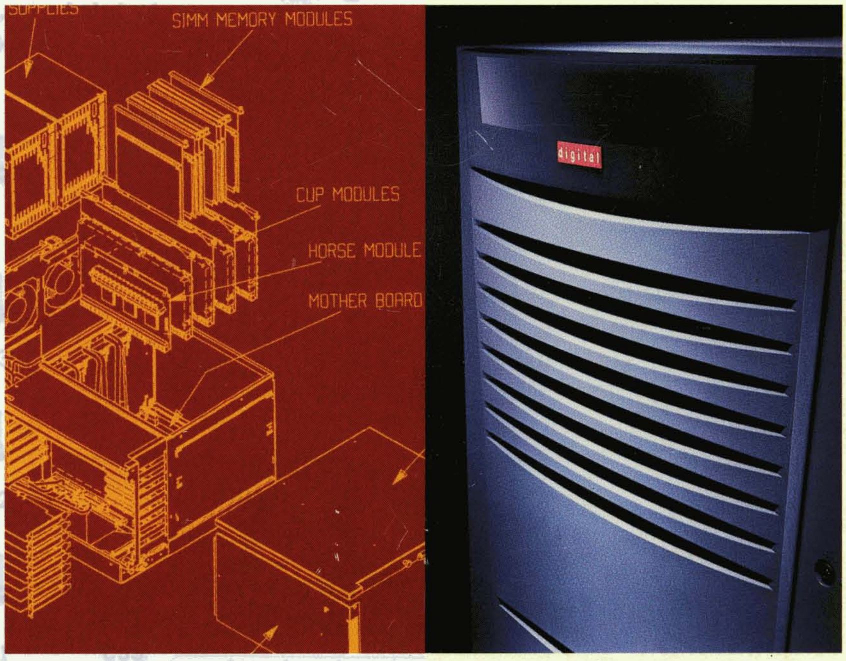
Digital Technical Journal

digital™

ALPHASERVER 4100 SYSTEM

ORACLE AND SYBASE DATABASE PRODUCTS FOR VLM

INSTRUCTION EXECUTION ON ALPHA PROCESSORS



Volume 8 Number 4
1996

Cover Design

The performance advantage of very large memory technology for commercial applications is a major theme in this issue of the *Journal*. The cover is a collage of images from the development of the AlphaServer 4100 four-processor symmetric multiprocessing system, which offers 8 gigabytes of memory and industry leadership performance. This four-processor symmetric multiprocessing system is not only characterized by very large memory but by low latency, high bandwidth, and 400-megahertz microprocessors.

The cover design is by Lucinda O'Neill of DIGITAL's Corporate Design Group.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 50 Nagog Park, AKO2-3/B3, Acton, MA 01720-9843.

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

Electronic subscriptions are available at no charge by accessing URL **http://www.digital.com/info/subscription.** This service will send an electronic mail notification when a new issue is available on the Internet.

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

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

Inquiries, address changes, and complimentary subscription orders can be sent to the *Digital Technical Journal* at the published-by address or the electronic mail address, **dtj@digital.com.** Inquiries can also be made by calling the *Journal* office at 508-264-7549.

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

The information in the *Journal* is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation or by the companies herein represented. Digital Equipment Corporation assumes no responsibility for any errors that may appear in the *Journal*.

ISSN 0898-901X

Documentation Number EC-N7629-18

Book production was done by Quantic Communications, Inc.

Contents

ALPHASERVER 4100 SYSTEM

ORACLE AND SYBASE DATABASE PRODUCTS FOR VLM

INSTRUCTION EXECUTION ON ALPHA PROCESSORS

Editor's Introduction

Just 40 years ago, a machine called the TX-0—a successor to Whirlwind—was built at MIT's Lincoln Laboratory to find out, among other things, if a core memory as large as 64 Kwords could be built. Over the years memory sizes have grown so large that, in the '90s, the industry has felt the need to characterize memory in big machines as *very* large. At five orders of magnitude greater in size than the TX-0 memory, the AlphaServer 4100 8-gigabyte memory is indeed very large, even by today's standards. Whole databases can be designed to reside in memory. Very large memory technology, or VLM, is a key to the system and application performance discussed in this issue of the *Journal,* which features the AlphaServer 4100 system, database enhancements from Oracle Corporation and from Sybase, Inc., and extensions to the Alpha architecture.

The AlphaServer 4100 is a mid-range, symmetric multiprocessing system designed for industry-leading performance at a low cost. The system accommodates up to four 64-bit Alpha 21164 microprocessors operating at 400 megahertz, four 64-bit PCI bus bridges, and 8 gigabytes of main memory. Opening the section about the 4100 system, Zarka Cvetanovic and Darrel Donaldson describe the project team's performance characterization of different AlphaServer 4100 models under technical and commercial workloads. Both the process and the findings are of interest. As one example set of data demonstrates, the model 5/300 is not only faster than its DIGITAL predecessors but 30 to 60 percent faster than a comparative industry platform when running memory-intensive workloads from the SPECfp95 benchmark.

The four papers that follow examine areas of the system that challenged designers to keep costs low and at the same time deliver high performance. The AlphaServer 4100 cached processor module design is presented by Mo Steinman, George Harris, Andrej Kocev, Ginny Lamere, and Roger Pannell. Built around the Alpha 21164 64-bit RISC microprocessor, the module is the first from DIGITAL to employ a high-performance, cost-effective synchronous cache rather than a traditional asynchronous cache. Next, Roger Dame reviews the clock distribution system, the use of off-the-shelf phase-locked loop circuits as the basic building block to keep costs low, and the signal integrity techniques developed to optimize performance of the clock distribution system for a worst-case clock skew of 2.2 nanoseconds, a goal which the team far exceeded. A unique memory architecture for the model 5/300E is the subject of Glenn Herdeg's paper. This memory design incorporates a processor module that has no external cache and instead takes advantage of the multiple-issue feature of the Alpha 21164 microprocessor. Closing the section on the 4100 design is the I/O subsystem's contribution to the system goals of low latency and high memory and I/O bandwidth. Sam Duncan, Craig Keefer, and Tom McLaughlin present several innovative techniques developed for the system bus-to-PCI bus bridge design, including partial cache line writes, peer-to-peer transactions across PCI bridges, and support for large bursts of data.

All efforts to make the hardware run faster are for the benefit of the applications that run on those systems. A paper from Oracle Corporation and another from Sybase, Inc., examine ways in which their respective database systems take advantage of VLM. Vipin Gokhale describes the 64 Bit Option implementation for the Oracle7 relational database system. A primary project goal was to demonstrate a clear performance benefit for decision support systems and online transaction processing. The author summarizes data that show a clear benefit for a database with the 64 Bit Option enabled running on the AlphaServer 8400 with 8 gigabytes of memory; in some cases, the performance increase was 200 times that of the standard configuration. Sybase engineers T.K. Rengarajan, Max Berenson, Ganesan Gopal, Bruce McCready, Sapan Panigrahi, Srikant Subramaniam, and Marc Sugiyama examine the technology of the System 11 SQL Server that was specifically designed for VLM systems. In addition to achieving record results with the SQL Server running on the AlphaServer 8400, the engineers have laid the groundwork for future main memory database systems.

Recently, byte and word instructions were added to DIGITAL's 64-bit Alpha architecture. Dave Hunter and Eric Betts describe the process of analyzing how these additions affect the performance of a commercial database. For testing, the team used prototype hardware, rebuilt Microsoft Corporation's SQL Server to use the new instructions, and ran the TPC-B benchmark.

The editors thank Darrel Donaldson of the AlphaServer 4100 team and Kuk Chung of the Database Application Partners group for their efforts to acquire the papers presented in this issue. Our upcoming issue will feature CMOS-6 process technologies.

Jane Blake

Jane C. Blake
Managing Editor

Zarka Cvetanovic
Darrel D. Donaldson

AlphaServer 4100 Performance Characterization

The AlphaServer 4100 is the newest four-processor symmetric multiprocessing addition to DIGITAL's line of midrange Alpha servers. The DIGITAL AlphaServer 4100 family, which consists of models 5/300E, 5/300, and 5/400, has major platform performance advantages as compared to previous-generation Alpha platforms and leading industry midrange systems. The primary performance strengths are low memory latency, high bandwidth, low-latency I/O, and very large memory (VLM) technology. Evaluating the characteristics of both technical and commercial workloads against each family member yielded recommendations for the best application match for each model. The performance of the model with no module-level cache and the advantages of using 2- and 4-megabyte module-level caches are quantified. The profiles based on the built-in performance monitors are used to evaluate cycles per instruction, stall time, multiple-issuing benefits, instruction frequencies, and the effect of cache misses, branch mispredictions, and replay traps. The authors propose a time allocation–based model for evaluating the performance effects of various stall components and for predicting future performance trends.

The AlphaServer 4100 is DIGITAL's latest four-processor symmetric multiprocessing (SMP) midrange Alpha server. This paper characterizes the performance of the AlphaServer 4100 family, which consists of three models:[1-5]

1. AlphaServer 4100 model 5/300E, which has up to four 300-megahertz (MHz) Alpha 21164 microprocessors, each without a module-level, third-level, write-back cache (B-cache) (a design referred to as *uncached* in this paper)
2. AlphaServer 4100 model 5/300, which has up to four 300-MHz Alpha 21164 microprocessors, each with a 2-megabyte (MB) B-cache
3. AlphaServer 4100 model 5/400, which has up to four 400-MHz Alpha 21164 microprocessors, each with a 4-MB B-cache

The performance analysis undertaken examined a number of workloads with different characteristics, including the SPEC95 benchmark suites (floating-point and integer), the LINPACK benchmark, AIM Suite VII (UNIX multiuser benchmark), the TPC-C transaction processing benchmark, image rendering, and memory latency and bandwidth tests.[6-15] Note that both commercial (AIM and TPC-C) and technical/scientific (SPEC, LINPACK, and image rendering) classes of workloads were included in this analysis.

The results of the analysis indicate that the major AlphaServer 4100 performance advantages result from the following server features:

- Significantly higher bandwidth (up to 2.6 times) and lower latency compared to the previous-generation midrange AlphaServer platforms and leading industry midrange systems. These improvements benefit the large, multistream applications that do not fit in the B-cache. For example, the AlphaServer 4100 5/300 is 30 to 60 percent faster than the HP 9000 K420 server in the memory-intensive workloads from the SPECfp95 benchmark suite. (Note that all competitive performance data presented in this paper is valid as of the submission of this paper in July 1996. The references cited refer the reader to the literature and the appropriate Web sites for the latest performance information.)

- An expanded very large memory (VLM). The maximum memory size increased from 2 gigabytes (GB) to 8 GB without sacrificing CPU slots. This increase in memory size benefits primarily the commercial, multistream applications. For example, the AlphaServer 4100 5/300 server achieves approximately twice the throughput of the Compaq ProLiant 4500 server and 1.4 times the throughput of the AlphaServer 2100 on the AIM Suite VII benchmark tests.
- A 4-MB B-cache and a clock speed of 400 MHz in the AlphaServer 4100 5/400 system. The larger B-cache size and 33 percent faster clock resulted in a 30 to 40 percent performance improvement over the AlphaServer 4100 5/300 system.

The performance improvement from the larger B-cache increases with the number of CPUs. For example, the AlphaServer 4100 5/300 system with its 2-MB B-cache design performs 5 to 20 percent faster with one CPU and 30 to 50 percent faster with four CPUs than the uncached 5/300E system. The majority of workloads included in this analysis benefit from the B-cache; however, the uncached system outperforms the cached implementation by 10 to 20 percent for large applications that do not fit in the 2-MB B-cache.

The performance counter profiles, based on the built-in hardware monitors, indicate that the majority of issuing time is spent on single and dual issuing and that a small number of floating-point workloads take advantage of triple and quad issuing. The load/store instructions make up 30 to 40 percent of all instructions issued. The stall time associated with waiting for data that missed in the various levels of cache hierarchy accounts for the most significant portion of the time the server spends processing commercial workloads.

Memory Latency

Memory latency and bandwidth have been recognized as important performance factors in the early Alpha-based implementations.[16,17] Since CPU speed is increasing at a much higher rate than memory speed, the "memory wall" limitation is expected to become even more important in the future. Therefore, reducing memory latency and increasing bandwidth have been major design goals for the AlphaServer 4100 platform.[1] The AlphaServer 4100 achieved the lowest memory latency of all DIGITAL products based on the Alpha 21164 microprocessor and all multiprocessor products by leading industry vendors. The major benefits come from the simpler interface, the use of synchronous dynamic random-access memory (DRAM) chips (i.e., synchronous memory), and the lower fill time.[1,2] Figure 1 shows the measured memory load latency using the lmbench benchmark with a 512-byte stride.[10] In this benchmark, each load depends on the result from the previous load, and therefore latency is not a good measure of performance for systems that can have multiple outstanding loads. (AlphaServer 4100 systems can have up to two outstanding requests per CPU on the bus.) The lmbench benchmark data indicates that the AlphaServer 4100 has the lowest memory latency of all industry-leading reduced-instruction set computing (RISC) vendors' servers.

As shown in Figure 2, using a slightly different workload where there is no dependency between consecutive loads, the AlphaServer 4100 achieves even lower per-load latency, since the latency for the two consecutive loads can be overlapped. The plateaus in Figure 2 show the load latency at each of the following levels of cache/memory hierarchy: 8-kilobyte (KB) on-chip data cache (D-cache), 96-KB on-chip secondary instruction/data cache (S-cache), 2- and 4-MB off-chip B-caches (except for model 5/300E), and memory. The uncached AlphaServer 4100 5/300E achieves an 85 percent lower memory load latency than the previous-generation AlphaServer 2100. The AlphaServer 4100 5/300, with its 2-MB B-cache, increases memory latency 30 percent for load operations and 6 percent for store operations compared to the uncached 5/300E system because of the time spent checking for data in the B-cache. The synchronous memory shows one cycle lower latency than the asynchronous extended data out (EDO) DRAM (i.e., asynchronous memory), which results in 9 percent faster load operations and 5 percent faster store operations. Note that the cached AlphaServer 4100 and AlphaServer 8200 systems, which have the same clock speeds of 300 MHz, achieve comparable B-cache latency, while the memory latency for all AlphaServer 4100 systems is significantly lower than on both the AlphaServer 8200 and the AlphaServer 2100 systems. The latency to the B-cache in this test is lower on the AlphaServer 2100 than on the other AlphaServer systems due to 32-byte blocks (compared to 64-byte blocks in the 4100 and 8200 systems). Although not shown in this test, many applications can benefit from the larger cache block size. The 400-MHz AlphaServer 4100 system uses a 33 percent faster CPU and achieves 11 percent reduction in B-cache and memory latency compared to the 300-MHz AlphaServer 4100 system.

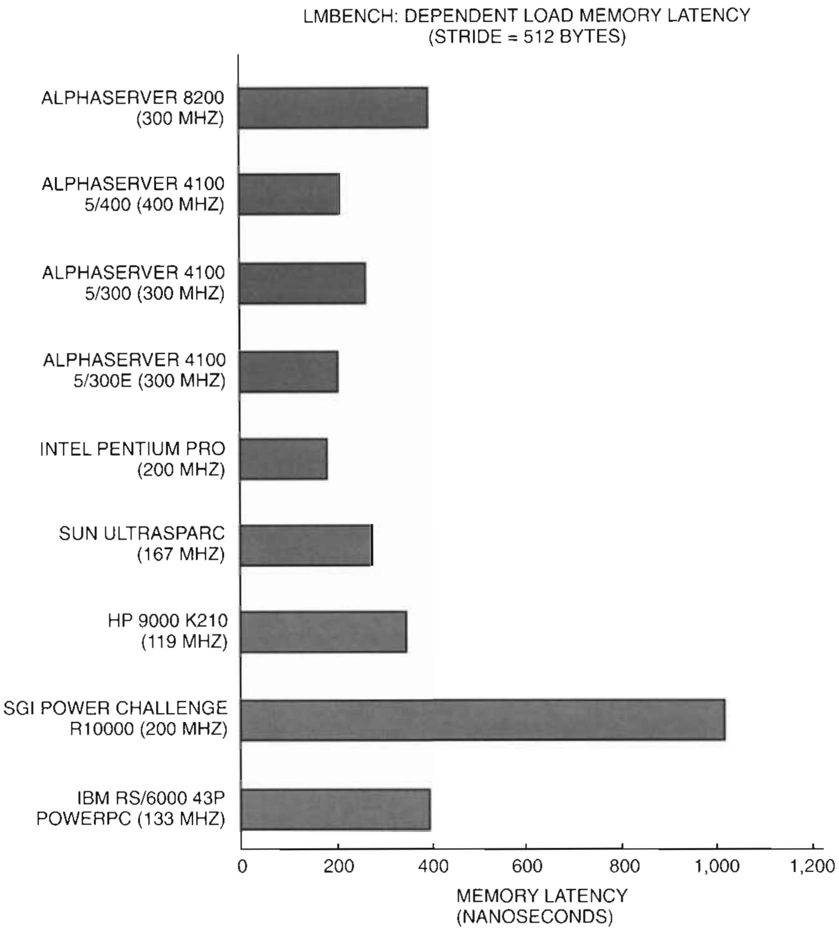


Figure 1
lmbench Benchmark Test Results Showing Memory Latency for Dependent Loads

Memory Bandwidth

The AlphaServer 4100 system bus achieves a peak bandwidth of 1.06 gigabytes per second (GB/s). The STREAM McCalpin benchmark measures sustainable memory bandwidth in megabytes per second (MB/s) across four vector kernels: Copy, Scale, Sum, and SAXPY.[11] Figure 3 shows measured memory bandwidth using the Copy kernel from the STREAM benchmark. Note that the STREAM bandwidth is 33 percent lower than the actual bandwidth observed on the AlphaServer 4100 bus because the bus data cycles are allocated for three transactions: read source, read destination, and write destination. The AlphaServer 4100 shows the best memory bandwidth of all multiprocessor platforms designed to support up to four CPUs. The platforms designed to support more than four CPUs (i.e., the AlphaServer 8400, the Silicon Graphics POWER CHALLENGE R10000, and the Sun Ultra Enterprise 6000 systems) show a higher bandwidth for four CPUs than the AlphaServer 4100. The STREAM bandwidth on the AlphaServer 4100 5/300 is 2.2 times higher than on the previous-generation AlphaServer 2100 5/250 (2.6 times higher with the AlphaServer 4100 5/400). The uncached AlphaServer 4100 model shows 22 percent higher memory bandwidth than the cached model 5/300.

The AlphaServer 4100 memory bandwidth improvement from synchronous memory compared to EDO ranges from 8 to 12 percent. The synchronous memory benefit increases with the number of CPUs, as shown in Table 1.

Low memory latency and high bandwidth have a significant effect on the performance of workloads that do not fit in 2- to 4-MB B-caches. For example, the majority of the SPECfp95 benchmarks do not fit in the 2-MB cache. (Figure 20, which appears later in this paper, shows the cache misses.) The SPECfp95 performance comparison presented in Figure 4 shows that the uncached AlphaServer 4100 5/300E system outperforms the 2-MB B-cache model 5/300 in the benchmarks with the highest number of B-cache misses (tomcatv, swim, applu, and hydro2d). The performance of the uncached model 5/300E is comparable to that of the 4-MB B-cache model 5/400 for the swim benchmark. However, the benchmarks that fit better in the 4-MB cache (apsi and wave5) run significantly slower on the 5/300E than on the 5/400.

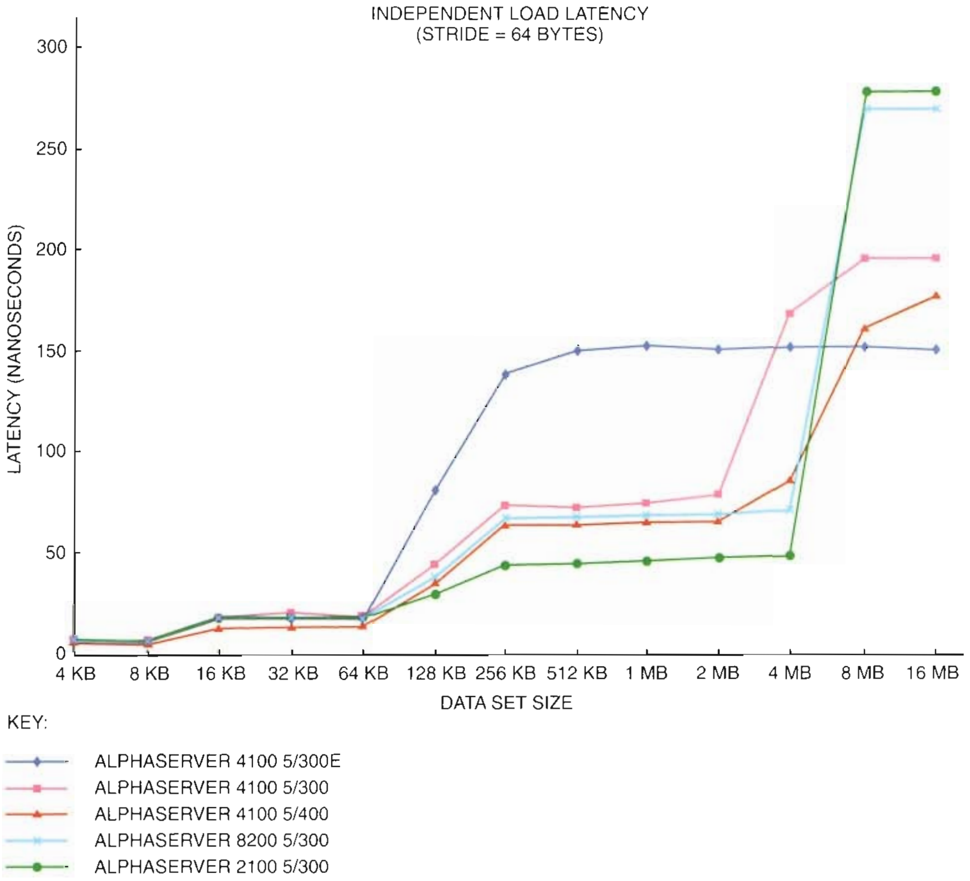


Figure 2
Cache/Memory Latency for Independent Loads

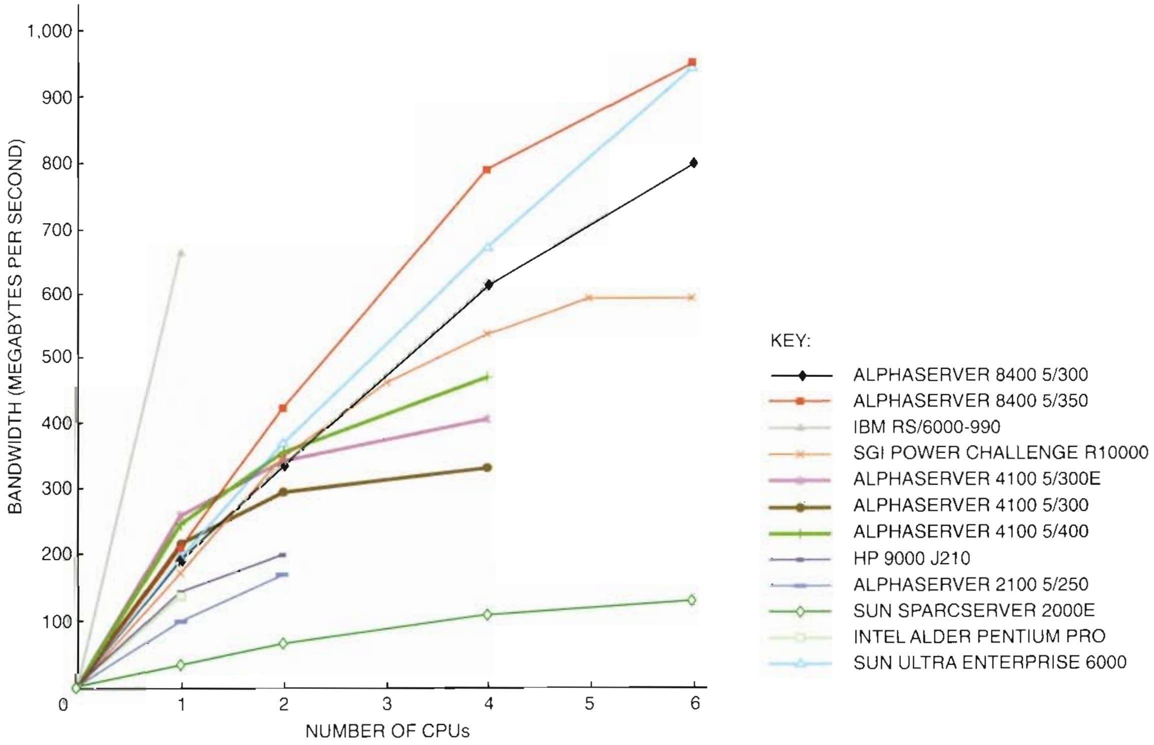


Figure 3
STREAM McCalpin Memory Copy Bandwidth Comparison

Table 1
Bandwidth Improvement from Synchronous Memory to Asynchronous Memory

	Number of CPUs			
	1	2	3	4
Bandwidth improvement	8%	8%	9%	12%

Figure 4 shows that the AlphaServer 4100 5/300 system has a significant (up to two times) performance advantage over the previous-generation AlphaServer 2100 system in the SPECfp95 benchmark tests with the highest number of B-cache misses. The SPECfp95 tests indicate that the 300-MHz AlphaServer 4100 is more than 50 percent faster than the HP 9000 K420 server, and the 400-MHz AlphaServer 4100 is twice as fast as the HP 9000 K420 in the SPECfp95 benchmarks that stress the memory subsystem.

SPEC95 Benchmarks

The SPEC95 benchmarks provide a measure of processor, memory hierarchy, and compiler performance. These benchmarks do not stress graphics, network, or I/O performance. The integer SPEC95 suite (CINT95) contains eight compute-intensive integer benchmarks written in C and includes the benchmarks shown in Table 2.[6,12]

The floating-point SPEC95 suite (CFP95) contains 10 compute-intensive floating-point benchmarks written in FORTRAN and includes the benchmarks shown in Table 3.[6,12]

The SPEC Homogeneous Capacity Method (SPEC95 rate) measures how fast an SMP system can perform multiple CINT95 or CFP95 copies (tasks). The SPEC95 rate metric measures the throughput of the system running a number of tasks and is used for evaluating multiprocessor system performance.

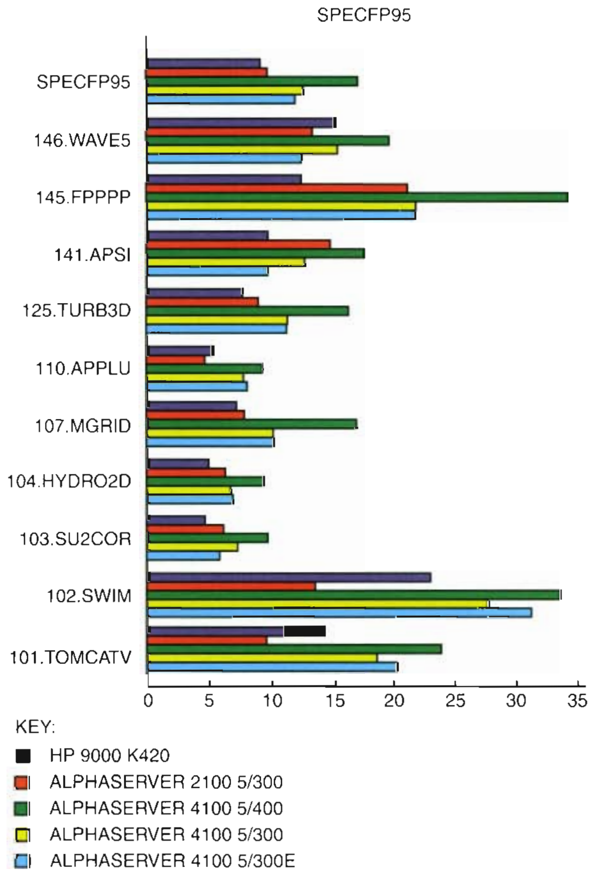


Figure 4
SPECfp95 Benchmarks Performance Comparison

Table 2
CINT95 Benchmarks (SPECint95)

Benchmark	Description
099.go	Artificial intelligence, plays the game of Go
124.m88ksim	A Motorola 88100 microprocessor simulator
126.gcc	A GNU C compiler that generates SPARC assembly code
129.compress	A program that compresses large text files (about 16 MB)
130.li	A LISP interpreter
132.ijpeg	A program that compresses/ decompresses an image
134.perl	A Perl interpreter that performs text and numeric manipulations
147.vortex	A database program that builds and manipulates three interrelational databases

Table 3
CFP95 Benchmarks (SPECfp95)

Benchmark	Description
101.tomcatv	A fluid dynamics mesh generation program
102.swim	A weather prediction shallow water model
103.su2cor	A quantum physics particle mass computation (Monte Carlo)
104.hydro2d	An astrophysics hydrodynamical Navier-Stokes equation
107.mgrid	A multigrid solver in a 3-D potential field (electromagnetism)
110.applu	Parabolic/elliptic partial differential equations (fluid dynamics)
125.turb3d	A program that simulates turbulence in a cube
141.apsi	A program that simulates temperature, wind, velocity, and pollutants (weather prediction)
145.fpppp	A quantum chemistry program that performs multielectron derivatives
146.wave5	A solver of Maxwell's equations on a Cartesian mesh (electromagnetics)

Figure 5 compares the SPEC95 performance of the AlphaServer 4100 systems to that of the other industry-leading vendors using published results as of July 1996. Figure 6 shows the same comparison in the multistream SPEC95 rates.[12] Note that all the SPEC95 comparisons in this paper are based on the peak results that include extensive compiler optimizations.[12] Figure 5 indicates that even the uncached AlphaServer 4100 5/300E performs better than the HP 9000 K420 system, and the AlphaServer 4100 5/400 shows approximately a two times performance advantage over the HP system. The AlphaServer 4100 5/300 SPECint95 performance exceeds that of the Intel Pentium Pro system, and the AlphaServer 4100 5/300 SPECfp95 performance is double that of the Pentium Pro. The AlphaServer 4100 5/400 is 1.5 times (SPECint95) and 2.5 times (SPECfp95) faster than the Pentium Pro system. The multiple-processor SPECfp95 on the AlphaServer 4100 is obtained by decomposing benchmarks using the KAP preprocessor from Kuck & Associates. Note that the cached four-CPU AlphaServer 4100 5/300 outperforms the Sun Ultra Enterprise 3000 with six CPUs in the SPECfp95 parallel test. The performance benefit of the cached versus the uncached AlphaServer 4100 is greater in multiprocessor configurations than in uniprocessor configurations.

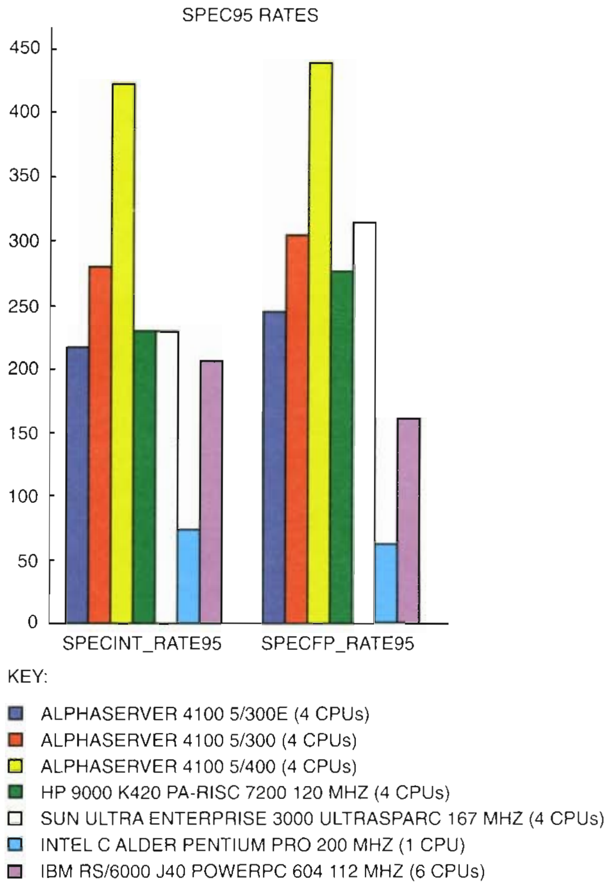


Figure 6
SPEC95 Throughput Results (SPEC95 Rates)

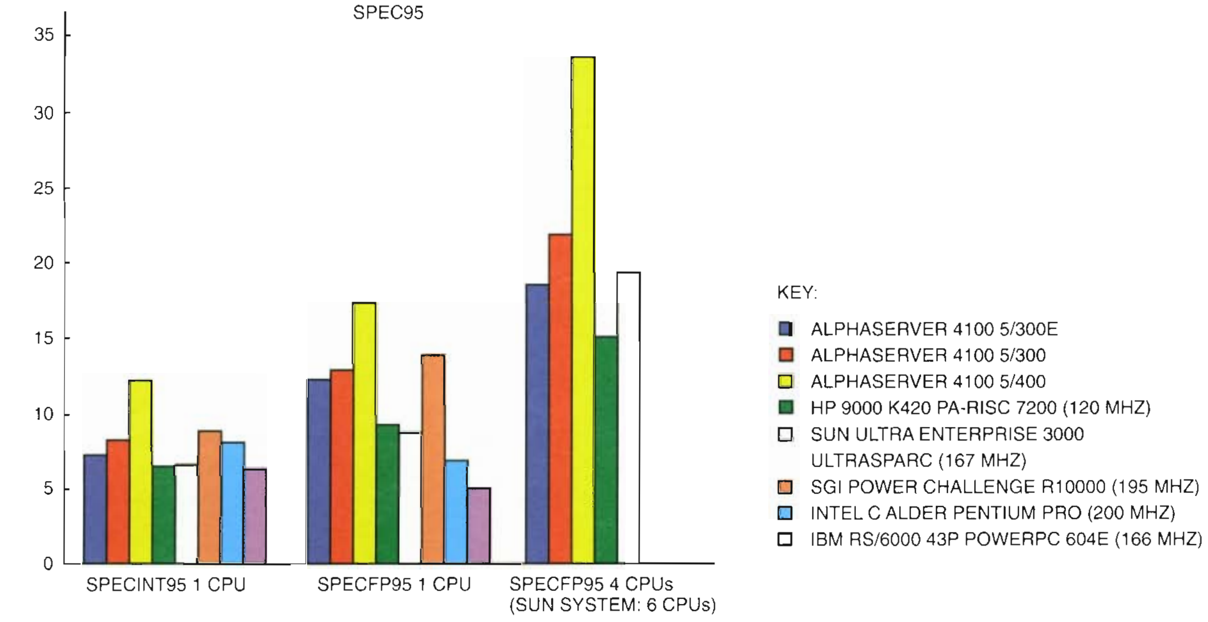


Figure 5
SPEC95 Speed Results

SPEC95 Multistream Performance Scaling

Figures 7 and 8 show SPEC95 multistream performance as the number of CPUs increases. The SMP scaling on the AlphaServer 4100 is comparable to that

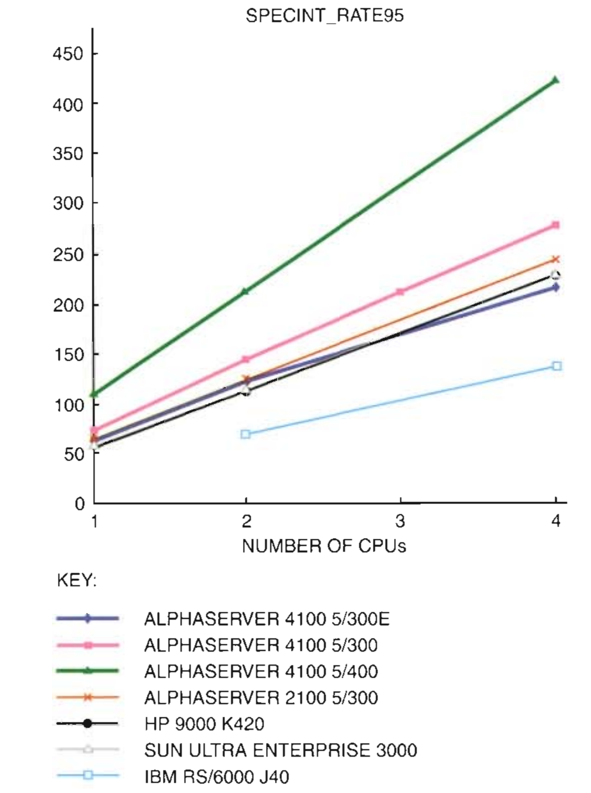


Figure 7
SPECint_rate95 Performance Scaling

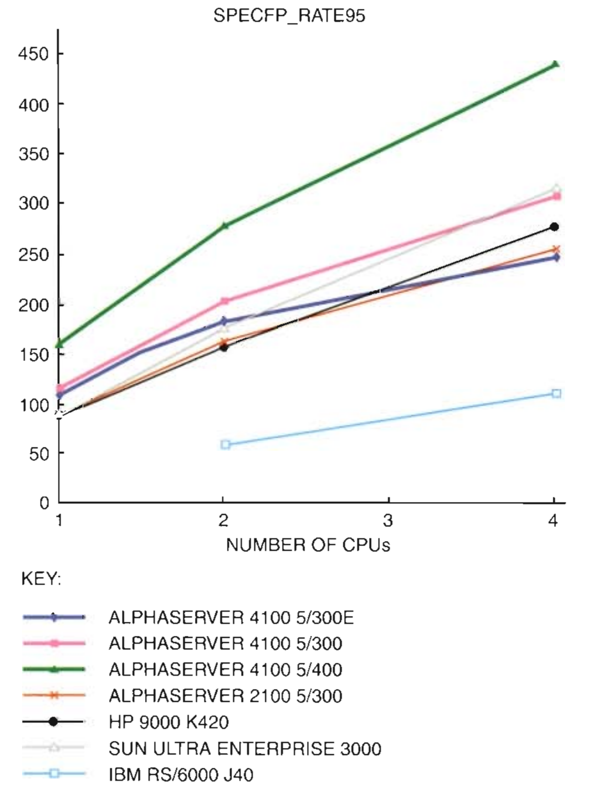


Figure 8
SPECfp_rate95 Performance Scaling

on the AlphaServer 2100 for integer workloads (that fit in the 5/300 2-MB B-cache). Note that SPECint_rate95 scales proportionally to the number of CPUs in the majority of systems, since these workloads do not stress the memory subsystem. The SMP scaling in SPECfp_rate95 is lower, since the majority of these workloads do not fit in 1- to 4-MB caches.

In the majority of applications, the AlphaServer 4100 5/300 and 5/400 systems improve SMP scaling compared to the uncached AlphaServer 4100 5/300E by reducing the bus traffic (from fewer B-cache misses) and by taking advantage of the duplicate tag store (DTAG) to reduce the number of S-cache probes. The cached 5/300 scaling, however, is lower than the uncached 5/300E scaling in memory bandwidth-intensive applications (e.g., tomcatv and swim). The analysis of traces collected by the logic analyzer that monitors system bus traffic indicates that the lower scaling is caused by (1) SetDirty overhead, where SetDirty is a cache coherency operation used to mark data as modified in the initiating CPU's cache; (2) stall cycles on the memory bus; and (3) memory bank conflicts.[2,3]

Symmetric Multiprocessing Performance Scaling for Parallel Workloads

Parallel workloads have higher data sharing and lower memory bandwidth requirements than multistream workloads. As shown in Figures 9 and 10, the AlphaServer 4100 models with module-level caches improve the SMP scaling compared to the uncached AlphaServer 4100 model in the LINPACK 1000 × 1000 (million floating-point operations per second [MFLOPS]) and the parallel SPECfp95 benchmarks that benefit from 2- and 4-MB B-caches. Figure 9 indicates that the AlphaServer 4100 5/400 outperforms the SGI Origin 2000 system in the LINPACK 1000 × 1000 benchmark by 40 percent. Figure 10 indicates that the four-CPU AlphaServer 4100 5/400 shows better scaling than any other system in its class and outperforms the six-CPU Sun Ultra Enterprise 3000 system by more than 70 percent.

Very Large Memory Advantage: Commercial Performance

As shown in Figures 11 and 12, the AlphaServer 4100 performs well in the commercial benchmarks TPC-C and AIM Suite VII.[13,14] In addition to the low memory and I/O latency, the AlphaServer 4100 takes advantage of the VLM design in these I/O-intensive workloads: with four CPUs, the platform can support up to 8 GB of memory compared to 1 GB of memory on the AlphaServer 2100 system with four CPUs and 2 GB with three CPUs.

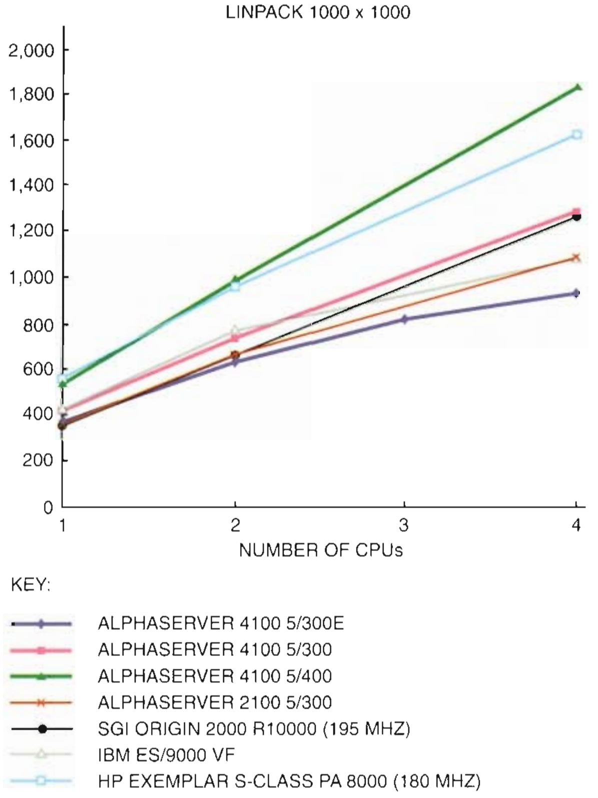


Figure 9
LINPACK 1000 × 1000 Parallel Performance Scaling

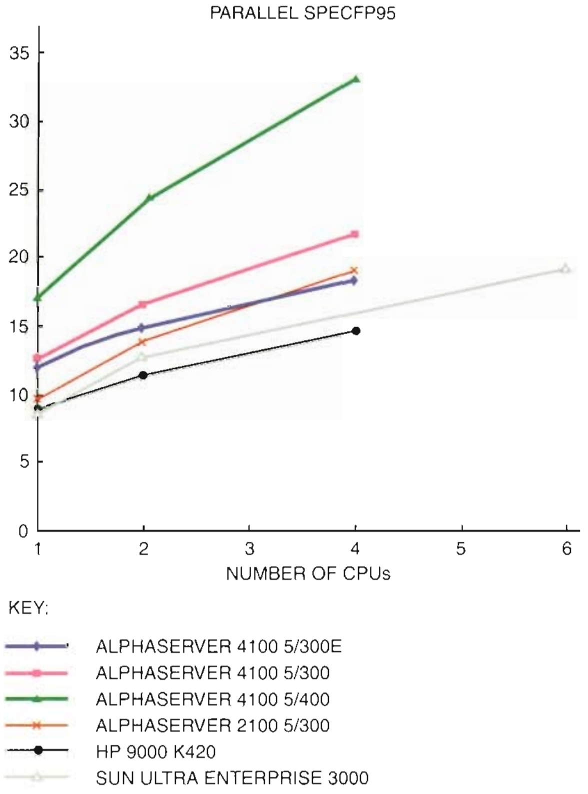


Figure 10
Parallel SPECfp95 Performance Scaling

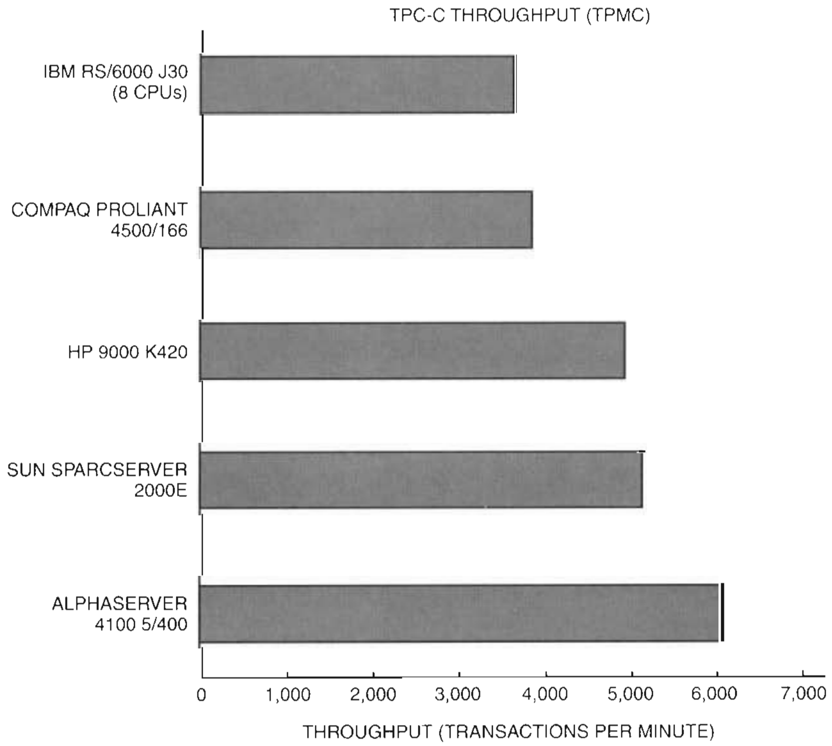


Figure 11
Transaction Processing Performance (TPC-C Using an Oracle Database)

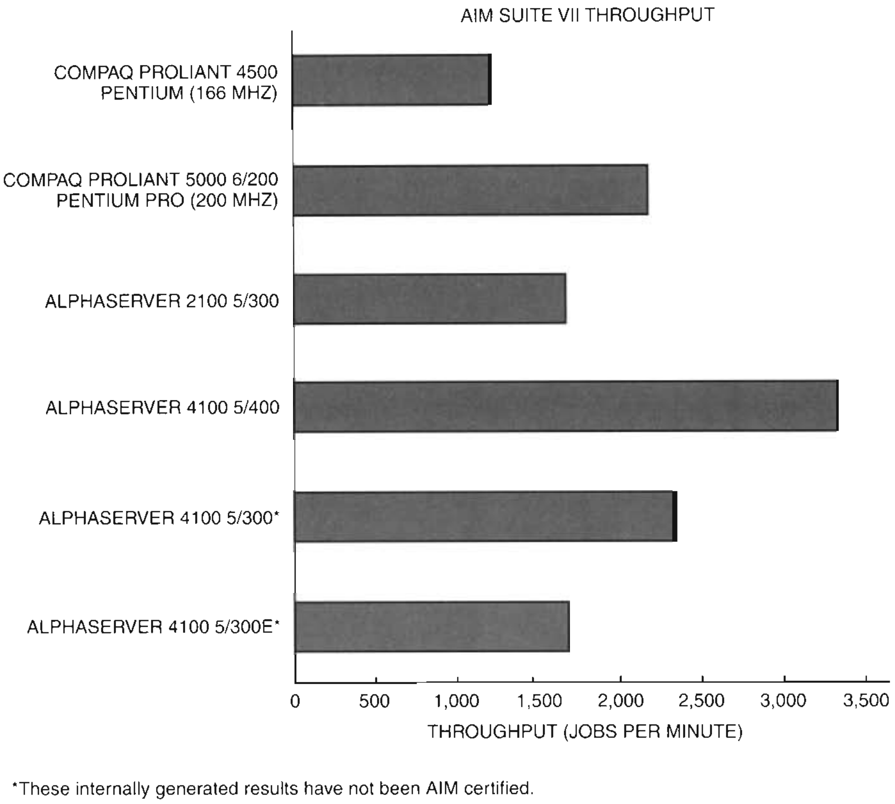


Figure 12
AIM Suite VII Multiuser/Shared UNIX Mix Performance

Figures 11 and 12 show the AlphaServer 4100 system's TPC-C performance (using an Oracle database) and AIM Suite VII throughput performance as compared to other industry-leading vendors. Note that the performance of the uncached AlphaServer 4100 5/300E is comparable to that of the 300-MHz AlphaServer 2100. (The AlphaServer 2100 system used in this test had three CPUs and 2 GB of memory, whereas the AlphaServer 4100 system had four CPUs and 2 GB of memory.)

With its 2-MB B-cache, the AlphaServer 4100 5/300 improves throughput by 40 percent in the AIM Suite VII benchmark tests as compared to the uncached AlphaServer 4100 5/300E. The AlphaServer 4100 5/400, with its 4-MB B-cache, benefits from its 33 percent faster clock and two times larger B-cache and provides 40 percent improvement over the AlphaServer 4100 5/300. Note that the AlphaServer 4100 5/300 and 5/300E results were obtained through internal testing and have not been AIM certified. The AlphaServer 5/400 results have AIM certification.

Compared to the best published industry AIM Suite VII performance, the AlphaServer 4100 5/300 throughput is almost twice that of the Compaq ProLiant 4500 server, and the AlphaServer 4100 5/400 throughput is more than 50 percent higher than that of the Compaq ProLiant 5000 server.[14] At the October 1996 UNIX Expo, the AlphaServer 4100 family won three AIM Hot Iron Awards: for the best performance on the Windows NT operating system (for systems priced at more than $50,000) and for the best price/performance in two UNIX mixes—multiuser shared and file system (for systems priced at more than $150,000).[14]

Cache Improvement on the AlphaServer 4100 System

Figures 13 and 14 show the percentage performance improvement provided by the 2-MB B-cache in the AlphaServer 4100 5/300 as compared to the uncached AlphaServer 4100 5/300E. Figure 13 shows the improvement across a variety of workloads; Figure 14 shows the improvement in individual SPEC95 benchmarks for one and four CPUs.

As shown in Figure 13, the 2-MB B-cache in the AlphaServer 4100 5/300 improves the performance by 5 to 20 percent for one CPU and 25 to 40 percent for four CPUs as compared to the uncached AlphaServer 4100 5/300E system. The benefits derived from having larger caches are significantly greater for four CPUs compared to one CPU, since large caches help alleviate bus traffic in multiprocessor systems.

The workloads that do not fit in the 2- to 4-MB B-cache (i.e., tomcatv, swim, applu) in Figure 14

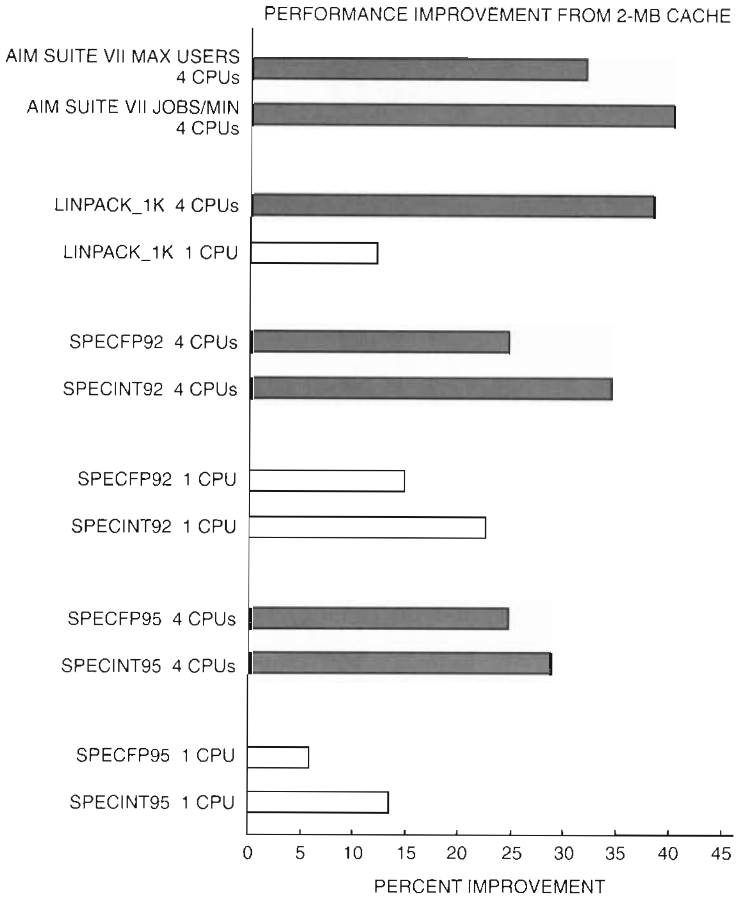


Figure 13
Performance Improvement across Various Workloads from a 2-MB B-Cache

run faster on the uncached AlphaServer 4100 than on the cached AlphaServer 4100 (up to 10 percent faster on one CPU and 20 percent faster on four CPUs) due to the overhead for probing the B-cache and the increase in SetDirty bandwidth. The majority of the other workloads benefit from larger caches.

The AlphaServer 4100 5/400 further improves the performance by increasing the size of the B-cache from 2 MB to 4 MB. In addition, the CPU clock improvement of 33 percent, B-cache improvement of 7 percent in latency and 11 percent in bandwidth, and the memory bus speed improvement of 11 percent together yield an overall 30 to 40 percent improvement in the AlphaServer 4100 model 5/400 performance as compared to that of the AlphaServer 4100 model 5/300.

Large Scientific Applications: Sparse LINPACK

The Sparse LINPACK benchmark solves a large, sparse symmetric system of linear equations using the conjugate gradient (CG) iterative method. The benchmark has three cases, each with a different type of preconditioner. Cases 1 and 2 use the incomplete Cholesky (IC) factorization as the preconditioner, whereas Case 3 uses the diagonal preconditioner.

This workload is representative of large scientific applications that do not fit in megabyte-size caches. The workload is important in large applications, e.g., models of electrical networks, economic systems, diffusion, radiation, and elasticity. It was decomposed to run on multiprocessor systems using the KAP preprocessor.

Figure 15 shows that the uncached AlphaServer 4100 5/300E outperforms the AlphaServer 8400 by 41 percent for one CPU and by 9 percent for two CPUs because of higher delivered system bus bandwidth. However, the AlphaServer 4100 5/300E falls behind with three and four CPUs, as it does in the McCalpin memory bandwidth tests shown in Figure 3. Note that with one CPU, the 300-MHz uncached AlphaServer 4100 performs at the same level as the 400-MHz cached AlphaServer 4100 and performs 18 percent better than the 300-MHz cached AlphaServer 4100. This is an example of the type of application for which the cache diminishes the performance. The AlphaServer 4100 5/300E is a better match for this class of applications than the cached systems.

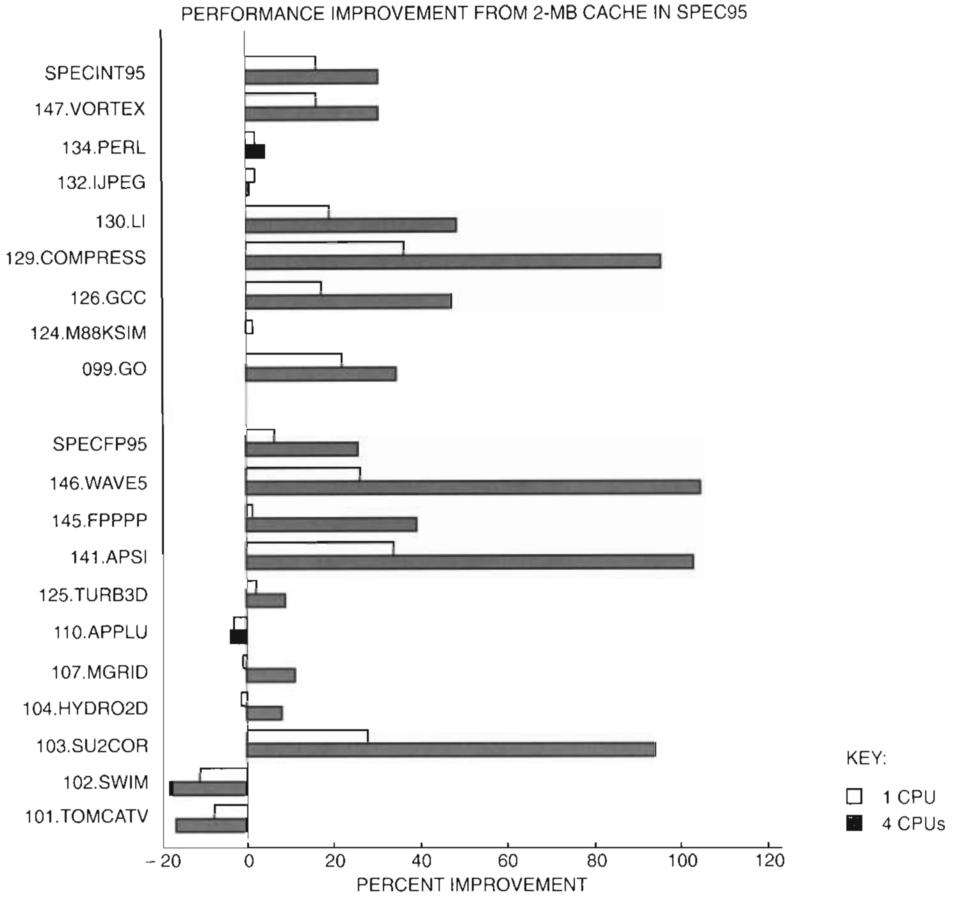


Figure 14
SPEC95 Performance Improvement from a 2-MB B-Cache

Image Rendering

The AlphaServer 4100 shows significant performance advantage in image rendering applications compared to the other industry-leading vendors. Figure 16 shows that the AlphaServer 4100 5/400 system is approximately 4 times faster than the Sun SPARC system that was used in the movie *Toy Story*, as measured in RenderMarks.[15] The AlphaServer 4100 is 2.6 times faster than the Silicon Graphics POWER CHALLENGE system and 2.4 times faster than the HP/Convex Exemplar SPP-1200 system on the Mental Ray image rendering application from Mental Images. These image rendering applications take advantage of larger caches, and the performance improves as the cache size increases, particularly with four CPUs.

Performance Counter Profiles

The figures in this section, Figures 17 through 22, show the performance statistics collected using the built-in Alpha 21164 performance counters on the AlphaServer 4100 5/400 system. These hardware monitors collect various events, including the number and type of instructions issued, multiple issues, single issues, branch mispredictions, stall components, and cache misses.[3,16,17] These statistics are useful for analyzing the system behavior under various workloads. The results of this analysis can be used by computer architects to drive hardware design trade-offs in future system designs.

The SPEC95 cycles per instruction (CPI) data presented in Figure 17 shows lower CPI values for the integer benchmarks (CPI values of 0.9 to 1.5) than for the floating-point benchmarks (CPI values of 0.9 to 2.2). The CPI in commercial workloads (e.g., TPC-C) is higher than in the SPEC benchmarks, primarily because commercial workloads have a higher stall time, as shown in Figure 18. Note that the performance counter statistics were collected with four CPUs running TPC-C (with a Sybase database), while SPEC95 statistics were collected on a single CPU.

The Alpha 21164 has two integer and two floating-point pipelines and is capable of issuing up to four instructions simultaneously. The integer pipeline 0 executes arithmetic, logical, load/store, and shift operations. The integer pipeline 1 executes arithmetic, logical, load, and branch/jump operations. The floating-point pipeline 0 executes add, subtract,

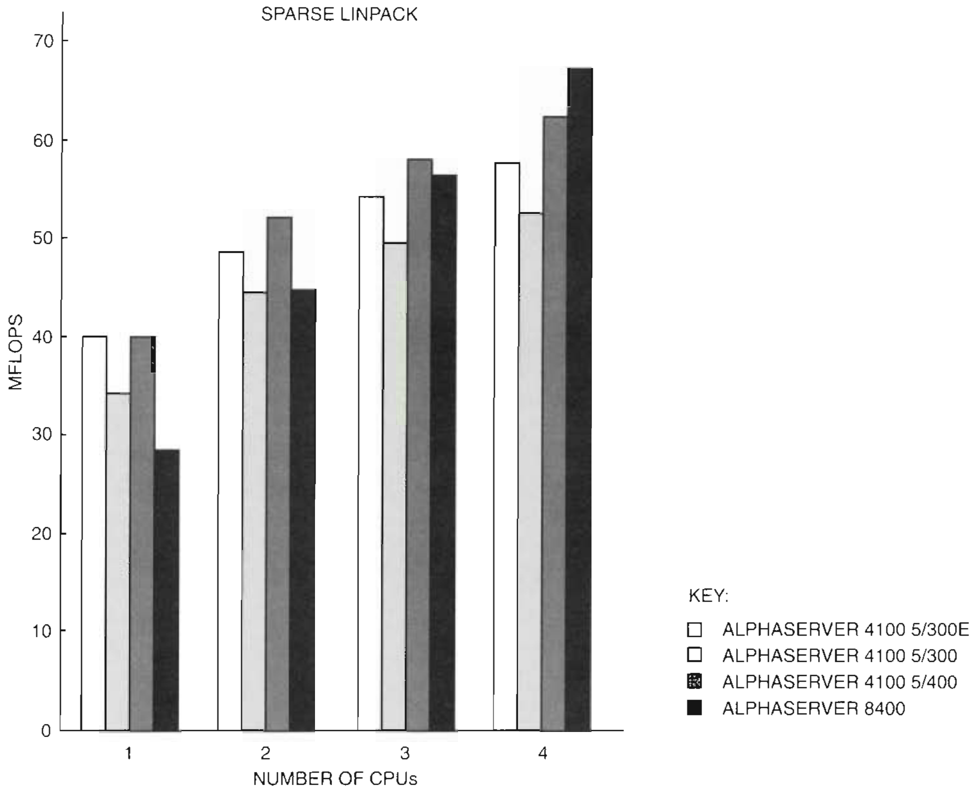


Figure 15
Sparse LINPACK Performance

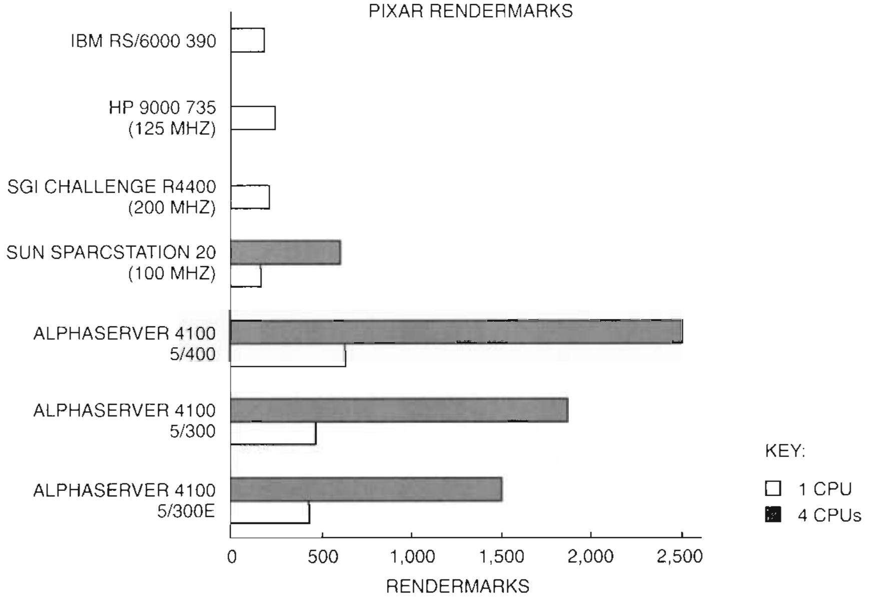


Figure 16
Image Rendering Performance

compare, and floating-point branch instructions. The floating-point pipeline 1 executes multiply instructions. The time distribution illustrated in Figure 18 indicates that most of the issuing time is spent in single and dual issuing. Triple and quad issuing is noticeable in several floating-point benchmarks, but, on average, only 3 percent of the time is spent on triple and quad issuing in the SPECfp95 benchmarks.

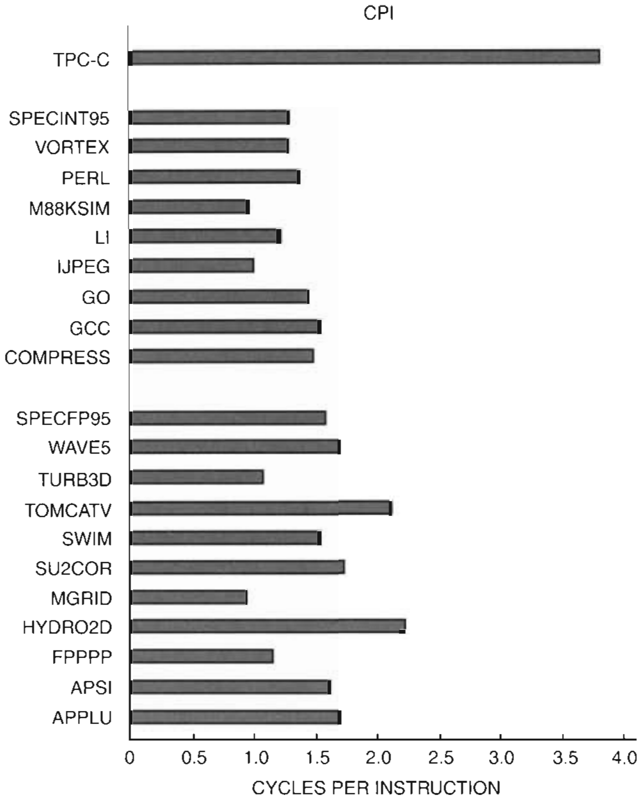


Figure 17
SPEC95 Cycles-per-instruction Comparison

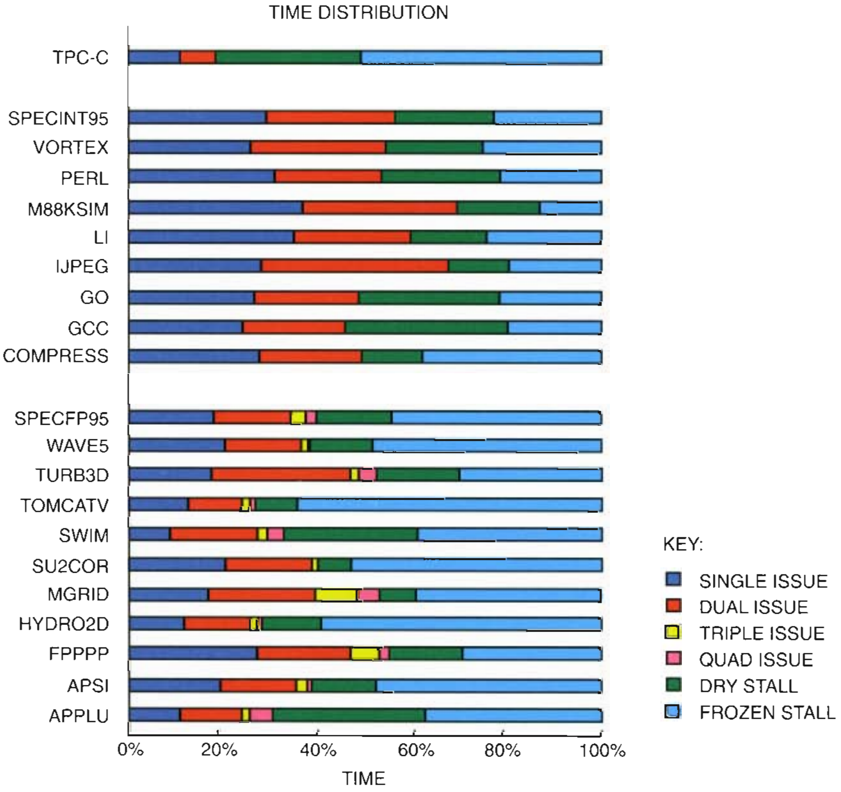


Figure 18
Issuing and Stall Time

The stall time (dry plus frozen stalls in Figure 18) is higher in the floating-point benchmarks than in the integer benchmarks and higher in the TPC-C benchmarks than in the SPEC95 benchmarks. Dry stalls include instruction stream (I-stream) stalls caused by the branch mispredictions, program counter (PC) mispredictions, replay traps, I-stream cache misses, and exception drain. Frozen stalls include data stream (D-stream) stalls caused by D-stream cache misses as well as register conflicts and unit busy. Dry stalls are higher in SPECint95 and TPC-C (mainly because of I-stream cache misses and replay traps), whereas frozen stalls are higher in SPECfp95 and TPC-C (mainly because of D-stream cache misses).

The Alpha 21164 microprocessor reduces the performance penalty due to cache misses by implementing a large, 96-KB on-chip S-cache.[3,4] This cache is three-way set associative and contains both instructions and data. The four-entry prefetch buffer allows prefetching of the next four consecutive cache blocks on an instruction cache (I-cache) miss. This reduces the penalty for I-stream stalls. The six-entry miss address file (MAF) merges loads in the same 32-byte block and allows servicing multiple load misses with one data fill. A six-entry write buffer is used to reduce the store bus traffic and to aggregate stores into 32-byte blocks.[3,4]

Figure 19 shows the instruction mix in SPEC95. The Alpha instructions are grouped into the following categories: load (both floating-point and integer), store (both floating-point and integer), integer (all integer instructions, excluding ones with only R31 or literal as operands), branch (all branch instructions including unconditional), and floating-point (except floating-point load and store instructions). Figure 19 shows the percentage of instructions in each category relative to the total number of instructions executed. Note that load/store instructions account for 30 to 40 percent of all instructions issued. Integer instructions are present in both integer and floating-point benchmarks, but no floating-point instructions exist in the SPECint95 and commercial TPC-C workloads. The integer and commercial workloads execute more branches, while the branch instructions make up only a few percent of all instructions issued in the floating-point workloads.

The cache misses shown in Figure 20 are higher in the floating-point benchmarks than in the integer benchmarks. The I-cache misses are low in the floating-point benchmarks (except for fpppp) and higher in the SPECint95 benchmarks and the TPC-C benchmark. The D-cache misses are high in the majority of the benchmarks, which indicates that a larger D-cache would reduce D-stream misses. The TPC-C benchmark would benefit from a larger S-cache and faster B-cache, since the number of S-cache misses is high. The B-cache misses are negligible in the SPECint95 benchmarks and higher in the majority of

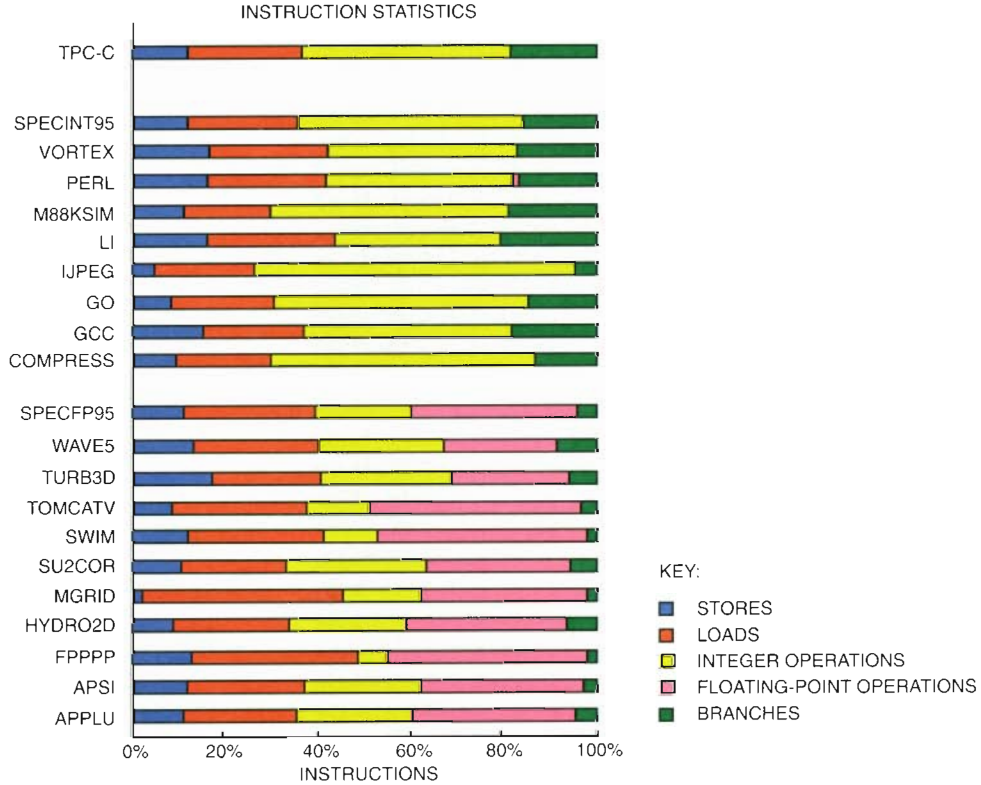


Figure 19
SPEC95 Instruction Profiles

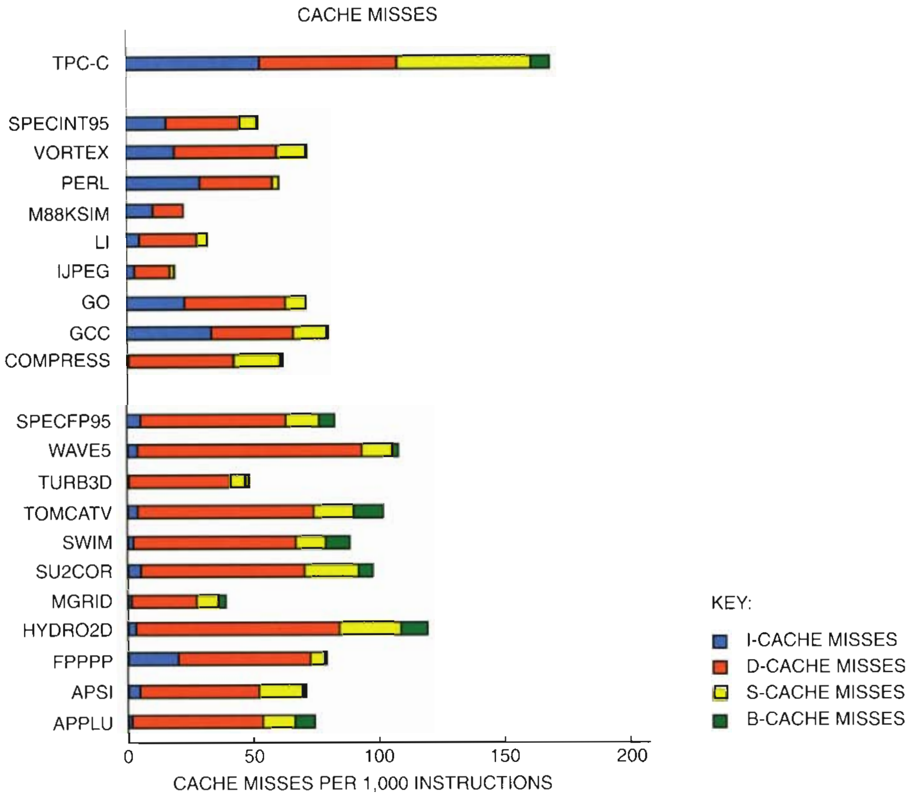


Figure 20
Cache Misses

the SPECfp95 TPC-C benchmarks. This data indicates that complex commercial workloads, such as TPC-C, are more profoundly affected by the cache design than simpler workloads, such as SPEC95.

The replay traps are generally caused by (1) full write-buffer (WB) traps (a full write buffer when a store instruction is executed) and full miss address file (MAF) traps (a full MAF when a load instruction is executed); and (2) load traps (speculative execution of an instruction that depends on a load instruction, and the load misses in the D-cache) and load-after-store traps (a load following a store that hits in the D-cache, and both access the same location).[3] The replay traps and branch/PC mispredictions shown in Figure 21 are not the major reason for the high stall time in the commercial workloads (TPC-C), since traps and mispredictions are higher in some of the SPECint95 benchmarks than in TPC-C. Instead, a high number of cache misses (see Figure 20) correlates well with the high stall time and CPI (see Figure 17) in TPC-C.

Figure 22 shows the estimated stall components in SPEC95 and TPC-C. A time-allocation model is used to analyze the performance effect of different stall components. The total execution time is divided into two components: the compute component (where the CPU is issuing instructions) and the stall component (where the CPU is not issuing instructions). The stall component is further divided into the dry and frozen stalls:

time = compute + stall
compute = single + dual + triple + quad issuing
stall = dry + frozen

dry = branch mispredictions + PC mispredictions
+ replay traps + I-stream cache misses
+ exception drain stalls
frozen = D-stream cache misses
+ register conflicts and unit busy

The branch and PC mispredictions affect the performance of SPECint95 workloads (6 percent of the time is spent in branch and PC mispredictions in SPECint95) and have little effect on the performance of SPECfp95 workloads (less than 1 percent of the time) and the TPC-C benchmark (1.4 percent of the time). The SPECint95 workloads are affected primarily by the load traps, whereas the SPECfp95 benchmarks are affected by both load and WB/MAF traps. Note that the time spent on a load replay trap is overlapped with the load-miss time.

The S-cache and B-cache stalls are high in the SPECfp95 and TPC-C benchmarks, where the stall time is dominated by the B-cache and memory latencies. Note the high stall time resulting from waiting for

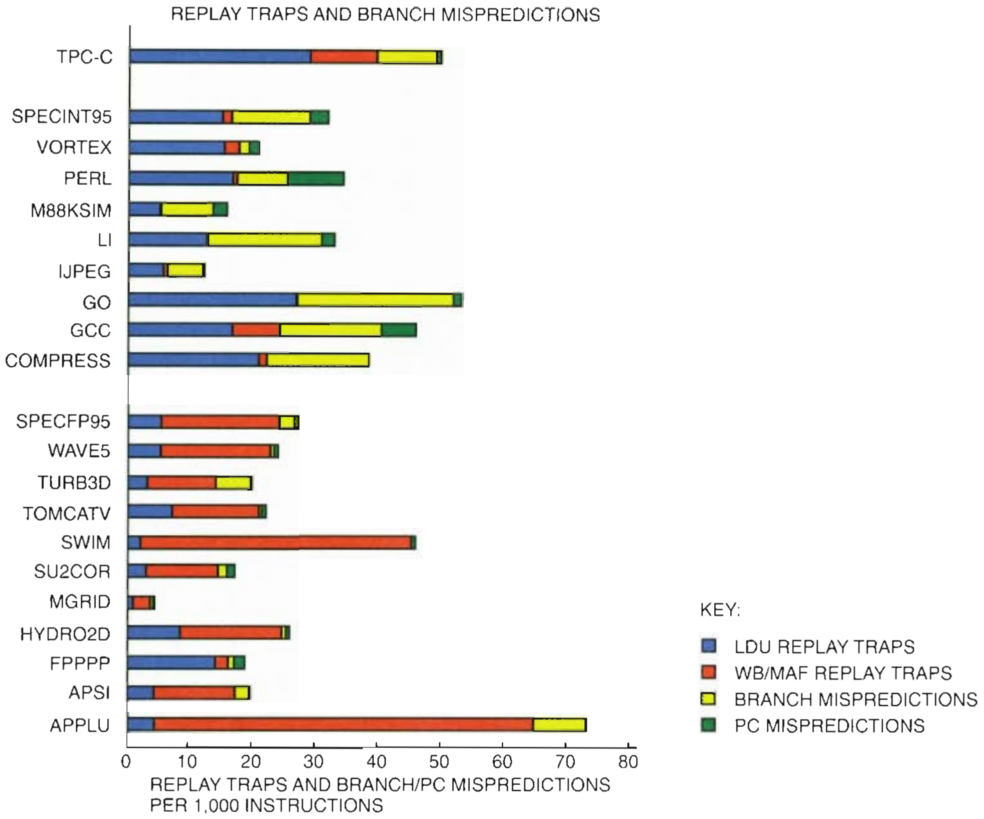


Figure 21
Replay Traps and Branch/PC Mispredictions

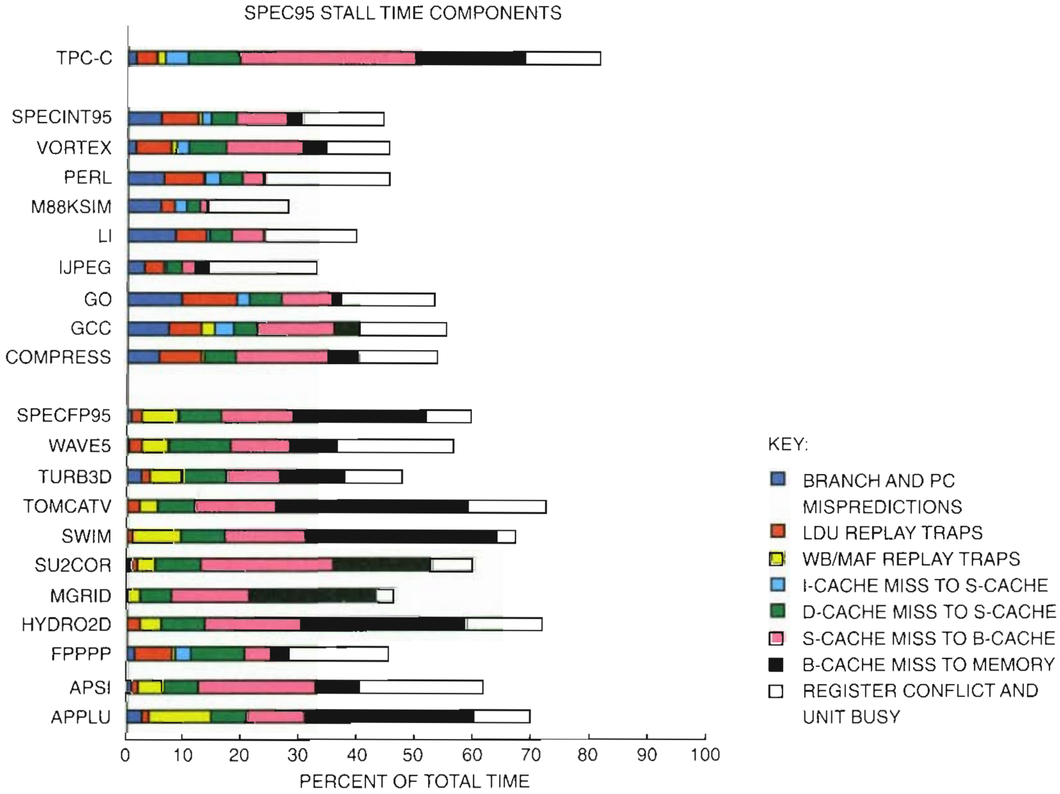


Figure 22
Estimated Stall Time Distribution

data from memory (close to 40 percent) in several of the SPECfp95 benchmarks that do not fit in a 4-MB cache. Although it contributes to the high SPECfp95 stall time, the memory component has a negligible effect on SPECint95 performance, since these benchmarks generate only a small number of B-cache misses (see Figure 20). Figure 22 indicates that stalls caused by cache misses are the largest component of the total stall time; therefore, reducing cache misses and improving cache and memory latencies would yield the largest performance benefit.

Once calibrated and validated with measurements, this model is an effective tool for evaluating the performance impact of various components on the overall system design. System architects can vary parameters, like the cache or memory access times or cache size, and adjust the appropriate stall component to predict performance of alternative designs without carrying out detailed and often time-consuming architectural simulations.

Conclusion

Using several performance metrics and a variety of workloads, we have demonstrated that the DIGITAL AlphaServer 4100 family of midrange servers provides significant performance improvements over the previous-generation AlphaServer platform and provides performance leadership compared to the leading industry vendors' platforms. The major AlphaServer 4100 performance strengths are the low memory and I/O latency and high memory bandwidth, the large-memory support (VLM), and the fast Alpha 21164 microprocessor. The work described in this paper has led to design changes that are expected to be implemented in future versions of the AlphaServer 4100 platform. The anticipated performance benefits will come from a faster CPU, faster and larger caches, faster memory, and improved memory bandwidth.

Acknowledgments

The authors would like to acknowledge the contributions of John Shakshober, Dave Stanley, Greg Tarsa, Dave Wilson, Paula Smith, John Henning, Michael Delaney, and Huy Phan for providing many of the benchmark measurements. In addition, special thanks go to Maurice Steinman, Glenn Herdeg, and Ted Gent for dedicating system resources and to Masood Heydari for supporting this work.

References

1. G. Herdeg, "Design and Implementation of the AlphaServer 4100 CPU and Memory Architecture," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 48–60.

2. M. Steinman, G. Harris, A. Kocev, V. Lamere, and R. Pannell, "The AlphaServer 4100 Cached Processor Module Architecture and Design," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 21–37.

3. *Alpha 21164 Microprocessor Hardware Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. EC-QAEQA-TE, 1994).

4. J. Edmondson, P. Rubinfeld, and V. Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro,* vol. 15, no. 2 (April 1995).

5. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, ISBN 1-55558-098-X, 1992).

6. SPEC95 Benchmarks (Manassas, Va.: Standard Performance Evaluation Corporation, 1995).

7. J. Dongarra, "Performance of Various Computers Using Standard Linear Equation Software" (Oak Ridge, Tenn.: Oak Ridge National Laboratory, 1996).

8. *UNIX System Price Performance Guide* (Menlo Park, Calif.: AIM Technology, Summer 1996).

9. J. Gray, ed., *The Handbook for Database and Transaction Processing Systems* (San Mateo, Calif.: Morgan Kauffman, 1991).

10. Information about the lmbench suite of benchmarks is available at http://reality.sgi.com/employees/lm_engr/lmbench/whatis_lmbench.html.

11. The STREAM benchmark program is described on-line by the University of Virginia, Department of Computer Science (Charlottesville, Va.) at http://www.cs.virginia.edu/stream.

12. The Standard Performance Evaluation Corporation (SPEC) makes available submitted results, benchmark descriptions, background information, and tools at http://www.specbench.org.

13. Information about the Transaction Processing Performance Council (TPC) is available at http://www.tpc.org.

14. Information about system performance benchmarking products from AIM Technology, Inc. (Menlo Park, Calif.) is available at http://www.aim.com.

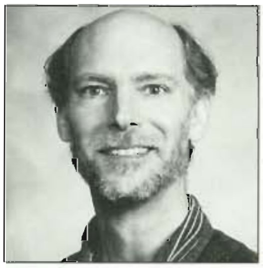
15. Information about Pixar Animation Studio's RenderMark benchmark is available at http://www.europe.digital.com/info/alphaserver/news/pixar.html.

16. Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads," *The 21st Annual International Symposium on Computer Architecture* (April 1994): 60–70.

17. Z. Cvetanovic and D. Bhandarkar, "Performance Characterization of the Alpha 21164 Microprocessor Using TP and SPEC Workloads," *The Second International Symposium on High-Performance Computer Architecture* (February 1996): 270–280.

Biographies

Zarka Cvetanovic
A consulting engineer in DIGITAL's Server Product Development Group, Zarka Cvetanovic was responsible for the performance characterization and analysis of the AlphaServer 4100, AlphaServer 8400/8200, AlphaServer 2100, DEC 7000, VAX 7000, and VAX 6000 systems, and for the performance modeling and definition of future AlphaServer platforms. Since joining DIGITAL in 1986, she has been involved in the development of fast database applications and efficient parallel applications for multiprocessor systems. Zarka received a Ph.D. in electrical and computer engineering from the University of Massachusetts, Amherst. She has published over a dozen technical papers at computer architecture conferences and in leading industry journals.



Darrel D. Donaldson
Darrel Donaldson is a senior consulting engineer and the technical leader and engineering manager for the AlphaServer 4100 project. He joined DIGITAL in 1983 and served as the lead technologist for the VAX 6000, VAX 7000, AlphaServer 7000, and AlphaServer 4100 projects. Darrel has a bachelor's degree in mathematics/physics from Miami University and a master's degree in electrical engineering from Cincinnati University, Cincinnati, Ohio. He holds 12 patents and has 10 patents pending, all related to protocols, signal integrity, and chip transceiver design for multiprocessor systems and nonvolatile memory chip design. Darrel maintains membership in the IEEE Electron Devices Society and the Solid-State Circuits Society.

Maurice B. Steinman
George J. Harris
Andrej Kocev
Virginia C. Lamere
Roger D. Pannell

The AlphaServer 4100 Cached Processor Module Architecture and Design

The DIGITAL AlphaServer 4100 processor module uses the Alpha 21164 microprocessor series combined with a large, module-level backup cache (B-cache). The cache uses synchronous cache memory chips and includes a duplicate tag store that allows CPU modules to monitor the state of each other's cache memories with minimal disturbance to the microprocessor. The synchronous B-cache, which can be easily synchronized with the system bus, permits short B-cache access times for the DIGITAL AlphaServer 4100 system. It also provides a smooth transition from accessing the B-cache to transferring data to or from main memory, without the need for re-synchronization or data buffering.

The DIGITAL AlphaServer 4100 series of servers represents the third generation of Alpha microprocessor-based, mid-range computer systems. Among the technical goals achieved in the system design were the use of four CPU modules, 8 gigabytes (GB) of memory, and partial block writes to improve I/O performance.

Unlike the previous generation of mid-range servers, the AlphaServer 4100 series can accommodate four processor modules, while retaining the maximum memory capacity. Using multiple CPUs to share the workload is known as symmetric multiprocessing (SMP). As more CPUs are added, the performance of an SMP system increases. This ability to increase performance by adding CPUs is known as scalability. To achieve perfect scalability, the performance of four CPUs would have to be exactly four times that of a single CPU system. One of the goals of the design was to keep scalability as high as possible yet consistent with low cost. For example, the AlphaServer 4100 system achieves a scalability factor of 3.33 on the Linpack 1000 × 1000, a large, parallel scientific benchmark. The same benchmark achieved 3.05 scalability on the previous-generation platform.[1]

The 8-GB memory in the AlphaServer 4100 system represents a factor of four improvement compared with the previous generation of mid-range servers.[2] The new memory is also faster in terms of the data volume flowing over the bus (bandwidth) and data access time (latency). Again, compared with the previous generation, available memory bandwidth is improved by a factor of 2.7 and latency is reduced by a factor of 0.6.

In systems of this class, memory is usually addressed in large blocks of 32 to 64 bytes. This can be inefficient when one or two bytes need to be modified because the entire block might have to be read out from memory, modified, and then written back into memory to achieve this minor modification. The ability to modify a small fraction of the block without having to extract the entire block from memory results in partial block writes. This capability also represents an advance over the previous generation of servers.

To take full advantage of the Alpha 21164 series of microprocessors, a new system bus was needed. The bus used in the previous generation of servers was not fast enough, and the cost and size of the bus used in high-end servers was not adaptable to mid-range servers.

Three separate teams worked on the project. One team defined the system architecture and the system bus, and designed the bus control logic and the memory modules.[3] The second team designed the peripheral interface (I/O), which consists of the Peripheral Component Interconnect (PCI) and the Extended Industry Standard Architecture (EISA) buses, and its interface to the system bus (I/O bridge).[4] The third team designed the CPU module. The remainder of this paper describes the CPU module design in detail. Before delving into the discussion of the CPU module, however, it is necessary to briefly describe how the system bus functions.

The system bus consists of 128 data bits, 16 check bits with the capability of correcting single-bit errors, 36 address bits, and some 30 control signals. As many as 4 CPU modules, 8 memory modules, and 1 I/O module plug into the bus. The bus is 10 inches long and, with all modules in place, occupies a space of 11 by 13 by 9 inches. With power supplies and the console, the entire system fits into an enclosure that is 26 by 12 by 17.5 inches in dimension.

CPU Module

The CPU module is built around the Alpha 21164 microprocessor. The module's main function is to provide an extended cache memory for the microprocessor and to allow it to access the system bus.

The microprocessor has its own internal cache memory consisting of a separate primary data cache (D-cache), a primary instruction cache (I-cache), and a second-level data and instruction cache (S-cache). These internal caches are relatively small, ranging in size from 8 kilobytes (KB) for the primary caches to 96 KB for the secondary cache. Although the internal cache operates at microprocessor speeds in the 400-megahertz (MHz) range, its small size would limit performance in most applications. To remedy this, the microprocessor has the controls for an optional external cache as large as 64 megabytes (MB) in size. As implemented on the CPU module, the external cache, also known as the backup cache or B-cache, ranges from 2 MB to 4 MB in size, depending on the size of the memory chips used. In this paper, all references to the cache assume the 4-MB implementation.

The cache is organized as a physical, direct-mapped, write-back cache with a 144-bit-wide data bus consisting of 128 data bits and 16 check bits, which matches the system bus. The check bits protect data integrity by providing a means for single-bit-error correction and double-bit-error detection. A physical cache is one in which the address used to address the cache memory is translated by a table inside the microprocessor that converts software addresses to physical memory locations. Direct-mapped refers to the way the cache memory is addressed, in which a subset of the physical address bits is used to uniquely place a main memory location at a particular location in the cache. When the microprocessor modifies data in a write-back cache, it only updates its local cache. Main memory is updated later, when the cache block needs to be used for a different memory address. When the microprocessor needs to access data not stored in the cache, it performs a system bus transaction (fill) that brings a 64-byte block of data from main memory into the cache. Thus the cache is said to have a 64-byte block size.

Two types of cache chips are in common use in modern computers: synchronous and asynchronous. The synchronous memory chips accept and deliver data at discrete times linked to an external clock. The asynchronous memory elements respond to input signals as they are received, without regard to a clock. Clocked cache memory is easier to interface to the clock-based system bus. As a result, all transactions involving data flowing from the bus to the cache (fill transactions) and from the cache to the bus (write microprocessor-based system transactions) are easier to implement and faster to execute.

Across the industry, personal computer and server vendors have moved from the traditional asynchronous cache designs to the higher-performing synchronous solutions. Small synchronous caches provide a cost-effective performance boost to personal computer designs. Server vendors push synchronous-memory technology to its limit to achieve data rates as high as 200 MHz; that is, the cache provides new data to the microprocessor every 5 nanoseconds.[5,6] The AlphaServer 4100 server is DIGITAL's first product to employ a synchronous module-level cache.

At power-up, the cache contains no useful data, so the first memory access the microprocessor makes results in a miss. In the block diagram shown in Figure 1, the microprocessor sends the address out on two sets of lines: the index lines connected to the cache and the address lines connected to the system bus address transceivers. One of the cache chips, called the TAG, is not used for data but instead contains a table of valid cache-block addresses, each of which is associated with a valid bit. When the microprocessor addresses the cache, a subset of the high-order bits addresses the tag table. A miss occurs when either of the following conditions has been met.

1. The addressed valid bit is clear, i.e., there is no valid data at that cache location.
2. The addressed valid bit is set, but the block address stored at that location does not match the address requested by the microprocessor.

Upon detection of a miss, the microprocessor asserts the READ MISS command on a set of four command lines. This starts a sequence of events

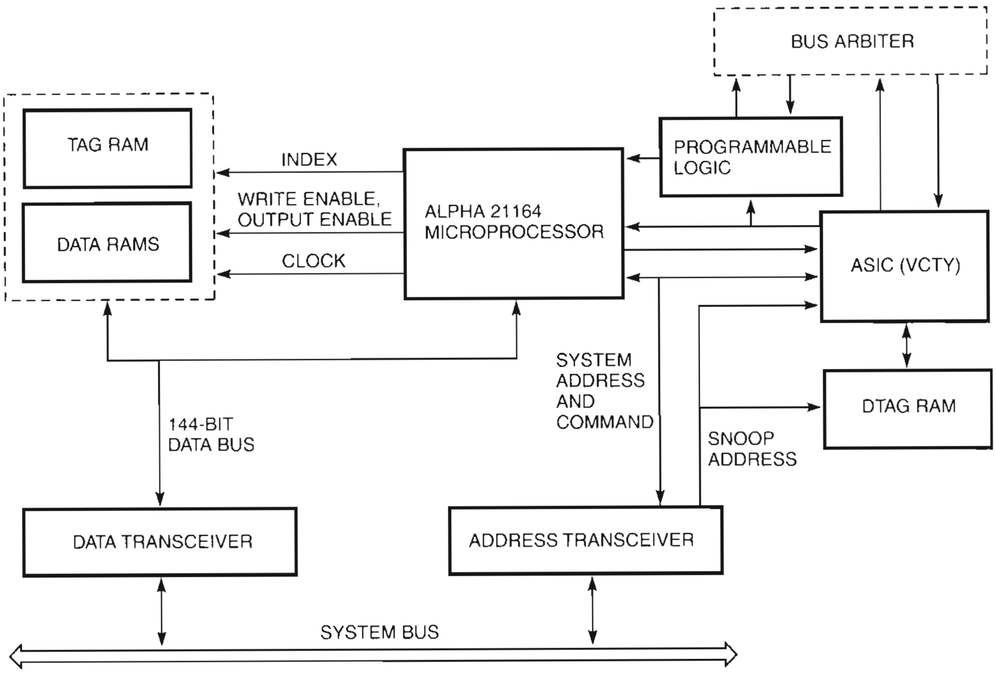


Figure 1
CPU Module

that results in the address being sent to the system bus. The memory receives this address and after a delay (memory latency), it sends the data on the system bus. Data transceivers on the CPU module receive the data and start a cache fill transaction that results in 64 bytes (a cache block) being written into the cache as four consecutive 128-bit words with their associated check bits.

In an SMP system, two or more CPUs may have the same data in their cache memories. Such data is known as shared, and the shared bit is set in the TAG for that address. The cache protocol used in the AlphaServer 4100 series of servers allows each CPU to modify entries in its own cache. Such modified data is known as dirty, and the dirty bit is set in the TAG. If the data about to be modified is shared, however, the microprocessor resets the shared bit, and other CPUs invalidate that data in their own caches. The need is thus apparent for a way to let all CPUs keep track of data in other caches. This is accomplished by the process known as snooping, aided by several dedicated bus signals.

To facilitate snooping, a separate copy of the TAG is maintained in a dedicated cache chip, called duplicate tag or DTAG. DTAG is controlled by an application-specific integrated circuit (ASIC) called VCTY. VCTY and DTAG are located next to each other and in close proximity to the address transceivers. Their timing is tied to the system bus so that the address associated with a bus transaction can easily be applied to the DTAG, which is a synchronous memory device, and the state of the cache at that address can be read out. If that cache location is valid and the address that is stored in the DTAG matches that of the system bus command (a hit in DTAG), the signal MC_SHARED may be asserted on the system bus by VCTY. If that location has been modified by the microprocessor, then MC_DIRTY is asserted. Thus each CPU is aware of the state of all the caches on the system. Other actions also take place on the module as part of this process, which is explained in greater detail in the section dealing specifically with the VCTY.

Because of the write-back cache organization, a special type of miss transaction occurs when new data needs to be stored in a cache location that is occupied by dirty data. The old data needs to be put back into the main memory; otherwise, the changes that the microprocessor made will be lost. The process of returning that data to memory is called a victim write-back transaction, and the cache location is said to be victimized. This process involves moving data out of the cache, through the system bus, and into the main memory, followed by new data moving from the main memory into the cache as in an ordinary fill transaction. Completing this fill quickly reduces the time that the microprocessor is waiting for the data. To speed up this process, a hardware data buffer on the module is used for storing the old data while the new data is being loaded into the cache. This buffer is physically a part of the data transceiver since each bit of the transceiver is a shift register four bits long. One hundred twenty-eight shift registers can hold the entire cache block (512 bits) of victim data while the new data is being read in through the bus receiver portion of the data transceiver chip. In this manner, the microprocessor does not have to wait until the victim data is transferred along the system bus and into the main memory before the fill portion of the transaction can take place. When the fill is completed, the victim data is shifted out of the victim buffer and into the main memory. This is known as an exchange, since the victim write-back and fill transactions execute on the system bus in reverse of the order that was initiated by the microprocessor. The transceiver has a signal called BYPASS; when asserted, it causes three of the four bits of the victim shift register to be bypassed. Consequently, for ordinary block write transactions, the transceiver operates without involving the victim buffer.

B-Cache Design

As previously mentioned, the B-cache uses synchronous random-access memory (RAM) devices. Each device requires a clock that loads signal inputs into a register. The RAM operates in the registered input, flow-through output mode. This means that an input flip-flop captures addresses, write enables, and write data, but the internal RAM array drives read output data directly as soon as it becomes available, without regard to the clock. The output enable signal activates RAM output drivers asynchronously, independently of the clock.

One of the fundamental properties of clocked logic is the requirement for the data to be present for some defined time (setup time) before the clock edge, and to remain unchanged for another interval following the clock edge (hold time). Obviously, to meet the setup time, the clock must arrive at the RAM some time after the data or other signals needed by the RAM. To help the module designer meet this requirement, the microprocessor may delay the RAM clock by one internal microprocessor cycle time (approximately 2.5 nanoseconds). A programmable register in the microprocessor controls whether or not this delay is invoked. This delay is used in the AlphaServer 4100 series CPU modules, and it eliminates the need for external delay lines.

For increased data bandwidth, the cache chips used on CPU modules are designed to overlap portions of successive data accesses. The first data block becomes available at the microprocessor input after a delay equal to the BC_READ_SPEED parameter, which is preset at power-up. The following data blocks are latched after a shorter delay, BC_READ_SPEED—WAVE. The BC_READ_SPEED is set at 10 microprocessor cycles and the WAVE value is set to 4, so that BC_READ_SPEED—WAVE is 6. Thus, after the first delay of 10 microprocessor cycles, successive data blocks are delivered every 6 microprocessor cycles. Figure 2 illustrates these concepts.

In Figure 2, the RAM clock at the microprocessor is delayed by one microprocessor cycle. The RAM clock at the RAM device is further delayed by clock buffer and network delays on the module. The address at the microprocessor is driven where the clock would have occurred had it not been delayed by one microprocessor cycle, and the address at the RAM is further delayed by index buffer and network delays. Index setup at the RAM satisfies the minimum setup time required by the chip, and so does address hold. Data is shown as appearing after data access time (a chip property), and data setup at the microprocessor is also illustrated.

VCTY

As described earlier, a duplicate copy of the microprocessor's primary TAG is maintained in the DTAG RAM. If DTAG were not present, each bus address would have to be applied by the microprocessor to the TAG to decide if the data at this address is present in the B-cache. This activity would impose a very large load on the microprocessor, thus reducing the amount of useful work it could perform. The main purpose of the DTAG and its supporting logic contained in the VCTY is to relieve the microprocessor from having to examine each address presented by the system bus. The microprocessor is only interrupted when its primary TAG must be updated or when data must be provided to satisfy the bus request.

VCTY Operation

The VCTY contains a system bus interface consisting of the system bus command and address signals, as well as some system bus control signals required for the VCTY to monitor each system bus transaction. There is also an interface to the microprocessor so that the VCTY can send commands to the microprocessor (system-to-CPU commands) and monitor the commands and addresses issued by the microprocessor. Last but not least, a bidirectional interface between the VCTY and the DTAG allows only those system bus addresses that require action to reach the microprocessor.

While monitoring the system bus for commands from other nodes, the VCTY checks for matches between the received system bus address and the data from the DTAG lookup. A DTAG lookup is initiated anytime a valid system bus address is received by the module. The DTAG location for the lookup is selected by using system bus Address<21:6> as the index into the DTAG. If the DTAG location had previously been marked valid, and there is a match between the received system bus Address<38:22> and the data from the DTAG lookup, then the block is present in the microprocessor's cache. This scenario is called a cache hit.

In parallel with this, the VCTY decodes the received system bus command to determine the appropriate update to the DTAG and determine the correct system bus response and CPU command needed to maintain system-wide cache coherency. A few cases are illustrated here, without any attempt at a comprehensive discussion of all possible transactions.

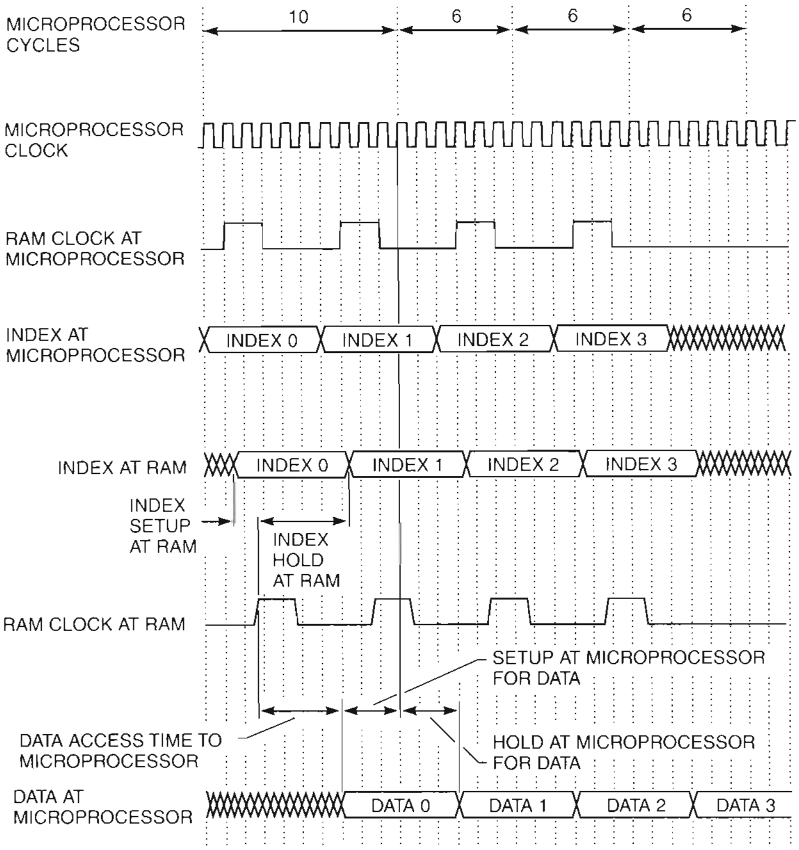


Figure 2
Cache Read Transaction Showing Timing

Assume that the DTAG shared bit is found to be set at this address, the dirty bit is not set, and the bus command indicates a write transaction. The DTAG valid bit is then reset by the VCTY, and the microprocessor is interrupted to do the same in the TAG.

If the dirty bit is found to be set, and the command is a read, the MC_DIRTY_EN signal is asserted on the system bus to tell the other CPU that the location it is trying to access is in cache and has been modified by this CPU. At the same time, a signal is sent to the microprocessor requesting it to supply the modified data to the bus so the other CPU can get an up-to-date version of the data.

If the address being examined by the VCTY was not shared in the DTAG and the transaction was a write, the valid bit is reset in the DTAG, and no bus signals are generated. The microprocessor is requested to reset the valid bit in the TAG. However, if the transaction was not a write, then shared is set in the DTAG, MC_SHARED is asserted on the bus, and a signal is sent to the microprocessor to set shared in the TAG.

From these examples, it becomes obvious that only transactions that change the state of the valid, shared, or dirty TAG bits require any action on the part of the microprocessor. Since these transactions are relatively infrequent, the DTAG saves a great deal of microprocessor time and improves overall system performance.

If the VCTY detects that the command originated from the microprocessor co-resident on the module, then the block is not checked for a hit, but the command is decoded so that the DTAG block is updated (if already valid) or allocated (i.e., marked valid, if not already valid). In the latter case, a fill transaction follows and the VCTY writes the valid bit into the TAG at that time. The fill transaction is the only one for which the VCTY writes directly into the TAG.

All cycles of a system bus transaction are numbered, with cycle 1 being the cycle in which the system bus address and command are valid on the bus. The controllers internal to VCTY rely on the cycle numbering scheme to remain synchronized with the system bus. By remaining synchronized with the system bus, all accesses to the DTAG and accesses from the VCTY to the microprocessor occur in fixed cycles relative to the system bus transaction in progress.

The index used for lookups to the DTAG is presented to the DTAG in cycle 1 of the system bus transaction. In the event of a hit requiring an update of the

DTAG and primary TAG, the microprocessor interface signal, EV_ABUS_REQ, is asserted in cycles 5 and 6 of that system bus transaction, with the appropriate system-to-CPU command being driven in cycle 6. The actual update to the DTAG occurs in cycle 7, as does the allocation of blocks in the DTAG.

Figure 3 shows the timing relationship of a system bus command to the update of the DTAG, including the sending of a system-to-CPU command to the microprocessor. The numbers along the top of the diagram indicate the cycle numbering. In cycle 1, when the signal MC_CA_L goes low, the system bus address is valid and is presented to the DTAG as the DTAG_INDEX bits. By the end of cycle 2, the DTAG data is valid and is clocked into the VCTY where it is checked for good parity and a match with the upper received system bus address bits. In the event of a hit, as is the case in this example, the microprocessor interface signal EV_ABUS_REQ is asserted in cycle 5 to indicate that the VCTY will be driving the microprocessor command and address bus in the next cycle. In cycle 6, the address that was received from the system bus is driven to the microprocessor along with the SETSHARED command. The microprocessor uses this command and address to update the primary tag control bits for that block. In cycle 7, the control signals DTAG_OE_L and DTAG_WE1_L are asserted low to update the control bits in the DTAG, thus indicating that the block is now shared by another module.

DTAG Initialization

Another important feature built into the VCTY design is a cursory self-test and initialization of the DTAG. After system reset, the VCTY writes all locations of the DTAG with a unique data pattern, and then reads the entire DTAG, comparing the data read versus what was written and checking the parity. A second write-read-compare pass is made using the inverted data pattern. This inversion ensures that all DTAG data bits are written and checked as both a 1 and a 0. In addition, the second pass of the initialization leaves each block of the DTAG marked as invalid (not present in the B-cache) and with good parity. The entire initialization sequence takes approximately 1 millisecond per megabyte of cache and finishes before the microprocessor completes its self-test, avoiding special handling by firmware.

Logic Synthesis

The VCTY ASIC was designed using the Verilog Hardware Description Language (HDL). The use of HDL enabled the design team to begin behavioral simulations quickly to start the debug process.

In parallel with this, the Verilog code was loaded into the Synopsys Design Compiler, which synthesized the behavioral equations into a gate-level design. The use of HDL and the Design Compiler enabled the designers to maintain a single set of behavioral models for the ASIC, without the need to manually enter

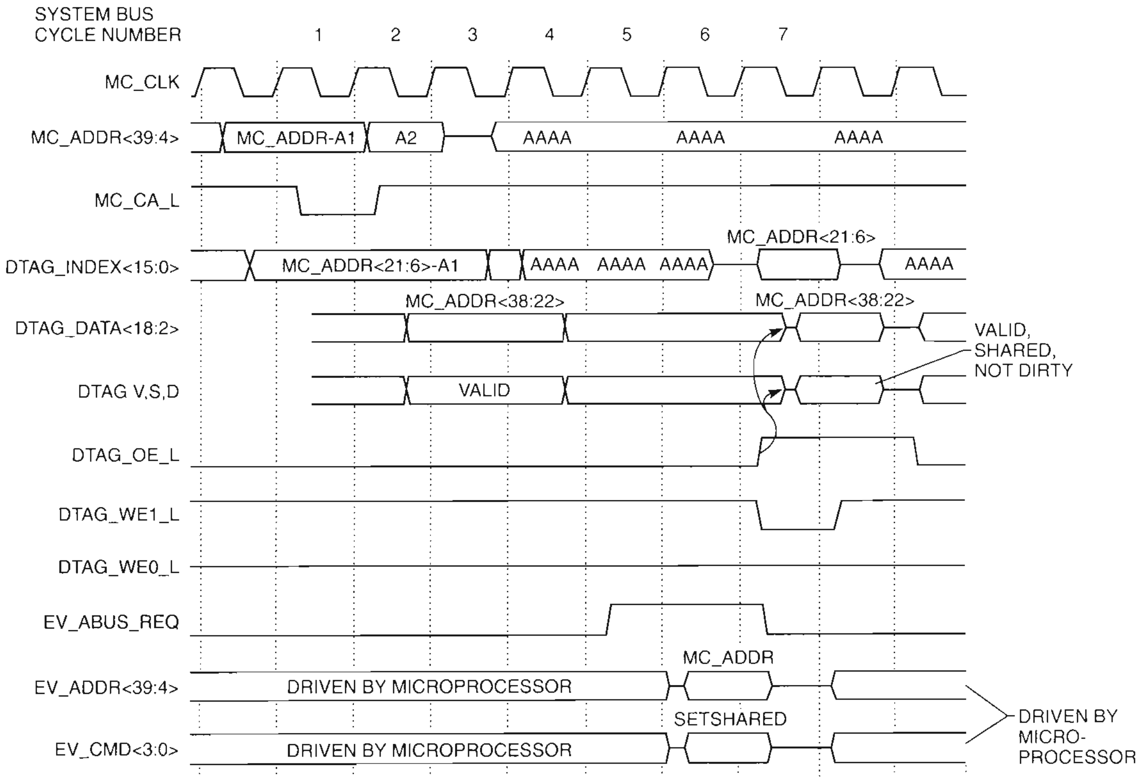


Figure 3
DTAG Operation

schematics to represent the gate-level design. The synthesis process is shown in a flowchart form in Figure 4. Logic verification is an integral part of this process, and the flowchart depicts both the synthesis and verification, and their interaction.

Only the synthesis is explained at this time. The verification process depicted on the right side of the flowchart is covered in a later section of this paper.

As shown on the left side of the flowchart, the logic synthesis process consists of multiple phases, in which the Design Compiler is invoked repeatedly on each subblock of the design, feeding back the results from the previous phase. The Synopsys Design Compiler was supplied with timing, loading, and area constraints to synthesize the VCTY into a physical design that met technology and cycle-time requirements. Since the ASIC is a small design compared to technology capabilities, the Design Compiler was run without an area constraint to facilitate timing optimization.

The process requires the designer to supply timing constraints only to the periphery of the ASIC (i.e., the I/O pins). The initial phase of the synthesis process calculates the timing constraints for internal networks that connect between subblocks by invoking the Design Compiler with a gross target cycle time of 100 nanoseconds (actual cycle time of the ASIC is 15 nanoseconds). At the completion of this phase, the process analyzes all paths that traverse multiple hierarchical subblocks within the design to determine the percentage of time spent in each block. The process then scales this data using the actual cycle time of 15 nanoseconds and assigns the timing constraints for internal networks at subblock boundaries. Multiple iterations may be required to ensure that each subblock is mapped to logic gates with the best timing optimization.

Once the Design Compiler completes the subblock optimization phase, an industry-standard electronic design interchange format (EDIF) file is output. The EDIF file is postprocessed by the SPIDER tool to generate files that are read into a timing analyzer, Topaz. A variety of industry-standard file formats can be input into SPIDER to process the data. Output files can then

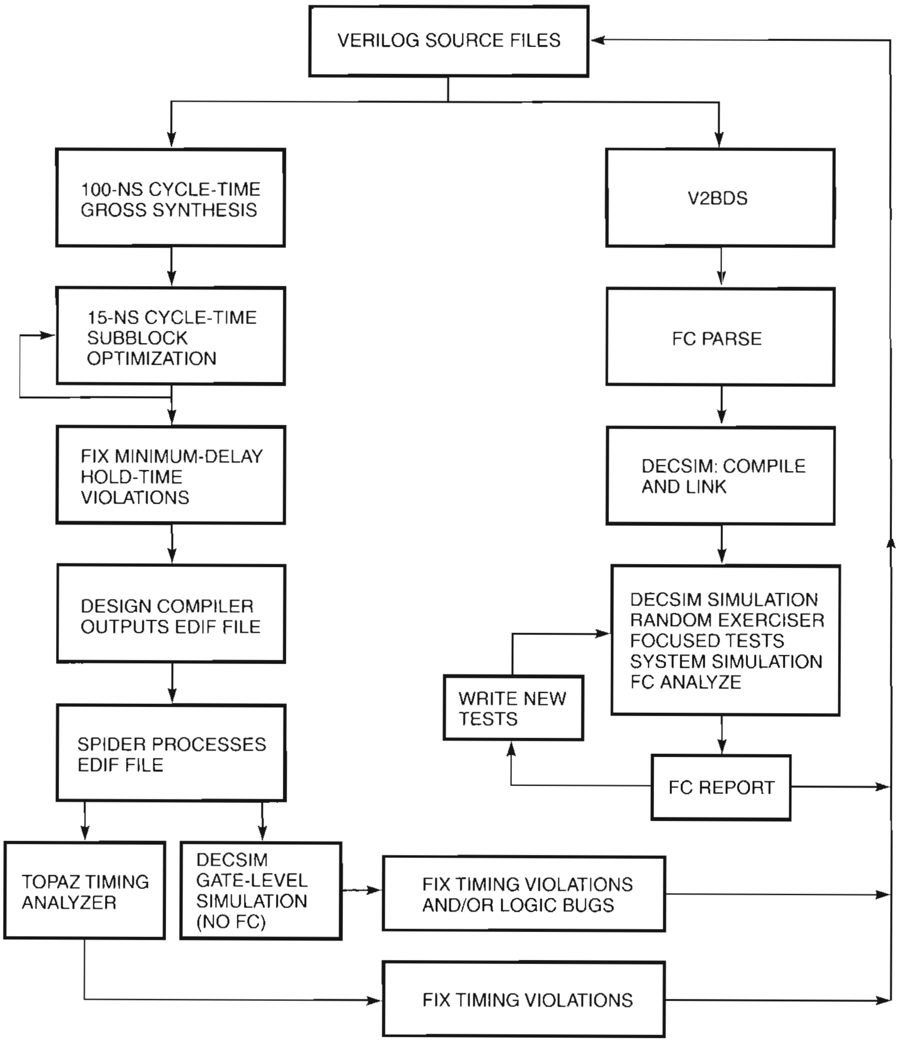

VERILOG SOURCE FILES
100-NS CYCLE-TIME GROSS SYNTHESIS
15-NS CYCLE-TIME SUBBLOCK OPTIMIZATION
FIX MINIMUM-DELAY HOLD-TIME VIOLATIONS
DESIGN COMPILER OUTPUTS EDIF FILE
SPIDER PROCESSES EDIF FILE
TOPAZ TIMING ANALYZER
DECSIM GATE-LEVEL SIMULATION (NO FC)
FIX TIMING VIOLATIONS AND/OR LOGIC BUGS
FIX TIMING VIOLATIONS
V2BDS
FC PARSE
DECSIM: COMPILE AND LINK
DECSIM SIMULATION RANDOM EXERCISER FOCUSED TESTS SYSTEM SIMULATION FC ANALYZE
WRITE NEW TESTS
FC REPORT

Figure 4
ASIC Design Synthesis and Verification Flow

be generated and easily read by internal CAD tools such as the DECSIM logic simulator and the Topaz timing analyzer.

Topaz uses information contained in the ASIC technology library to analyze the timing of the design as it was mapped by the Design Compiler. This analysis results in output data files that are used to constrain the ASIC layout process and obtain the optimal layout. Logic paths are prioritized for placement of the gates and routing of the connections based on the timing margins as determined by Topaz. Those paths with the least timing margin are given the highest priority in the layout process.

Logic Verification

This section of the paper discusses logic verification and focuses on the use of behavioral model simulation. It should also be noted that once the Design Compiler had mapped the design to gates, SPIDER was also used to postprocess the EDIF file so that DECSIM simulation could be run on the structural design. This process allowed for the verification of the actual gates as they would be built in the ASIC.

The right-hand side of Figure 4 illustrates the logic verification process using a behavioral simulation model. To verify the logic, the system must be performing transactions that exercise all or most of its logic. Ideally, the same software used in physical systems should be run on the design, but this is not practical because of the long run times that would be required. Therefore, specialized software tools are used that can accomplish the task in a shorter time. The verification team developed two such tools: the Random Exerciser and the Functional Checker. They are described in detail in this section.

Random Exerciser

Verification strategy is crucial to the success of the design. There are two approaches to verification testing, directed and random. Directed or focused tests require short run times and target specific parts of the design. To fully test a complex design using directed tests requires a very large number of tests, which take a long time to write and to run. Moreover, a directed test strategy assumes that the designer can foresee every possible system interaction and is able to write a test that will adequately exercise it. For these reasons, random testing has become the preferred methodology in modern logic designs.[7] Directed tests were not completely abandoned, but they compose only a small portion of the test suite.

Random tests rely on a random sequence of events to create the failing conditions. The goal of the Random Exerciser was to create a framework that would allow the verification team to create random tests quickly and efficiently without sacrificing flexibility and portability. It consisted of three parts: the test generator, the exerciser code, and the bus monitor.

Test Generator This collection of DECSIM commands randomly generates the test data consisting of addresses (both I/O space and memory space) and data patterns. The user controls the test data generator by setting test parameters. For example, to limit the range of working address space to the uppermost 2 MB of a 4-MB memory space, the working address space parameter is defined as [200000, 400000]. It tells the test generator to choose addresses within that range only—greater than 2 MB and less than 4 MB.

Exerciser Code This code is a collection of routines or sequences of Alpha macrocode instructions to be executed by the microprocessors. Each routine performs a unique task using one of the addresses supplied by the test generator. For example, routine 1 performs a read-verify-modify-write sequence. Routine 2 is similar to routine 1, but it reads another address that is 8 MB away from the original address, before writing to the cache. Since the B-cache is one-way associative, the original address is then evicted from the cache. Lastly, routine 3 performs a lock operation.

Most routines were of the type described above; they used simple load and store instructions. A few routines were of a special type: one generated interprocessor interrupts, others serviced interrupts, another routine generated errors (using addresses to nonexistent memory and I/O space) and checked that the errors were handled properly, and another routine exercised lock-type instructions more heavily.

The activity on the system bus generated by the CPUs was not enough to verify the logic. Two additional system bus agents (models of system bus devices) simulating the I/O were needed to simulate a full system-level environment. The I/O was modeled using so-called commander models. These are not HDL or DECSIM behavioral models of the logic but are written in a high-level language, such as C. From the perspective of the CPU, the commander models behave like real logic and therefore are adequate for the purpose of verifying the CPU module. There were several reasons for using a commander model instead of a logic/behavioral model. A complete I/O model was not yet available when the CPU module design began. The commander model was an evolution of a model used in a previous project, and it offered much needed flexibility. It could be configured to act as either an I/O interface or a CPU module and was easily programmable to flood the system bus with even more activity: memory reads and writes; interrupts to the CPUs by randomly inserting stall cycles in the pipeline; and assertion of system bus signals at random times.

Bus Monitor The bus monitor is a collection of DECSIM simulation watches that monitor the system bus and the CPU internal bus. The watches also report when various bus signals are being asserted and deasserted and have the ability to halt simulation if they encounter cache incoherency or a violation.

Cache incoherency is a data inconsistency, for example, a piece of nondirty data residing in the B-cache and differing from data residing in main memory. A data inconsistency can occur among the CPU modules: for example, two CPU modules may have different data in their caches at the same memory address. Data inconsistencies are detected by the CPU. Each one maintains an exclusive (nonshared) copy of its data that it uses to compare with the data it reads from the test addresses. If the two copies differ, the CPU signals to the bus monitor to stop the simulation and report an error.

The bus monitor also detects other violations:

1. No activity on the system bus for 1,000 consecutive cycles
2. Stalled system bus for 100 cycles
3. Illegal commands on the system bus and CPU internal bus
4. Catastrophic system error (machine check)

The combination of random CPU and I/O activity flooded the system bus with heavy traffic. With the help of the bus monitor, this technique exposed bugs quickly.

As mentioned, a few directed tests were also written. Directed tests were used to re-create a situation that occurred in random tests. If a bug was uncovered using a random test that ran three days, a directed test was written to re-create the same failing scenario. Then, after the bug was fixed, a quick run of the directed test confirmed that the problem was indeed corrected.

Functional Checker

During the initial design stages, the verification team developed the Functional Checker (FC) for the following purposes:

- To functionally verify the HDL models of all ASICs in the AlphaServer 4100 system
- To assess the test coverage

The FC tool consists of three applications: the parser, the analyzer, and the report generator. The right-hand side of Figure 4 illustrates how the FC was used to aid in the functional verification process.

Parser Since DECSIM was the chosen logic simulator, the first step was to translate all HDL code to BDS, a DECSIM behavior language. This task was performed using a tool called V2BDS. The parser's task was to postprocess a BDS file: extract information and generate a modified version of it. The information extracted was a list of control signals and logic statements (such as logical expressions, if-then-else statements, case statements, and loop constructs). This information was later supplied to the analyzer. The modified BDS was functionally equivalent to the original code, but it contained some embedded calls to routines whose task was to monitor the activity of the control signals in the context of the logic statements.

Analyzer Written in C, the analyzer is a collection of monitoring routines. Along with the modified BDS code, the analyzer is compiled and linked to form the simulation model. During simulation, the analyzer is invoked and the routines begin to monitor the activity of the control signals. It keeps a record of all control signals that form a logic statement. For example, assume the following statement was recognized by the parser as one to be monitored.

(A XOR B) AND C

The analyzer created a table of all possible combinations of logic values for A, B, and C; it then recorded which ones were achieved. At the start of simulation, there was zero coverage achieved.

ABC	Achieved
000	No
001	No
010	No
011	No
100	No
101	No
110	No
111	No

Achieved coverage = 0 percent

Further assume that during one of the simulation tests generated by the Random Exerciser, A assumed both 0 and 1 logic states, while B and C remained constantly at 0. At the end of simulation, the state of the table would be the following:

ABC	Achieved
000	Yes
001	No
010	No
011	No
100	Yes
101	No
110	No
111	No

Achieved coverage = 25 percent

Report Generator The report generator application gathered all tables created by the analyzer and generated a report file indicating which combinations were not achieved. The report file was then reviewed by the verification team and by the logic design team.

The report pointed out deficiencies in the verification tests. The verification team created more tests that would increase the "yes" count in the "Achieved" column. For the example shown above, new tests might be created that would make signals B and C assume both 0 and 1 logic states.

The report also pointed out faults in the design, such as redundant logic. In the example shown, the logic that produces signal B might be the same as the logic that produces signal C, a case of redundant logic.

The FC tool proved to be an invaluable aid to the verification process. It was a transparent addition to the simulation environment. With FC, the incurred degradation in compilation and simulation time was negligible. It performed two types of coverage analysis: exhaustive combinatorial analysis (as was described above) and bit-toggle analysis, which was used for vectored signals such as data and address buses. Perhaps the most valuable feature of the tool was the fact that it replaced the time-consuming and compute-intensive process of fault grading the physical design to verify test coverage. FC established a new measure of test coverage, the percentage of achieved coverage. In the above example, the calculated coverage would be two out of eight possible achievable combinations, or 25 percent.

For the verification of the cached CPU module, the FC tool achieved a final test coverage of 95.3 percent.

Module Design Process

As the first step in the module design process, we used the Powerview schematic editor, part of the Viewlogic CAD tool suite, for schematic capture. An internally developed tool, V2LD, converted the schematic to a form that could be simulated by DECSIM. This process was repeated until DECSIM ran without errors.

During this time, the printed circuit (PC) layout of the module was proceeding independently, using the ALLEGRO CAD tools. The layout process was partly manual and partly automated with the CCT router, which was effective in following the layout engineer's design rules contained in the DO files.

Each version of the completed layout was translated to a format suitable for signal integrity modeling, using the internally developed tools ADSconvert and MODULEX. The MODULEX tool was used to extract a module's electrical parameters from its physical description. Signal integrity modeling was performed with the HSPICE analog simulator. We selected HSPICE because of its universal acceptance by the industry. Virtually all component vendors will, on request, supply HSPICE models of their products. Problems detected by HSPICE were corrected either by layout modifications or by schematic changes. The module design process flow is depicted in Figure 5.

Software Tools and Models

Three internally developed tools were of great value. One was MSPG, which was used to display the HSPICE plots; another was MODULEX, which automatically generated HSPICE subcircuits from PC layout files and performed cross-talk calculations. Cross-talk amplitude violations were reported by MODULEX, and the offending PC traces were moved to reduce coupling. Finally, SALT, a visual PC display tool, was used to verify that signal routing and branching conformed to the design requirements.

One of the important successes was in data line modeling, where the signal lengths from the RAMs to the microprocessor and the transceivers were very critical. By using the HSPICE .ALTER statement and MODULEX subcircuit generator command, we could configure a single HSPICE deck to simulate as many as 36 data lines. As a result, the entire data line group could be simulated in only four HSPICE runs. In an excellent example of synergy between tools, the script capability of the MSPG plotting tool was used to extract, annotate, and create PostScript files of waveform plots directly from the simulation results, without having to manually display each waveform on the screen. A mass printing command was then used to print all stored PostScript files.

Another useful HSPICE statement was .MEASURE, which measured signal delays at the specified threshold levels and sent the results to a file. From this, a separate program extracted clean delay values and calculated the maximum and minimum delays, tabulating the results in a separate file. Reflections crossing the threshold levels caused incorrect results to be reported by the .MEASURE statement, which were easily seen in the tabulation. We then simply looked at the waveform printout to see where the reflections were occurring. The layout engineer was then asked to modify those signals by changing the PC trace lengths to either the microprocessor or the transceiver. The modified signals were then resimulated to verify the changes.

Timing Verification

Overall cache timing was verified with the Timing Designer timing analyzer from Chronology Corporation. Relevant timing diagrams were drawn using the waveform plotting facility, and delay values and controlling parameters such as the microprocessor cycle interval, read speed, wave, and other constants were entered into the associated spreadsheet. All

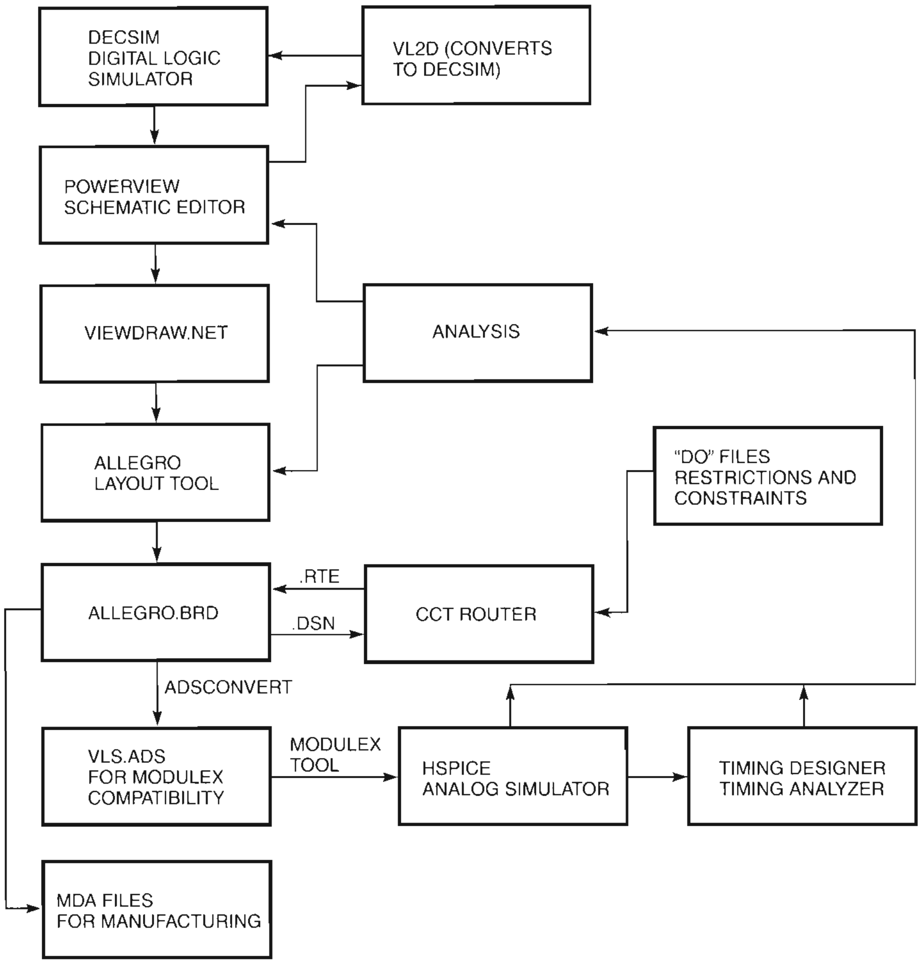


Figure 5
Design Process Flow

delays were expressed in terms of HSPICE-simulated values and those constants, as appropriate. This method simplified changing parameters to try various "what if" strategies. The timing analyzer would instantly recalculate the delays and the resulting margins and report all constraint violations. This tool was also used to check timing elsewhere on the module, outside of the cache area, and it provided a reasonable level of confidence that the design did not contain any timing violations.

Signal Integrity

In high-speed designs, where signal propagation times are a significant portion of the clock-to-clock interval, reflections due to impedance mismatches can degrade the signal quality to such an extent that the system will fail. For this reason, signal integrity (SI) analysis is an important part of the design process. Electrical connections on a module can be made following a direct point-to-point path, but in high-speed designs, many signals must be routed in more complicated patterns. The most common pattern involves bringing a signal to a point where it branches out in several directions, and each branch is connected to one or more receivers. This method is referred to as treeing.

The SI design of this module was based on the principle that component placement and proper signal treeing are the two most important elements of a good SI design. However, ideal component placement is not always achievable due to overriding factors other than SI. This section describes how successful design was achieved in spite of less than ideal component placement.

Data Line Length Optimization

Most of the SI work was directed to optimizing the B-cache, which presented a difficult challenge because of long data paths. The placement of major module data bus components (microprocessor and data transceivers) was dictated by the enclosure requirements and the need to fit four CPUs and eight memory modules into the system box. Rather than allowing the microprocessor heat-sink height to dictate module spacing, the system designers opted for fitting smaller memory modules next to the CPUs, filling the space that would have been left empty if module spacing were uniform. As a consequence, the microprocessor and data transceivers had to be placed on opposite ends of the module, which made the data bus exceed 11 inches in length. Figure 6 shows the placement of the major components.

Each cache data line is connected to four components: the microprocessor chip, two RAMs, and the bus transceiver. As shown in Table 1, any one of these components can act as the driver, depending on the transaction in progress.

The goal of data line design was to obtain clean signals at the receivers. Assuming that the microprocessor, RAMs, and the transceiver are all located in-line without branching, with the distance between the two RAMs near zero, and since the positions of the microprocessor and the transceivers are fixed, the only variable is the location of the two RAMs on the data line. As shown in the waveform plots of Figures 7 and 8, the quality of the received signals is strongly affected by this variable. In Figure 7, the reflections are so large that they exceed threshold levels. By contrast, the reflections in Figure 8 are very small, and their waveforms show signs of cancellation. From this it can be inferred that optimum PC trace lengths cause the reflections to cancel. A range of acceptable RAM positions was found through HSPICE simulation. The results of these simulations are summarized in Table 2.

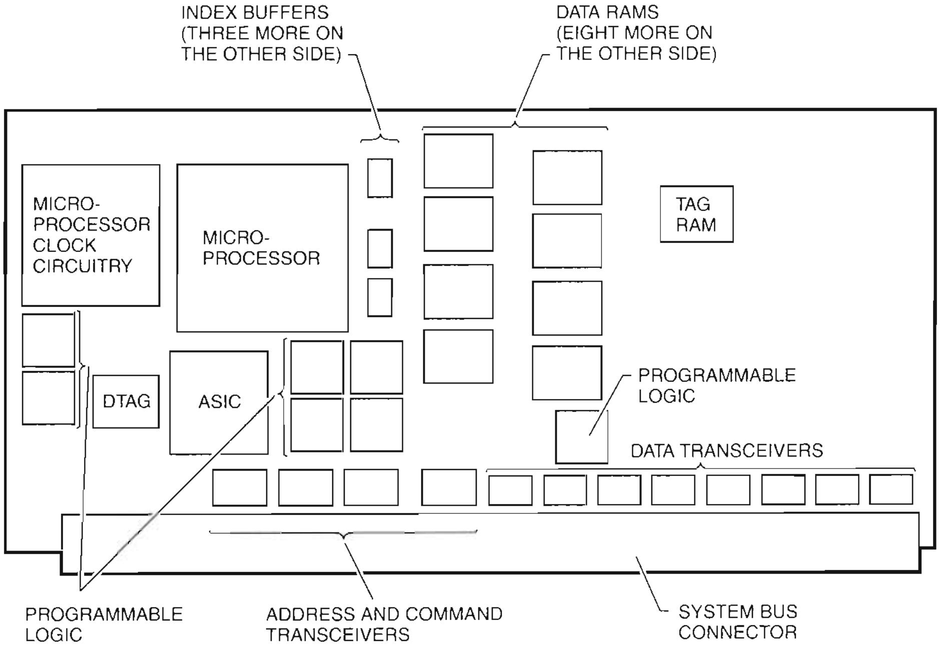

INDEX BUFFERS (THREE MORE ON THE OTHER SIDE)
DATA RAMS (EIGHT MORE ON THE OTHER SIDE)
MICRO-PROCESSOR CLOCK CIRCUITRY
MICRO-PROCESSOR
TAG RAM
DTAG
ASIC
PROGRAMMABLE LOGIC
DATA TRANSCEIVERS
PROGRAMMABLE LOGIC
ADDRESS AND COMMAND TRANSCEIVERS
SYSTEM BUS CONNECTOR

Figure 6
Placement of Major Components

Table 1
Data Line Components

Transaction	Driver	Receiver
Private cache read	RAM	Microprocessor
Private cache write	Microprocessor	RAM
Cache fill	Transceiver	RAM and microprocessor
Cache miss with victim	RAM	Transceiver
Write block	Microprocessor	RAM and transceiver

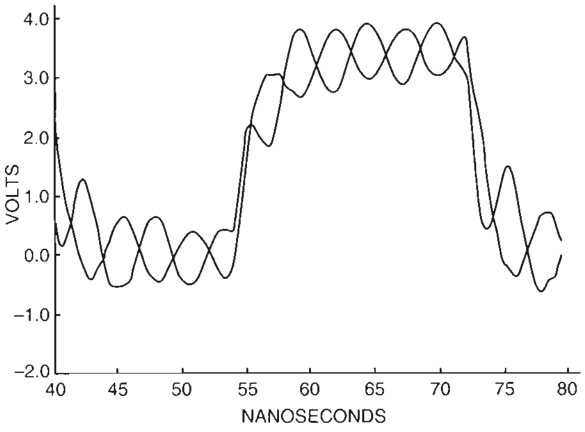


Figure 7
Private Cache Read Showing Large Reflections Due to Unfavorable Trace Length Ratios

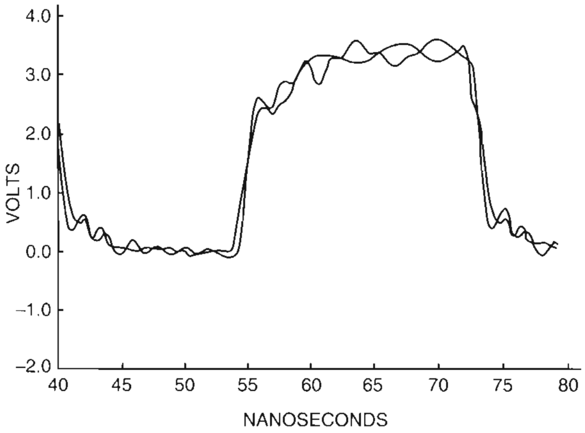


Figure 8
Private Cache Read Showing Reduced Reflections with Optimized Trace Lengths

In the series of simulations given in Table 2, the threshold levels were set at 1.1 and 1.8 volts. This was justified by the use of perfect transmission lines. The lines were lossless, had no vias, and were at the lowest impedance level theoretically possible on the module (55 ohms). The entries labeled SR in Table 2 indicate unacceptably large delays caused by signal reflections recrossing the threshold levels. Discarding these entries leaves only those with microprocessor-to-RAM distance of 3 or more inches and the RAM-to-transceiver distance of at least 6 inches, with the total microprocessor-to-transceiver distance not exceeding 11 inches. The layout was done within this range, and all data lines were then simulated using the network subcircuits generated by MODULEX with threshold levels set at 0.8 and 2.0 volts. These subcircuits included the effect of vias and PC traces run on several signal planes. That simulation showed that all but 12 of the 144 data- and check-bit lines had good signal integrity and did not recross any threshold levels. The failing lines were recrossing the 0.8-volt threshold at the transceiver. Increasing the length of the RAM-to-transceiver segment by 0.5 inches corrected this problem and kept signal delays within acceptable limits.

Approaches other than placing the components in-line were investigated but discarded. Extra signal lengths require additional signal layers and increase the cost of the module and its thickness.

RAM Clock Design

We selected Texas Instruments' CDC2351 clock drivers to handle the RAM clock distribution network. The CDC2351 device has a well-controlled input-to-output delay (3.8 to 4.8 nanoseconds) and 10 drivers in each package that are controlled from one input. The fairly

Table 2
Acceptable RAM Positions Found with HSPICE Simulations

PC Trace Length (Inches)		Write Delay (Nanoseconds)		Read Delay (Nanoseconds)			
Microprocessor to RAM	RAM to Transceiver	Microprocessor to RAM		RAM to Microprocessor		RAM to Transceiver	
		Rise	Fall	Rise	Fall	Rise	Fall
2	7	0.7	2.3	0.9	SR	1.1	1.4
2	8	0.7	2.7	SR	SR	1.5	1.4
2	9	0.6	3.1	SR	SR	1.7	1.5
3	6	0.9	2.1	1.2	1.1	0.9	1.0
3	7	0.9	2.4	1.0	1.1	1.4	1.3
3	8	0.9	2.9	1.0	1.3	1.5	1.3
4	5	1.1	1.8	1.2	1.4	0.9	SR
4	6	1.3	2.2	1.4	1.4	0.9	1.0
4	7	1.2	2.6	1.3	1.4	1.2	1.2
5	4	1.5	1.7	1.5	1.7	SR	SR
5	5	1.4	2.1	1.8	1.7	SR	SR
5	6	1.6	2.4	1.7	1.4	0.9	1.2

Note: Signal reflections recrossing the threshold levels caused unacceptable delays; these entries were discarded.

long delay through the part was beneficial because, as shown in Figure 2, clock delay is needed to achieve adequate setup times. Two CDC2351 clock drivers, mounted back to back on both sides of the PC board, were required to deliver clock signals to the 17 RAMs.

The RAMs were divided into seven groups based on their physical proximity. As shown in Figure 9, there are four groups of three, two groups of two, and a single RAM. Each of the first six groups was driven by two clock driver sections connected in parallel through resistors in series with each driver to achieve good load sharing. The seventh group has only one load, and one CDC2351 section was sufficient to drive it. HSPICE simulation showed that multiple drivers were needed to adequately drive the transmission line and the load. The load connections were made by short equal branches of fewer than two inches each. The length of the branches was critical for achieving good signal integrity at the RAMs.

Data Line Damping

In the ideal world, all signals switch only once per clock interval, allowing plenty of setup and hold time. In the real world, however, narrow pulses often precede valid data transitions. These tend to create multiple reflections superimposed on the edges of valid signals. The reflections can recross the threshold levels and increase the effective delay, thus causing data errors.

Anticipating these phenomena, and having seen their effects in previous designs, designers included series-damping resistors in each cache data line, as shown in Figure 10. Automatic component placement machines and availability of resistors in small packages made mounting 288 resistors on the module a painless task, and the payoff was huge: nearly perfect signals even in the presence of spurious data transitions caused by the microprocessor's architectural features and RAM characteristics. Figure 11 illustrates the handling of some of the more difficult waveforms.

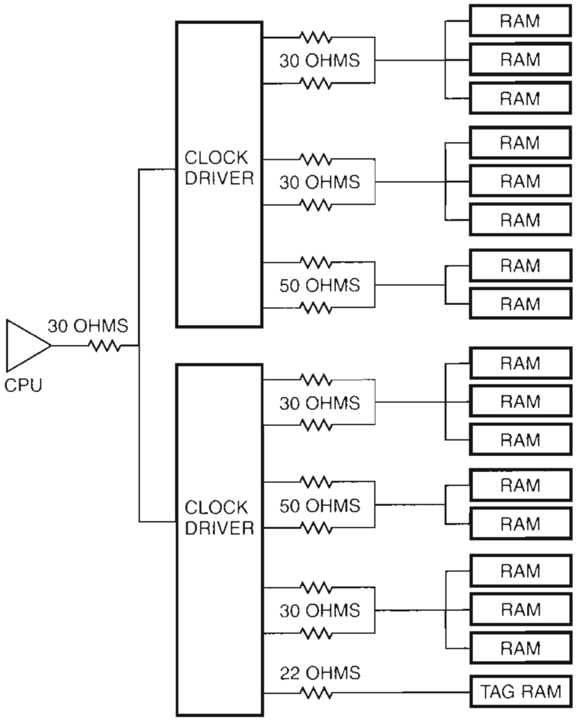


Figure 9
RAM Clock Distribution

Performance Features

This section discusses the performance of the AlphaServer 4100 system derived from the physical aspects of the CPU module design and the effects of the duplicate TAG store.

Physical Aspects of the Design

As previously mentioned, the synchronous cache was chosen primarily for performance reasons. The architecture of the Alpha 21164 microprocessor is such that its data bus is used for transfers to and from main memory (fills and writes) as well as its B-cache.[8] As system cycle times decrease, it becomes a challenge to manage memory transactions without requiring wait cycles using asynchronous cache RAM devices. For example, a transfer from the B-cache to main memory (victim transaction) has the following delay components:

1. The microprocessor drives the address off-chip.
2. The address is fanned out to the RAM devices.
3. The RAMs retrieve data.
4. The RAMs drive data to the bus interface device.
5. The bus interface device requires a setup time.

Worst-case delay values for the above items might be the following:

1. 2.6 nanoseconds[8]
2. 5.0 nanoseconds
3. 9.0 nanoseconds
4. 2.0 nanoseconds
5. 1.0 nanoseconds

Total: 19.6 nanoseconds

Thus, for system cycle times that are significantly shorter than 20 nanoseconds, it becomes impossible

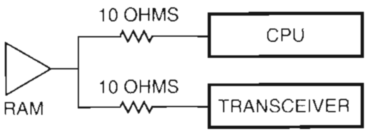
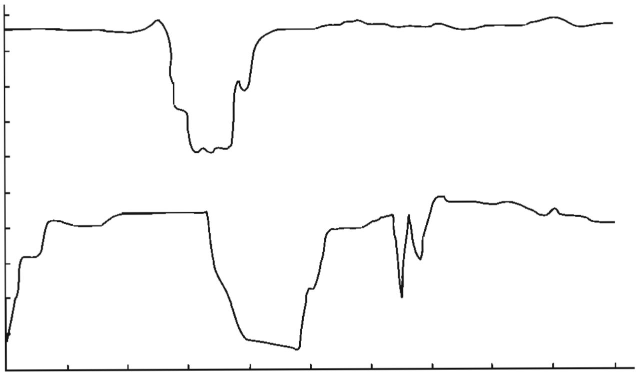


Figure 10
RAM Driving the Microprocessor and Transceiver through 10-ohm Series Resistors


DATA LINE SCALE:
1.00 VOLT/DIVISION,
OFFSET 2.000 VOLTS,
INPUT DC 50 OHMS

TIME BASE SCALE:
10.0 NANOSECONDS/
DIVISION

Figure 11
Handling of Difficult Waveforms

to access the RAM without using multiple cycles per read operation, and since the full transfer involving memory comprises four of these operations, the penalty mounts considerably. Due to pipelining, the synchronous cache enables this type of read operation to occur at a rate of one per system cycle, which is 15 nanoseconds in the AlphaServer 4100 system, greatly increasing the bandwidth for data transfers to and from memory. Since the synchronous RAM is a pipeline stage, rather than a delay element, the window of valid data available to be captured at the bus interface is large. By driving the RAMs with a delayed copy of the system clock, delay components 1 and 2 are hidden, allowing faster cycling of the B-cache.

When an asynchronous cache communicates with the system bus, all data read out from the cache must be synchronized with the bus clock, which can add as many as two clock cycles to the transaction. The synchronous B-cache avoids this performance penalty by cycling at the same rate as the system bus.[2]

In addition, the choice of synchronous RAMs provides a strategic benefit; other microprocessor vendors are moving toward synchronous caches. For example, numerous Intel Pentium microprocessor-based systems employ pipeline-burst, module-level caches using synchronous RAM devices. The popularity of these systems has a large bearing on the RAM industry.[9] It is in DIGITAL's best interest to follow the synchronous RAM trend of the industry, even for Alpha-based systems, since the vendor base will be larger. These vendors will also be likely to put their efforts into improving the speeds and densities of the best-selling synchronous RAM products, which will facilitate improving the cache performance in future variants of the processor modules.

Effect of Duplicate Tag Store (DTAG)

As mentioned previously, the DTAG provides a mechanism to filter irrelevant bus transactions from the Alpha 21164 microprocessor. In addition, it provides an opportunity to speed up memory writes by the I/O bridge when they modify an amount of data that is smaller than the cache block size of 64 bytes (partial block writes).

The AlphaServer 4100 I/O subsystem consists of a PCI mother board and a bridge. The PCI mother board accepts I/O adapters such as network interfaces, disk controllers, or video controllers. The bridge provides the interface between PCI devices and between the CPUs and system memory. The I/O bridge reads and writes memory in much the same way as the CPUs, but special extensions are built into the system bus protocol to handle the requirements of the I/O bridge.

Typically, writes by the I/O bridge that are smaller than the cache block size require a read-modify-write sequence on the system bus to merge the new data with data from main memory or a processor's cache. The AlphaServer 4100 memory system typically transfers data in 64-byte blocks; however, it has the ability to accept writes to aligned 16-byte locations when the I/O bridge is sourcing the data. When such a partial block write occurs, the processor module checks the DTAG to determine if the address hits in the Alpha 21164 cache hierarchy. If it misses, the partial write is permitted to complete unhindered. If there is a hit, and the processor module contains the most recently modified copy of the data, the I/O bridge is alerted to replay the partial write as a read-modify-write sequence. This feature enhances DMA write performance for transfers smaller than 64 bytes since most of these references do not hit in the processor cache.[4]

Conclusions

The synchronous B-cache allows the CPU modules to provide high performance with a simple architecture, achieving the price and performance goals of the AlphaServer 4100 system. The AlphaServer 4100

CPU design team pioneered the use of synchronous RAMs in an Alpha microprocessor-based system design, and the knowledge gained in bringing a design from conception to volume shipment will benefit future upgrades in the AlphaServer 4100 server family, as well as products in other platforms.

Acknowledgments

The development of this processor module would not have been possible without the support of numerous individuals. Rick Hetherington performed early conceptual design and built the project team. Pete Bannon implemented the synchronous RAM support features in the CPU design. Ed Rozman championed the use of random testing techniques. Norm Plante's skill and patience in implementing the often tedious PC layout requirements contributed in no small measure to the project's success. Many others contributed to firmware design, system testing, and performance analysis, and their contributions are gratefully acknowledged. Special thanks must go to Darrel Donaldson for supporting this project throughout the entire development cycle.

References

1. DIGITAL AlphaServer Family DIGITAL UNIX Performance Flash (Maynard, Mass.: Digital Equipment Corporation, 1996), http://www.europe.digital.com/info/performance/sys/unix-svr-flash-9.abs.html.

2. Z. Cvetanovic and D. Donaldson, "AlphaServer 4100 Performance Characterization," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 3–20.

3. G. Herdeg, "Design and Implementation of the AlphaServer 4100 CPU and Memory Architecture," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 48–60.

4. S. Duncan, C. Keefer, and T. McLaughlin, "High Performance I/O Design in the AlphaServer 4100 Symmetric Multiprocessing System," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 61–75.

5. "Microprocessor Report," *MicroDesign Resources,* vol. 8, no. 15 (1994).

6. *IBM Personal Computer Power Series 800 Performance* (Armonk, N.Y.: International Business Machines Corporation, 1995), http://ikc.engr.washington.edu/news/whitep/ps-perf.html.

7. L. Saunders and Y. Trivedi, "Testbench Tutorial," *Integrated System Design,* vol. 7 (April and May 1995).

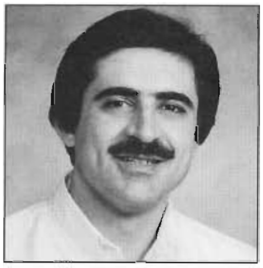
8. *DIGITAL Semiconductor 21164 (366 MHz Through 433 MHz) Alpha Microprocessor Hardware Reference Manual* (Hudson, Mass.: Digital Equipment Corporation, 1996).

9. J. Handy, "Synchronous SRAM Roundup," *Dataquest* (September 11, 1995).

General Reference

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

Biographies



Maurice B. Steinman
Maurice Steinman is a hardware principal engineer in the Server Product Development Group and was the leader of the CPU design team for the DIGITAL AlphaServer 4100 system. In previous projects, he was one of the designers of the AlphaServer 8400 CPU module and a designer of the cache control subsystem for the VAX 9000 computer system. Maurice received a B.S. in computer and systems engineering from Rensselaer Polytechnic Institute in 1986. He was awarded two patents related to cache control and coherence and has two patents pending.

George J. Harris
George Harris was responsible for the signal integrity and cache design of the CPU module in the AlphaServer 4100 series. He joined DIGITAL in 1981 and is a hardware principal engineer in the Server Product Development Group. Before joining DIGITAL, he designed digital circuits at the computer divisions of Honeywell, RCA, and Ferranti. He also designed computer-assisted medical monitoring systems using PDP-11 computers for the American Optical Division of Warner Lambert. He received a master's degree in electronic communications from McGill University, Montreal, Quebec, and was awarded ten patents relating to computer-assisted medical monitoring and one patent related to work at DIGITAL in the area of circuit design.

Andrej Kocev
Andrej Kocev joined DIGITAL in 1994 after receiving a B.S. in computer science from Rensselaer Polytechnic Institute. He is a senior hardware engineer in the Server Product Development Group and a member of the CPU verification team. He designed the logic verification software described in this paper.

Virginia C. Lamere
Virginia Lamere is a hardware principal engineer in the Server Product Development Group and was responsible for CPU module design in the DIGITAL AlphaServer 4100 series. Ginny was a member of the verification teams for the AlphaServer 8400 and AlphaServer 2000 CPU modules. Prior to those projects, she contributed to the design of the floating-point processor on the VAX 8600 and the execution unit on the VAX 9000 computer system. She received a B.S. in electrical engineering and computer science from Princeton University in 1981. Ginny was awarded two patents in the area of the execution unit design and is a co-author of the paper "Floating Point Processor for the VAX 8600" published in this *Journal*.

Roger D. Pannell
Roger Pannell was the leader of the VCTY ASIC design team for the AlphaServer 4100 system. He is a hardware principal engineer in the Server Product Development Group. Roger has worked on several projects since joining Digital in 1977. Most recently, he has been a module/ASIC designer on the AlphaServer 8400 and VAX 7000 I/O port modules and a bus-to-bus I/O bridge. Roger received a B.S. in electronic engineering technology from the University of Lowell.

Roger A. Dame

The AlphaServer 4100 Low-cost Clock Distribution System

High-performance server systems generally require expensive custom clock distribution systems to meet tight timing constraints. These clock systems typically have expensive, application-specific integrated circuits for the bus interface and require controlled etch impedance for the clock distribution on each module in the server system. The DIGITAL AlphaServer 4100 system utilizes phase-locked loop circuits, clock treeing, and termination techniques to provide a cost-effective, low-skew clock distribution system. This system provides multiple copies of the clock, which allows off-the-shelf components to be used for the bus interface, which in turn results in lower costs and a quicker system power-up. Component placement and network compensation eliminated the need for controlled-impedance circuit boards. The clock system design makes it possible to upgrade servers with faster processor options and bus speeds without changing components.

Every digital computer system needs a clock distribution system to synchronize electronic communication. The primary metric used to quantify the performance of a clock distribution system is clock skew. Synchronous systems require multiple copies (outputs) of the same clock, and clock skew is the unwanted delay between any two of the copies. In general, the lower the skew, the better the clock system. Clock skew is one of several parameters that affect bus speed. Bus length, bus loading, driver and receiver technology, and bus signal voltage swing also affect bus speed. If problems arise that jeopardize meeting timing goals, though, these additional parameters are difficult to change because of physical and architectural constraints.

The DIGITAL AlphaServer 4100 clock distribution system is a compact, low-cost solution for a high-performance midrange server. The clock system provides more copies of the clock than machines in the same class typically need. The distribution system allows expansion on those module designs where more copies of the clock are needed with minimal skew. The system is based on a low-cost, off-the-shelf phase-locked loop (PLL) as the basic building block. The simple application of the PLL alone would not provide low clock skew, though. Signal integrity techniques and trade-offs were needed to manage skew throughout the system. The technical challenges were to design a low-cost system that would (1) require only a small area on the printed wiring boards (PWBs), (2) be adaptable to various speed grades (options) of CPUs, and (3) have good performance, i.e., low skew. This paper discusses the techniques used to optimize the performance of an off-the-shelf PLL-based clock distribution system.

Design Goals

Based on its experience with previous platform designs, the design team considered a clock skew under 10 percent of the bus cycle time a reasonable target for a midrange server system. The cycle time design target of the AlphaServer 4100 system was 15 nanoseconds (ns); consequently, the skew goal was 1.5 ns or less. This goal would allow a total of 13.5 ns for clock to output of the transmitting module (Tco) (the time the transmitting module needs to drive data to a stable state from a clock edge); setup and hold time requirements for the receiving module (the minimum time that data needs to be stable at the receiver [flop] before and after the local clock edge); and bus settling time. The following is a breakdown of the timing based on the selection of components for the bus interface:

Bus cycle	15.0 ns
Transmitting module (Tco)	5.1 ns
Setup and hold time for the receiving module	1.5 ns
Clock skew	1.5 ns
Time allocated for bus settling	6.9 ns

The selection of components was based on availability, speed, cost, and size. The goal was to eliminate the need for costly application-specific integrated circuits (ASICs) and still meet the critical timing performance.

The AlphaServer 4100 bus is a simple distributed bus, 305 millimeters (mm) long, with 10 loads (modules) and parallel termination at both ends. The first-order estimate of bus settling time assumed one full reflection or twice the loaded velocity of propagation delay end to end. The estimate took into account bus timing optimization, which is discussed later in this paper. It was also estimated that 25 copies of the clock would be required for the processor modules, and 46 copies of the clock would be required for certain memory modules (synchronous dynamic random-access memory [SDRAM]–based designs). Only the rising edge of the clock could be used for critical timing. If the falling edge were used for latches, then clock skew would dramatically increase because of the duty cycle distortion associated with PLLs. The memory module design allowed very little space for clock circuitry and needed more copies of the clock than any other module design in the system. Further, the physical size of the memory module determined the actual size of the server box. Trade-offs had to be made in the design and timing to make the off-the-shelf solution work. The key goal was to optimize the solution to get the worst-case skew as close as possible to the 1.5 ns estimated goal and to find system trade-offs to allow higher module-to-module skew for a 15 ns bus.

A survey of custom clock circuits available within DIGITAL and off-the-shelf, commercially available PLLs suggested that a custom circuit was required. Unfortunately, the circuits that would be available within our project schedule were costly, consumed far too much circuit board area, required emitter-coupled logic (ECL) or positive emitter-coupled logic (PECL) inputs, and dissipated substantial power. The best off-the-shelf solution was cost-effective, required less space than custom circuits, and provided adequate fan-out. The skew performance, however, ranged from 2 ns to 4 ns, which exceeded the design goal. Given the project time constraints and the design benefits of the off-the-shelf solution, it was paramount that we make the off-the-shelf solution work.

Bus Trade-offs

The design philosophy of using stock components for the bus interface allowed some latitude in the bus design. Typical bus interfaces use large ASICs, each handling up to 50 percent of the data bits. Such a design results in a relatively long dispersion etch from the connector to the ASIC. These devices can range in size from 200 to 400 pins and can require up to 38 mm of etch from the ASIC to the connector. SPICE simulations demonstrated that the length of each module's dispersion etch or bus "stubbing" had a profound effect on bus settling time.[1] Figure 1 shows bus settling time (worst-case driver-receiver combination) as a function of module dispersion etch. The bus trunk length was fixed at 305 mm.

The designers used an 18-bit-wide transceiver in a low-profile surface mount package with a pin pitch of 0.5 mm. The location of the I/O pins for the bus connections on the interface transceiver (located on the same side of the package, which allows the device to be placed very close to the bus connector) and the connector pitch facilitated short dispersion etch (less than 13 mm). This design decreased by 1 ns the settling time typically found on ASIC-based interfaces with comparable trunk lengths and loading.

Bus termination is another parameter that designers can manipulate to further improve settling time. We used parallel terminators at both ends of the bus on the AlphaServer 4100 system. The bus protocol has two features that allow aggressive termination, approaching the unloaded impedance of the trunk. We placed an anticontention cycle between the module that relinquishes the bus and the module that begins to drive the bus. This arrangement reduces the possibility for driver contention (stress) as well as the possibility of generating ringing on the bus caused by large changes in current after contention. The bus "parking" feature forces the last driving module to continue driving the bus to

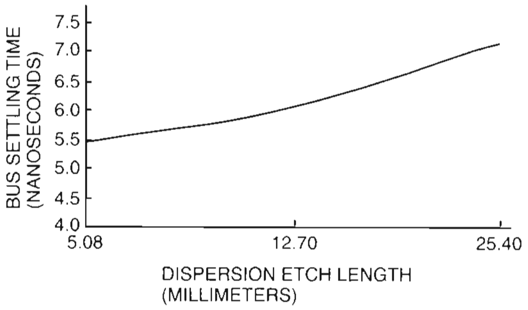


Figure 1
Bus Settling Time As a Function of Dispersion Etch Length

a logic state during long idle times until another module wants to use the bus. Without this feature, the bus would settle at the terminator Thevenin voltage if no modules were driving the bus. Both protocols allow for Thevenin voltage to be close to the thresholds of the receivers. Normally this is avoided if the bus is left idle, because the receivers can go metastable, i.e., arrive at the unstable condition where its input voltage is between its specified logic 0 and logic 1 voltage levels, resulting in uncontrolled oscillation. Centering the Thevenin voltage in the normal full voltage swing had two advantages: (1) it balanced the settling time for both transitions, and (2) it reduced the driver current. The reduced driver current allowed for a lower Thevenin resistance, which brought the terminators closer to the unloaded (no modules) impedance of the bus, thus ensuring that the bus would settle within 6 ns.

The Basic Building Block

Texas Instruments' CDC586 clock distribution circuit was chosen as the basic building block for the system because of its low cost and functionality. The device has a fan-out of 12 outputs with a single compensation loop and a frequency range of 25 megahertz (MHz) to 100 MHz, and is a 3.3-volt (V) bipolar complementary metal-oxide semiconductor (BiCMOS) part. Process skew is 1 ns between any two parts with the same reference input clock, and root mean square (RMS) jitter is 25 picoseconds (ps).[2] The CDC586 has a built-in loop filter, which reduces the number of support components. Unlike custom clock circuits with multiple, independent compensation loops, the simple, single loop design required critical attention to the layout of each module design to ensure the best possible skew performance. The circuit board layout designer had to determine the maximum etch length from the PLL to the receiver. All copies of the clock had to be precisely matched in length to the maximum length found, and routed on the same etch layer with 0.51 mm (20 mil) spacing to other etches and minimum etch crossovers from other etch layers on dual strip-line lay-ups. Typical strip-line etch in multilayer PWBs is a signal layer that has reference planes, usually assigned to power or ground, in the layer above and the layer below. This design allows better impedance control and eliminates cross talk from other signal layers. PWB thickness and cost constraints often result in modified forms on the inner layers, however. Dual strip-line etch is often used in these cases. This design consists of two signal layers sandwiched between reference planes in the layers above and below. Generally the dielectric thickness between the two signal layers is greater than the dielectric thickness between either signal layer and its related (nearest) reference plane to minimize cross talk between the two signal layers. Figure 2 illustrates a typical application.

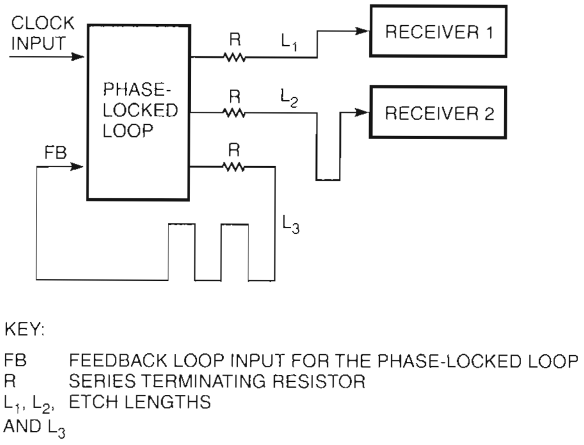

CLOCK INPUT
PHASE-LOCKED LOOP
FB
R
R
R
L_1
L_2
L_3
RECEIVER 1
RECEIVER 2
KEY:
FB FEEDBACK LOOP INPUT FOR THE PHASE-LOCKED LOOP
R SERIES TERMINATING RESISTOR
L_1, L_2, AND L_3 ETCH LENGTHS

Figure 2
Typical Phase-locked Loop Connection

Etch Layout

The PWB lay-ups used on various modules in the AlphaServer 4100 system contain microstrip etch (surface etch) and dual strip-line etch. Ideally, single strip-line etch would be optimum for clock etch; however, it requires more layers at higher cost for PWB material. One drawback to dual strip-line lay-ups is etch crossover. A crossover is a point along an etch trace where another etch, one on a different layer not separated by a reference plane, crosses. The crossover forms small capacitance patches, which can load the clock etch and affect its impedance and velocity of propagation. The result is additional skew from clock etch to clock etch. Designers avoided crossovers on all clock etch, and the design does not permit parallel etch on the other layer within the dual strip-line, which could induce cross talk.

Figure 2 shows matched etch lengths L_1, L_2, and L_3. On some module designs, this etch can be fairly long. The layout designers would generally "serpentine" or "trombone" these long etch runs to comply with the aforementioned layout rules. Spacing between the loops on the same etch run in the serpentine or trombone is critical. If the spacing is too close, then coupling will occur, thus changing the velocity of propagation as well as signal quality. Designers used simulation to determine a minimum etch-to-etch spacing for each PWB lay-up. The maximum allowable cross-talk noise level for any minimum spacing was 400 millivolts (mV). This level is within the maximum transistor-transistor logic (TTL) low-state level of 800 mV. Larger spacings were used where no other layout rules would be affected.

The Use of External Series Terminating Resistors

External series terminating resistors (also called terminators), denoted by R, are used at the source (see Figure 2). Although Texas Instruments offers another version of the PLL, namely CDC2586, which has built-in series terminators, the AlphaServer 4100 designers did not use this variation for the following reasons:

- Some forms of clock treeing (a method of connecting multiple receivers to the same clock output) require multiple source terminators.
- The nominal value for the internal series terminator was not optimum for the target impedance of the PWBs.
- The tolerance of the internal series terminators over the process range of the part could be as high as 20 percent compared to 1 percent for external resistors.

Local Power Decoupling

PLLs are analog components and are susceptible to power supply noise. One major point source for noise is the PLL itself. Most applications require all 12 outputs to drive substantial loads, which generates local noise. A substantial number of local decoupling capacitors (one for every four output pins) and short, wide dispersion etch on the power and ground pins of the PLL were required to help counter the noise. Designers also used tangential vias to minimize parasitic inductance, which can severely reduce the effectiveness of the decoupling capacitors. Typical surface mount components have dispersion etch, which connects the surface pad to a via. Tangential vias attach directly to the pad and eliminate any surface etch that can act like inductance at high frequency. The PLLs were also located away from other potential noise sources such as the Alpha microprocessor chip.

Analog Power Supply Filter

The most important external circuit to the PLL is the low-pass filter on the analog power pins. Typically, PLL designs have separate analog and digital power and ground pins. This allows the use of a low-pass filter to prevent local switching noise from entering the analog core of the PLL (primarily the voltage-controlled oscillator [VCO]). If a filter is not used, then large edge-to-edge jitter will develop and will greatly increase clock skew. Most PLL vendors suggest filter designs and PWB layout patterns to help reduce the noise entering the analog core. The CDC586 PLL was introduced at the beginning of the AlphaServer 4100 design, and the vendor had not yet specified a filter for the analog power input. It is important to note that if any new PLL is considered and preliminary vendor specifications do not include details about the analog power, the designer should contact the vendor for details.

Two forms of low-pass filters were considered: L-C and R-C. The L-C filter consists of a series inductor L from the power source to the analog power pins of the PLL and a capacitor C from the same power pins to ground. The R-C filter consists of a series resistor R from the power source to the analog power pins of the PLL and a capacitor C from the same power pins to ground.

The L-C filter can be implemented in two ways: (1) by using a surface mount inductor and (2) by using a length of etch for the inductor. In either case, the Q of the circuit has to be kept low to prevent oscillation. Q is a dimensionless number referred to as the quality factor and is computed from the inductance L and resistance R (in this case the inductor's resistance) of a resonant circuit using the formula $Q = \omega L/R$, where ω equals $2\pi f$, and f is the frequency. A low-value resistor in series with the inductor can help. Extreme care should be taken if the length-of-etch (used to generate inductance) implementation is considered. The etch must be strip-line-etch isolated from any other adjacent etch or etch on other layers not separated by power or ground planes. A two-dimensional (2-D) modeling tool should be used to calculate the length of etch needed to get the proper inductance value for the filter. Simple rules of thumb for inductance will not work with reference planes (i.e., power and ground planes).

The R-C filter is limited to PLLs with moderately low current draw on the analog power pins. The current generates an IR drop (the voltage drop caused by the current through the resistor) across the resistor R. Typical PLL analog power inputs require less than 1 milliamp (mA), which would allow a reasonable value resistor R. Two capacitors should be used in the R-C type filter: a bulk capacitor for basic filter response and a radio frequency (RF) capacitor to filter higher frequencies. Bulk capacitors are any electrolytic-style capacitor 1 microfarad (μF) or greater. These capacitors have intrinsic parasitics that keep them from responding to high-frequency noise. The benefit of the L-C filter is that, although a single capacitor can be used (two are still suggested with this style filter), the reactance of the inductor increases with frequency and helps block noise. Both filter styles were used in the AlphaServer 4100 system.

System Distribution Description

The AlphaServer motherboard has four CPU slots, eight memory slots, and an I/O bridge module slot. Each module in the system, including the motherboard, has at least one PLL. The starting point of the system is the CPU that plugs into CPU slot 0. Each CPU module has an oscillator and a buffer to drive the main system distribution, but the CPU that plugs into slot 0 actually drives the system distribution. A PLL on the motherboard receives the clock source generated by the CPU in slot 0 and distributes low skew copies of the clock to each module slot in the system. Each module in the system has one and in some cases two PLLs to supply the required copies of the clock locally. Figure 3 shows the basic system flow of clocks.

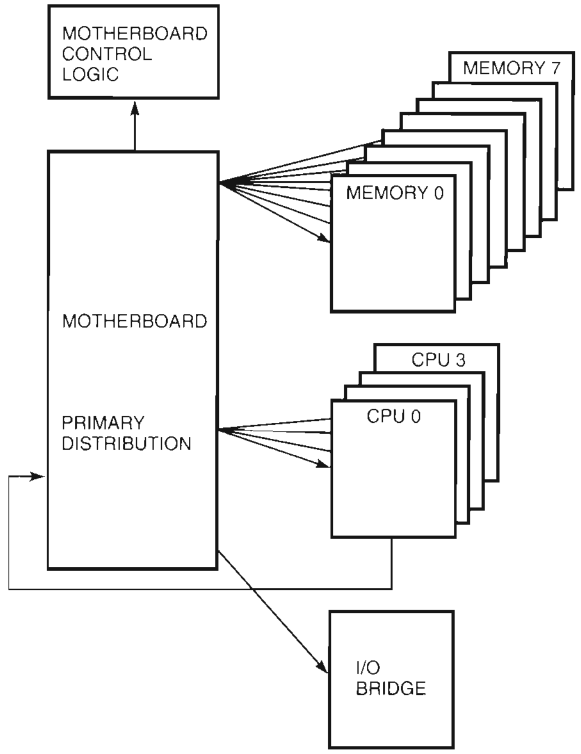


Figure 3
System Clock Flow Diagram

The Alpha microprocessor used on all CPU options for the AlphaServer 4100 system has its own local clock circuitry. The microprocessor uses a built-in digital PLL that allows it to lock to an external reference clock at a multiple of its internal clock.[3] In the context of the AlphaServer 4100 system, the reference clock is generated by the local clock distribution system. The AlphaServer 4100 is fully synchronous.

Each CPU in the system has two clock sources: one for the bus distribution (system cycle time) and one for the microprocessor. This design may appear to be a costly one, but this approach is extremely cost-effective when field upgrades are considered. When new, faster versions of the Alpha microprocessor become available, new CPU options will be introduced. To remain synchronous, the Alpha microprocessor internal clocks need to run at a multiple of the system cycle time. Although the system cycle time goal is 15 ns, the cycle time needs to be adjusted to the speed of the CPU option used. Placing the bus oscillator, which drives the primary PLL for the clock system (cycle time), on the CPU module and designing the clock distribution system to function over a wide frequency range makes field upgrades as simple as replacing the CPU modules. The motherboard does not need to be changed.

Skew Management Techniques

The AlphaServer 4100 system had four design teams. Each team was assigned a portion of the system. Signal integrity techniques had to be developed to keep the skew across the system as low as possible. These techniques were structured into a set of design rules that each team had to apply to their portion of the design. To develop these rules, designers explored several areas, including impedance range, termination, treeing, PLL placement, and compensation.

Impedance Range

Controlled impedance (+/−10 percent from a target impedance) raises the PWB cost by 10 percent to 20 percent, depending on board size. Each raw PWB has to be tested and documented by the PWB suppliers, which results in a fixed charge for each PWB, regardless of size. Therefore, smaller PWBs have the highest cost burden. The AlphaServer 4100 uses relatively small daughter cards. Since low system cost was a primary goal, noncontrolled impedance PWBs had to be considered. Unfortunately, allowing the PWB impedance range (over process) to spread to greater than +/−10 percent makes the task of keeping clock skew low more difficult. Specification of mechanical dimensions with tolerances was the only way to provide some control of the impedance range with no additional costs.

Table 1 contains the results of simulations performed using SIMPEST, a 2-D modeling tool developed by DIGITAL, for a six-layer PWB used on one of the AlphaServer 4100 modules. The PWB dimensions and tolerances specified to the vendors were used in the simulations. The dielectric constant, the only parameter not specified to the vendor, ranged from 3.8 to 5.2, which overlaps the typical industry-published range of 4.0 to 5.0 for FR4-type material (epoxy-glass PWB).[4] Since our PWB material acceptance with the vendor is based on meeting dimension tolerances, we used the 6σ impedance range on all SPICE simulations, thus ensuring that all acceptable PWB material would work electrically.

Table 2 shows the impedance range for a controlled impedance PWB for the target impedance reported in

Table 1
Vendor Impedance Ranges Specifying Dimensions Only

	4σ Yield	6σ Yield
Mean target impedance	71 ohms	71 ohms
Impedance range	62 ohms to 83 ohms	57 ohms to 89 ohms

Table 2
Vendor Impedance Range for an Impedance Tolerance of +/−10 Percent

	+/−10 Specification Range
Mean target Impedance	71 ohms
Impedance range	64 ohms to 78 ohms

Table 1. The difference in impedance range between specifying dimensions and impedance is −7 ohms to 11 ohms. The simulations suggested that the range differences have a minor impact on signal behavior.

The target impedance was based on nominal dimensions and dielectric constant. The target of 71 ohms was chosen to optimize routing density and to keep the layer count down for most designs. Another advantage was that keeping the minimum impedance above 50 ohms would minimize loading. The impedance range covers the full mechanical dimensions and dielectric constant ranges. Properly implemented, the PLLs would effectively eliminate local etch delay module to module over the full process range of the PWBs. The main challenge was to adequately terminate without sacrificing skew performance at the extreme process range (6σ) of the PWB material.

Termination

The designers used series termination on all clocks in the system. Parallel terminators would have exceeded the drive capability of the CDC586. Diode clamping was not practical when so many copies of the clock were required because of PWB surface area constraints. Normally, the optimal termination value is one that provides critical damping for the case where the driver's impedance is the lowest and the etch impedance is the highest. Designers can then make adjustments at the other extreme corner (high driver impedance and low etch impedance) to avoid nonmonotonic behavior such as plateaus. This generally introduces slope delay uncertainty at the slow corner (high driver impedance and low etch impedance), which can be substantial. To minimize this effect, designers selected terminator values that allow overshoot and some bounce-away from the threshold region at the extreme process corner. Overshoot can reach the maximum specified alternating current (AC) input of the receivers over the worst-case process range. Some receivers have built-in diode clamping to their power supply rails as a result of ESD circuits in their input structures (ESD circuits are used for static discharge protection). In these cases, the clock signal is clamped, which in turn dampens bounce. The injection currents caused by clamping would be tested in SPICE simulations to be sure that the parts were not stressed. If the tests indicated stressed parts, designers would adjust the terminator value accordingly.

Treeing

Treeing is a method of distributing clocks from a single source driver to many receivers. This practice, which is well known to memory designers, was used on the AlphaServer 4100 memory modules, bus interface logic, and primary distribution clocks on the motherboard. The designers used two basic forms of treeing: the balanced H tree and the shared output tree. The balanced H tree is best suited for fixed loads (receivers) of the same type (i.e., memories, transceivers, etc.). A single, series-terminated clock output feeds a trunk line to a via and then branches to each load. Each branch is equal in length. The total compensated path includes the pre-terminator stub, the main trunk, and the branch extending to the load. Figure 4 illustrates the clock treeing topology. The shared output tree was used where various module configurations could alter clock loading. Specifically, the distribution on the motherboard is restricted to one PLL to keep the clock skew low. Consequently, some outputs needed to drive more than one slot. A single output driver drove two terminators—one for each load. The low driver impedance isolated reflections from perturbing a module when a module slot was left open.

PLL Placement

Placement of the PLL on each module is critical. Figure 5 is a simplified view of the primary distribution up to and including the PLL on a module. The AlphaServer

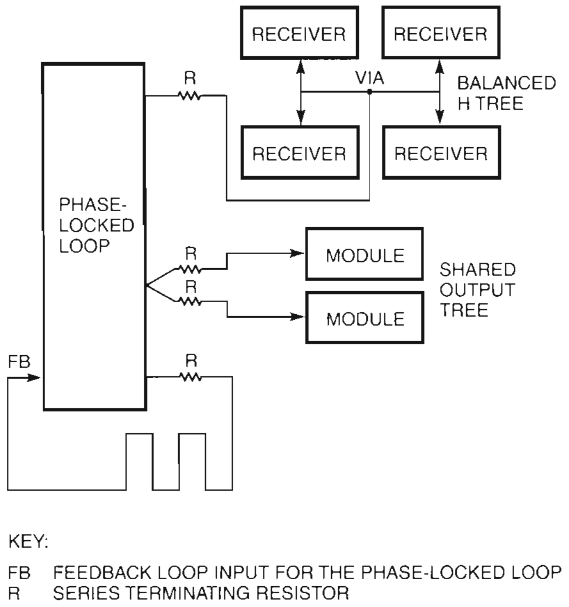


Figure 4
Clock Treeing

4100 system has two types of module connectors: a Metral connector (Futurebus+-style connector) is used on the CPU modules and the I/O bridge module, and an Extended Industry Standard Architecture (EISA) connector is used on the memory modules. Intrinsic delay on these connectors could differ substantially depending on pinning and the signal-to-return ratio in the application. The Metral connector is a right-angle, two-piece connector with four rows of pins: rows A, B, C, and D. The row A pins are the shortest, and the row D pins are the longest. The EISA connector is an edge connector with two rows of pins with minor length differences pin to pin on either side of the connector. Designers had to balance the pinning of these connectors for the clock circuits in such a way that the module-to-module skew would not be affected. The Metral connector was pinned to replicate the loop inductance of the EISA connector.

Dispersion etch is required on each module to connect the PLL to the connector. This etch can have different impedance and velocity of propagation from module to module as a result of PWB process range, which translates into additional module-to-module clock skew. Designers can deal with this problem in two ways.

First, adding the same dispersion length L_3 (see Figure 5) to the compensation loop L_2 nulls this error. This becomes obvious if you look at the PLL's basic function. The insertion delay T_{id} from the clock-in pin of the PLL to the input pin of the receiver is approximately 0 ns if $L_1 = L_2$, or

$$T_{id} = (T_{L_1} + T_{L_3}) - T_{L_2}.$$

For $T_{L_1} = T_{L_2}$ (equal etch lengths), $T_{id} = T_{L_3}$.

Adding T_{L_3} to the compensation path yields

$$T_{id} = (T_{L_1} + T_{L_3}) - (T_{L_2} + T_{L_3}).$$

For $T_{L_1} = T_{L_2}$ (etch equal lengths), $T_{id} = 0$ ns,

where

T_{id} = the insertion delay from the connector pin to the receiver input

T_{L_1} = the etch delay from the PLL output to the receiver input

T_{L_2} = the etch delay of the PLL compensation loop

T_{L_3} = the dispersion etch delay connector to the clock-in of the PLL.

One drawback to this method is that the etch lengths could get fairly large, which would result in edge rate degradation. AlphaServer 4100 designers did not use this placement method on the current set of modules; however, they will consider using it on new designs that require a different location for the PLL.

The second way of dealing with the dispersion etch from the module connector to the clock-in pin of the PLL is to make the dispersion etch very short and to take a skew penalty over the PWB process. Placement studies on the various module designs suggest that a 25-mm dispersion etch would allow reasonable placement of PLLs. The additional skew is just under 50 ps, based on a velocity of propagation range of 5.59 ps/mm to 7.36 ps/mm.

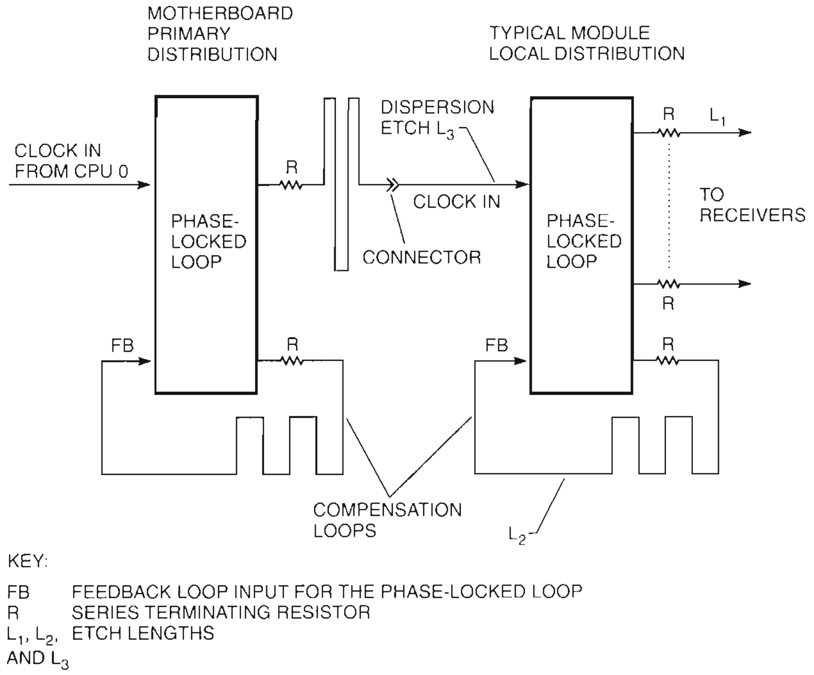


Figure 5
Primary Distribution

Compensation

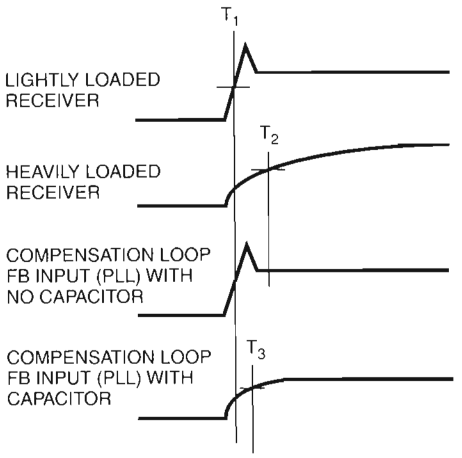
Some modules have a wide variety of circuits receiving clocks that, because of input loading, do not balance well with the various treeing methods. Designers used dummy capacitor loading to help balance the treeing. This approach was particularly useful on memory modules, which could be depopulated to provide different options using the same etch. Surface-mount pads were added to the etch such that if the depopulated version were built, a capacitor could be added to replicate the missing load on the tree, thus keeping it in balance. The CPU modules have a wide variety of clock needs, which results in two forms of skew: (1) load-to-load skew at the module and (2) control logic–to–CPU skew, for control logic located on the motherboard. The local load-to-load skew is acceptable because only one PLL is used and the output-to-output skew is only 500 ps. Motherboard-to-CPU control logic skew, though, is critical because of timing constraints.

Dummy capacitor loading at each lightly loaded receiver would have reduced the skew, but to compensate for just one heavily loaded receiver would have required many capacitors. PWB surface area and the requirement of simplicity dictated the need for an alternative. The solution was to keep the clock edges as fast as possible (by adjusting the series terminators) and to add a compensation capacitor at the input (the feedback [FB]) of the PLL's compensation loop. This effectively reduced the skew from the slowest load on the CPU to the control logic on the motherboard. Figure 6 shows the disparity between light and heavy loading from T_1 to T_2. Without feedback compensation, the PLL self-adjusts to the lightly loaded receiver. This adjustment results in skew T_1 to T_2 from the heavy load to the control logic on the motherboard. A capacitor on the FB input of the PLL split the difference between T_3 to T_2 and T_3 to T_1 and minimized the perceived skew.

Skew Target

Designers generated the worst-case module-to-module clock skew specification for the AlphaServer 4100 from vendor specifications, SPICE simulations, and bench tests using the techniques discussed in this paper. The worst-case skew goal is 2.2 ns and is summarized in Table 3.

The reader will note that eight times the vendor's specification may appear to be a rather conservative specification. The use of this value was based on two concerns: (1) the PLL was new at the time, and experience suggested that the vendor's specification was aggressive; and (2) some level of padding was required if the exception to the rules was needed. Actual system testing bore out these concerns. The vendor had to relax the jitter specification from 25 ps to 70 ps RMS, and there were some difficulties getting good load balance. The specification did not change, however. Reassessing the allocated bus settling time yields the following:


T1
LIGHTLY LOADED RECEIVER
T2
HEAVILY LOADED RECEIVER
COMPENSATION LOOP FB INPUT (PLL) WITH NO CAPACITOR
T3
COMPENSATION LOOP FB INPUT (PLL) WITH CAPACITOR

KEY:

T_1	LIGHTLY LOADED RECEIVER CLOCK EDGE TIME (REFERENCE)
T_2	HEAVILY LOADED RECEIVER CLOCK EDGE TIME
T_3	COMPENSATION LOOP FB INPUT EDGE TIME WITH CAPACITOR
FB	FEEDBACK LOOP INPUT FOR THE PHASE-LOCKED LOOP

Figure 6
Feedback Loop Compensation

Bus cycle	15.0 ns
Transmitting module (Tco)	5.1 ns
Setup and hold time for the receiving module	1.5 ns
Clock skew	2.2 ns
Time allocated for bus settling	6.2 ns

SPICE simulations for a fully loaded bus with the worst possible driver receiver position yielded a bus settling time of 5.7 ns. The relaxed skew of 2.2 ns maximum was acceptable for the design.

Comparative Analysis

A comparison of clock distribution systems between two other platforms best summarizes the AlphaServer 4100 system. The AlphaServer 4100 has a price and performance target between those of the AlphaServer 2100 and the AlphaServer 8400 systems. Table 4 compares the basic differences among these systems relating to clock distribution for a CPU module from each platform.

Both the AlphaServer 2100 and the AlphaServer 8400 systems have large custom ASICs for their module's bus interface. The AlphaServer 4100 and the AlphaServer 8400 systems have bus termination; the AlphaServer 2100 system does not. Allowing a bus to

Table 3
Worst-case Clock Skew

Stage	Source	Skew Component
Motherboard	Out-to-out skew	500 ps (vendor specification)[2]
Inputs to modules	Load mismatch	100 ps (simulation/bench test)
Module to module	PLL process	1,000 ps (vendor specification)[2]
Inputs to receivers	Load mismatch	200 ps (simulation/bench test)
Inputs to receivers	PLL jitter	400 ps (eight times the vendor specification)[2]
Total clock skew		2,200 ps = 2.2 ns

Table 4
Clock Distribution Comparison of Three Platforms

	AlphaServer 2100 System	AlphaServer 4100 System	AlphaServer 8400 System
Bus width	128 + ECC	128 + ECC	256 + ECC
Bus speed	24 ns	15 ns	10 ns
Clock skew	1.5 ns	2.2 ns (max.)	1.1 ns (max.)
Inputs requiring clocks	10	25	14
Clock drivers used	12	13	11
Number of clock phases	4	1	1

settle naturally (with no termination), as in the case of the AlphaServer 2100 system, requires a tighter skew budget from the clock system. The trade-off is higher cost, power, and PWB area for lower bus speed. Higher performance systems, such as the AlphaServer 8400 and AlphaServer 4100 systems, generally require faster bus speeds with terminators. The AlphaServer 4100 has shorter bus stubbing (module transceiver to connector dispersion etch) and slower bus speed than the AlphaServer 8400, which allows larger skew (as a percentage of the bus speed).

Table 5 is a comparison of board area needed and cost for the clock system. Designers analyzed an entry-level system consisting of one CPU module, one memory module, and one I/O bridge or interface module. The board area shows the space required by the active components only (the digital phase-locked loops, PLLs, drivers, etc.).

Both Tables 4 and 5 show that the clock system design for the AlphaServer 4100 system requires only one-third the space of either the AlphaServer 2100 system or the AlphaServer 8400 system at a fraction of the cost and distributes more copies of the clock.

Conclusions

An effective, low-cost, high-performance clock distribution system can be designed using an off-the-shelf component as the basic building block. DIGITAL AlphaServer 4100 system designers accomplished this by optimizing the bus and developing simple techniques structured in the form of design rules. These rules are

- Use positive edges for critical clocking.
- Match delay through different connectors using appropriate pinning.
- Use a fixed dispersion etch length from the connector to the PLL.
- Route and balance all clock nets on the same PWB layer.
- Minimize adjacent-layer crossovers and maximize spacings.
- Use minimum value terminators.
- Use tree and loop compensation where needed.
- Use conservative local decoupling and a low-pass filter on the PLL (analog power).

Table 5
Board Utilization and Cost Comparison

	AlphaServer 2100 System	AlphaServer 4100 System	AlphaServer 8400 System
Board area used*	352.8 square centimeters	111.4 square centimeters	371.3 square centimeters
Normalized cost	1.00	0.46	4.40

*Note that these measurements do not include decoupling capacitors and terminators.

The worst-case lab measurement of clock skew between any two modules in a fully configured system was 1.1 ns, which is well within the 2.2 ns calculated maximum skew.

Acknowledgments

Terry Skrypek and Bruce Alford assisted with the prototyping and measurements. Cheryl Preston, Andy Koning, Steve Coe, George Harris, and Larry Derenne worked with the designers to ensure compliance with the signal integrity rules. Darrel Donaldson, Don Smelser, Glenn Herdeg, and Dan Wissell provided invaluable technical guidance.

Note and References

1. 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 at Berkeley.

2. CDC—*Clock Distribution Circuits,* Data Book (Dallas, Tex.: Texas Instruments Incorporated, 1994).

3. *Alpha 21164 Microprocessor Hardware Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, September 1994).

4. C. Guiles, *Everything You Ever Wanted to Know About Laminates. . . But Were Afraid to Ask,* 4th ed. (Maitland, Fla.: Arlon, Inc., January 1989).

Biography

Roger A. Dame
A principal signal integrity engineer in the Midrange Servers group, Roger Dame is currently working on the AlphaServer 4100 project. During the 10 years he has been with this group, he has also contributed to the VAX 6000, VAX 5800, VAX 7000, DEC 7000, and DEC 10000 projects. In earlier work at DIGITAL, in the Industrial Products group, he developed analog-to-digital process control system interfaces. Roger joined DIGITAL in 1971. He holds an A.S.E.E.T. degree from Springfield Technical Community College and a B.S.E.E.T. (summa cum laude) from Central New England College. Roger is coinventor of the laser bus used in the DEC 7000 and DEC 10000 systems.

Glenn A. Herdeg

Design and Implementation of the AlphaServer 4100 CPU and Memory Architecture

The DIGITAL AlphaServer 4100 system is Digital Equipment Corporation's newest four-processor midrange server product. The server design is based on the Alpha 21164 CPU, DIGITAL's latest 64-bit microprocessor, operating at speeds of up to 400 megahertz and beyond. The memory architecture was designed to interconnect up to four Alpha 21164 CPU chips and up to four 64-bit PCI bus bridges (the AlphaServer 4100 supports up to two buses) to as much as 8 gigabytes of main memory. The performance goal for the AlphaServer 4100 memory interconnect was to deliver a four-multiprocessor server with the lowest memory latency and highest memory bandwidth in the industry by the end of June 1996. These goals were met by the time the AlphaServer 4100 system was introduced in May 1996. The memory interconnect design enables the server system to achieve a minimum memory latency of 120 nanoseconds and a maximum memory bandwidth of 1 gigabyte per second by using off-the-shelf data path and address components and programmable logic between the CPU and the main memory, which is based on the new synchronous dynamic random-access memory technology.

The DIGITAL AlphaServer 4100 system is a symmetric multiprocessing (SMP) midrange server that supports up to four Alpha 21164 microprocessors. A single Alpha 21164 CPU chip may simultaneously issue multiple external accesses to main memory. The AlphaServer 4100 memory interconnect was designed to maximize this multiple-issue feature of the Alpha 21164 CPU chip and to take advantage of the performance benefits of the new family of memory chips called synchronous dynamic random-access memories (SDRAMs). To meet the best-in-industry latency and bandwidth performance goals, DIGITAL developed a simple memory interconnect architecture that combines the existing Alpha 21164 CPU memory interface with the industry-standard SDRAM interface.

Throughout this paper the term latency refers to the time required to return data from the memory chips to the CPU chips—the lower the latency, the better the performance. The AlphaServer 4100 system achieves a minimum latency of 120 nanoseconds (ns) from the time the address appears at the pins of the Alpha 21164 CPU to the time the CPU internally receives the corresponding data from any address in main memory. The term bandwidth refers to the amount of data, i.e., the number of bytes, transferred between the memory chips and the CPU chips per unit of time—the higher the bandwidth, the better the performance. The AlphaServer 4100 delivers a maximum memory bandwidth of 1 gigabyte per second (GB/s).

Before introducing the DIGITAL AlphaServer 4100 product in May 1996, the development team conducted an extensive performance comparison of the top servers in the industry. The benchmark tests showed that the AlphaServer 4100 delivered the lowest memory latency and the highest McCalpin memory bandwidth of all the two- to four-processor systems in the industry. A companion paper in this issue of the *Journal*, "AlphaServer 4100 Performance Characterization," contains the comparative information.[1]

This paper focuses on the architecture and design of the three core modules that were developed concurrently to optimize the performance of the entire memory architecture. These three modules—the motherboard, the synchronous memory module, and the no-external-cache processor module—are shown in Figure 1.

Motherboard

The motherboard contains connectors for up to four processor modules, up to four memory module pairs, up to two I/O interface modules (four peripheral component interconnect [PCI] bus bridge chips total), memory address multiplexers/drivers, and logic for memory control and arbitration.[2] All control logic on the motherboard is implemented using simple 5-ns 28-pin programmable array logic (PAL) devices and more complex 90-megahertz (MHz) 44-pin programmable logic devices (PLDs) clocked at 66 MHz. Several motherboards have been produced to support various numbers of processor modules, memory modules, and I/O interface modules. The AlphaServer 4100 supports one to four processor modules, one to four memory module pairs (8-GB maximum memory), and one I/O interface module (up to two PCI buses).[3]

Synchronous Memory Module

The synchronous memory modules are custom-designed, 72-bit-wide plug-in cards installed in pairs to cover the full width of the 144-bit memory data bus. Synchronous memory modules that provide 32 megabytes (MB) to 256 MB per pair were designed using 16-megabit (Mb) SDRAM chips. These memory modules contain nine, eighteen, thirty-six, or seventy-two 100-MHz SDRAM chips clocked at 66 MHz, four 18-bit clocked data transceivers, address fan-out buffers, and control provided by 5-ns 28-pin PALs. To increase the maximum amount of memory in the system, a family of plug-in compatible memory modules was designed, providing up to 2 GB per pair using 64-Mb extended data out dynamic random-access memory (EDO DRAM) chips. These modules contain 72 or 144 EDO DRAM chips controlled by two custom application-specific integrated circuits (ASICs) providing data multiplexing and control, four 18-bit clocked data transceivers, and address fan-out buffers. Consequently, the AlphaServer 4100 memory architecture provides main memory capacities of 32 MB to 8 GB with a minimum latency of 120 ns to any address. This paper concentrates on the implementation of the synchronous memory modules, although the EDO memory modules are functionally compatible. The reconfigurability description later in this paper provides more details of the implementation of the EDO memory modules.

No-External-Cache Processor Module

The no-external-cache processor module is a plug-in card with a 144-bit memory interface that contains one Alpha 21164 CPU chip, eight 18-bit clocked data transceivers, four 12-bit bidirectional address latches, and control provided by 5-ns 28-pin PALs and 90-MHz 44-pin PLDs clocked at 66 MHz. The Alpha 21164 CPU chip is programmed to operate at a synchronous memory interface cycle time of 66 MHz (15 ns) to match the speed of the SDRAM chips on the memory modules. Although there are no external cache random-access memory (RAM) chips on the module, the Alpha 21164 itself contains two levels of on-chip caches: a primary 8-kilobyte (KB) data cache and a primary 8-KB instruction cache, and a second-level 96-KB three-way set-associative data and instruction cache. The no-external-cache processor module was designed to take advantage of the multiple-issue feature of the Alpha 21164 CPU. By keeping the latency to main memory low and by issuing multiple references from the Alpha 21164 CPU to main memory at the same time to increase memory bandwidth, the performance of many applications actually exceeds the performance of a processor module with a third-level external cache.[1] Numerous applications perform better, however, with a large on-board cache. For this reason, the AlphaServer 4100 offers several variants of plug-in compatible processor modules containing a 2-MB, 4-MB, or greater module-level cache. The paper "The AlphaServer 4100 Cached Processor Module Architecture and Design," which appears in this issue of the *Journal*, contains more related information.[4]

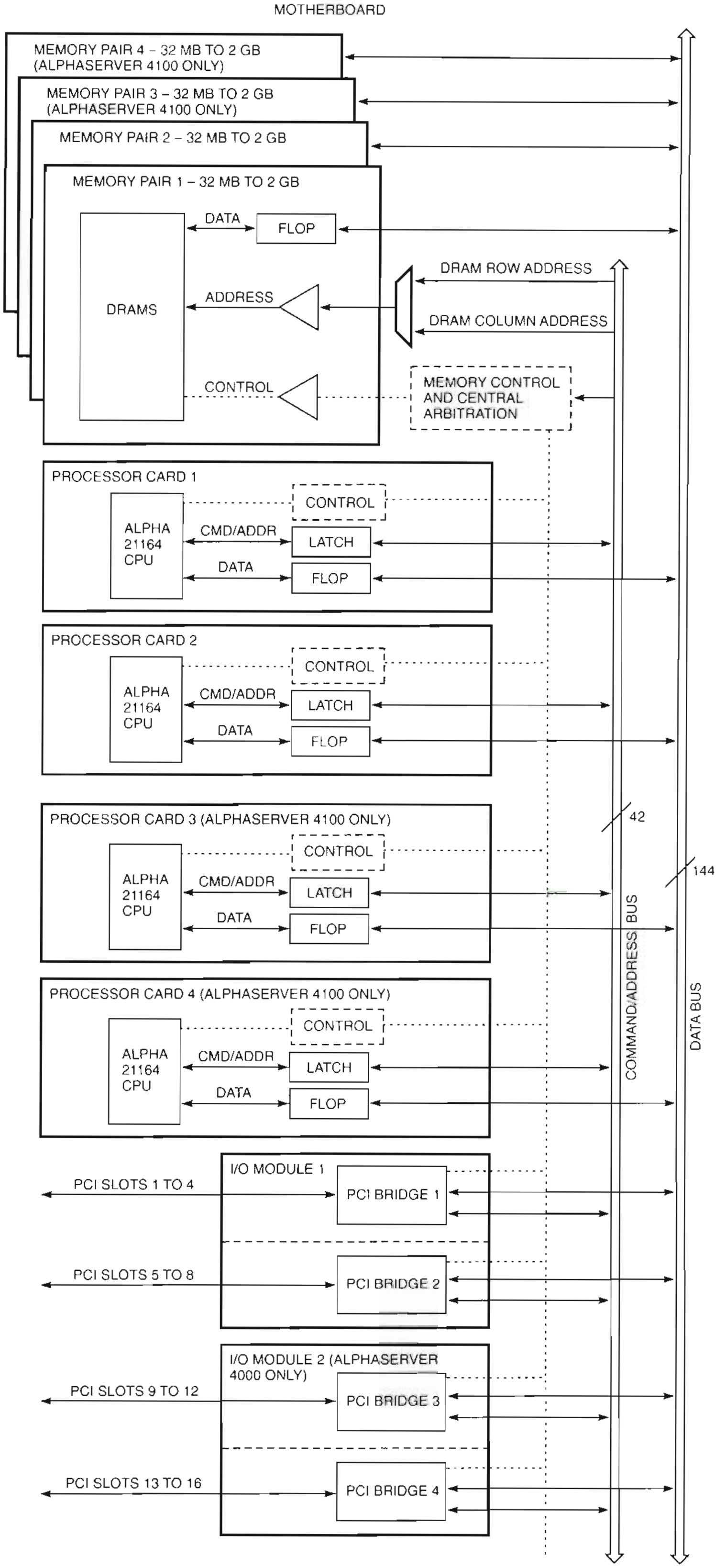
The three components of the core module set were designed concurrently to address five issues:

1. Simple design
2. Quick design time
3. Low memory latency
4. High memory bandwidth
5. Reconfigurability

Simple Design

The Alpha 21164 CPU chip is based on a reduced instruction set computing (RISC) architecture, which has a small, simple set of instructions operating as fast as possible. AlphaServer 4100 designers set the same goal of simplicity for the rest of the server system.

The AlphaServer 4100 interconnect between the CPU and main memory was optimized for the Alpha 21164 chip and the SDRAM chip. To keep the design simple, only off-the-shelf data path and address components and reprogrammable control logic devices were placed between the Alpha 21164 and SDRAM


MOTHERBOARD
MEMORY PAIR 4 – 32 MB TO 2 GB (ALPHASERVER 4100 ONLY)
MEMORY PAIR 3 – 32 MB TO 2 GB (ALPHASERVER 4100 ONLY)
MEMORY PAIR 2 – 32 MB TO 2 GB
MEMORY PAIR 1 – 32 MB TO 2 GB
DRAMS
DATA
FLOP
ADDRESS
CONTROL
DRAM ROW ADDRESS
DRAM COLUMN ADDRESS
MEMORY CONTROL AND CENTRAL ARBITRATION
PROCESSOR CARD 1
PROCESSOR CARD 2
PROCESSOR CARD 3 (ALPHASERVER 4100 ONLY)
PROCESSOR CARD 4 (ALPHASERVER 4100 ONLY)
ALPHA 21164 CPU
CMD/ADDR
LATCH
42
144
COMMAND/ADDRESS BUS
DATA BUS
I/O MODULE 1
PCI SLOTS 1 TO 4
PCI BRIDGE 1
PCI SLOTS 5 TO 8
PCI BRIDGE 2
I/O MODULE 2 (ALPHASERVER 4000 ONLY)
PCI SLOTS 9 TO 12
PCI BRIDGE 3
PCI SLOTS 13 TO 16
PCI BRIDGE 4

Note that the AlphaServer 4000 system contains the same CPU-to-memory interface as the AlphaServer 4100 but supports half the number of processors and memory modules and twice the number of PCI bridges. The AlphaServer 4000 motherboard was designed at the same time as the AlphaServer 4100 motherboard but was not produced until after the AlphaServer 4100 motherboard was available.

Figure 1
AlphaServer 4100 Memory Interconnect

chips. The designers removed excess logic and hardware features, minimized the "glue" logic between the CPU chip and main memory, reduced memory latencies as much as possible, and used custom ASICs only when necessary.

Data Path between the CPU and Memory

The external interface of the Alpha 21164 chip provides 128 bits of data plus 16 bits of error-correcting code (ECC), thus enabling single-bit error correction and multiple-bit error detection over the full width of the data path, which is shown in Figure 2. These 144 signals are connected to eight 18-bit bidirectional transceivers on the processor module. As illustrated in Figure 1, the motherboard connects up to four processor modules and up to four memory module pairs. Each memory module contains 72 bits of information; therefore, a pair of memory modules is required to provide the necessary 144 data signals. Each pair of memory modules contains eight additional 18-bit bidirectional transceivers that are connected directly to a number of SDRAM chips. The data transceiver used on the processor module and on the memory module is the 56-pin Philips ALVC162601 in a 14-millimeter (mm)-long package with 0.5-mm pitch pins. Error detection and correction using the 16 ECC bits is performed inside the Alpha 21164 chip on all read transactions. Data path errors are checked by the PCI bridge chips on all transactions, including read and write transactions between each CPU and memory, and any errors are reported to the operating system.

The data path is clocked at each stage by a copy of a single-phase clock. The clock is provided by a low-skew clock distribution system built from the 52-pin CDC586 phase-locked loop clock driver.[5] The clock cycle is controlled by an oscillator on the processor module and runs as fast as 66 MHz (15-ns minimum cycle time) while delivering less than a 2-ns worst-case skew (i.e., the difference in the rising edge of the clock) between any two components, including the Alpha 21164, SDRAMs, and any transceiver on any module.

Read transaction data is returned from the pins of the SDRAMs to the pins of the Alpha 21164 in two clock cycles (30 ns), as shown in Table 1. The no-external-cache processor has no module-level data cache, so data is clocked directly into the Alpha 21164 from the transceiver. In Table 1, read data that corresponds to transactions Rd1 and Rd2 is returned from the same set of SDRAM chips in consecutive cycles. Read data that corresponds to transaction Rd3 is returned from a different set of SDRAM chips with a one-cycle gap to allow the data path drivers from transaction Rd2 to be turned off before the data path drivers for transaction Rd3 can be turned on. This process prevents tri-state overlap. As a result, consecutive read transactions have address bus commands either four or five cycles apart. Note that the Alpha 21164 data, command, and address signals are shown for only one processor (CPU1), which issues transaction Rd1. The other transactions are issued by other processors.

Write transaction data is also transferred from the pins of the Alpha 21164 CPU to the pins of the SDRAMs in two clock cycles (see Table 2). Write data

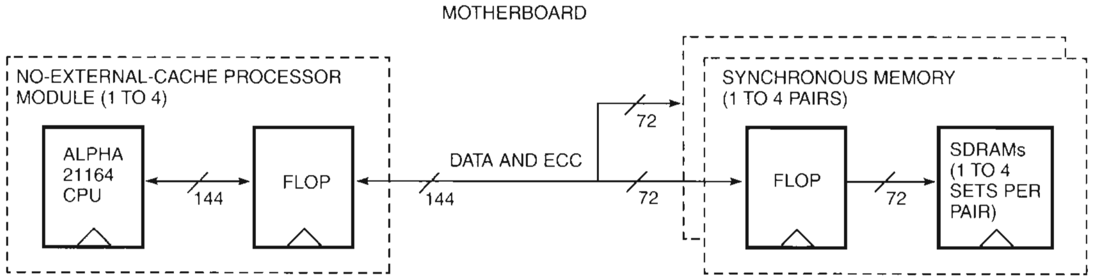

MOTHERBOARD
NO-EXTERNAL-CACHE PROCESSOR MODULE (1 TO 4)
ALPHA 21164 CPU
144
FLOP
DATA AND ECC
144
72
72
SYNCHRONOUS MEMORY (1 TO 4 PAIRS)
FLOP
72
SDRAMs (1 TO 4 SETS PER PAIR)

Figure 2
Data Path between the CPU and Memory

Table 1
CPU Read Memory Data Timing

Cycle (15 ns)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Address Bus Command		Rd1				Rd2				Rd3		▨			Rd4			
SDRAM Data						1	1	1	1	2	2	2	2	▨	3	3	3	3
Motherboard Data							1	1	1	1	2	2	2	2	▨	3	3	3
CPU1: Alpha 21164 Data								1	1	1	1					▨		
CPU1: Alpha 21164 Command	Rd1																	
CPU1: Alpha 21164 Address	Addr1																	

Table 2
CPU Write Memory Data Timing

Cycle (15 ns)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Address Bus Command	Wr1				Wr2					Wr3					Wr4			
SDRAM Data			1	1	1	1		2	2	2	2		3	3	3	3		4
Motherboard Data		1	1	1	1		2	2	2	2		3	3	3	3		4	4
Alpha 21164 Data	1	1	1	1		2	2	2	2		3	3	3	3		4	4	4

always incurs a one-cycle gap between transactions. As a result, all but the first two consecutive write transactions have address bus commands five cycles apart.

Since the AlphaServer 4100 interconnect between the CPU and main memory was optimized for the SDRAM memory chip, the transaction timing, as shown in Tables 1 and 2, was designed to provide data in the correct cycles for the SDRAMs without the need for custom ASICs to buffer the data between the motherboard and SDRAM chips. This design works well for an infinite stream of all reads or all writes because of the SDRAM pipelined interface; however, when a write transaction immediately follows a read transaction, a gap or "bubble" must be inserted in the data stream to account for the fact that read data is returned later in the transaction than write data. As a result, every write transaction that immediately follows a read transaction produces a five-cycle gap in the command pipeline. Table 3 shows the read/write transaction timing.

Address Path between the CPU and Memory

The Alpha 21164 provides 36 address signals (byte address <39:4>, i.e., bits 4 through 39), 5 command bits, and 1 bit of parity protection. These 42 signals are connected directly to four 12-bit bidirectional latched transceivers on the processor module, as illustrated in Figure 3. The motherboard latches the full address and drives first the row and then the column portion of the address to the memory modules. Each synchronous memory module buffers the row/column address and fans out a copy to each of the SDRAM chips using four 24-bit buffers. Similar to traditional dynamic random-access memory (DRAM) chips, SDRAM chips use the row address on their pins to access the page in their memory arrays and the column address that appears later on the same pins to read or write the desired location within the page. Consequently, there is no need to provide the entire 36-bit-wide address to the memory modules. All address components used for transceivers, latches, multiplexers, and drivers on the no-external-cache processor module, the motherboard, and the synchronous memory module consist of the 56-pin ALVC16260 or the ALVC162260, which is the same part with internal output resistors. Address parity is checked by the PCI bridge chips on all transactions, and any errors are reported to the operating system.

The address path uses flow-through latches for the first half of the address transfer (i.e., the row address) from the Alpha 21164 to the SDRAMs. When the address appears at the pins of the Alpha 21164, the latched transceiver on the processor module, the multiplexed row address driver on the motherboard,

Table 3
CPU Read/Write Memory Data Timing

Cycle (15 ns)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Address Bus Command	Rd1				Wr2									Wr3				
SDRAM Data					1	1	1	1				2	2	2	2		3	3
Motherboard Data						1	1	1	1		2	2	2	2		3	3	3

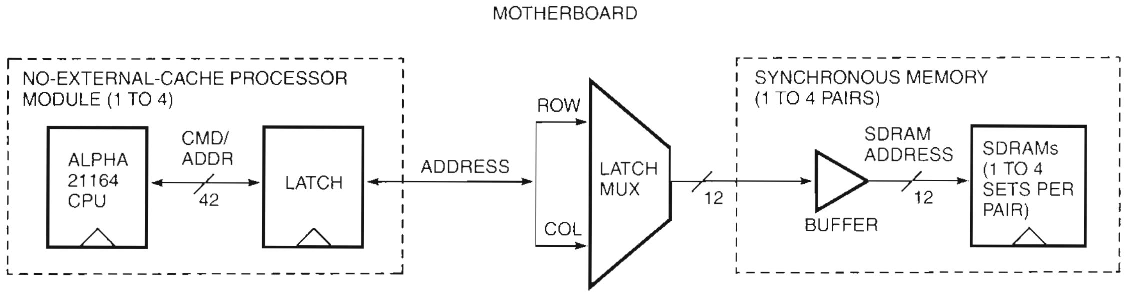

MOTHERBOARD
NO-EXTERNAL-CACHE PROCESSOR MODULE (1 TO 4)
ALPHA 21164 CPU
CMD/ ADDR
42
LATCH
ADDRESS
ROW
COL
LATCH MUX
12
SYNCHRONOUS MEMORY (1 TO 4 PAIRS)
BUFFER
SDRAM ADDRESS
12
SDRAMs (1 TO 4 SETS PER PAIR)

Figure 3
Address Path between the CPU and Memory

and the fan-out buffers on the memory modules are all open and turned on, enabling the address information to propagate directly from the Alpha 21164 pins to the SDRAM pins in two cycles. The motherboard then switches the multiplexer and drives the column address to the memory modules to complete the transaction (see Table 4). Back-to-back memory transactions are pipelined to deliver a new address to the SDRAM chips every four cycles. The full memory address is driven to the motherboard in two cycles (cycles 0–1, 4–5, 8–9), whereas additional information about the corresponding transaction (which is used only by the processor and the I/O modules) follows in a third cycle (cycles 2, 6, 10). To avoid tristate overlap, the fourth cycle is allocated as a dead cycle, which allows the address drivers of the current transaction to be turned off before the address drivers for the next transaction can be turned on (cycles 3, 7, 11). These four cycles constitute the address transfer that is repeated every four or five cycles for consecutive transactions. Note that the one-cycle gap inserted between transactions Rd3 and Rd4 for reasons indicated earlier in the read data timing description causes the row address for transaction Rd4 to appear at the pins of the SDRAMs for three cycles instead of two.

Control Path between the CPU and Memory

The Alpha 21164 provides five command bits (four Alpha 21164 CMD signals plus the Alpha 21164 Victim_Pending signal) that indicate the operation being requested by the Alpha 21164 external interface.[6] These five command bits are included in the 42 command/address (CA) signals indicated in Figure 3 and are driven directly and unmodified through the latched address transceivers on the processor module to become the motherboard command/address. Since the AlphaServer 4100 interconnect between the CPU and main memory was optimized for the Alpha 21164 CPU chip, the Alpha 21164 external CMD signals map directly into the 6-bit encoding of the memory interconnect command used on the motherboard, thus avoiding the need for custom ASICs to manipulate the commands between the CPU and motherboard.

Prudently chosen encodings of the Alpha 21164 external CMD signals resulted in only two command bits (to determine a read or a write transaction) and one address bit (to determine the memory bank) being used by a 5-ns PAL on the processor module to directly assert a Request signal to the motherboard to use the memory interconnect. Figure 4 shows the control path between the CPU and memory. If the central arbiter is ready to allow a new transaction by the processor module asserting a Request signal (i.e., if the memory interconnect is not in use), then a 5-ns PAL on the motherboard asserts the control signal Row_CS to each of the memory modules in the following cycle. At the same time, another 5-ns PAL on the motherboard decodes 7 bits of the address and drives the Sel<1:0> signal to all memory modules to indicate which of the four memory module pairs is being selected by the transaction. Each synchronous memory module uses another 5-ns PAL to immediately send the corresponding chip select (CS) signal to the requested SDRAM chips on one of the CS<3:0> signals when the Row_CS control signal is asserted if selected by the value encoded on Sel<1:0>, as shown in Figure 4.

Table 4
CPU Read Memory Address Timing

Cycle (15 ns)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Address Bus Command		Rd1				Rd2				Rd3					Rd4		
SDRAM Address	Row Addr1		Col Addr1		Row Addr2		Col Addr2		Row Addr3		Col Addr3		... Row Addr4			Col Addr4	
Motherboard Address	Mem Addr1		Info1		Mem Addr2		Info2		Mem Addr3		Info3		... Mem Addr4			Info4	
Alpha 21164 Address	Addr1		Addr2		Addr3				Addr4				Addr5				

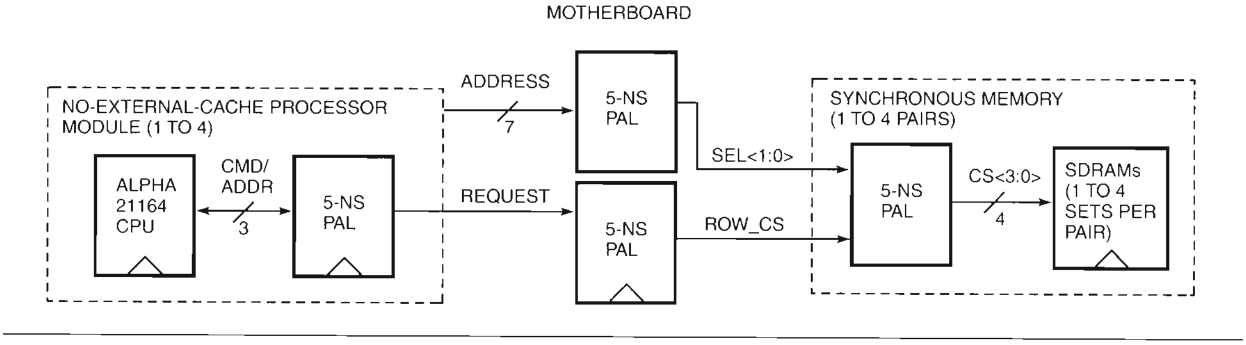

MOTHERBOARD
NO-EXTERNAL-CACHE PROCESSOR MODULE (1 TO 4)
ALPHA 21164 CPU
CMD/ ADDR
3
5-NS PAL
ADDRESS
7
REQUEST
5-NS PAL
5-NS PAL
SEL<1:0>
ROW_CS
SYNCHRONOUS MEMORY (1 TO 4 PAIRS)
5-NS PAL
CS<3:0>
4
SDRAMs (1 TO 4 SETS PER PAIR)

Figure 4
Control Path between the CPU and Memory

Table 5 shows the control signals between the processor modules, the memory modules, and the central arbiter on the motherboard for multiple processor modules issuing single read transactions. The central arbiter receives one or more Request<*n*> signals from the processor modules and asserts a unique Grant<*n*> signal to the processor module that currently owns the bus. The arbiter then drives a copy of the CA signal to every processor module along with the identical Row_CS signal to every memory module to mark cycle 1 of a new transaction. Note that the cycle counter begins at cycle 1 with each new CA/Row_CS assertion and may stall for one or more cycles when gaps appear on the memory interconnect. Two transactions may be pipelined at the same time. For simplicity of implementation in programmable logic devices, the cycle counter of each transaction is always exactly four cycles from the other.

Table 6 shows a single processor module issuing two consecutive read transactions (dual-issue) followed by a third read transaction at a later time. Normally, the node issuing the transaction on the bus deasserts the Request signal in cycle 2. If a node continues to assert the Request signal, the central arbiter continues to assert the Grant signal to that node to allow guaranteed back-to-back transactions to occur. Note that the first CA cycle occurs three cycles after the assertion of the Request signal because of the delay within the central arbiter to switch the Grant signal between processors. The third CA cycle occurs only one cycle after the node asserts the Request signal, however, because of bus parking. Bus parking is an arbitration feature that causes the central arbiter to assert the Grant signal to the last node to use the bus when the bus is idle (following cycle 7 of transaction Rd2). Consequently, if the same processor wishes to use the bus again, the assertion of CA and Row_CS signals occurs two cycles earlier than it would without the bus parking feature.

Data Transfers between Two CPU Chips (Dirty Read Data)

The Alpha 21164 CPU chips contain internal write-back caches. When a CPU writes to a block of data, the modified data is held locally in the write-back cache until it is written back to main memory at a later time. The modified (dirty) copy of the block of data must be returned in place of the unmodified (stale) copy from main memory when another CPU issues a read transaction on the memory interconnect. The memory modules return the stale data at the normal time on the memory interconnect, and the dirty data is returned by the processor module containing the modified copy in the cycles that follow. The processor module issuing the read transaction ignores the stale data from memory.

Therefore, to maintain cache coherency between the write-back caches contained in multiple Alpha

Table 5
Multiple CPU Read Memory Control Timing

Cycle Counter (15-ns cycle)			1	2	3	4	5 1	6 2	7 3	– 4	1 5	2 6	(3) (7)	3 7	4 –	5 1	6 2	7 3
Request<*n*>		1234	1234	24	24	24	24	3	3	3	3	4	4	4	4	4		
Grant<*n*>	...	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	4
CA, Row_CS (New transaction)			X				X				X					X		
Address/Command Bus		Addr/Rd1		Info1		Addr/Rd2		Info2		Addr/Rd3		Info3		...	Addr/Rd4		Info4	
SDRAM CMD (RAS,CAS,WE)		ACT 1		Read 1		ACT 2		Read 2		ACT 3		...	Read 3		ACT 4		Read 4	
SDRAM CS			X		X		X		X		X			X		X		X

Table 6
Single CPU Read Memory Control Timing

Cycle Counter (15-ns cycle)				1	2	3	4	5 1	6 2	7 3	– 4	5	6	7	–		1	2
Request<*n*>	1	1	1	1	1	1	1	1								1	1	
Grant<*n*>	2	2	1	1	1	1	1	1	1	1					1	1	1	1
CA, Row_CS (New transaction)				X				X									X	
Address/Command Bus			Addr/Rd1		Info1		Addr/Rd2		Info2							Addr/Rd3		Info3
CPU1: Alpha 21164 Data										1	1	1	1	2	2	2	2	

21164 CPU chips, each read transaction that appears on the memory interconnect causes a cache probe (snoop) to occur at all other CPU chips to determine if a modified (dirty) copy of the requested data is found in one of the internal caches of another Alpha 21164 CPU chip. If it is, then the appropriate processor module asserts the signal Dirty_Enable<*n*> for a minimum of five cycles to allow the memory module to finish driving the old data. The processor module deasserts the signal when the dirty data has been fetched from one of the internal caches and is ready to be driven onto the motherboard data bus. Table 7 shows read data corresponding to transaction Rd1 being returned from CPU2 to CPU1 five cycles later than the data from memory, which is ignored by CPU1. Note the one-cycle gap in cycles 10 and 15 to avoid tri-state overlap between the memory module and processor module data path drivers.

As discussed earlier in this section, the AlphaServer 4100 system implements memory address decoding and memory control without using custom ASICs on the motherboard, synchronous memory, or no-external-cache processor modules. Using PALs allows the address decode function and the fan-out buffering to the large number of SDRAMs to be performed at the same time, thus reducing the component count and the access time to main memory. All the necessary glue logic between the Alpha 21164 CPU and the SDRAMs, including the central arbiter on the motherboard, was implemented using 5-ns 28-pin programmable PALs or 90-MHz 44-pin ispLSI 1016 in-circuit reprogrammable PLDs produced by Lattice Semiconductor. These devices can be reprogrammed directly on the module using the parallel port of a laptop personal computer. Each no-external-cache processor module uses five PALs and four PLDs; the motherboard (arbiter and memory control) uses eight PALs and three PLDs; and each synchronous memory module uses three PALs.

As shown in Table 1, the minimum memory read latency (read data access time) is eight cycles (120 ns) from the time a new command and address arrive at the pins of the Alpha 21164 chip to the time the first data arrives back at the pins. The SDRAMs are programmed for a burst of four data cycles, so data is returned in four consecutive 15-ns cycles. Two transactions at a time are interleaved on the memory interconnect (one to each of the two memory banks), which allows data to be continuously driven in every bus cycle. This results in the maximum memory read bandwidth of 1 GB/s.

Trade-offs Made to Reduce Complexity

The Alpha 21164 external interface contains many commands required exclusively to support an external cache. By not including a module-level cache on the no-external-cache processor module, only Read, Write, and Fetch commands are generated by the Alpha 21164 external interface; the Lock, MB, SetDirty, WriteBlockLock, BCacheVictim, and ReadMissModSTC commands are not used.[6,7] This design allows the logic on the processor module that is asserting the Request signal to the central arbiter to be implemented simply in a small 28-pin PAL because only two of the Alpha 21164 CMD signals are required to encode a Read or a Write command. Similarly, allowing a maximum of two memory banks in the system, independent of the number of memory modules installed, enables the Request logic to the central arbiter to be implemented in the 28-pin PAL, since only one address bit (byte address <6>) is required to determine the memory bank.

Table 7
Dirty Read Data Timing

Cycle (15 ns)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Address Bus Command		Rd1				Rd2										Rd3		
SDRAM CS		X		X		X								X		X		X
SDRAM CMD (RAS,CAS,WE)	ACT 1		Read 1		ACT 2		. . .	. . .	. . .	. . .	. . .	. . .	Read 2		ACT 3		Read 3	
SDRAM Data						1	1	1	1							2	2	2
Motherboard Data							1	1	1	1		Dirty1	Dirty1	Dirty1	Dirty1		2	2
CPU1: Alpha 21164 Command	Rd1		Rd3				Snp2								Rd5			
CPU1: Alpha 21164 Address	Addr1		Addr3				Addr2								Addr5			
CPU1: Alpha 21164 Response										Miss2								
CPU1: Alpha 21164 Data								(1)	(1)	(1)	(1)		Dirty1	Dirty1	Dirty1	Dirty1		
CPU2: Alpha 21164 Command		Rd2	Snp1		Rd4												Snp3	
CPU2: Alpha 21164 Address		Addr2	Addr1		Addr4												Addr3	
CPU2: Alpha 21164 Response						Dirty1												
CPU2: Alpha 21164 Data											Dirty1	Dirty1	Dirty1	Dirty1				2
Dirty_Enable<*n*>							Dirty	Dirty	Dirty	Dirty	Dirty							

To decode memory addresses in 28-pin PALs, the AlphaServer 4100 system uses the concept of memory holes. The memory interconnect architecture and console code support seven different sizes of memory modules and up to four pairs of memory modules per system for a total system memory capacity of 32 MB to 8 GB. Any mix of memory module pairs is supported as long as the largest memory pair is placed in the lowest-numbered memory slot. The physical memory address range for each of the four memory slots is assigned as if all four memory module pairs are the same size. Consequently, if two additional memory pairs that are smaller than the pair in the lowest-numbered slot are installed in the upper memory slots, there will be a gap or "hole" in the physical memory space between the two smaller memory pairs (see Table 8). Rather than require each memory module to compare the full memory address to a base address and size register to determine if it should respond to the memory transaction, the 28-pin PAL driving Sel<1:0> on the motherboard (see Figure 4) uses the seven address bits Addr<32:26> and the size of the memory module in the lowest-numbered slot to encode the memory slot number of the selected memory module pair. Console code detects any memory holes at power-up and tells the operating systems that these are unusable physical memory addresses.

Another simplification that the AlphaServer 4100 system uses is to remove I/O space registers from the data path of the processor and memory modules. Because there are no custom ASICs on these modules, reading and writing control registers would have required additional data path components. Since all the error checking is performed by either the 21164 CPU chip or the PCI bridge chips and since there are no address decoding control registers required on the memory modules, there was no need for more than a few bits of control information to be accessed by software on the processor or memory modules. The I^2C bus (slow serial bus) already present in the I/O subsystem was used for transferring this small amount of information.

Furthermore, in the process of removing the I/O space data path from the motherboard and processor modules, the firmware (i.e., the console code, Alpha 21164 PAL code, and diagnostic software), which is often placed in read-only memories (ROMs) on the processor module or motherboard, was moved to the I/O subsystem. Only a small 8-KB single-bit serial ROM (SROM) was placed on each processor module that would initialize the Alpha 21164 chip on power-up and instruct the Alpha 21164 to access the rest of the firmware code from the I/O subsystem.

Quick Design Time

To provide stable CPU and memory hardware for I/O subsystem hardware debug and operating system software debug and thus allow the DIGITAL AlphaServer 4100 to be introduced on schedule in May 1996, the core module set was designed and powered on in less than six months. This primary goal of the AlphaServer 4100 project was achieved by keeping the design team small, by using only programmable logic and existing data path components, and by keeping the amount of documentation of design interfaces to a minimum.

The design team for the motherboard, no-external-cache processor module, and synchronous memory module consisted of one design engineer, one schematic/layout assistant, one signal integrity engineer, and two simulation engineers. The team also enlisted the help of members of the other AlphaServer 4100 design teams.

The architecture and actual final logic design of the core module set were developed at the same time. By using programmable logic and off-the-shelf address and data path components, the logic was written in ABL code (a language used to describe the logic functions of programmable devices) and compiled immediately into the PALs and PLDs while the architecture was being specified. If the desired functionality did not fit into the programmable devices, the architecture was modified until the logic did fit. All three modules were designed by the same engineer at the same time, so there was no need for interface specifications to be written for each module. Furthermore, modifications and enhancements could be made in parallel to each design to optimize performance and reduce complexity across all three modules.

Table 8
Memory Hole Example

Memory Slot 1	2-GB Module Pair	000000000 – 07FFFFFFF
Memory Slot 2	2-GB Module Pair	080000000 – 0FFFFFFFF
Memory Slot 3	1-GB Module Pair	100000000 – 13FFFFFFF
	Memory Hole	140000000 – 17FFFFFFF
Memory Slot 4	1-GB Module Pair	180000000 – 1BFFFFFFF
	Unused Memory	1C0000000 – 1FFFFFFFF

Because the design did not incorporate any custom ASICs, the core system was powered on as soon as the modules were built. Any last-minute logic changes required to fix problems identified by simulation could be made directly to the reprogrammable logic devices installed on the modules in the laboratory. In particular, the reset and power sequencing logic on the motherboard was not even simulated before power-on and was developed directly on actual hardware.

Since the I/O subsystem was not available when the core module set was first powered on, the software that ran on the core hardware was loaded from the serial port of a laptop personal computer and through the Alpha 21164 serial port, and then written directly into main memory. Diagnostic programs that had been developed for simulation were loaded into the memory of actual hardware and run to test a four-processor, fully loaded memory configuration. This testing enabled signal integrity fixes to be made on the hardware at full speed before the I/O subsystem was available. When the I/O subsystem was powered on, the core module set was operating bug free at full speed, allowing the AlphaServer 4100 to ship in volume six months later.

As mentioned in the section Simple Design, the central arbiter logic on the motherboard was implemented in programmable logic. Consequently, by quickly changing to the reprogrammable logic on the motherboard instead of performing a lengthy redesign of a custom ASIC, designers were able to avoid several logic design bugs that were found later in the custom ASICs of other AlphaServer 4100 processor and memory modules.

Low Memory Latency

Minimizing the access time of data being returned to the CPU on a read transaction was a major design goal for the core module set. The core module set design was optimized to deliver the Addr and CS signals to the SDRAMs in two cycles (30 ns) from the pins of the Alpha 21164 CPU and to return the data from the SDRAMs to the Alpha 21164 pins in another two cycles (30 ns). With the SDRAMs operating at a two-cycle internal row access and a two-cycle internal column access to the first data (60 ns total internal SDRAM access time), the main memory latency is 120 ns.

The low latency was accomplished in four ways:

1. By removing custom ASICs and error checking from the data path between the pins of the Alpha 21164 CPU chip and main memory
2. By combining the SDRAM row/column address multiplexer with address fan-out buffering on the motherboard
3. By simplifying the memory address decode and memory interconnect request logic
4. By using bus parking

Many multiprocessor servers share a common command/address bus by issuing a request to use the bus in one cycle, by either waiting for a grant to be returned from a central arbiter or performing local arbitration in the next cycle, and by driving the command/address on the bus in the cycle that follows. This sequence occurs for all transactions, even when the memory bus is not being used by other nodes. The AlphaServer 4100 memory interconnect implements bus parking, which allows a module to turn on its address drivers even though it is not currently using the bus. If the Alpha 21164 on that module initiates a new transaction, the command/address flows directly to memory in two less cycles than it would take to perform a costly arbitration sequence. Transaction Rd3 in Table 6 shows an example of the effects of bus parking.

High Memory Bandwidth

One of the most important features of the SDRAM chip is that a single chip can provide or consume data in every cycle for long burst lengths. The AlphaServer 4100 operates the SDRAMs with a burst length of four cycles for both reads and writes. Each SDRAM chip contains two banks determined by Addr<6>, which selects consecutive memory blocks. If accesses are made to alternating banks, then a single SDRAM can continuously drive read data in every cycle. The arbitration of the AlphaServer 4100 memory interconnect supports only two memory banks, so the smallest memory module, which consists of one set of SDRAMs, can provide the same 1-GB/s maximum read bandwidth as a fully populated memory configuration, i.e., a system configured with the minimum amount of memory can perform as well as a fully configured system.

To increase the single-processor memory bandwidth, the arbitration allows two simultaneous read transactions to be issued from a single processor module. As long as the arbitration memory bank restrictions and arbitration fairness restrictions are obeyed, it is possible to issue back-to-back read transactions to memory from a single CPU with read data being returned to the Alpha 21164 CPU in eight consecutive cycles instead of the usual four (see Tables 1 and 6). This dual-issue feature and the other low memory latency and high memory bandwidth features of the AlphaServer 4100 architecture enabled the AlphaServer 4100 system to meet the best-in-industry performance goals for McCalpin memory bandwidth.[1]

As discussed in the section Simple Design and illustrated in Figure 3, to avoid tri-state overlap, whenever read data is returned by a different set of SDRAMs (on the same memory module or on a different memory module), a dead cycle is placed between bursts of four data cycles to allow one driver to turn off before the next driver turns on. By keeping the lower-order address bits connected to all SDRAMs, i.e., by not interleaving additional banks of memory chips on low-order address bits, consecutive accesses to alternating memory banks such as large direct memory access (DMA) sequences can potentially achieve the full 1-GB/s read bandwidth of the data bus. With the dead cycle inserted, the read bandwidth of the memory interconnect is reduced by 20 percent.

The data bus connecting the processor, memory, and I/O modules was implemented as a traditional shared 3.3-volt tri-state bus with a single-phase synchronous clock at all modules. As a result, the bus becomes saturated as more processors are added and bus traffic increases. To keep the design time as short as possible, the AlphaServer 4100 designers chose not to explore the concept of a switched bus, on which more than one private transfer may occur at a time between multiple pairs of nodes. Clearly, the AlphaServer 4100 system has reached the practical upper limit of bus bandwidth using the traditional tri-state bus approach.

Reconfigurability

The AlphaServer 4100 hardware modules were designed to allow enhancements to be made in the future without having to redesign every element in the system.

Motherboard Options

The AlphaServer 4100 motherboard contains four dedicated processor slots, eight dedicated memory slots (four memory pairs), and one slot for an I/O module with two PCI bus bridges. Designed at the same time but not produced until after the AlphaServer 4100 motherboard was available, the AlphaServer 4000 motherboard contains only two processor slots, four memory slots (two memory pairs), and slots for two I/O modules allowing four PCI bus bridges. Since module hardware verification in the laboratory is a lengthy process, the AlphaServer 4000 motherboard was designed to use the same logic as the AlphaServer 4100 except for the programmable arbitration logic, which had a different algorithm because of the extra I/O module. When the signals on the AlphaServer 4000 motherboard were routed, all nets were kept shorter than the corresponding nets on the AlphaServer 4100 motherboard so that every signal did not need to be reexamined. Only those signals that were uniquely different were subject to the full signal integrity verification process.

Memory Options

The synchronous memory modules available for the AlphaServer 4100 are all based on the 16-Mb SDRAM. Using this size chip allowed designers to build synchronous memory modules that contain 9, 18, 36, and 72 SDRAMs and provide, respectively, 32 MB, 64 MB, 128 MB, and 256 MB of main memory per pair. The memory architecture supports synchronous memory modules that contain up to 1 GB of main memory per pair (up to 4 GB per system) by using the 64-Mb SDRAMs; however, when the AlphaServer 4100 system was introduced, the pricing and availability of the 64-Mb SDRAM did not allow these larger capacity synchronous memory modules to be built.

At the same time the synchronous memory modules were being designed, a family of plug-in compatible memory modules built with EDO DRAMs was designed and built. The memory architecture supports EDO memory modules containing up to 2 GB of main memory per pair (up to 8 GB per system) by using the 64-Mb EDO DRAM. When the AlphaServer 4100 system was introduced, the 64-Mb EDO DRAM was available and EDO memory modules containing 72 or 144 EDO DRAMs were built providing 1 GB and 2 GB of main memory per pair. To round out the range of memory capacities and to provide an alternative to the synchronous memory modules in case there was a cost or design problem with the new 16-Mb SDRAM chips, a family of EDO memory modules was also built using 16-Mb and 4-Mb EDO DRAMs, providing 64 MB, 256 MB, and 512 MB of main memory per pair.

Although EDO DRAMs can provide data at a higher bandwidth than standard DRAMs, a single EDO DRAM cannot return data in four consecutive 15-ns cycles like the single SDRAM used on the synchronous memory modules. Therefore, a custom ASIC was used on the EDO memory module to access 288 bits of data every 30 ns from the EDO DRAMs and multiplex the data onto the 144-bit memory interconnect every 15 ns. To imitate the two-bank feature of a single SDRAM, a second bank of EDO DRAMs is required. Consequently, the minimum number of memory chips per EDO memory module is 72 four-bit-wide EDO DRAM chips, whereas the minimum number of memory chips per synchronous memory module is only 18 four-bit-wide SDRAM chips or as few as 9 eight-bit-wide SDRAM chips.

When the AlphaServer 4100 system was introduced, the fastest EDO DRAM available that met the pricing requirements was the 60-ns version. When this chip is used on the EDO memory module, data cannot be returned to the motherboard as fast as data can be returned from the synchronous memory modules. To support the 60-ns EDO DRAMs, a one-cycle (15 ns) increase in the access time to main memory is required. Support for this extra cycle of latency was designed into the memory interconnect by placing a one-cycle gap between cycles 2 and 3 (see Table 1) of any read transaction accessing a 60-ns EDO memory module. Consequently, the read memory latency is one cycle longer and the maximum read bandwidth is 20 percent less when using EDO memory modules built with 60-ns EDO DRAMs. Note that it is possible to have a mixture of EDO memory modules and synchronous memory modules in the same system. In such a case, only the memory read transactions to the 60-ns EDO memory module would result in a loss of performance.

New versions of the EDO memory modules that contain 50-ns EDO DRAMs providing up to 8 GB of total system memory are scheduled to be introduced within a year after the introduction of the AlphaServer 4100. These modules will not require the additional cycle of latency, and as a result they will have identical performance to the synchronous memory modules.

Processor Options

The no-external-cache processor module was designed to support either a 300-MHz Alpha 21164 CPU chip with a 60-MHz (16.6-ns) synchronous memory interconnect or a 400-MHz Alpha 21164 CPU chip with a 66 MHz (15-ns) synchronous memory interconnect. As previously mentioned, the Alpha 21164 itself contains a primary 8-KB data cache, a primary 8-KB instruction cache, and a second-level 96-KB three-way set-associative data and instruction cache. The no-external-cache processor module contains no third-level cache, but by keeping the latency to main memory low and by issuing multiple references from the same Alpha 21164 to main memory at the same time to increase memory bandwidth, the performance of many applications is better than that of a processor module containing a third-level external cache.[1]

Applications that are small enough to fit in a large third-level cache perform better with an external cache, however, so the AlphaServer 4100 offers several variants of plug-in compatible processor modules containing a 2-MB, 4-MB, or greater module-level cache. In addition, cached processor modules are being designed to support Alpha 21164 CPU chips that run faster than 400 MHz while still maintaining the maximum 66-MHz synchronous memory interconnect. The architecture of the cached processor module was developed in parallel with the core module set, and several enhancements were made to the CPU and memory architecture to support the module-level cache. See the companion paper "The AlphaServer 4100 Cached Processor Module Architecture and Design" for more information.[4]

Versions of the Alpha 21164 chip that operate at 400 MHz and faster require 2-volt power, while slower versions of the Alpha 21164 require only 3.3 volts. The AlphaServer 4100 motherboard does not provide 2 volts of power to the processor module connectors; consequently, a 3.3-to-2-volt converter card is used on the higher-speed processor modules to provide this unique voltage. Each new version of processor module is plug-in compatible, and systems can be upgraded without changing the motherboard. This is true even if the frequency of the synchronous memory interconnect changes, although all processor modules in the system must be configured to operate at the same speed. The oscillators for both the high-speed internal CPU clock and the memory interconnect bus clock are located on the processor modules to allow processor upgrades to be made without modifying the motherboard.

Summary

The high-performance DIGITAL AlphaServer 4100 SMP server, which supports up to four Alpha 21164 CPUs, was designed simply and quickly using off-the-shelf components and programmable logic. When the AlphaServer 4100 system was introduced in May 1996, the memory interconnect design enabled the server to achieve a minimum memory latency of 120 nanoseconds and a maximum memory bandwidth of 1 gigabyte per second. This industry-leading performance was achieved by using off-the-shelf data path and address components and programmable logic between the CPU and the SDRAM-based main memory. The motherboard, the synchronous memory module, and the no-external-cache processor module were developed concurrently to optimize the performance of the memory architecture. These core modules were operating successfully within six months of the start of the design. The AlphaServer 4100 hardware modules were designed to allow future enhancements without redesigning the system.

Acknowledgments

Bruce Alford from Revenue Systems Engineering assisted with the schematic entry, module layout, manufacturing issues, and power-up logic design, and succeeded in smoothly transitioning the core module set to his long-term engineering support organization. Roger Dame handled signal integrity and timing analysis, while Dale Keck and Arina Finkelstein worked on simulation. Don Smelser and Darrel Donaldson provided technical guidance and moral support.

References and Notes

1. Z. Cvetanovic and D. Donaldson, "AlphaServer 4100 Performance Characterization," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 3–20.

2. S. Duncan, C. Keefer, and T. McLaughlin, "High Performance I/O Design in the AlphaServer 4100 Symmetric Multiprocessing System," *Digital Technical Journal,* vol. 8, no. 4 (1996, this issue): 61–75.

3. The AlphaServer 4000 system contains the same CPU-to-memory interface as the AlphaServer 4100 system but supports half the number of processors and memory modules and twice the number of PCI bridges. The AlphaServer 4000 motherboard was designed at the same time as the AlphaServer 4100 motherboard but was not produced until after the AlphaServer 4100 motherboard was available.

4. M. Steinman et al., "The AlphaServer 4100 Cached Processor Module Architecture and Design," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 21–37.

5. R. Dame, "The AlphaServer 4100 Low-cost Clock Distribution System," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 38–47.

6. *Alpha 21164 Microprocessor Hardware Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. EC-QAEQA-TE, September 1994).

7. The Fetch command is not implemented on the AlphaServer 4100 system, but there is no mechanism to keep it from appearing on the CMD pins of the Alpha 21164 CPU chip. The Fetch command is simply terminated without any additional action.

Biography

Glenn A. Herdeg
Glenn Herdeg has been working on the design of computer modules since joining Digital in 1983. A principal hardware engineer in the AlphaServer Platform Development group, he was the project leader, architect, logic designer, and module designer for the AlphaServer 4100 motherboard, no-external-cache processor modules, and synchronous memory modules. He also led the design of the AlphaServer 4000 motherboard. In earlier work, Glenn served as the principal ASIC and module designer for several DEC 7000, VAX 7000, and VAX 6000 projects. He holds a B.A. in physics from Colby College and an M.S. in computer systems from Rensselaer Polytechnic Institute and has two patents. Glenn is currently involved in further Alpha-based server system development.

Samuel H. Duncan
Craig D. Keefer
Thomas A. McLaughlin

High Performance I/O Design in the AlphaServer 4100 Symmetric Multiprocessing System

The DIGITAL AlphaServer 4100 symmetric multiprocessing system is based on the Alpha 64-bit RISC microprocessor and is designed for fast CPU performance, low memory latency, and high memory and I/O bandwidth. The server's I/O subsystem contributes to the achievement of these goals by implementing several innovative design techniques, primarily in the system bus-to-PCI bus bridge. A partial cache line write technique for small transactions reduces traffic on the system bus and improves memory latency. A design for deadlock-free peer-to-peer transactions across multiple 64-bit PCI bus bridges reduces system bus, PCI bus, and CPU utilization by as much as 70 percent when measured in DIGITAL AlphaServer 4100 MEMORY CHANNEL clusters. Prefetch logic and buffering supports very large bursts of data without stalls, yielding a system that can amortize overhead and deliver performance limited only by the PCI devices used in the system.

The AlphaServer 4100 is a symmetric multiprocessing system based on the Alpha 21164 64-bit RISC microprocessor. This midrange system supports one to four CPUs, one to four 64-bit-wide peer bridges to the peripheral component interconnect (PCI), and one to four logical memory slots. The goals for the AlphaServer 4100 system were fast CPU performance, low memory latency, and high memory and I/O bandwidth. One measure of success in achieving these goals is the AIM benchmark multiprocessor performance results. The AlphaServer 4100 system was audited at 3,337 peak jobs per minute, with a sustained number of 3,018 user loads, and won the AIM Hot Iron price/performance award in October 1996.[1]

The subject of this paper is the contribution of the I/O subsystem to these high-performance goals. In an in-house test, I/O performance of an AlphaServer 4100 system based on a 300-megahertz (MHz) processor shows a 10 to 19 percent improvement in I/O when compared with a previous-generation midrange Alpha system based on a 350-MHz processor. Reduction in CPU utilization is particularly beneficial for applications that use small transfers, e.g., transaction processing.

I/O Subsystem Goals

The goal for the AlphaServer 4100 I/O subsystem was to increase overall system performance by

- Reducing CPU and system bus utilization for all applications
- Delivering full I/O bandwidth, specifically, a bandwidth limited only by the PCI standard protocol, which is 266 megabytes per second (MB/s) on 64-bit option cards and 133 MB/s on 32-bit option cards
- Minimizing latency for all direct memory access (DMA) and programmed I/O (PIO) transactions

Our discussion focuses on several innovative techniques used in the design of the I/O subsystem 64-bit-wide peer host bus bridges that dramatically reduce CPU and bus utilization and deliver full PCI bandwidth:

- A partial cache line write technique for coherent DMA writes. This technique permits an I/O device to insert data that is smaller than a cache line, or block, into the cache-coherent domain without first obtaining ownership of the cache block and performing a read-modify-write operation. Partial cache line writes reduce traffic on the system bus and improve latency, particularly for messages passed in a MEMORY CHANNEL cluster.[2]
- Support for device-initiated transactions that target other devices (peers) across multiple (peer) PCI buses. Peer-to-peer transactions reduce system bus utilization, PCI bus utilization, and CPU utilization by as much as 70 percent when measured in MEMORY CHANNEL clusters. In testing, we ran a MEMORY CHANNEL application without peer-to-peer DMA, and observed 85 percent CPU utilization; running the same application with peer-to-peer DMA enabled, we observed 15 percent CPU utilization. The peer-to-peer technique is successfully implemented on the AlphaServer 4100 system without causing deadlocks.
- Large bursts of PCI-device-initiated DMA data to or from system memory. I/O subsystem support for large bursts of DMA data enables efficient PCI bus utilization because fixed bus latency can be amortized over these large transactions.
- Prefetched read data and posted write data buffering designed to keep up with the highest performance PCI devices. When used in combination with the PCI delayed-read protocol, the buffering and prefetching approach allows the system to avoid PCI bus stalls introduced by the bridge during PCI-device-initiated transactions.

The following overview of the system concentrates on the areas in which these techniques are used to enhance performance, that is, efficiency in the system bus and in the PCI bus bridge. In subsequent sections, we describe in greater detail the performance issues, other possible approaches to resolving the issues, and the techniques we developed. We conclude the paper with performance results.

AlphaServer 4100 System Overview

The AlphaServer 4100 system shown in Figure 1 includes four CPUs connected to the system bus, which comprises the data and error correction code (ECC) and the command and address lines. Also connected to the system bus are main memory and a single module with two independent peer PCI bus bridges. The single module, the PCI bridge module, provides the physical and the logical bridge between the system bus and the PCI buses. Each independent peer PCI bus bridge is constructed of a set of three application-specific integrated circuit (ASIC) chips, one control chip, and two sliced data path chips.

The two independent PCI bus bridges are the interfaces between the system bus and their respective PCI buses. A PCI bus is 64 or 32 bits wide, transferring data at a peak of 266 MB/s or 133 MB/s, respectively. In the AlphaServer 4100 system, the PCI buses are 64 bits wide.

The PCI buses connect to a PCI backplane module with a number of expansion slots and a bridge to the Extended Industry Standard Architecture (EISA) bus. In Figure 1, each PCI bus is shown to support up to four devices in option slots.

The AlphaServer 4000 series also supports a configuration in which two of the CPU cards are replaced with two additional independent peer PCI bus bridges. In the quad PCI bus configuration, there are 16 option slots available for PCI devices, at the cost of bounding the system to a maximum of two CPUs and two logical memory slots. This quad PCI bus configuration is shown in Figure 2.

Most of the techniques described in this paper are implemented in the PCI bus bridge. The partial cache line write technique, presented next, is also designed into the protocol on the system bus and into the CPU cards.

Improvements in CPU and System Bus Utilization through Use of Partial Cache Line Writes

Inefficient use of system resources can limit performance on heavily loaded systems. System designers must be attentive to potential performance bottlenecks beyond the commonly addressed CPU speed, cache loop time, and CPU memory latency. Our focus in the I/O subsystem design was to balance system performance in the face of a wide range of I/O device behaviors. We therefore implemented techniques that minimize the load on the PCI bus, the system bus, and the CPUs. The technique described in this section—partial cache line writes—reduces the load on the system bus and improves overall system performance.

Many first- and second-generation PCI controller devices were designed to operate in platforms that support 32-byte cache lines and 16-byte write buffers. It is common for an older PCI device to limit the amount of DMA data it reads or writes to match this characteristic of computers that were on the market at the time those devices were designed. Some classes of devices will, by their nature, always limit the amount of data in a burst transaction.

As do most Alpha platforms, the AlphaServer 4100 system supports a 64-byte cache line that is twice that of other common systems. When a PCI device performs a memory write of less than a complete cache line, the system must merge the data into a cache line while maintaining a consistent (coherent) view of

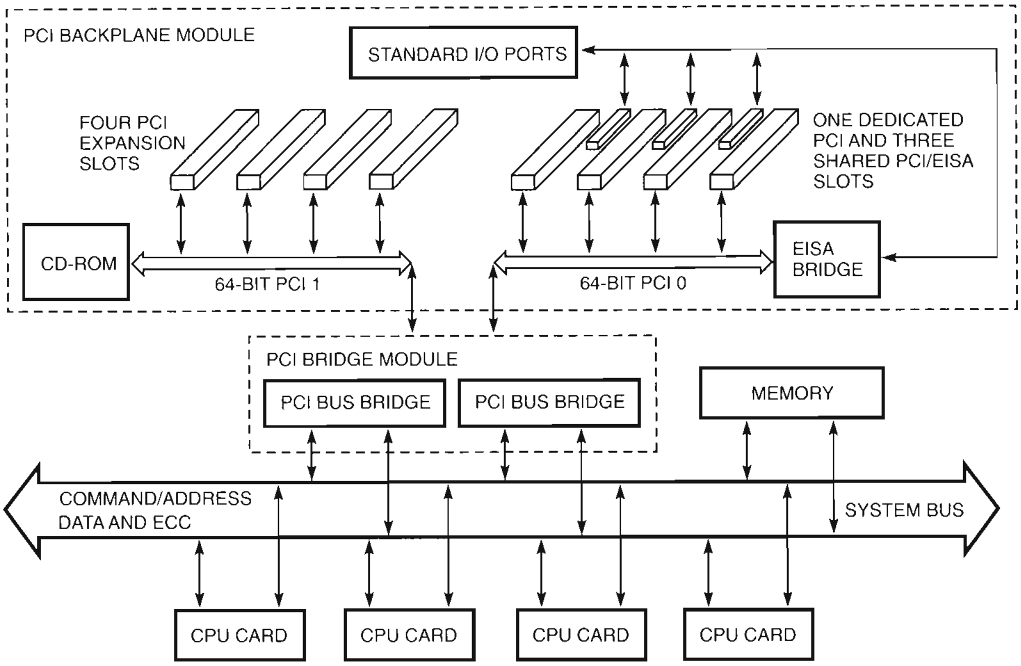


Figure 1
AlphaServer 4100 System with Four CPUs, Two 64-bit Buses

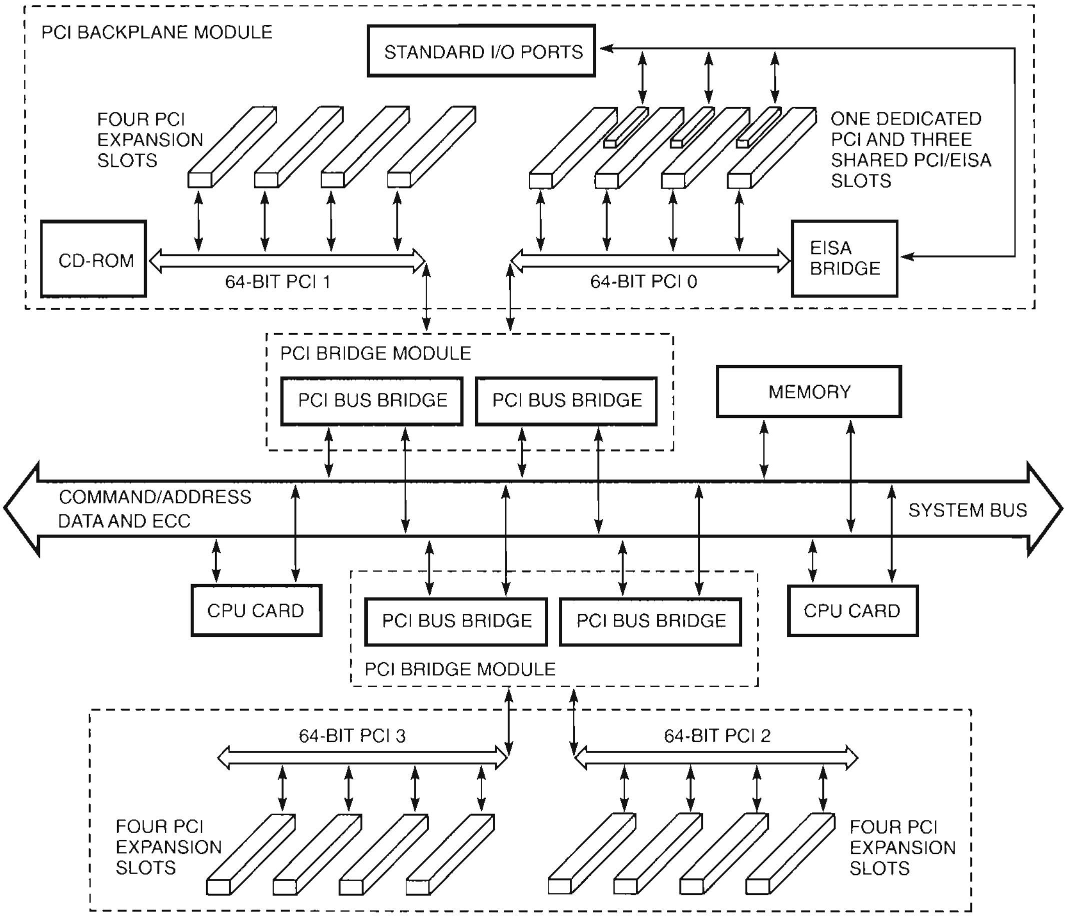


Figure 2
AlphaServer 4000 System with Two CPUs, Four 64-bit Buses

memory for all CPUs on the system bus. This merging of write data into the cache-coherent domain is typically done on the PCI bus bridge, which reads the cache line, merges the new bytes, and writes the cache line back out to memory. The read-modify-write must be performed as an atomic operation to maintain memory consistency. For the duration of the atomic read-modify-write operation, the system bus is busy. Consequently, a write of less than a cache line results in a read-modify-write that takes at least three times as many cycles on the system bus as a simple 64-byte-aligned cache line write.

For example, if we had used an earlier DIGITAL implementation of a system bus protocol on the AlphaServer 4100 system, an I/O device operation on the PCI that performed a single 16-byte-aligned memory write would have consumed system bus bandwidth that could have moved 256 bytes of data, or 16 times the amount of data. We therefore had to find a more efficient approach to writing subblocks into the cache-coherent domain.

We first examined opportunities for efficiency gains in the memory system.[3] The AlphaServer 4100 memory system interface is 16 bytes wide; a 64-byte cache line read or write takes four cycles on the system bus. The memory modules themselves can be designed to mask one or more of the writes and allow aligned blocks that are multiples of 16 bytes to be written to memory in a single system bus transaction. The problem with permitting a less than complete cache line write, i.e., less than 64 bytes, is that the write goes to main memory, but the only up-to-date/complete copy of a cache line may be in a CPU card's cache.

To permit the more efficient partial cache line write operations, we modified the system bus cache-coherency protocol. When a PCI bus bridge issues a partial cache line write on the system bus, each CPU card performs a cache lookup to see if the target of the write is dirty. In the event that the target cache block is dirty, the CPU signals the PCI bus bridge before the end of the partial write. On dirty partial cache line write transactions, the bridge simply performs a second transaction as a read-modify-write. If the target cache block is not dirty, the operation completes in a single system bus transaction.

Address traces taken during product development were simulated to determine the frequency of dirty cache blocks that are targets of DMA writes. Our simulations showed that, for the address trace we used, frequency was extremely rare. Measurement taken from several applications and benchmarks confirmed that a dirty cache block is almost never asserted with a partial cache line write.

The DMA transfer of blocks that are aligned multiples of 16 bytes but less than a cache line is four times more efficient in the 4100 system than in earlier DIGITAL implementations.

Movement of blocks of less than 64 bytes is important to application performance because there are high-performance devices that move less than 64 bytes. One example is DIGITAL's MEMORY CHANNEL adapter, which moves 32-byte blocks in a burst.[2] As MEMORY CHANNEL adapters move large numbers of blocks that are all less than a cache line of data, the I/O subsystem partial cache line write feature improves system bus utilization and eliminates the system bus as a bottleneck. Message latency across the fabric of an AlphaServer 4100 MEMORY CHANNEL cluster (version 1.0) is approximately 6 microseconds (μs). There are two DMA writes in the message: the first is a message, and the second is a flag to validate the message. These DMA writes on the target AlphaServer 4100 contribute to message latency. The improvement in latency provided by the partial cache line write feature is approximately 0.5 μs per write. With two writes per message, latency is reduced by approximately 15 percent over an AlphaServer 4100 system with the partial cache line write feature. With version 1.5 of MEMORY CHANNEL adapters, net latency will improve by about 3 μs, and the effect of partial cache line writes will approach a 30 percent improvement in message latency.

In summary, the challenge is to efficiently move a block of data of a common size (multiple of 16 bytes) that is smaller than a cache line into the cache-coherent domain. Without any further improvement, the technique reduces system bus utilization by as much as a factor of four. This technique allows subblocks to be merged without incurring the overhead of read-modify-write, yet maintains cache coherency. The only drawback to the technique is some increased complexity in the CPU cache controller to support this mode. We considered the alternative of adding a small cache to the PCI bridge. Writes into the same memory region that occur within a short period of time could merge directly into a cache. This approach adds significant complexity and increases performance only if transactions that target the same cache line are very close together in time.

Peer-to-Peer Transaction Support

System bus and PCI bus utilization can be optimized for certain applications by limiting the number of times the same block of data moves through the system. As noted in the section AlphaServer 4100 System Overview, the PCI subsystem can contain two or four independent PCI bus bridges. Our design allows external devices connected to these separate peer PCI bus bridges to share data without accessing main memory and by using a minimal amount of host bus bandwidth. In other words, external devices can effect direct access to data on a peer-to-peer basis.

In conventional systems, a data file on a disk that is requested by a client node is transferred by DMA from the disk, across the PCI and the system bus, and into main memory. Once the data is in main memory, a network device can read the data directly in memory and send it across the network to the client node. In a 4100 system, device peer-to-peer transaction circumvents the transfer to main memory. However, peer-to-peer transaction requires that the target device have certain properties. The essential property is that the device target appear to the source device as if it is main memory.

The balance of this section explains how conventional DMA reads and writes are performed on the AlphaServer 4100 system, how the infrastructure for conventional DMA can be used for peer-to-peer transactions, and how deadlock avoidance is accomplished.

Conventional DMA

We extended the features of conventional DMA on the AlphaServer 4100 system to support peer-to-peer transaction. Conventional DMA in the 4100 system works as follows.

Address space on the Alpha processor is 2^{40} or 1 terabyte; the AlphaServer 4100 system supports up to 8 gigabytes (GB) of main memory. To directly address all of memory without using memory management hardware, an address must be 33 bits. (Eight GB is equivalent to 2^{33} bytes.)

Because the amount of memory is large compared to address space available on the PCI, some sort of memory management hardware and software is needed to make memory directly addressable by PCI devices. Most PCI devices use 32-bit DMA addresses. To provide direct access for every PCI device to all of the system address space, the PCI bus bridge has memory management hardware similar to that which is used on a CPU daughter card. Each PCI bridge to the system bus has a translation look-aside buffer (TLB) that converts PCI addresses into system bus addresses. The use of a TLB permits hardware to make all of physical memory visible through a relatively small region of address space that we call a DMA window.

A DMA window can be specified as "direct mapped" or "scatter-gather mapped." A direct-mapped DMA window adds an offset to the PCI address and passes it on to the system bus. A scatter-gather mapped DMA window uses the TLB to look up the system bus address.

Figure 3 is an example of how PCI memory address space might be allocated for DMA windows and for PCI device control status registers (CSRs) and memory.

A PCI device initiates a DMA write by driving an address on the bus. In Figure 4, data from PCI devices 0 and 1 are sent to the scatter-gather DMA windows; data from PCI device 2 are sent to the direct-mapped DMA window. When an address hits in one of the DMA windows, the PCI bus bridge acknowledges the address and immediately begins to accept write data. While consuming write data in a buffer, the PCI bus bridge translates the PCI address into a system address. The bridge then arbitrates for the system bus and, using the translated address, completes the write transaction. The write transaction completes on the PCI before it completes on the system bus.

A DMA read transaction has a longer latency than a DMA write because the PCI bus bridge must first translate the PCI address into a system bus address and fetch the data before completing the transaction. That is to say, the read transaction completes on the system bus before it can complete on the PCI.

Figure 5 shows the address path through the PCI bus bridge. All DMA writes and reads are ordered

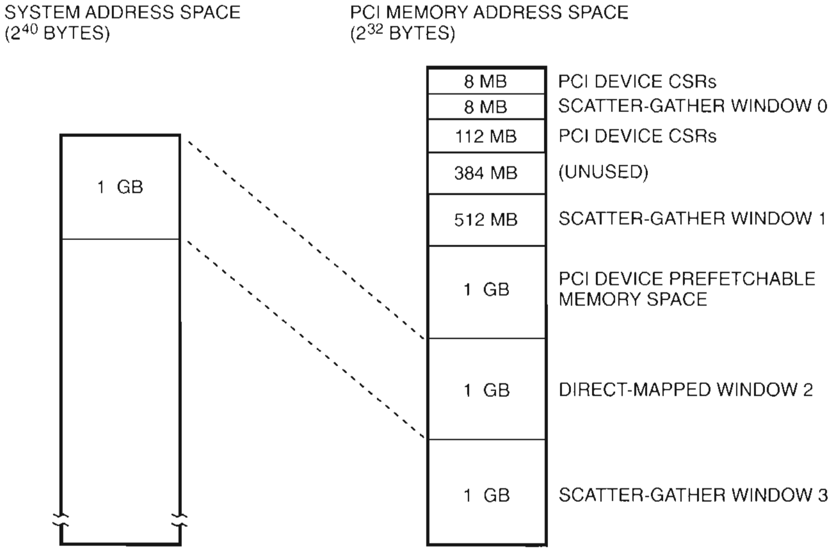


Figure 3
Example of PCI Memory Address Space Mapped to DMA Windows

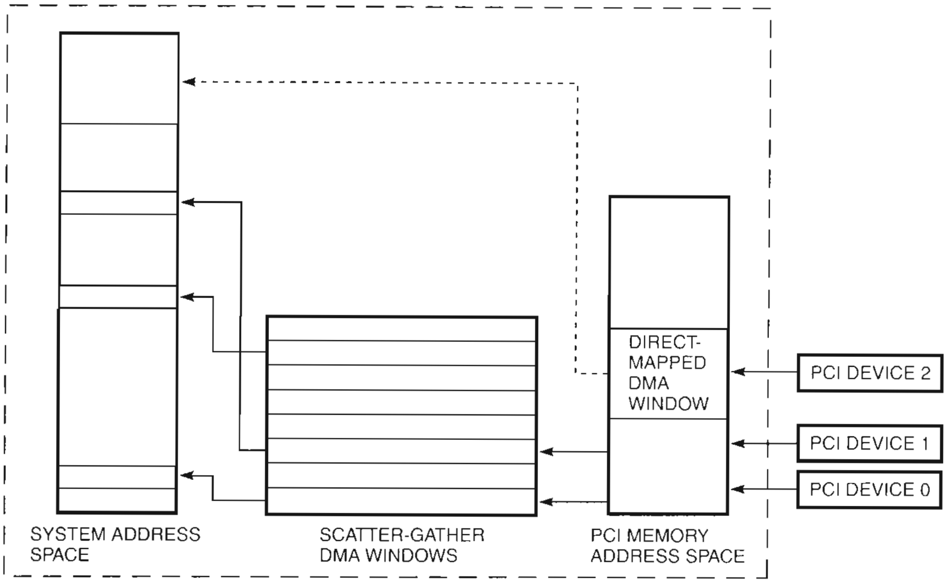


Figure 4
Example of PCI Device Reads or Writes to DMA Windows and Address Translation to System Bus Addresses

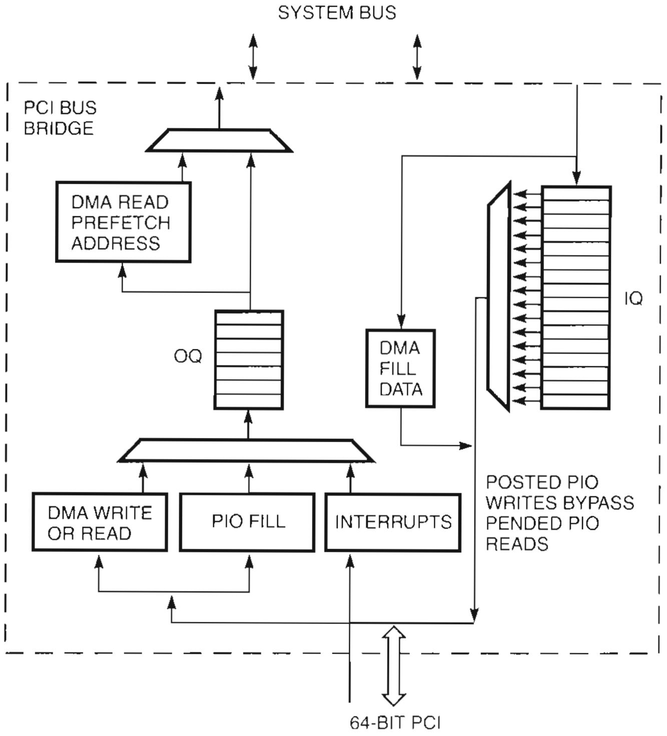


Figure 5
Diagram of Data Paths in a Single PCI Bus Bridge

through the outgoing queue (OQ) en route to the system bus. DMA read data is passed through an incoming queue (IQ) bypass by way of a DMA fill data buffer en route to the PCI.

Note that the IQ orders CPU-initiated PIO transactions. The IQ bypass is necessary for correct, deadlock-free operation of peer-to-peer transactions, which are explained in the next section.

Following is an example of how a conventional "bounce" DMA operation is used to move a file from a local storage device to a network device. The example illustrates how data is written into memory by one device where it is temporarily stored. Later the data is read by another DMA device. This operation is called a "bounce I/O" because the data "bounces" off memory and out a network port, a common operation for a network file server application.

Assume PCI device A is a storage controller and PCI device B is a network device:

1. The storage controller, PCI device A, writes the file into a buffer on the PCI bus bridge using an address that hits a DMA window.
2. The PCI bridge translates the PCI memory address into a system bus address and writes the data into memory.
3. The CPU passes the network device a PCI memory space address that corresponds to the system bus address of the data in memory.
4. The network controller, PCI device B, reads the file in main memory using a DMA window and sends the data across the network.

If both controllers are on the same PCI bus segment and if the storage controller (PCI device A) could write directly to the network controller (PCI device B), no traffic would be introduced on the system bus. Traffic on the system bus is reduced by saving one DMA write, possibly one copy operation, and one DMA read. On the PCI bus, traffic is also reduced because there is one transaction rather than two. When the target of a transaction is a device other than main memory, the transaction is called a peer-to-peer. Peer-to-peer transactions on a single-bus system are simple, bordering on trivial; but deadlock-free support on a system with multiple peer PCI buses is quite a bit more difficult.

This section has presented a high-level description of how a PCI device DMA address is translated into a system bus address and data are moved to or from main memory. In the next section, we show how the same mechanism is used to support device peer-to-peer transactions and how traffic is managed for deadlock avoidance.

A Peer-to-Peer Link Mechanism

For direct peer-to-peer transactions to work, the target device must behave as if it is main memory; that is, it must have a target address in prefetchable PCI memory space.[4] The PCI specification further states that devices are not allowed to depend on completion of a transaction as master.[5] Two devices supported by the DIGITAL UNIX operating system meet these criteria today with some restrictions; these are the MEMORY CHANNEL adapter noted earlier and the Prestoserve NVRAM, a nonvolatile memory storage device used as an accelerator for transaction processing. The PNVRAM was part of the configuration in which the AIM benchmark results cited in the introduction were achieved.

Both conventional DMA and peer-to-peer transactions work the same way from the perspective of the PCI master: The device driver provides the master device with a target address, size of the transfer, and identification of data to be moved. In the case in which a data file is to be read from a disk, the device driver software gives the PCI device that controls the disk a "handle," which is an identifier for the data file and the PCI target address to which the file should be written. To reiterate, in a conventional DMA transaction, the target address is in one of the PCI bus bridge DMA windows. The DMA window logic translates the address into a main memory address on the system bus. In a peer-to-peer transaction, the target address is translated to an address assigned to another PCI device.

Any PCI device capable of DMA can perform peer-to-peer transactions on the AlphaServer 4100 system. For example, in Figure 6, PCI device A can transfer data to or from PCI device B without using any resources or facilities in the system bus bridge. The use of a peer-to-peer transaction is controlled entirely by software: The device driver passes a target address to PCI device A, and device A uses the address as the DMA data source or destination.

If the target of the transaction is PCI device C, then system services software allocates a region in a scatter-gather map and specifies a translation that maps the scatter-gather-mapped address on PCI bus 0 to a system bus address that maps to PCI device C. This address translation is placed in the scatter-gather map. When PCI device A initiates a transaction, the address matches one of the DMA windows that has been initialized for scatter-gather. The PCI bus bridge accepts the transaction, looks up the translation in the scatter-gather map, and uses a system address that maps through PCI bus bridge 1 to hit PCI device C. The transaction on the system bus is between the two PCI bridges, with no involvement by memory or CPUs. In this transaction, the system bus is utilized, but the data is not stored in main memory. This eliminates the intermediate steps and overhead associated with conventional DMA, traditionally done by the "bounce" of the data through main memory.

The features that allow software to make a device on one PCI bus segment visible to a device on another are all implicit in the scatter-gather mapping TLB. For peer-to-peer transaction support, we extended the range of translated addresses to include memory space on peer PCI buses. This allows address space on one independent PCI bus segment to appear in a window of address space on a second independent peer PCI bus segment. On the system bus, the peer transaction hits in the address space of the other PCI bridge.

Deadlock Avoidance in Device Peer-to-Peer Transactions

The definition of deadlock, as it is solved in this design, is the state in which no progress can be made on any transaction across a bridge because the queues are filled with transactions that will never complete.

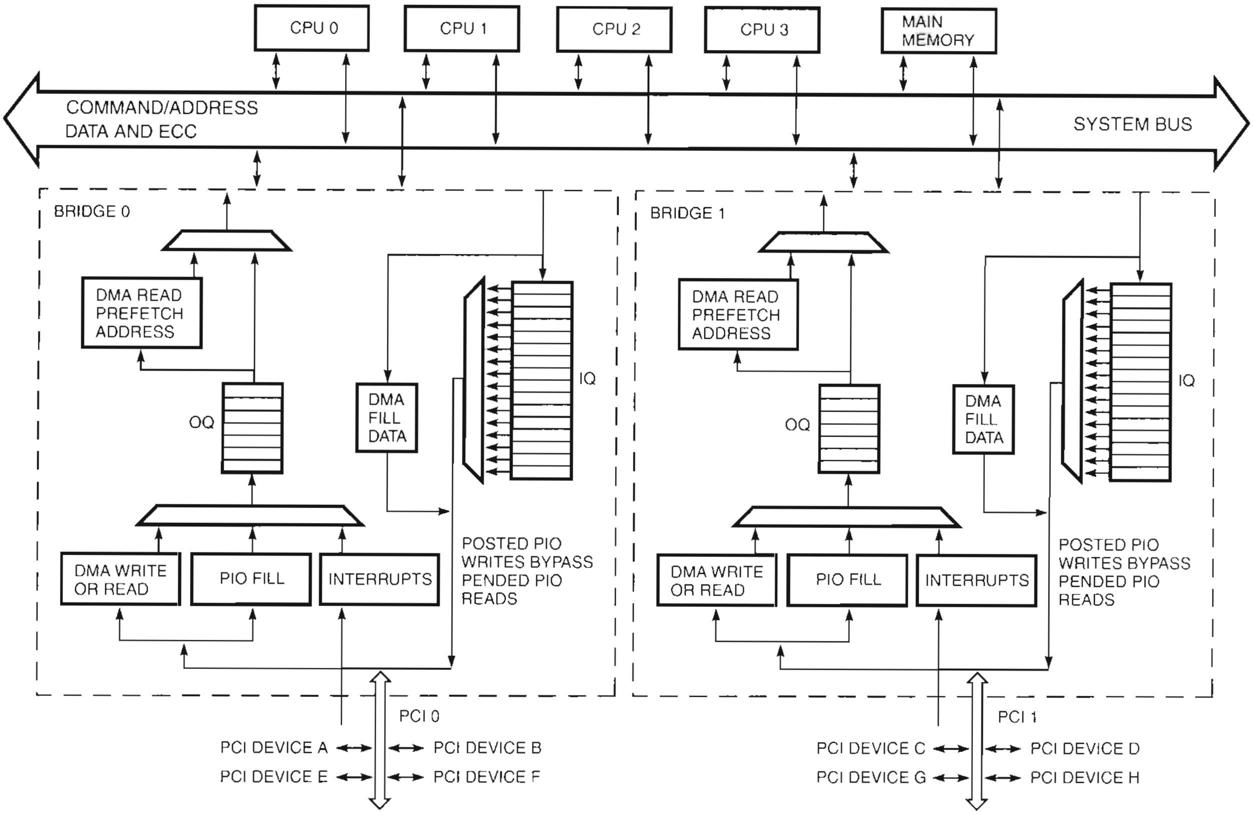


Figure 6
AlphaServer 4100 System Diagram Showing Data Paths through PCI Bus Bridges

A deadlock situation is analogous to highway gridlock in which two lines of automobiles face each other on a single-lane road; there is no room to pass and no way to back up. Rules for deadlock avoidance are analogous to the rules for directing vehicle traffic on a narrow bridge.

An example of peer-to-peer deadlock is one in which two PCI devices are dependent on the completion of a write as masters before they will accept writes as targets. When these two devices target one another, the result is deadlock; each device responds with RETRY to every write in which it is the target, and each device is unable to complete its current write transaction because it is being retried.

A device that does *not* depend on completion of a transaction as master before accepting a transaction as target may also cause deadlocks in a bridged environment. Situations can occur on a bridge in which multiple outstanding posted transactions must be kept in order. Careful design is required to avoid the potential for deadlock.

The design for deadlock-free peer-to-peer transaction support in the AlphaServer 4100 system includes the

- Implementation of PCI delayed-read transactions
- Use of bypass paths in the IQ and in read-return data

This section assumes that the reader is familiar with the PCI protocol and ordering rules.[4]

Figure 6 shows the data paths through two PCI bus bridges. Transactions pass through these bridges as follows:

- CPU software-initiated PIO reads and PIO writes are entries in the IQ.
- Device peer-to-peer transactions targeting devices on peer PCI segments also use the IQ.
- PCI-device–initiated reads and writes (DMA or peer-to-peer), interrupts, and PIO fill data are entries in the OQ.
- The multiplexer selecting entries in the IQ allows writes (PIO or peer-to-peer) to bypass delayed (pended) reads (PIO or peer-to-peer).
- The read prefetch address register permits read-return in the OQ data to bypass PCI delayed reads.

The two bypass paths around the IQ and OQ are required to avoid deadlocks that may occur during device peer-to-peer transactions. All PCI ordering rules are satisfied from the point of view of any single device in the system. The following example demonstrates deadlock avoidance in a device peer-to-peer write and a device peer-to-peer read, referencing Figure 7.

The configuration in the example is an AlphaServer 4100 system with four CPUs and two PCI bus bridges. Devices A and C are simple master-capable DMA controllers, and devices B and D are simple targets, e.g., video RAMs, network controllers, PNVRAM, or any device with prefetchable memory as defined in the PCI standard.

Example of device peer-to-peer write block completion of pended PIO read-return data:

1. PCI device A initiates a peer-to-peer burst write targeting PCI device D.
2. Write data enters the OQ on bridge 0, filling three posted write buffers.
3. The target bridge, bridge 1, writes data from bridge 0.
4. When the IQ on bridge 1 hits a threshold, it uses the system bus flow-control to hold off the next write.
5. As each 64-byte block of write data is retired out of the IQ on bridge 1, an additional 64-byte (cache line size) write of data is allowed to move from the OQ on bridge 0 to the IQ on bridge 1.
6. If the OQ on bridge 0 is full, bridge 0 will disconnect from the current PCI transaction and will retry all transactions on PCI 0 until an OQ slot becomes available.
7. PCI device C initiates a peer-to-peer burst write, targeting PCI device B; the same scenario follows as steps 1 through 6 above but in the opposite direction.
8. CPU 0 posts a read of PCI memory space on PCI device E.
9. CPU 1 posts a read of PCI memory space on PCI device G.
10. CPU 2 posts a read of PCI memory space on PCI device F.
11. CPU 3 posts a read of PCI memory space on PCI device H.
12. Deadlock:
    - Both OQs are stalled waiting for the corresponding IQ to complete an earlier posted write.
    - The design has two PIO read-return data (fill) buffers; each is full.
    - The PIO read-return data must stay behind the posted writes to satisfy PCI-specified posted write buffer flushing rules.
    - A third read is at the bottom of each IQ, and it cannot complete because there is no fill buffer available in which to put the data.

To avoid this deadlock, posted writes are allowed to bypass delayed (pended) reads in the IQ, as shown in Figure 6. In the AlphaServer 4100 deadlock-avoidance design, the IQ will always empty, which in turn allows the OQ to empty.

Note that the IQ bypass logic implemented for deadlock avoidance on the AlphaServer 4100 system may appear to violate General Rule 5 from the PCI specification, Appendix E:

> A read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.[4]

In fact, because of the characteristics of the CPUs and the flow-control mechanism on the system bus, all rules are followed as observed from any single CPU or PCI device in the system. Because reads that target a PCI address are always split into separate request and response transactions, the appropriate ordering rule for this case is PCI Specification Delayed Transaction Rule 7 in Section 3.3.3.3 of the PCI specification:

> Delayed Requests and Delayed Completions have no ordering requirements with respect to themselves or each other. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.[4]

Also note that, as shown in Figure 6, the DMA fill data buffers bypass the IQ, apparently violating General Rule 5. The purpose of General Rule 5 is to provide a mechanism in a device on one side of a bridge to ensure that all posted writes have completed. This rule is required because interrupts on PCI are side-band signals that may bypass all posted data and signal completion of a transaction before the transaction has actually completed. In the AlphaServer 4100 system, all writes to or from PCI devices are strictly ordered, and there is no side-band signal notifying a PCI device of an event. These system characteristics allow the PCI bus bridge to permit DMA fill data (in PCI lexicon, this could be a delayed-read completion, or read data in a connected transaction) to bypass posted memory writes in the IQ. This bypass is necessary to limit PCI target latency on DMA read transactions.

We have presented two IQ bypass paths in the AlphaServer 4100 design. We describe one IQ bypass as a required feature for deadlock avoidance in peer-to-peer transactions between devices on different buses. The second bypass is required for performance reasons and is discussed in the section I/O Bandwidth and Efficiency.

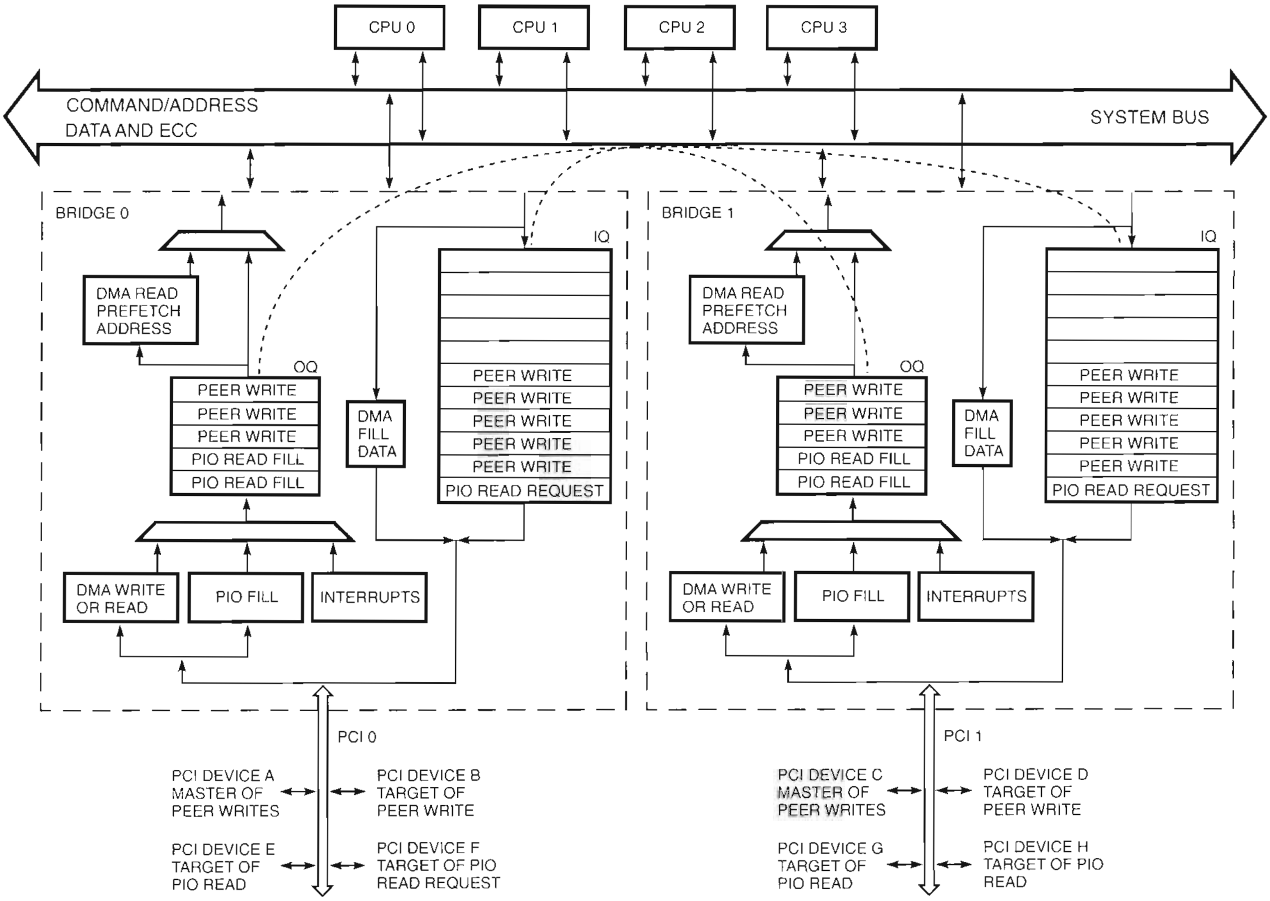

CPU 0
CPU 1
CPU 2
CPU 3
COMMAND/ADDRESS
DATA AND ECC
SYSTEM BUS
BRIDGE 0
BRIDGE 1
IQ
OQ
DMA READ PREFETCH ADDRESS
PEER WRITE
PIO READ FILL
DMA FILL DATA
PIO READ REQUEST
DMA WRITE OR READ
PIO FILL
INTERRUPTS
PCI 0
PCI 1
PCI DEVICE A MASTER OF PEER WRITES
PCI DEVICE B TARGET OF PEER WRITE
PCI DEVICE E TARGET OF PIO READ
PCI DEVICE F TARGET OF PIO READ REQUEST
PCI DEVICE C MASTER OF PEER WRITES
PCI DEVICE D TARGET OF PEER WRITE
PCI DEVICE G TARGET OF PIO READ
PCI DEVICE H TARGET OF PIO READ

Figure 7
Block Diagram Showing Deadlock Case without IQ Bypass Path

Required Characteristics for Deadlock-free Peer-to-Peer Target Devices

PCI devices must follow all PCI standard ordering rules for deadlock-free peer-to-peer transaction. The specific rule relevant to the AlphaServer 4100 design for peer-to-peer transaction support is Delayed Transaction Rule 6, which guarantees that the IQ will always empty:

> A target must accept all memory writes addressed to it while completing a request using Delayed Transaction termination.[4]

Our design includes a link mechanism using scatter-gather TLBs to create a logical connection between two PCI devices. It includes a set of rules for bypassing data that ensures deadlock-free operation when all participants in a peer-to-peer transaction follow the ordering rules in the PCI standard. The link mechanism provides a logical path for peer-to-peer transactions and the bypassing rules guarantee the IQ will always drain. The key feature, then, is a guarantee that the IQ will always drain, thus ensuring deadlock-free operation.

I/O Bandwidth and Efficiency

With overall system performance as our goal, we selected two design approaches to deliver full PCI bandwidth without bus stalls. These were support for large bursts of PCI-device-initiated DMA, and sufficient buffering and prefetching logic to keep up with the PCI and avoid introducing stalls. We open this section with a review of the bandwidth and latency issues we examined in our efforts to achieve greater bandwidth efficiency.

The bandwidth available on a platform is dependent on the efficiency of the design and on the type of transactions performed. Bandwidth is measured in millions of bytes per second (MB/s). On a 32-bit PCI, the available bandwidth is efficiency multiplied by 133 MB/s; on a 64-bit PCI, available bandwidth is efficiency multiplied by 266 MB/s. By efficiency, we mean the amount of time spent actually transferring data as compared with total transaction time.

Both parties in a transaction contribute to efficiency on the bus. The AlphaServer 4100 I/O design keeps the overhead introduced by the system to a minimum and supports large burst sizes over which the per-transaction overhead can be amortized.

Support for Large Burst Sizes

To predict the efficiency of a given design, one must break a transaction into its constituent parts. For example, when an I/O device initiates a transaction it must

- Arbitrate for the bus
- Connect to the bus (by driving the address of the transaction target)
- Transfer data (one or more bytes move in one or more bus cycles)
- Disconnect from the bus

Time actually spent in an I/O transaction is the sum of arbitration, connection, data transfer, and disconnection.

The period of time before any data is transferred is typically called latency. With small burst sizes, bandwidth is limited regardless of latency. Latency of arbitration, connection, and disconnection is fairly constant, but the amount of data moved per unit of time can increase by making the I/O bus wider. The AlphaServer 4100 PCI buses are 64 bits wide, yielding (efficiency × 266 MB/s) of available bandwidth.

As shown in Figure 8, efficiency improves as burst size increases and overhead (i.e., latency plus stall time) decreases. Overhead introduced by the AlphaServer 4100 is fairly constant. As discussed earlier, a DMA write can complete on the PCI before it completes on the system bus. As a consequence, we were able to keep overhead introduced by the platform to a minimum for DMA writes. Recognizing that efficiency improves with burst size, we used a queuing model of the system to predict how many posted write buffers were needed to sustain DMA write bursts without stalling the PCI bus. Based on a simulation model of the configurations shown in Figures 1 and 2, we determined that three 64-byte buffers were sufficient to stream DMA writes from the (266 MB/s) PCI bus to the (1 GB/s) system bus.

Later in this paper, we present measured performance of DMA write bandwidth that matches the simulation model results and, with large burst sizes, actually exceeds 95 percent efficiency.

Prefetch Logic

DMA writes complete on the PCI before they complete on the system bus, but DMA reads must wait for data fetched from memory or from a peer on another PCI. As such, latency for DMA reads is always worse than it is for writes. *PCI Local Bus Specification Revision 2.1* provides a delayed-transaction mechanism for devices with latencies that exceed the PCI initial-latency requirement.[4] The initial-latency requirement on host bus bridges is 32 PCI cycles, which is the maximum overhead that may be introduced before the first data cycle. The AlphaServer 4100 initial latency for memory DMA reads is between 18 and 20 PCI cycles. Peer-to-peer reads of devices on different bus segments are always converted to delayed-read transactions because the best-case initial latency will be longer than 32 PCI cycles.

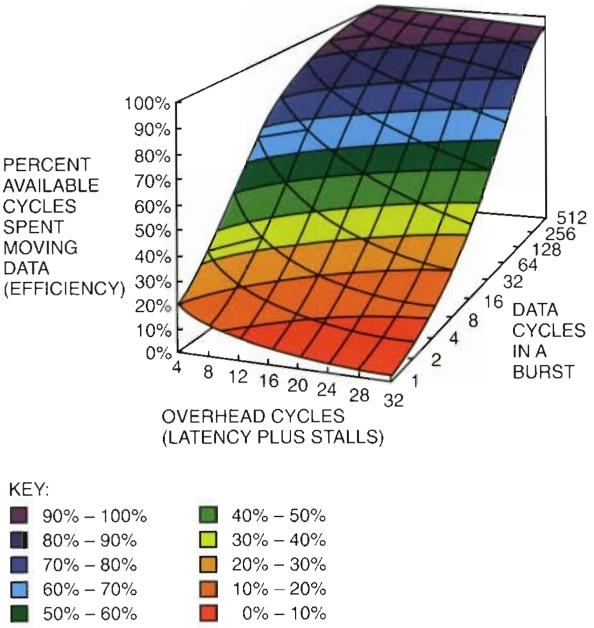


Figure 8
PCI Efficiency as a Function of Burst Size and Latency

PCI initial latency for DMA reads on the AlphaServer 4100 system is commensurate with expectations for current generation quad-processor SMP systems. To maximize efficiency, we designed prefetching logic to stream data to a 64-bit PCI device without stalls after the initial-latency penalty has been paid. To make sure the design could keep up with an uninterrupted 64-bit DMA read, we used the queuing model and analysis of the system bus protocol and decided that three cache-line-size prefetch buffers would be sufficient. The algorithm for prefetching uses the advanced PCI commands as hints to determine how far memory data prefetching should stay ahead of the PCI bus:

- Memory Read (MR): Fetch a single 64-byte cache line.
- Memory Read Line (MRL): Fetch two 64-byte cache lines.
- Memory Read Multiple (MRM): Fetch two 64-byte cache lines, and then fetch one line at a time to keep the pipeline full.

After the PCI bus bridge responds to an MRM command by fetching two 64-byte cache lines and the second line is returned, the bridge posts another read; as the oldest buffer is unloaded, new reads are posted, keeping one buffer ahead of the PCI. The third prefetch buffer is reserved for the case in which a DMA

MRM completes while there are still prefetch reads outstanding. Reservation of this buffer accomplishes two things: (1) it eliminates a time-delay bubble that would appear between consecutive DMA read transactions, and (2) it maintains a resource to fetch a scatter-gather translation in the event that the next transaction address is not in the TLB. Measured DMA bandwidth is presented later in this paper.

The point at which the design stops prefetching is on page boundaries. As the DMA window scatter-gather map is partitioned into 8-KB pages, the interface is designed to disconnect on 8-KB–aligned addresses.

The advantage of prefetching reads and absorbing posted writes on this system is that the burst size can be as large as 8 KB. With large burst size, the overhead of connecting and disconnecting from the bus is amortized and approaches a negligible penalty.

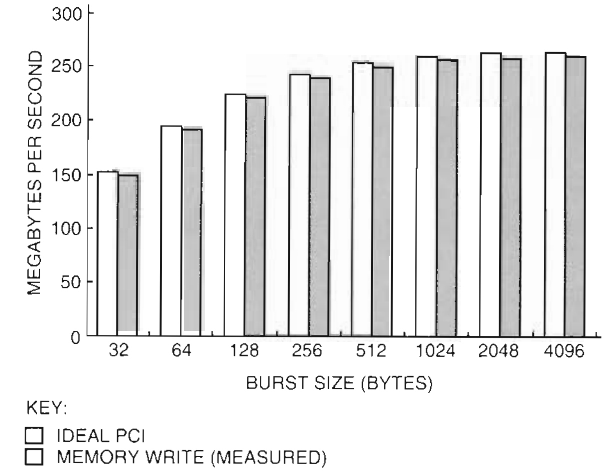


Figure 9
Comparison of Measured DMA Write Performance on an Ideal 64-bit PCI and on an AlphaServer 4100 System

DMA and PIO Performance Results

We have discussed the relationship between burst size, initial latency, and bandwidth and described several techniques we used in the AlphaServer 4100 PCI bus bridge design to meet the goals for high-bandwidth I/O. This section presents the performance delivered by the 4100 I/O subsystem design, which has been measured using a high-performance PCI transaction generator.

We collected performance data under the UNIX operating system with a reconfigurable interface card developed at DIGITAL, called PCI Pamette. It is a 64-bit PCI option with a Xilinx FPGA interface to PCI. The board was configured as a programmable PCI transaction generator. In this configuration, the board can generate burst lengths of 1 to 512 cycles. DMA either runs to a fixed count of words transferred or runs continuously (software selected). The DMA engine runs at a fixed cadence (delay between bursts) of 0 to 15 cycles in the case of a fixed count and at 0 to 63 cycles when run continuously.

The source of the DMA is a combination of a free-running counter that is clocked using the PCI clock and a PCI transaction count. The free-running counter time-stamps successive words and detects wait states and delays between transactions. The transaction count identifies retries as well as transaction boundaries.

As the target of PIO read or write, the board can accept arbitrarily large bursts of either 32 or 64 bits. It is a medium decode device and always operates with zero wait states.

DMA Write Efficiency and Performance

Figure 9 shows the close comparison between the AlphaServer 4100 system and a nearly perfect PCI design in measured DMA write bandwidth. As explained above, to sustain large bursts of DMA writes, we implemented three 64-byte posted write buffers. Simulation predicted that this number of buffers would be sufficient to sustain full bandwidth DMA writes—even when the system bus is extremely busy—because the bridges to the PCI are on a shared system bus that has roughly 1 GB/s available bandwidth. The PCI bus bridges arbitrate for the shared system bus at a priority higher than the CPUs, but the bridges are permitted to execute only a single transaction each time they win the system bus. Therefore, in the worst case, a PCI bus bridge will wait behind three other PCI bus bridges for a slot on the bus, and each bridge will have at least one quarter of the available system bus bandwidth. With 250 MB/s available but with potential delay in accessing the bus, three posted write buffers are sufficient to maintain full PCI bandwidth for memory writes.

The ideal PCI system is represented by calculated performance data for comparison purposes. It is a system that has three cycles of target latency for writes. Three cycles is the best possible for a medium decode device. The goal for DMA writes was to deliver performance limited only by the PCI device itself, and this goal was achieved. Figure 9 demonstrates that measured DMA write performance on the AlphaServer 4100 system approaches theoretical maximums. The combination of optimizations and innovations used on this platform yielded an implementation that meets the goal for DMA writes.

DMA Read Efficiency and Performance

As noted in the section Prefetch Logic, bandwidth performance of DMA reads will be lower than the performance of DMA writes on all systems because there is delay in fetching the read data from memory. For this reason, we included three cache-line–size prefetch buffers in the design.

Figure 10 compares DMA read bandwidth measured on the AlphaServer 4100 system with a PCI system that has 8 cycles of initial latency in delivering DMA read data. This figure shows that delivered bandwidth improves on the AlphaServer 4100 system as burst size increases, and that the effect of initial latency on measured performance is diminished with larger DMA bursts.

The ideal PCI system used calculated performance data for comparison, assuming a read target latency of 8 cycles; 2 cycles are for medium decode of the address, and 6 cycles are for memory latency of 180 nanoseconds (ns). This represents about the best performance that can be achieved today.

Figure 10 shows memory read and memory read line commands with burst sizes limited to what is expected from these commands. As explained elsewhere in this paper, *memory read* is used for bursts of less than a cache line; *memory read line* is used for transactions that cross one cache line boundary but are less than two cache lines; and *memory read multiple* is for transactions that cross two or more cache line boundaries.

The efficiency of *memory read* and *memory read line* does not improve with larger bursts because there is no prefetching beyond the first or second cache line respectively. This shows that large bursts and use of the appropriate PCI commands are both necessary for efficiency.

Performance of PIO Operations

PIO transactions are initiated by a CPU. AlphaServer 4100 PIO performance has been measured on a system with a single CPU, and the results are presented in Figure 11. The pended protocol for flow control on the system bus limits the number of read transactions that can be outstanding from a single CPU. A single CPU issuing reads will stall waiting for read-return data and cannot issue enough reads to approach the bandwidth limit of the bridge. Measured read performance is quite a bit lower than the theoretical limit. A system with multiple CPUs doing PIO reads—or peer-to-peer reads—will deliver PIO read bandwidth that approaches the predicted performance of the PCI bus bridge. PIO writes are posted and the CPU stalls only when the writes reach the IQ threshold. Figure 11 shows that PIO writes approach the theoretical limit of the host bus bridge.

PIO bursts are limited by the size of the I/O read and write merge buffers on the CPU. A single AlphaServer 4100 CPU is capable of bursts up to 32 bytes. PIO writes are posted; therefore, to avoid stalling the system with system bus flow control, in the maximum configuration (see Figure 2), we provide a minimum of three posted write buffers that may be filled before flow control is used. Configurations with fewer than the maximum number of CPUs can post more PIO writes before encountering flow control.

Summary

The DIGITAL AlphaServer 4100 system incorporates design innovations in the PCI bus bridge that provide a highly efficient interface to I/O devices. Partial cache line writes improve the efficiency of small writes to memory. The peer link mechanism uses TLBs to

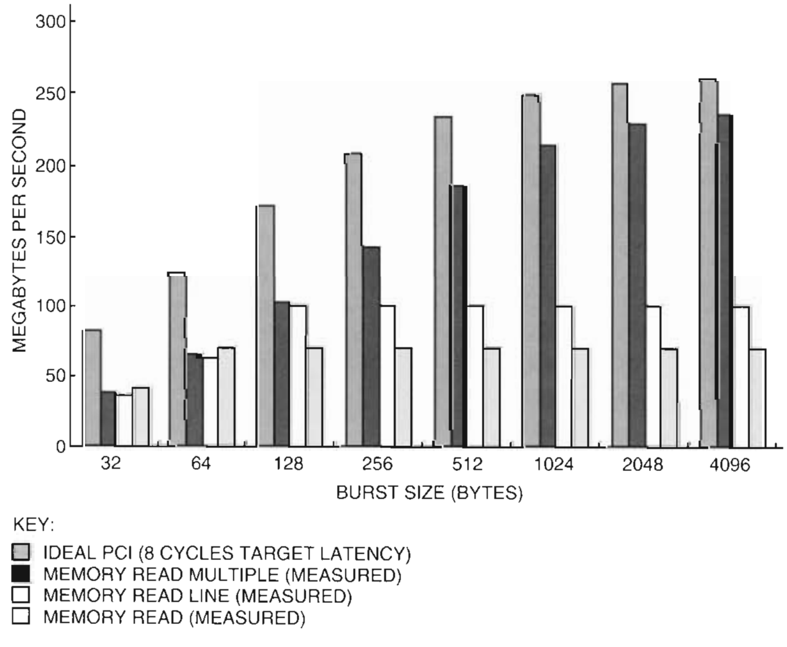


Figure 10
Comparison of DMA Read Bandwidth on the AlphaServer 4100 System and on an Ideal PCI System

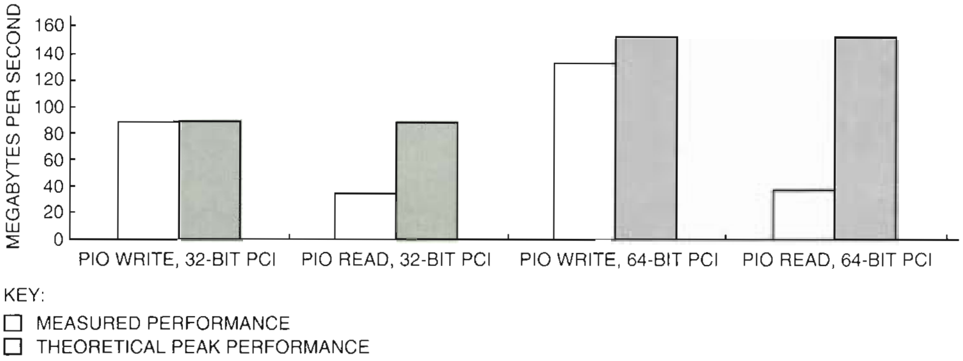


Figure 11
Comparison of AlphaServer 4100 PIO Performance with Theoretical 32-byte Burst Peak Performance

map device address space on independent peer PCI buses to permit direct peer transactions. Reordering of transactions in queues on the PCI bridge, combined with the use of PCI delayed transactions, provides a deadlock-free design for peer transactions. Buffers and prefetch logic that support very large bursts without stalls yield a system that can amortize overhead and deliver performance limited only by the PCI devices used in the system.

In summary, this system meets and exceeds the performance goals established for the I/O subsystem. Notably, I/O subsystem support for partial cache line writes and for direct peer-to-peer transactions significantly improves efficiency of operation in a MEMORY CHANNEL cluster system.

Acknowledgments

The DIGITAL AlphaServer 4100 I/O design team was responsible for the I/O subsystem implementation. The design team included Bill Bruce, Steve Coe, Dennis Hayes, Craig Keefer, Andy Koning, Tom McLaughlin, and John Lynch. The I/O design verification team was also key to delivering this product: Dick Beaven, Dmetro Kormeluk, Art Singer, and Hitesh Vyas, with CAD support from Mark Matulatis and Dick Lombard.

Several system team members contributed to inventions that improved product performance; most notable were Paul Guglielmi, Rick Hetherington, Glen Herdeg, and Maurice Steinman. We also extend thanks to our performance partners Zarka Cvetanovic and Susan Carr, who developed and ran the queuing models.

Mark Shand designed the PCI Pamette and provided the performance measurements used in this paper. Many thanks for the nights and weekends spent remotely connected to the system in our lab to gather this data.

References and Note

1. Winter UNIX Hot Iron Awards, UNIX EXPO Plus, October 9, 1996, http://www.aim.com (Menlo Park, Calif.: AIM Technology).

2. R. Gillett, "MEMORY CHANNEL Network for PCI," *IEEE Micro* (February 1996): 12–18.

3. G. Herdeg, "Design and Implementation of the AlphaServer 4100 CPU and Memory Architecture," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 48–60.

4. *PCI Local Bus Specification, Revision 2.1* (Portland, Oreg.: PCI Special Interest Group, 1995).

5. In PCI terminology, a master is any device that arbitrates for the bus and initiates transactions on the PCI (i.e., performs DMA) before accepting a transaction as target.

Biographies

Samuel H. Duncan
A consultant engineer and the architect for the AlphaServer 4100 I/O subsystem design, Sam Duncan is currently working on core logic design and architecture for the next generation of Alpha servers and workstations. Since joining DIGITAL in 1979, he has been part of Alpha and VAX system engineering teams and has represented DIGITAL on several industry standards bodies, including the PCI Special Interest Group. He also chaired the group that developed the IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory. He has been awarded one patent and has four patents filed for inventions in the AlphaServer 4100 system. Sam received a B.S.E.E. from Tufts University.

Craig D. Keefer
Craig Keefer is a principal hardware engineer whose engineering expertise is designing gate arrays. He was the gate array designer for one of the two 235K CMOS gate arrays in the AlphaServer 8200 system and the team leader for the command and address gate array in the AlphaServer 8400 I/O module. A member of the Server Product Development Group, he is now responsible for designing gate arrays for hierarchical switch hubs. Craig joined DIGITAL in 1977 and holds a B.S.E.E from the University of Lowell.

Thomas A. McLaughlin
Tom McLaughlin is a principal hardware engineer working in DIGITAL's Server Product Development Group. He is currently involved with the next generation of high-end server platforms and is focusing on logic synthesis and ASIC design processes. For the AlphaServer 4100 project, he was responsible for the logic design of the I/O subsystem, including ASIC design, logic synthesis, logic verification, and timing verification. Prior to joining the AlphaServer 4100 project, he was a member of Design and Applications Engineering within DIGITAL's External Semiconductor Technology Group. Tom joined DIGITAL in 1986 after receiving a B.T.E.E.T. from the Rochester Institute of Technology; he also holds an M.S.C.S. degree from the Worcester Polytechnic Institute.

Vipin V. Gokhale

Design of the 64-bit Option for the Oracle7 Relational Database Management System

Like most database management systems, the Oracle7 database server uses memory to cache data in disk files and improve the performance. In general, larger memory caches result in better performance. Until recently, the practical limit on the amount of memory the Oracle7 server could use was well under 3 gigabytes on most 32-bit system platforms. Digital Equipment Corporation's combination of the 64-bit Alpha system and the DIGITAL UNIX operating system differentiates itself from the rest of the computer industry by being the first standards-compliant UNIX implementation to support linear 64-bit memory addressing and 64-bit application programming interfaces, allowing high-performance applications to directly access memory in excess of 4 gigabytes. The Oracle7 database server is the first commercial database product in the industry to exploit the performance potential of the very large memory configurations provided by DIGITAL. This paper explores aspects of the design and implementation of the Oracle 64 Bit Option.

Introduction

Historically, the limiting factor for the Oracle7 relational database management system (RDBMS) performance on any given platform has been the amount of computational and I/O resources available on a single node. Although CPUs have become faster by an order of magnitude over the last several years, I/O speeds have not improved commensurately. For instance, the Alpha CPU clock speed alone has increased four times since its introduction; during the same time period, disk access times have improved by a factor of two at best. The overall throughput of database software is critically dependent on the speed of access to data.

To overcome the I/O speed limitation and to maximize performance, the standard Oracle7 database server already utilizes and is optimized for various parallelization techniques in software (e.g., intelligent caching, data prefetching, and parallel query execution) and in hardware (e.g., symmetric multiprocessing [SMP] systems, clusters, and massively parallel processing [MPP] systems). Given the disparity in latency for data access between memory (a few tens of nanoseconds) and disk (a few milliseconds), a common technique for maximizing performance is to minimize disk I/O. Our project originated as an investigation into possible additional performance improvements in the Oracle7 database server in the context of increased memory addressability and execution speed provided by the AlphaServer and DIGITAL UNIX system. Work done as part of this project subsequently became the foundation for product development of the Oracle 64 Bit Option.

Of the memory resource that the Oracle7 database uses, the largest portion is used to cache the most frequently used data blocks. With hardware and operating system support for 64-bit memory addresses, new possibilities have opened up for high-performance application software to take advantage of large memory configurations.

Two of the concepts utilized are hardly new in database development, i.e., improving database server performance by caching more data in memory and improving I/O subsystem throughput by increasing data transfer sizes. However, various conflicting factors contribute to the practical upper bounds on performance improvement. These factors include CPU architectures; memory addressability; operating system features; cost; and product requirements for portability, compatibility, and time-to-market. An additional design challenge for the Oracle 64 Bit Option project was a requirement for significant performance increases for a broad class of existing database applications that use an open, general-purpose operating system and database software.

This paper provides an overview of the Oracle 64 Bit Option, factors that influenced its design and implementation, and performance implications for some database application areas. In-depth information on Oracle7 RDBMS architecture, administrative commands, and tuning guidelines can be found in the *Oracle7 Server Documentation Set*.[1] Detailed analysis, database server, and application-tuning issues are deferred to the references cited. Overall observations and conclusions from experiments, rather than specific details and data points, are used in this paper except where such data is publicly available.

Oracle 64 Bit Option Goals

The goals for the Oracle 64 Bit Option project were as follows:

- Demonstrate a clearly identifiable performance increase for Oracle7 running on DIGITAL UNIX systems across two commonly used classes of database applications: decision support systems (DSS) and online transaction processing (OLTP).
- Ensure that 64-bit addressability and large memory configurations are the only two control variables that influence overall application performance.
- Break the 1- to 2-GB barrier on the amount of directly accessible memory that can practically be used for typical Oracle7 database cache implementations.
- Add scalability and performance features that complement, rather than replace, current Oracle7 server SMP and cluster offerings.
- Implement all of the above goals without significantly rewriting Oracle7 code or introducing application incompatibilities across any of the other platforms on which the Oracle7 system runs.

Oracle 64 Bit Option Components

Two major components make up the Oracle 64 Bit Option: big Oracle blocks (BOB) and large shared global area (LSGA). They are briefly described in this section.

The BOB component takes advantage of large memory by making individual database blocks larger than those on 32-bit platforms. A database block is a basic unit for I/O and disk space allocation in the Oracle7 RDBMS. Large block sizes mean greater density in the rows per block for the data and indexes, and typically benefit decision-support applications. Large blocks are also useful to applications that require long, contiguous rows, for example, applications that store multimedia data such as images and sound. Rows that span multiple blocks in Oracle7 require proportionately more I/O transactions to read all the pieces, resulting in performance degradation. Most platforms that run the Oracle7 system support a maximum database block size of 8 kilobytes (KB); the DIGITAL UNIX system supports block sizes of up to 32 KB.

The shared global area (SGA) is that area of memory used by Oracle7 processes to hold critical shared data structures such as process state, structured query language (SQL)–level caches, session and transaction states, and redo buffers. The bulk of the SGA in terms of size, however, is the database buffer (or block) cache. Use of the buffer cache means that costly disk I/O is avoided; therefore, the performance of the Oracle7 database server relates directly to the amount of data cached in the buffer cache. LSGA seeks to use as much memory as possible to cache database blocks. Ideally, an entire database can be cached in memory (an "in-memory" database) and avoid almost all I/O during normal operation.

A transaction whose data request is satisfied from the database buffer cache executes an order of magnitude faster than a transaction that must read its data from disk. The difference in performance is a direct consequence of the disparity in access times for main memory and disk storage. A database block found in the buffer cache is termed a "cache hit." A cache miss, in contrast, is the single largest contributor to degradation in transaction latency. Both BOB and LSGA use memory to avoid cache misses. The Oracle7 buffer cache implementation is the same as that of a typical write-back cache. As such, a cache miss, in addition to resulting in a costly disk I/O, can have secondary effects. For instance, one or more of the least recently used buffers may be evicted from the buffer cache if no free buffers are available, and additional I/O transactions may be incurred if the evicted block has been modified since the last time it was read from the disk. Oracle7 buffer cache management algorithms already implement aggressive and intelligent caching schemes and seek to avoid disk I/O. Although cache-miss penalties apply with or without the 64-bit option, "cache thrashing" that results from constrained cache sizes and large data sets can be reduced with the option to the benefit of many existing applications.

The Oracle7 buffer cache is specifically designed and optimized for Oracle's multi-versioning read-consistency transactional model. (Oracle7 buffer cache is independent of the DIGITAL UNIX unified buffer cache, or UBC.) Since Oracle7 can manage its own buffer cache more effectively than file system buffer caches, it is often recommended that the file system cache size be reduced in favor of a larger Oracle7 buffer cache when the database resides on a file system. Reducing file system cache size also minimizes redundant caching of data at the file system level. For this reason, we rejected early on the obvious design solution of using the DIGITAL UNIX file system as a large cache for taking advantage of large memory configurations—even though it had the appeal of complete transparency and no code changes to the Oracle7 system.

Background and Rationale for Design Decisions

The primary impetus for this project was to evaluate the impact on the Oracle7 database server of emerging 64-bit platforms, such as the AlphaServer system and DIGITAL UNIX operating system. Goals set forth for this project and subsequent design considerations therefore excluded any performance and functionality enhancements in the Oracle7 RDBMS that could not be attributed to the benefits offered by a typical 64-bit platform or otherwise encapsulated within platform-specific layers of the database server code or the operating system itself.

Common areas of potential benefit for a typical 64-bit platform (when compared to its 32-bit counterpart) are (a) increased direct memory addressability, and (b) the potential for configuring systems with greater than 4 GB of memory. As noted above, application performance of the Oracle7 database server depends on whether or not data are found in the database buffer cache. A 64-bit platform provides the opportunity to expand the database buffer cache in Oracle7 to sizes well beyond those of a 32-bit platform. BOB and LSGA reflect the only logical design choices available in Oracle7 to take advantage of this extended addressability and meet the project goals. Implementation of these components focused on ensuring scalability and maximizing the effectiveness of available memory resources.

BOB: Decisions Relevant to On-disk Database Size

Larger database blocks consume proportionately larger amounts of memory when the data contained in those blocks are read from the disk into the database buffer cache. Consequently, the size of the buffer cache itself must be increased if an application requires a greater number of these larger blocks to be cached. For any given size of database buffer cache, Oracle7 database administrators of 32-bit platforms have had to choose between the size of each database block and the number of database blocks that must be in the cache to minimize disk I/O, the choice depending on data access patterns of the applications. Memory available for the database buffer cache is further constrained by the fact that this resource is also shared by many other critical data structures in the SGA besides the buffer cache and the memory needed by the operating system. By eliminating the need to choose between the size of the database blocks and buffer cache, Oracle7 on a 64-bit platform can run a greater application mix without sacrificing performance.

Despite the codependency and the common goal of reducing costly disk I/O, BOB and LSGA address two different dimensions of database scalability: BOB addresses on-disk database size, and the LSGA addresses in-memory database size. Application developers and database administrators have complete flexibility to favor one over the other or to use them in combination.

In Oracle7, the on-disk data structures that locate a row of data in the database use a block-address–byte-offset tuple. The data block address (DBA) is a 32-bit quantity, which is further broken up into file number and block offset within that file. The byte offset within a block is a 16-bit quantity. Although the number of bits in the DBA used for file number and block offset are platform dependent (10 bits for the file number and 22 bits for the block offset is a common format), there exists a theoretical upper limit to the size of an Oracle7 database. With some exceptions, most 32-bit platforms support a maximum data block size of 8 KB, with 2 KB as the default. For example, using a 2-KB block size, the upper limit for the size of the database on DIGITAL UNIX is slightly under 8 terabytes (TB); whereas a 32-KB block size raises that limit to slightly under 128 TB. The ability to support buffer cache sizes well beyond those of 32-bit platforms was a critical prerequisite to enabling larger sized data blocks and consequently larger sized databases. Some 32-bit platforms are also constrained by the fact that each data file cannot exceed a size of 4 GB (especially if the data file is a file system managed object) and therefore may not be able to use all of the available block offset range in the existing DBA format. The largest database size that can be supported in such a case is even smaller. Addressing the perceived limits on the size of an Oracle7 database was an important consideration. Design alternatives that required changes to the layout or an interpretation of DBA format were rejected, at least in this project, because such changes would have introduced incompatibilities in on-disk data structures.

It should be pointed out that on current Alpha processors using an 8-KB page size, a 32-KB data block spans four memory pages, and I/O code paths in the operating system need to lock/unlock four times as many pages when performing an I/O transaction. The larger transfer size also adds to the total time taken to perform an I/O. For instance, four pages of memory that contain the 32-KB data block may not be physically contiguous, and a scatter-gather operation may be required. Although the Oracle7 database supports row-level locking for maximum concurrency in cases where unrelated transactions may be accessing different rows within a given data block, access to the data block is serialized as each individual change (a transaction-level change is broken down into multiple, smaller units of change) is applied to the data block. Larger data blocks accommodate more rows of data and consequently increase the probability of contention at the data block level if applications change (insert, update, delete) data and have a locality of reference. Experiments have shown, however, that this added cost is only marginal relative to the overall performance gains and can be offset easily by carefully tuning the application. Moreover, applications that mostly query the data rather than modify it (e.g., DSS applications) greatly benefit from larger block sizes since in this case access to the data block need not be serialized. Subtle costs such as the ones mentioned above nevertheless help explain why some applications may not necessarily see, for example, a fourfold performance increase when the change is made from an 8-KB block size to a 32-KB block size.

As with Oracle7 implementations on other platforms, database block size with the 64-bit option is determined at database creation time using db_block_size configuration parameter.[1] It cannot be changed dynamically at a later time.

LSGA: Decisions Relevant to In-memory Database Size

The focus for the LSGA effort was to identify and eliminate any constraints in Oracle7 on the sizes to which the database buffer cache could grow. DIGITAL UNIX memory allocation application programming interfaces (APIs) and process address space layout make it fairly straightforward to allocate and manage System V shared memory segments. Although the size of each shared memory segment is limited to a maximum of 2 GB (due to the requirement to comply with UNIX standards), multiple segments can be used to work around this restriction. The memory management layer in Oracle7 code therefore was the initial area of focus. Much of the Oracle7 code is written and architected to make it highly portable across a diverse range of platforms, including memory-constrained 16-bit desktop platforms. A particularly interesting aspect of 16-bit platforms with respect to memory management is that these platforms cannot support contiguous memory allocations beyond 64 KB. Users are forced to resort to a segmented memory model such that each individual segment does not exceed 64 KB in size. Although such restrictions are somewhat constraining (and perhaps irrelevant) for most 32-bit platforms—more so for 64-bit platforms—which can easily handle contiguous memory allocations well in excess of 64 KB, memory management layers in Oracle7 code are designed to be sensitive and cautious about large contiguous memory allocations and would use segmented allocations if the size of the memory allocation request exceeds a platform-dependent threshold. In particular, the size in bytes for each memory allocation request (a platform-dependent value) was assumed to be well under 4 GB, which was a correct assumption for all 32-bit platforms (and even for a 64-bit platform without LSGA). Internal data structures used 32-bit integers to represent the size of a memory allocation request.

For each buffer in the buffer cache, SGA also contains an additional data structure (buffer header) to hold all the metadata associated with that buffer. Although memory for the buffer cache itself was allocated using a special interface into the memory management layer, memory allocation for buffer headers used conventional interfaces. A different allocation scheme was needed to allocate memory for buffer headers. The buffer header is the only major data structure in Oracle7 code whose size requirements are directly dependent on the number of buffers in the buffer cache. Existing memory management interfaces and algorithms used prior to LSGA work were adequate until the number of buffers in the buffer cache exceeded approximately 700,000 (or buffer cache size of approximately 6.5 GB). Minor code changes were necessary in memory management algorithms to accommodate bigger allocation requests possible with existing high-end and future AlphaServer configurations.

The AlphaServer 8400 platform can support memory configurations ranging from 2 to 14 GB, using 2-GB memory modules. Some existing 32-bit platforms can support physical memory configurations that exceed their 4-GB addressing limit by way of segmentation, such that only 4 GB of that memory is directly accessible at any time. Programming complexity associated with such segmented memory models precluded any serious consideration in the design process to extend LSGA work to such platforms. Significantly rewriting the Oracle7 code was specifically identified as a goal not to be pursued by this project. The Alpha processor and DIGITAL UNIX system provides a flat 64-bit virtual address space model to the applications. DIGITAL UNIX extends standard UNIX APIs into a 64-bit programming environment. Our choice of the AlphaServer and DIGITAL UNIX as a development platform for this project was a fairly simple one from a time-to-market perspective because it allowed us to keep code changes to a minimum.

Efficiently managing a buffer cache of, for example, 8 or 10 GB in size was an interesting challenge. More than five million buffers can be accommodated in a 10-GB cache, with a 2-KB block size. That number of buffers is already an order of magnitude greater than what we were able to experiment with prior to the LSGA work. The Oracle7 buffer cache is organized as an associative write-back cache. The mechanism for locating a data block of interest in this cache is supported by common algorithms and data structures such as hash functions and linked lists. In many cases, traversing critical linked lists is serialized among contending threads of execution to maintain the integrity of the lists themselves and secondary data structures managed by these lists. As a result, the size of such critical lists, for example, has an impact on overall concurrency. The larger buffer count now possible in LSGA configurations had the net effect of reduced concurrency because the size of these lists is proportionately larger. LSGA provided a framework to test contributions from other unrelated projects that addressed such potential bottlenecks to concurrency, as it could realistically simulate relatively more stringent boundary conditions than before.

Scalability Issues

Engineering teams at Oracle have worked very closely with their counterparts in the DIGITAL UNIX operating system group throughout this project. The data collected in the course of the project was useful in analyzing and addressing the scalability issues in the base operating system as well as in the Oracle7 product. Examples of this work are in the base operating system granularity hint regions and in the shared page tables.[2,3]

For every page of physical and virtual memory, an operating system must maintain various data structures such as page tables, data structures to track regions of memory with certain attributes (such as System V shared memory regions, or text and data segments), or data structures that track processes which have references to these memory regions. Ancillary operating system data structures such as page tables grow in size proportionately to the size of physical memory. Changes to page table management associated with System V shared memory regions were made such that processes that mapped the shared memory regions could share page tables in addition to the data pages themselves. Prior to this change, each process mapping the shared memory region used a copy of associated page tables. A change like this reduced physical memory consumption by the operating system. For example, on an Alpha CPU supporting an 8-KB page size, it would take 8 KB in page table entries to map 8 MB of physical memory. For an SGA of 8 GB, it would take 1 MB in page table entries. It is not uncommon in the Oracle7 system for hundreds of processes to connect to the database, and therefore map the 8 GB of SGA. Without shared page tables, 100 such processes would have consumed 100 MB of physical memory by maintaining a per-process copy of page tables.

A granularity hint region is a region of physically contiguous pages of memory that share virtual and physical mappings between all the processes that map them. Such a memory layout allows DIGITAL UNIX to take advantage of the granularity hint feature supported by Alpha processors. Granularity hint bits in a page table entry allow the Alpha CPU to use a single translation look-aside buffer (TLB) entry to map a 512K physical memory space. Using one TLB entry to map larger physical memory has the potential to reduce processor stalls during TLB misses and refills. Also, because of the requirement that the granularity hint region be both virtually and physically contiguous, it is allocated at system startup time and is not subject to normal virtual memory management; for example, it is never paged in or out, and subsequently the cost of a page fault is minimal. Since pages in granularity hint regions are physically contiguous, any I/O done from this region of memory is relatively more efficient because it need not go through the scatter-gather phase.

Summary of Test Results

One of the project goals was to demonstrate clear performance benefits for two common classes of database applications, DSS and OLTP. The Transaction Processing Council (TPC) provides an industry-standard benchmark suite for both applications, that is, TPC-C for OLTP and TPC-D for DSS. An industry-standard benchmark would have been a logical choice for a workload that would demonstrate performance benefits. However, the enormous time, resources, and effort required to stage an audited TPC benchmark and the strict guidelines for any direct comparison of published benchmark results were major factors in the decision to develop a workload for this project that matched the spirit of the TPC benchmark but not necessarily the letter.

In late 1995, Oracle Corporation ran a series of performance tests for a DSS-class workload of the Oracle7 system, with and without the 64-bit option on the AlphaServer 8400 system running the DIGITAL UNIX operating system with 8 GB of physical memory. A detailed report on this test is published and available from Oracle Corporation.[4] These results, shown in Figure 1, clearly demonstrate the benefits of a large amount of physical memory in a configuration with the 64-bit option. A summary of the tests conducted is presented here along with some data points and key observations.

(Readers interested in performance characteristics of an audited industry-standard OLTP benchmark are referred to the *Digital Technical Journal*, Volume 8, Number 3. Two papers present performance characteristics of Oracle7 Parallel Server release 7.3 using 5.0 GB SGA, and a TPC-C workload on a four-node cluster.[5])

The test database consisted of five tables, representing approximately 6 GB of data. The tests included two separate configurations:

- A "standard" configuration with a 128-MB SGA with a 2-KB database block size
- A 64-bit option-enabled configuration with a 7-GB SGA and 32-KB database block size

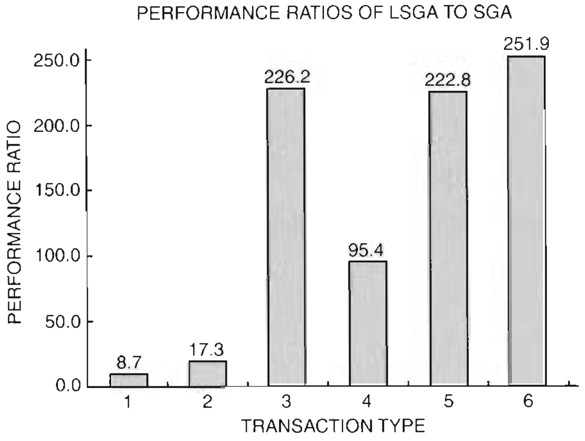


Figure 1
Performance Improvements for a DSS-class Workload, Ratios of LSGA to SGA

The evaluation included running six separate transaction types against these two configurations:

1. Full table scan against a table with 42 million rows (without the Parallel Query Option)
2. Full table scan against a table with 42 million rows (with the Parallel Query Option)
3. Set of ad hoc queries against a table with 42 million rows
4. Set of ad hoc queries involving a join against three tables with 10.5 million, 1.4 million, and 42 million rows, respectively
5. Set of ad hoc queries involving a join against four tables with 1 million, 10.5 million, 1.4 million, and 42 million rows, respectively
6. Set of ad hoc queries involving a join against five tables with 70,000, 1 million, 10.5 million, 1.4 million, and 42 million rows, respectively

Each bar in Figure 1 represents a ratio of execution time (elapsed) between a large SGA (64-bit option) and a small SGA ("standard" configuration) for each of the six transaction types. In every case, the configuration with the 64-bit option enabled consistently outperformed a "standard" configuration. In some cases, the performance increase with the option enabled was over 200 times that of the standard configuration.

The transaction mix chosen for this test represents database operations commonly used in DSS-class applications (e.g., full table scans, sort/merge, and joins). The test also uses a characteristically large data set. Transaction types 1 and 2 are identical except for the use of the Parallel Query Option. The Parallel Query Option in Oracle7 breaks up some database operations such as table scans and sorts/merge into smaller work units, and executes them concurrently. By default, these operations are executed serially, using only one thread of execution. The Parallel Query Option (independent of the 64-bit option) is a standard offering in the Oracle7 database server product since release 7.1. Use of parallel query in this test illustrates the effect of the 64-bit option enhancements on preexisting mechanisms for database performance improvement.

All other things being equal, if the only difference between a standard configuration and a 64-bit-option–enabled configuration is that the entire data set is cached in memory in the latter configuration and that typical times for main memory accesses are a few tens of nanoseconds whereas times for disk accesses are a few milliseconds, only the six to seven times performance increase in transaction type 1 would seem far below expectation. For a full table scan operation, the Oracle7 server is already optimized to use aggressive data prefetch. Before the server begins processing data in a given data block, it launches a read operation for the next block. This technique significantly reduces application-visible disk access latencies by overlapping computation and I/O. Disparity in access time for main memory and disk is still large enough to cause the computation to stall while waiting for the read-ahead I/O to finish. When data is cached in memory, this remaining stall point in the query processing is eliminated.

It is also important to note that a full table scan operation tends to access the disk sequentially. It is typical for disk access times to be better by a factor of at least two in sequential access as compared with random access. Keeping block size and disk and main memory access times the same as before in this equation, a faster Alpha CPU would yield better ratios in this test because it would finish computation proportionately faster and would wait longer for the read-ahead I/O to finish. Follow-on tests with faster CPUs supported this observation. Overlapping computation and I/O as in a table scan operation may not be possible in an index lookup operation. The sequence of operations for accessing a row of data using a B-tree index, in the best case, involves an I/O to read the index block matching the key value first, followed by another I/O to read the data block; a second I/O cannot be launched until the first finishes because the address of the data block to be read can only be determined by examining the contents of the index block read in the previous operation. Unlike table scans, these I/Os are nonsequential. Latencies of the disk I/O for an index lookup, as seen from the application perspective, are consequently greater than latencies for a table scan. Minimizing or eliminating I/Os in the index lookup, therefore, has the potential for even greater increases in speed. Index lookups are typical in OLTP workloads.

The test using transaction type 2 illustrates a cumulative effect because performance benefits for a single thread of execution extend to all the threads when the workload is parallelized.

Unlike full table scans, the sort/merge operation generates intermediate results. Depending on the size of these partial results, they may be stored in main memory if an adequate amount of memory is available; or they may be written back to temporary storage space in the database. The latter operation results in additional I/Os, proportionately more in number as inputs to the sort/merge grow in size or count. The 64-bit option makes it possible to eliminate these I/Os as well, as illustrated in transaction types 4 through 6. Performance improvements are greater as the complexity of queries increases.

Conclusion

The disparity between memory speeds and disk speeds is likely to continue for the foreseeable future. Large memory configurations represent an opportunity to overcome this disparity and to increase application performance by caching a large amount of data in memory. Even though the Oracle 64 Bit Option improves database performance—two orders of magnitude in some cases—specific application characteristics must be evaluated to determine the best means for maximizing overall performance and to balance the significant increase in hardware cost for the large amount of memory. The Oracle 64 Bit Option complements existing Oracle7 features and functionality. The exact extent of the increases in speed with the 64-bit option varies based on the type of database operation. Faster CPUs and denser memory allow for even more performance improvements than have been demonstrated. Factors of importance to new or existing applications, particularly those sensitive to response time, are an order of magnitude performance in terms of speed increases and the ability to utilize memory configurations much larger than previously possible in Oracle7 or for applications that use moderate-size data sets. With sufficient physical memory, the databases used by these response-time–sensitive applications can now be entirely cached in memory, eliminating virtually all disk I/O, which is often a major constraint to response time. In-memory (or fully cached) Oracle7 databases do not compromise transactional integrity in any way; nor do such configurations require special hardware (for example, nonvolatile random access memory [RAM]).

Because a 64-bit AlphaServer and DIGITAL UNIX operating system transparently extends existing 32-bit APIs into a 64-bit programming model, applications can take advantage of added addressability without using specialized APIs or making significant code changes. Performance levels equal to or better than previously possible with specialized hardware and software can now be achieved with industry-standard, open, general-purpose platforms.

Acknowledgments

Many people within several groups and disciplines at both Oracle and DIGITAL have contributed to the success of this project. I would like to thank the following individuals from Oracle: Walter Battistella, Saar Maoz, Jef Kennedy and David Irwin of the DIGITAL System Business Unit; and from DIGITAL: Jim Woodward, Paula Long, Darrell Dunnuck, and Dave Winchell of the DIGITAL UNIX Engineering group. Members of the Computer Systems Division's Performance Group at DIGITAL have also contributed to this project.

References

1. *Oracle7 Server Documentation Set* (Redwood Shores, Calif.: Oracle Corporation).

2. *DIGITAL UNIX V4.0 Release Notes* (Maynard, Mass.: Digital Equipment Corporation, 1996).

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

4. *Oracle 64 Bit Option Performance Report on Digital UNIX* (Redwood Shores, Calif.: Oracle Corporation, part number C10430, 1996).

5. J. Piantedosi, A. Sathaye, and D. Shakshober, "Performance Measurement of TruCluster Systems under the TPC-C Benchmark," and T. Kawaf, D. Shakshober, and D. Stanley, "Performance Analysis Using Very Large Memory on the 64-bit AlphaServer System," *Digital Technical Journal*, vol. 8, no. 3 (1996): 46–65.

Biography

Vipin V. Gokhale
Vipin Gokhale is a Consulting Software Engineer at Oracle Corporation in the DIGITAL System Business Unit where he has contributed to porting, optimization, and platform-specific features and functionality extensions to Oracle's database server on DIGITAL's operating systems and servers. He was responsible for delivering the first Oracle7 port to the DIGITAL UNIX platform. Prior to joining Oracle in 1990, Vipin was a Senior Software Engineer in India, developing telecommunications software. He received a B.Tech. in Electronics and Telecommunications from the Institute of Technology, Banaras Hindu University, India, in 1985.

VLM Capabilities of the Sybase System 11 SQL Server

T.K. Rengarajan
Maxwell Berenson
Ganesan Gopal
Bruce McCready
Sapan Panigrahi
Srikant Subramaniam
Marc B. Sugiyama

Software applications must be enhanced to take advantage of very large memory (VLM) system capabilities. The System 11 SQL Server from Sybase, Inc. has expanded the semantics of database tables for better use of memory on DIGITAL 64-bit Alpha microprocessor-based systems. Database memory management for the Sybase System 11 SQL Server includes the ability to partition the physical memory available to database buffers into multiple caches and subdivide the named caches into multiple buffer pools for various I/O sizes. The database management system can bind a database or one table in a database to any cache. A new facility on the SQL Server engine provides nonintrusive checkpoints in a VLM system.

The advent of the System 11 SQL Server from Sybase, Inc. coincided with the widespread availability and use of very large memory (VLM) technology on DIGITAL's Alpha microprocessor-based computer systems. Technological features of the System 11 SQL Server were used to achieve record results of 14,176 transactions-per-minute C (tpmC) at $198/tpmC on the DIGITAL AlphaServer 8400 server product.[1] One of these features, the Logical Memory Manager, provides the ability to fine-tune memory management. It is the first step in exploiting the semantics of database tables for better use of memory in VLM systems. To partition memory, a database administrator (DBA) creates multiple named buffer caches. The DBA then subdivides each named cache into multiple buffer pools for various I/O sizes. The DBA can bind a database or one table in a database to any cache. A new thread in the SQL Server engine, called the Housekeeper, uses idle cycles to provide free (nonintrusive) checkpoints in a large memory system.

In this paper, we briefly discuss VLM technology. Then we describe the capabilities of the Sybase System 11 SQL Server that address the issues of fast access, checkpoint, and recovery of VLM systems, namely, the Logical Memory Manager, a VLM query optimizer, the Housekeeper, and fuzzy checkpoint.

VLM Technology

The term very large memory is subjective, and its widespread meaning changes with time. By VLM, we mean systems with more than 4 gigabytes (GB) of memory. In late 1996, personal computer servers with 4 GB of memory appeared in the marketplace. At $10 per megabyte (MB), 4 GB of memory becomes affordable ($40,000) at the departmental level for corporations. We expect that most of the mid-range and high-end systems will be built with more memory in 1997. Growth in the amount of system memory is an ongoing trend. Growth beyond 4 GB, however, is a significant expansion; 32-bit systems run out of memory after 4 GB.

DIGITAL developed 64-bit computing with its Alpha line of microprocessors. Digital is now well-positioned to facilitate the transition from 32-bit to 64-bit systems. Sybase, Inc. provided one of the first relational database management systems to use VLM technology. The Sybase System 11 SQL Server provides full, native support of 64-bit Alpha microprocessors and the 64-bit DIGITAL UNIX operating system. DIGITAL UNIX is the first operating system to provide a 64-bit address space for all processes. The System 11 SQL Server uses this large address space primarily to cache large portions of the database in memory.

VLM technology is appropriate for use with applications that have stringent response time requirements. With these applications, for example, call-routing, it becomes necessary to fit the entire database in memory.[2,3] The use of VLM systems can also be beneficial when the price/performance is improved by adding more memory.[4]

Main Memory Database Systems

The widespread availability of VLM systems raises the possibility of building main memory database (MMDB) systems. Several techniques to improve the performance of MMDB systems have been discussed in the database literature. Reference 5 provides an excellent, detailed survey. We provide a brief discussion in this section.

Lock contention is low in MMDB systems since the data resides in memory. Hence, the granularity of concurrency control can be increased to minimize the overhead of lock operations. The lock manager data structures can be combined with the database objects to reduce memory usage. Specialized, stable memory hardware can be used to minimize latency of logging. Early release of transaction locks and group commit during commit processing can be used to increase concurrency and throughput. Since random access is fast in MMDBs, access methods can be developed with no key values in the index but only pointers to data rows in memory.[6] Query optimizers need to consider CPU costs, not I/O costs, when comparing various alternative plans for a query. In an MMDB, checkpointing and failure recovery are the only reasons for performing disk operations. A checkpoint process can be made "fuzzy" with low impact on transaction throughput. After a system failure, incremental recovery processing allows transaction processing to resume before the recovery is complete.[7]

As memory sizes increase with VLM systems, database sizes are also increasing. In general, we expect that databases will not fit in memory in the next decade. Therefore, for most of the databases, MMDB techniques can be exploited only for those parts of the database that do fit in memory.[5]

In addition to the capability of caching the entire database in buffers, the Sybase System 11 SQL Server provides technological advances that take advantage of VLM systems. These are the Logical Memory Manager, VLM query optimization, the Housekeeper thread, and fuzzy checkpoints. We discuss the significance of these advances in the remaining sections of this paper.

Logical Memory Manager

The Sybase SQL Server consists of several DIGITAL UNIX processes, called engines. The DBA configures the number of engines. As shown in Figure 1, each engine is permanently dedicated to one CPU of a symmetric multiprocessing (SMP) machine. The Sybase engines share virtual memory, which has been sized to include the SQL Server executable. The virtual memory is locked to physical memory. As a result, there is never any operating system paging for the Sybase memory. This shared memory region also uses large operating system pages to minimize translation lookaside buffer (TLB) entries for the CPU.[8] The shared memory holds the database buffers, stored procedure cache, sort buffers, and other dynamic memory. This memory is managed exclusively by the SQL Server. One SQL Server usually processes transactions on multiple databases. Each database has its own log. Transactions can span databases using two-phase commit. For further details on the SQL Server architecture, please see reference 9.

The Logical Memory Manager (LMM) provides the ability for a DBA to partition the physical memory available to database buffers. The DBA can partition the memory used for the database buffers into multiple caches. The DBA needs to specify a size and a name for each cache. After all named caches have been defined, the system defines the remaining memory as the default cache. Once the DBA partitions the memory, it can then bind database entities to a particular cache. The database entity is one of the following: an

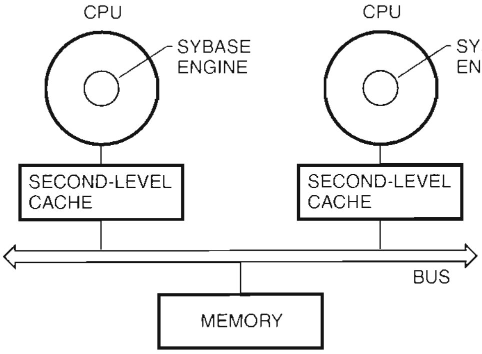


Figure 1
SQL Server on an SMP System

entire database, one table in a database, or one index on one table in a database. There is no limit to the number of such entities that can be bound to a cache. This cache binding directs the SQL Server to use only that cache for the pages that belong to the entity. Thus, the DBA can bind a small database to one cache. In a VLM system, if the cache were sized to be larger than the database, an MMDB would result.

Figure 2 shows the table bindings to named caches with the LMM. The procedure cache is used only for keeping compiled stored procedures in memory and is shown for completeness. The item cache is a small cache of 1 GB in size and is used for storing a small read-only table (item) in memory. The default cache holds the remaining tables. Figure 2 shows one table bound to the item cache and the other tables bound to the default cache. By being able to partition the use of memory for the item table separately, the SQL Server is now able to take advantage of MMDB techniques for only the item cache.

Each named cache can be larger than 4 GB. The size is limited only by the amount of memory present in the system. Although we do not expect such a need, it is also possible to have hundreds of named caches; 64-bit pointers are used throughout the SQL Server to address large memory spaces.

The LMM enables the DBA to fine-tune the use of memory. The LMM also allows for the introduction of specific MMDB algorithms in the SQL Server based on the semantics of database entities and the size of named caches. For example, in the future, it becomes possible for a DBA to express the fact that most of one table fits in one named cache in memory, so that SQL Server can use clock buffer replacement.

VLM Query Optimization

The SQL Server query optimizer computes the cost of query plans in terms of CPU as well as I/O. Both costs are reduced to an estimate of time. Since the number of I/O operations depends on the amount of memory available, the optimizer uses the size of the cache in the cost calculations. With LMM, the optimizer uses the size of the named cache to which a certain table is bound. Therefore, in the case of a database that completely fits in memory in a VLM system, the optimizer choices are made purely on the basis of CPU cost. In particular, the I/O cost is zero, when a table or an index fits in a named cache.

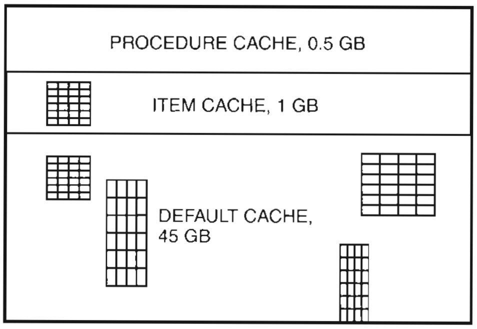


Figure 2
Table Bindings to Named Caches with Logical Memory Manager

The Sybase System 11 SQL Server introduced the notion of the fetch-and-discard buffer replacement policy. This strategy indicates that a buffer read from disk will not be used in the near future and hence is a good candidate to be replaced from the cache. The buffer management algorithms leave this buffer close to the least-recently-used end of the buffer chain. In the simplest example, a sequential scan of a table uses this strategy. With VLM, this strategy is turned off if the table can be completely cached in memory. The fetch-and-discard strategy can also be tuned by application developers and DBAs if necessary.

Housekeeper

One of the motivations for developing VLM was the extremely quick response time requirements for transactions. These environments also require high availability of systems. A key component in achieving high availability is the recovery time. Database systems write dirty pages to disk primarily for page replacement. The checkpoint procedure writes dirty pages to disk to minimize recovery time.

The Sybase System 11 SQL Server introduces a new thread called the Housekeeper that runs only at idle time for the system and does useful work. This thread is the basis for lazy processing in the SQL Server for now and the future. In System 11, the Housekeeper writes dirty pages to disk. At first, it writes pages to disk from the least-recently-used buffer. In this sense, it helps page replacement. In addition to ensuring that there are enough clean buffers, the Housekeeper also attempts to minimize both the checkpoint time and the recovery time. If the system becomes idle at any time during transaction processing, even for a few milliseconds, the Housekeeper keeps the disks (as many as possible) busy by writing dirty pages to disk. It also makes sure that none of the disks is overloaded, thus preventing an undue delay if transaction processing resumes. In the best case, the Housekeeper automatically generates a free checkpoint for the system, thereby reducing the performance impact of the checkpoint during transaction processing. In steady state, the Housekeeper continuously writes dirty pages to disk, while minimizing the number of extra writes incurred by premature writes to disk.[10]

Checkpoint and Recovery

As the size of memory increases, the following two factors increase as well: (1) the number of writes to disk during the checkpoint and (2) the number of disk I/Os to be done during recovery. The Sybase System 11 SQL Server allows the DBA to tune the amount of buffers that will be kept clean all the time. This is called the wash region. In essence, the wash region represents the amount of memory that is always clean (or strictly, in the process of being written to disk). For example, if the total amount of memory for database buffers is 6 GB and the wash region is 2 GB, then at any time, only 4 GB of memory can be in an updated state (dirty). The ability to tune the wash region reduces the load on the checkpoint procedure, as well as recovery.

The Sybase System 11 SQL Server has implemented a fuzzy checkpoint that allows transactions to proceed even during a checkpoint operation. Transactions are stalled only when they try to update a database page that is being written to disk by the checkpoint. Even in that case, the stall lasts only for the time it takes the disk write to complete. In addition, in the SQL Server, the checkpoint process can keep multiple disks busy by issuing a large number of asynchronous writes one after another. During the time of the checkpoint, the Housekeeper often becomes active due to extra idle time created by the checkpoint. The Housekeeper is self-pacing; it does not swamp the storage system with writes.

Commit Processing

The SQL Server uses the group commit algorithm to improve throughput.[8,11] The group commit algorithm collects the log records of multiple transactions and writes them to the disk in one I/O. This allows higher transaction throughput due to the amortization of disk I/O costs, as well as committing more and more transactions in each disk write to the log file. The SQL Server does not use a timer, however, to improve the grouping of transactions. Instead, the duration of the previous log I/O is used to collect transactions to be committed in the next batch. The size of the batch is determined by the number of transactions that reach commit processing during one rotation of the log disk. This self-tuning algorithm adapts itself to various speeds of disks. For the same transaction processing system, the grouping occurs more often with slower disks than with faster disks.

Consider, for example, a system performing 1,000 transactions per second. Let us assume the log disk is rated at 7,200 rpm. Each rotation of the disk takes 8 milliseconds. Within this duration, we expect (on the average) 8 transactions to complete, assuming uniform arrival rates at commit point. This indicates a natural grouping of 8 transactions per log write. For the same system, if the log disk is rated at 3,600 rpm, the same calculation yields 16 transactions per log write.

The group commit algorithm used by the SQL Server also takes advantage of disk arrays by initiating multiple asynchronous writes to different members of the disk array. The SQL Server is also able to issue up to 16 kilobytes in one write to a single disk. Together, the group commit algorithm, large writes, and the ability to drive multiple disks in a disk array eliminate the log bottleneck for high-throughput systems.

Future Work

When a VLM system fails, the large number of database buffers in memory that are dirty need to be recovered. Therefore, database recovery time grows with the size of memory in the VLM system, at least for all database systems that use log-based recovery. In addition, since there are a large number of dirty buffers in memory, the performance impact of checkpoint on transactions also increases with memory size. To minimize the recovery time, one may increase the checkpoint frequency. The checkpoints have a higher impact, however, and need to be done infrequently. These conflicting requirements need to be addressed for VLM systems.

When a database fits in memory, the buffer replacement algorithm can be eliminated. For example, for a single table that fits in one named cache, this optimization can be done with the LMM. In addition, if a table is read-only, it is possible to minimize the synchronization necessary to access the buffers in memory. These optimizations require syntax for the DBA to specify properties (for example, read-only) of tables, as well as properties of named caches (for example, buffer replacement algorithms).

These two areas as well as other MMDB techniques will be explored by the SQL Server developers for incorporation in future releases.

Summary

The Sybase System 11 SQL Server supports VLM systems built and sold by DIGITAL. The SQL Server can completely cache parts of a database in memory. It can also cache the entire database in memory if the database size is smaller than the amount of memory. System 11 has facilities that address issues of fast access, checkpoint, and recovery of VLM systems; these facilities are the Logical Memory Manager, the VLM query optimizer, the Housekeeper, and fuzzy checkpoint. The SQL Server product achieved

SMP TPC performance of 14,176 tpmC at $198/tpmC on a DIGITAL VLM system. The technology developed in System 11 lays the groundwork for further implementation of MMDB techniques in the SQL Server.

Acknowledgments

We gratefully acknowledge the various members of the SQL Server development team who contributed to the VLM capabilities described in this paper.

References and Notes

1. For more information about audited tpmC measurements, see the Transaction Processing Performance Council home page on the World Wide Web, http://www.tpc.org.

2. S.-O. Hvasshovd, O. Torbjornsen, S. Bratsberg, and P. Holager, "The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response," *Proceedings of the 21st Very Large Database Conference,* Zurich, Switzerland, 1995.

3. H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudharshan, "Dali: A High Performance Main Memory Storage Manager," *Proceedings of the 20th Very Large Database Conference Conference,* Santiago, Chile, 1994.

4. M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson, "Smallbase: A Main-Memory DBMS for High-Performance Applications" (1995).

5. H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering,* vol. 4, no. 6 (1992): 509–516.

6. D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *Database Engineering Bulletin,* vol. 8, no. 2 (1985): 3–10.

7. E. Levy and A. Silberschatz, *Incremental Recovery in Main Memory Database Systems* (University of Texas at Austin, Technical Report TR-92-01, January 1992).

8. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Second Edition (San Francisco: Morgan Kaufmann Publishers, Inc., 1995).

9. S. Roy and M. Sugiyama, *Sybase Performance Tuning* (Upper Saddle River, N.J.: Prentice Hall Professional Technical Reference, 1996).

10. *Sybase System 11 SQL Server Documentation Set* (Emeryville, Calif.: Sybase, Inc., 1996).

11. P. Spiro, A. Joshi, and T. Rengarajan, "Designing an Optimized Transaction Commit Protocol," *Digital Technical Journal,* vol. 3, no. 1 (Winter 1991): 70–78.

Biographies

T.K. Rengarajan
T. K. Rengarajan has been building high-performance database systems for the past 10 years. He now leads the Server Performance Engineering and Development (SPeeD) Group in SQL Server Engineering at Sybase, Inc. His most recent focus has been System 11 scalability and self-tuning algorithms. Prior to joining Sybase, he contributed to the DEC Rdb system at DIGITAL in the areas of buffer management, high availability, OLTP performance on Alpha systems, and multimedia databases. He holds M.S. degrees in computer-aided design and computer science from the University of Kentucky and the University of Wisconsin, respectively.

Maxwell Berenson
Max Berenson is a staff software engineer in the Server Performance Engineering and Development Group in SQL Server Engineering at Sybase, Inc. During his four years at Sybase, Max has developed the Logical Memory Manager for System 11 and has made many buffer manager modifications to improve SMP scalability. Prior to joining Sybase, Max worked at DIGITAL, where he developed a relational database engine.

Ganesan Gopal
Ganesan Gopal is a senior member of the Server Performance Engineering and Development Group at Sybase, Inc. He was a member of the team that implemented the Housekeeper in System 11. In addition, he has worked on a number of projects that have enhanced the performance and scaling of the Sybase SQL Server. At present, he is working on a performance feature for an upcoming release. He holds bachelor degrees in advanced physics and in electronics and communication engineering from the Indian Institute of Science, Bangalore, India.

Bruce McCready
Bruce McCready is an SQL Server performance engineer in the Server Performance Engineering and Development Group at Sybase, Inc. Bruce received a B.S. in computer science from the University of California at Berkeley in 1989.

Sapan Panigrahi
A senior performance engineer, Sapan Panigrahi works in the Server Performance Engineering and Development Group at Sybase, Inc. He was responsible for TPC benchmarks and performance analysis for the Sybase SQL Server.

Srikant Subramaniam
A member of the Server Performance Engineering and Development Group at Sybase, Inc., Srikant Subramaniam was involved in the design and implementation of the VLM support in the Sybase SQL Server. He was a member of the team that implemented the Logical Memory Manager in System 11. In addition, he has worked on projects that have enhanced the performance and scaling of the Sybase SQL Server. At present, he is working on performance optimizations for an upcoming release. He holds an M.S. in computer science from the University of Saskatchewan, Canada. His specialty area was the performance of shared-memory multiprocessor systems.

Marc B. Sugiyama
Marc Sugiyama is a staff software engineer in the SQL Server Performance Engineering and Development Group at Sybase, Inc. He was the technical lead for the original port of Sybase SQL Server to the DIGITAL Alpha OSF/1 system. He is coauthor of *Sybase Performance Tuning*, published by Prentice Hall, 1996.

David P. Hunter
Eric B. Betts

Measured Effects of Adding Byte and Word Instructions to the Alpha Architecture

The performance of an application can be expressed as the product of three variables: (1) the number of instructions executed, (2) the average number of machine cycles required to execute a single instruction, and (3) the cycle time of the machine. The recent decision to add byte and word manipulation instructions to the DIGITAL Alpha Architecture has an effect upon the first of these variables. The performance of a commercial database running on the Windows NT operating system has been analyzed to determine the effect of the addition of the new byte and word instructions. Static and dynamic analysis of the new instructions' effect on instruction counts, function calls, and instruction distribution have been conducted. Test measurements indicate an increase in performance of 5 percent and a decrease of 4 to 7 percent in instructions executed. The use of prototype Alpha 21164 microprocessor-based hardware and instruction tracing tools showed that these two measurements are due to the use of the Alpha Architecture's new instructions within the application.

The Alpha Architecture and its initial implementations were limited in their ability to manipulate data values at the byte and word granularity. Instead of allowing single instructions to manipulate byte and word values, the original Alpha Architecture required as many as sixteen instructions. Recently, DIGITAL extended the Alpha Architecture to manipulate byte and word data values with a single instruction. The second generation of the Alpha 21164 microprocessor, operating at 400 megahertz (MHz) or greater, is the first implementation to include the new instructions.

This paper presents the results of an analysis of the effects that the new instructions in the Alpha Architecture have on the performance, code size, and dynamic instruction distribution of a consistent execution path through a commercial database. To exercise the database, we modified the Transaction Processing Performance Council's (TPC) obsolete TPC-B benchmark. Although it is no longer a valid TPC benchmark, the TPC-B benchmark, along with other TPC benchmarks, has been widely used to study database performance.[1-5]

We began our project by rebuilding Microsoft Corporation's SQL Server product to use the new Alpha instructions. We proceeded to conduct a static code analysis of the resulting images and dynamic link libraries (DLLs). The focus of the study was to investigate the impact that the new instructions had upon a large application and not their impact upon the operating system. To this end, we did not rebuild the Windows NT operating system to use the new byte and word instructions.

We measured the dynamic effects by gathering instruction and function traces with several profiling and image analysis tools. The results indicate that the Microsoft SQL Server product benefits from the additional byte and word instructions to the Alpha microprocessor. Our measurements of the images and DLLs show a decrease in code size, ranging from negligible to almost 9 percent. For the cached TPC-B transactions, the number of instructions executed per transaction decreased from 111,288 to 106,521 (a 4 percent reduction). For the scaled TPC-B transactions, the number of instructions executed per transaction decreased from 115,895 to 107,854 (a 7 percent reduction).

The rest of this paper is divided as follows: we begin with a brief overview of the Alpha Architecture and its introduction of the new byte and word manipulation instructions. Next, we describe the hardware, software, and tools used in our experiments. Lastly, we provide an analysis of the instruction distribution and count.

Alpha Architecture

The Alpha Architecture is a 64-bit, load and store, reduced instruction set computer (RISC) architecture that was designed with high performance and longevity in mind. Its major areas of concentration are the processor clock speed, the multiple instruction issue, and multiple processor implementations. For a detailed account of the Alpha Architecture, its major design choices, and overall benefits, see the paper by R. Sites.[6] The original architecture did not define the capability to manipulate byte- and word-level data with a single instruction. As a result, the first three implementations of the Alpha Architecture, the 21064, the 21064A, and the 21164 microprocessors, were forced to use as many as sixteen additional instructions to accomplish this task. The Alpha Architecture was recently extended to include six new instructions for manipulating data at byte and word boundaries. The second implementation of the 21164 family of microprocessors includes these extensions.

The first implementation of the Alpha Architecture, the 21064 microprocessor, was introduced in November 1992. It was fabricated in a 0.75-micrometer (μm) complementary metal-oxide semiconductor (CMOS) process and operated at speeds up to 200 MHz. It had both an 8-kilobyte (KB), direct-mapped, write-through, 32-byte line instruction cache (I-cache) and data cache (D-cache). The 21064 microprocessor was able to issue two instructions per clock cycle to a 7-stage integer pipeline or a 10-stage floating-point pipeline.[7] The second implementation of the 21064 generation was the Alpha 21064A microprocessor, introduced in October 1993. It was manufactured in a 0.5-μm CMOS process and operated at speeds of 233 MHz to 275 MHz. This implementation increased the size of the I-cache and D-cache to 16 KB. Various other differences exist between the two implementations and are outlined in the product data sheet.[8]

The Alpha 21164 microprocessor was the second-generation implementation of the Alpha Architecture and was introduced in October 1994. It was manufactured in a 0.5-μm CMOS technology and has the ability to issue four instructions per clock cycle. It contains a 64-entry data translation buffer (DTB) and a 48-entry instruction translation buffer (ITB) compared to the 21064A microprocessor's 32-entry DTB and 12-entry ITB. The chip contains three on-chip caches. The level one (L1) caches include an 8-KB, direct-mapped I-cache and an 8-KB, dual-ported, direct-mapped, write-through D-cache. A third on-chip cache is a 96-KB, three-way set-associative, write back mixed instruction and data cache. The floating-point pipeline was reduced to nine stages, and the CPU has two integer units and two floating-point execution units.[9]

The Exclusion of Byte and Word Instructions

The original Alpha Architecture intended that operations involved in loading or storing aligned bytes and words would involve sequences as given in Tables 1 and 2.[10] As many as 16 additional instructions are required to accomplish these operations on unaligned data. These same operations in the MIPS Architecture involve only a single instruction: LB, LW, SB, and SW.[11] The MIPS Architecture also includes single instructions to do the same for unaligned data. Given a situation in which all other factors are consistent, this would appear to give the MIPS Architecture an advantage in its ability to reduce the number of instructions executed per workload.

Sites has presented several key Alpha Architecture design decisions.[6] Among them is the decision not to include byte load and store instructions. Key design assumptions related to the exclusion of these features include the following:

- The majority of operations would involve naturally aligned data elements.

Table 1
Loading Aligned Bytes and Words on Alpha

	Load and Sign Extend a Byte
LDL	R1, D.lw(Rx)
EXTBL	R1, #D.mod, R1
	Load and Zero Extend a Byte
LDL	R1, D.lw(Rx)
SLL	R1, #56-8*D.mod, R1
SRA	R1, #56, R1
	Load and Sign Extend a Word
LDL	R1, D.lw(Rx)
EXTWL	R1, #D.mod, R1
	Load and Zero Extend a Word
LDL	R1, D.lw(Rx)
SLL	R1, #48-8*D.mod, R1
SRA	R1, #48, R1

Table 2
Storing Aligned Bytes and Words on Alpha

Store a Byte	
LDL	R1, D.lw(Rx)
INSBL	R5,#D.mod, R3
MSKBL	R1, #D.mod, R1
BIS	R3, R1, R1
STL	R1, D.1w(Rx)
Store a Word	
LDL	R1, D.lw(Rx)
INSWL	R5,#D.mod, R3
MSKWL	R1, #D.mod, R1
BIS	R3, R1, R1
STL	R1, D.1w(Rx)

- In the best possible scheme for multiple instruction issue, single byte and write instructions to memory are not allowed.
- The addition of byte and write instructions would require an additional byte shifter in the load and store path.

These factors indicated that the exclusion of specific instructions to manipulate bytes and words would be advantageous to the performance of the Alpha Architecture.

The decision not to include byte and word manipulation instructions is not without precedents. The original MIPS Architecture developed at Stanford University did not have byte instructions.[12] Hennessy et al. have discussed a series of hardware and software trade-offs for performance with respect to the MIPS processor.[13] Among those trade-offs are reasons for not including the ability to do byte addressing operations. Hennessy et al. argue that the additional cost of including the mechanisms to do byte addressing was not justified. Their studies showed that word references occur more frequently in applications than do byte references. Hennessy et al. conclude that to make a word-addressed machine feasible, special instructions are required for inserting and extracting bytes. These instructions are available in both the MIPS and the Alpha Architectures.

Reversing the Byte and Word Instructions Decision

During the development of the Alpha Architecture, DIGITAL supported two operating systems, OpenVMS and ULTRIX. The developers had as a goal the ability to maintain both customer bases and to facilitate their transitions to the new Alpha microprocessor-based machines. In 1991, Microsoft and DIGITAL began work on porting Microsoft's new operating system, Windows NT, to the Alpha platform. The Windows NT operating system had strong links to the Intel x86 and the MIPS Architectures, both of which included instructions for single byte and word manipulation.[14] This strong connection influenced the Microsoft developers and independent software vendors (ISVs) to favor those architectures over the Alpha design.

Another factor contributed to this issue: the majority of code being run on the new operating system came from the Microsoft Windows and MS-DOS environments. In designing software applications for these two environments, the manipulation of data at the byte and word boundary is prevalent. With the Alpha microprocessor's inability to accomplish this manipulation in a single instruction, it suffered an average of 3:1 and 4:1 instructions per workload on load and store operations, respectively, compared to those architectures with single instructions for byte and word manipulation.

To assist in running the ISV applications under the Windows NT operating system, a new technology was needed that would allow 16-bit applications to run as if they were on the older operating system. Microsoft developed the Virtual DOS Machine (VDM) environment for the Intel Architecture and the Windows-on-Windows (WOW) environment to allow 16-bit Windows applications to work. For non-Intel architectures, Insignia developed a VDM environment that emulated an Intel 80286 microprocessor-based computer. Upon examining this emulator more closely, DIGITAL found opportunities for improving performance if the Alpha Architecture had single byte and word instructions.

Based upon this information and other factors, a corporate task force was commissioned in March 1994 to investigate improving the general performance of Windows NT running on Alpha machines. The further DIGITAL studied the issues, the more convincing the argument became to extend the Alpha Architecture to include single byte and word instructions.

This reversal in position on byte and word instructions was also seen in the evolution of the MIPS Architecture. In the original MIPS Architecture developed at Stanford University, there were no load or store byte instructions.[12] However, for the first commercially produced chip of the MIPS Architecture, the MIPS R2000 RISC processor, developers added instructions for the loading and storing of bytes.[11] One reason for this choice stemmed from the challenges posed by the UNIX operating system. Many implicit byte assumptions inside the UNIX kernel caused performance problems. Since the operating system being implemented was UNIX, it made sense to add the byte instructions to the MIPS Architecture.[15]

In June 1994, one of the coarchitects of the Alpha Architecture, Richard Sites, submitted an Engineering

Change Order (ECO) for the extension of the architecture to include byte and word instructions. It was speculated at the time that an increase of as much as 4 percent in overall performance would be achieved using the new instructions. In June 1995, six new instructions were added to the Alpha Architecture. The new instructions are outlined in Table 3. The first implementation to include support for the new instructions was the second generation of the Alpha 21164 microprocessor series. This reimplementation of the first Alpha 21164 design was manufactured in a 0.35-μm CMOS process and was introduced in October 1995.

Testing Environment

We set up tests to measure the performance of equipment with and without the new instructions. To conduct our experiments, we used prototype hardware that included the second-generation Alpha 21164 microprocessor, and we devised a method to enable and disable the new instructions in hardware. At the same time, we investigated the projected performance of the software emulation mechanism to execute the new instructions on older processors. Finally, we built two separate versions of the Microsoft SQL Server application, one that used the new instructions and one that did not. For the purposes of discussing the different scenarios under study, we summarize the three execution schemes in Table 4. We use the associated nomenclature given there in the rest of this paper. In the remainder of this section, we describe each of the hardware, software, compiler, and analysis tools.

Prototype Hardware

As previously mentioned, our machine was capable of operating with and without the new instructions. By using the same machine, we were able to minimize effects that could be introduced from variations in machine designs or processor families that could cause an increase in the executed code path through the operating system. All experiments were run on a prototype of the AlphaStation 500 workstation that was based upon the second-generation 21164 microprocessor operating at 400 MHz. (The AlphaStation 500 is a family of high-performance, mid-range graphics workstations.) The prototype was configured with 128 megabytes (MB) of memory and a single, 4-gigabyte (GB) fast-wide-differential (FWD) small computer systems interface (SCSI-2) disk.

Table 3
New Byte and Word Manipulation Instructions

Mnemonic	Opcode	Function
stb	0E	Store byte from register to memory
stw	0D	Store word from register to memory
ldbu	0A	Load zero-extended byte from memory to register
ldwu	0C	Load zero-extended word from memory to register
sextb	1C.0000	Sign extend byte
sextw	1C.0001	Sign extend word

Table 4
Three Methods for Execution of the New Instructions

Nomenclature	Description
Original	Compiled with instructions that can execute on all Alpha implementations
Byte/Word	Compiled using the new instructions that will execute on second-generation 21164 implementations at full speed
Emulation	Compiled with new instructions and emulated through software

New firmware allowed us to alternate between direct hardware execution and software emulation of the new byte and word instructions. We modified the Advanced RISC Consortium (ARC) code to allow us to switch between the two firmware versions through a simple power-cycle utility, called the fail-safe loader.[16] When the machine is powered on, it loads code from a serial read-only memory (SROM) storage device. This code then loads the ARC firmware from nonvolatile flash ROM. The fail-safe loader allowed the ARC firmware to be loaded into physical memory and not into the flash ROM. The new firmware was initialized by a reset of the processor and was executed as if it were loaded from the flash ROM. When the machine was turned off and then back on, the version of firmware that was stored in nonvolatile memory was loaded and executed.

Operating System

We used a beta copy of the Microsoft Windows NT version 4.0 operating system. We chose this operating system for its capability to allow us to examine the impact of emulating the new byte and word instructions in the operating system.

By default, version 4.0 of the Windows NT operating system disables the trap and emulation capability for the new instructions. This approach is similar to the one Windows NT provides for the Alpha microprocessor to handle unaligned data references. For testing purposes, we enabled and disabled the trap and emulation capability of the new instructions. When this option is enabled, the operating system treats each new instruction listed in Table 3 as an illegal instruction and emulates the instruction. The trap and emulate strategy takes approximately 5 to 7 microseconds per emulated instruction. When it is disabled or not present, the action taken depends upon the hardware support for the new instructions. If disabled in hardware, the instruction is treated as an illegal instruction; if enabled, it is executed like any other instruction.

Microsoft SQL Server

To observe the effects of the new instructions, we chose the Microsoft SQL Server, a relational database management system (RDBMS) for the Windows NT operating system. Microsoft SQL Server was engineered to be a scalable, multiplatform, multithreaded RDBMS, supporting symmetric multiprocessing (SMP) systems. It was designed specifically for distributed client-server computing, data warehousing, and database applications on the Internet.

In an earlier investigation, Sites and Perl present a profile of the Microsoft SQL Server running the TPC-B benchmark.[4] They identify the executables and DLLs that are involved in running the benchmark and break down the percentage of time that each contributes to the benchmark. Their results, summarized in Figure 1, show that only a few SQL Server executables and DLLs were heavily exercised during the benchmark. After verifying these results with the SQL Server development group at Microsoft, we decided to rebuild only the images and DLLs identified in Figure 1 to use the new byte and word instructions.

Table 5 lists the executables and DLLs that we modified and their correlation to the ones identified by Sites and Perl. The variations exist because of name changes of DLLs or the use of a different network protocol. We changed network protocols for performance reasons. Sites and Perl used an early version of the Microsoft SQL Server version 6.0, in which the fastest network transport available at that time was Named Pipes. In the final release of SQL Server version 6.0 and subsequent versions of the product, the Transmission Control Protocol/Internet Protocol (TCP/IP) replaced Named Pipes in this category. Based upon this, we rebuilt the libraries associated with TCP/IP instead of those associated with Named Pipes. Other networking libraries, such as those for DECnet and Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX), were not rebuilt.

Table 5
Images and DLLs Modified for the Microsoft SQL Server

Sites DLL/EXE	V6.0 DLL/EXE	Function
sqlserver.exe	sqlservr.exe	SQL Server Main Executable
ntwdblib.dll	ntwdblib.dll	Network Communications Library
opends50.dll	opends60.dll	Open Data Services Networking Library
dbnmpntw.dll	N/A	V4.21A Client Side Named Pipes Library
ssnmpntw.dll	N/A	V4.21A Named Pipes Library
N/A	dbmssocn.dll	V6.5 Client Side TCP/IP Library
N/A	ssmsso60.dll	V6.5 Netlibs TCP/IP Library

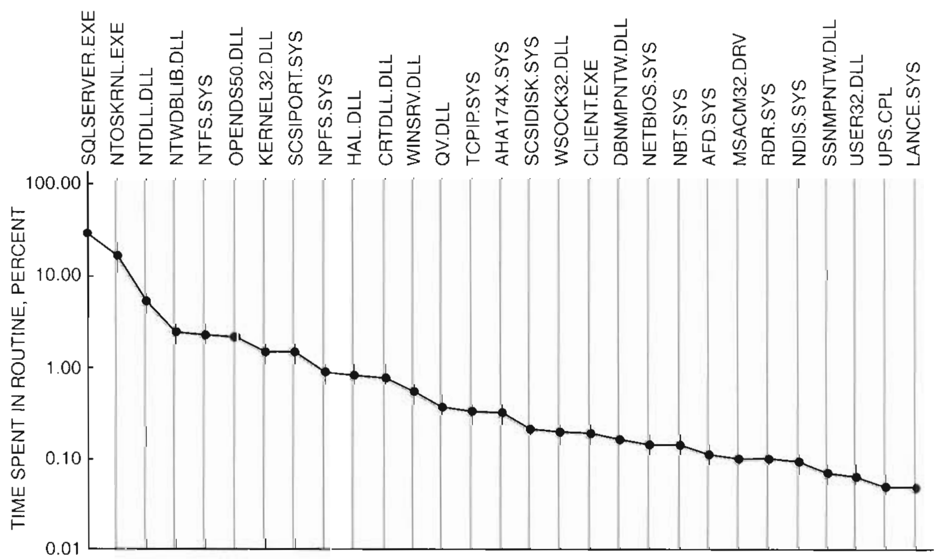


Figure 1
Images/DLLs Involved in a TPC-B Transaction for Microsoft SQL Server Based on Sites and Perl's Analysis

Compiling Microsoft SQL Server to Use the New Instructions

Our goal was to measure only the effects introduced by using the new instructions and not effects introduced by different versions or generations of compilers. Therefore, we needed to find a way to use the same version of a compiler that differed only in its use or nonuse of the new instructions. To do this, we used a compiler option available on the Microsoft Visual C++ compiler. This switch, available on all RISC platforms that support Visual C++, allows the generation of optimized code for a specific processor within a processor family while maintaining binary compatibility with all processors in the processor family. Processor optimizations are accomplished by a combination of specific code-pattern selection and code scheduling. The default action of the compiler is to use a blended model, resulting in code that executes equally well across all processors within a platform family.

Using this compiler option, we built two versions of the aforementioned images within the SQL Server application, varying only their use of the code-generation switch. The first version, referred to as the Original build, was built without specifying an argument for the code-generation switch. The second one, referred to as Byte/Word, set the switch to generate code patterns using the new byte and word manipulation instructions. All other required files came from the SQL Server version 6.5 Beta II distribution CD-ROM.

The Benchmark

The benchmark we chose was derived from the TPC-B benchmark. As previously mentioned, the TPC-B benchmark is now obsolete; however, it is still useful for stressing a database and its interaction with a computer system. The TPC-B benchmark is relatively easy to set up and scales readily. It has been used by both database vendors and computer manufacturers to measure the performance of either the computer system or the actual database. We did not include all the required metrics of the TPC-B benchmark; therefore, it is not in full compliance with published guidelines of the TPC. We refer to it henceforth simply as the application benchmark.

The application benchmark is characterized by significant disk I/O activity, moderate system and application execution time, and transaction integrity. The application benchmark exercises and measures the efficiency of the processor, I/O architecture, and RDBMS. The results measure performance by indicating how many simulated banking transactions can be completed per second. This is defined as transactions per second (tps) and is the total number of committed transactions that were started and completed during the measurement interval.

The application benchmark can be run in two different modes: cached and scaled. The cached, or in-memory mode, is used to estimate the system's maximum performance in this benchmark environment. This is accomplished by building a small database that resides completely in the database cache, which in turn fits within the system's physical random-access memory (RAM). Since the entire database resides in memory, all I/O activity is eliminated with the exception of log writes. Consequently, the benchmark only performs one disk I/O for each transaction, once the entire database is read off the disk and into the database cache. The result is a representation of the maximum number of tps that the system is capable of sustaining.

The scaled mode is run using a bigger database with a larger amount of disk I/O activity. The increase in disk I/O is a result of the need to read and write data to locations that are not within the database cache. These additional reads and writes add extra disk I/Os. The result is normally characterized as having to do one read and one write to the database and a single write to the transaction log for each transaction. The combination of a larger database and additional I/O activity decreases the tps value from the cached version. Based upon our previous experience running this benchmark, the scaled benchmark can be expected to reach approximately 80 percent of the cached performance.

For the scaled tests, we built a database sized to accommodate 50 tps. This was less than 80 percent of the maximum tps produced by the cached results. We chose this size because we were concentrating on isolating a single scaled transaction under a moderate load and not under the maximum scaled performance possible.

Image Tracing and Analysis Tools

Collecting only static measurements of the executables and DLLs affected was insufficient to determine the applicability of the new instructions. We collected the actual instruction traces of SQL Server while it executed the application benchmark. Furthermore, we decided that the ability to trace the actual instructions being executed was more desirable than developing or extending a simulator. To obtain the traces, we needed a tool that would allow us to

- Collect both system- and user-mode code.
- Collect function traces, which would allow us to align the starting and stopping points of different benchmark runs.
- Work without modifying either the application or the operating system.

In the past, the only tool that would provide instruction traces under the Windows NT operating system was the debugger running in single-step mode.

Obtaining traces through either the ntsd or the windbg debugger is quite limited due to the following problems:

- The tracing rate is only about 500 instructions per second. This is far too slow to trace anything other than isolated pieces of code.
- The trace fails across system calls.
- The trace loops infinitely in critical section code.
- Register contents are not easily displayed for each instruction.
- Real-time analysis of instruction usage and cache misses are not possible.

Instruction traces can also be obtained using the PatchWrks trace analysis tool.[4] Although this tool operates with near real-time performance and can trace instructions executing in kernel mode, it has the following limitations:

- It operates only on a DIGITAL Alpha AXP personal computer.
- It requires an extra 40 MB of memory.
- All images to be traced must be patched, thus slightly distorting text addresses and function sizes.
- Successive runs of application code are not repeatable due to unpredictable kernel interrupt behavior (the traces are too accurate).

The solution was Ntstep, a tool that can trace user-mode instruction execution of any image in the Windows NT/Alpha environment through an innovative combination of breakpointing and "Alpha-on-Alpha" emulation. It has the ability to trace a program's execution at rates approaching a million instructions per second. Ntstep can trace individual instructions, loads, stores, function calls, I-cache and D-cache misses, unaligned data accesses, and anything else that can be observed when given access to each instruction as it is being executed. It produces summary reports of the instruction distribution, cache line usage, page usage (working set), and cache simulation statistics for a variety of Alpha systems.

Ntstep acts like a debugger that can execute single-step instructions except that it executes instructions using emulation instead of single-step breakpoints whenever possible. In practice, emulation accounts for the majority of instructions executed within Ntstep. Since a single-step execution of an instruction with breakpoints takes approximately 2 milliseconds and emulation of an Alpha instruction requires only 1 or 2 microseconds, Ntstep can trace approximately 1,000 times faster than a debugger. Unlike most emulators, the application executes normally in its own address space and environment.

Results

We collected data on three different experiments. In the first investigation, we looked at the relative performance of the three different versions of the Microsoft SQL Server outlined in Table 4. We compared the three variations using the cached version of the application benchmark.

In the second experiment, we observed how the new instructions affect the instruction distribution in the static images and DLLs that we rebuilt. We compared the Byte/Word versions to the Original versions of the images and DLLs. We also attempted to link the differences in instruction counts to the use of the new instructions.

Lastly, we investigated the variation between the Original and the Byte/Word versions with respect to instruction distribution on the scaled version of the benchmark. This comparison was based upon the code path executed by a single transaction.

Cached Performance

In the first experiments, we compared the relative performance impact of using the new instructions. We chose to measure performance of only the cached version of the application benchmark because the I/O subsystem available on the prototype of the AlphaStation 500 was not adequate for a full-scaled measurement. We ensured that the database was fully cached by using a ramp-up period of 60 seconds and a ramp-down period of 30 seconds. This was verified as steady state by observing that the SQL Server buffer cache hit ratio remained at or above 95 percent. The measurement period for the benchmark was 60 seconds. We ran the benchmark several times and took the average tps for each of the three variations outlined in Table 4.

The results of the three schemes are as follows: 444 tps for the Original version, 460 tps for the Byte/Word version, and 116 tps for the Emulation version. The new instructions contributed a 3.5 percent gain in performance. The impact of emulating the instructions is a loss of 73.9 percent of the potential performance.

Static Instruction Counts

To analyze the mixture of instructions in the images and DLLs, we disassembled each image and DLL in the Original and Byte/Word versions. We then looked at only those instructions that exhibited a difference between the two versions within the images or DLLs. The variations in instruction counts of these are shown in Table 6.

To examine the images more closely, we disassembled each image and DLL and collected counts of code

Table 6
Instruction Deltas (Normal Minus Byte/Word) for the SQL Server Images and DLLs

Instruction	dbmssocn.dll	ntwdblib.dll	opends60.dll	sqlservr.exe	ssmsso60.dll
lda	0	−3	−247	−8524	−4
ldah	0	0	−27	18−18	0
ldl	−9	−11	−597	−13133	−46
ldq	0	0	−29	−2980	0
ldq_l	0	0	0	−9	0
ldq_u	−10	−2	−311	−8529	−18
stl	−5	−11	−278	−7932	−11
stb	+3	−1	+216	+3969	+7
stw	+2	−5	+59	+2798	+3
stq	0	0	−4	−53	0
stq_c	0	0	0	−9	0
beq	0	5	+1	−1236	0
bge	0	0	0	+8	0
bgt	0	0	0	+3	0
blbc	0	0	−1	−19	0
blbs	0	0	0	−4	0
blt	0	0	0	0	0
bne	0	0	+1	+24	0
br	0	−4	+1	−1120	0
bsr	0	0	0	−6	0
ret	0	0	+4	+15	0
cmpeq	0	0	0	+9	0
cmplt	0	0	0	+15	0
cmple	0	0	0	+5	0
cmpult	0	0	−1	1−1	0
cmpule	0	−5	−2	1183–1183	0
and	−2	−6	−364	−6435	−8
bic	−3	−11	−287	−7242	−8
bis	−4	−7	−208	−7097	−9
ornot	0	0	0	+4	0
xor	0	0	−2	+119	0
sll	0	0	+2	−2359	0
sra	0	0	−15	−3534	−4
srl	0	0	0	−295	0
cmpbge	0	0	−1	−18	0
mskbl	−3	−1	−196	−3647	−8
mskwl	0	5	−41	−1604	0
zapnot	−5	0	−115	−2135	−33
addl	0	0	0	−8	0
addq	0	0	0	+3	0
s4addl	0	0	0	−4	0
cmovge	0	0	0	+1	0
cmovne	0	0	0	+2	0
cmovlt	0	0	0	−1	0
cmovlbc	0	0	0	−2	0
callsys	0	0	0	0	0
extqh	0	0	−14	−426	−4
ldwu	+4	0	+193	+6320	+35
ldbu	+9	+3	+464	+10231	+18
mull	0	0	0	+1	0
subl	0	0	+1	+6	0
subq	0	0	0	+3	0
insll	0	0	0	1−1	0
inswl	−2	−3	−54	−2647	−3
call_pal	+2	+1	+1	+161	0
extlh	0	0	0	−14	0
insbl	−2	−1	−135	−3163	−6
extll	0	0	0	−20	0
extbl	−10	−6	−367	−10656	−14
extwl	−1	0	−84	−324	−1

size, the number of functions, the number and type of new byte and word instructions, and lastly, nop and trapb instructions. The results are presented in Tables 7 through 10.

We expected that the instructions used to manipulate bytes and words in the original Alpha Architecture (Tables 1 and 2) would decrease proportionally to the usage of the new instructions. These assumptions held true for all the images and DLLs that used the new instructions. For example, in the original Alpha Architecture, the instructions MSKBL and MSKWL are used to store a byte and word, respectively. In the sqlservr.exe image, these two instructions showed a decrease of 3,647 and 1,604 instructions, respectively. Compare this with the corresponding addition of 3,969 STB and 2,798 STW instructions in the same image. Looking further into the sqlservr.exe image, we also saw that 10,231 LDBU instructions were used and the usage of the EXTBL instruction was reduced by 10,656. Although these numbers do not correlate on a one-for-one basis, we believe this is due to other usage of these instructions. Other usage might include the compiler scheme for introducing the new instructions in places where it used an LDL or an LDQ in the Original image.

Of the rebuilt images and DLLs, sqlservr.exe and opends60.dll showed the most variations, with the new instructions making up 3.73 percent and 3.9 percent of these files. The most frequently occurring new instruction was ldbu, followed by ldwu. The least-used instructions were sextb and sextw. The size of the images was reduced in three out of five images. The image size reduction ranged from negligible to just over 4 percent. In all cases, the size of the code section was reduced and ranged from insignificant to approximately 8.5 percent. There was no change in the number of functions in any of the files.

Dynamic Instruction Counts

We gathered data from the application benchmark running in both cached and scaled modes. We ran at least one iteration of the benchmark test prior to gathering trace data to allow both the Windows NT operating system and the Microsoft SQL Server database to reach a steady state of operation on the system under test (SUT). Steady state was achieved when the SQL Server cache-hit ratio reached 95 percent or greater, the number of transactions per second was constant, and the CPU utilization was as close to 100 percent as possible. The traces were gathered over a sufficient

Table 7
Byte/Word Images and DLLs

Image/DLL	Total File Bytes	Total Text Bytes	Total Code Bytes	Number of Functions	Total Byte/ Word	% Byte/ Word	LDBU Count	LDBU %	LDWU Count	LDWU %	STB Count	STB %	STW Count	STW %	SEXTB Count	SEXTB %	SEXTW Count	SEXTW %	Total NOPs	Total TRAPB
sqlservr.exe	8053624	2981148	2884776	3364	26869	3.73	10231	38.077	6320	23.5215	3969	14.7717	2798	10.4135	139	0.517325	3412	12.6986	5929	2219
dbmssocn.dll	13824	5884	5520	13	18	1.3	9	50	4	22.2222	3	16.6667	2	11.1111	0	0	0	0	21	0
ntwdblib.dll	318464	246316	231688	429	9	0.02	3	33.333	0	0	1	11.1111	5	55.5556	0	0	0	0	767	10
opends60.dll	212992	104204	97240	243	948	3.9	464	48.945	193	20.3586	216	22.7848	59	6.22363	9	0.949367	7	0.738397	391	128
ssmsso60.dll	70760	9884	9128	19	67	2.94	18	26.866	35	52.2388	7	10.4478	3	4.47761	4	5.97015	0	0	25	0

Table 8
Original Build of Images and DLLs

Image/DLL	Total File Bytes	Total Text Bytes	Total Code Bytes	Number of Functions	Total Byte/ Word	% Byte/ Word	LDBU Count	LDBU %	LDWU Count	LDWU %	STB Count	STB %	STW Count	STW %	SEXTB Count	SEXTB %	SEXTW Count	SEXTW %	Total NOPs	Total TRAPB
sqlservr.exe	8337248	3264108	3153480	3364	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6207	2252
dbmssocn.dll	13824	6012	5656	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	0
ntwdblib.dll	318464	246620	231904	429	0	0	0	0	0	0	0	0	0	0	0	0	0	0	770	10
opends60.dll	222720	114012	105536	243	0	0	0	0	0	0	0	0	0	0	0	0	0	0	405	128
ssmsso60.dll	71284	10300	9424	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0

Table 9
Numerical Differences of Original Minus Byte/Word Images and DLLs

Image/DLL	Total File Bytes	Total Text Bytes	Total Code Bytes	Number of Functions	Total Byte/ Word	% Byte/ Word	LDBU Count	LDBU %	LDWU Count	LDWU %	STB Count	STB %	STW Count	STW %	SEXTB Count	SEXTB %	SEXTW Count	SEXTW %	Total NOPs	Total TRAPB
lsqlservr.exe	-283624	-282960	-268704	0	+26869	+4	+10231	+38	+6320	+24	+3969	+15	+2798	+10	+139	+1	+3412	+13	-278	-33
dbmssocn.dll	0	-128	-136	0	+18	+1	+9	+50	+4	+22	+3	+17	+2	+11	0	0	0	0	+5	0
ntwdblib.dll	0	-304	-216	0	+9	0	+3	+33	0	0	+1	+11	+5	+56	0	0	0	0	-3	0
opends60.dll	-9728	-9808	-8296	0	+948	+4	+464	+49	+193	+20	+216	+23	+59	+6	+9	+1	+7	+1	-14	0
ssmsso60.dll	-524	-416	-296	0	+67	+3	+18	+27	+35	+52	+7	+10	+3	+4	+4	+6	0	0	+7	0

Table 10
Percentage Variation of Original Minus Byte/Word Images and DLLs

Image/DLL	Total File Bytes	Total Text Bytes	Total Code Bytes	Number of Functions	Total Byte/ Word	% Byte/ Word	LDBU Count	LDBU %	LDWU Count	LDWU %	STB Count	STB %	STW Count	STW %	SEXTB Count	SEXTB %	SEXTW Count	SEXTW %	Total NOPs	Total TRAPB
sqlservr.exe	-3.402%	-8.669%	-8.521%	0.000%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-4.479%	-1.465%
dbmssocn.dll	0.000%	-2.129%	-2.405%	0.000%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	+31.250%	N/A
ntwdblib.dll	0.000%	-0.123%	-0.093%	0.000%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-0.390%	0.000%
opends60.dll	-4.368%	-8.603%	-7.861%	0.000%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-3.457%	0.000%
ssmsso60.dll	-0.735%	-4.039%	-3.141%	0.000%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	+38.889%	N/A

period of time to ensure that we captured several transactions. The traces were then edited into separate individual transactions. The geometric mean was taken from the resulting traces and used for all subsequent analysis.

We used Ntstep to gather complete instruction and function traces of both versions of the SQL Server database while it executed the application benchmark. Figure 2 shows an example output for an instruction trace, and Figure 3 shows an example output for a function trace from Ntstep. Since Ntstep can attach to a running process, we allowed the application benchmark to achieve steady state prior to data collection. This approach ensured that we did not see the effects of warming up either the machine caches or the SQL Server database cache. Each instruction trace consisted of approximately one million instructions, which was sufficient to cover multiple transactions. The data was

```
  0   ** Breakpoint (Pid 0xd1, Tid 0xb2) SQLSERVR.EXE pc 77f39b34
  0   ** Trace begins at 242698
      opends60!FetchNextCommand
  1   00242698: 23deffb0   lda       sp, -50(sp)          // sp now 72bff00
  2   0024269c: b53e0000   stq       s0, 0(sp)            // @072bff00 = 148440
  3   002426a0: b55e0008   stq       s1, 8(sp)            // @072bff08 = 0
  4   002426a4: b57e0010   stq       s2, 10(sp)           // @072bff10 = 5
  5   002426a8: b59e0018   stq       s3, 18(sp)           // @072bff18 = 1476a8
  6   002426ac: b5be0020   stq       s4, 20(sp)           // @072bff20 = 2c4
  7   002426b0: b5de0028   stq       s5, 28(sp)           // @072bff28 = 41
  8   002426b4: b5fe0030   stq       fp, 30(sp)           // @072bff30 = 0
  9   002426b8: b75e0038   stq       ra, 38(sp)           // @072bff38 = 242398
 10   002426bc: 47f00409   bis       zero, a0, s0         // s0 now 148440
 11   002426c0: 47f1040a   bis       zero, a1, s1         // s1 now 72bffa0
 12   002426c4: 47f2040b   bis       zero, a2, s2         // s2 now 72bffa8
 13   002426c8: d3404e67   bsr       ra, 00256068         // ra now 2426cc
      opends60!netIOReadData
 14   00256068: 23deffa0   lda       sp, -60(sp)          // sp now 72bfea0
 15   0025606c: 43f10002   addl      zero, a1, t1         // t1 now 72bffa0
 16   00256070: b53e0000   stq       s0, 0(sp)            // @072bfea0 = 148440
 17   00256074: b55e0008   stq       s1, 8(sp)            // @072bfea8 = 72bffa0
 18   00256078: b57e0010   stq       s2, 10(sp)           // @072bfeb0 = 72bffa8
 19   0025607c: b59e0018   stq       s3, 18(sp)           // @072bfeb8 = 1476a8
 20   00256080: b5be0020   stq       s4, 20(sp)           // @072bfec0 = 2c4
 21   00256084: b5de0028   stq       s5, 28(sp)           // @072bfec8 = 41
 22   00256088: b5fe0030   stq       fp, 30(sp)           // @072bfed0 = 0
 23   0025608c: b75e0038   stq       ra, 38(sp)           // @072bfed8 = 2426cc
 24   00256090: a1d01140   ldl       s5, 1140(a0)         // @00149580 1479e8
 25   00256094: 47f00409   bis       zero, a0, s0         // s0 now 148440
 26   00256098: a1f001d0   ldl       fp, 1d0(a0)          // @00148610 dbba0
 27   0025609c: 47e0340d   bis       zero, #1, s4         // s4 now 1
 28   002560a0: a0620000   ldl       t2, 0(t1)            // @072bffa0 155c58
 29   002560a4: b23e004c   stl       a1, 4c(sp)           // @072bfeec = 72bffa0
 30   002560a8: b25e0050   stl       a2, 50(sp)           // @072bfef0 = 72bffa8
 31   002560ac: b27e0054   stl       a3, 54(sp)           // @072bfef4 = 1476a8
 32   002560b0: e460001d   beq       t2, 00256128         // (t2 is 155c58)
 33   002560b4: 220303e0   lda       a0, 3e0(t2)          // a0 now 156038
 34   002560b8: 47f00404   bis       zero, a0, t3         // t3 now 156038
 35   002560bc: 63ff4000   mb                             //
 36   002560c0: 47e03400   bis       zero, #1, v0         // v0 now 1
 37   002560c4: a8240000   ldl_l     t0, 0(t3)            // @00156038 0
 38   002560c8: b8040000   stl_c     v0, 0(t3)            // @00156038 = 1
 39   002560cc: e40000b6   beq       v0, 002563a8         // (v0 is 1)
 40   002560d0: 63ff4000   mb                             //
 41   002560d4: e4200001   beq       t0, 002560dc         // (t0 is 0)
      opends60!netIOReadData+0x74:
 42   002560dc: a1be004c   ldl       s4, 4c(sp)           // @072bfeec 72bffa0
 43   002560e0: a00d0000   ldl       v0, 0(s4)            // @072bffa0 155c58
 44   002560e4: a04003dc   ldl       t1, 3dc(v0)          // @00156034 0
 45   002560e8: 20800404   lda       t3, 404(v0)          // t3 now 15605c
 46   002560ec: 405f05a2   cmpeq     t1, zero, t1         // t1 now 1
 47   002560f0: e4400003   beq       t1, 00256100         // (t1 is 1)
 48   002560f4: a0600404   ldl       t2, 404(v0)          // @0015605c 15605c
 49   002560f8: 406405a3   cmpeq     t2, t3, t2           // t2 now 1
 50   002560fc: 47e30402   bis       zero, t2, t1         // t1 now 1
 51   00256100: 47e2040d   bis       zero, t1, s4         // s4 now 1
```

Figure 2
Example of Instruction Trace Output from Ntstep

```
52  00256104: e4400005   beq       t1, 0025611c        // (t1 is 1)
53  00256108: a0a00000   ldl       t4, 0(v0)           // @00155c58 204200
54  0025610c: 24df0080   ldah      t5, 80(zero)        // t5 now 800000
55  00256110: 48a07625   zapnot    t4, #3, t4          // t4 now 4200
56  00256114: 40a60005   addl      t4, t5, t4          // t4 now 804200
57  00256118: b0a00000   stl       t4, 0(v0)           // @00155c58 = 804200
58  0025611c: a0fe004c   ldl       t6, 4c(sp)          // @072bfeec 72bffa0
59  00256120: a0e70000   ldl       t6, 0(t6)           // @072bffa0 155c58
60  00256124: b3e703e0   stl       zero, 3e0(t6)       // @00156038 = 0
61  00256128: e5a00061   beq       s4, 002562b0        // (s4 is 1)
62  0025612c: 257f0026   ldah      s2, 26(zero)        // s2 now 260000
63  00256130: 216b62f8   lda       s2, 62f8(s2)        // s2 now 2662f8
64  00256134: 5fff041f   cpys      f31, f31, f31       //
65  00256138: a21e0054   ldl       a0, 54(sp)          // @072bfef4 1476a8
66  0025613c: 225e0040   lda       a2, 40(sp)          // a2 now 72bfee0
67  00256140: a00b0000   ldl       v0, 0(s2)           // @002662f8 77e985a0
68  00256144: 227e0048   lda       a3, 48(sp)          // a3 now 72bfee8
69  00256148: a23e0050   ldl       a1, 50(sp)          // @072bfef0 72bffa8
70  0025614c: 47ef0414   bis       zero, fp, a4        // a4 now dbba0
71  00256150: a2100000   ldl       a0, 0(a0)           // @001476a8 2c0
72  00256154: 6b404000   jsr       ra, (v0),0          // ra now 256158
    KERNEL32!GetQueuedCompletionStatus:
73  77e985a0: 23deffc0   lda       sp, -40(sp)         // sp now 72bfe60
74  77e985a4: b53e0000   stq       s0, 0(sp)           // @072bfe60 = 148440
75  77e985a8: b55e0008   stq       s1, 8(sp)           //@072bfe68 = 72bffa0
76  77e985ac: b57e0010   stq       s2, 10(sp)          // @072bfe70 = 2662f8
77  77e985b0: b59e0018   stq       s3, 18(sp)          // @072bfe78 = 1476a8
78  77e985b4: b75e0020   stq       ra, 20(sp)          // @072bfe80 = 256158
79  77e985b8: 47f00409   bis       zero, a0, s0        // s0 now 2c0
80  77e985bc: 47f1040a   bis       zero, a1, s1        // s1 now 72bffa8
81  77e985c0: 47f2040b   bis       zero, a2, s2        // s2 now 72bfee0
82  77e985c4: 47f3040c   bis       zero, a3, s3        // s3 now 72bfee8
83  77e985c8: 47f40411   bis       zero, a4, a1        // a1 now dbba0
84  77e985cc: 221e0038   lda       a0, 38(sp)          // a0 now 72bfe98
85  77e985d0: d3405893   bsr       ra, 77eae820        // ra now 77e985d4
```

Figure 2 (continued)
Example of Instruction Trace Output from Ntstep

then reduced to a series of single transactions and analyzed for instruction distribution. For both the cached- and the scaled-transaction instruction counts, we combined at least three separate transactions and took the geometric mean of the instructions executed, which caused slight variations in the instruction counts. All resulting instruction counts were within an acceptable standard deviation as compared to individual transaction instruction counts.

We collected the function traces in a similar fashion. Once the application benchmark was at a steady state, we began collecting the function call tree. Based on previous work with the SQL Server database and consultation with Microsoft engineers, we could pinpoint the beginning of a single transaction. We then began collecting samples for both traces at the same instant, using an Ntstep feature that allowed us to start or stop sample collection based upon a particular address.

The dynamic instruction counts for both the scaled and the cached transactions are given in Tables 11 and 12. We also show the variation and percentage variation between the Original and the Byte/Word versions of the SQL Server. Two of the six new instructions, sextb and sextw, are not present in the Byte/Word trace. The remaining four instructions combine to make up 2.6 percent and 2.7 percent of the instructions executed per scaled and cached transaction, respectively. Other observations include the following:

- The number of instructions executed decreased 7 percent for scaled and 4 percent for cached transactions.
- The number of ldl_l/stl_c sequences decreased 3 percent for scaled transactions.
- All the instructions that are identified in Tables 1 and 2 show a decrease in usage. Not surprisingly, the instructions mskwl and mskbl completely disappeared. The inswl and insbl instructions decreased by 47 percent and 90 percent, respectively. The sll instruction decreased by 38 percent, and the sra instruction usage decreased by 53 percent. These reductions hold true within 1 to 2 percent for both scaled and cached transactions.
- The instructions ldq_u and lda, which are used in unaligned load and store operations, show a decrease in the range of 20 to 22 percent and 15 to 16 percent, respectively.

```
   0   **   Breakpoint (Pid 0xd7, Tid 0xdb) SQLSERVR.EXE pc 77f39b34
   0   **   Trace begins at 00242698
   0   **   . opends60!FetchNextCommand
  13   **   . . opends60!netIOReadData
  72   **   . . . KERNEL32!GetQueuedCompletionStatus
  85   **   . . . . KERNEL32!BaseFormatTimeOut
  99   **   . . . . ntdll!NtRemoveIoCompletion
 129   **   . . . opends60!netIOCompletionRoutine
 272   **   . . opends60!netIORequestRead
 285   **   . . . KERNEL32!ResetEvent
 290   **   . . . . ntdll!NtClearEvent
 318   **   . . . SSNMPN60!*0x06a131f0*
 348   **   . . . . KERNEL32!ReadFile
 399   **   . . . . . ntdll!NtReadFile
 412   **   . . . . . KERNEL32!BaseSetLastNTError
 417   **   . . . . . . ntdll!RtlNtStatusToDosError
 423   **   . . . . . . . ntdll!RtlNtStatusToDosErrorNoTeb
 509   **   . . . . KERNEL32!GetLastError
 560   **   . opends60!get_client_event
 665   **   . . opends60!processRPC
 682   **   . . . opends60!unpack_rpc
 749   **   . opends60!execute_event
 762   **   . . opends60!execute_sqlserver_event
 802   **   . . . opends60!unpack_rpc
 864   **   . . . SQLSERVR!execrpc
 911   **   . . . . KERNEL32!WaitForSingleObjectEx
 937   **   . . . . . KERNEL32!BaseFormatTimeOut
 950   **   . . . . . ntdll!NtWaitForSingleObject
1024   **   . . . . SQLSERVR!UserPerfStats
1038   **   . . . . . KERNEL32!GetThreadTimes
1055   **   . . . . . . ntdll!NtQueryInformationThread
1173   **   . . . . SQLSERVR!init_recvbuf
1208   **   . . . . SQLSERVR!init_sendbuf
1227   **   . . . . SQLSERVR!port_ex_handle
1263   **   . . . . SQLSERVR!_Otssetjmp3
1313   **   . . . . SQLSERVR!memalloc
1365   **   . . . . . SQLSERVR!_OtsZero
1405   **   . . . . SQLSERVR!recvhost
1437   **   . . . . . SQLSERVR!_OtsMove
1500   **   . . . . SQLSERVR!memalloc
1577   **   . . . . SQLSERVR!rn_char
1580   **   . . . . . SQLSERVR!recvhost
1612   **   . . . . . . SQLSERVR!_OtsMove
1777   **   . . . . SQLSERVR!parse_name
1808   **   . . . . . SQLSERVR!dbcs_strnchr
2115   **   . . . . SQLSERVR!rpcprot
2131   **   . . . . . SQLSERVR!memalloc
2183   **   . . . . . . SQLSERVR!_OtsZero
2252   **   . . . . . SQLSERVR!getprocid
2319   **   . . . . . . SQLSERVR!procrelink+0x1250
2546   **   . . . . . . . SQLSERVR!_OtsRemainder32
2559   **   . . . . . . . SQLSERVR!_OtsDivide32+0x94
2597   **   . . . . . . SQLSERVR!opentable
2642   **   . . . . . . . SQLSERVR!parse_name
2673   **   . . . . . . . . SQLSERVR!dbcs_strnchr
2979   **   . . . . . . . SQLSERVR!parse_name
3010   **   . . . . . . . . SQLSERVR!dbcs_strnchr
3323   **   . . . . . . . SQLSERVR!opentabid
3363   **   . . . . . . . . SQLSERVR!getdes
3493   **   . . . . . . . . SQLSERVR!GetRunidFromDefid+0x40
3510   **   . . . . . . . . . SQLSERVR!_OtsZero
3658   **   . . . . . . . SQLSERVR!initarg
3668   **   . . . . . . . SQLSERVR!setarg
3703   **   . . . . . . . . SQLSERVR!_OtsFieldInsert
3764   **   . . . . . . . SQLSERVR!setarg
3799   **   . . . . . . . . SQLSERVR!_OtsFieldInsert
3857   **   . . . . . . . SQLSERVR!startscan
3901   **   . . . . . . . . SQLSERVR!getindex2
3978   **   . . . . . . . . . SQLSERVR!getkeepslot
4064   **   . . . . . . . . . SQLSERVR!rowoffset
4109   **   . . . . . . . . . SQLSERVR!rowoffset
4170   **   . . . . . . . . . SQLSERVR!_OtsMove
4331   **   . . . . . . . . . SQLSERVR!memcmp
5323   **   . . . . . . . . . SQLSERVR!bufunhold
5436   **   . . . . . . . . SQLSERVR!prepscan
5550   **   . . . . . . . . . SQLSERVR!match_sargs_to_index
```

Figure 3
Example of Function Trace Output from Ntstep

```
 5828  **   . . . . . . . . . SQLSERVR!srchindex
 5895  **   . . . . . . . . . . SQLSERVR!getpage
 5942  **   . . . . . . . . . . . SQLSERVR!bufget
 5976  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide
 5985  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide32+0x94
 6090  **   . . . . . . . . . . . . SQLSERVR!getkeepslot
 6356  **   . . . . . . . . . . SQLSERVR!bufrlockwait
 6539  **   . . . . . . . . . . SQLSERVR!srchpage
 6720  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 6912  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 7309  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 7728  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8125  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8522  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8919  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 9410  **   . . . . . . . . . . SQLSERVR!index_beforesleep+0x100
 9465  **   . . . . . . . . . . . SQLSERVR!bufrunlock
 9641  **   . . . . . . . . SQLSERVR!trim_sqoff+0xf0
 9661  **   . . . . . . . . SQLSERVR!qualpage
 9809  **   . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
10212  **   . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
10616  **   . . . . . . . . . SQLSERVR!rowoffset
10702  **   . . . . . . . SQLSERVR!getnext
10769  **   . . . . . . . . SQLSERVR!_OtsFieldInsert
10822  **   . . . . . . . . SQLSERVR!getrow2
10838  **   . . . . . . . . . SQLSERVR!getpage
10885  **   . . . . . . . . . . SQLSERVR!bufget
10919  **   . . . . . . . . . . . SQLSERVR!_OtsDivide
10928  **   . . . . . . . . . . . SQLSERVR!_OtsDivide32+0x94
11033  **   . . . . . . . . . . . SQLSERVR!getkeepslot
11359  **   . . . . . . . SQLSERVR!_OtsMove
11489  **   . . . . . . . SQLSERVR!endscan
11557  **   . . . . . . . . SQLSERVR!bufunkeep
11675  **   . . . . . . . . SQLSERVR!bufunkeep
11853  **   . . . . . . . SQLSERVR!closetable
11907  **   . . . . . . . . SQLSERVR!endscan
12044  **   . . . . . . . . SQLSERVR!get_spinlock
12103  **   . . . . . . . SQLSERVR!opentabid
12138  **   . . . . . . . . SQLSERVR!getdes
12291  **   . . . . . . . . SQLSERVR!_OtsZero
12464  **   . . . . . . SQLSERVR!closetable
12524  **   . . . . . . . SQLSERVR!endscan
12661  **   . . . . . . . SQLSERVR!get_spinlock
12729  **   . . . . . SQLSERVR!protect
12756  **   . . . . . . SQLSERVR!port_ex_handle
12792  **   . . . . . . SQLSERVR!_Otssetjmp3
12845  **   . . . . . . SQLSERVR!prot_search
12887  **   . . . . . . . SQLSERVR!dbtblfind
12958  **   . . . . . . SQLSERVR!check_protect
13025  **   . . . . SQLSERVR!memalloc
13077  **   . . . . . SQLSERVR!_OtsZero
13127  **   . . . . SQLSERVR!memalloc
13179  **   . . . . . SQLSERVR!_OtsZero
13263  **   . . . . SQLSERVR!rn_i2
13267  **   . . . . . SQLSERVR!recvhost
13299  **   . . . . . . SQLSERVR!_OtsMove
13369  **   . . . . SQLSERVR!recvhost
13401  **   . . . . . SQLSERVR!_OtsMove
13477  **   . . . . SQLSERVR!recvhost
13509  **   . . . . . SQLSERVR!_OtsMove
13562  **   . . . . SQLSERVR!recvhost
13594  **   . . . . . SQLSERVR!_OtsMove
13670  **   . . . . SQLSERVR!recvhost
13702  **   . . . . . SQLSERVR!_OtsMove
13755  **   . . . . SQLSERVR!recvhost
13787  **   . . . . . SQLSERVR!_OtsMove
13847  **   . . . . SQLSERVR!bconst
13895  **   . . . . . SQLSERVR!mkconstant
13921  **   . . . . . . SQLSERVR!memalloc
14046  **   . . . . . . SQLSERVR!memalloc
14098  **   . . . . . . . SQLSERVR!_OtsZero
14157  **   . . . . . SQLSERVR!rn_i4
14161  **   . . . . . . SQLSERVR!recvhost
14193  **   . . . . . . . SQLSERVR!_OtsMove
```

Figure 3 (continued)
Example of Function Trace Output from Ntstep

Table 11
Instruction Count and Variations for Scaled Transaction

Instruction	Original	Byte/Word	Delta	% Delta	Instruction	Original	Byte/Word	Delta	% Delta
stb	0	174	+174	N/A	stt	334	334	0	0%
stw	0	219	+219	N/A	cmple	368	358	10	−3%
ldwu	0	1215	+1215	N/A	inswl	390	207	183	−47%
ldbu	0	1216	+1216	N/A	srl	457	398	59	−13%
cmpbge	2	0	−2	−100%	extqh	441	317	124	−28%
cmovlbs	2	2	0	0%	cmpule	468	450	18	−4%
addt	3	3	0	0%	cmpult	563	518	45	−8%
cmovlbc	5	4	−1	−20%	cmplt	565	534	31	−5%
cmovle	5	5	0	0%	rdteb	604	597	7	−1%
insqh	6	6	0	0%	extwl	660	345	315	−48%
cmovgt	13	13	0	0%	stq_u	688	688	0	0%
callsys	18	14	−4	−22%	blt	784	771	13	−2%
mulq	13	13	0	0%	bic	771	347	424	−55%
s8subq	17	17	0	0%	extll	789	761	28	−4%
cmovlt	16	16	0	0%	extlh	789	761	28	−4%
ldt	25	25	0	0%	bge	828	819	9	−1%
zap	34	33	−1	−3%	mb	961	941	20	−2%
umulh	35	35	0	0%	sll	949	590	359	−38%
mull	60	62	+2	+3%	subl	1052	1061	(9)	+1%
ornot	52	52	0	0%	br	1160	1080	80	−7%
cmpeq	64	61	−3	−5%	sra	1211	562	649	−54%
insql	61	61	0	0%	bsr	1203	1191	12	−1%
blbs	69	69	0	0%	s4addl	1176	1166	10	−1%
s8addl	71	74	+3	+4%	ret	1282	1264	18	−1%
mskwl	74	0	−74	−100%	zapnot	1262	910	352	−28%
jsr	98	89	−9	−9%	addq	1704	1685	19	−1%
cpys	104	41	−63	−61%	subq	2159	2140	19	−1%
mskqh	155	153	−2	−1%	ldah	2793	2746	47	−2%
cmovne	147	141	−6	−4%	extbl	2902	1668	1234	−43%
mskbl	163	0	−163	−100%	xor	3426	3380	46	−1%
cmoveq	183	173	−10	−5%	and	3402	2969	433	−13%
insbl	182	19	−163	−90%	bne	4537	4440	97	−2%
extwh	196	196	0	0%	addl	4897	4855	42	−1%
trapb	203	215	+12	+6%	ldq_u	5046	3933	1113	−22%
mskql	204	202	−2	−1%	stl	5753	5301	452	−8%
jmp	208	200	−8	−4%	lda	6496	5435	1061	−16%
cmovge	291	287	−4	−1%	stq	6778	6713	65	−1%
blbc	249	249	0	0%	ldq	7018	6519	−499	+7%
bgt	331	328	−3	−1%	beq	7607	7455	152	−2%
ldl_l	344	335	−9	−3%	bis	11284	10707	577	−5%
stl_c	344	335	−9	−3%	ldl	15962	14260	1702	−11%
extql	329	327	−2	−1%	**Totals**	**115895**	**107854**	**8042**	**−7%**

For the scaled transaction, a decrease in 58 out of 81 instructions types occurred. Of the remaining 25 instructions, 21 had no change and only 4 instructions, mull, s8addl, trapb, and subl, showed an increase. For cached transactions, 22 instruction counts decreased, 29 increased, and 22 remained unchanged.

The performance gain of 3.5 percent measured for the cached version of the application benchmark correlates closely to the decrease in the number of instructions per transaction measured in Table 13. If this correlation holds true, we would expect to see an increase in performance of approximately 7 percent for scaled transactions runs.

Dynamic Instruction Distribution

The performance of the Alpha microprocessor using technical and commercial workloads has been evaluated.[1] The commercial workload used was debit-

Table 12
Instruction Count and Variations for Cached Transaction

Instruction	Original	Byte/Word	Delta	% Delta
stb	0	174	+174	N/A
stw	0	217	+217	N/A
ldwu	0	1189	+1189	N/A
ldbu	0	1333	+1333	N/A
cmpbge	2	0	−2	−100%
cmovlbs	2	2	0	0%
addt	3	3	0	0%
cmovlbc	4	5	+1	+25%
cmovle	5	5	0	0%
insqh	6	6	0	0%
cmovgt	13	13	0	0%
callsys	15	16	+1	+7%
mulq	13	13	0	0%
s8subq	13	14	+1	+8%
cmovlt	16	16	0	0%
ldt	25	25	0	0%
zap	26	27	+1	+4%
umulh	32	32	0	0%
mull	46	48	+2	+4%
ornot	46	46	0	0%
cmpeq	53	53	0	0%
insql	61	61	0	0%
blbs	63	63	0	0%
s8addl	69	70	+1	+1%
mskwl	73	0	−73	−100%
jsr	90	92	+2	+2%
cpys	87	41	−46	−53%
mskqh	152	157	+5	+3%
cmovne	160	165	+5	+3%
mskbl	163	0	−163	−100%
cmoveq	182	190	+8	+4%
insbl	182	19	−163	−90%
extwh	195	196	+1	+1%
trapb	210	211	+1	0%
mskql	201	203	+2	+1%
jmp	209	215	+6	+3%
cmovge	226	236	+10	+4%
blbc	238	238	0	0%
bgt	292	302	+10	+3%
ldl_l	314	320	+6	+2%
stl_c	314	320	+6	+2%
extql	326	329	+3	+1%
stt	334	334	0	0%
cmple	367	374	+7	+2%
inswl	381	203	−178	−47%
srl	433	383	−50	−12%
extqh	434	314	−120	−28%
cmpule	450	440	−10	−2%
cmpult	550	572	+22	+4%
cmplt	561	585	+24	+4%
rdteb	587	590	+3	+1%
extwl	654	340	−314	−48%
stq_u	689	687	−2	0%
blt	751	770	+19	+3%
bic	759	346	−413	−54%
extll	784	805	+21	+3%
extlh	784	805	+21	+3%
bge	813	831	+18	+2%
mb	883	901	+18	+2%
sll	899	569	−330	−37%
subl	983	995	+12	+1%
br	1130	1100	−30	−3%
sra	1134	528	−606	−53%
bsr	1158	1165	+7	+1%
s4addl	1160	1170	+10	+1%
ret	1232	1239	+7	+1%
zapnot	1247	911	−336	−27%
addq	1589	1631	+42	+3%
subq	1994	2046	+52	+3%
ldah	2684	2691	+7	+0%
extbl	2921	1682	−1239	−42%
xor	3278	3332	+54	+2%
and	3361	2990	−371	−11%
bne	4328	4376	+48	+1%
addl	4734	4856	+122	+3%
ldq_u	5061	4046	−1015	−20%
stl	5418	5052	−366	−7%
lda	6289	5344	−945	−15%
stq	6464	6588	+124	+2%
ldq	6685	6359	−326	−5%
beq	7355	7466	+111	+2%
bis	10890	10668	−222	−2%
ldl	14964	13772	−1192	−8%
Totals	**111288**	**106521**	**−4767**	**−4%**

credit, which is similar to the TPC-A benchmark. The TPC-B benchmark is similar to the TPC-A, differing only in its method of execution. Cvetanovic and Bhandarkar presented an instruction distribution matrix for the debit-credit workload. The Alpha instruction type mix is dominated by the integer class, followed by other, load, branch, and store instructions, in descending order.[17] We took a similar approach but divided the instructions into more groups to achieve a finer detailed distribution. Table 13 gives the instruction makeup of each group. Figure 4 shows the percentage of instructions in each group for the four alternatives we studied. In all four cases, INTEGER LOADs make up 32 percent of the instructions executed. In the scaled Byte/Word category, the new ldbu and ldwu instructions compose 1 percent of the integer instructions, and the new stb and stw instructions accounted for 18 percent of the integer store instructions executed.

Table 13
Instruction Groupings

Instruction Group	Group Members
Integer loads	ldwu, ldbu, ldl_l, ldah, ldq_u, lda, ldq, ldl
Integer stores	stb, stw, stl_c, stq_u, stl, stq
Integer control	blbs, jsr, jmp, blbc, bgt, blt, bge, br, bsr, ret, bne, beg
Integer arithmetic	cmpbge, s8subq, umulh, mull, cmpeq, s8addl, cmple, cmpule, cmpult, cmplt, subl, s4addl, addq, subq, addl
Logical shift	cmovlbs, cmovlbc, cmovle, cmovgt, cmovlt, ornot, cmovne, cmoveq, cmovge, srl, bic, sll, sra, xor, and, bis
Byte manipulation	insll, inslh, mskll, mskhl, insqh, zap, insql, mskwl, mskqh, mskbl, insbl, extwh, insbl, extwh, mskql, extql, inswl, extqh, extwl, extll, extlh, zapnot, extbl
Other	addt, ldt, stt, mulq, callsys, cpys, trapb, rdteb, mb

During the scaled transactions, each instruction group showed a decrease in the number of instructions executed, ranging from negligible to as much as 54 percent. In addition, the number of byte manipulation and logical shift instructions decreased, because the method of loading or storing bytes and words on the original Alpha Architecture made heavy use of these types of instructions.

In our last examination, we looked at the instruction variation between a scaled and a cached transaction. The major difference between the two transactions is the additional I/O required by the scaled version of the benchmark. Table 14 gives the results. The Original version of the SQL Server database executed an extra 4,596 instructions during the cached transaction as compared to the scaled transaction. For the Byte/Word version, only an additional 1,334 instructions were executed.

Conclusions

The introduction of the new single byte and word manipulation instructions in the Alpha Architecture improved the performance of the Microsoft SQL Server database. We observed a decrease in the number of instructions executed per transaction, the elimination of some instructions in the workload, a redistribution of the instruction mix, and an increase in relative performance. The results are in line with expectations when the addition of the new instructions was proposed.

We limited our investigation to a single commercial workload and operating system. Testing a workload with more I/O, such as the TPC-C benchmark, would

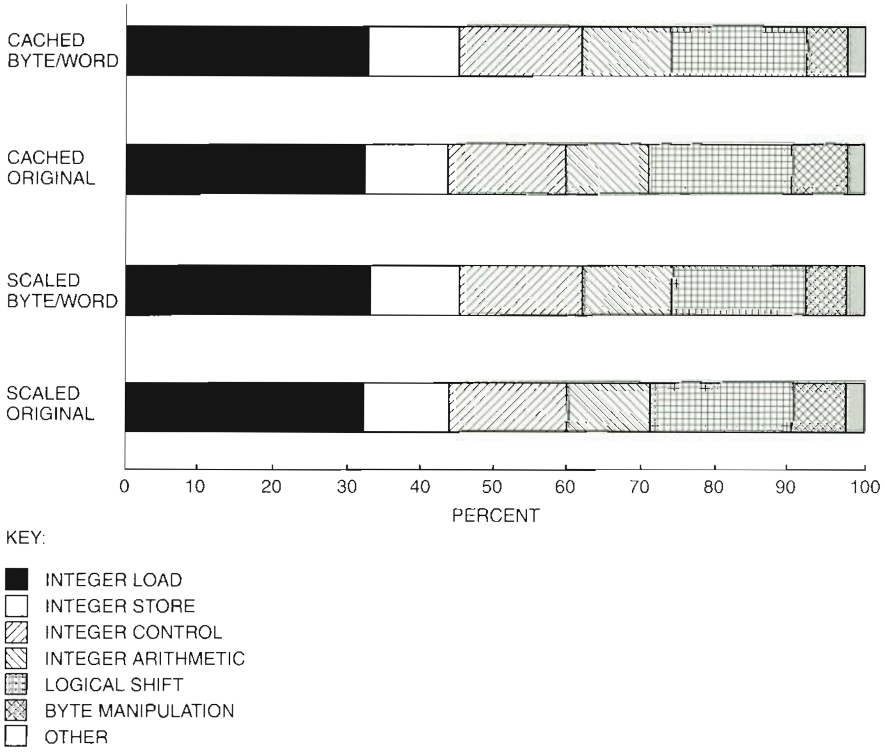


Figure 4
Instruction Group Distribution

Table 14
Instruction Variations (Scaled Minus Cached Transactions)

Instruction	Original	Byte/Word	Instruction	Original	Byte/Word	Instruction	Original	Byte/Word
stw	0	−2	cmplt	−4	+51	subl	−69	−66
ldwu	0	−26	rdteb	−17	−7	br	−30	+20
ldbu	0	+117	extwl	−6	−5	sra	−77	−34
cmovlbc	−1	+1	stq_u	+1	−1	bsr	−45	−26
callsys	−3	+2	blt	−33	−1	s4addl	−16	+4
s8subq	−4	−3	bic	−12	−1	ret	−50	−25
zap	−8	−6	extll	−5	+44	zapnot	−15	+1
umulh	−3	−3	extlh	−5	+44	addq	−115	−54
mull	−14	−14	bge	−15	+12	subq	−165	−94
ornot	−6	−6	mb	−78	−40	ldah	−109	−55
cmpeq	−11	−8	sll	−50	−21	extbl	+19	+14
blbs	−6	−6	cmovge	−65	−51	xor	−148	−48
s8addl	−2	−4	blbc	−11	−11	and	−41	+21
mskwl	−1	0	bgt	−39	−26	bne	−209	−64
jsr	−8	+3	ldl_l	−30	−15	addl	−163	+1
cpys	−17	0	stl_c	−30	−15	ldq_u	+15	+113
mskqh	−3	+4	extql	−3	+2	stl	−335	−249
cmovne	+13	+24	cmple	−1	+16	lda	−207	−91
cmoveq	−1	+17	inswl	−9	−4	stq	−314	−125
extwh	−1	0	srl	−24	−15	ldq	−333	−160
trapb	+7	−4	extqh	−7	−3	beq	−252	+11
mskql	−3	+1	cmpule	−18	−10	bis	−394	−39
jmp	+1	+15	cmpult	−13	+54	ldl	−998	−488
						Totals	**−4596**	**−1334**

produce a different set of results and would merit investigation. The use of another database, such as the Oracle RDBMS, which makes greater use of byte operations, would possibly result in an even greater performance impact. Lastly, rebuilding the entire operating system to use the new instructions would make an interesting and worthwhile study.

Acknowledgments

As with any project, many people were instrumental in this effort. Wim Colgate, Miche Baker-Harvey, and Steve Jenness gave us numerous insights into the Windows NT operating system. Tom Van Baak provided several analysis and tracing/simulation tools for the Windows NT environment. Rich Grove provided access to early builds of the GEM compiler back end that contained byte and word support. Stan Gazaway built the SQL Server application with the modifications. Vehbi Tasar provided encouragement and sanity checking. John Shakshober lent insight into the world of TPC. Peter Bannon provided the early prototype machine. Contributors from Microsoft Corporation included Todd Ragland, who helped rebuild the SQL Server; Rick Vicik, who provided detailed insights into the operation of the SQL Server; and Damien Lindauer, who helped set up and run the TPC benchmark. Finally, we thank Dick Sites for encouraging us to undertake this effort.

References and Notes

1. Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads," *21st Annual International Symposium on Computer Architecture*, Chicago (1994).

2. W. Kohler et al., "Performance Evaluation of Transaction Processing," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991): 45–57.

3. S. Leutenegger and D. Dias, "A Modeling Study of the TPC-C Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD Record 22 (2), (June 1993).

4. R. Sites and E. Perl, *PatchWrks—A Dynamic Execution Tracing Tool* (Palo Alto, Calif.: Digital Equipment Corporation, Systems Research Center, 1995).

5. W. Kohler, A. Shah, and F. Raab, *Overview of TPC Benchmark C: The Order-Entry Benchmark* (San Jose, Calif.: Transaction Processing Performance Council Technical Report, 1991).

6. R. Sites, "Alpha AXP Architecture," *Digital Technical Journal*, vol. 4, no. 4 (Special Issue 1992): 19–34.

7. *Alpha AXP Systems Handbook* (Maynard, Mass.: Digital Equipment Corporation, 1993).

8. *DECchip 21064A-233, -275 Alpha AXP Microprocessor Data Sheet* (Maynard, Mass.: Digital Equipment Corporation, 1994).

9. *Alpha 21164 Microprocessor Hardware Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1994).

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

11. G. Kane, *MIPS R2000 RISC Architecture* (Englewood Cliffs, N.J.: Prentice Hall, 1987).

12. J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, *MIPS: A VLSI Processor Architecture* (Stanford, Calif.: Computer Systems Laboratory, Stanford University, Technical Report No. 223, 1981).

13. J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill, and S. Przybylski, *Hardware/Software Tradeoffs for Increased Performance* (Stanford, Calif.: Computer Systems Laboratory, Stanford University, Technical Report No. 228, 1983).

14. The original MIPS Architecture at Stanford University did not contain single byte manipulation instructions; this decision was reversed for the first commercially produced MIPS R2000 processor. The Intel x86 Architecture has always included these instructions.

15. C. Cole and L. Crudele, personal correspondence, December 1996.

16. Microsoft Corporation developed the ARC firmware for the MIPS platform. During the early days of the port of Windows NT to Alpha, DIGITAL's engineers ported the ARC firmware to the Alpha platform.

17. The Alpha instruction type mix included PALcode calls, barriers, and other implementation-specific PALcode instructions.

Biographies

David P. Hunter
David Hunter is the engineering manager of the DIGITAL Software Partners Engineering Advanced Development Group, where he has been involved in performance investigations of databases and their interactions with UNIX and Windows NT. Prior to this work, he held positions in the Alpha Migration Organization, the ISV Porting Group, and the Government Group's Technical Program Management Office. He joined DIGITAL in the Laboratory Data Products Group in 1983, where he developed the VAXlab User Management System. He was the project leader of the advanced development project, ITS, an executive information system, for which he designed hardware and software components. David has two patent applications pending in the area of software engineering. He holds a degree in electrical and computer engineering from Northeastern University.

Eric B. Betts
Eric Betts is a principal software engineer in the DIGITAL Software Partners Engineering Group, where he has been involved with performance engineering, project management, and benchmarking for the Microsoft SQL Server and Windows NT products. Previously with the Federal Government Region, Eric was a member of the technical support group and a technical lead on several government programs. Before joining DIGITAL in 1990, he worked in many different software development areas at Martin Marietta and the Defense Information Systems Agency. Eric received a B.S. in computer science from North Carolina Central University.

Further Readings

The *Digital Technical Journal* is a refereed, quarterly publication of papers that explore the foundations of DIGITAL's products and technologies. *Journal* content is selected by the Journal Advisory Board, and papers are written by DIGITAL's engineers and engineering partners. Engineers who would like to contribute a paper to the *Journal* should contact the managing editor, Jane Blake, at Jane.Blake@ljo.dec.com.

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

Internet Protocol V.6/Preservation of Historical Computer Systems/Fortran for Parallel Computing/ Server Performance Evaluation and Optimization/ Internet Collaboration Software
Vol. 8, No. 3, 1996, EC-N7285-18

Spiralog Log-structured File System/ OpenVMS for 64-bit Addressable Virtual Memory/ High-performance Message Passing for Clusters/ Speech Recognition Software
Vol. 8, No. 2, 1996, EY-N6992-18

Digital UNIX Clusters/Object Modification Tools/ eXcursion for Windows Operating Systems/ Network Directory Services
Vol. 8, No. 1, 1996, EY-U025E-TJ

Audio and Video Technologies/ UNIX Available Servers/Real-time Debugging Tools
Vol. 7, No. 4, 1995, EY-U002E-TJ

High Performance Fortran in Parallel Environments/ Sequoia 2000 Research
Vol. 7, No. 3, 1995, EY-T838E-TJ
(Available only on the Internet)

Graphical Software Development/Systems Engineering
Vol. 7, No. 2, 1995, EY-U001E-TJ

Database Integration/Alpha Servers & Workstations/ Alpha 21164 CPU
Vol. 7, No. 1, 1995, EY-T135E-TJ
(Available only on the Internet)

RAID Array Controllers/Workflow Models/ PC LAN and System Management Tools
Vol. 6, No. 4, Fall 1994, EY-T118E-TJ

AlphaServer Multiprocessing Systems/ DEC OSF/1 Symmetric Multiprocessing/ Scientific Computing Optimization for Alpha
Vol. 6, No. 3, Summer 1994, EY-S799E-TJ

Alpha AXP Partners—Cray, Raytheon, Kubota/ DECchip 21071/21072 PCI Chip Sets/ DLT2000 Tape Drive
Vol. 6, No. 2, Spring 1994, EY-F947E-TJ

High-performance Networking/OpenVMS AXP System Software/Alpha AXP PC Hardware
Vol. 6, No. 1, Winter 1994, EY-Q011E-TJ

Software Process and Quality
Vol. 5, No. 4, Fall 1993, EY-P920E-DP

Product Internationalization
Vol. 5, No. 3, Summer 1993, EY-P986E-DP

Multimedia/Application Control
Vol. 5, No. 2, Spring 1993, EY-P963E-DP

DECnet Open Networking
Vol. 5, No. 1, Winter 1993, EY-M770E-DP

Alpha AXP Architecture and Systems
Vol. 4, No. 4, Special Issue 1992, EY-J886E-DP

NVAX-microprocessor VAX Systems
Vol. 4, No. 3, Summer 1992, EY-J884E-DP

Semiconductor Technologies
Vol. 4, No. 2, Spring 1992, EY-L521E-DP

PATHWORKS: PC Integration Software
Vol. 4, No. 1, Winter 1992, EY-J825E-DP

Image Processing, Video Terminals, and Printer Technologies
Vol. 3, No. 4, Fall 1991, EY-H889E-DP

Availability in VAXcluster Systems/ Network Performance and Adapters
Vol. 3, No. 3, Summer 1991, EY-H890E-DP

Fiber Distributed Data Interface
Vol. 3, No. 2, Spring 1991, EY-H876E-DP

Transaction Processing, Databases, and Fault-tolerant Systems
Vol. 3, No. 1, Winter 1991, EY-F588E-DP

VAX 9000 Series
Vol. 2, No. 4, Fall 1990, EY-E762E-DP

DECwindows Program
Vol. 2, No. 3, Summer 1990, EY-E756E-DP

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990, EY-C197E-DP

Compound Document Architecture
Vol. 2, No. 1, Winter 1990, EY-C196E-DP

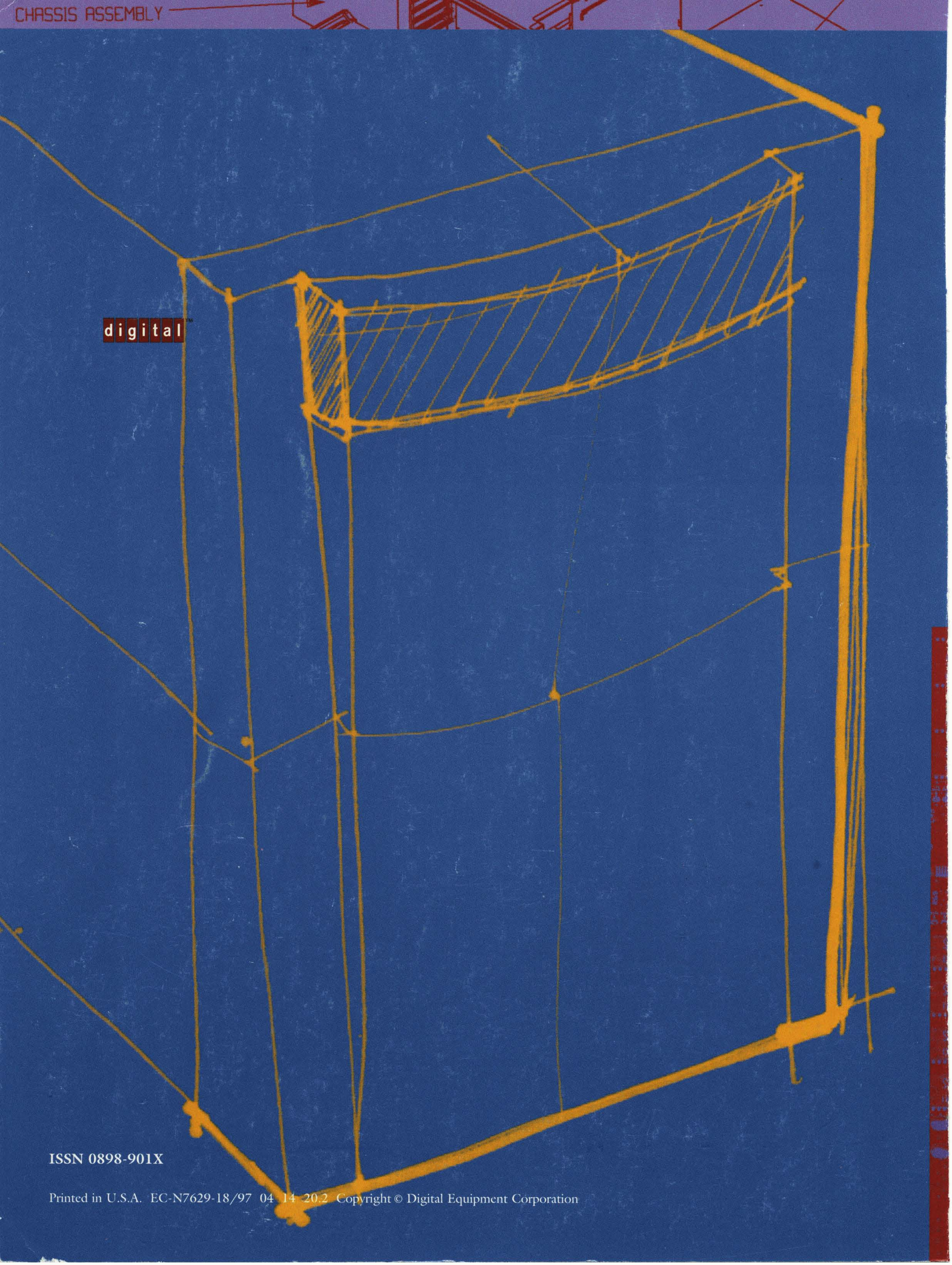
Call for Authors from Digital Press

Digital Press is an imprint of Butterworth-Heinemann, a major international publisher of professional books and a member of the Reed Elsevier group. Digital Press is the authorized publisher for Digital Equipment Corporation: The two companies are working in partnership to identify and publish new books under the Digital Press imprint and create opportunities for authors to publish their work.

Digital Press is committed to publishing high-quality books on a wide variety of subjects. We would like to hear from you if you are writing or thinking about writing a book.

Contact: Liz McCarthy, Associate Acquisitions Editor, or
Mike Cash, Digital Press Manager

DIGITAL PRESS
313 Washington Street
Newton, MA 02158-1626
U.S.A.
Tel: (617) 928-2649, Fax: (617) 928-2640
E-mail: Liz.McCarthy@repp.com or
Mike.Cash@BHein.rel.co.uk



ISSN 0898-901X