes of data from memory at the address range specified by the PCI device.[2] It then merges the old data (read data) with the new data (PCI write data) and writes the full 32 bytes back into memory.

ISA Fixed-address Mapping

We encountered a third interesting mapping problem when we decided to support certain ISA devices with fixed I/O addresses in the AlphaServer 2100 system. These ISA devices (e.g., ISA local area network [LAN] card or an ISA frame buffer) have fixed (hardwired) memory-mapped I/O addresses in the 1-MB to 16-MB address range.

The ISA devices being discussed were designed for use in the first PCs, which contained less than 1 MB of main memory. In these PCs, the I/O devices had fixed access addresses above main memory in the 1-MB to 16-MB address range. Today's PCs have significantly more physical memory and use the 1-MB to 16-MB region as a part of main memory. Unfortunately, these ISA devices were never redesigned to accommodate this change. Therefore, to support these ISA options, the PC designers created I/O access gaps in main memory in the 1-MB to 16-MB address range. With this technology, an access by a CPU in that address range is automatically forwarded to the ISA device.

To remain compatible with the ISA community, the T2 bridge chip also had to allow for a gap in main memory at the 1-MB to 16-MB address range so that these addresses could be forwarded to the appropriate ISA device.

BIOS Caching Compatibility

Today's Microsoft-compatible PCs provide another performance-enhancing mechanism. We decided to implement this function inside the T2 bridge chip as well.

During system initialization, MS-DOS–based PCs read several BIOS ROMs from their I/O space. Once the ROMs are read, their contents are placed in fixed locations in main memory in the 512-kilobyte (KB) to 1-MB address range. The software then has the ability to mark certain addresses within this range as read cacheable, write cacheable, read noncacheable, or write noncacheable. The basic intention is to mark frequently accessed sections of code as read cacheable but write noncacheable. In this way, read accesses "hit" in main memory (or cache), and writes update the ROMs directly.

Interrupt Mechanism

No computer system would be complete without providing a mechanism for an I/O device to send interrupts to a CPU. The I/O interrupt scheme on the AlphaServer 2100 system combines familiar technology with custom support logic to provide a new mechanism.

Electrical and architectural restrictions prohibited the interrupt control logic from being directly accessed by either the system bus or the PCI bus. As a result, the interrupt control logic is physically located on a utility bus called the XBUS. The XBUS is an 8-bit slave ISA bus placed nearby the PCI-to-EISA bridge chips.

The base technology of the I/O interrupt logic is a cascaded sequence of Intel 8259 interrupt controllers. The 8259 chip was chosen because it is a standard, accepted, and well-known controller used by the PC industry today. The use of the 8259 interrupt controller translated to low design risk as well. Although the 8259 interrupt controller is not new, its integration into a high-performance multiprocessing server, without incurring undue performance degradation, required some novel thinking.

The integration of the 8259 interrupt controller into the AlphaServer 2100 system presented two considerable problems. First, the designers had to satisfy the 8259 interface requirements in a way that would have a minimal impact on the performance of the interrupt-servicing CPU. The 8259 requires two consecutive special-acknowledge cycles before it will present the interrupt vector. To resolve this problem, we designed a set of handshaking IACK programmable array logic (PAL) devices. These PALs enhance the functions of the 8259 controllers as XBUS slaves. The interrupt-servicing CPU performs only a single read to a designated address that is decoded to the XBUS. The IACK-control PALs decode this read and then generate the special, double-acknowledge cycles required to access the vector. The PAL logic also deasserts CHRDY, a ready signal to the ISA bus, so that the cycle has ample time to proceed without causing a conformance error for a standard ISA slave cycle. When the double acknowledge is complete and the vector is guaranteed to be driven on the bus, the PALs assert the CHRDY ready signal.

The second problem involved the location of the interrupt controller. As mentioned earlier, because of electrical and architectural restrictions, the interrupt controller was located on the XBUS near the PCI-to-EISA bridge chips. With the interrupt controller located on the XBUS, an interrupt-servicing CPU is required to perform a vector read that spans two I/O bus structures. For this reason and its potential effect on system performance, vector reads had to be kept to a minimum, which is not easy in a system that allows more than one CPU to service a pending interrupt request.

Since the AlphaServer 2100 system can have as many as four CPUs, all four CPUs can attempt to service the same pending interrupt request at the same time. Without special provisions, each CPU would perform a vector read of the interrupt controller only to find that the interrupt has already been serviced by another CPU. Requiring each CPU to perform a vector read of the interrupt controller on the XBUS wastes system resources, especially when each vector read spans two bus structures. Of course, this problem could be resolved by assigning only one CPU to service pending interrupts, but this would negate the advantage of having multiple CPUs in a system. To solve this problem, the T2 bridge chip on the system bus implements special "passive-release" logic that informs a CPU at the earliest possible time that the pending interrupt is being serviced by another CPU. This allows the "released" CPU to resume other, more important tasks.

The term passive release typically refers to a vector code given to an interrupt-servicing CPU during a vector read operation. The passive-release code informs the CPU that no more interrupts are pending. The special passive-release logic allows the T2 bridge chip to return the passive-release code to a servicing CPU on behalf of the interrupt controller. The T2 bridge chip performs this function to save time and bus bandwidth.

After the designers implemented all the features described above, they needed to address the problem of how to deal with all the slow, highly volatile, "off-the-shelf" parts. To integrate these components into the I/O subsystem, they invented the standard I/O module.

The Standard I/O Module

As part of the development effort of the I/O subsystem, the engineering team faced the challenge of integrating several inexpensive, low-performance, off-the-shelf, PC-oriented I/O functions (e.g., TOY clock, keyboard, mouse, speaker) into a high-performance Alpha multiprocessing system, without affecting the higher performing architectural resources. The multilevel I/O bus structure served to alleviate the performance issues, but the development of a PC-style I/O subsystem with off-the-shelf components involved inherent risk and challenge.

To reduce the risks inherent with using new and unfamiliar devices, such as the PCI-to-EISA bridge chip set, we chose to build an I/O module (called the standard I/O module) that plugs into the AlphaServer 2100 system backplane and contains the PCI-to-EISA bridge, associated control logic, controllers for mouse, keyboard, printer, and floppy drive as well as the integral Ethernet and SCSI controllers. Without this plug-in module, fixing any problems with the PCI-to-EISA bridge chip set or any of the supporting logic would have required a backplane upgrade, which is a costly and time-consuming effort.

The standard I/O module is relatively small, inexpensive both to manufacture and to modify, and easily accessible as a field replaceable unit (FRU). As shown in Figure 3, the standard I/O module contains the following logic:

- PCI-to-Ethernet controller chip
- PCI-to-SCSI controller chip
- PCI-to-EISA bridge chips
- Real-time clock speaker control
- 8-KB, nonvolatile, EISA-configuration, random-access memory (RAM)
- 1-MB BIOS flash ROM
- Keyboard and mouse control
- Parallel port
- FDC floppy controller
- Two serial ports
- I^2C support: controller, expander, and ROM
- Intel 8259 interrupt controllers
- Ethernet station address ROM
- Reset and sysevent logic
- Fan speed monitor
- Remote fault management connector
- External PCI subarbiter
- 3.3-volt and − 5.0-volt generation

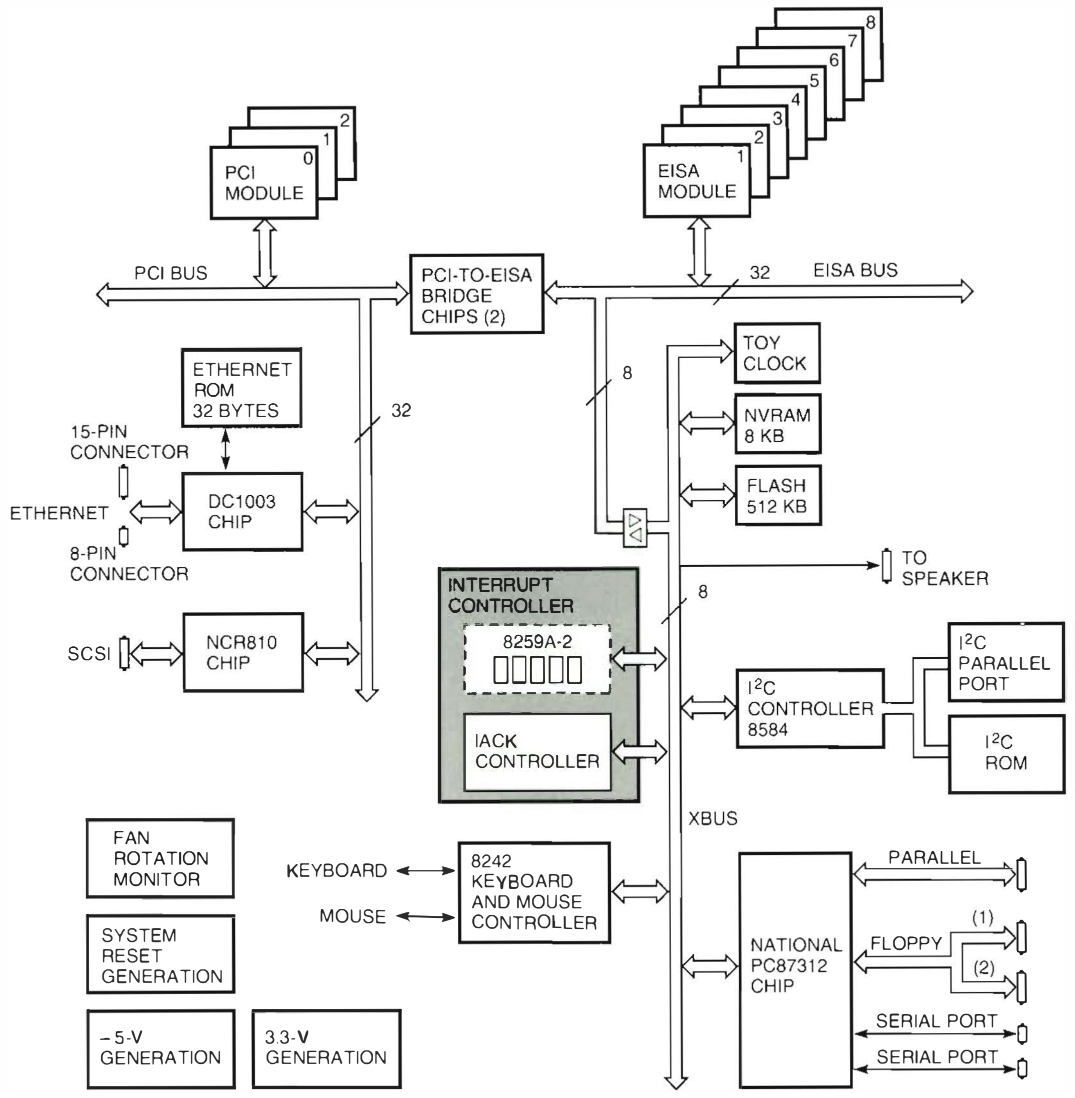


Figure 3 The Standard I/O Module

For the most part, all these functions were generated by using integrated, off-the-shelf components at commodity pricing. Solutions known to work on other products were used as often as possible. The flash memory resides on the EISA memory bus and is controlled by the PCI-to-EISA bridge chip. A simple multiplexing scheme with minimal hardware enabled the server to address more locations than the bridge chip allowed, as much as a full 1 MB of BIOS ROM. The National PC87312, which provides the serial and parallel port control logic, and the floppy disk controller reside directly on the ISA bus. The rest of the devices are located on the XBUS (an 8-bit buffered slave ISA bus), with control managed by the PCI-to-EISA bridge chips.

In addition, the common PC functions are located at typical PC addresses to ease their integration and access by software. As expected, hardware changes were required to the standard I/O module during its hardware development cycle. However, the standard I/O module, which takes only minutes to replace, provided an easy and efficient method of integrating hardware changes into the AlphaServer 2100 system. We expect the usefulness of the standard I/O module to continue and hope that it will provide an easy and inexpensive repair process.

Summary

The I/O subsystem on the AlphaServer 2100 system contains a two-level hierarchical bus structure consisting of a high-performance PCI bus connected to a secondary EISA bus. The PCI bus is connected to the AlphaServer 2100 system's multiprocessing system bus through the T2 bridge chip. The secondary I/O bus is connected to the PCI bus through a standard bridge chip set. The I/O subsystem demonstrated sufficient flexibility to become the I/O interface for the small pedestal AlphaServer 2000 and the rackmountable version of the AlphaServer 2100 products.

Acknowledgments

The AlphaServer 2100 I/O would not be what it is today, without the dedicated, focused efforts of several people. Although not complete, the following list gives credit to those who were the most deeply involved. Thanks to Fidelma Hayes for leading the Sable effort; to Vicky Triolo for the Sable mother board and her support of the T2 bridge chip effort; to Rachael Berman for her unflagging support of the standard I/O module; to Lee Ridlon for his much needed early conceptual contributions; to Stephen Shirron for driving Sable software I/O issues; to John Bridge for cleaning up the second-pass T2; and to Tom Hunt and Paul Rotker for their contributions to the first-pass T2.

References

1. F. Hayes, "Design of the AlphaServer Multiprocessor Server Systems," *Digital Technical Journal,* vol. 6, no. 3 (Summer 1994, this issue): 8-19.

2. *PCI Local Bus Specification, Revision 2.0* (Hillsboro, OR: PCI Special Interest Group, Order No. 281446-001, April 1993).

3. *82420/82430 PCIset ISA and EISA Bridges* (Santa Clara, CA: Intel Corporation, 1993).

4. E. Solari, *ISA and EISA, Theory and Operation* (San Diego, CA: Annabooks, 1992).

Jeffrey M. Denham
Paula Long
James A. Woodward

DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation

The primary goal for an operating system in a symmetric multiprocessing (SMP) implementation is to convert the additional computing power provided to the system, as processors are added, into improved system performance without compromising system quality. The DEC OSF/1 version 3.0 operating system uses a number of techniques to achieve this goal. The techniques include algorithmic enhancements to improve parallelism within the kernel and additional lock-based synchronization to protect global system state. Synchronization primitives include spin locks and blocking locks. An optional locking hierarchy was imposed to detect latent symmetric multiprocessor synchronization issues. Enhancements to the kernel scheduler improve cache usage by enabling soft affinity of threads to the processor on which the thread last ran; a load-balancing algorithm keeps the number of runnable threads spread evenly across the available processors. A highly scalable and stable SMP implementation resulted from the project.

The DEC OSF/1 operating system is a Digital product based in part on the Open Software Foundation's OSF/1 operating system.[1] One major goal of the DEC OSF/1 version 3.0 project was to provide a leadership multiprocessing implementation of the UNIX operating system for Alpha server systems, such as the Digital AlphaServer 2100 product. This paper describes the goals and development of this operating system feature for the version 3.0 release.

The DEC OSF/1 Version 3.0 Multiprocessing Project

Multiprocessing platforms like the AlphaServer 2100 product provide a cost-effective means of increasing the computing power of a server. Additional computing capacity can be obtained at a potentially significant cost advantage by simply adding CPU modules to the system rather than by adding a new system to a more loosely coupled network-server arrangement. An effective execution of this server-scaling strategy requires significant cooperation between the hardware and software components of the system. The hardware must provide symmetrical (i.e., equal) access to system resources, such as memory and I/O, for all processors; the operating system software must provide for enough parallelism in its major subsystems to allow applications to take advantage of the additional CPUs in the system. That is, the operating system cost of multiprocessing must be kept low enough to enable most of an additional CPU's computing power to be used by applications rather than by the operating system's efforts to synchronize simultaneous access to shared memory by multiple processors.

Regarding hardware, the AlphaServer 2100 product and the other Alpha multiprocessing platforms provide the shared memory and symmetric access to the system and I/O buses desired by the operating system designers.[2] The design allows all CPUs to share a single copy of the operating system in memory. The hardware also has a load-locked/store-conditional instruction sequence, which provides both a mechanism for atomic updates to shared memory by a single processor and an interprocessor interrupt mechanism.

Given these hardware features, operating system software developers have a great deal of freedom in developing a multiprocessing strategy. The approach used in DEC OSF/1 version 3.0 is called symmetric multiprocessing (SMP), in which all processors can participate fully in the execution of operating system code. This symmetric design contrasts with asymmetric multiprocessing (ASMP), in which all operating system code must be executed on a single designated "master" processor. Such an a valuable product feature and was a preview of the effort that would be required to adapt the OSF/1 code for the DEC 2000, 4000, and 7000 multiprocessing platforms. Supporting separate preemptive kernels for three versions prior to DEC OSF/1 version 3.0, combined with the developers' experience on other multiprocessing systems (including ULTRIX version 4 and an advanced development project using MIPS multiprocessing platforms), uncovered the following challenges and problems that the team had to overcome to produce a competitive multiprocessing product:

- Supporting two complete sets of kernel binary objects—base and real-time—was burdensome for the operating system engineers and awkward for third-party developers. Therefore, the DEC OSF/1 multiprocessing product team had to strive to ship a single, unified set of kernel binaries. This set should encompass the full range of real-time features, including preemption and POSIX fixed-priority scheduling. For that to be practical, the resulting multiprocessing kernel would have to perform as well on a uniprocessor as the non-SMP kernel.

- Diagnosing locking problems in the preemptive kernel was expensive in developer time. The process required painstaking inspection of the simple-locking source code, which is often disguised in subsystem-specific macros. Locking or unlocking a spin lock multiple times or not unlocking it at all (usually in code loops) would disable preemption well beyond the end of a critical section or enable it before the end. A coherent locking architecture with automated debugging facilities was needed to ship a reliable product on time. The lock-debugging facility present in the original OSF/1 code was probably inadequate for the task.

- Experiments with the real-time kernel revealed unacceptable preemption latencies, especially in funneled code paths. This deficiency indicated that, when moved to a multiprocessing platform, the existing kernel would fail to use additional processors effectively. That is, the kernel would not exhibit adequate parallelism to scale effectively. Clearly, major work was required to significantly increase parallelism in the kernel. This task would likely involve removing most uses of funneling, eliminating some spin locks, and adding other spin locks to create a finer granularity of locking.

Adapting the Base Operating System for Symmetric Multiprocessing

Making the leap from a preemptive uniprocessor kernel to an effective SMP implementation, built from a single set of kernel binaries, required contributions from the OSF/1 version 1.2 and the DEC OSF/1 version 3.0 projects. Fundamental changes were introduced into the system to support SMP.

The basic approach planned by the SMP project team was first to bootstrap the DEC OSF/1 version 1.3 kernel on the existing Alpha multiprocessing platforms. This task was accomplished by funneling all major subsystems to a single processor while stabilizing the underpinnings of the multiprocessing system (i.e., the scheduler, the virtual memory subsystem, the virtual file system, and the hardware support) in the new environment. This approach allowed the team to make progress in understanding the scope of the effort while analyzing the multiprocessing requirements of each kernel subsystem. The in-depth analysis was necessary to identify those subsystems in the kernel that required modifications to run safely and efficiently under SMP. As each subsystem was confirmed to exhibit parallelism or was made parallel, it was unfunneled and thus freed to run on any processor. This process was iterative. If incorrectly parallelized, a subsystem will reveal itself by (1) leaving data incorrectly unprotected and thus open for corruption and (2) developing a deadlock, i.e., a situation in which each of two threads holds a spin lock required by the other thread and thus neither thread can take the lock and proceed.

The efforts at parallelizing the kernel fell into two classes of modification: lock-based synchronization to ensure multiprocessing correctness and algorithmic changes to increase the level of parallelism achieved.

Lock-based Synchronization

The code base on which the DEC OSF/1 product is built, i.e., the Open Software Foundation's OSF/1 software, provides a strong foundation for SMP. The OSF further strengthened this foundation in OSF/1 versions 1.1 and 1.2, when it corrected multiple SMP problems in the code base and parallelized (and thus unfunneled) additional subsystems. As the multiprocessing bootstrap effort continued, the team analyzed and incorporated the OSF/1 version 1.2 SMP improvements into DEC OSF/1 version 3.0. As strong as this starting point was, however, some structures in the system did not receive the appropriate level of synchronization. The team corrected these problems as they were uncovered through testing and code inspection.

The DEC OSF/1 operating system uses a combination of simple locks, complex locks, elevated SPL, and funneling to guarantee synchronized access to system resources and data structures. Simple locks, SPL, and funneling were described briefly in the earlier discussion of preemption. Complex locks, like elevated SPL, are used in both uniprocessor and multiprocessor environments. These locks are usually sleep locks—threads can block while they wait for the lock—which offer additional features, including multiple-reader/single-writer access and recursive acquisition.

An example of the use of each synchronization technique follows:

- A simple lock is used to protect the kernel's callout (timer) queue. In an SMP environment, multiple threads can update the callout queue at the same time, as each of them adds a timer entry to the queue. Each thread must obtain the callout lock before adding an entry and release the lock when done. The callout simple lock is also a good example of SPL synchronization under multiprocessing because the callout queue is scanned by the system clock ISR. Therefore, before locking the callout lock, a thread must raise the SPL to the clock's IPL. Otherwise, the thread holding the callout lock at an SPL of zero can be interrupted by the clock ISR, which will in turn attempt to take the callout lock. The result is a permanent deadlock.

- A complex lock protects the file system directory structure. A blocking lock is required because the directory lock holder must perform I/O to update the directory, which itself can block. Whenever blocking can occur while a lock is held, a complex lock is required.

- Funneling is used to synchronize access to the ISO 9660 CD-ROM file system.[7] The decision to funnel this file system was largely due to limitations in the DEC OSF/1 version 3.0 schedule; however, the file system is a good choice for funneling because of its generally slow operation and light usage.

To ensure adequate performance and scaling as processors are added to the system, an SMP implementation must provide for as much parallelism through the kernel as possible. The granularity of locks placed in the system has a major impact on the amount of parallelism obtained.

During multiprocessing development, locking strategies were designed to

- Reduce the total number of locks per subsystem
- Reduce the number of locks taken per subsystem operation
- Improve the level of parallelism throughout the kernel

At times, these goals clashed: enhancing parallelism usually involves adding a lock to some structure or code path. This outcome conflicts with the goal of reducing lock counts. Consequently, in practice, the process of successfully parallelizing a subsystem involves striking a balance between lock reduction and the resulting increase in lock granularity. Often, benchmarking different approaches is required to fine-tune this balance.

Several general trends were uncovered during lock analysis and tuning. In some cases locks were removed because they were not needed; they were the products of overzealous synchronization. For example, a structure that is private to a thread may require no locking at all. Moreover, a data element that is read atomically needs no locking. An example of lock removal is the gettimeofday( ) system call, which is used frequently by DBMS servers. The system call simply reads the system time, a 64-bit quantity, and copies it to a buffer provided by the caller. The original OSF/1 system call, running on a 32-bit architecture, had to take a simple lock before reading the time to guarantee a consistent value. On the Alpha architecture, the system call can read the entire 64-bit time value atomically. Removing the lock resulted in a 40 percent speedup.

In other cases, analyzing how structures are used revealed that no locking was needed. For example, an I/O control block called the buf structure was being locked in several device drivers while the block was in a state that allowed only the device driver to access it. Removing these unnecessary locks saved one complex and one simple locking sequence per I/O operation in these drivers.

Another effective optimization involved postponing locking until a thread determined that it had actual work to do. This technique was used successfully in a routine frequently called in a transaction processing benchmark. The routine, which was locking structures in anticipation of following a rarely used code path, was modified to lock only when the uncommon code path was needed. This optimization significantly reduced lock overhead.

To improve parallelism across the system, the DEC OSF/1 SMP development team modified the lock strategies in numerous other cases.

Algorithm Changes

In some instances, the effective migration of a subsystem to the multiprocessing environment required significant reworking of its fundamental algorithms. This section presents three examples of this work. The first example involves the rework of the process management subsystem; the second example is a new technique for a thread to refer to its own state; and the third example deals with enhancements in translation buffer coherency or "shootdown."

Managing Processes and Process State Early versions of the DEC OSF/1 software maintained a set of systemwide process lists, most notably proc (static proc structure array), allproc (active process list), and zomproc (zombie process list). These lists tend to be fairly long and are normally traversed sequentially. Operations involving access to these lists include process-creation time (fork( )), signal posting, and process termination. The original OSF/1 code protected these process lists and the individual proc structures themselves by means of funneling. This meant that virtually every system call that involved process state, such as exit( ), wait( ), ptrace( ), and sigaction( ), was also forced into a single funnel. Experience with real-time preemption indicated that this approach would exact excessive multiprocessing costs. Although it is possible to protect these lists with locks, the development team decided that this basic portion of the kernel must be optimized for maximum multiprocessing performance. The OSF also recognized the need for optimization; they addressed the problem in OSF/1 version 1.2 by adopting a redesign of the process management developed for their Multimax systems by Encore Computer Corporation. The DEC OSF/1 team adopted and enhanced this design for handling process lists, process management system calls, and signal processing.

The redesign replaces the statically sized array of proc structures with an array of smaller process identification (PID) entry structures. Each PID entry structure potentially points to a dynamically allocated proc structure. Under this new scheme, finding the proc structure associated with a user PID has been reduced to hashing the PID value to an index into the PID entry array. The process state associated with that PID (active, zombie, or nonexistent) is maintained in the PID entry structure. This allows process structures to be allocated dynamically, as needed, rather than statically at boot time, as before. Simple locks are also added to the process structure to allow multiple threads in the process to perform process management system calls and signal handling concurrently. These changes allowed process management funneling to be removed entirely, which significantly improved the degree of parallelism in the process management subsystem.

Accessing Current Thread State One critical design choice in implementing SMP on the DEC OSF/1 system concerned how to access the state of the currently running thread. This state includes the current thread's process, task, and virtual memory structures, and the so-called uarea, which contains the pageable UNIX state. Access to this state, which threads require frequently as they run in kernel context, must have low overhead. Further, because the DEC OSF/1 operating system supports kernel-mode preemption, the method for accessing the current thread's state must work even if a context switch to another CPU occurs during the access operation.

The original OSF/1 code used arrays indexed by the CPU number to look up the state of a running thread. One of these arrays was the U_ADDRESS array, which was used to access the currently active uarea. The U_ADDRESS array was loaded at context switch time and accessed while the thread executed. Before the advent of multiprocessing, the CPU number was a compile-time constant, so that thread-state lookup involved simply reading a global variable to form the pointer to the data. Adding multiprocessing support meant changing the CPU number from a constant to the result of the WHAMI ("Who am I?") PALcode call to get the current CPU number. (PALcode is the operating-system-specific privileged architecture library that provides control over interrupts, exceptions, context switching, etc.[8])

Using such global arrays for accessing the current thread's state presented three shortcomings:

1. The WHAMI PALcode call added a minimum overhead of 21 machine cycles on the AlphaServer 2100 server, not including further overhead due to cache misses or instruction stream stalls. The multiprocessing team felt that this was too large a performance price to pay.

2. Allowing multiple CPUs to write sequential pointers caused cache thrashing and extra overhead during context switching.

3. Indexing by CPU number was not a safe practice when kernel-mode preemption is enabled. A thread could switch processors in the middle of an array access, and the wrong pointer would be fetched. Providing additional locking to prevent this had unacceptable performance implications because the operation is so common.

These problems convinced the team that a new algorithm was required for accessing the current thread's state.

The solution selected was modeled on the way the OpenVMS VAX system uses the processor interrupt stack pointer to derive the pointer to per-CPU state.[9] In the OSF/1 system, each thread has its own kernel stack. By aligning this stack on a power-of-two boundary, a simple masking of the stack pointer yields a pointer to the per-thread data, such as the process control block (PCB) and uthread structure. Any data item in the per-thread area can be accessed with the following code sequence:

```
lda r16, MASK          # Get mask value
bic sp, r16, r0        # Mask stack pointer to
                         point to stack base
ldq rx, OFFSET(r0)     # Add offset to base
                         and fetch item
```

Accessing thread state using the kernel stack pointer solves all three problems with CPU-number-based indexing. First, this technique has very low overhead; accessing the current thread's data involves only a simple masking operation and a read operation. Second, using the kernel stack pointer incurs no extra overhead during context switching because the pointer has to be loaded for other uses. Third, because thread stack areas are pages, no cache conflicts exist between threads running on different processors. Finally, the data access can be preempted at any point, and the correct pointer is still fetched. No processor-dependent state is used to access the current thread state.

Interprocessor Translation Lookaside Buffer Shootdown Alpha processors employ translation lookaside buffers (TLBs) to speed up the translation of physical-to-virtual mappings. The TLB caches page table entries (PTEs) that contain virtual-to-physical address mappings and access control information. Unlike data cache coherency, which the hardware maintains, TLB cache coherency is a task of the software. The DEC OSF/1 system uses an enhanced version of the TLB shootdown algorithm developed for the Mach kernel to maintain TLB coherency.[10] First, a modification to the original shootdown algorithm was needed to implement the Alpha architecture's address space numbers (ASNs). Second, a synchronization feature of the original algorithm was removed entirely to enhance shootdown performance. This feature provided synchronization for architectures in which the hardware can modify PTEs, such as the VAX platform; the added protection is unnecessary for the Alpha architecture.

The final shootdown algorithm is as follows. The physical map (PMAP) is the software structure that holds the virtual-to-physical mapping information. Each task within the system has a PMAP; operating system mappings have a special kernel PMAP. Each PMAP contains a list of processors currently using the associated address space. To initiate a virtual-to-physical translation change, a processor (the initiator) first locks the PMAP to prevent any other threads from modifying it. Next, the initiator updates the PTE mapping in memory and flushes the local TLB. The processor then sends an interprocessor interrupt to all other processors (the responders) that are currently active in the same address space. Each responder needs to acknowledge the initiator and invalidate its own mapping. Once all responders are accounted for, the initiator is free to continue with the knowledge that all TLBs are coherent on the system. The initiator marks nonactive processors' ASNs inactive, spins while it waits for other processors to check in, and then unlocks the PMAP. Figure 1 shows this final TLB shootdown algorithm as it progresses from the initiating processor to the potential responding processors.

Developing the Lock Package

Key to meeting the performance and reliability goals for the multiprocessing portion of the DEC OSF/1 version 3.0 release was the development of a lock package with the following characteristics:

- Low execution and memory overhead
- Flexible support for both uniprocessor and multiprocessor platforms, with and without real-time preemption
- Automated debugging facilities to detect incorrect locking practices at run time

Initiator:	Responders:
Lock the PMAP. Update the translation map (PTE). Invalidate the processor TLB entry. Send an interprocessor interrupt to all processors that are using the PMAP.	
	Acknowledge the shootdown. Invalidate the processor TLB entry. Return from the interrupt.
Mark the nonactive processors' ASNs inactive. Spin while it waits for other processors to check in. Unlock the PMAP.	

Figure 1 Translation Lookaside Buffer Shootdown Algorithm

- Statistical facilities to track the number of locks used, how many times a lock is taken, and how long threads wait to obtain locks

Of course, the overall role of the lock package is to provide a set of synchronization primitives, that is, the simple and complex locks described in earlier sections. To support kernel-mode thread preemption, DEC OSF/1 version 1.0 had extended the lock package originally delivered with OSF/1 version 1.0. Early in the DEC OSF/1 version 3.0 project, the development team extended the package again to optimize its performance and to add the desired debugging and statistical features.

As previously noted, a major goal for DEC OSF/1 version 3.0 was to ship a single version of its kernel objects, instead of the base and real-time sets of previous releases. Therefore, simple locks would have to be compiled into the kernel, even for kernels that would run only on uniprocessor systems. Achieving this goal required minimizing the size of the lock structure; it would be unacceptable to have hundreds of kilobytes (KB) of memory dedicated to lock structures in systems that did not use such structures. Further, the simple lock and unlock invocations required by the multiprocessing code would have to be present for all platforms, which would raise serious performance issues for uniprocessor systems. In fact, in the original OSF/1 lock package, the CPU overhead cost of compiling in the lock code was between 1 and 20 percent. Compute-intensive benchmarks showed the cost to be less than 5 percent, but the cost for multiuser benchmarks was greater than 10 percent, which represents an unacceptable performance degradation. To meet the goal of a single set of binaries, the development team had to enhance the lock package to be configurable at boot time. That is, the package needed to be able to tailor itself to fit the configuration and real-time requirements of the platform on which it would run.

The lock package supplied by the OSF/1 system was further deficient in that it did not support error checking when locks were asserted. This deficiency left developers open to the most common tormentor of concurrent programmers, i.e., deadlocks. Without error checking, potential system hangs caused by locks being asserted in the wrong order could go undetected for years and be difficult to debug. A formal locking order or hierarchy for all locks in the system had to be established, and the lock package needed the ability to check the hierarchy on each lock taken.

These needs were met by introducing the notion of lock mode to the lock package. Developers defined the following five modes and associated roles:

- Mode 0: No lock operations; for production uniprocessor systems
- Mode 1: Lock counting only to manage kernel preemption; for production real-time uniprocessor systems
- Mode 2: Locking without kernel preemption; for production multiprocessing systems
- Mode 3: Locking with kernel preemption; for production real-time multiprocessing systems
- Mode 4: Full lock debugging with or without preemption; for any development system

The default uniprocessor lock mode is 0; the multiprocessing default is lock mode 2. Both selections favor non-real-time production systems. The system's lock mode, however, can be selected at boot time by a number of mechanisms. Lock modes are implemented through a dynamic lock configuration scheme that essentially installs the appropriate set of lock primitives for the selected lock mode. Installation is realized by patching the compiled-in function calls, such as simple_lock( ), to dispatch to the corresponding lock primitive for the selected lock mode. This technique avoids the overhead of dispatching indirectly to different sets of lock primitives for each call, based on the lock mode. The compiled-in lock function calls to the lock package are all entry points that branch to a call-patching routine called simple_lock_patch( ). This routine changes the calling machine instruction to be patched out (for lock mode 0) or to branch to the corresponding primitive in the appropriate set of actual primitives, and then branches there (for lock modes 1 through 4). Thus, the overhead for dynamically switching between the versions of simple lock primitives occurs only once for each code path. In the case of lock mode 0, calls to simple lock primitives are "back patched" out. Under this model, uniprocessor systems pay a one-time cost to invoke the simple lock primitives, after which the expense of executing a lock primitive is reduced to executing a few no-op instructions where the code for the lock call once resided.

To address memory consumption issues and to provide better system debug capabilities, the developers reorganized the lock data structures around the concept of the lockinfo structure. This structure is an encapsulation of the lock's ordering (hierarchical relationship) with surrounding locks and its minimum SPL requirement. Lock debugging information and the lock statistics were decoupled from the lock structures themselves. To facilitate the expression of a lock hierarchy, the developers introduced the concept of classes and instances. A lock class is a grouping of locks of the same type. For example, the process structure lock constitutes a lock class. A lock instance is a particular lock of a given class. For example, one process structure simple lock is an instance of the class process structure lock. Error checking and statistics-gathering are performed on a lock-class basis and only in lock mode 4.

Decoupling the lock debugging information from the lock itself significantly reduced the sizes of the simple and complex lock structures to 8 and 32 bytes, respectively. Embedded in both structures is a 16-bit index into the lockinfo structure table for that particular lock class. The lockinfo structure is dynamically created at system startup in lock mode 4. All classes in the system are assigned a relative position in a single unified lock hierarchy. A lock class's position in the lockinfo table is also its position in the lock hierarchy; that is, locks must be taken in the order in which they appear in the table. Lock statistics are also maintained on a per-class basis with separate entries for each processor. Keeping lock statistics per processor and separating this information by cache blocks eliminates the need to synchronize lock-primitive access to the statistics. This design, which is illustrated in Figure 2, prevents negative cache effects that could result from sharing this data.

Once this powerful lock package was operational, developers analyzed the lock design of their kernel subsystems and attempted to place the locks used into classes in the overall system lock hierarchy. The position of a class depends on the order in which its locks are taken and released in relation to other locks in the same code path and in the system. At times, this static lock analysis revealed problems in existing lock protocols, in which locks were taken in varying orders at different points in the code. Clearly, the lock protocol needed to be reworked to produce a consistent order that could be codified in the hierarchy. Thus, the exercise of producing an overall lock hierarchy resulted in

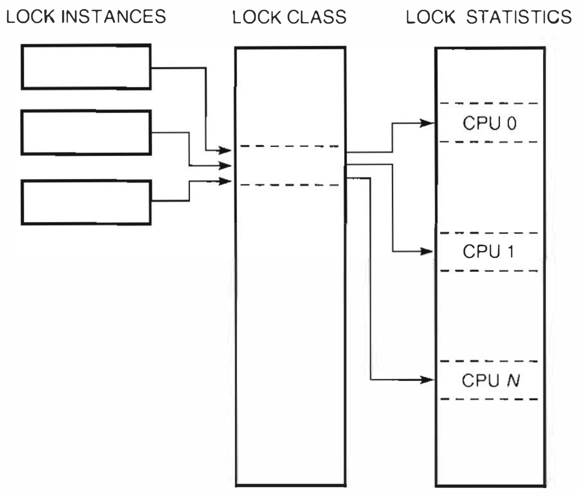


Figure 2 Lock Structure

a significant cleanup of the original multiprocessing code base. To add a new lock to the system, a developer would have to determine the hierarchical position for the new lock class and the minimum SPL at which the lock must be taken.

Running the system in lock mode 4 and exercising code paths of interest provided developers with immediate feedback on their lock protocols. Using the hierarchy and SPL information stored in the run-time lockinfo table, the lock primitives aggressively check for a variety of locking errors, which include the following:

- Locking a lock out of hierarchical order
- Locking a simple lock at an SPL below the required minimum
- Locking a simple lock already held by the caller
- Unlocking an unlocked simple lock
- Unlocking a simple lock owned by another CPU
- Locking a complex lock with a simple lock held
- Locking a complex lock at interrupt level
- Sleeping with a simple lock held
- Locking or unlocking an uninitialized lock

Encountering any of these types of violation results in a lock fault, i.e., a system bug check that records the information required by the developer to quickly track down the lock error.

The reduction in lock sizes and the major enhancement of the lock package enabled the team to realize its goal of a single set of kernel binaries. Benchmarks that compare a pure uniprocessor kernel and a kernel in lock mode 0 that are both running on the same hardware show a less than 3 percent difference in performance, a cost considered by the team to be well worth the many advantages to returning to a unified kernel. Moreover, the debugging capabilities of the lock package with its hierarchical scheme streamlined the process of lock analysis and provided precise and immediate feedback as developers adapted their subsystems to the multiprocessing environment.

Adapting the Scheduler for Multiprocessing

The normal scheduling behavior, i.e., policy, of the OSF/1 system is traditional UNIX time-sharing. The system time-slices processes based on a time quantum and adjusts process priorities to favor interactive jobs over compute-intensive jobs. To support the POSIX real-time standard, the DEC OSF/1 system incorporates two additional fixed-priority scheduling policies: first in, first out (POLICY_FIFO) and round robin (POLICY_RR).

A time-share thread's priority degrades with CPU usage; the more recent the thread's CPU usage, the more its priority degrades. (Note that OSF/1 scheduling entities are threads rather than processes.) In contrast, a fixed-priority thread never suffers priority degradation. Instead, a POLICY_RR thread runs until it blocks voluntarily, is preempted by a higher-priority thread, or exhausts a quantum (and even then, the round robin scheduling applies only to threads of equal priority). A POLICY_FIFO thread has no scheduling quantum; it runs until it blocks or is preempted. These specialized policies are used by real-time applications and by threads created and managed by the kernel. Examples of these kernel threads include the swapper and paging threads, device driver threads, and network protocol handlers. A feature called thread binding, or hard affinity, was added to DEC OSF/1 version 3.0. Binding allows a user or the kernel to force a thread to run only on a specified processor. Binding supports the funneling feature used by unparallelized code and the bind_to_cpu( ) system call.

The goal of a multiprocessing operating system in scheduling threads is to run the top *N* priority threads on *N* processors at any given time. A simple way to accomplish this would be to schedule threads that are not bound to a CPU in a single, global run queue and schedule bound threads in a run queue local to its bound processor. When a processor reschedules, it would select the highest-priority thread available in the local or the global run queue.

Scheduling threads out of a global run queue is highly effective at keeping the *N* highest-priority threads running; however, two problems arise with this approach:

1. A single run queue leads to contention between processors that are attempting to reschedule, as they race to lock the run queue and remove the highest-priority thread.
2. Scheduling with a global run queue does not take advantage of the cache state that a thread builds on the CPU where it last ran. A thread that migrates to a different processor must reload its state into the new processor's cache. This can substantially degrade performance.

To help preserve cache state and reduce wasteful global run queue contention, the developers enhanced the multiprocessing scheduler by adding two new scheduling models: a soft-affinity scheduling model for time-share threads and a last-processor-preference model for fixed-priority threads. Under these models, each processor schedules time-share and bound threads in its local run queue, and it schedules unbound fixed-priority threads out of a global run queue.

Fixed-priority threads scheduled from a global run queue are able to run as soon as possible. This behavior is required for high-priority tasks like kernel threads and real-time applications. The last-processor-preference model gives a fixed-priority thread a preference for running on the processor where it last ran; if that processor is busy, the thread runs on the next available processor. Each time-share thread is softly bound to the processor on which it last ran; that is, the thread shows an affinity for that processor. Unlike funneling or user binding, which support hard (mandatory) affinity, soft affinity allows a thread to run elsewhere if it is advantageous, i.e., if such activity balances the load. Otherwise, the softly bound thread tries to return to the processor where it last ran and where its recent cache state may still reside.

Under load, however, a soft affinity model used alone can degenerate to a state where one processor builds up a large queue of threads, leaving the other processors with little to do and thus diminishing the performance of the multiprocessing system. To mitigate these side effects of soft affinity, developers paired the soft affinity feature with the ability to load-balance the runnable threads in the system. To keep the load of time-share jobs spread evenly across processors, the scheduler must periodically load-balance the system. In addition to distributing threads evenly across the local run queues in the system, this load-balancing activity must

- Cost no more in processing time than it saves
- Prevent excessive thread movement among processors
- Recognize and effectively accommodate changes in the job mix

To implement load balancing, each processor maintains a time-share load average, i.e., the average local run queue depth over the last five seconds. Each processor updates its own load average on each system clock tick. Processors also keep track of the time they spend handling interrupts and running fixed-priority threads, which are not accounted for in the local run queue depth. Taking a processor's total potential execution time for a scheduling period and subtracting from this time the interrupt-processing and fixed-priority run times yields the total time available on a processor (processor ticks available) to run time-share threads. In the worse case, a processor could be completely consumed by fixed-priority threads and/or interrupt processing and have no time to run time-share threads. In this extreme case, the scheduler should give no time-share load to that processor.

Adding the time-share load averages of all processors determines the aggregate time-share load for the system. Similarly, summing the processor ticks available yields the total time available on the system for handling time-share tasks. Using this data, the scheduler calculates the desired load for each processor once per second, as follows:

$$\text{Desired load} = \frac{\text{Processor ticks available} \times \text{System time-share load}}{\text{System ticks available}}$$

Load balancing is called for when at least one processor is above and one is below its desired load by a minimal amount. If this condition arises, then those processors under their desired loads are declared to be "out of balance." The next time an out-of-balance processor reschedules, it will try to take a thread from the local run queue of a processor that is above its desired load ("thread stealing"). A processor can declare itself back in balance when its current load is above its desired load or when there are no eligible threads to steal. Figure 3 shows a simplified load-balancing scenario, in which a processor below its desired load steals a thread from a processor above its desired load.

To help preserve the cache benefits of soft affinity, a thread is eligible for stealing only when it has not run on its current processor for some configurable number of clock ticks. After this time has elapsed without a thread running, the chance of it having significant cache state remaining has diminished sufficiently that the thread is more likely to benefit from migrating to another processor and running immediately than from waiting longer to run on its current processor.

To demonstrate that soft affinity with load balancing improves multiprocessing performance through cache benefits and the elimination of run queue contention, developers ran a simple test program. The program, which writes 128 KB of data, yields the processor, and then reads the same data back, was run on a four-processor DEC 7000 system.

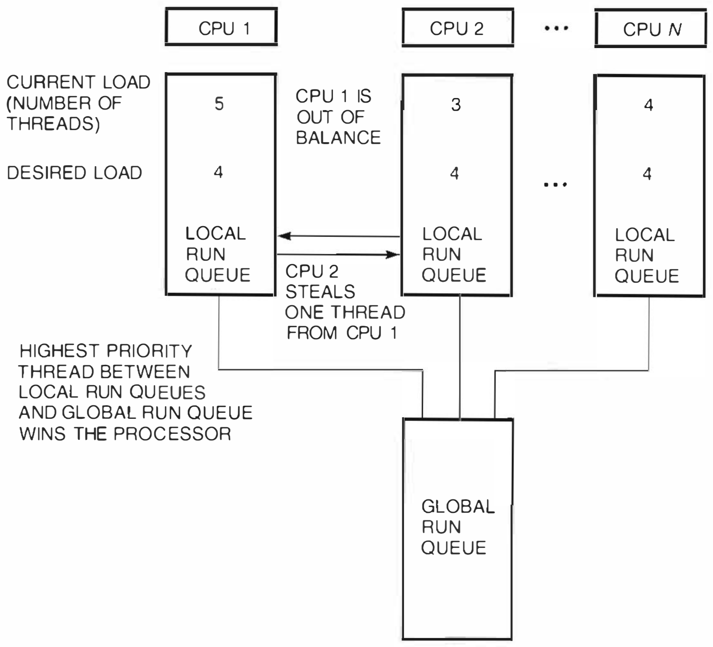

CPU 1
CPU 2
CPU N
CURRENT LOAD (NUMBER OF THREADS)
DESIRED LOAD
5
4
3
4
4
4
CPU 1 IS OUT OF BALANCE
LOCAL RUN QUEUE
CPU 2 STEALS ONE THREAD FROM CPU 1
HIGHEST PRIORITY THREAD BETWEEN LOCAL RUN QUEUES AND GLOBAL RUN QUEUE WINS THE PROCESSOR
GLOBAL RUN QUEUE

Figure 3 Load Balancing

Table 1 shows the results of running multiple versions of this program with and without soft affinity and load balancing in operation. Performance benefits appear only when multiple copies of the program begin piling up in the run queues at the 16-job level. Prior to this point, each job keeps running on the same processor, i.e., the cache it had just filled still had its data cached when the program read it back—the ideal case. At the 16-job level, the four processors must be time-shared. The jobs that are running with soft affinity now benefit significantly because they continue to run on the same processor and thus find some of their cache state preserved from when they last ran. The systems that schedule from a global run queue provide no such benefit. Jobs take longer to complete, since they are likely to run on a different processor for each time slice and find no cache state that they can reuse.

The soft affinity and load-balancing features improved many other multiuser benchmarks. For example, a transaction processing benchmark showed a 17 percent performance improvement.

Focusing on Quality

The error-checking focus of the lock package is just one example of how the DEC OSF/1 version 3.0 project focused on the quality and stability of the product. Most members of the multiprocessing team had been involved in an SMP development effort prior to their DEC OSF/1 effort. This past experience, coupled with the difficulties other vendors had experienced with their own multiprocessing implementations, reinforced the need to have a strong quality focus in the SMP project.

Developers took multiple steps to ensure that the SMP solution delivered in DEC OSF/1 version 3.0 would be production quality, including

Table 1 Benefits of Soft Affinity with Load Balancing (SA/LB)

Number of Jobs	Time with SA/LB (Seconds)	Time without SA/LB (Seconds)	Benefit from SA/LB (Percent)
1	25.9	26.0	0
4	25.9	26.0	0
16	106.5	141.9	25

- Code reviews
- Lock debugging
- In-line assertion checking
- Multithreaded test suite development for SMP qualification

The base kernel code was reviewed for multiprocessing correctness. During this review phase, checks were made to ensure that the proper level of synchronization was employed to protect global data structures. Numerous defects were uncovered during this process and corrected. Running code with lock checking enabled provided empirical evidence of the incremental improvements of the multiprocessing implementation.

Beyond code reviews and lock debugging, internal consistency checks (assertions) were coded into the kernel to verify correctness of operations at key points. Assertion checking was enabled during the development process to ensure that the kernel was functioning correctly; it was then compiled out for the production version of the kernel.

In parallel with the operating system development effort, new component tests were designed to force as much concurrency as possible through particular code paths. The core of the test suite is a thread-race library, which consists of a set of routines that can be used to construct multithreaded system-call exercisers. The library provides the ability to commence multiple test instances simultaneously. The individual tests are then combined to form focused subsystem tests and systemwide tests. These tests have been used to uncover multiple race conditions in the operating system.

The UNIX development organization had a four-processor DEC 7000 system deployed in its development environment for more than 7 months prior to releasing the SMP product. This system has been extremely stable, with few complaints from the user community. Extensive internal and external field testing produced similar results.

Measuring Multiprocessing Performance Outcomes

The major functionality delivered with SMP is improved performance through application concurrency. The goal of the SMP project was to provide leadership performance in the areas of compute and data servers. To gauge success in this effort, several industry-standard benchmarks were utilized. These benchmarks include SPECrate_INT92, SPECrate_FP92, and AIM Suite III.

SMP performance is measured in terms of the incremental performance gained as processors are added to the system. Adding processors by no means guarantees increased system performance. Systems that have I/O or memory limitations rarely benefit from introducing additional CPUs. Systems that are compute bound tend to have the largest potential for gain from additional processors. Note that large, monolithic applications tend to see little performance improvement as processors are added because such applications employ little concurrency in their designs.

Performance tuning for SMP was an iterative process that can be characterized as follows:

1. Collect and analyze performance data.
   - CPU utilization across the processors
   - Lock statistics
   - I/O rates
   - Context switch rates
   - Kernel profiling
2. Identify areas that require improvement.
3. Prototype changes.
4. Incorporate changes that demonstrate improvement.
5. Repeat steps 1 through 4.

In reality, the process has two stages for each benchmark. The initial phase was devoted to driving the system to become compute bound. The second phase improved code efficiencies.

Figures 4 and 5 show that, as expected, the SPECrate_INT92 and SPECrate_FP92 benchmarks scale almost linearly. Both of these benchmarks are compute intensive and make only nominal demands on the operating system.

AIM Suite III is a multiuser benchmark that stresses multiple components of an operating system, including the virtual memory system, the scheduler, UNIX pipes, and the I/O subsystem. Figure 6 shows AIM III results for one and four processors, with a resulting 3.27 to 4 scaling factor. This equates to a greater than 80 percent scaling factor, a figure well within the goals for this benchmark at first multiprocessing release. Efforts to produce still better results are under way.

AIM Suite III scaling appears to be gated by a single test in the AIM Suite III benchmark, i.e.,

Chandrika Kamath
Roy Ho
Dwight P. Manley

DXML: A High-performance Scientific Subroutine Library

Mathematical subroutine libraries for science and engineering applications are an important tool in high-performance computing. By identifying and optimizing frequently used, numerically intensive operations, these libraries help in reducing the cost of computation, enhancing portability, and improving productivity. The Digital eXtended Math Library is a set of public domain and Digital proprietary software that has been optimized for high performance on Alpha systems. In this paper, DXML and the issues related to library software technology are described. Specific examples illustrate how algorithms can be optimized to take advantage of the architecture of Alpha systems. Modern algorithms that effectively exploit the memory hierarchy enable DXML routines to provide substantial improvements in performance.

The Digital eXtended Math Library (DXML) is a set of mathematical subroutines, optimized for high performance on Alpha systems. These subroutines perform numerically intensive subtasks that occur frequently in scientific computing. They can therefore be used as building blocks for the optimization of various science and engineering applications in industries such as chemical, aerospace, petroleum, automotive, electronics, finance, and transportation.

In this paper, we discuss the role of mathematical software libraries, followed by an overview of the contents of the Digital eXtended Math Library. DXML includes optimized versions of both the standard BLAS and LAPACK libraries as well as libraries designed and developed by Digital for signal processing and the solution of sparse linear systems of equations. Next, we describe various aspects of library software technology, including the design and testing of DXML subroutines. Using key routines as examples, we illustrate the techniques used in the performance optimization of the library. Finally, we present data that demonstrates the performance improvement obtained through the use of DXML.

The Role of Math Libraries

Early mathematical libraries concentrated on supplementing the functionality provided by the Fortran compilers. In addition to routines such as sin and exp, which were included in the run-time math library, more complicated special functions, linear algebra algorithms, and Fourier transform algorithms were included in the software layer between the hardware and the user application.

Then, in the early 1970s, there was a concerted effort to produce high-quality numerical software, with the aim of providing end users with implementations of numerical algorithms that were stable, robust, and accurate. This led to the development of several math libraries, with the public domain LINPACK and EISPACK libraries for the solution of linear and eigen systems, setting the standards for future development of math software.[1-3]

The late 1970s and early 1980s saw the availability of advanced architectures, including vector and parallel computers, as well as high-performance workstations. This added another facet to the development of math libraries, namely, the implementation of algorithms for high efficiency on an underlying architecture.

The effort to produce mathematical software thus became a task of building bridges between numerical analysts, who devise algorithms, computer architects, who design high-performance computer systems, and computer users, who need efficient, reliable software for solving their problems. Consequently, these libraries embody expert knowledge in applied mathematics, numerical analysis, data structures, software engineering, compilers, operating systems, and computer architecture and are an important programming tool in the use of high-performance computers.

Modern superscalar RISC architectures with floating-point pipelines, such as the Alpha, have deep memory hierarchies. These include floating-point registers, multiple levels of caches, and virtual memory. The significant latency and bandwidth differences between these memory levels imply that numerical algorithms have to be restructured to make effective use of the data brought into any one level. The performance of an algorithm is also susceptible to the order in which computations are scheduled as well as the higher cost associated with some operations such as floating-point square-root and division.

The architecture of the Alpha systems and the technology of the Fortran and C compilers usually provide an efficient computing environment with adequate performance. However, there is often room for improvement, especially in engineering and science applications, where vast amounts of data are processed and repeated operations are performed on each data element. One way to achieve these improvements is through the use of optimized subroutine libraries.

The Digital eXtended Math Library is a collection of routines that performs frequently occurring, numerically intensive operations. By identifying such operations and optimizing them for high performance on Alpha systems, DXML provides several benefits to the computational scientist.

- It allows definition of functions at a sufficiently high level and therefore optimization beyond the capabilities of the compiler.
- It makes the architecture of the systems more transparent to the user.
- It improves productivity by providing easy access to highly optimized, efficient code.
- It enhances the portability of user software through the support of standard libraries and interfaces.
- It promotes good software engineering practice and avoids duplication of work by identifying and optimizing common functions across several application areas.

Overview of DXML

DXML contains almost 400 user-callable routines, optimized for Alpha systems.[5] This includes both software developed by Digital as well as the BLAS and LAPACK libraries. Most routines are available in four versions: real single precision, real double precision, complex single precision, and complex double precision.

DXML is available on both OpenVMS and DEC OSF/1 operating systems. Its routines can be called from either Fortran or C, provided the difference in array storage between these languages is taken into account. DXML is available as a shareable library, with a simple interface, enabling easy access to the routines. On DEC OSF/1 systems, DXML supports the IEEE floating-point format. On OpenVMS systems, either the IEEE floating-point format or the VAX F-float/G-float format can be selected.

DXML routines can be broadly categorized into the following four areas:

- BLAS. The Basic Linear Algebra Subroutines include the standard BLAS and Digital enhancements.
- LAPACK. The Linear Algebra PACKage includes linear and eigen-system solvers.
- Signal processing. This includes fast Fourier transforms (FFTs), convolution, and correlation.
- Sparse linear system solvers. These include direct and iterative solvers.

Of these, the signal-processing library and the sparse linear system solvers are designed, developed, and optimized by Digital. The majority of the BLAS routines and the LAPACK library are versions of the public domain standard that were optimized for the Alpha architecture. By supporting the industry standard interfaces of these libraries, DXML provides both portability of user code and high performance of the optimized software.

We next provide a brief description of the functionality provided by each subcomponent of DXML. Further details are available in the Digital eXtended Math Library Reference Manual.[5]

VLIB

The vector library consists of seven double-precision routines that perform operations such as sine, cosine, and natural logarithm, on data stored as vectors.

BLAS 1

The Basic Linear Algebra level 1 subprograms perform low-granularity operations on vectors that involve one or two vectors as input and return either a vector or a scalar as output.[6] Examples of BLAS 1 routines include dot product, index of the maximum element in a vector, and so on.

BLAS 1 Extensions (BLAS 1E)

Digital has extended the functionality of the BLAS 1 routines by including 13 similar operations. These include index of the minimum element of a vector, sum of the elements of a vector, and so on.

BLAS 1 Sparse (BLAS 1S)

DXML also includes nine routines that are sparse extensions of the BLAS 1 routines. Of these, six are from the sparse BLAS 1 standard and three are enhancements.[7] These routines operate on two vectors, one of which is sparse and stored in a compressed form. As most of the elements in a sparse vector are zero, both computational time and memory are reduced by storing and operating on only the nonzeros. BLAS 1S routines include construction of a sparse vector from the specified elements of a dense vector, dot product, and so on.

BLAS 2

The BLAS level 2 routines perform operations of a higher granularity than the level 1 routines.[8] These include matrix-vector operations such as matrix-vector product, rank-one and rank-two updates, and solutions of triangular systems of equations. Various storage schemes are supported, including general, symmetric, banded, and packed.

BLAS 3

The BLAS level 3 routines perform matrix-matrix operations, which are of a higher granularity than the BLAS 2 operations. These routines include matrix-matrix product, rank-k updates, solution of triangular systems with multiple right-hand sides, and multiplication of a matrix by a triangular matrix. Where appropriate, these operations are defined for matrices that may be general, symmetric, or triangular.[9] The functionality of the public domain BLAS 3 library has been enhanced by three additional routines for matrix addition, subtraction, and transpose.

LAPACK

DXML includes the standard Linear Algebra PACKage, LAPACK, which supersedes the LINPACK and EISPACK packages by extending the functionality, using algorithms with higher accuracy, and improving the performance through the use of the optimized BLAS library.[10] LAPACK can be used for solving many common linear algebra problems, including solution of linear systems, linear least-squares problems, eigenvalue problems, and singular value problems. Various storage schemes are supported, including general, band, tridiagonal, symmetric positive definite, and so on.

Signal Processing

The signal-processing subcomponent of DXML includes FFTs, convolutions, and correlations. A comprehensive set of Fourier transforms is provided, including

- FFTs in one, two, and three dimensions
- FFTs in forward and inverse directions
- Multiple one-dimensional transforms

There is no limit on the number of elements being transformed, though the performance is best when the data length is a power of 2. Popular storage formats for the input and output data are supported, allowing for possible symmetry in the output data and consequent reduction in the storage required. Further efficiency is provided through the use of the three-step FFT, which separates the process of setting up and deallocating the internal data structures from the actual application of the FFT. This results in significant performance gain when repeated application of FFTs is required.

The convolution and correlation routines in DXML support both periodic (circular) and nonperiodic (linear) definition. A discrete summing technique is used for calculation. Special versions of the routines allow control of output options such as the range of coefficients computed, scaling of the output, and addition of the output to an array.

All FFT, convolution, and correlation routines are available in both single and double precision and support both real and complex data.

Sparse Iterative Solvers

DXML includes a set of routines for the iterative solution of sparse linear systems of equations using preconditioned, conjugate-gradient-like methods.[11,12] A flexible user interface, based on a matrix-free formulation of the solver, allows a choice among various solvers, storage schemes, and preconditioners. This formulation permits the user to define his or her own preconditioner and/or storage scheme for the matrix. It also allows the user to store the matrix using one of the storage schemes defined by DXML and/or use the preconditioners provided. A driver routine provides a simple interface to the iterative solvers when the DXML storage schemes and preconditioners are used.

The different iterative methods provided are (1) conjugate gradient, (2) least-squares conjugate gradient, (3) biconjugate gradient, (4) conjugate-gradient squared, and (5) generalized minimum residual. Each method supports various applications of the preconditioner: left, right, split, and no preconditioning.

The matrix can be stored in the symmetric diagonal storage scheme, the unsymmetric diagonal storage scheme, or the general storage (by rows) scheme. Three preconditioners are provided for each storage scheme: diagonal, polynomial (Neumann), and incomplete LU with zero diagonals added.

A choice of four stopping criteria is provided, in addition to a user-defined stopping criterion. The iteration process can be controlled by setting various input parameters such as the maximum number of iterations, the degree of polynomial preconditioning, the level of output provided, and the tolerance for convergence. These solvers are available in real double precision only.

Sparse Skyline Solvers

The sparse skyline solver library in DXML includes a set of routines for the direct solution of a sparse linear system of equations with the matrix stored using the skyline storage scheme.[13,14] The following functions are provided.

- LDU factorization, which includes options for the evaluation of the determinant and inertia, partial factorization, statistics on the matrix, and options for handling small pivots.
- Solve, which includes multiple right-hand sides and solves systems involving either the matrix or its transpose.
- Norm evaluation, including 1-norm, infinity-norm, Frobenius norm, and the maximum absolute value of the matrix.
- Condition number estimation, which includes both the 1-norm and the infinity norm.
- Iterative refinement, including the component-wise relative backward error and the estimated forward error bound for each solution vector.
- Simple and expert drivers.

This functionality is provided for each of the following storage schemes:

- For symmetric matrices:
  - Profile-in storage mode
  - Diagonal-out storage mode
- For unsymmetric matrices:
  - Profile-in storage mode
  - Diagonal-out storage mode
  - Structurally symmetric profile-in storage mode

These solvers are available in real double precision only.

Software Considerations

As with any software effort, many software engineering issues were encountered during the design and development of DXML. Some issues were specific to math libraries such as the numerical accuracy and stability of the routines, while others were more general such as the design of a user interface, testing of the software, error checking, ease of use, and portability. We next discuss some of these key design issues in further detail.

Designing the Interface

The first task in creating a library was to decide the functionality, followed by the design of the interface. This included both the naming of the subroutines as well as the design of the parameter list. For each subcomponent in DXML, the calling sequence was designed to be consistent across all routines in that subcomponent. In the case of the BLAS and LAPACK libraries, the public domain interface was maintained to enable portability of user code.

For the routines added by Digital, the routine names were chosen to indicate the function being performed as well as the precision of the data. Furthermore, the parameter lists were chosen to provide a simple interface, yet allow flexibility for the sophisticated user. For example, the sparse solvers require various real and integer parameters. By using arrays instead of scalar variables, a more concise interface that did not vary from routine to routine was obtained. In addition, all solver routines have arguments for real and integer work arrays, even if these are not used in the code. This not only provides a uniform interface but also acts as a placeholder for work arrays, should they be required in the future.

Accuracy

The numerical accuracy of the routines in DXML is dependent on the problem size as well as the algorithm used, which may vary within a routine. Since performance optimization often changes the order in which a computation is performed, identical results between the DXML routines and the public domain BLAS and LAPACK routines may not occur. The accuracy of the results obtained is checked by ensuring that the optimized versions of the BLAS and LAPACK routines pass the public domain tests to within the specified tolerance.

Error Processing

Most of the routines in DXML trap usage errors and provide sufficient information so that the user can identify and fix the problem. The low-level, fine-grained computational routines, such as the BLAS level 1, do not provide this function because the overhead of testing and error trapping would seriously degrade the performance.

In the case of BLAS 2, BLAS 3, and LAPACK, the public domain error-reporting mechanism has been maintained. If an input argument is invalid, such as a negative value for the order of the matrix, the routine prints out an error message and stops. If a failure occurs in the course of the algorithm, such as a matrix being singular to working precision, an error flag is set and control is returned to the calling program.

The signal-processing routines report success or failure using a status function value. Further information on the error can be obtained by using a user-callable routine that prints out an error message and an error flag. The user documentation indicates the actions to be taken to recover from the error.

In the case of the sparse solvers, error is indicated by setting an error flag and printing an appropriate message if the printing option is enabled. Control is always returned to the calling program.

Testing

DXML routines are tested for correctness and accuracy using a regression test suite. This includes both test code developed by Digital, as well as the public domain test codes for BLAS and LAPACK. These codes are used not only during the implementation and performance optimization of the routines, but also during the building of the complete library from each of the subcomponents.

The test codes check each routine extensively, including checks for error exits, accuracy of the results obtained, invariance of read-only data and the correctness of all paths through the code. As the complete regression tests take over 20 hours to execute, two input data sets are used: a short one that tests each routine and can be used to make a quick check that all subcomponents compiled and built correctly, and a long data set that tests each path through a routine and is thus more exhaustive.

Many of the routines, such as the FFTs and BLAS 3, are tested using random input data. However, some routines, such as the sparse solvers, operate on specific data structures or matrices with specific properties. These have been tested using matrices generated from the finite difference discretization of partial differential equations or using the matrices in the Harwell-Boeing test suite.[15]

Another aspect to the DXML regression test package is the inclusion of a performance test gauge. This software tests the performance of key routines in each component of DXML and is used to ensure that the performance of DXML routines is not adversely affected by changes in compilers or the operating systems.

Performance Trade-offs

The design and optimization of the routines in DXML often prompted a trade-off between performance on one hand, and accuracy and generality on the other. Although every effort has been made not to sacrifice accuracy for performance, the reordering of computations during performance optimization may lead to results before optimization that are not bit-for-bit identical to the results after optimization. In other cases, performance has been sacrificed to ensure generality of a routine. For example, although the matrix-free formulation of the iterative solvers permits the use of any sparse matrix storage scheme, it could result in a slight degradation in performance due to less efficient use of the instruction cache and the inability to reuse some of the data in the registers.

Performance Optimization

DXML routines have been designed to provide high performance on the Alpha systems.[16] These routines are tailored to take advantage of the system characteristics such as the number of floating-point registers, the size of the primary and secondary data caches, and the page size. This optimization involves changes to data structures and the use of new algorithms as well as the restructuring of computation to effectively manage the memory hierarchy.

Several general techniques are used across all DXML subcomponents to improve the performance.[17] These include the following techniques:

- Unrolling loops to make better use of the floating-point pipelines
- Reusing data in registers and caches whenever possible

- Managing the data caches effectively so that the cache hit ratio is maximized
- Accessing data using stride-1 computation
- Using algorithms that exploit the memory hierarchy effectively
- Reordering computations to minimize cache and translation buffer thrashing

Although many of these optimizations are done by the compiler, occasionally, for example in the case of the skyline solver, the data structures or the implementation of the algorithm are such that they do not lend themselves to optimization by the compiler. In these cases, explicit reordering of the computations is required.

We next discuss these optimization techniques as used in specific examples. All performance data is for the DEC 3000 Model 900 system using the DEC OSF/1 version 3.0 operating system. This workstation uses the Alpha 21064A chip, running at 275 megahertz (MHz). The on-chip data and instruction caches are each 16 kilobytes (KB) in size, and the secondary cache is 2 megabytes (MB) in size.

In the next section, we compare the performance of DXML BLAS and LAPACK routines with the corresponding public domain routines. Both versions are written in standard Fortran and compiled using identical compiler options.

Optimization of BLAS 1

BLAS 1 routines operate on vector and scalar data only. As the operations and data structures are simple, there is little opportunity to use advanced data blocking and register reuse techniques. Nevertheless, as the plots in Figure 1 demonstrate, it is possible to optimize the BLAS 1 routines by careful coding that takes advantage of the data prefetch features of the Alpha 21064A chip and avoids data-path-related stalls.[16,18]

Generally, the DXML routines are 10 percent to 15 percent faster than the corresponding public domain routines. Occasionally, as in the case of DDOT for very short, cache-resident vectors, the benefits can be much greater.

The shapes of the plots in Figure 1 rather dramatically demonstrate the benefits of data caches. Each plot shows very high performance for short vectors that reside in the 16-KB, on-chip data cache, much lower performance for data vectors that reside in the 2-MB, on-board secondary data cache, and even lower performance when the vectors reside completely in memory.

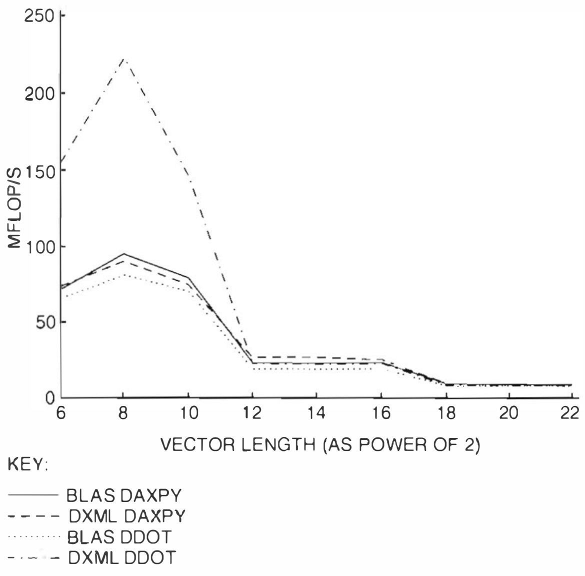


Figure 1 Performance of BLAS 1 Routines DDOT and DAXPY

Optimization of BLAS 2

BLAS 2 routines operate on matrix, vector, and scalar data. The data structures are larger and more complex than the BLAS 1 data structures and the operations more complicated. Accordingly, these routines lend themselves to more sophisticated optimization techniques.

Optimized DXML BLAS 2 routines are typically 20 percent to 100 percent faster than the public domain routines. Figure 2 illustrates this performance improvement for the matrix-vector multiply routine, DGEMV, and the triangular solve routine, DTRSV.[8]

The DXML DGEMV uses a data-blocking technique that asymptotically performs two floating-point operations for each memory access, compared to the public domain version, which performs two floating-point operations for every three memory accesses.[19] This technique is designed to minimize translation buffer and data cache misses and maximize the use of floating-point registers.[16,18,20] The same data prefetch considerations used on the BLAS 1 routines are also used on the BLAS 2 routines.

The DXML version of the DTRSV routine partitions the problem such that a small triangular solve operation is performed. The result of this solve operation is then used in a DGEMV operation to update the remainder of the vector. The process is repeated until the final triangular update completes the operation. Thus the DTRSV routine relies heavily on the optimizations used in the DGEMV routine.

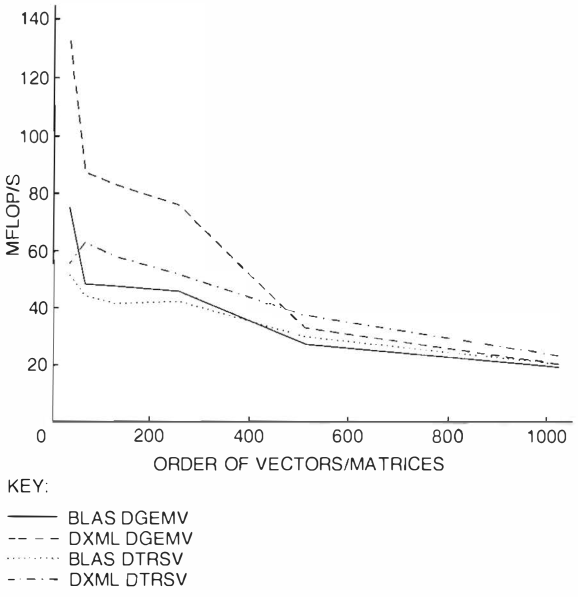


Figure 2 Performance of BLAS 2 Routines DGEMV and DTRSV

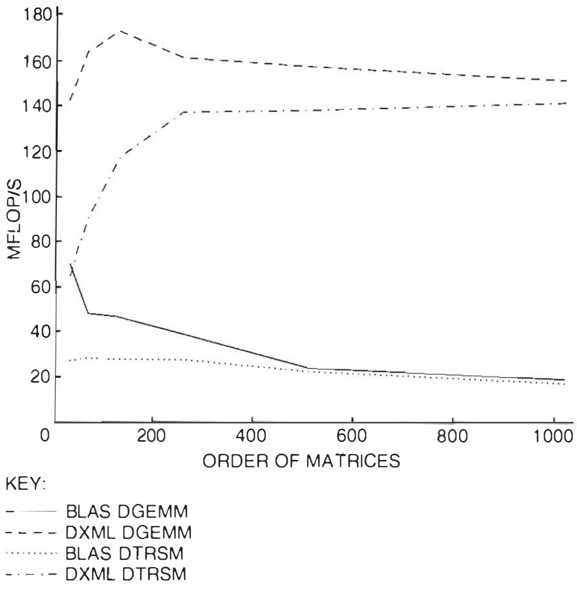


Figure 3 Performance of BLAS 3 Routines DGEMM and DTRSM

As with BLAS 1 routines, BLAS 2 routines benefit greatly from data cache. Although the effect is less dramatic for the BLAS 2 routines, Figure 2 clearly shows the three-step profile observed in Figure 1. Best performance is achieved when both matrix and vector fit in the primary cache. Performance is lower but flat over the region where the data fits on the secondary board level cache. The final performance plateau is reached when data resides entirely in memory.

Optimization of BLAS 3

BLAS 3 routines operate primarily on matrices. The operations and data structures are more complicated that those of BLAS 1 and BLAS 2 routines. Typically, BLAS 3 routines perform many computations on each data element. These routines exhibit a great deal of data reuse and thus naturally lend themselves to sophisticated optimization techniques.

DXML BLAS 3 routines are generally two to ten times faster than their public domain counterparts. The plots in Figure 3 show these performance differences for the matrix-matrix multiply routine, DGEMM, and the triangular solve routine with multiple right-hand sides, DTRSM.[9]

All performance optimization techniques used for the DXML BLAS 1 and BLAS 2 routines are used on the DXML BLAS 3 routines. In particular, data-blocking techniques are used extensively. Portions of matrices are copied to page-aligned work areas where secondary cache and translation buffer misses are eliminated and primary cache misses are absolutely minimized.

As an example, within the primary compute loop of the DXML DGEMM routine, there are no translation buffer misses, no secondary cache misses, and, on average, only one primary cache miss for every 42 floating-point operations. Performance within this key loop is also enhanced by carefully using floating-point registers so that four floating-point operations are performed for each memory read access. Much of the DXML BLAS 3 performance advantage over the public domain routines is a direct consequence of a greatly improved ratio of floating-point operations per memory access.

The DXML DTRSM routine is optimized in a manner similar to its BLAS 2 counterpart, DTRSV. A small triangular system is solved. The resulting matrix is then used by DGEMM to update the remainder of the right-hand-side matrix. Consequently, most of the DXML DTRSM performance is directly attributable to the DXML DGEMM routine. In fact, the techniques used in DGEMM pervade DXML BLAS 3 routines.

Figure 3 illustrates a key feature of DXML BLAS 3 routines. Whereas the performance of public domain routines degrades significantly as the matrices become too large to fit in caches, DXML routines are relatively insensitive to array size, shape, or orientation.[5,9] The performance of a DXML BLAS 3 routine typically reaches an asymptote and remains there regardless of problem size.

Optimization of LAPACK

The LAPACK subroutine library derives a large part of its high performance by using the optimized BLAS as building blocks.[10] The DXML version of LAPACK is largely unmodified from the public domain version. However, in the case of the factorization routine for general matrices, DGETRF, we have introduced changes to the algorithm to improve the performance on Alpha systems.

For example, while the original public domain DGETRF routine uses Crout's method to factor a matrix, the DXML version uses a left-looking method.[11] Left-looking methods make better use of the secondary cache and translation buffers than the Crout method. Furthermore, the public domain version of the DLASWP routine swaps a single matrix row across an entire matrix. This is a very bad technique for RISC machines; it causes severe cache and translation buffer thrashing. To avoid this, the DXML version of DLASWP performs all swaps within columns, which makes much better use of the caches and the translation buffer and results in a much improved performance of the DXML DGETRF routine.

The DGETRS routine was not modified. Its performance is solely attributable to use of optimized DXML routines.

Figure 4 shows the benefits of the optimizations made to DGETRF and the BLAS routines. DGETRF makes extensive use of the BLAS 3 DGEMM and DTRSM routines. The performance of DXML DGETRF improves with increasing problem size largely because DXML BLAS 3 routines do not degrade in the face of larger problems.

The plots of Figure 4 also show the performance of DGETRS when processing a single right-hand-side vector. In this case, DTRSV is the dominant BLAS routine, and the performance differences between the public domain and DXML DGETRS routines reflect the performance of the respective DTRSV routines. Finally, although not shown, we note that the performance of DXML DGETRS is much better than the public domain version when many right-hand sides are used and DTRSM becomes the dominant BLAS routine.

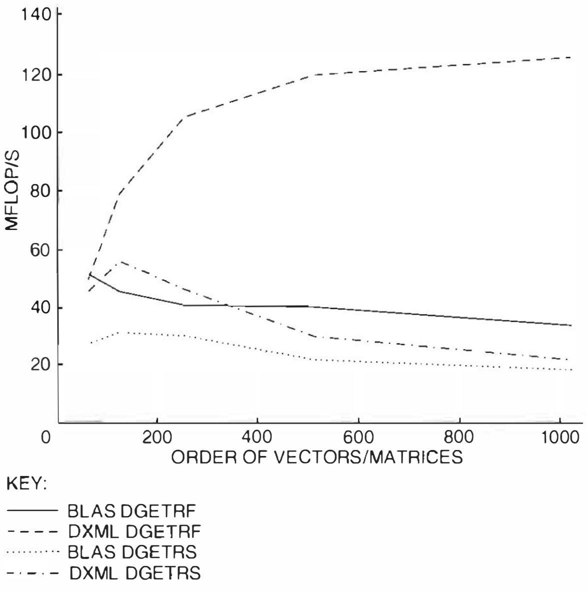


Figure 4 Performance of LAPACK Routines DGETRF and DGETRS (LDA = N + 1)

Optimization of the Signal-processing Routines

We illustrate the techniques used in optimizing the signal-processing routines using the one-dimensional, power-of-2, complex FFT.[21] The algorithm used is a version of Stockham's autosorting algorithm, which was originally designed for vector computers but works well, with a few modifications, on a RISC architecture such as Alpha.[22,23]

The main advantage in using an autosorting algorithm is that it avoids the initial bit-reversal permutation stage characteristic of the Cooley-Tukey algorithm or the Sande-Tukey algorithm. This stage is implemented by either precalculating and loading the permutation indices or calculating them on-the-fly. In either case, substantial amounts of integer multiplications are needed. By avoiding these multiplications, the autosorting algorithm provides better performance on Alpha systems.

This algorithm does have the disadvantage that it cannot be done in-place, resulting in the use of a temporary work space, which makes more demands on the cache than an algorithm that can be done in-place. However, this disadvantage is more than offset by the avoidance of the bit-reversal stage.

The implementation of the FFT on the Alpha makes effective use of the hierarchical memory of the system, specifically, the 31 usable floating-point registers, which are at the lowest, and therefore the fastest, level of this hierarchy. These registers are utilized as much as possible, and any data brought into these registers is reused to the extent possible. To accomplish this, the FFT routines implement the largest radices possible for all stages of the power-of-2 FFT calculation. Radix-8 was used for all stages except the first, utilizing 16 registers for the data and 14 for the twiddle factors.[21] For the first stage, as all twiddle factors are 1, radix-16 was used.

Figure 5 illustrates the performance of this algorithm for various sizes. Although the performance is very good for small data sizes that fit into the primary, 16-KB data cache, it drops off quickly as the data exceeds the primary cache. To remedy this, a blocking algorithm was used to better utilize the primary cache.

The blocking algorithm, which was developed for computers with hierarchical memory systems, decomposes a large FFT into two sets of smaller FFTs.[24] The algorithm is implemented using four steps:

1. Compute $N1$ sets of FFTs of size $N2$.
2. Apply twiddle factors.
3. Compute $N2$ sets of FFTs of size $N1$.
4. Transpose the $N1$ by $N2$ matrix into an $N2$ by $N1$ matrix.

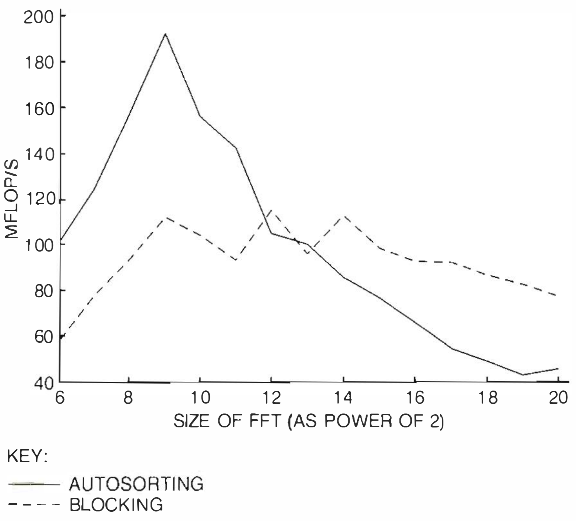


Figure 5 Performance of 1-D Complex FFT

In the above, $N = N1 \times N2$. Steps (1) and (3) use the autosorting algorithm for small sizes. In step (2), instead of precomputing all N twiddle factors, a table of selected twiddle factors is computed and the rest calculated using trigonometric identities.

Figure 5 compares the performance of the blocking algorithm with the autosorting algorithm. Due to the added cost of steps (2) and (4), the maximum computation speed for the blocking algorithm (115 million floating-point operations per second [Mflops] at $N=2^{12}$) is lower than the maximum computation speed of the autosorting algorithm (192 Mflops at $N=2^9$). The crossover point between the two algorithms is at a size of approximately 2K, with the autosorting algorithm performing better at smaller sizes. Based on the length of the FFT, the DXML routine automatically picks the faster algorithm. Note that at $N=2^{16}$, as the size of the data and workspace exceeds the 2-MB secondary cache, the performance of the blocking algorithm drops off.

Optimization of the Skyline Solvers

A skyline matrix (Figure 6) is one where only the elements within the envelope of the sparse matrix are stored. This storage scheme exploits the fact that zeros that occur before the first nonzero element in a row or column of the matrix, remain zero during the factorization of the matrix, provided no row or column interchanges are made.[14] Thus, by storing the envelope of the matrix, no additional storage is required for the fill-in that occurs during the factorization. Though the skyline storage scheme does not exploit the sparsity within the envelope, it allows for a static data structure, and is therefore a reasonable compromise between organizational simplicity and computational efficiency.

In the skyline solver, the system, $Ax=b$, where A is an N by N matrix, and b and x are N-vectors, is solved by first factorizing A as $A=LDU$, where L and U are unit lower and upper triangular matrices, and D is a diagonal matrix. The solution x is then calculated by solving in order, $Ly=b$, $Dz=y$, and $Ux=z$, where y and z are N-vectors.

In our discussion of performance optimization, we concentrate on the factorization routine as it is often the most time-consuming part of an application. The algorithm implemented in DXML uses a technique that generates a column (or row) of the

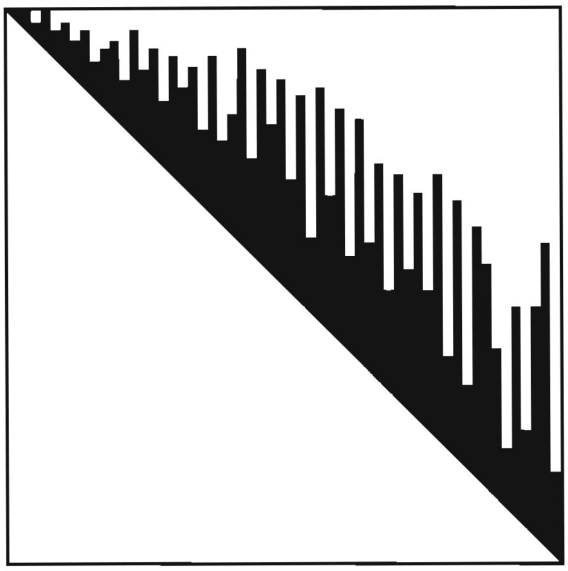


Figure 6 Skyline Column Storage of a Symmetric Matrix

factorization using an inner product formulation. Specifically, for a symmetric matrix A, let

$$A = \begin{pmatrix} M & v \\ v^T & s \end{pmatrix} = \begin{pmatrix} U_M^T & 0 \\ w^T & 1 \end{pmatrix} \begin{pmatrix} D_M & 0 \\ 0 & d \end{pmatrix} \begin{pmatrix} U_M & w \\ 0 & 1 \end{pmatrix}$$

where the symmetric factorization of the leading $(N - 1)$ by $(N - 1)$ leading principal submatrix M has already been obtained as

$$M = U_M^T D_M U_M$$

Since the vector v, of length $(N - 1)$, and the scalar s are known, the vector w, of length $(N - 1)$ and the scalar d can be determined as

$$w = D_M^{-1} U_M^{-T} v$$

and

$$d = s - w^T D_M w$$

The definition of w indicates that a column of the factorization is obtained by taking the inner product of the appropriate segment of that column with one of the previous columns that has already been calculated. Referring to Figure 7, the value of the element in location (i,j) is calculated by taking the inner product of the elements in column j above the element in location (i,j) with the corresponding elements in column i. The entire column

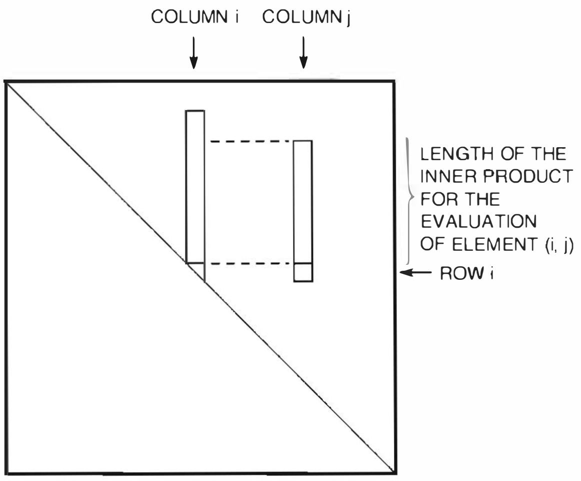

COLUMN i
COLUMN j
LENGTH OF THE INNER PRODUCT FOR THE EVALUATION OF ELEMENT (i, j)
ROW i

Figure 7 Unoptimized Skyline Computational Kernel

j is thus calculated starting with the first nonzero element in the column and moving down to the diagonal entry.

The optimization of the skyline factorization is based on the following two observations[25,26]:

- The elements of column j, used in the evaluation of the element in location (i,j), are also used in the evaluation of the element in location $(i+1,j)$.
- The elements of column i, used in the evaluation of the element in location (i,j), are also used in the evaluation of the element in location $(i,j+1)$.

Therefore, by unrolling both the inner loop on i and the outer loop on j, twice, we can generate the entries in locations (i,j), $(i+1,j)$, $(i,j+1)$, $(i+1,j+1)$ at the same time, as shown in Figure 8. These four elements are generated using only half the memory references made by the standard algorithm. The memory references can be reduced further by increasing the level of unrolling. This is, however, limited by two factors:

- The number of floating-point registers required to store the elements being calculated and the elements in the columns.
- The length of consecutive columns in the matrix, which should be close to each other to derive full benefit from the unrolling.

Based on these factors, we have unrolled to a depth of 4, generating 16 elements at a time.

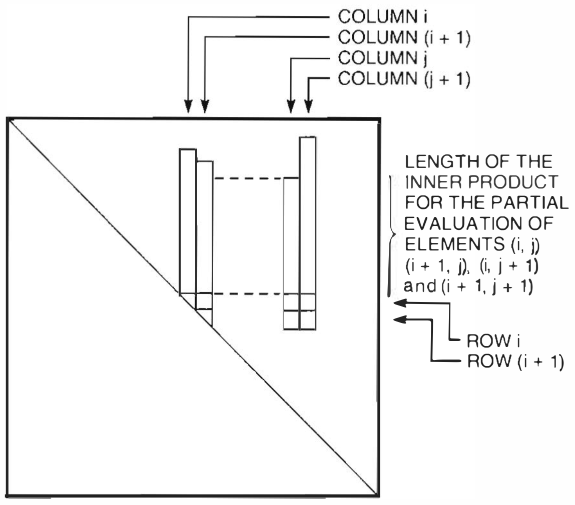

COLUMN i
COLUMN (i + 1)
COLUMN j
COLUMN (j + 1)
LENGTH OF THE INNER PRODUCT FOR THE PARTIAL EVALUATION OF ELEMENTS (i, j) (i + 1, j), (i, j + 1) and (i + 1, j + 1)
ROW i
ROW (i + 1)

Figure 8 Optimized Skyline Computational Kernel

A similar technique is used in optimizing the forward elimination and the backward substitution.

Table 1 gives the performance improvements obtained with the above techniques for a symmetric and an unsymmetric matrix from the Harwell-Boeing collection.[15] The characteristics of the matrix are generated using DXML routines and were included because the performance is dependent on the profile of the skyline. The data presented is for a single right-hand side, which has been generated using a known random solution vector.

The results show that for the matrices under consideration, the technique of reducing memory references by unrolling loops at two levels leads to a factor of 2 improvement in performance.

Summary

In this paper, we have shown that optimized mathematical subroutine libraries can be a useful tool in improving the performance of science and engineering applications on Alpha systems. We have

Table 1 Performance Improvement in the Solution of *Ax=b,* Using the Skyline Solver on the DEC 3000 Model 900 System

	Example 1	Example 2
Harwell-Boeing matrix[15]	BCSSTK24	ORSREG1
Description	Stiffness matrix of the Calgary Olympic Saddledome Arena	Jacobian from a model of an oil reservoir
Storage scheme	Symmetric diagonal-out	Unsymmetric profile-in
Matrix characteristics		
Order	3562	2205
Type	Symmetric	Unsymmetric with structural symmetry
Condition number estimate	6.37E+11	1.54E+4
Number of nonzeros	81736	14133
Size of skyline	2031722	1575733
Sparsity of skyline	95.98%	99.10%
Maximum row (column) height	3334	442 (442)
Average row (column) height	570.39	357.81 (357.81)
RMS row (column) height	1135.69	395.45 (395.45)
Factorization time (in seconds)		
Before optimization	66.80	23.12
After optimization	35.02	13.02
Solution time (in seconds)		
Before optimization	0.82	0.32
After optimization	0.43	0.17
Maximum component-wise relative error in solution (See equation below.)	0.16E−5	0.50E−10

$$\max_i \frac{|x(i) - \bar{x}(i)|}{|x(i)|}$$

where $x(i)$ is the i-th component of the true solution, and $\bar{x}(i)$ is the i-th component of the calculated solution.

described the functionality provided by DXML, discussed various software engineering issues and illustrated techniques used in performance optimization.

Future enhancements to DXML include symmetric multiprocessing support for key routines, enhancements in the areas of signal processing and sparse solvers, as well as further optimization of routines as warranted by changes in hardware and system software.

Acknowledgments

DXML is the joint effort of a number of individuals over the past several years. We would like to acknowledge the contributions of our colleagues, both past and present. The engineers: Luca Broglio, Richard Chase, Claudio Deiro, Laura Farinetti, Leo Lavin, Ping-Charng Lue, Joe O'Connor, Mark Schure, Linda Tella, Sisira Weeratunga and John Wilson; the technical writers: Cheryl Barabani, Barbara Higgins, Marll McDonald, Barbara Schott and Richard Wolanske; and the management: Ned Anderson, Carlos Baradello, Gerald Haigh, Buren Hoffman, Tomas Lofgren, Vehbi Tasar and David Velten. We would also like to thank Roger Grimes at Boeing Computer Services for making the Harwell-Boeing matrices so readily available.

References

1. W. Cowell, ed., *Sources and Development of Mathematical Software* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

2. D. Jacobs, ed., *Numerical Software—Needs and Availability* (New York: Academic Press, 1978).

3. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK Users' Guide* (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1979).

4. B. Smith et al., *Matrix Eigensystem Routines—EISPACK Guide* (Berlin: Springer-Verlag, 1976).

5. *Digital eXtended Math Library Reference Manual* (Maynard, MA: Digital Equipment Corporation, Order No. AA-Q0MBB-TE for VMS and AA-Q0NHB-TE for OSF/1).

6. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3 (September 1979): 308-323.

7. D. Dodson, R. Grimes, and J. Lewis, "Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 17, no. 2 (June 1991): 253-263.

8. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1 (March 1988): 1-17.

9. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1 (March 1990): 1-17.

10. E. Anderson et al., *LAPACK Users' Guide* (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1992).

11. J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1991).

12. R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1993).

13. C. Felippa, "Solution of Linear Equations with Skyline Stored Symmetric Matrix," *Computer and Structures*, vol. 5, no. 1 (April 1975): 13-29.

14. I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices* (New York: Oxford University Press, 1986).

15. I. Duff, R. Grimes, and J. Lewis, "Sparse Matrix Test Problems," *ACM Transactions on Mathematical Software*, vol. 15, no. 1 (March 1989): 1-14.

16. Alpha AXP Architecture and Systems, *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992).

17. K. Dowd, *High Performance Computing* (Sebastopol, CA: O'Reilly & Associates, Inc., 1993).

18. *DECchip 21064-AA Microprocessor—Hardware Reference Manual* (Maynard, MA: Digital Equipment Corporation, Order No. EC-N0079-72, October 1992).

19. J. Dongarra and S. Eisenstat, "Squeezing the Most Out of an Algorithm in CRAY FORTRAN," *ACM Transactions on Mathematical Software,* vol. 10, no. 3 (September 1984): 219-230.

20. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, 1992).

21. H. Nussbaumer, *Fast Fourier Transforms and Convolution Algorithms,* Second Edition (New York: Springer Verlag, 1982).

22. D. Bailey, "A High-performance FFT Algorithm for Vector Supercomputers," *The International Journal of Supercomputer Applications,* vol. 2, no. 1 (Spring 1988): 82-87.

23. P. Swarztrauber, "FFT Algorithms for Vector Computers," *Parallel Computing,* vol. 1, no. 1 (August 1984): 45-63.

24. D. Bailey, "FFTs in External or Hierarchical Memory," *The Journal of Supercomputing,* vol. 4, no. 1 (March 1990): 23-35.

25. O. Storaasli, D. Nguyen, and T. Agarwal, "Parallel-Vector Solution of Large-Scale Structural Analysis Problems on Supercomputers," *American Institute of Aeronautics and Astronautics (AIAA) Journal,* vol. 28, no. 7 (July 1990): 1211-1216.

26. H. Samukawa, "A Proposal of Level 3 Interface for Band and Skyline Matrix Factorization Subroutine," *Proceedings of the 1993 ACM International Conference on Super Computing,* Tokyo, Japan (July 1993): 397-406.

Robert H. Kuhn
Bruce Leasure
Sanjiv M. Shah

The KAP Parallelizer for DEC Fortran and DEC C Programs

The KAP preprocessor optimizes DEC Fortran and DEC C programs to achieve their best performance on Digital Alpha systems. One key optimization that KAP performs is the parallelization of programs for Alpha shared memory multiprocessors that use the new capabilities of the DEC OSF/1 version 3.0 operating system with DECthreads. The heart of the optimizer is a sophisticated decision process that selects the best loop to parallelize from the many loops in a program. The preprocessor implements a robust data dependence analysis to determine whether a loop is inherently serial or parallel. In engineering a high-quality optimizer, the designers specified the KAP software architecture as a sequence of modular optimization passes. These passes are designed to restructure the program to resolve many of the apparent serializations that are artifacts of coding in Fortran or C. End users can also annotate their DEC Fortran or DEC C programs with directives or pragmas to guide KAP's decision process. As an alternative to using KAP's automatic parallelization capability, end users can explicitly identify parallelism to KAP using the emerging industry-standard X3H5 directives.

The KAP preprocessor developed by Kuck & Associates, Inc. (KAI) is used on Digital Alpha systems to increase the performance of DEC Fortran and DEC C programs. KAP accomplishes this by restructuring fragments of code that are not efficient for the Alpha architecture. Essentially a super-optimizer, KAP performs optimizations at the source code level that augment those performed by the DEC Fortran or DEC C compilers.[1]

To enhance the performance of DEC Fortran and DEC C programs on Alpha systems, KAI engineers selected two challenging aspects of the Alpha architecture as KAP targets: symmetric multiprocessing (SMP) and cache memory. An additional design goal was to assist the compiler in optimizing source code for the reduced instruction set computer (RISC) instruction processing pipeline and multiple functional units.

This paper discusses how the KAP preprocessor design was adapted to parallelize programs for SMP systems running under the DEC OSF/1 version 3.0 operating system. This version of the DEC OSF/1 system contains the DECthreads product, Digital's POSIX-compliant multithreading library. The first part of the paper describes the process of mapping parallel programs to DECthreads. The paper then discusses the key techniques used in the KAP design. Finally, the paper presents examples of how KAP performs on actual code and mentions some remaining challenges. Readers with a compiler background may wish to explore *Optimizing Supercompilers for Supercomputers* for more details on KAP's techniques.[2]

In this paper, the term directive is used interchangeably to mean directive, when referring to DEC Fortran programs, and pragma, when referring to DEC C programs. The term processor generally represents the system component used in parallel processing. In discussions in which it is significant to distinguish the operating system component used for parallel processing, the term thread is used.

The Parallelism Mapping Process

Figure 1 shows the input modes and major phases of the compilation process. Parallelism is represented at three levels in programs using the KAP preprocessor on an Alpha SMP system. The first two are input to the KAP preprocessor; the third is the

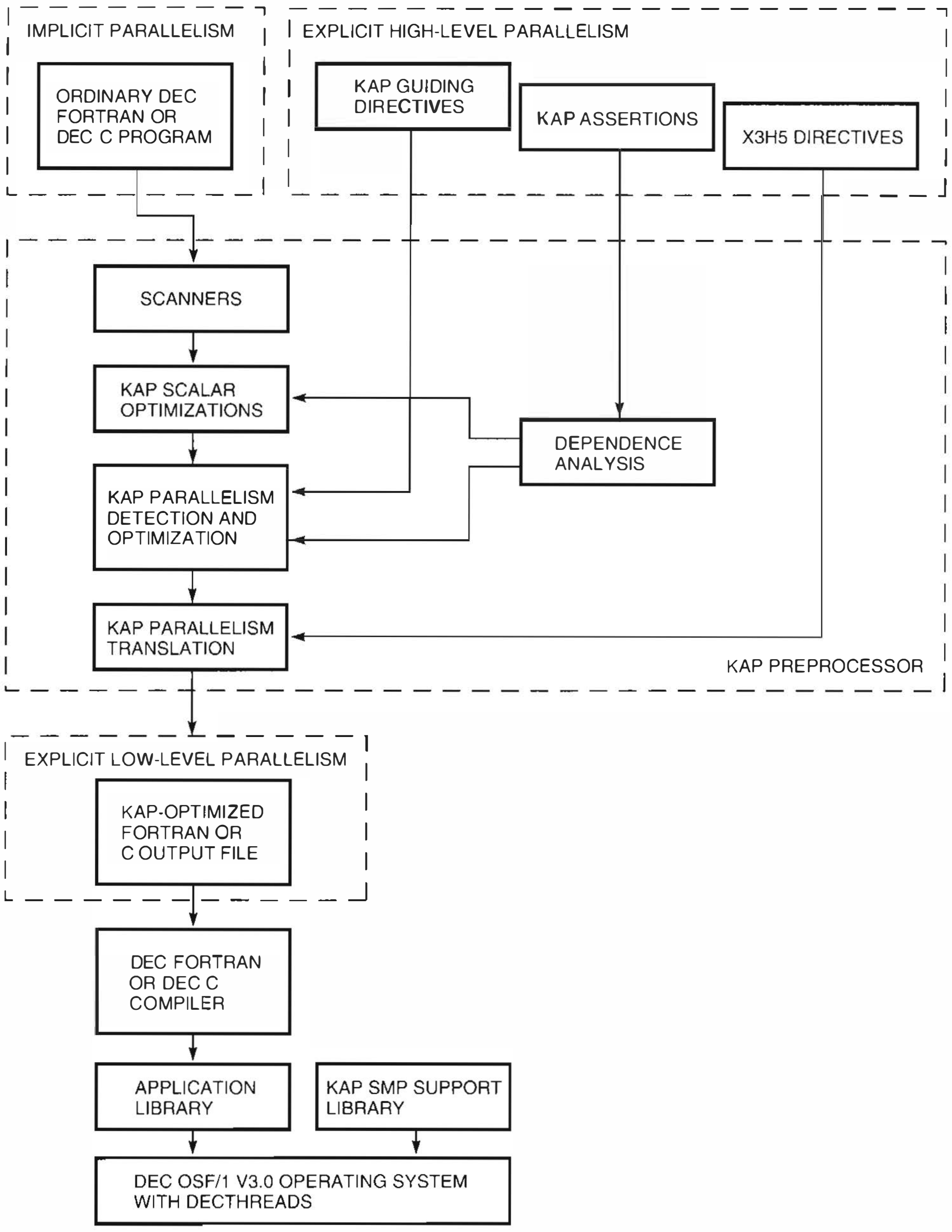

IMPLICIT PARALLELISM
ORDINARY DEC FORTRAN OR DEC C PROGRAM
EXPLICIT HIGH-LEVEL PARALLELISM
KAP GUIDING DIRECTIVES
KAP ASSERTIONS
X3H5 DIRECTIVES
SCANNERS
KAP SCALAR OPTIMIZATIONS
DEPENDENCE ANALYSIS
KAP PARALLELISM DETECTION AND OPTIMIZATION
KAP PARALLELISM TRANSLATION
KAP PREPROCESSOR
EXPLICIT LOW-LEVEL PARALLELISM
KAP-OPTIMIZED FORTRAN OR C OUTPUT FILE
DEC FORTRAN OR DEC C COMPILER
APPLICATION LIBRARY
KAP SMP SUPPORT LIBRARY
DEC OSF/1 V3.0 OPERATING SYSTEM WITH DECTHREADS

Figure 1 Parallelism Mapping Process

representation of parallelism that KAP generates. The three levels of parallelism are

1. Implicit parallelism. Starting from DEC Fortran or DEC C programs, KAP automatically detects parallelism.

2. Explicit high-level parallelism. As an advanced feature, users can provide any of three forms: KAP guiding directives, KAP assertions, or X3H5 directives. KAP guiding directives give KAP hints on which program constructs to parallelize. KAP assertions are used to convey information about the program that cannot be described in the DEC Fortran or DEC C language. This information can sometimes be used by KAP to optimize the program. Using X3H5 directives, the user can force KAP to parallelize the program in a certain way.[3]

3. Explicit low-level parallelism. KAP translates either of the above forms to DECthreads with the help of an SMP support library. (The user could specify parallelism directly, using DECthreads; however, KAP does not perform any optimization of source code with DECthreads. Therefore, the user should not mix this form of parallelism with the others.)

Because the user can employ parallelism at any of the three levels, a discussion of the trade-offs involved with using each level follows.

From DEC Fortran or DEC C Programs

The KAP preprocessor accepts DEC Fortran and DEC C programs as input. Although starting with such programs requires the compilers to intelligently utilize a high-performance SMP system, there are several reasons why this is a natural point at which to start.

- Lots of software. Since DEC Fortran and DEC C are de facto standards, there exists a large base of applications that can be parallelized relatively easily and inexpensively.
- Ease of use. Given the high rate at which hardware costs are decreasing, every workstation may soon have multiple processors. At that point, it will be critical that programming a multiprocessor be as easy as programming a single processor.
- Portability. Many software developers with access to a multiprocessor already work in a heterogeneous networking environment. Some systems in such an environment do not support explicit forms of parallelism (either X3H5 or DECthreads). The developers would probably like to have one version of their code that runs well on all their systems, whether uniprocessor or multiprocessor, and using DECthreads would cause their uniprocessors to slow down.
- Maintainability. Using an intricate programming model of parallelism such as X3H5 or DECthreads makes it more difficult to maintain the software.

KAP produces KAP-optimized DEC Fortran or DEC C as output. This fact is important for the following reasons:

- Performance. Users can leverage optimizations from both Digital's compilers and KAP.
- Integration. Users can employ all of Digital's performance tools.
- Ease of use. Expert users like to "tweak" the output of KAP to fine-tune the optimizations performed.

With KAP Guiding Directives, KAP Assertions, or X3H5 Directives

Although the automatic detection of parallelism is frequently within the range of KAP capabilities on SMP systems, in some cases, as described below, users may wish to specify the parallelism.

- In the SMP environment, coarse-grained parallelism is sometimes important. The higher in the call tree of a program a preprocessor (or compiler, as well) operates, the more difficult it is for a preprocessor to parallelize automatically. Even though the KAP preprocessor performs both inlining and interprocedural analysis, the higher in the call tree KAP operates, the more likely it is that KAP will conservatively assume that the parallelization is invalid.
- Sometimes information that is available only at run time precludes the preprocessor from automatically finding parallelism.
- Occasionally, experts can fine-tune the parallelism to get the highest efficiency for programs that are run frequently.
- For software that is more portable between systems, it is sometimes important to get repeatable parallel performance or to indicate where parallelism has been applied. In such cases, explicit parallelism may be preferable.

Three mechanisms are available to the user for directing KAP to parallelism. The first mechanism uses KAP guiding directives to guide KAP to the preferred way to parallelize the program. The second mechanism uses KAP assertions. The third mechanism uses X3H5-compliant directives to directly describe the parallelism. The first two mechanisms differ significantly from the third. With the first two, KAP analyzes the program for the feasibility of parallelism. With the third, KAP assumes that parallelism is feasible and restricts itself to managing the details of implementing parallelism. In particular, the user does not have to be concerned with either the scoping of variables across processors, i.e., designating which are private and which are shared, or the synchronization of accesses to shared variables.[4] KAP guiding directives will not be discussed in this paper. KAP assertions and how they are implemented are discussed later in the section Advanced Ways to Affect Dependences. A description of the X3H5 directives follows.

The X3H5 model of parallelism is well structured; all operations have a begin operation-end operation format. The parallel region construct identifies the fork and join points for parallel processing. Parallel loops identify units of work to be distributed to the available processors. The critical section and one processor section constructs are used to synchronize processors where necessary. Table 1 shows the X3H5 directives as implemented in KAP.

To the DEC OSF/1 Operating System with DECthreads

Although KAP does not optimize programs that use DECthreads directly, there may be some benefits to specifying parallelism explicitly using DECthreads.

- DECthreads allows a user to construct almost any model of parallel processing fairly efficiently. The high-level approaches described above are limited to loop-structured parallel processing. Some applications obtain more parallelism by using an unstructured model. It can even be argued that for some cases, unstructured parallelism is easier to understand and maintain.
- A user who invests the time to analyze exactly where parallelism exists in a program may wish to forego the benefits mentioned above and to capture the parallelism in detail with DECthreads. In that manner, no efficiency is lost because the preprocessor misses an optimization.
- The POSIX threads standard to which DECthreads conforms is available on several platforms. Because this standard is broadly adopted and language independent, it is only slightly less portable than implicit parallelism.

The KAP preprocessor translates a program in which KAP has detected implicit parallelism or a program in which the user explicitly directs parallelism to DECthreads. KAP performs this translation in two steps. First, it translates the internal representation into calls to a parallel SMP support library. Second, the support library makes calls to DECthreads.

The SMP support library implements various aspects of X3H5 notation, as can be seen by comparing Tables 1 and 2.

In the parallelism translation phase, KAP significantly restructures a program by moving the code in a parallel region to a separate subroutine. A call to the SMP support library replaces the parallel region. This call references the new subroutine. KAP examines the scope of each variable used in the parallel region and, if possible, converts each variable to a local variable of the new subroutine. Otherwise, the variable becomes an argument to the subroutine so that it can be passed back out of the parallel region.

Converting variables to local variables makes accessing these variables more efficient. A variable that is referenced outside the parallel region cannot be made local and must be passed as an argument.

Shared Memory Multiprocessor Architecture Concerns

Given its parallelism model, the KAP preprocessor requires operating system and hardware support from the system for efficient parallel execution. There are three areas of concern: thread creation and scheduling, synchronization between threads, and data caching and system bus bandwidth.

Table 1 X3H5 Directives As Implemented in KAP

Function	X3H5 Directives
To specify regions of parallel execution	*C*KAP* PARALLEL REGION* *C*KAP* END PARALLEL REGION*
To specify parallel loops	*C*KAP* PARALLEL DO* *C*KAP* END PARALLEL DO*
To specify synchronized sections of code such that all processors synchronize	*C*KAP* BARRIER*
To specify that all processors execute sequentially	*C*KAP* CRITICAL SECTION* *C*KAP* END CRITICAL SECTION*
To specify that only the first processor executes	*C*KAP* ONE PROCESSOR SECTION* *C*KAP* END ONE PROCESSOR SECTION*

Table 2 KAP SMP Support Library

C Entry Point Name	Fortran Name	Function	OSF/1 DECthreads Subroutines Used
__kmp_enter_csec	mppecs	To enter a critical section	pthread_mutex_lock
__kmp_exit_csec	mppxcs	To exit a critical section	pthread_mutex_unlock
__kmp_fork	mppfrk	To fork to several threads	pthread_attr_create, pthread_create
__kmp_fork_active	mppfkd	To inquire if already parallel	(none)
__kmp_end	mppend	To join threads	pthread_join, thread_detach
__kmp_enter_onepsec	mppbop	To enter a single processor section	pthread_mutex_lock, pthread_mutex_unlock
__kmp_exit_onepsec	mppeop	To exit a single processor section	pthread_mutex_lock, pthread_mutex_unlock
__kmp_barrier	mppbar	To execute a barrier wait	pthread_mutex_lock, pthread_cond_wait, pthread_mutex_unlock

Thread Creation and Scheduling Thread creation is the most expensive operation. The X3H5 standard minimizes the need for creating threads through the use of parallel regions. The SMP support library goes further by reusing threads from one parallel region to the next. The SMP support library examines the value of an environment variable to determine how many threads to use. The appropriate scheduling of threads onto hardware processors is extremely important for efficient execution. The support library relies on the DECthreads implementation to achieve this. For the most efficient operation, the library should schedule at most one thread per processor.

Synchronization between Threads In the KAP model of parallelism, threads can synchronize at

- A point where loop iterations are scheduled
- A point where data passes between iterations (for collection of local reduction variables only)
- A barrier point leaving a work-sharing construct
- Single processor sections

Two versions of the SMP support library have been developed: one with spin locks for a single-user environment and the second with mutex locks for a multiuser environment. Either library works in either environment; however, using the spin lock version in a single-user environment yields the most efficient parallelism.

Using spin locks in a multiuser environment may waste processor cycles when there are other users who could use them. Using mutex locks for a single-user environment creates unnecessary operating system overhead. In practice, however, a system may shift from single-user to multiuser and back again in the course of a single run of a large program. Therefore, KAP supports all lock-environment combinations.

Data Caching and System Bus Bandwidth Multiprocessor Alpha systems support coherent caches between processors.[5] To use these caches efficiently, as a policy, KAP localizes data as much as possible, keeping repeated references within the same processor. Localizing data reduces the load on the system bus and reduces the chances of cache thrashing.

When all the processors simultaneously request data from the memory, system bus bandwidth can limit SMP performance. If optimizations enhance cache locality, less system bus bandwidth is used, and therefore SMP performance is less likely to be limited.

KAP Technology

This section covers the issues of data dependence analysis, preprocessor architecture, and the selection of loops to parallelize.

Data Dependence Analysis—The Kernel of Parallelism Detection

DEC Fortran and DEC C have standard rules for the order of execution of statements and expressions. These rules are based on a serial model of program execution. Data dependence analysis allows a compiler to see where this serial order of execution can be modified without changing the meaning of the program.

Types of Dependence KAP works with the four basic types of dependence:[6]

1. Flow dependence, i.e., when a program writes a variable before it reads the variable
2. Antidependence, i.e., when a program reads a variable before it writes the variable
3. Output dependence, i.e., when a program writes the same variable twice
4. Control dependence, i.e., when a program statement depends on a previous conditional

Because dependences involve two actions on the same variable, for example, a write and then a read, KAP uses the term dependence arc to represent information flow, in this example from the write to the read.

Since these dependences can prevent parallelization, KAP uses various optimizations to eliminate the different dependences. For example, an optimization called scalar renaming removes some but not all antidependences.

Loop-related Dependences When dependences occur within a loop, the control flow relations are captured with direction vector symbols tagged to each dependence arc.[2] The transformations that can be applied to a loop depend on what dependence direction vectors exist for that loop. The symbols used in KAP and their meanings are

=	The dependence occurs within the same loop iteration.
>	The dependence crosses one or several iterations.
<	The dependence goes to a preceding iteration of the loop.
*	The dependence relation between iterations is not clear.

or a combination of the above, for example,

<>	The dependence is known not to be on the same iteration.

When a dependence occurs in a nested loop, KAP uses one symbol for each level in the loop nest. A dependence is said to be carried by a loop if the corresponding direction vector symbol for that loop includes <, >, or *.

In the following program segment

```
1  for (i=1; i<=n; i++) {
2      temp = a[i];
3      a[i] = b[i];
4      b[i] = temp; }
```

there is a flow dependence from statement 2 to statement 4. There is an antidependence from statement 2 to statement 3 and from statement 3 to statement 4. There are control dependences from statement 1 to statements 2, 3, and 4 because executing 2, 3, and 4 depends on the $i <= n$ condition. All these dependences are on the same loop iteration; their direction vector is =.

Some dependences in this program cross loop iterations. Because temp is reused on each iteration, there is an output dependence from statement 2 to statement 2, and there is an antidependence from statement 4 to statement 2. These two dependences are carried by the loop in the program segment and have the direction vector >.

Data Dependence Analysis The purpose of dependence analysis is to build a dependence graph, i.e., the collection of all the dependence arcs in the program. KAP builds the dependence graph in two stages. First, it builds the best possible conservative dependence graph.[7] Then, it applies filters that identify and remove dependences that are known to be conservative, based on special circumstances.

What does the phrase "best possible conservative dependence graph" mean? Because the values of a program's variables are not known at preprocessing time, in some situations it may not be clear whether a dependence actually exists. KAP reflects this situation in terms of assumed dependences based on imperfect information. Therefore, a dependence graph must be conservative so that KAP does not optimize a program incorrectly. On the other hand, a dependence graph that is too conservative results in insufficient optimization.

In building the best possible dependence graph, KAP uses the following optimizations: constant propagation, variable forward substitution, and scalar expansion. KAP does not, however, leave the program optimized in this manner unless the optimizations will improve performance.

Advanced Ways to Affect Dependences When there are assumed dependences in the program, KAP may not have enough information to decide on parallelism opportunities. KAP implements two techniques to mitigate the effects of imperfect information at preprocessing time: assertions and alternate code sequences.

Assertions, which are similar to directives in syntax, are used to provide information not otherwise known at preprocessing time. KAP supports many assertions that have the effect of removing assumed dependences. Table 3 shows KAP assertions and their effects.[8,9] When the user specifies an assertion, the information contained in the assertion is saved by a data abstraction called the oracle. When an optimization requests that a data dependence graph be built for a loop, the dependence analyzer inquires whether the oracle has any information about certain arcs that it wants to remove.

When accurate information is not known at compile time, a few KAP optimizations generate two versions of the source program loop: one assumes that the assumed dependence exists; the other assumes that it does not exist. In the latter case, KAP can apply subsequent optimizations, such as parallelizing the loop. KAP applies the two-version loop optimizations selectively to avoid dramatically increasing the size of the program. However, the payback of parallelizing a frequently executed loop warrants their use.

For example, the KAP C pointer disambiguation optimization is employed in cases in which C pointers are used as a base address and then incremented in a loop. Neither the base address of a pointer nor how many times the pointer will be incremented is usually known at compile time. At run time, however, they can be computed in terms of a loop index. KAP generates code that checks the range of the pointer references at the tail and at the head of a dependence. If the two ranges do not overlap, the dependence does not exist and the optimized code is executed.

KAP Preprocessor Architecture

A controversial control architecture decision in KAP is to organize the preprocessor as a sequence of passes, generally one for each optimization performed. This design decision was controversial because of the following concerns:

- Run-time inefficiency would occur in processing programs because each pass would sweep through the intermediate representation for the program being processed, causing some amount of virtual memory thrashing.

Table 3 KAP Assertions

Assertion	Specifiers	Primary Effect
`[NO] ARGUMENT ALIASING`		Removes assumed dependence arcs
`[NO] BOUNDS VIOLATIONS`		Removes assumed dependence arcs
`CONCURRENT CALL`		Removes assumed dependence arcs
`DO` (<specifier>)	`SERIAL, CONCURRENT`	Guides selection of loop order strongly
`DO PREFER` (<specifier>)	`SERIAL, CONCURRENT`	Guides selection of loop order loosely
`[NO] EQUIVALENCE HAZARD`		Removes assumed dependence arcs (Fortran only)
`[NO] LAST VALUE NEEDED` (<specifier>)	Variable names for which [no] last value is needed	Tunes the parallel code and sometimes removes assumed dependences
`PERMUTATION` (<specifier>)	Names of permutation variables	Removes assumed dependence arcs
`NO RECURRENCE` (<specifier>)	Names of recurrence variables	Removes assumed dependence arcs
`RELATION` (<specifier>)	Relation loop index known to be true	Removes assumed dependence arcs
`NO SYNC`		Tunes the parallel code which is produced

- Added software development cost would be incurred because the KAP code that loops through the intermediate representation would be repeated in each pass.

The second concern has been dispelled. The added modularity of KAP, provided by its multipass structure, has saved development time as KAP has grown from a moderately complex piece of code to an extremely complex piece of code.

The KAP preprocessor uses more than 50 major optimizations. The pass structure has helped to organize them. In some cases, such as cache management, one optimization is broken into several passes. KAP performs some basic optimizations, e.g., deadcode elimination, more than once in different ways. In some cases, such as scalar expansion, KAP performs an optimization to uncover other optimizations and then performs the reverse optimization to tighten up the program again.

The run-time efficiency issue is still of interest. There is always some benefit to making the preprocessor smaller and faster.

Selecting Loops to Parallelize

Parallelizing a loop can greatly enhance the performance of the program. Testing whether a loop can be parallelized is actually quite simple, given the data dependence analysis that KAP performs. A loop can be parallelized if there are no dependence arcs carried by that loop. The situation, however, can be more complicated. If the program contains several nested loops, it is important to pick the best loop to parallelize. Additionally, it may be possible not only to parallelize the loop but also to optimize the loop to enhance its performance. Moreover, the loops in a program can be nested in very complex structures so that there are many different ways to parallelize the same program. In fact, the best option may be to leave all the loops serial because the overhead of parallel execution may outweigh the performance improvement of using multiple processors.

The KAP preprocessor optimizes programs for parallelism by searching for the optimum program in a set of possible configurations, i.e., ways in which the original program can be transformed for parallel execution. (In this regard, KAP optimizes programs from a classical definition of numerical optimization.) There is an objective function for evaluating each configuration. Each member of the set of configurations is called a loop order. The optimum program is the loop order whose objective function has the highest performance score, as discussed later in this section.

Descriptions of loop orders, the role of dependence analysis, and the objective function, i.e., how each program is scored, follow.

Loop Orders A loop order is a combination of loop transformations that the KAP preprocessor has performed on the program. The loop transformations that KAP performs while searching for the optimal parallel form are

- Loop distribution
- Loop fusion
- Loop interchange

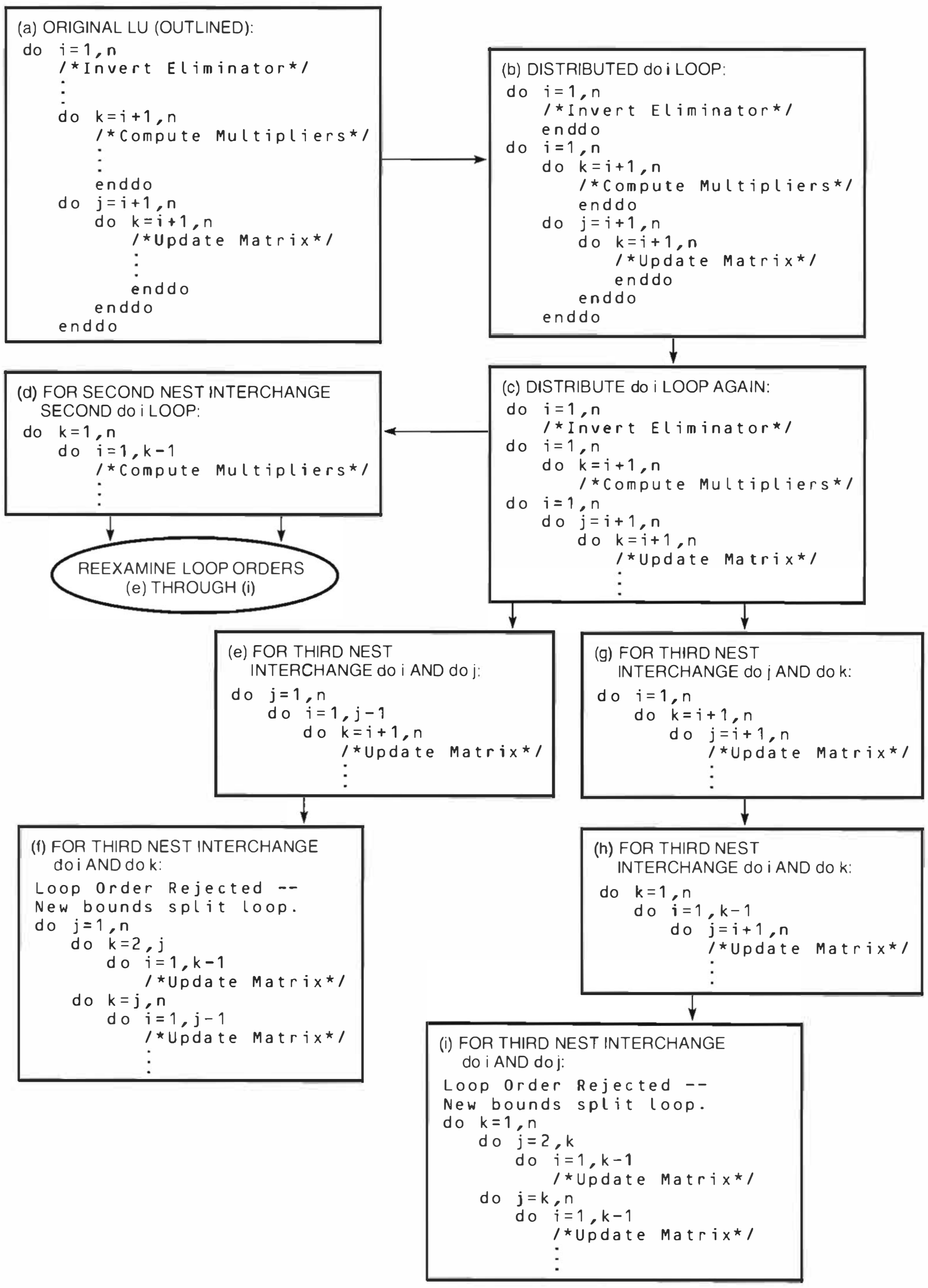
Loop distribution splits a loop into two or more loops. Loop fusion merges two loops. Loop fusion is used to combine loops to increase the size of the parallel tasks and to reduce loop overhead.

Loop interchange occurs between a pair of loops. This transformation takes the inner loop outside the outer loop, reversing their relation. If a loop is triply nested, there are three factorial (3!), i.e., six, different ways to interchange the loops. Each order is arrived at by a sequence of pairwise interchanges.

To increase the opportunities to interchange loops, KAP tries to make a loop nest into one that is perfectly nested. This means that there are no executable statements between nested loop statements. Loop distribution is used to create perfectly nested loops.

KAP examines all possible loop orders for each loop nest. Each loop nest is treated independently because no transformations between loop nests occur at this phase of optimization.

For example, an LU factorization program consists of one loop nest that is three deep and not perfectly nested. Figure 2 shows the loop orders. Loop order (a) is the original LU program. The KAP preprocessor first distributes the outer loop in loop orders (b) and (c). Next, KAP performs a loop interchange on the second loop nest which is two deep, as shown in loop order (d). Then, KAP interchanges the third loop nest in loop orders (e) through (i). Note that KAP eliminates some loop orders, (i) for example, when the loop-bound expressions cannot be interchanged. As explained above, there are six different loop orders because the nest is triply



(a) ORIGINAL LU (OUTLINED):

```
do i=1,n
   /*Invert Eliminator*/
   :
   do k=i+1,n
      /*Compute Multipliers*/
      :
      enddo
   do j=i+1,n
      do k=i+1,n
         /*Update Matrix*/
         :
         enddo
      enddo
   enddo
```

(b) DISTRIBUTED do i LOOP:

```
do i=1,n
   /*Invert Eliminator*/
   enddo
do i=1,n
   do k=i+1,n
      /*Compute Multipliers*/
      enddo
   do j=i+1,n
      do k=i+1,n
         /*Update Matrix*/
         enddo
      enddo
   enddo
```

(c) DISTRIBUTE do i LOOP AGAIN:

```
do i=1,n
   /*Invert Eliminator*/
do i=1,n
   do k=i+1,n
      /*Compute Multipliers*/
do i=1,n
   do j=i+1,n
      do k=i+1,n
         /*Update Matrix*/
         :
```

(d) FOR SECOND NEST INTERCHANGE SECOND do i LOOP:

```
do k=1,n
   do i=1,k-1
      /*Compute Multipliers*/
      :
```

(e) FOR THIRD NEST INTERCHANGE do i AND do j:

```
do j=1,n
   do i=1,j-1
      do k=i+1,n
         /*Update Matrix*/
         :
```

(g) FOR THIRD NEST INTERCHANGE do j AND do k:

```
do i=1,n
   do k=i+1,n
      do j=i+1,n
         /*Update Matrix*/
         :
```

(f) FOR THIRD NEST INTERCHANGE do i AND do k:

```
Loop Order Rejected --
New bounds split loop.
do j=1,n
   do k=2,j
      do i=1,k-1
         /*Update Matrix*/
   do k=j,n
      do i=1,j-1
         /*Update Matrix*/
         :
```

(h) FOR THIRD NEST INTERCHANGE do i AND do k:

```
do k=1,n
   do i=1,k-1
      do j=i+1,n
         /*Update Matrix*/
         :
```

(i) FOR THIRD NEST INTERCHANGE do i AND do j:

```
Loop Order Rejected --
New bounds split loop.
do k=1,n
   do j=2,k
      do i=1,k-1
         /*Update Matrix*/
   do j=k,n
      do i=1,k-1
         /*Update Matrix*/
         :
```

Figure 2 Loop Orders for LU Factorization

nested. Since the loop nest in (d) was originally nested with the triply nested loop at the outermost do loop, KAP will reexamine these six loop orders after the interchange in (d).

Dependence Analysis for Loop Orders Before a loop order can be evaluated for efficiency, KAP determines the validity of the loop order. A loop order is valid if the resulting program would produce equivalent behavior. KAP tests validity by examining the dependences in the dependence graph according to the transformation being applied.

For example, the test for loop interchange validity involves searching for dependence direction vectors of a certain type. The direction vector (<,>) indicates that a loop interchange is invalid. The direction vectors (<,*), (*,>), or (*,*), if present, also indicate that the loop interchange may be invalid.

Evaluation of a Loop Order After the KAP preprocessor determines that a loop order is valid, it scores the loop order for performance. KAP considers two major factors: (1) the amount of work that will be performed in parallel and (2) the memory reference efficiency.

The memory reference efficiency of a loop order can degrade performance so much that it outweighs the performance gained by executing a loop in parallel. On an SMP, if a processor references one word on a cache line, it should reference all the words contiguously on that line. In Fortran, a two-dimensional array reference, A(*i,j*), should be parallelized so that the *j* loop is parallel and each processor references contiguous columns of memory. If a loop order indicated that the *i* loop is parallel, this reference would score low. If a loop order indicated that the *j* loop is parallel, it would score high. The score for the loop order is the sum of the scores for all the references, and the highest-scoring loop order is preferred.

The score for a loop order depends on which loops in the order can be parallelized. For a given loop nest, there may be several (or no) loops that can be parallelized. The first step is to determine if any loops can be parallelized. If multiple loops can be parallelized, KAP selects the best one. KAP chooses at most one loop for parallel execution.

KAP tests loops to determine whether they can be executed in parallel by analyzing both the statements in the loop and the dependence graph. The loop may contain certain statements that block concurrentization. I/O statements or a call to a function or subroutine are examples. (Users can code KAP assertions to flag these statements as parallelizable.) Second, data dependence conditions may preclude parallelization. In general, a loop that carries a dependence is not parallelizable. (In some cases, the user may override the data dependence condition by allowing synchronization between loop iterations.) Finally, the user may give assertions that indicate a preference for making a loop parallel or for keeping it serial.

Barring data dependence conditions that would prevent parallelization, the amount of work that will be performed in parallel determines the score of parallelizing a loop. (The user can also specify with a directive that loops should not be parallelized unless they score greater than a specified value.) In this manner, KAP prefers to parallelize outer loops or loops that are interchanged to the outside because they contain the most work to amortize the overhead of creating threads for parallelism.

The actual parallelization process is even more complex than this discussion indicates. KAP applies a number of optimizations to improve the quality of the parallel code. If there is a reduction operation across a loop, KAP parallelizes the loop. Too much loop distribution can decrease program efficiency, so loop fusion is run to try to coalesce loops.

Performance Analysis

How does the KAP preprocessor perform on real applications? The answer is as complex as the software written for these applications. Consider the real-world example, DYNA3D, which demonstrates some KAP strengths and weaknesses.

DYNA3D is nonlinear structural dynamics code that uses the finite element analysis method. The code was developed by the Lawrence Livermore National Laboratory Methods Development Group and has been used extensively for a broad range of structural analysis problems. DYNA3D contains about 70,000 lines of Fortran code in more than 700 subroutines.

When KAP is being used on a large program, it is sometimes preferable to concentrate on the compute-intensive kernels. For example, KAP developers ran six of the standard benchmarks for DYNA3D through a performance profiling tool and isolated two groups of three subroutines that account for approximately 75 percent of the run time in these cases. This data is shown in Table 4.

KAP's performance on some of these key subroutines appears in Table 5. KAP parallelized all the

Table 4 Performance Profiles of Six DYNA3D Problems

Problem	Profile (First Two Initials of the Subroutine and Percent of Run Time)	Key Call Sequences*
NIKE2D Example	ST 19%, FO 15%, FE 12%, PR 10%, HG 7%, HR 5%	(a) and (b)
Cylinder Drop	ST 20%, FO 15%, FE 11%, PR 10%, HG 7%, HR 5%	(a) and (b)
Bar Impact	WR 17%, ST 7%, FE 6%	None of interest
Impacted Plate	SH 22%, TN 16%, TA 16%, YH 14%, BL 7%	(c)
Single Contact	YH 24%, SH 21%, TN 7%, TA 7%, BL 6%	(c)
Clamped Beam	EL 12%, SH 12%, TN 8%, TA 8%, BL 6%	(c)

*Call Sequences
(a) ST is called; ST calls PR; and then FE is called.
(b) HR is called; HR calls HG; and then FO is called.
(c) BL calls SH, then TA, and then TN.

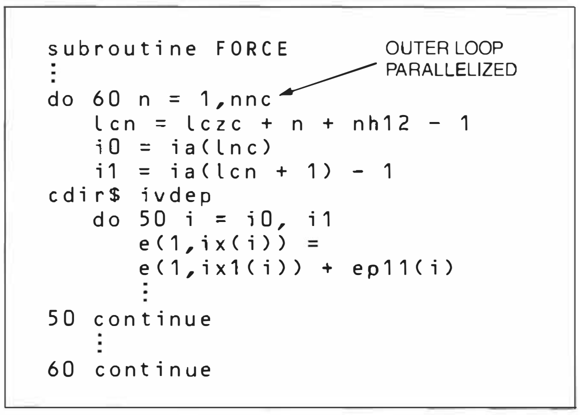
Table 5 KAP's Performance on Key Subroutines

Subroutine	Number of Loops	Number of Loops Parallelized	Maximum Nest Depth	Number of Loops after Fusion
STRAIN	5	5	1	3
PRTAL	9	9	1	1
FELEN	6	6	1	1
FORCE	9	9	2	2
HRGMD	5	5	1	3
HGX	4	4	1	1

loops in these subroutines. Since DYNA3D was designed for a CRAY-1 vector processor, it is perhaps to be expected that the KAP preprocessor would perform well. KAP, however, is intended for a shared memory multiprocessor rather than for a vector machine. For this reason, KAP does more than parallelize the loops. The entries in the column labeled "Number of Loops after Fusion" show how KAP reduced loop overhead by fusing as many loops together as it could. KAP fused the five loops in subroutine STRAIN into three loops and fused all nine loops in subroutine PRTAL.

Another example of KAP's optimization for an SMP system is that in the doubly nested loop cases, such as subroutine FORCE (see Figure 3), the KAP preprocessor automatically selects the outer loop for parallel execution. In contrast, a vector machine such as the CRAY-1 prefers the inner loop.

Because the kernels of DYNA3D code span multiple subroutines, cross compilation optimization is suggested. There are three ways to do this: inlining, interprocedural analysis, and directives specifying that the inner subroutines can be concurrentized.

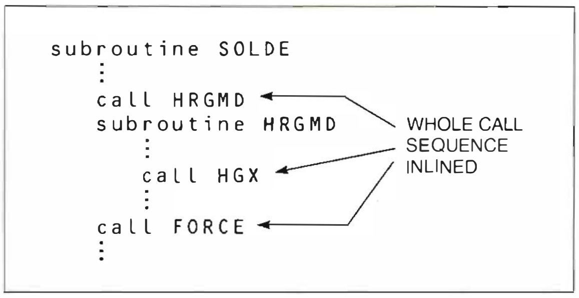


```
subroutine FORCE
:
do 60 n = 1,nnc
   lcn = lczc + n + nh12 - 1
   i0 = ia(lnc)
   i1 = ia(lcn + 1) - 1
cdir$ ivdep
   do 50 i = i0, i1
      e(1,ix(i)) =
      e(1,ix1(i)) + ep11(i)
      :
50 continue
   :
60 continue
```

Figure 3 Parallel Loop Selection

Using KAP's inlining capability gives KAP the most freedom to optimize the program because in this manner KAP can restructure code across subroutines.

Figure 4 shows part of the call sequence of subroutine SOLDE. (Subroutine SOLDE contains call



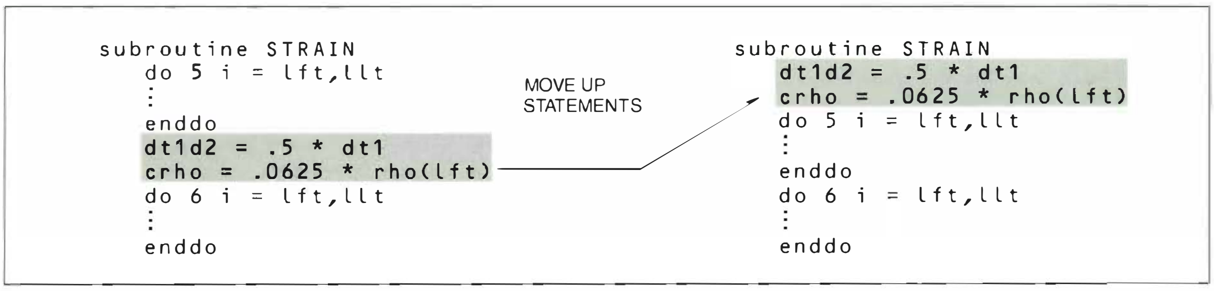
```
subroutine SOLDE
:
   call HRGMD
   subroutine HRGMD
      :
      call HGX
      :
   call FORCE
   :
```

Figure 4 Inlining a Kernel

sequence (b) of Table 4.) Subroutine SOLDE calls subroutine HRGMD which calls subroutine HGX. Then subroutine SOLDE calls subroutine FORCE. KAP supports inlining to an arbitrary depth. Inlining in KAP can be automatic or controlled from the command line. In this case, we did not want to enable inlining automatically to depth two of subroutine SOLDE because it contains calls to many other subroutines that are not in the kernel. Here, the user specified the subroutines to inline on the command line. When the user specified inlining, KAP fused all the loops in subroutines HRGMD, HGX, and FORCE to minimize loop overhead, and then it parallelized the fused loop.

In some cases, the user can make simple restructuring changes that improve KAP's optimizations. Figure 5 shows a case in which fusion was blocked by two scalar statements between a pair of loops. The first loop does not assign any values to the variables used to create these scalars, so the user can move the assignments above the loop to enable KAP to fuse them.

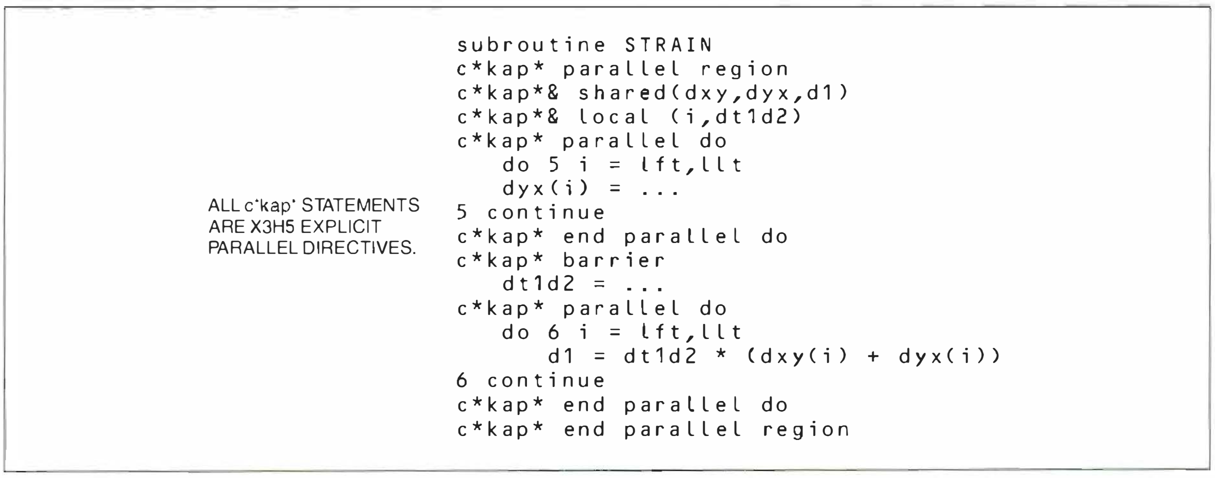
Finally, the user can elect to specify the parallelism directly. Figure 6 shows subroutine STRAIN with X3H5 directives used to describe the



```
subroutine STRAIN
   do 5 i = lft,llt
   :
   enddo
   dt1d2 = .5 * dt1
   crho = .0625 * rho(lft)
   do 6 i = lft,llt
   :
   enddo
```

```
subroutine STRAIN
   dt1d2 = .5 * dt1
   crho = .0625 * rho(lft)
   do 5 i = lft,llt
   :
   enddo
   do 6 i = lft,llt
   :
   enddo
```

Figure 5 Assisted Loop Fusion



```
subroutine STRAIN
c*kap* parallel region
c*kap*& shared(dxy,dyx,d1)
c*kap*& local (i,dt1d2)
c*kap* parallel do
   do 5 i = lft,llt
   dyx(i) = ...
5 continue
c*kap* end parallel do
c*kap* barrier
   dt1d2 = ...
c*kap* parallel do
   do 6 i = lft,llt
      d1 = dt1d2 * (dxy(i) + dyx(i))
6 continue
c*kap* end parallel do
c*kap* end parallel region
```

ALL c*kap* STATEMENTS ARE X3H5 EXPLICIT PARALLEL DIRECTIVES.

Figure 6 X3H5 Explicit Parallelism

parallelism. In this case, the user elected to keep the same unfused loop structure as in the original code. This case is not dramatically less efficient than the fused version because the parallel region causes KAP to fork threads only once.

A very sophisticated example of KAP usage occurs when a user inputs a program to KAP that has already been optimized by KAP. This is an advantage of a preprocessor that does not apply to a compiler because a preprocessor produces source code output. In this case, the statements shown in Figure 6 were generated by KAP to illustrate X3H5 parallelism. A user may want to perform some hand optimization on this output, such as removing the barrier statement, and then optimize the modified program with KAP again.

Challenges That Remain

Although the KAP preprocessor is a robust tool that performs well in a production software development environment, several challenges remain. Among them are adding new languages, further enhancing the optimization technology, and improving KAP's everyday usability.

As the popular programming languages evolve, KAP evolves also. KAI will soon extend KAP support for DEC Fortran to Fortran 90 and is developing C++ optimization capabilities.

In optimization technology, KAI's goal is to make an SMP server as easy to use as a single-processor workstation is today. "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing" contains a leading-edge analysis of parallelization technology.[10] The research reported shows that further developing current techniques can improve optimization technology. These techniques frequently involve the grand challenge of compiler optimization—whole program analysis.

In a much more pragmatic direction, the KAP preprocessor should be integrated with Digital's compiler technology at the intermediate representation level. Such integration would increase processing efficiency because the compiler would not have to reparse the source code. In addition, integration would increase the coordination between KAP and the compiler to improve performance for the end user.

Increasing the usability of the KAP preprocessor, however, benefits the end user directly. KAP engineers frequently talk to beta users and encourage feedback. The following are examples of user comments:

- Optimizing programs is difficult when no subroutine in the program takes more than a few percent of the run time. As its usability in this area improves, KAP will become a substantial productivity aid. If a program is generally slow, optimizing repeated usage patterns will allow the programmer to use a comfortable programming style and still expect peak system performance.

- Increasing feedback to the user would improve KAP's usability. When KAP cannot perform an optimization, often the user can help in several ways (e.g., by providing more information at compile time, by changing the options or directives, or by making small changes to the source code). KAP does not always make it clear to the user what needs to be done. Providing such feedback would improve KAP's usability.

- Integration with other performance tools would be useful. Alpha systems have a good set of performance monitoring tools that can provide clues about what to optimize in a program and how. The next release of the KAP preprocessor will provide some simple tools that a user can employ to integrate KAP with tools like prof and to track down performance differences.

On a final note, the fact that KAP does not speed up a program should not always be cause for disappointment. Some programs already run as fast as possible without the benefit of a KAP preprocessor.

Acknowledgments

We wish to acknowledge the Lawrence Livermore National Laboratory Methods Development Group and other users for providing applications that give us insight into how to improve the KAP preprocessor. We would like to thank those at Digital who have been instrumental in helping us deliver KAP on the DEC OSF/1 platform, especially Karen DeGregory, John Shakshober, Dwight Manley, and Dave Velten. Everyone at Kuck & Associates participated in the making of this product but of special note are Mark Byler, Debbie Carr, Ken Crawford, Steve Healey, David Nelson, and Sree Simhadri.

References

1. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 121-136.

2. M. Wolfe, *Optimizing Supercompilers for Supercomputers* (Cambridge, MA: MIT Press, 1989).

3. *Parallel Processing Model for High Level Programming Languages,* ANSI X3H5 Document Number X3H5/94-SD2, 1994.

4. P. Tu and D. Padua, "Automatic Array Privatization," *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing,* vol. 768 of *Lecture Notes in Computer Science* (New York: Springer-Verlag, 1993): 500-521.

5. B. Maskas et al., "Design and Performance of the DEC 4000 AXP Departmental Server Computing System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 82-99.

6. R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems,* vol. 9, no. 4 (October 1987): 491-542.

7. U. Banerjee, *Dependence Analysis for Supercomputing* (Norwell, MA: Kluwer Academic Publishers, 1988).

8. *KAP for DEC Fortran for DEC OSF/1 AXP User Guide* (Maynard, MA: Digital Equipment Corporation, 1994).

9. *KAP for C for DEC OSF/1 AXP User Guide* (Maynard, MA: Digital Equipment Corporation, 1994).

10. W. Blume et al., "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," CSRD Report No. 1348 (Urbana, IL: Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1994).

Further Readings

The Digital Technical Journal *publishes papers that explore the technological foundations of Digital's major products. Each* Journal *focuses on at least one product area and presents a compilation of refereed papers written by the engineers who developed the products. The content for the* Journal *is selected by the Journal Advisory Board. Digital engineers who would like to contribute a paper to the* Journal *should contact the editor at RDVAX::BLAKE.*

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

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

Distributed Systems
Vol. 1, No. 9, June 1989, EY-C179E-DP

Storage Technology
Vol. 1, No. 8, February 1989, EY-C166E-DP

CVAX-based Systems
Vol. 1, No. 7, August 1988, EY-6742E-DP

Software Productivity Tools
Vol. 1, No. 6, February 1988, EY-8259E-DP

VAXcluster Systems
Vol. 1, No. 5, September 1987, EY-8258E-DP

VAX 8800 Family
Vol. 1, No. 4, February 1987, EY-6711E-DP

Networking Products
Vol. 1, No. 3, September 1986, EY-6715E-DP

MicroVAX II System
Vol. 1, No. 2, March 1986, EY-3474E-DP

VAX 8600 Processor
Vol. 1, No. 1, August 1985, EY-3435E-DP

Subscriptions and Back Issues

Subscriptions to the *Digital Technical Journal* are available on a prepaid basis. The subscription rate is $40.00 (non-U.S. $60.00) for four issues and $75.00 (non-U.S. $115.00) for eight issues. Orders should be sent to Cathy Phillips, Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460, U.S.A., Telephone: (508) 486-2538, FAX: (508) 486-2444. Inquiries can be sent electronically to dtj@digital.com. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are available on the Internet at http://www.digital.com/info/DTJ/home.html. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital Research Laboratory Reports

Reports published by Digital's research laboratories can be accessed on the Internet through the World Wide Web or FTP. For access information on the electronic or hard-copy versions of the reports, see http://gatekeeper.dec.com/hypertext/info/cra.reports.html.

Digital Product Information

Readers of the *Journal* can keep up-to-date on Digital's products and services by subscribing to the *Digital Reference Service.* To receive current information on all Digital's products and services on a regular basis, contact the Digital Reference Service, P.O. Box 6464, Holliston, MA 01746. Within the United States, call (800) 494-4377 or (508) 429-5515, extension 765. Outside the United States, call (508) 429-3015 or send a facsimile to (508) 429-6921.

Technical Books and Papers by Digital Authors

Applications of Petri Nets in Manufacturing Systems Modeling, Control, and Performance Analysis, Alan A. Desrochers and Robert Y. Al-Jaar, IEEE Press, New York, 1994 (ISBN 0-87942-295-5).

This practical, highly specialized book presents theory and examples that clearly show how to use the Petri net approach to model, control, and then analyze the performance of these complex systems. This book also brings together newly documented applications of Petri nets in Japan and Europe and makes them available to practitioners worldwide.

From this book, the reader will learn how to model complex manufacturing systems using Petri nets; analyze the performance of the overall manufacturing system in terms of production rates, machine utilization, average in-process inventory, and other measures; generate control software from the Petri net model of an automated manufacturing system; and synthesize Petri net models for large automated manufacturing systems.

Applications of Petri Nets in Manufacturing Systems Modeling, Control, and Performance Analysis will be of particular interest to researchers in manufacturing systems engineering and individuals involved in production planning and control, plant layout and design, and scheduling of manufacturing operations.

R. Abugov and K. Zinke, "Prioritization of Defect Reduction Activity by Yield Impact," *1994 SEMICON Ultraclean Manufacturing Conference* (October 1994).

M. Ackerman and R. Buckland, "Multiple Matrices for a Marketing QFD," *Fifth Symposium on Quality Function Deployment* (June 1994).

M. Ackerman and R. Buckland, "Successful QFD Application at Digital: Unique Approaches and Applications of QFD to Address Business Needs," *Fifth Symposium on Quality Function Deployment* (June 1994).

H. Ali, J. Steele, J. Bosco, and G. Bartlett, "Electromechanical Study of 'No-Clean' Flux Corrosivity," *Proceedings of the Eighth Electronic Materials and Processing Congress* (August 1993).

R. Allmon, "Design of Portable Systems," *Proceedings of the IEEE Custom Integrated Circuits Conference* (May 1994).

P. Anick, "Adapting a Full-text Information Retrieval System to the Computer Troubleshooting Domain," *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval* (July 1994).

N. Arora and B. Doyle, "Modeling the I-V Characteristics of Fully Depleted SOI MOSFETs Including Self Heating," *IEEE International Silicon-on-Insulator Conference Proceedings* (October 1994).

N. Arora, B. Doyle, and D. Krakauer, "SPICE Model and Parameters for Fully Depleted SOI MOSFETs, Including Self-heating," *IEEE Electron Device Letters* (October 1994).

N. Arora, R. Rios, and C. Huang, "Impact of Polysilicon Depletion Effect on Circuit Performance for 0.35μ CMOS Technology," *Proceedings of the Twenty-fourth European Solid State Device Research Conference* (September 1994).

D. Bhavsar and J. Edmondson, "Testability Strategy of the Alpha AXP 21164 Microprocessor," *Proceedings of the IEEE International Test Conference* (October 1994).

S. Bilotta and D. Proctor, "Development of a Manufacturable Low Pressure ROXNOX Oxidation Process," *Advanced Semiconductor Manufacturing Conference and Workshop Proceedings* (November 1994).

C. Brench and B. Archambeault, "Proposed Standard EMI Modeling Problems," *IEEE International Symposium on Electromagnetic Compatibility* (August 1994).

C. Brench, "Heatsink Radiation as a Function of Geometry," *IEEE International Symposium on Electromagnetic Compatibility* (August 1994).

C. Brench, "Shield Degradation in the Presence of External Conductors," *IEEE International Symposium on Electromagnetic Compatibility* (August 1994).

J. Clement and A. Enver, "Modeling Electromigration-induced Stress Buildup Due to Nonuniform Temperature," *Materials Reliability in Microelectronics IV Symposium Proceedings* (April 1994).

W. Cronin, J. Hutchison, K. Ramakrishnan, and H. Yang, "A Comparison of High-speed LANs," *Proceedings of the IEEE Nineteenth Conference on Local Computer Networks* (October 1994).

W. Dubie, "Networds: The Impact of Electronic Text-Processing Utilities on Writing," *Journal of Social and Evolutionary Systems* (November 1994).

J. Edmondson and P. Rubinfeld, "An Overview of the 21164 Alpha AXP Microprocessor," *Hot Chips VI Symposium* (August 1994).

T. Fox, "The Design of High-Performance Microprocessors at Digital," *Thirty-first Design Automation Conference Proceedings* (June 1994).

J. Grodstein, E. Lehman, H. Harkness, and W. Grundmann, "Optimal Latch Mapping and Retiming within a Tree," *IEEE/ACM International Conference on Computer-aided Design* (November 1994).

C. Gross, "Method for Selecting Semiconductor Equipment Using Empowered Teams," *Advanced Semiconductor Manufacturing Conference and Workshop Proceedings* (November 1994).

T. Guay, "CASE-based Reasoning for Knowledge Acquisition Suggestions," *International Journal of Artificial Intelligence Tools* (July 1994).

S. Jong, "Exploring Paths toward Quality Information," *Forty-first Annual Society for Technical Communication* (May 1994).

C. Juszczak and D. Lebel, "NFS Version 3—Design and Implementation," *Summer 1994 USENIX Technical Conference* (June 1994).

D. Krakauer and K. Mistry, "Circuit Interactions During Electrostatic Discharge," *IEEE Electrical Over Stress/Electrostatic Discharge Symposium Proceedings* (September 1994).

J. Lloyd, "Electromigration Failure of Narrow Al Alloy Conductors Containing Stress Voids," *Materials Reliability in Microelectronics IV Symposium Proceedings* (April 1994).

J. Lloyd, "Electromigration Failure in Thin Film Conductors," *Materials Reliability in Microelectronics IV Symposium Proceedings* (April 1994).

P. Martino and G. Freedman, "Predicting Solder Joint Shape by Computer Modeling," *Proceedings of the Forty-fourth Electronic Components and Technology Conference* (May 1994).

T. Moore, "A Test Process Optimization and Cost Modeling Tool," *Proceedings of the IEEE International Test Conference* (October 1994).

D. Morin, T. Comard, M. Joshi, and K. Sprague, "Calculating Error of Measurement on High-speed Microprocessor Test," *Proceedings of the IEEE International Test Conference* (October 1994).

G. Papadeas and D. Gauthier, "An On-line Data Collection and Analysis System for VLSI Devices at Wafer Probe and Final Test," *Proceedings of the IEEE International Test Conference* (October 1994).

M. Piasecki, K. Orvek, R. Jones, and S. Dass, "Deep UV Technology for 0.35μm Lithography," *1994 IEEE Lithography Workshop* (September 1994).

K. Ramakrishnan and P. Biswas, "Performance Benefits of Nonvolatile Caches in Distributed File Systems," *Concurrency: Practice and Experience* (July 1994).

K. Ramakrishnan and H. Yang, "The Ethernet Capture Effect: Analysis and Solution," *Proceedings of the IEEE Nineteenth Conference on Local Computer Networks* (October 1994).

K. Ramakrishnan and H. Yang, "FIFO Design for a High-speed Network Interface," *Proceedings of the IEEE Nineteenth Conference on Local Computer Networks* (October 1994).

R. Razdan and K. Brace, "PRISC Software Acceleration Techniques," *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (October 1994).

A. Sathaye, "Application of Supervisor Synthesis for Controlled Time Petri Nets to Real-time Database Systems," *1994 American Control Conference* (June 1994).

S. Sathaye, "Conventional and Early Token Release Scheduling Models for the IEEE 802.5 Token Ring," *Journal of Real-Time Systems* (May 1994).

S. Sathaye, "A Real-time Scheduling Framework for Packet-switched Networks," *Fourteenth International Conference on Distributed Computing Systems* (June 1994).

C. Schiebl, "Application of EDX Spectroscopy to Accurate Nondestructive Measurement of CoSi Film Thicknesses during Semiconductor Processing," *Twenty-eighth Annual Microbeam Analysis Society Meeting* (August 1994).

C. Schiebl, "Continuous Fluorescence Correction Factor for Layered Specimen," *Twenty-eighth Annual Microbeam Analysis Society Meeting* (August 1994).

C. Schiebl, "Secondary Depth Distribution Generated by Characteristic Fluorescence in Multilayer Samples for Use in Quantitative EPMA," *Twenty-eighth Annual Microbeam Analysis Society Meeting* (August 1994).

J. Seyyedi, "Soldered Joint Reliability for Interstitial Pin Grid Array Packages," *Journal of Surface Mount and Related Technologies Group* (October 1994).

H. Soleimani, "An Investigation of Phosphorous Transient Diffusion in Silicon below the Solid Solubility Limit and at a Low Implant Energy," *Journal of the Electrochemical Society* (August 1994).

K. Steeples and D. Chang Kau, "Multiply Charged, Channeled, Ion Implantation," *Tenth International Conference on Ion Implantation Technology* (June 1994).

K. Steeples, D. Chang Kau, M. Andreoli, and K. Mistry, "Rapid Implementation of a LATID Process," *Tenth International Conference on Ion Implantation Technology* (June 1994).

N. Sullivan and S. Arsenault, "SEM/EDS Analysis Method for Bare Silicon Particle Monitor Wafers," *Advanced Semiconductor Manufacturing Conference and Workshop Proceedings* (November 1994).

N. Sullivan and R. Newcomb, "Critical Dimension Measurement in the SEM: Comparison of Backscattered vs. Secondary Electron Detection," *Proceedings of the International Society of Photo-Optical Instrumentation Engineers (SPIE): Integrated Circuit Metrology, Inspection, and Process Control VIII* (February 1994).

B. Thomas, "OpenVMS I/O Concepts: Kernel Processes," *Digital Systems Journal* (July 1994).

B. Thomas, "OpenVMS I/O Concepts: Software," *Digital Systems Journal* (July 1994).

B. Thomas and K. Morse, "OpenVMS AXP I/O Concepts," *Digital Systems Journal* (June 1994).

A. Torabi, M. Mallary, and S. Marshall, "The Effect of Rise Time and Field Gradient on Nonlinear Bit Shift in Thin Film Heads," *The Sixth Joint MMM-Intermag Conference* (June 1994).

A. Torabi, M. Mallary, S. Marshall, S. Batra, and S. Ramaswamy, "Performance Evaluation of Different Pole Geometries in Thin Film Heads," *The Sixth Joint MMM-Intermag Conference* (June 1994).

M. Tsuk, "FASTHENRY: A Multipole-accelerated 3-D Inductance Extraction Program," *IEEE Transactions on Microwave Theory and Techniques* (September 1994).

M. Tsuk and R. Evans, "Modeling and Measurement of the Power Distribution System of a High-performance Computer System," *IEEE Topical Meeting on Electrical Performance of Electronic Packaging* (October 1993).

R. Ulichney, "Halftone Characterization in the Frequency Domain," *The Society for Imaging Science and Technology's (IS&T's) Forty-seventh Annual Conference* (May 1994).

R. Ulichney, "The Void-and-cluster Method for Dither Array Generation," *Proceedings of the International Society of Photo-Optical Instrumentation Engineers (SPIE)* (September 1993).

M. Utt, "A System for Discovering Relationships by Feature," *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval* (July 1994).

J. Vicente, "Network Capacity Planning," *CMG '93 Conference* (December 1993).

A. Villani, "Cohesive Mechanical Behavior of Adhesive Materials," *Proceedings of the 1993 ASME International Electronics Packaging Conference: Advances in Electronic Packaging 1993* (October 1993).

J. Yang, "Reliability Performance of an R3000-Based MCM for Desktop Workstations," *International Electronics Packaging Conference* (September 1993).

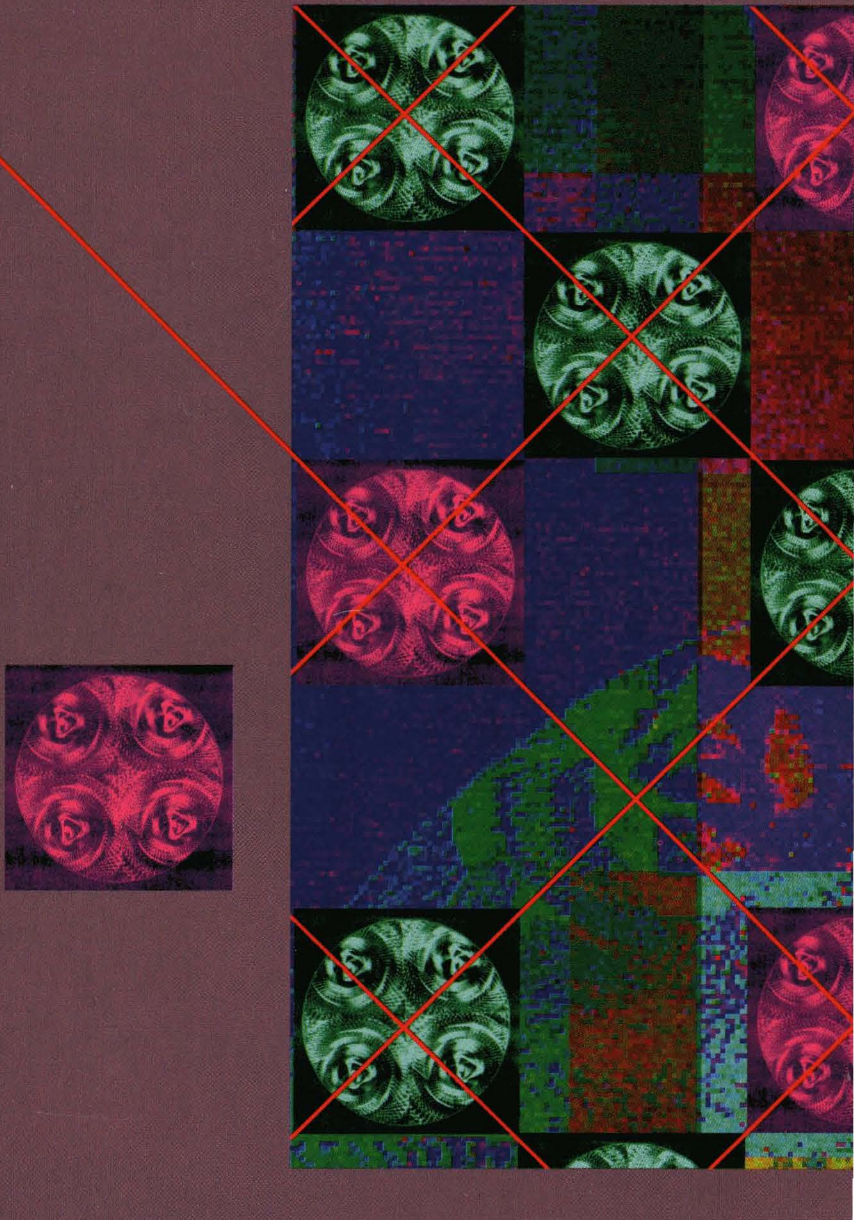
W. Zahavi, "Modeling the Performance Budget," *Computer Measurement Group Proceedings (CMG '93)* (September 1993).

Recent Digital U.S. Patents

The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied to us by the U.S. Patent and Trademark Office are reproduced exactly as they appear on the original published patent.

4,592,072	R.E. Stewart	Decoder for Self-Clocking Serial Data Communications
5,210,834	W. Beach and J. Zurawski	High Speed Transfer of Instructions from a Master to a Slave Process
5,210,874	P. Karger	Cross-Domain Call System in a Capability Based Digital Data Processing System
5,212,650	D. Hooper and S. Kundu	Procedure and Data Structure for Synthesis and Transformation of Logic Circuit Designs
5,212,783	S. Sherman	System Which Directionally Sums Signals for Identifying and Resolving Timing Inconsistencies
5,214,770	R. Ramanujan, P. Bannon, and S. Steely	System for Flushing Instruction-Cache Only When Instruction-Cache Address and Data-Cache Address Are Matched and the Execution of a Return-from-Execution-or-Interrupt Command
5.216,413	L. Seiler, J. Pappas, and R. Rose	Apparatus and Method for Specifying Windows with Priority Ordered Rectangles in a Computer Video Graphics System
5,218,684	D. Hayes and V. Triolo	Memory Configuration System
5,218,712	D. Bhandarkar, W. Cardoza, D. Cutler, D. Orbits, and R. Witek	Providing a Data Processor with a User-mode Accessible Mode of Operations in Which the Processor Performs Processing Operations without Interruption
5,220,468	M. Sidman	Disk Drive with Constant Bandwidth Automatic Gain Control
5,220,674	W. Morgan, D. Cobb, G. Bell, and A. Carlson	Local Area Print Server for Requesting and Storing Required Resource Data and Forwarding Printer Status Message to Selected Destination
5,221,422	S. Das and J. Khan	Lithographic Technique Using Laser Scanning for Fabrication of Electronic Components and the Like
5,222,029	D. Hooper and S. Kundu	Bitwise Implementation Mechanism for a Circuit Design Synthesis Procedure
5,222,223	R. Hetherington, D. Webb, T. Fossum, J. Murray, and D. Manley	Method and Apparatus for Ordering and Queueing Multiple Memory Requests
5,222,224	S. Arnold, S. Delahunt, M. Flynn, T. Fossum, R. Hetherington, and D. Webb	Scheme for Insuring Data Consistency between a Plurality of Cache Memories and the Main Memory in a Multiprocessor System
5,226,170	P. Rubinfeld	Interface between Processor and Special Instruction Processor in Digital Data Processing System
5,228,129	S. Bryant and M. Harwood	Synchronous Communication Interface for Reducing the Effect of Data Processor Latency
5,230,067	B. Buch	Bus Control Circuit for Latching and Maintaining Data Independently of Timing Event on the Bus Until New Data Is Driven Onto
5,230,071	B. Newman	Method of Controlling the Variable Baud Rate of Peripheral Devices
5,230,072	D. Smith and K. O'Rourke	System for Managing Hierarchical Information in a Digital Data Processing System
5,230,079	R. Grondalski	Massively Parallel Array Processing System with Processors Selectively Accessing Memory Module Locations Using Address in Microword or in Address Register

5,235,693	J. Lynch, K. Chinnaswamy, P. Goodwin, J. Tessari, and M. Gagliardo	Method and Apparatus for Reducing Buffer Storage in a Read-Modify-Write Operation
5,237,574	L. Weng	Error-resilient Information Encoding
5,239,635	R.E. Stewart, T.E. Leonard, and S.T. Lee	Virtual Address to Physical Address Translation Using Page Tables in Virtual Memory
5,247,398	M. Sidman	Automatic Correction of Position Demodulator Offsets
5,249,187	W. Bruckert and T. Bissett	Dual Rail Processors with Error Checking on I/O Reads
5,249,293	B. Schreiber, C. Cockcroft, M. Ozur, R. Bismouth, and D. Doherty	Computer Network Providing Transparent Operation on a Compute Server and Associated Method
5,251,322	P. Doyle, J. Ellenberger, E. Jones, D. Carver, S. DiPirro, B. Gerovac, W. Armstrong, E. Gibson, R. Shapiro, K. Rushforth, and W.C. Roach	Method of Operating a Computer Graphics System Including Asynchronously Traversing Its Nodes
5,255,367	W. Bruckert, T. Bissett, D. Mazur, and J. Munzer	Fault Tolerant, Synchronized Twin Computer System with Error Checking of I/O Communication
5,261,113	N. P. Jouppi	Apparatus and Method for a Single Operand Register Array for Vector and Scalar Data Processing Operations
5,266,409	P.H. Schmidt and J.C. Angus	Hydrogenated Carbon Compositions
5,267,175	D. Hooper	Database Access Mechanism for Rules Utilized by a Synthesis Procedure for Logic Circuit Design
5,276,892	A.S. Olesin and R.M. Supnik	Destination Control Logic for Arithmetic and Logic Unit for Digital Data Processor
5,278,840	D. Bhandarkar, W. Cardoza, D. Cutler, D. Orbits, and R. Witek	Apparatus and Method for Data Induced Condition Signalling
5,280,617	R.F. Brender and B.R. Brett	Automatic Program Code Generation in a Compiler System for an Instantiation of a Generic Program Structure and Based on Formal Parameters and Characteristics of Actual Parameters
5,287,463	R.C. Frame and F.A. Zayas	Method and Apparatus for Transferring Information over a Common Parallel Bus Using a Fixed Sequence of Bus Phase Transitions
5,291,581	D. Bhandarkar, W. Cardoza, D. Cutler, D. Orbits, and R. Witek	Apparatus and Method for Synchronization of Access to Main Memory Signal Groups in a Multiprocessor Data Processing System
5,297,283	D. Cutler, J. Kelly, and F. Perazzoli	Object Transferring System and Method in an Object Based Computer Operating System
5,301,329	R.C. Frame and F.A. Zayas	Double Unequal Bus Timeout
5,303,380	B. Foster, G. Brown, J. Piazza, J. Tenny, B. Nelson, W. Van Roggen, and P. Anagnostopoulos	System for Processing Data to Facilitate the Creation of Executable Images
5,305,462	R. Grondalski	Mechanism for Broadcasting Data in a Massively Parallel Array Processing System
5,313,467	G. Varghese, M. Fine, A. Smith, and R. Szmauz	Integrated Communication Link Having Dynamically Allocatable Bandwidth and Protocol for Transmission of Allocation Information over the Link
5,317,717	D. Bhandarkar, W. Cardoza, D. Cutler, D. Orbits, and R. Witek	Apparatus and Method for Main Memory Unit Protection Using Access and Fault Logic Signals



ISSN 0898-901X