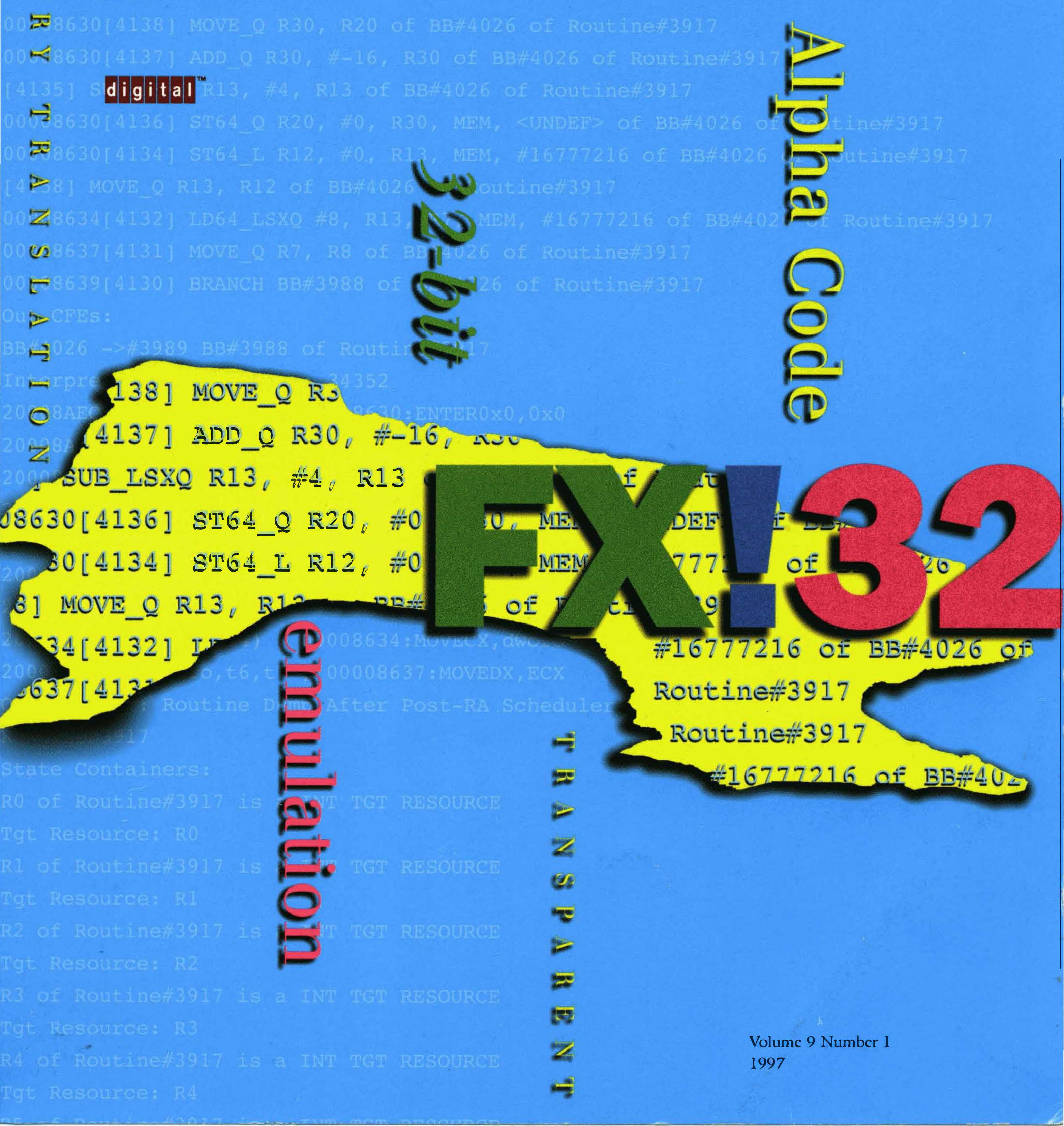
Digital Technical Journal

digital™



Volume 9 Number 1
1997

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

Circulation
Catherine M. Phillips, Administrator

Production
Christa W. Jessico, Production Editor
Elizabeth McGrail, Typographer
Peter R. Woodbury, Illustrator

Advisory Board
Samuel H. Fuller, Chairman
Donald Z. Harbert
Richard J. Hollingsworth
William A. Laing
Richard F. Lary
Alan G. Nemeth
Robert M. Supnik

Cover Design
The display of program code in the foreground and the background of our cover represents one of the unique aspects of the DIGITAL FX!32 software, the opening topic in this issue. By emulating an application in the foreground and later translating the execution profile into native Alpha code in the background, FX!32 enables 32-bit applications that run on Intel-based machines to also run on Alpha-based machines. The combination of emulation and binary translation provides Alpha users with additional applications and good performance with transparent operation.

The cover design is by Lucinda O'Neill of the DIGITAL Industrial and Graphic 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 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 can be ordered by sending the requested issue's volume and number and a check for $16.00 (non-U.S. $18) each to the published-by address. 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.

ISSN 0898-901X

Documentation Number EC-N7963-18

Book production was done by Quantic Communications, Inc.

Contents

Editor's Introduction

No matter how powerful the underlying hardware, most important to users is how that power translates to greater application performance and availability. Among the diverse topics in this issue of the *Journal* are innovative ways engineers have devised to meet application performance and availability requirements, and new tools for applications developers.

DIGITAL FX!32 is a unique software product that makes available hundreds of applications written for Intel machines to users of Alpha machines. Described by Ray Hookway and Mark Herdeg, FX!32 combines software emulation and advanced binary translation techniques to enable 32-bit applications that run on Intel-based machines with Windows NT to also run on 64-bit RISC Alpha-based machines with Windows NT. The design provides both the performance benefits and the transparency of operation that the project engineering team sought for users.

Also designed for the Windows environment is DIGITAL Visual Fortran, a tool for Fortran developers that combines technologies from DIGITAL and Microsoft Corporation. Leo Treggiari reviews the tool's components, which include the Component Object Model (COM), Fortran 90, and Microsoft Developer Studio. He addresses the question of why developers need help accessing dynamic link libraries and servers based on COM, and then focuses on the newly created tool that provides this functionality, the Fortran Module Wizard.

DIGITAL's shared-memory cluster interconnect, MEMORY CHANNEL 2, delivers the high levels of computational performance necessary to support the largest technical and commercial applications. Marco Fillo and Rick Gillett assess experiences with the first implementation of MEMORY CHANNEL that led to such enhancements as the cross-bar design in this latest implementation. They conclude with performance data that demonstrate unparalleled performance in terms of latency and bandwidth compared with traditional interconnects. MEMORY CHANNEL 2 provides latency of less than 2.2 microseconds and bandwidth of 1,000 megabytes per second in an 8-node cluster.

Data security has long been important to system managers but not easily achieved in distributed heterogeneous systems. DIGITAL and BEA Systems have integrated ObjectBroker middleware with the Distributed Computing Environment's Generic Security Service Application Programming Interface (GSS-API), as described here by John Parodi and Fred Burgher. The authors examine the choice of GSS-API for ObjectBroker and future directions in authentication software.

Design decisions made in the development of DIGITAL's StrongARM microprocessor were driven by the sometimes opposing requirements of high performance and low power consumption. Targeted for use in handheld appliances usually powered by conventional batteries, StrongARM offers significantly higher performance than comparable microprocessors: It operates at 160 MHz, dissipating less than 450 milliwatts. James Montanaro, Rich Witek et al. step through the decisions designers made to implement the ARM V4 instruction set from Advanced RISC Machines Ltd.

Upcoming in the next issue of the *Journal* are technical papers about new AltaVista software and a new Windows NT personal workstation based on an Alpha 64-bit RISC processor. To view the results of a recent survey sent to *Journal* Web subscribers, see http://www.digital.com/info/dtj.

Jane Blake

Jane C. Blake
Managing Editor

Raymond J. Hookway
Mark A. Herdeg

DIGITAL FX!32: Combining Emulation and Binary Translation

The DIGITAL FX!32 software product uniquely combines emulation and binary translation to enable any 32-bit application that executes on an Intel x86 microprocessor running the Windows NT 4.0 operating system to be installed and to execute on an Alpha microprocessor running Windows NT 4.0. Benchmark tests indicate that after translation, x86 applications run as fast on a 500-MHz Alpha system with DIGITAL FX!32 software installed as on a 200-MHz Pentium Pro system. The emulator and its associated run-time software provide transparent execution of applications written for x86-based platforms. The emulator produces profile data that is used by the translator and takes advantage of translation results as they become available. The translator provides native Alpha code for the portions of an x86 application that have previously been executed. A server manages the translation process for the user, making the process completely transparent.

Three factors contribute to the success of a microprocessor: price, performance, and software availability. The DIGITAL FX!32 product addresses the third factor, software availability, by making hundreds of new applications available on Alpha-based platforms running the Windows NT operating system. DIGITAL FX!32 software combines emulation and binary translation to provide fast, transparent execution of Intel x86 applications on Alpha systems.

Since its introduction in 1992, the Alpha microprocessor has been the fastest microprocessor available. A large number of native applications are available on Alpha systems, particularly those applications that require a high-performance processor. With the introduction of DIGITAL FX!32 software, 32-bit programs that can be installed and executed on x86 systems running the Windows NT 4.0 operating system can also be installed and executed on Alpha systems running Window NT 4.0. Except for having to specify that a program is an x86 application, installing and running an application is the same on an Alpha system as on an x86 system. The performance of an x86 application running on a high-end Alpha system is similar to the performance of the same application running on a high-end x86 system.

A number of systems have successfully used emulators to run applications on platforms for which the applications were not initially targeted.[1,2] The major drawback has been poor performance.[2] Several emulators have used dynamic translation, translating small segments of a program as it is executed, to achieve better performance than that obtained by an interpreter alone.[2-4] Dynamic translation involves a basic trade-off between the amount of time spent translating and the resulting benefit of the translation. If an emulator spends too much time on the translation and related processing, the executing program will be unresponsive. This limits the optimizations that can be performed by the emulator using dynamic translation.

FX!32 overcomes the performance problem by not doing any translation while the application is executing. Rather, FX!32 captures an execution profile that is later used by a binary translator[5] to translate into native Alpha code those parts of the application that have been executed. Since the translator runs in the background, it can use computationally intensive algorithms to improve the quality of the generated code. To our knowledge, FX!32 is the first system to explore this combination of emulation and binary translation.

In this paper, we describe how FX!32 works. We begin with an overview and discuss each of the major components in more detail. We then present some benchmark test results and briefly describe several limitations of the current version of DIGITAL FX!32 software.

Overview

On Alpha systems, the Windows NT operating system uses an emulator to run 16-bit x86 applications. These applications can be installed and run in the same way as they are installed and run on x86 systems, but the execution is slower. The emulator built into FX!32 provides a similar capability for 32-bit x86 applications.

Unlike the emulation software in the 16-bit environment, FX!32 provides a binary translator that translates 32-bit x86 applications into native Alpha code. The translation is done in the background and requires no user interaction. Using background translation allows the translator to perform optimizations that, in terms of computational resources, would be too expensive to accomplish while an application is running. An application translated by means of FX!32 runs up to 10 times faster than the same application running under the emulator.

DIGITAL FX!32 software consists of the following seven major components:

1. The transparency agent, which provides for transparent launching of 32-bit x86 applications.
2. The runtime, which loads x86 images and sets up the run-time environment to execute them. As part of loading an image, the runtime component jackets imported application programming interface (API) routines. Jackets are small code fragments that allow the x86 code to call Alpha Windows NT API routines.
3. The emulator, which runs an x86 application making use of translated code when it is available.
4. The translator, which produces a translated image using profile information received from the emulator.
5. The database, which stores execution profiles produced by the emulator and used by the translator. Translated images are also stored in the database, along with configuration information.
6. The server, which maintains the database and runs the translator as appropriate.
7. The manager, which allows the user to control resources used by the DIGITAL FX!32 software.

Figure 1 shows the relationships between these major components, each of which is discussed in more detail in the sections that follow.

The Transparency Agent

The transparency agent provides for transparent launching of 32-bit x86 applications. Launching an application on the Windows NT operating system always results in a call to the CreateProcess API routine. By hooking calls to this routine, the transparency agent can examine every image as it is about to be executed. If a call to CreateProcess specifies that an x86 image is to be executed, the transparency agent invokes the runtime component to execute the image.

FX!32 inserts the transparency agent into the address space of each process. A process that contains the trans-

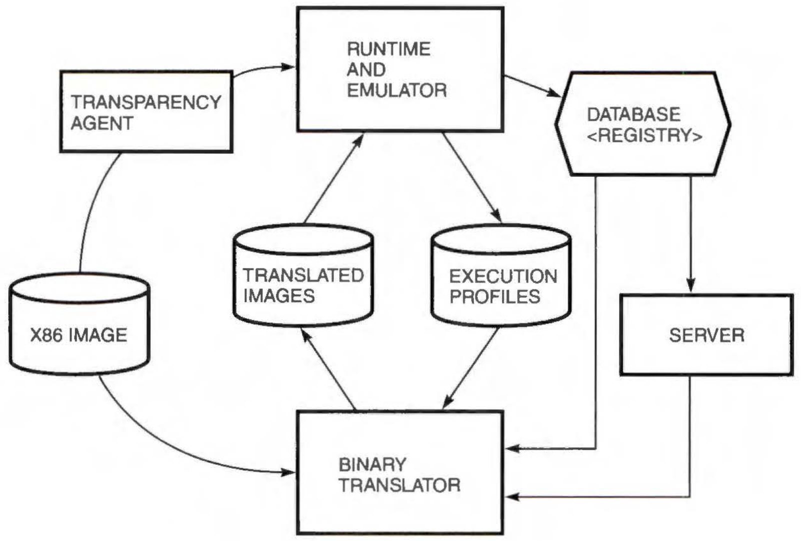


Figure 1
DIGITAL FX!32 System Components

parency agent is said to be enabled. Once a process is enabled, any attempt to execute an x86 image causes the runtime to be invoked to execute the process. The agent is propagated through the system because each attempt to create a process to run an Alpha image results in that created process being enabled.

By the time a user is logged on, FX!32 has enabled all the top-level processes, and any attempt to execute a 32-bit x86 application invokes the runtime component. The initial processes that are enabled are the Windows shell (explorer.exe), the service control manager (services.exe), and the remote procedure call server (rpcss.exe). When FX!32 is installed, the fx32strt.exe file is registered as the Windows shell. When a user logs on, fx32strt.exe runs and enables the real Windows shell, explorer.exe. The FX!32 server enables the service control manager when it starts, usually when the system is booted. Currently, any service process that is started by the service control manager before the server is started is not enabled. (The only exception is rpcss.exe, which is explicitly enabled by the server). We hope to alleviate this limitation in a future version of the DIGITAL FX!32 software.

Processes are enabled using a technique described by Jeffrey Richter in Chapter 16 of his book *Advanced Windows NT*[6] to inject a copy of the transparency agent into the process' address space.

The Runtime

The transparency agent invokes the runtime whenever an attempt is made to execute an x86 image. The runtime loads the image into memory, sets up the runtime environment required by the emulator, and then calls the emulator to execute the image.

The runtime replaces the Windows NT loader, which can only load Alpha images; the Windows NT loader returns an error reporting an image of the wrong architecture if it is invoked to load an x86 image. The runtime duplicates the functionality of the Windows NT loader, which includes relocating images that are not loaded at their preferred base address, setting up shared sections, and processing static thread local storage sections.

The runtime registers each image it processes with the Windows NT operating system by inserting pointers to that image into various lists that are used internally by the system. Maintaining these lists allows the native Windows NT code to correctly implement API routines, such as LoadResource and GetModuleHandle, which require access to images that have been loaded. The registration also ensures that the DllMain functions of the loaded dynamic link libraries (DLLs) are called as appropriate. (The entry points of x86 DLLs are jacketed by the runtime.)

Fortunately, the image lists that FX!32 must modify are in the user's address space, and no modification of the Windows NT operating system was required to register images with the system. Unfortunately, the structure of these lists is not part of the documented Win32 interface, and using them creates a dependency on the Windows NT version that is being run. FX!32 has dependencies on a number of undocumented features of the Windows NT operating system. Although the DIGITAL FX!32 product is more dependent on a particular version of the operating system than a typical layered application is, it is remarkable that the implementation of FX!32 did not require any changes to the Windows NT operating system.

The runtime also registers the image in the FX!32 database. This database maintains information about x86 images that have been loaded, including the application that loaded the image, profile data that was produced by the interpreter, and any translation of the image. The runtime accesses the database with a unique image identifier (ID), which the runtime obtains by hashing the image's header. Therefore, the image ID is determined by the content of the image, not by its location in the file system, and the information that FX!32 associates with the image can be accessed independently of the image's location on the disk. For example, if an application is installed in one directory and some of the images loaded by the application are subsequently translated by FX!32, the translated images will be located by FX!32 even if the application is later installed in a different directory.

When the runtime finds a translated image in the database, it loads this image along with the corresponding x86 image. Translated images are normal DLLs, loaded by the native LoadLibrary API routine. Translated images contain additional sections that store information required by the runtime to map x86 routines to the corresponding Alpha code.

The runtime duplicates the Windows NT loader function of binding an image's imports, using symbolic information in the image to locate the address of the imported routine or data. The runtime treats imports that refer to entries in Alpha images specially, however, by redirecting the imports to refer to the correct jacket entry in the FX!32 DLL, jacket.dll.

The jacket routines in jacket.dll enable an x86 user program to call the native Alpha implementation of the Win32 API. These jacket routines are extremely important because they allow x86 applications to use high-performance code that has been tuned to the Alpha platform. Some x86 applications run faster on the Alpha platform than on the x86 platform, even without being translated, because of the large amount of time the applications spend in native DLLs.

Each jacket contains an illegal x86 instruction that serves as a signal to the interpreter that a change is to be made to the Alpha environment. The interpreter calls an Alpha jacket routine at a fixed offset from the illegal x86 instruction. The basic operation of most jacket routines is to move arguments from the x86 stack to the appropriate Alpha registers, as dictated by the Alpha calling standard. Some jacket routines provide special semantics for the native routine being called, as required by FX!32. For example, the jacket for the GetSystemDirectory routine returns the path to the FX!32 directory rather than the path to the true system directory so that x86 applications do not overwrite native Alpha DLLs.

For an x86 application to run under FX!32, every image it loads must be either an x86 image or an Alpha image for which jackets exist. Therefore, FX!32 provides jackets for all the DLLs that implement the Win32 interface and for many redistributable DLLs. FX!32 currently provides jackets for more than 50 native Alpha DLLs, which has enabled the FX!32 development team to run almost all the commercial applications tested. Each new release of DIGITAL FX!32 software provides additional jackets, and the developers intend to jacket new interfaces as they are released.

The Emulator

The fundamental job of the emulator is to run x86 applications before they are translated. The first time an x86 image executes under FX!32, the image is executed by the emulator.

The emulator also serves as a backup for translated code. Because it is not possible to statically determine all the code that can ever be executed by an application (especially for applications that generate code on-the-fly), the emulator is always present to execute such untranslated x86 application code. Previous binary translators built by DIGITAL also depended on the presence of an emulator in this role.[5] Emulator performance is more of an issue for FX!32 because, unlike those earlier binary translators, all application code is interpreted when the x86 application is first run.

The emulator is an Alpha assembly language program that interprets the subset of x86 instructions that can be executed by a Win32 application. While an x86 application is running, the x86 processor state is kept partially in Alpha registers and partially in a per-thread data structure called the CONTEXT. The x86 integer registers are permanently mapped to Alpha registers, and Alpha registers store the state of the x86 condition codes. While the emulator is running, a dedicated Alpha register points to the CONTEXT. The CONTEXT stores the x86 per-thread processor context and any part of the x86 processor state that must be maintained across calls to other parts of the system, for example, calls to Alpha API routines.

Pipelined Dispatch

The structure of the emulator is a classic fetch-and-evaluate loop. The emulator dispatches on the first two bytes of each instruction, performing the lookup in a table of 64K entries. Each entry contains the address of the routine to execute to interpret an instruction and the length of the instruction.

The structure of the dispatch loop has been carefully crafted to make efficient use of 64-bit Alpha registers and to efficiently schedule the execution of code in the loop. Software pipelining is used to overlap the fetch and dispatch table lookup for the next instruction with the execution of the current instruction. At the top of the loop, at least eight bytes, starting at the address of the current instruction, are in Alpha registers. Length information from the dispatch table determines the first two bytes of the next instruction, allowing the dispatch table lookup to be overlapped with the execution of the current instruction. A fetch of additional bytes from the instruction stream is also initiated. Finally, the loop dispatches to the routine whose address was obtained from the table on the previous iteration of the loop.

The individual routines have been factored by using subroutines and coroutines to perform operations like operand fetching, making them as small as possible. As a result, the emulator code required to execute the most frequently executed x86 instructions fits in the first-level cache.

Condition Code Evaluation

Condition codes are generated by the execution of many of the x86 instructions. We have observed that condition codes are frequently set and relatively infrequently examined. The emulator takes advantage of this by evaluating the condition codes only when they are used, that is, by using a "lazy evaluation" technique. The execution of a typical instruction saves only enough state to allow the evaluation of condition codes, if required, at a later time. This takes much less effort than initially evaluating the condition codes. The additional advantage in deferring the evaluation is that only the condition codes that are used need to be generated. For example, the overflow condition code may never be computed if only the zero flag is used.

Floating-point Instruction Emulation

The 80-bit x86 floating-point registers are modeled by a stack of 64-bit memory locations that contain floating-point values. The decision to use 64-bit intermediate values, rather than to faithfully replicate the 80-bit model, was based on the need to achieve good performance when executing x86 floating-point code on the Alpha processor. This decision was supported by the fact that the Windows NT operating system also uses a 64-bit floating-point model. Although this is an approximation, our experience to date has shown that this was a good compromise. Very few applications rely on the full precision provided by the x86 floating-point unit's (FPU's) 80-bit registers.

The emulator also implements a somewhat simplified model of the x86 FPU's register file. Most instructions use the x86 FPU register file as a traditional operand stack; however, several instructions can create a register file state that is not strictly a stack by freeing registers in the middle of the stack, by moving the stack pointer without pushing or popping, or by initializing the register file in a way that breaks the stack model. Modeling the full complexity of the x86 FPU register file would be extremely expensive, and experience has shown that almost all programs use the register file strictly as a stack. The current version of the emulator takes advantage of this. We are investigating ways to model the floating-point registers in a way that maintains good performance but does not depend on their being treated as a stack.

Generation of Profiles

While it is interpreting an x86 program, the emulator generates profile data for use by the translator. The profile data includes the following information:

- Addresses that are the targets of call instructions
- (*Source address, target address*) pairs for indirect control transfers
- Addresses of instructions that make unaligned references to memory

The translator uses this information to generate *routines,* that is, units of translation that approximate a source code routine. The emulator generates profile data by inserting values in a hash table whenever a relevant instruction is interpreted. For example, as part of interpreting the call instruction, the emulator makes an entry in a hash table that records the target of the call. When an image is unloaded (either as a result of a call on the FreeLibrary routine or when the application exits), the runtime processes the hash table to produce a profile file for that image. This profile is processed by the server and can result in the server invoking the translator to create a new translation of the image.

To detect available translated code, the emulator uses the same hash table that it employs to gather the profile data. The x86 addresses for which there are translated routines and the address of the corresponding translated code are entered into the hash table by the runtime when it loads an x86 image that has been translated. When a call instruction is interpreted, the emulator looks up the target address. If a corresponding translated address exists, the emulator transfers control to that address.

The Translator

The server invokes the translator to translate x86 images for which a profile exists in the database. The translator uses the profile to produce a translated image. On subsequent executions of the image, the translated code is used, substantially speeding up the application.

Structure and Order of Operations

The translator has eight major components (or phases): the regionizer, build, the register mangler, the condition code mangler, improve, the code selector, the scheduler, and the assembler. (An additional phase that performs various peephole optimizations is disabled in the DIGITAL FX!32 V1.0 translator.) The major components function as follows:

1. The Regionizer—The regionizer uses data in the profile to divide the source image code into routines, which are described in the section Generation of Profiles. Each call target in the profile is used to generate an entry to a routine. The regionizer represents routines as a collection of regions. Each region is a range of contiguous addresses, which contains instructions that can be reached from the entry address of the routine. Unlike basic blocks, regions can have multiple entry points. The smallest collection of regions that contain all the instructions that can be reached from the routine entry is used to represent the routine. Many routines have a single region. This representation was chosen to efficiently describe the division of the source image into units of translation.

   The regionizer builds routines by following the control flow of the source image. When an indirect jump instruction is encountered while following the control flow, the possible targets of the instruction are obtained from the profile. Without this profile information, it would be very difficult to reliably identify these targets, and indirect jumps would have to be treated as returns from the routine. The profile information makes it possible to reliably generate a more complete representation of routines with correct control flow.

After the regionizer runs, each of the other major components is run in sequence for each routine.

2. Build—Build reparses the x86 instructions in the routine to create an internal representation (IR) of the routine for use by the subsequent components. The IR is a graph of basic blocks and is similar to the IR used by many optimizing compilers.
3. The Register Mangler—The initial IR is a straightforward representation of the source x86 code. This representation ignores the overlap of the x86 registers; the IR treats each occurrence of EAX, AX, AH, and AL as a separate register. The register mangler adds insert and extract operations as necessary to represent the actual semantics of the x86 registers.

4. The Condition Code Mangler—The effect of x86 instructions on condition codes is represented implicitly in the initial IR. The condition code mangler adds instructions to explicitly generate condition codes. Since the condition code mangler understands the control flow of the entire routine, it knows when condition codes are live and only adds code to generate condition codes when they are used later in the routine.
5. Improve—Improve performs several transformations that produce code more suited to the Alpha architecture. In the initial IR, each push and pop instruction is explicitly represented as a decrement/increment of the x86 stack pointer, accompanied by a store/load. Improve collects all the manipulation of the x86 stack pointer into a single decrement at the beginning of a basic block and a single increment at the end of that block. Improve also uses simple value numbering and analysis of memory references to try to eliminate loads and stores to both the x86 stack and the floating-point stack and to perform constant folding. Although Improve performs only relatively simple optimizations on a single basic block, we have found it to be quite effective in improving the quality of the code that is generated.
6. The Code Selector—The code selector transforms the IR from a representation that contains mostly x86 instructions to one that contains only Alpha instructions. This transformation is done instruction by instruction, with each x86 instruction being replaced by a sequence of Alpha instructions that produce the same effect. The implementation of the code selector is based on the TWIG code generator.[7] Although the code selector is capable of dealing with much more complicated patterns of instructions, this capability is not currently used.
7. The Scheduler—After the code selector is run, all the instructions in the IR are Alpha instructions. The scheduler reorders the instructions within a basic block to minimize the cycle count for the target processor.
8. The Assembler—The assembler builds the output translated image.

Use of Profile Data

The regionizer is the only component of the current translator that uses the control flow information in the profile. The regionizer uses the profile to determine which parts of the source image are translated. Future versions of the translater will use the profile to perform path-directed optimizations and to place code so as to reduce cache misses. Those changes will improve the performance of translated code.

Retranslation of an image is triggered by growth in the size of the profile. Because profile data is generated only when the emulator executes previously untranslated parts of the source image, an increase in the size of the profile indicates that new parts of the program have been executed. Retranslating with the new profile will cause these additional parts of the image to be translated.

Alignment Issues

On an Alpha system, references to memory locations that are not naturally aligned result in exceptions that are handled by the Windows NT kernel. Alignment exceptions can be avoided by using unaligned code sequences that use the LDQ_U and STQ_U instructions. Unaligned code sequences are slower than aligned sequences for accessing locations that are naturally aligned but much faster for accessing locations that are not naturally aligned. Native Alpha compilers always try to generate unaligned code sequences when referencing unaligned data to avoid the expense of dealing with alignment exceptions.

When generating the code for an instruction that references memory, the code selector must determine whether to use an aligned sequence or an unaligned sequence. To make the determination, the code selector needs to know the alignment of the address being referenced. In general, this cannot be determined by static analysis of the x86 code. To solve the problem, the code selector uses information in the profile about the alignment of memory addresses. The profile contains the address of every instruction that made an unaligned reference to memory. The code selector generates unaligned sequences for those instructions and aligned sequences for all other memory references. Although this code generation process is effective most of the time, some programs exhibit different memory reference behavior on successive runs. For those programs, alignment exceptions can still occur.

Shadow Stack

Translating return instructions presented particular problems for the translator. The translation of a call instruction saves the x86 return address on the x86 stack and then calls the translated code for the routine. After the translated call, the x86 return address is on the x86 stack and the corresponding native return address is in an Alpha register. This maintains the x86 stack in the expected x86 state. One way to translate a return instruction would be to use the x86 return address to look up a corresponding Alpha address; however, it is desirable to avoid the expense of a hash table lookup on every return. In the usual case, the return address is not changed by the routine and the translated code can pop the x86 stack and perform a native return by using the native return address. Two problems must be solved, though. First, some mechanism is needed to determine if the x86 return address has been modified. Second, a location is needed to save the native return address. Both problems are solved by using the shadow stack.

The shadow stack resides at the top of the native Alpha stack and is maintained by the translated code (with help from the emulator). A shadow stack frame is created for each call of a translated routine. When one translated routine calls another, the calling routine saves the x86 return address and the current x86 stack pointer in its shadow stack frame. The called routine then saves the native return address in the calling routine's shadow stack frame. On return, the called routine expects to find the x86 return address and the current x86 stack pointer in the calling routine's shadow stack frame. In this case, the called routine is returning to the environment that the calling routine expected and performs a native return. If the value of either the return address or the stack pointer has changed from the value expected by the calling routine, the called routine returns to the emulator.

In a similar manner, the emulator uses the information in the shadow stack to determine when it can return to translated code. A number of conditions can cause translated code to reenter the emulator. For example, the emulator is entered if the target of a translated indirect jump instruction is not known at translation time. Having the emulator return to translated code on a return instruction minimizes the amount of time that is spent in the emulator; however, the emulator can only return to the translated code if it knows that it has a valid return address. The shadow stack provides a mechanism to perform that validation.

The Database

The database consists of two parts. As described for the runtime, the first part of the database is a directory tree that contains profile files, translator log files, and translated images. The second part of the database is kept in the registry and consists of information about x86 applications and images that the DIGITAL FX!32 software has run on the system, together with configuration information. The configuration information includes the maximum amount of disk space that can be used by FX!32, the maximum number of images that can be stored in the database, the default translation options, the work list that the server uses to schedule translations, and the DatabaseDirectoryList. The DatabaseDirectoryList is a list of paths to additional databases that are to be searched for image profiles and translation results when the image is first executed. Directories on this list can be used to access information about the image from other machines on a network, making available to a user translations performed on another, perhaps more powerful, machine.

The Server

The server is a Windows NT service that normally starts whenever the system is rebooted. The server automatically runs the translator when appropriate, thus making the translation process completely transparent to the user. The server also maintains the database to control DIGITAL FX!32 resource usage.

The Manager

Usually the operation of DIGITAL FX!32 software is completely transparent to the user. Like any other program, though, FX!32 consumes system resources and a user must be able to control that resource usage. One of the roles of the manager is to provide a user interface to the configuration information kept in the database.

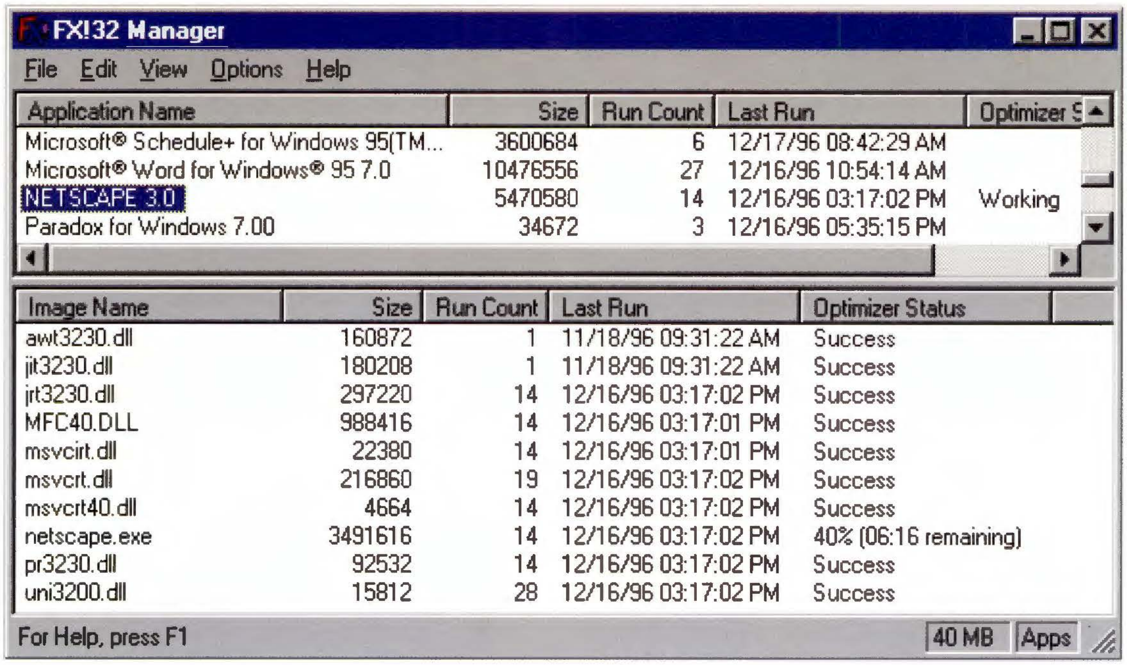
Figure 2 shows the manager window. The upper pane contains information about the various applications that have been run on the system: the total amount of disk space being used for profiles and translations of images loaded by the application, the number of times the application has been run, the date when it was last run, and the optimizer (translator) status. The lower pane contains information about the images that have been loaded by the highlighted application in the upper pane: the total amount of disk space used to store the profile and translation of the image, the number of times the image has been loaded, the date on which it was last loaded, and the status of the last translation of the image.

By interacting with the manager, the user can control various aspects of FX!32 operation, such as the maximum amount of disk space to use, which information to retain in the database, and when the translator should run.

Results

The DIGITAL FX!32 development team had two primary goals for the software: (1) to achieve transparent execution of 32-bit x86 applications and (2) to yield approximately the same performance as a high-end x86 platform when running applications on a high-performance Alpha system. The DIGITAL FX!32 product meets both goals.

Transparency is provided by the transparency agent and a run-time environment that can load and execute an x86 application without a translation step. Applications can be launched and executed on an Alpha system that is running FX!32 just as they can on an x86 system. We have performed extensive testing of more than 75 applications that run using FX!32, including major commercial applications such as Microsoft Office 95, Visual Basic 4.0, Photoshop 4.0, and CorelDRAW 6.0.



Application Name	Size	Run Count	Last Run	Optimizer S
Microsoft® Schedule+ for Windows 95(TM...	3600684	6	12/17/96 08:42:29 AM	
Microsoft® Word for Windows® 95 7.0	10476556	27	12/16/96 10:54:14 AM	
NETSCAPE 3.0	5470580	14	12/16/96 03:17:02 PM	Working
Paradox for Windows 7.00	34672	3	12/16/96 05:35:15 PM	

Image Name	Size	Run Count	Last Run	Optimizer Status
awt3230.dll	160872	1	11/18/96 09:31:22 AM	Success
jit3230.dll	180208	1	11/18/96 09:31:22 AM	Success
jrt3230.dll	297220	14	12/16/96 03:17:02 PM	Success
MFC40.DLL	988416	14	12/16/96 03:17:01 PM	Success
msvcirt.dll	22380	14	12/16/96 03:17:01 PM	Success
msvcrt.dll	216860	19	12/16/96 03:17:02 PM	Success
msvcrt40.dll	4664	14	12/16/96 03:17:02 PM	Success
netscape.exe	3491616	14	12/16/96 03:17:02 PM	40% (06:16 remaining)
pr3230.dll	92532	14	12/16/96 03:17:02 PM	Success
uni3200.dll	15812	28	12/16/96 03:17:02 PM	Success

For Help, press F1 | 40 MB | Apps

Figure 2
The DIGITAL FX!32 Manager

DIGITAL FX!32 software also met its performance goal. Figure 3 shows the relative performance on *BYTE Magazine*'s BYTEmark benchmark of a 200-megahertz (MHz) Pentium Pro system and a 500-MHz Alpha system running FX!32. For this benchmark, the Alpha system provides about the same performance as the 200-MHz Pentium Pro system. Figure 3 also shows that the Alpha native version of the benchmark runs twice as fast as the Pentium Pro version.

Of course, no single benchmark characterizes the performance of a system. Even so, when running translated x86 applications, we have consistently measured performance on a 500-MHz Alpha system to be in the range between that of a 200-MHz Pentium system and that of a 200-MHz Pentium Pro system. For

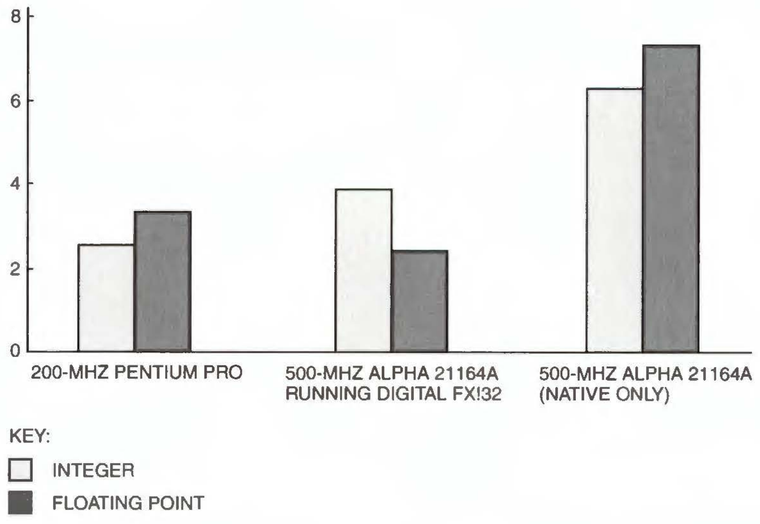


Figure 3
DIGITAL FX!32 Performance on the BYTE Benchmark)

some applications, performance can exceed that of a Pentium Pro system.

The initial version of the DIGITAL FX!32 software has some limitations. FX!32 executes only application code; it does not execute drivers. Consequently, native drivers are required for any peripheral that is installed on an Alpha system. Also, as described in the Transparency Agent section, FX!32 does not provide complete support for x86 services. Further, FX!32 does not support the Windows NT Debug API. Supporting that interface would require the capability to rematerialize the x86 state after every x86 instruction, thus severely limiting optimizations that the translator could perform. Optimizing compilers make a similar trade-off by restricting optimization when debugging information is required. Since FX!32 does not support the Debug interface, applications that require it do not run under FX!32. Those applications are mostly x86 development environments, and it probably makes more sense to run them on an x86 system. The limitations described are not serious, and most x86 applications that execute on an x86 processor that is running the Windows NT operating system also execute on an Alpha system running Windows NT and DIGITAL FX!32 software.

Summary

DIGITAL FX!32 software provides fast, transparent execution of 32-bit x86 applications on Alpha systems running the Windows NT operating system. This is accomplished using a unique combination of emulation and binary translation. The emulator runs an application, interprets the code, and generates profile information. For subsequent executions, the translator uses the profile data to produce translated images that contain optimized native Alpha code. An application translated by means of DIGITAL FX!32 software runs up to 10 times faster than the same application running under the emulator alone. Moreover, the translation takes place in the background and is therefore transparent to the user.

Acknowledgments

Building the DIGITAL FX!32 product required some extremely talented people to perform a lot of difficult work. The members of the DIGITAL FX!32 development team include Jim Campbell, Anton Chernoff, George Darcy, Tom Evans, Jim Givler, Charlie Greenman, Pippa Jollie, Mark Herdeg, Ray Hookway, Maurice Marks, Srinivasan Murari, Brian Nelson, Scott Robinson, Norm Rubin, Sherry Seskavich, Joyce Spencer, Tony Tye, and John Yates. Many of these individuals contributed the ideas described in this paper.

References

1. B. Case, "Rehosting Binary Code for Software Portability," Microprocessor Report (Sebastopol, Calif.: MicroDesign Resources, January 1989).

2. T. Halfhill, "Emulation: RISC's Secret Weapon," *BYTE Magazine* (April 1994).

3. R. Bedichek, "Some Efficient Architecture Simulation Techniques," *USENIX* (Winter 1990).

4. L. Deutsch and A. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages* (1983).

5. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal,* vol. 4, no. 4 (Maynard, Mass.: Digital Equipment Corporation, 1992).

6. J. Richter, *Advanced Windows NT,* chap. 16 (Redmond, Wash.: Microsoft Press, 1994).

7. A. Aho, M. Ganapathi, and S. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Transactions on Programming Languages and Systems,* vol. 11, no. 4 (October 1989).

Biographies

Raymond J. Hookway
Ray Hookway led the DIGITAL FX!32 development team and was a key contributor to the binary translation component of the DIGITAL FX!32 software product. He has been a member of the AMT group of DIGITAL Semiconductor since 1993. Ray joined DIGITAL in 1989 and has worked in the CAD and AD groups of DIGITAL Semiconductor, where he contributed to the first Alpha PC project. Prior to joining DIGITAL , he was Director of Engineering for Endot, Inc., where he developed one of the first VHDL simulation environments. He was also an Assistant Professor at Case Western Reserve University, where he did research on program verification, and he was a Visiting Professor at the University of Upsalla, Sweden. Ray received M.S. and Ph.D. degrees in computer science from Case Western Reserve University and a B.S. in engineering from Case Institute of Technology. He has applied for several patents related to his DIGITAL FX!32 work.

Mark A. Herdeg
Mark Herdeg has been with DIGITAL since 1985. He is currently a principal software engineer in the AMT group of DIGITAL Semiconductor. Previously, he worked on console software for the Nautilus (VAX 8500) and Argonaut projects. The Alpha simulator developed for the Argonaut project, MANNEQUIN, became the first Alpha system on which the OpenVMS operating system successfully booted. Mark contributed to a related project that used the Alpha simulator and a dual–architecture-aware debugger to allow development and execution of applications with a mix of VAX and Alpha code. A founding member of the Alpha Migration Tools group, Mark worked on its first product, VEST, the OpenVMS VAX–to–Alpha binary translator. He then helped design and develop the DIGITAL FX!32 software product, with particular focus on the runtime component. Currently, he is the project leader for the next release of DIGITAL FX!32 software. Mark has submitted several patent applications for work on the multiple-architecture execution environment and for the DIGITAL FX!32 design.

Leo P. Treggiari

Development of the Fortran Module Wizard within DIGITAL Visual Fortran

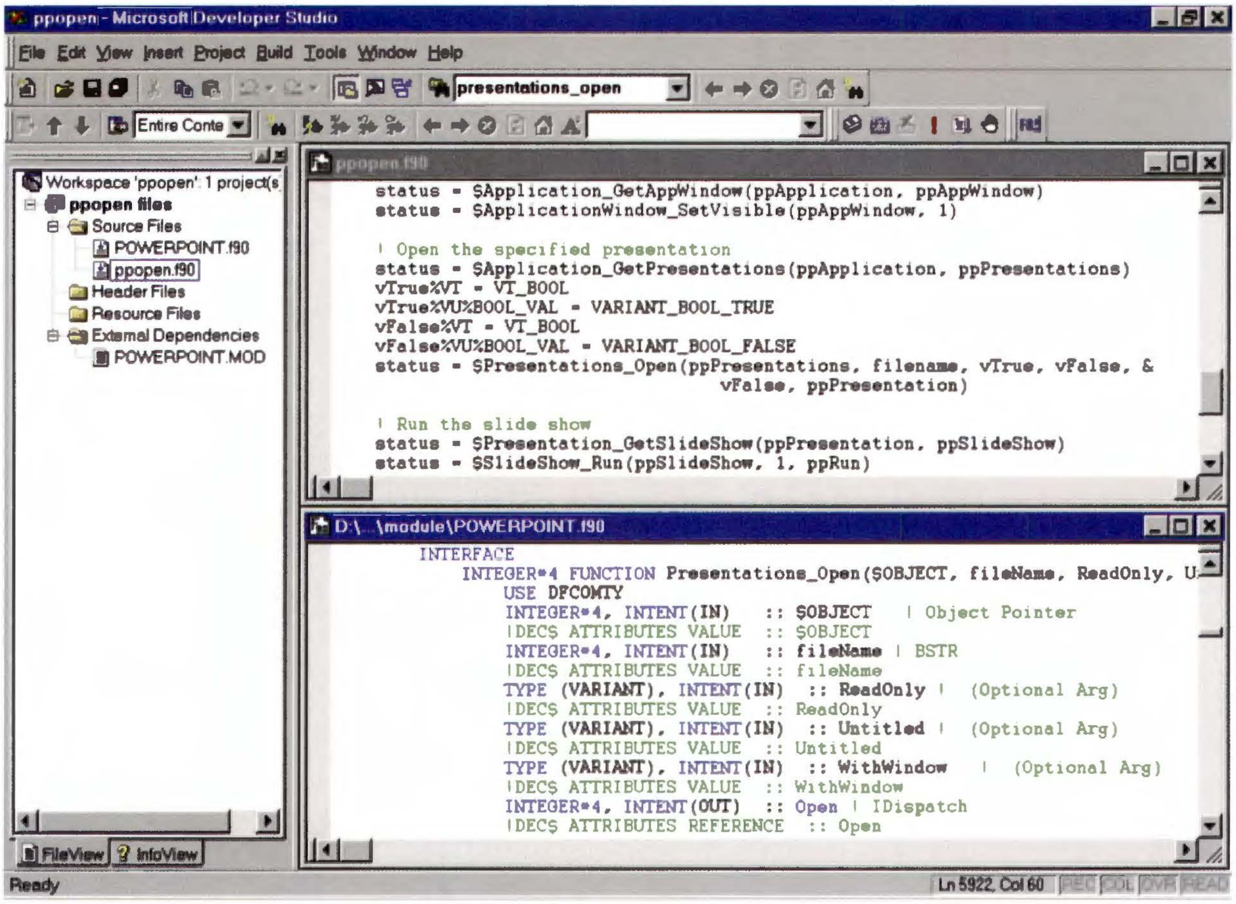
The Fortran Module Wizard is one of the tools in DIGITAL Visual Fortran, a DIGITAL product for the Fortran development environment. Visual Fortran consists of the DIGITAL Fortran 90 compiler and run-time libraries and the Microsoft Developer Studio. Together, these technologies provide a rich set of tools for the Fortran developer who is using the Windows NT and Windows 95 systems. The Fortran Module Wizard generates complete Fortran source code, allowing Fortran applications to invoke routines in a dynamic link library, methods of an Automation object, and member functions of a Component Object Model (COM) object.

DIGITAL Visual Fortran is an integrated development environment for Fortran applications.[1] It is supported on the Windows NT version 4.0 operating system on both Alpha and Intel hardware and on the Windows 95 system. DIGITAL Visual Fortran is a combination of technologies from DIGITAL and Microsoft Corporation. The DIGITAL-supplied compiler and run-time libraries support the DIGITAL Fortran 90 language.[2] DIGITAL Fortran 90 conforms to American National Standard Fortran 90 (ANSI X3.198-1992) and provides many extensions to the Fortran 90 standard. The Microsoft-supplied integrated development environment is the Microsoft Developer Studio, which is also used by Microsoft Visual C++, Microsoft Visual J++ (for Java), other Microsoft tools, and other companies' development tools. Developer Studio includes a text editor, resource editors, project build facilities, an incremental linker, a source code browser, an integrated debugger, and a profiler. The operation of all these tools is controlled from a single application. Figure 1 shows an example of Microsoft Developer Studio from which two Fortran source files are being edited. DIGITAL adds a number of Fortran-specific tools to the environment, one of which is the Fortran Module Wizard.

Design of the Fortran Module Wizard

DIGITAL designed the Fortran Module Wizard to help Fortran developers working in the application-rich Windows environment. The Fortran Module Wizard supports access to dynamic link libraries (DLLs) and servers based upon Microsoft's Component Object Model (COM). This support allows Fortran developers to use the popular mechanisms that make functionality (services) available to other software (clients).

Traditionally, Microsoft and others have provided system interfaces and reusable libraries of code as DLLs. A DLL is a file containing functions that can be called by programs and other DLLs. The role of DLLs on a Windows system is very similar to that of shareable images on the OpenVMS operating system and shared libraries on the UNIX system. Today, DLLs are still the primary mechanism for accessing system interfaces on Windows.



```
    status = $Application_GetAppWindow(ppApplication, ppAppWindow)
    status = $ApplicationWindow_SetVisible(ppAppWindow, 1)

    ! Open the specified presentation
    status = $Application_GetPresentations(ppApplication, ppPresentations)
    vTrue%VT = VT_BOOL
    vTrue%VU%BOOL_VAL = VARIANT_BOOL_TRUE
    vFalse%VT = VT_BOOL
    vFalse%VU%BOOL_VAL = VARIANT_BOOL_FALSE
    status = $Presentations_Open(ppPresentations, filename, vTrue, vFalse, &
                                 vFalse, ppPresentation)

    ! Run the slide show
    status = $Presentation_GetSlideShow(ppPresentation, ppSlideShow)
    status = $SlideShow_Run(ppSlideShow, 1, ppRun)
```

```
        INTERFACE
            INTEGER*4 FUNCTION Presentations_Open($OBJECT, fileName, ReadOnly, U
                USE DFCOMTY
                INTEGER*4, INTENT(IN)     :: $OBJECT   ! Object Pointer
                !DEC$ ATTRIBUTES VALUE    :: $OBJECT
                INTEGER*4, INTENT(IN)     :: fileName ! BSTR
                !DEC$ ATTRIBUTES VALUE    :: fileName
                TYPE (VARIANT), INTENT(IN)  :: ReadOnly !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE    :: ReadOnly
                TYPE (VARIANT), INTENT(IN)  :: Untitled !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE    :: Untitled
                TYPE (VARIANT), INTENT(IN)  :: WithWindow   !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE    :: WithWindow
                INTEGER*4, INTENT(OUT)    :: Open ! IDispatch
                !DEC$ ATTRIBUTES REFERENCE  :: Open
```

Figure 1
Microsoft Developer Studio, Two Fortran Source Files Being Edited

When Microsoft introduced OLE version 1, the name OLE was an acronym for object linking and embedding. OLE version 1 enabled compound documents by allowing a document to link to, or embed data from, another document. In 1993, Microsoft introduced COM as the base architecture of OLE version 2.[3] COM is an extensible architecture that provides mechanisms for creating and using software components. A software component consists of reusable pieces of code and data in binary form that can be plugged into other software components from other vendors with relatively little effort.[4] Like DLLs, COM allows a software developer to provide a set of services to multiple clients. In addition, COM has the advantage of allowing the services to reside in another process and on another machine. (Distributed COM [DCOM] allows objects to be created and used on remote machines.) COM also contains features that aid in the deployment and evolution of the services.[5] Microsoft has extended its languages and tools to aid software developers in the creation of clients and servers based upon COM (hereafter referred to as clients and servers in this paper).

Why does a Fortran developer need help accessing services in DLLs and servers? Calling code that is written in another programming language is, in general, difficult. There are complex issues around calling standards and data type representations. If a mistake is made in manually translating a function signature from one language into another, today's programming environments are of little help. The application can fail at a point in the code, for example in the routine prolog, which does little to suggest the cause of the problem. Often, solving these problems requires understanding the intricacies of calling standards and single stepping through assembly code. Calling the components in a server also requires understanding and properly using a number of COM programming interfaces.

The Fortran Module Wizard deals with the difficulties. It reads a description of a service, which the service provider created, and generates Fortran source code. This automatically generated code makes calling these services as easy as calling another Fortran function or subroutine.

Enabling Technologies

Components of COM, Fortran 90, and the Microsoft Developer Studio enable the functionality of the Fortran Module Wizard. This section gives an overview of these technologies.

COM Technologies

As mentioned earlier, COM provides mechanisms for creating reusable software components. This paper attempts to explain only those parts of COM, and some technologies based on COM, necessary for the reader to understand the use of server functionality from code generated by the Fortran Module Wizard. COM, OLE, and ActiveX, of course, contain many more mechanisms.[6] A number of the references listed at the end of this paper are good sources of further reading.[4–7] Much of the description of COM in the following section is taken from the Component Object Model Specification.[8]

COM Objects COM is an object-based programming model designed to promote software interoperability. In other words, COM allows two or more applications or components to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. COM defines a completely standardized mechanism for creating objects and for clients and objects to communicate. Unlike traditional object-oriented programming environments, these mechanisms are independent of the applications that use object services and of the programming languages used to create the objects. COM therefore defines a binary interoperability standard rather than a language-based interoperability standard on any given operating system and hardware platform.

To support its interoperability features, COM defines and implements mechanisms that allow components to connect to each other as objects. The definition of an object is a piece of software that contains the functions that represent what the object can do (its intelligence) and associated state information for those functions (data). In other words, an object is some data structure and some functions to manipulate that data. In this paper, we use the term object to mean an object instance, as opposed to an object class. An object class is similar to a derived-type in Fortran 90 or a structure in C. It specifies a blueprint for object instances that a server will create upon a client's request. An important principle of object-oriented programming is encapsulation, in which the exact implementation of those functions and the exact format and layout of the data is only of concern to the object itself. This information is hidden from the clients of an object and can therefore be changed without affecting the client.

With COM, components interact with each other and with the system through collections of function calls, also known as methods or member functions or requests, called interfaces. An interface is a semantically related set of member functions. The interface as a whole represents a feature of an object. The member functions of an interface represent the operations that make up the feature.

For a quick look at a simple example of a COM object, imagine a Calculator object that is willing to provide arithmetic services to any client. It could support an interface named ICalculate. By convention, the letter I always prefixes the name of an interface. The ICalculate interface could contain member functions named Add, Subtract, Multiply, Divide, etc. If a client wanted to use the services of the Calculator object, it would request COM to create an object of class Calculator and request the ICalculate interface. It could then call the member functions of the ICalculate interfaces (Add, Subtract, etc.).

With COM, a pointer to an object is actually a pointer to a particular interface that the object supports. All COM objects support the interface named Iunknown, which contains the member functions named AddRef, Release, and QueryInterface. All COM objects must implement these member functions. AddRef and Release implement object reference counting. Clients use them to tell an object when they are using it and when they are done. Objects delete themselves when they are no longer being used by any client. QueryInterface is the basis for a process called interface negotiation, whereby a client asks an object what services it is capable of providing. For example, if a client had a pointer to the Calculator object's IUnknown interface, it could get a pointer to its ICalculate interface by calling the IUnknown QueryInterface member function. In general, an object can support multiple interfaces and a client can use QueryInterface to get a pointer to any of them. Examples in which Fortran code calls member functions in interfaces are given in the section Fortran Module Wizard Functionality. Microsoft defines a number of useful interfaces. Object class creators are free to use existing interfaces and define their own.

Automation Objects One Microsoft-defined interface, IDispatch, is the basis for Automation.[9] Any object that supports this interface, also known as a dispinterface, is an Automation object, and can be accessed by any Automation client. An Automation object exposes methods and properties. Methods are functions that perform an action on an object and are similar to the member functions of COM objects. Properties hold information about the state of an object. A property can be represented by a pair of methods; one for getting the property's current value, and one for setting the property's value.

The capabilities of an Automation object are similar to those of a COM object. An Automation object is, in fact, a COM object; that is, it supports the IUnknown interface as well as the IDispatch interface. However, the mechanisms for using the services of the two are very different. Microsoft designed Automation based on the needs of scripting or macro languages (i.e., Visual Basic). It does not require understanding the intricacies of calling conventions as does COM. It supports mechanisms more suitable to the dynamic querying of an object's capabilities. This makes Automation more suited to late binding of objects, that is, invoking methods of a previously unknown object at run time.

An Automation client accesses all the methods and properties of an Automation object through a single member function of the IDispatch interface named Invoke. The client passes Invoke a number of arguments that identify

- The method, its arguments, and a place to receive the return value, or
- The property and its new value, or
- The property and a place to receive its current value

In fact, Invoke could be described as the Swiss army knife of Automation programming.

Most of the differences between Automation objects and COM objects are hidden by the Fortran interfaces that the Wizard generates.

Object Identification To enable the use of COM objects created by disparate groups of developers, there must be a method of uniquely identifying an object class regardless of its origin. COM uses globally unique identifiers (GUIDs) to do this. A GUID is a 16-byte integer value that is guaranteed (for all practical purposes) to be unique across space and time. COM uses GUIDs to identify object classes, interfaces, and other things that require unique identification. COM provides a routine named CoCreateGUID, and Microsoft provides a utility named GUIDGEN, that a developer uses to generate a GUID. Assigning a GUID to an object class or interface is the job of the creator of the class or interface. To create an instance of an object, the developer needs to tell COM the GUID of the object. Using 16-byte integers for identification is fine for computers, but it poses a challenge for the typical developer. COM supports the use of a less precise, textual name called a programmatic identifier (ProgID). A ProgID takes the form:

```
application_name.object_name.object_version
```

For example, the name of the Basic object of the Microsoft Word application is Word.Basic.1. Similarly, interfaces are usually discussed using their Ixxx name (for example, IUnknown), but their GUID uniquely identifies them. ProgIDs are not supplied for all objects. They are normally supplied only for Application objects. An Application object is a top-level object that becomes active when the application starts. It provides a starting point for clients to access all of an application's subordinate objects.

Type Information Type information contains descriptions of object classes, interfaces, DLLs, data structures, and so forth that are independent of any programming language. A developer accesses type information through an interface named ITypeInfo.[7] A client can get a pointer to type information from

- A running Automation object
- A running COM object that supports the IProvideClassInfo interface
- A type library

A type library is a collection of type information for any number of object classes, interfaces, etc. A developer can store a type library in a separate file (using a .TLB extension by convention), or as part of another file. For example, the type library that describes the type information for a DLL can be stored in the .DLL file itself. Since the type information is stored in a file, it is available regardless of whether or not the client has a pointer to the object(s) that the information describes.

The easiest way to create a type library is to write a script in the Microsoft Interface Definition Language (IDL). The Microsoft IDL compiler (MIDL) reads an IDL script and creates a .TLB file.[10] An IDL script is similar to a C++ header file with additional syntax for information required by COM. An example of such information is whether an argument to a member function is an input, an output, or an input/output argument.

To use the Fortran Module Wizard, the developer must know where to find type information for the functionality to be used. Some examples of this are given in the section Fortran Module Wizard Functionality.

Fortran 90

This section describes features of the DIGITAL Fortran 90 language that the Fortran Module Wizard uses in the code that it generates.

Modules Fortran 90 does not support objects, but it does provide a new form of program unit called a module. A Fortran module is a set of declarations that are grouped together under a global name and are made available to other program units by means of the Fortran USE statement. These modules have similarities to C include files but are more powerful.

The Fortran Module Wizard generates a source file containing one or more Fortran modules and places the following types of information in the modules:

- Derived-type definitions—Fortran equivalents of data structures that are found in the type information.

- Procedure interface definitions—Fortran interface blocks that describe the procedures found in the type information.
- Procedure definitions—Fortran functions and subroutines that are wrappers for the procedures found in the type information. The wrappers make the external procedures easier to call from Fortran by handling data conversion and low-level invocation details.

The use of modules allows the Fortran Module Wizard to encapsulate the data structures and procedures exposed by an object or DLL in a single place. These definitions can be shared in multiple Fortran programs.

Attributes The DIGITAL Fortran 90 language supports a number of calling convention attributes that allow Fortran programs to call programs written in other programming languages. Some attributes select the calling convention (STDCALL, C, VARYING). Others determine whether an argument is passed by value or by reference (VALUE, REFERENCE). Another attribute defines the external name of the procedure (ALIAS).

Pointer To Procedure The address of a COM member function is never known at program link time. The developer must get a pointer to an object's interface at run time, and the address of a particular member function is computed from that. We have extended the DIGITAL Fortran 90 language to support a Pointer To procedure.

Microsoft Developer Studio

Microsoft Developer Studio provides a number of methods that allow software developers to extend its environment.[11] This section describes these methods.

Tools Menu Developer Studio contains a Customize dialog box through which the developer can add utilities to the Tools menu and then run those utilities from within Developer Studio.

Gallery The Developer Studio Gallery provides a central repository for all reusable parts of projects. The reusable parts can range from something as simple as a bitmap to something as complex as a DLL.

Developer Studio Object Model Developer Studio provides a set of COM objects that give developers programmatic control of its functionality. Users can create commands that perform specific tasks and add them to a toolbar. The Developer Studio Object Model is programmed in three ways: (1) by creating macros in the Visual Basic Scripting Edition Language (VBScript); (2) by creating a Developer Studio DLL Add-in, which is a server implemented as a DLL; and (3) by creating a separate Automation client that connects to the Developer Studio objects.

Wizards A wizard is code that creates the starter files for a new application or adds a feature to an existing application. Wizards that add features are stored in the Developer Studio Gallery. Wizards that create starter files for a new application are called AppWizards. When the developer requests the creation of a new project, Developer Studio presents a list of the types of project that can be created (for example, a console application or a DLL). In addition, it lists the installed AppWizards that can generate complete applications. Often they contain options that allow the developer to choose the features of a generated application.

Microsoft Visual C++ provides a number of AppWizards; most of them can create typical C++ applications. In addition, to aid developers in extending Developer Studio, one AppWizard creates the starter files for a custom AppWizard, and another creates the starter files for a DLL Add-in. The Fortran Module Wizard is currently implemented as an application that runs from the Developer Studio Tools menu. In the future, it may be a Developer Studio AppWizard.

Fortran Module Wizard Functionality

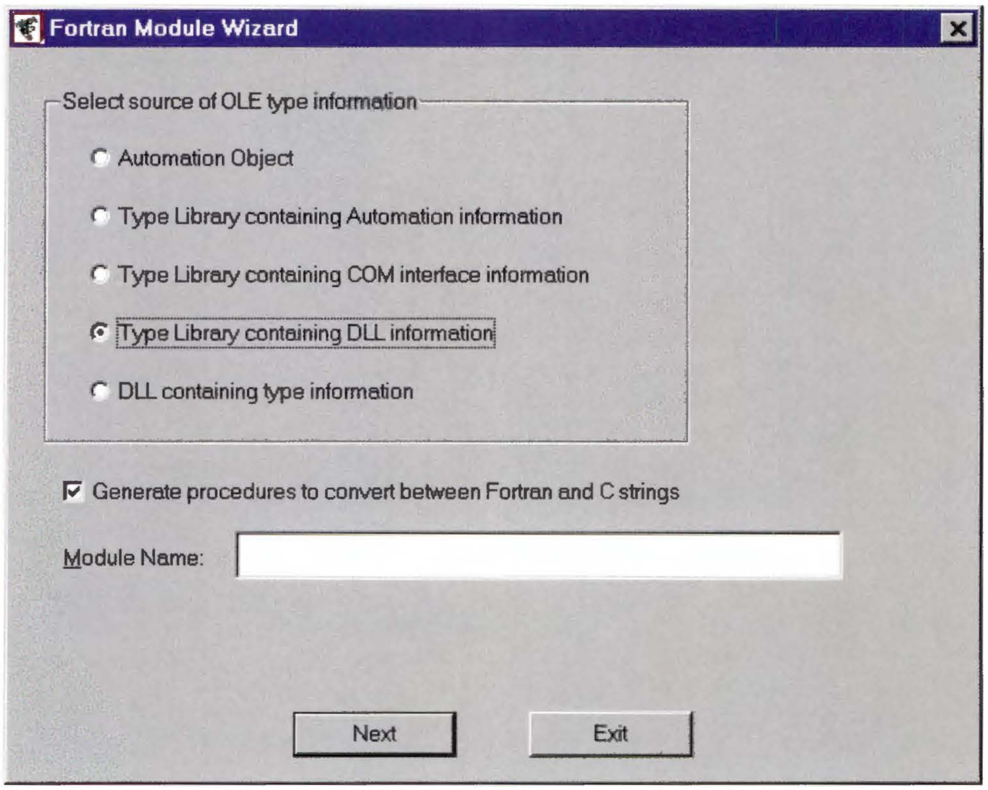
This section describes the user interface of the Fortran Module Wizard and presents some samples of the code generated by the Wizard. It also shows examples of calling the generated code from Fortran.

User Interface

Upon opening the Fortran Module Wizard from the Tools menu, the user is presented with a series of dialog boxes. From these, the user selects the type information for the functionality needed.

Figure 2 shows the first dialog box. It requests the user to choose the source of the type information that describes the required functionality. The developer must consult the documentation to determine what type of object (or DLL) the functionality is implemented as, and where to find its associated type information. The choices are the following:

- Automation object
- Type library containing automation information
- Type library containing COM interface information
- Type library containing DLL information
- DLL containing type information


Fortran Module Wizard

Select source of OLE type information

( ) Automation Object

( ) Type Library containing Automation information

( ) Type Library containing COM interface information

(●) Type Library containing DLL information

( ) DLL containing type information

[✓] Generate procedures to convert between Fortran and C strings

Module Name:

Figure 2
Fortran Module Wizard Dialog Box

Automation Object Microsoft recommends that servers provide a type library. Some applications, for example Microsoft Word version 7.0, do not, but they do provide type information dynamically when running. When this option is selected, Developer Studio displays the dialog box shown in Figure 3. The user then enters the name of the application, the name of the object, and optionally the version number. Note that this method works only for objects that provide a ProgID. ProgIDs are entered into the system registry and identify, among other things, the executable program that is the object's server.

After the user enters the information and presses the "Generate button," the Fortran Module Wizard asks COM to create an instance of the object identified by the ProgID that the Wizard constructs from the user-supplied information. COM starts the object's server if it needs to do so. The Wizard then asks the object for its type information and generates a file containing Fortran modules.

Other Options If the user chooses one of the remaining options, that is, any of the type libraries or the DLL (see Figure 2), Developer Studio displays the dialog box shown in Figure 4. From this dialog box, the user chooses the type library (or file containing the type library) and, optionally, the specific components of the type library.

At the top of the dialog box, a "combo box" lists all the type libraries that have been registered with the system. Their file names have a number of different file extensions, for example, .OLB (object libraries) and .OCX (ActiveX controls). The user either selects a type library from the list or presses the "Browse button" to find the file using the standard "Open dialog box." After selecting a type library, the user presses the "Show button" to list the interfaces described in the type library. By default, the Fortran Module Wizard uses all the interfaces; however, the developer can select the ones desired from the list.

After the user enters the information and presses the "Generate button," the Fortran Module Wizard asks COM to open the type library and generates a file containing Fortran modules.

Generated Code

The Fortran Module Wizard generates different code, depending upon the type of object or DLL described by the type information. Note that the generated code is a static representation of an object's type information. If the type information should change in a future release of the object, the Wizard would need to be run again.

Fortran Run-time Support DIGITAL Visual Fortran provides a set of run-time routines that present to the Fortran programmer a higher-level abstraction of the

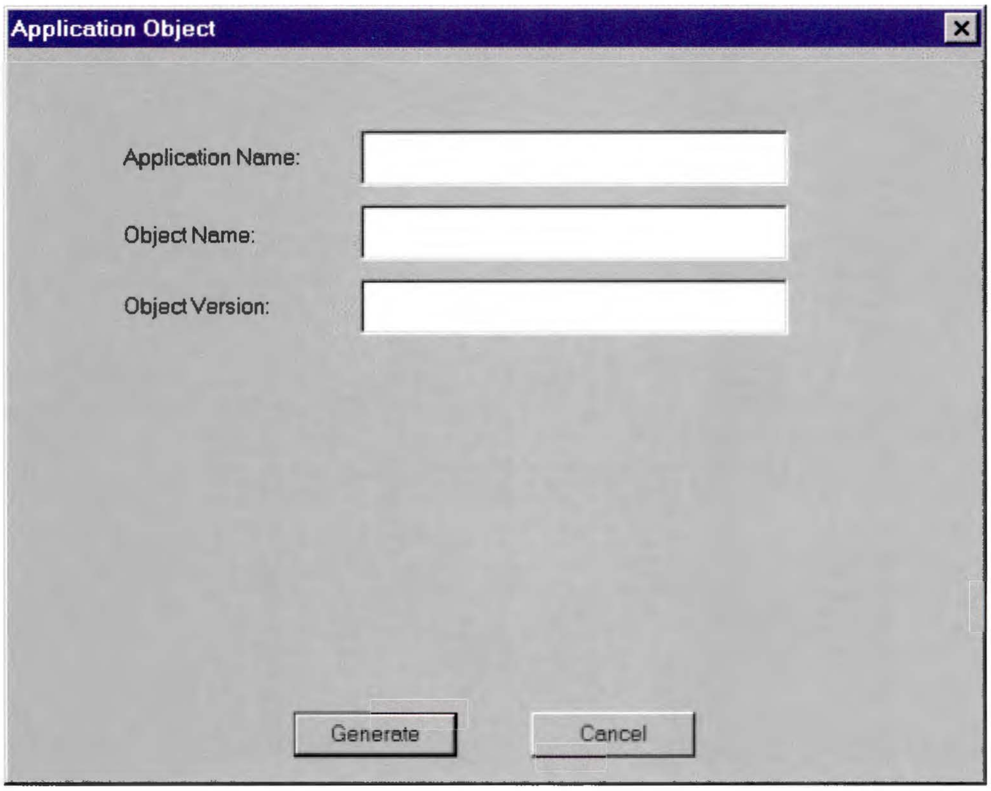
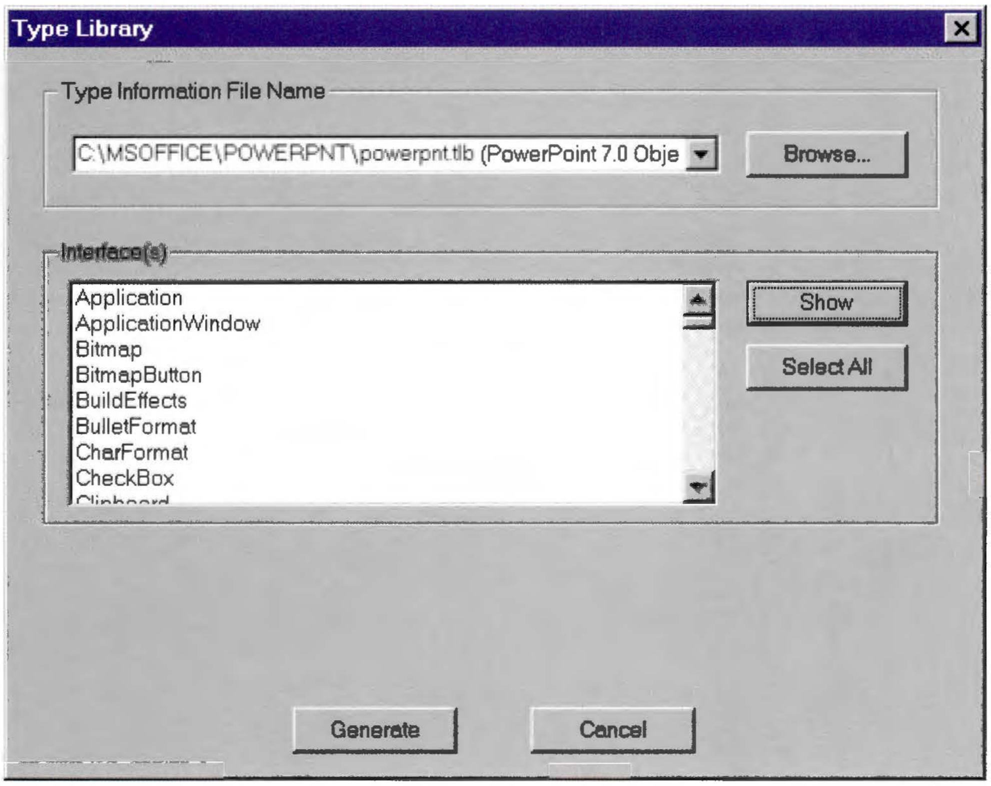


Figure 3
Microsoft Developer Studio Dialog Box for Application Object Selection



C:\MSOFFICE\POWERPNT\powerpnt.tlb (PowerPoint 7.0 Obje

Application
ApplicationWindow
Bitmap
BitmapButton
BuildEffects
BulletFormat
CharFormat
CheckBox

Figure 4
Microsoft Developer Studio Dialog Box for Type Library Selection

IDispatch member functions and other COM functions. The routines are used in the code that the Wizard generates. They allow the programmer to perform the following tasks:

- Initialize the COM library.
  - COMInitialize initializes the COM library.
  - COMUninitialize uninitializes the COM library.
- Get an interface pointer of an object.
  - COMCreateObject passes a programmatic identifier or class identifier, and it creates an instance of an object and returns a pointer to one of the object's interfaces.
  - COMGetActiveObject passes a programmatic identifier or class identifier, and it returns a pointer to an interface of a currently active object.
  - COMGetFileObject passes a file name, and it returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
  - COMCLSIDFromPROGID passes a programmatic identifier, and it returns the corresponding class identifier.
  - COMCLSIDFromString passes a class identifier string, and it returns the corresponding class identifier.
- Get or set the value of a property of an Automation object.
  - AUTOSetProperty passes the name or identifier of the property and a value, and it sets the value of the Automation object's property.
  - AUTOGetProperty passes the name or identifier of the property, and it gets the value of the Automation object's property.
- Invoke a method of an Automation object.
  - AUTOAllocateInvokeArgs allocates an argument list data structure that holds the arguments that the user will pass to AUTOInvoke.
  - AUTOAddArg passes an argument name and value, and it adds the argument to the argument list data structure.
  - AUTOInvoke passes the name or identifier of an object's method and an argument list data structure, and it invokes the method with the passed arguments.
  - AUTODeallocateInvokeArgs deallocates an argument list data structure.
  - AUTOGetExceptionInfo retrieves the exception information when a method has returned an exception status.
- Perform IUnknown interface member functions.
  - COMAddObjectReference adds a reference to an object's interface.
  - COMReleaseObject indicates that the program is done with a reference to an object's interface.
  - COMQueryInterface passes an interface identifier, and it returns a pointer to an object's interface.

DIGITAL Visual Fortran provides three Fortran modules that define basic COM information:

- DFCOMTY defines basic COM types.
- DFCOM defines the interfaces to the DIGITAL Visual Fortran COM routines and to some COM system routines.
- DFAUTO defines the interfaces to the DIGITAL Visual Fortran Automation routines.

Automation Objects Figure 5 contains code generated by the Fortran Module Wizard for the Word.Basic object of Microsoft Word version 7.0. Word.Basic is an Automation object with almost 1,000 methods. These methods represent the functionality of the Word Basic language, which is the programming interface to Microsoft Word. The Microsoft Word, Word Basic documentation contains information on the methods and their arguments.[12] We discuss some of the methods here in a simple example of Fortran code automating Word Basic to perform the task of replacing all the occurrences of a word in a document with another word. The Word.Basic methods of interest for this example are the following:

- AppShow makes the Microsoft Word application visible.
- FileOpen opens a document.
- EditReplace replaces a string with another string.
- FileSaveAs saves a document.

Figure 5 contains code from the Fortran subroutine generated for the Word Basic FileOpen method. It is representative of the code generated for all Automation methods. The lines are annotated on the left side with numbers that are not part of the source code but correspond to the list below. Note that the naming convention used for the generated wrappers is *objectname_methodname*. Any periods in the name are replaced by underscores.

1. If the type information provides a comment that describes the method, the comment is placed before the beginning of the procedure.
2. The first argument to the procedure is always $OBJECT. It is a pointer to an Automation object's IDispatch interface. The last argument to the procedure is always $STATUS. This optional argument can be specified if the Fortran programmer wishes to examine the return status of the method. The IDispatch Invoke member function returns a status of type HRESULT, which is a 32-bit value. HRESULT has the same structure as a Win32 error code. In between the $OBJECT and $STATUS arguments are the method arguments' names determined from the type information. When the type information does not provide a name for an argument, the Fortran Module Wizard creates a $ARGn name.

```
1-  !Opens an existing document or template
2-  SUBROUTINE Word_Basic_FileOpen($OBJECT, Name, ConfirmConversions,
      ReadOnly, LinkToSource, AddToMru, PasswordDoc, PasswordDot,
      Revert, WritePasswordDoc, WritePasswordDot, Connection,
      SQLStatement, SQLStatement1, $STATUS)
    !DEC$ ATTRIBUTES DLLEXPORT     :: Word_Basic_FileOpen
    IMPLICIT NONE
    INTEGER*4, INTENT(IN)          :: $OBJECT   ! Object Pointer
3-  !DEC$ ATTRIBUTES VALUE         :: $OBJECT
4-  CHARACTER*(*), INTENT(IN), OPTIONAL :: Name ! BSTR
    !DEC$ ATTRIBUTES REFERENCE     :: Name

    ...
    INTEGER*4, INTENT(OUT), OPTIONAL :: $STATUS ! Method status
    !DEC$ ATTRIBUTES REFERENCE     :: $STATUS
    INTEGER*4 $$STATUS
    INTEGER*4 invokeargs
5-  invokeargs = AUTOALLOCATEINVOKEARGS()
6-  IF (PRESENT(Name)) CALL AUTOADDARG(invokeargs, 'Name', Name,
                                     .FALSE., VT_BSTR)

    ...
7-  $$STATUS = AUTOINVOKE($OBJECT, 'FileOpen', invokeargs)
8-  IF (PRESENT($STATUS)) $STATUS = $$STATUS
9-  CALL AUTODEALLOCATEINVOKEARGS (invokeargs)
    END SUBROUTINE Word_Basic_FileOpen
```

Figure 5
Representative Code Generated for Automation Methods

3. This is an example of an attribute statement used to specify the calling convention of an argument.
4. Methods can take optional arguments that must follow all the required arguments. In this method, there are no required arguments. The Fortran Module Wizard generates source lines for each argument using the data type and calling conventions found in the type information.
5. AUTOAllocateInvokeArgs allocates a data structure that is used to collect the arguments that the programmer passes to the method. AUTOAddArg adds an argument to this data structure.
6. For each optional argument, the Fortran PRESENT function is used to determine if the caller supplied the argument. If so, the argument is added to the argument list.
7. AUTOInvoke invokes the named method passing the argument list. This returns a status result.
8. If the caller supplied a status argument, the code copies the status result to it.
9. AUTODeallocateInvokeArgs deallocates the memory used by the argument list data structure.

Figure 6 shows code from a user-written Fortran program that invokes Microsoft Word to replace all the occurrences of a word in a document with another word. The example code is annotated with numbers that correspond to the following list.

1. COMCreateObject requests COM to create an object with the ProgID Word.Basic. A pointer to the Word.Basic object's IDispatch interface is returned in "wordapp." The IDispatch interface is returned with a reference count of 1.
2. The code checks to ensure that an IDispatch pointer was returned. If not, it displays an error message and exits. The programmer can examine the status variable for the specific status return code.
3. The code calls Word.Basic methods to show the Microsoft Word window, open the document, replace the string, and save the modified document.
4. COMReleaseObject releases the single reference to the object's IDispatch interface so that Microsoft Word can terminate.

COM Objects The Microsoft PowerPoint version 7.0 type library contains a description of a number of COM objects and interfaces that make up the programmable interface to the Microsoft PowerPoint application. Figures 7 and 8 contain code generated by the Fortran Module Wizard from the Microsoft PowerPoint version 7.0 type library. Unlike Microsoft Word, which provides a single object that presents all of Word's programmable functionality, PowerPoint provides a hierarchy of objects. The top-level object, Application, is identified by the ProgID PowerPoint.Application.7. The Application object contains member functions that return a pointer to subordinate objects, including the Presentations

```
     ! Create a Word object and make it visible
  1-   CALL COMCREATEOBJECT ("Word.Basic," wordapp, status)
  2-   IF (wordapp == 0) THEN
           WRITE (*,
              '(" Unable to create Microsoft Word object; Aborting")')
           CALL EXIT(-1)
       END IF
  3-   CALL Word_Basic_AppShow(wordapp, "," $STATUS=status)

       ! Open the document
       CALL Word_Basic_FileOpen(wordapp, filename, $STATUS=status)

       ! Replace all occurrences of the string
       CALL Word_Basic_EditReplace(wordapp, findstring, replacestring,
                        ReplaceAll=.TRUE., $STATUS=status)

       ! Save the file
       CALL Word_Basic_FileSaveAs(wordapp, filename, $STATUS=status)

       ! Release the Word.Basic object since we are done
  4-   status = COMRELEASEOBJECT(wordapp)
```

Figure 6
Code from a User-written Fortran Program That Invokes Microsoft Word

object. The Presentations object consists of a collection of Presentation objects. A Presentation contains a member function that returns a pointer to its SlideShow object, and so on. By navigating this hierarchy, the developer can select a pointer to a particular object's interface. A code example in which we use some of the PowerPoint objects and interfaces to run a slide presentation from PowerPoint is given later in this section.

Figure 7 contains the interface description of the Presentations object's member function named Open. It is representative of the interfaces generated for all COM member functions. The procedure naming convention is *objectname_memberfunctionname*. The Open function opens an existing PowerPoint presentation.

1. The first argument to the procedure is always $OBJECT. It is a pointer to the object's interface. The remaining argument names are determined from the type information.
2. A BSTR is a length-prefixed string data type primarily for use by Automation objects. The wrappers generated for COM member functions convert from Fortran strings to BSTRs and vice versa.
3. A VARIANT is a data structure that can contain any type of Automation data. It contains a field that identifies the type of data and a union that holds the data value. The use of a VARIANT argument allows the caller to use any data type that can be converted into the data type expected by the member function.

```
       INTERFACE
  1-     INTEGER*4 FUNCTION Presentations_Open($OBJECT, fileName,
                   ReadOnly, Untitled, WithWindow, Open)
          USE DFCOMTY
          INTEGER*4, INTENT(IN)          :: $OBJECT    ! Object Pointer
          !DEC$ ATTRIBUTES VALUE   :: $OBJECT
  2-      INTEGER*4, INTENT(IN)          :: fileName   ! BSTR
          !DEC$ ATTRIBUTES VALUE   :: fileName
  3-      TYPE (VARIANT), INTENT(IN), :: ReadOnly ! (Optional Arg)
          !DEC$ ATTRIBUTES VALUE   :: ReadOnly
          TYPE (VARIANT), INTENT(IN), :: Untitled ! (Optional Arg)
          !DEC$ ATTRIBUTES VALUE   :: Untitled
          TYPE (VARIANT), INTENT(IN), :: WithWindow ! (Optional Arg)
          !DEC$ ATTRIBUTES VALUE   :: WithWindow
  4-      INTEGER*4, INTENT(OUT)  :: Open
          !DEC$ ATTRIBUTES REFERENCE     :: Open
        !DEC$ ATTRIBUTES STDCALL   :: Presentations_Open
        END FUNCTION Presentations_Open
       END INTERFACE
  5- POINTER(Presentations_Open_PTR, Presentations_Open)
```

Figure 7
Code Generated by Fortran Module Wizard from Microsoft PowerPoint, Interface Description of Open Function

4. Nearly every COM member function returns a status of type HRESULT. Therefore if a COM member function produces output, it uses output arguments to return the values. In this example, the Open argument returns a pointer to a PowerPoint Presentation object.
5. The interface of a COM member function looks similar to the interface for a DLL function with one major exception. Unlike a DLL function, the address of a COM member function is never known at program link time. To compute the address of a particular member function, the developer must get a pointer to an object's interface at run time. We have extended the DIGITAL Fortran 90 language to support a Pointer To procedure. Figure 8 shows an example of its use.

Figure 8 contains the wrapper generated by the Fortran Module Wizard for the Open function. The name of a wrapper is the same as the name of the corresponding member function, prefixed with a $. The numbers inserted at the left margin of the code example correspond to the following list.

1. The wrapper takes the same argument names as the member function interface.
2. Member function arguments of type BSTR are of type CHARACTER*(*) in the wrapper.
3. The wrapper computes the address of the member function from the interface pointer and an offset found in the interface's type information. In implementation terms, the sequence is the following: an interface pointer to a pointer to an array of function pointers called an Interface Function Table (see Figure 9).
4. The wrapper declares a local variable to hold the BSTR to be passed to the member function. The next line does the conversion.
5. Optional VARIANT arguments of a COM member function are represented by a VARIANT with distinguished values. OPTIONAL_VARIANT is defined in the DFCOMTY module with the distinguished values.
6. The offset of the Open member function is 60. The code assigns the computed address to the function pointer Presentations_Open_PTR, which was declared in Figure 7, and then calls the function.

```
1-  INTEGER*4 FUNCTION $Presentations_Open($OBJECT, fileName,
              ReadOnly, Untitled, WithWindow, Open)
      !DEC$ ATTRIBUTES DLLEXPORT   :: $Presentations_Open
      IMPLICIT NONE
      INTEGER*4, INTENT(IN)        :: $OBJECT   ! Object Pointer
      !DEC$ ATTRIBUTES VALUE       :: $OBJECT
2-    CHARACTER*(*), INTENT(IN)    :: fileName  ! BSTR
      !DEC$ ATTRIBUTES REFERENCE   :: fileName
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: ReadOnly
      !DEC$ ATTRIBUTES REFERENCE   :: ReadOnly
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: Untitled
      !DEC$ ATTRIBUTES REFERENCE   :: Untitled
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: WithWindow
      !DEC$ ATTRIBUTES REFERENCE   :: WithWindow
      INTEGER*4, INTENT(OUT)       :: Open     ! IDispatch
      !DEC$ ATTRIBUTES REFERENCE   :: Open
      INTEGER*4 $RETURN
3-    INTEGER*4 $VTBL              ! Interface Function Table
      POINTER($VPTR, $VTBL)
      TYPE (VARIANT), :: $ VAR_ReadOnly
      TYPE (VARIANT), :: $ VAR_Untitled
      TYPE (VARIANT), :: $ VAR_WithWindow
4-    INTEGER*4 $BSTR_fileName     ! BSTR
      $BSTR_fileName = ConvertStringToBSTR(fileName)
5-    IF (PRESENT (ReadOnly)) THEN
        $VAR_ReadOnly = ReadOnly
      ELSE
        $VAR_ReadOnly = OPTIONAL_VARIANT
      Presentations_Open_PTR = $VTBL
      END IF
      ...
6-    $VPTR = $OBJECT              ! Interface Function Table
      $VPTR = $VTBL + 60           ! Add routine table offset
      Presentations_Open_PTR = $VTBL
      $RETURN = Presentations_Open($OBJECT, $BSTR_fileName,
              ReadOnly, Untitled, WithWindow, Open)
      $Presentations_Open = $RETURN
    END FUNCTION $Presentations_Open
```

Figure 8
Code Generated by Fortran Module Wizard from Microsoft PowerPoint, Wrapper for Open Function

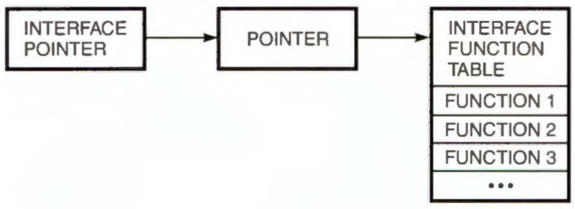


Figure 9
Interface Pointer to an Array of Function Pointers

In fact, PowerPoint provides dual interfaces. A dual interface is a combination of an IDispatch interface and COM member functions. The IDispatch interface of the dual interface can be used by Automation clients, and the COM member functions can be used by COM clients. This means that for PowerPoint, and any server that provides dual interfaces, the Fortran developer can choose to generate a Fortran module for the Automation interfaces or the COM interfaces. The Fortran interfaces generated by the Wizard likely will not be much different. COM interfaces typically provide better performance since there is less overhead in invoking COM member functions than dispinterface methods through the IDispatch Invoke member function.

Figure 10 shows code from a user-written Fortran program that invokes PowerPoint to run a slide presentation. The code example is annotated with numbers that correspond to the following list.

1. COMCLSIDFromPROGID and COMCreateObject request COM to create an object with the ProgID PowerPoint.Application.7, and to return a pointer to the object's IApplication interface.
2. The code gets the AppWindow object from the Application object and calls its Visible member function to make PowerPoint visible.
3. The code gets the Presentations object from the Application object and calls its Open member function to open a Presentation. Note that three of the arguments to Open are of the VARIANT data type. The code sets them to the values true and false.
4. The code gets the SlideShow object from the Presentation object and calls its Run member function to run the slide show.

DLLs When the Fortran Module Wizard reads the type information describing a DLL, it generates an interface description for each function in the DLL. It also generates Fortran-derived types for data structures defined in the DLL type information. This relieves the Fortran developer from manually translating header file descriptions to Fortran descriptions. The Wizard also provides the option of generating wrappers that convert from the Fortran representation of strings to the C representation of strings and vice versa. This option can be selected from the Wizard's initial dialog box (see Figure 2).

```
        ! Create a PowerPoint Application object
        ! and make the AppWindow visible
    1-  CALL COMCLSIDFROMPROGID ("PowerPoint.Application.7,"
                                   clsid, status)
        CALL COMCREATEOBJECT (clsid, CLSCTX_SERVER, IID_Application,
                                   ppApplication, status)
        IF (ppApplication == 0) THEN
          WRITE (*, '(" Unable to create PowerPoint object; Aborting")')
          CALL EXIT(-1)
        END IF
    2-  status = $Application_GetAppWindow(ppApplication, ppAppWindow)
        status = $ApplicationWindow_SetVisible(ppAppWindow, 1)

        ! Open the specified presentation
    3-  status = $Application_GetPresentations(ppApplication,
                                                ppPresentations)
        vTrue%VT = VT_BOOL
        vTrue%VU%BOOL_VAL = VARIANT_BOOL_TRUE
        vFalse%VT = VT_BOOL
        vFalse%VU%BOOL_VAL = VARIANT_BOOL_FALSE
        status = $Presentations_Open(ppPresentations, filename,
                              vTrue, vFalse, vTrue, ppPresentation)

        ! Run the slide show
    4-  status = $Presentation_GetSlideShow(ppPresentation, ppSlideShow)
        status = $SlideShow_Run(ppSlideShow, 1, ppRun)
```

Figure 10
Fortran Program to Invoke PowerPoint to Run Slide Presentation

Comparison of the Wizard to the Capabilities of Other Languages

Visual C++ version 5.0, Visual J++ version 1.1, and Visual Basic version 5.0 all have wizards that can read a type library and allow applications to use COM and/or Automation objects.

The Visual C++ ClassWizard can read a type library and create a class with all the functions of the IDispatch interface described in the library. Visual C++ version 5.0 also adds a preprocessor directive, #import. The #import directive reads a type library and generates two header files that contain the definitions of the COM objects defined in the type library.[13]

The Java Type Library Wizard within Visual J++ invokes the JavaTLB utility to convert the information in a type library into Java .class files. A Java .class file is the binary form of a Java class or interface.[14]

To use an object defined in a type library from Visual Basic, the developer must add a reference to the object using the Project menu, References command. The References dialog box allows the user to select from the list of registered type libraries in a manner similar to the Fortran Module Wizard.[15]

The Fortran Module Wizard is unique in the following ways. The Fortran 90 programming language does not inherently support objects. The Fortran Module Wizard employs a combination of language and run-time support to provide this capability. The supporting language features are modules and procedure pointers. The supporting run-time modules are DFCOMTY, DFCOM, and DFAUTO. The Fortran Module Wizard provides support for type libraries containing the descriptions of DLL routines.

Fortran Module Wizard Architecture

The architecture of the Fortran Module Wizard is fairly simple. The shell of the Wizard was generated by the Custom AppWizard within Visual C++. The inner workings of the Wizard consist of three major pieces:

- Type information reader
- Type symbol table
- Fortran code generator

Figure 11 shows a high-level data flow of the Fortran Module Wizard. The type information reader traverses the data structures in the type information and creates the type symbol table. The Win32 SDK provides a sample application named BROWSE OLE sample that is an example of traversing the information in a type library. The type symbol table is a symbol table similar to those used by compilers. It maps type names to the descriptions of types. For simplicity, the information is stored using the same data structures used by the type information. The Fortran code generator traverses the symbol table and generates a Fortran module.

The use of a symbol table allows for a complete separation of the functionality of the type information reader from the Fortran code generator. A code generator for another programming language could be easily substituted, as could another source of type information (for example, a C header file).

Future Directions

There are a number of possibilities for future work that would add to the capabilities provided by the Fortran Module Wizard.

- Fortran support for ActiveX controls. An ActiveX control is an Automation object. It is a reusable component that normally provides a user interface and is used in dialog boxes and other windows. The Fortran Module Wizard can generate a module that would allow a Fortran developer to use the methods and properties of an ActiveX control. However, additional functionality would be needed in the Fortran run-time libraries to make controls usable from a Fortran application. A control has to be placed in a special type of window called a Control Container. The Fortran run-time libraries do not currently contain support for a Control Container. In addition to methods and properties, a control can define events. An event allows a control to notify its container when something of interest happens to the control. For example, a "Button control" could define a "Clicked event."
- Fortran Windows Application Wizard. This Wizard could generate starter files for a Fortran Windows application. This would be especially useful if we were to implement the Fortran support for ActiveX controls.

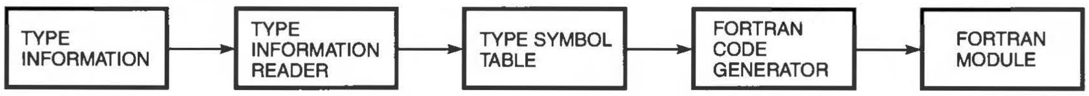


Figure 11
Data Flow of the Fortran Module Wizard

- Fortran modules from C header files. By replacing the type information reader described in the previous section with a C parser, we could generate Fortran modules directly from .h files. This would expand the set of services that are easily available to Fortran developers.
- Fortran Server Wizard. This Wizard would take a Fortran module provided by a Fortran developer and package it as a COM object. It would also generate a type library that describes the object. This object could then be used by any COM client, for example, Visual Basic, Visual C++, and Visual J++ applications.

References and Notes

1. *Digital Fortran Books Online* (Maynard, Mass.: Digital Equipment Corporation, 1997).
2. *Digital Fortran 90 Language Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1997).
3. For a period of time, Microsoft used the name OLE to encompass all of its component integration technology, including COM. Now OLE is applied only to compound document technology.
4. K. Brockschmidt, *Inside OLE*, Second Edition (Redmond, Wash.: Microsoft Press, 1995).
5. K. Brockschmidt, "How OLE and COM Solve the Problems of Component Software Design," *Microsoft Systems Journal*, vol. 11, no. 5 (May 1996): 63–80.
6. D. Chappell, *Understanding ActiveX and OLE* (Redmond, Wash.: Microsoft Press, 1996).
7. *OLE 2 Programmer's Reference, Volume Two* (Redmond, Wash.: Microsoft Press, 1994).
8. *The Component Object Model Specification 0.9* (Redmond, Wash.: Microsoft Corporation, 1995).
9. Automation was originally called OLE Automation.
10. Before IDL and MIDL, Microsoft provided the Object Description Language (ODL) and a compiler named MKTYPLIB.
11. *Developer Studio Environment User's Guide* (Redmond, Wash.: Microsoft Corporation, 1997).
12. Microsoft Office 97 includes a new Office object model that offers another set of interfaces to Word services.
13. G. Shepherd, "Visual C++ Simplifies the Process for Developing and Using COM Objects," *Microsoft Systems Journal*, vol. 12, no. 5 (May 1997): 37–48.
14. G. Eddon and H. Eddon, "Understanding the Java/COM Integration Model," *Microsoft Interactive Developer*, vol. 2, no. 4 (April 1997): 56–68.
15. *Microsoft Visual Basic 5.0 Books Online* (Redmond, Wash.: Microsoft Corporation, 1997).

Biography

Leo P. Treggiari
Leo Treggiari is a consulting software engineer in the Core Technology Group. He was responsible for developing the Module Wizard in the DIGITAL Visual Fortran product for the Fortran programmer working in a Microsoft Windows environment. Previous to this work, he was project leader for the development of several programming tools, including the Motif toolkit. Leo came to DIGITAL in 1979 from Wang Laboratories. He holds a B.S. (1975, summa cum laude) in chemistry from Boston College and is a member of ACM.

Marco Fillo
Richard B. Gillett

Architecture and Implementation of MEMORY CHANNEL 2

The MEMORY CHANNEL network is a dedicated cluster interconnect that provides virtual shared memory among nodes by means of internodal address space mapping. The interconnect implements direct user-level messaging and guarantees strict message ordering under all conditions, including transmission errors. These characteristics allow industry-standard communication interfaces and parallel programming paradigms to achieve much higher efficiency than on conventional networks. This paper presents an overview of the MEMORY CHANNEL network architecture and describes DIGITAL's crossbar-based implementation of the second-generation MEMORY CHANNEL network, MEMORY CHANNEL 2. This network provides bisection bandwidths of 1,000 to 2,000 megabytes per second and a sustained process-to-process bandwidth of 88 megabytes per second. One-way, process-to-process message latency is less than 2.2 microseconds.

In computing, a cluster is loosely defined as a parallel system comprising a collection of stand-alone computers (each called a node) connected by a network. Each node runs its own copy of the operating system, and cluster software coordinating the entire parallel system attempts to provide users with a unified system view. Since each node in the cluster is an off-the-shelf computer system, clusters offer several advantages over traditional massively parallel processors (MPPs) and large-scale symmetric multiprocessors (SMPs). Specifically, clusters provide[1]

- Much better price/performance ratios, opening a wide range of computing possibilities for users who could not otherwise afford a single large system.
- Much better availability. With appropriate software support, clusters can survive node failures, whereas SMP and MPP systems generally do not.
- Impressive scaling (hundreds of processors), when the individual nodes are medium-scale SMP systems.
- Easy and economical upgrading and technology migration. Users can simply attach the latest-generation node to the existing cluster network.

Despite their advantages and their impressive peak computational power, clusters have been unable to displace traditional parallel systems in the marketplace because their effective performance on many real-world parallel applications has often been disappointing. Clusters' lack of computational efficiency can be attributed to their traditionally poor communication, which is a result of the use of standard networking technology as a cluster interconnect. The development of the MEMORY CHANNEL network as a cluster interconnect was motivated by the realization that the gap in effective performance between clusters and SMPs can be bridged by designing a communication network to deliver low latency and high bandwidth all the way to the user applications.

Over the years, many researchers have recognized that the performance of the majority of real-world parallel applications is affected by the latency and bandwidth available for communication.[2–5] In particular, it has been shown[2,6,7] that the efficiency of parallel scientific applications is strongly influenced by the system's architectural balance as quantified by its communication-to-computation ratio, which is sometimes called the q-ratio.[2] The q-ratio is defined as the ratio between the time it takes to send an 8-byte floating-point result from one process to another (communication) and the time it takes to perform a floating-point operation (computation). In a system with a q-ratio equal to 1, it takes the same time for a node to compute a result as it does for the node to communicate the result to another node in the system. Thus, the higher the q-ratio, the more difficult it is to program a parallel system to achieve a given level of performance. Q-ratios close to unity have been obtained only in experimental machines, such as iWarp[8] and the M-Machine,[9] by employing direct register-based communication.

Table 1 shows actual q-ratios for several commercial systems.[10,11] These q-ratios vary from about 100 for a DIGITAL AlphaServer 4100 SMP system using shared memory to 30,000 for a cluster of these SMP systems interconnected over a fiber distributed data interface (FDDI) network using the transmission control protocol/internet protocol (TCP/IP). An MPP system, such as the IBM SP2, using the Message Passing Interface (MPI) has a q-ratio of 5,714. The MEMORY CHANNEL network developed by Digital Equipment Corporation reduces the q-ratio of an AlphaServer-based cluster by a factor of 38 to 82 to be within the range of 367 to 1,067. Q-ratios in this range permit clusters to efficiently tackle a large class of parallel technical and commercial problems.

The benefits of low-latency, high-bandwidth networks are well understood.[12,13] As shown by many studies,[14,15] high communication latency over traditional networks is the result of the operating system overhead involved in transmitting and receiving messages. The MEMORY CHANNEL network eliminates this latency by supporting direct process-to-process communication that bypasses the operating system. The MEMORY CHANNEL network supports this type of communication by implementing a natural extension of the virtual memory space, which provides direct, but protected, access to the memory residing in other nodes.

Based on this approach, DIGITAL developed its first-generation MEMORY CHANNEL network (MEMORY CHANNEL 1),[16] which has been shipping in production since April 1996. The network does not require any functionality beyond the peripheral component interconnect (PCI) bus and therefore can be used on any system with a PCI I/O slot. DIGITAL currently supports production MEMORY CHANNEL clusters as large as 8 nodes by 12 processors per node (a total of 96 processors). One of these clusters was presented at Supercomputing '95 and ran clusterwide applications using High Performance Fortran (HPF),[4] Parallel Virtual Machine (PVM),[17] and MPI[18] in DIGITAL's Parallel Software Environment (PSE). This 96-processor system has a q-ratio of 500 to 1,000, depending on the communication interface. A 4-node MEMORY CHANNEL cluster running DIGITAL TruCluster software[19] and the Oracle Parallel Server has held the cluster performance world record on the TPC-C benchmark[20]—the industry standard in on-line transaction processing—since April 1996.

We next present an overview of the generic MEMORY CHANNEL network to justify the design goals of the second-generation MEMORY CHANNEL network (MEMORY CHANNEL 2). Following this overview, we describe in detail the architecture of the two components that make up the MEMORY CHANNEL 2 network: the hub and the adapter. Last, we present hardware-measured performance data.

MEMORY CHANNEL Overview

The MEMORY CHANNEL network is a dedicated cluster interconnection network, based on Encore's

Table 1
Comparison of Communication and Computation Performance (q-ratio) for Various Parallel Systems

System	Communication Performance Latency (Microseconds)	Computation Performance Based on LINPACK 100 × 100 (Microseconds/FLOP)	Communication-to-computation Ratio (q-ratio)
AlphaServer 4100 Model 300 configurations			
SMP using shared memory messaging	0.6	0.006	100
SMP using MPI	3.4	0.006	567
FDDI cluster using TCP/IP	180.0	0.006	30,000
MEMORY CHANNEL cluster using native messaging	2.2	0.006	367
MEMORY CHANNEL cluster using MPI	6.4	0.006	1,067
IBM SP2 using MPI	40.0	0.006	5,714

MEMORY CHANNEL technology, that supports virtual shared memory space by means of internodal memory address space mapping, similar to that used in the SHRIMP system.[21] The MEMORY CHANNEL substrate is a flat, fully interconnected network that provides *push*-only message-based communication.[16,22] Unlike traditional networks, the MEMORY CHANNEL network provides low-latency communication by supporting direct user access to the network. As in Scalable Coherent Interface (SCI)[23] and Myrinet[24] networks, connections between nodes are established by mapping part of the nodes' virtual address space to the MEMORY CHANNEL interface.

A MEMORY CHANNEL connection can be opened as either an outgoing connection (in which case an address-to-destination node mapping must be provided) or an incoming connection. Before a pair of nodes can communicate by means of the MEMORY CHANNEL network, they must consent to share part of their address space—one side as outgoing and the other as incoming. The MEMORY CHANNEL network has no storage of its own. The granularity of the mapping is the same as the operating system page size.

MEMORY CHANNEL Address Space Mapping

Mapping is accomplished through manipulation of page tables. Each node that maps a page as incoming allocates a single page of physical memory and makes it available to be shared by the cluster. The page is always resident and is shared by all processes in the node that map the page. The first map of the page causes the memory allocation, and subsequent reads/maps point to the same page. No memory is allocated for pages mapped as outgoing. The mapper simply assigns the page table entry to a portion of the MEMORY CHANNEL hardware transmit window and defines the destination node for that transmit subspace. Thus, the amount of physical memory consumed for the clusterwide network is the product of the operating system page size and the total number of pages mapped as incoming on each node.

After mapping, MEMORY CHANNEL accesses are accomplished by simple load and store instructions, as for any other portion of virtual memory, without any operating system or run-time library calls. A store instruction to a MEMORY CHANNEL outgoing address results in data being transferred across the MEMORY CHANNEL network to the memory allocated on the destination node. A load instruction from a MEMORY CHANNEL incoming channel address space results in a read from the local physical memory initialized as a MEMORY CHANNEL incoming channel. The overhead (in CPU cycles) in establishing a MEMORY CHANNEL connection is much higher than that of using the connection. Because of the memory-mapped nature of the interface, the transmit or receive overhead is similar to an access to local main memory. This mechanism is the fundamental reason for the low MEMORY CHANNEL latency. Figure 1 illustrates an example of MEMORY CHANNEL address mapping.

The figure shows two sets of independent connections. Node 1 has established an outgoing channel to node 3 and node 4 and also an incoming channel to itself. Node 4 has an outgoing channel to node 2.

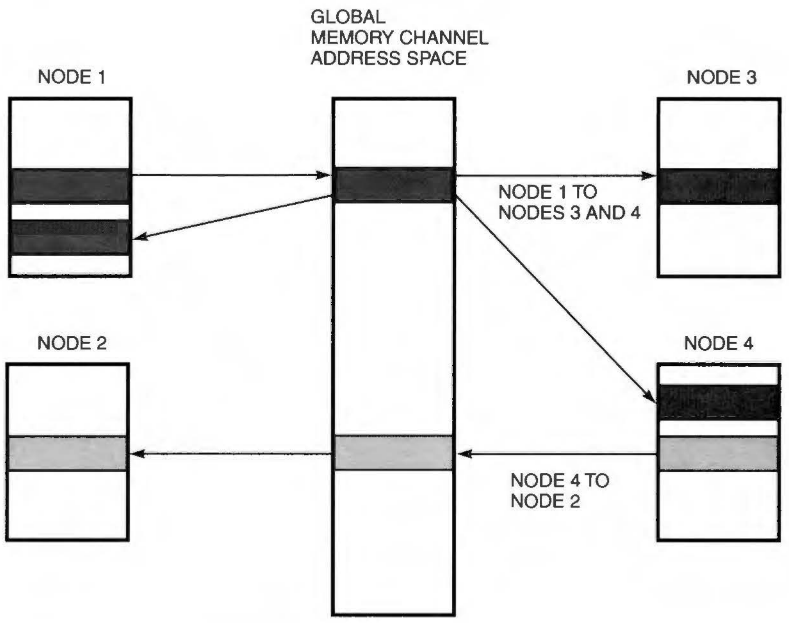


Figure 1
MEMORY CHANNEL Mapping of a Portion of the Clusterwide Address Space

All connections are unidirectional, either outgoing or incoming. To map a channel as both outgoing and incoming to the same shared address space, node 1 maps the channel two times into a single process' virtual address space. The mapping example in Figure 1 requires a total of four pages of physical memory, one for each of the four arrows pointed toward the nodes' virtual address spaces.

MEMORY CHANNEL mappings reside in two page control tables (PCTs) located on the MEMORY CHANNEL interface, one on the sender side and one on the receiver side. As shown in Figure 2, each page entry in the PCT has a set of attributes that specify the MEMORY CHANNEL behavior for that page.

The page attributes on the sender side are

- Transmit enabled, which must be set to allow transmission from store instructions to a specific page
- Local copy on transmit, which directs an ordered copy of the transmitted packet to the local memory
- Acknowledge request, which is used to request acknowledgments from the receiver node
- Transmit enabled under error, which is used in error recovery communication
- Broadcast or point-to-point, which defines the type of packet to all nodes or to a single node in the cluster
- Request acknowledge, which requests a reception acknowledgment from the receiver

The page attributes on the receiver side are

- Receive enabled, which must be set to allow reception of messages addressed to a specific virtual page
- Interrupt on receive, which generates an interrupt on reception of a packet
- Receive enabled under error, which is asserted for error recovery communication pages
- Remote read, which identifies all packets that arrive at a page as requests for a remote read operation
- Conditional write, which identifies all packets that arrive at a page as conditional write packets

MEMORY CHANNEL Ordering Rules

The MEMORY CHANNEL communication paradigm is based on three fundamental ordering rules:

1. Single-sender Rule: All destination nodes will receive packets in the order in which they were generated by the sender.
2. Multisender Rule: Packets from multiple sender nodes will be received in the same order at all destination nodes.
3. Ordering-under-errors Rule: Rules 1 and 2 must apply even when an error occurs in the network.

Let $Pj_{M \to X}$ be the jth point-to-point packet from a sender node M to a destination node X, and let Bj_M be the jth broadcast packet from node M to all other nodes. If node M sends the following sequence of packets,

$$\underset{\text{(last)}}{P2_{M\to X}},\ P1_{M\to Y},\ B1_M,\ \underset{\text{(first)}}{P1_{M\to X}},$$

Rule 1 dictates that nodes X and Y will receive the packets in the following order:

at node X, $\underset{\text{(last)}}{P2_{M\to X}},\ B1_M,\ \underset{\text{(first)}}{P1_{M\to X}}$

at node Y, $\underset{\text{(last)}}{P1_{M\to Y}},\ \underset{\text{(first)}}{B1_M}.$

If a node N is also sending a sequence of packets, in the following order,

$$\underset{\text{(last)}}{P3_{N\to X}},\ P2_{N\to X},\ B2_N,\ P2_{N\to Y},\ B1_N,\ P1_{N\to Y},\ \underset{\text{(first)}}{P1_{N\to X}},$$

there is a finite set of valid reception orders at destination nodes X and Y, depending on the actual arrival time of the requests to the point of global ordering. Rule 1 dictates that all packets from node M (or N) to node X (or Y) must arrive at node X (or Y) in the order in which they were transmitted. Rule 2 dictates that, regardless of the relative order among the senders, messages destined to both receivers must be received in the same order. For example, if X receives $B2_N$, $B1_M$, and $B1_N$, then Y should receive these packets in the

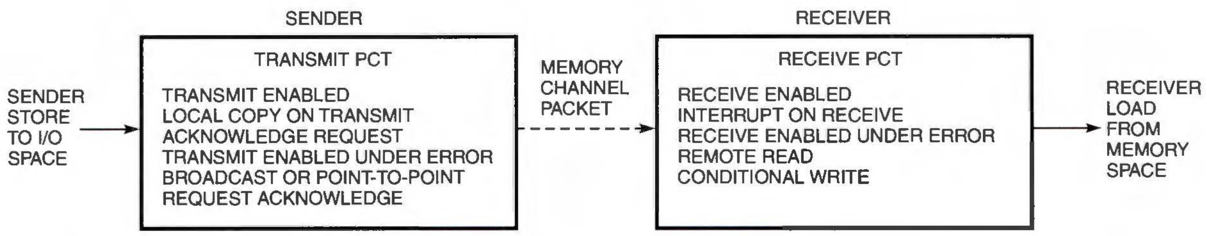


Figure 2
MEMORY CHANNEL Page Control Attributes

same order. One arrival order congruent with both of these rules is the following:

at node X,
$P3_{N\to X}$, $P2_{N\to X}$, $P2_{M\to X}$, $B2_N$, $B1_M$, $B1_N$, $P1_{N\to X}$, $P1_{M\to X}$
(last) (first)

at node Y,
$B2_N$, $P2_{N\to Y}$, $P1_{M\to Y}$, $B1_M$, $B1_N$, $P1_{N\to Y}$.

These rules are independent of a particular interconnection topology or implementation and must be obeyed in all generations of the MEMORY CHANNEL network.

On the MEMORY CHANNEL network, error handling is a shared responsibility of the hardware and the cluster management software. The hardware provides real-time precise error handling and strict packet ordering by discarding all packets in a particular path that follow an erroneous one. The software is responsible for recovering the network from the faulty state back to its normal state and for retransmitting the lost packets.

Additional MEMORY CHANNEL Network Features

Three additional features of the MEMORY CHANNEL network make it ideal for cluster interconnection:

1. A hardware-based barrier acknowledge that sweeps the network and all its buffers
2. A fast, hardware-supported lock primitive
3. Node failure detection and isolation

Because of the three ordering rules, the MEMORY CHANNEL network acknowledge packets are implemented with little variation over ordinary packets. To request acknowledgment of packet reception, a node sends an ordinary packet marked with the request-acknowledge attribute. The packet is used to sweep clean the network queues in the sender destination path and to ensure that all previously transmitted packets have reached the destination. In response to the reception of a MEMORY CHANNEL acknowledge request, the destination node transmits a MEMORY CHANNEL acknowledgment back to the originator. The arrival of the acknowledgment at the originating node signals that all preceding packets on that path have been successfully received.

MEMORY CHANNEL locks are implemented using a lock-acquire software data structure mapped as both incoming and outgoing by all nodes in the cluster. That is, each node will have a local copy of the page kept coherent by the mapping. To acquire a lock, a node writes to the shared data structure at an offset corresponding to its node identifier. MEMORY CHANNEL ordering rules guarantee that the write order to the data structure—including the update of the copy local to the node that is setting the lock—is the same for all nodes. The node can then determine if it was the only bidder for the lock, in which case the node has won the lock. If the node sees multiple bidders for the same lock, it resorts to an operating system–specific back-off-and-retry algorithm. Thanks to the MEMORY CHANNEL guaranteed packet ordering, even under error the above mechanism ensures that at most one node in the cluster perceives that it was the first to write the lock data structure. To guarantee that data structures are never locked indefinitely by a node that is removed from a cluster, the cluster manager software also monitors lock acquisition and release.

The MEMORY CHANNEL network supports a strong-consistency shared-memory model due to its strict packet ordering. In addition, the I/O operations used to access the MEMORY CHANNEL are fully integrated within the node's cache coherency scheme. Besides greatly simplifying the programming model, such consistency allows for an implementation of spinlocks that does not saturate the memory system. For instance, while a receiver is polling for a flag that signals the arrival of data from the MEMORY CHANNEL network, the node processor accesses only the locally cached copy of the flag, which will be updated whenever the corresponding main memory location is written by a MEMORY CHANNEL packet.

Unlike other networks, the MEMORY CHANNEL hardware maintains information on which nodes are currently part of the cluster. Through a collection of timeouts, the MEMORY CHANNEL hardware continuously monitors all nodes in the cluster for illegal behavior. When a failure is detected, the node is isolated from the cluster and recovery software is invoked. A MEMORY CHANNEL cluster is equipped with software capable of reconfiguration when a node is added or removed from the cluster. The node is simply brought on-line or off-line, the event is broadcast to all other nodes, and operation continues. To provide tolerance to network failures, the cluster can be equipped with a pair of topologically identical MEMORY CHANNEL networks, one for normal operation and the other for failover. That is, when a nonrecoverable error is detected on the active MEMORY CHANNEL network, the software switches over to the standby network, in a manner transparent to the application.[19]

The First-generation MEMORY CHANNEL Network

The first generation of the MEMORY CHANNEL network consists of a node interface card and a concentrator or hub. The interface card, called an adapter, plugs into the I/O PCI. To send a packet, the CPU writes to the portion of I/O space mapped to the PCI bus. The store-to-memory is handled by the node's PCI interface device, which initiates a PCI transfer targeting the MEMORY CHANNEL adapter transmit window. When a message is received, the MEMORY CHANNEL adapter initiates a PCI transfer to write to the node's CPU memory, targeting the node's PCI interface, which then accesses the node's main memory.

Besides writing to the node's CPU, an I/O device on the PCI bus can transmit directly to a MEMORY CHANNEL adapter. This allows, for example, a disk controller to transfer data directly from the disk to a remote node's memory. The data transfer does not affect the host system's memory bus. The design choice of interfacing MEMORY CHANNEL to the PCI bus instead of directly to the node CPU bus is not an architectural one, but rather one of practicality and universality. The PCI is available on most of today's systems of varying performance and size and is, therefore, an ideal interface point that allows hybrid clusters to be built. The obvious disadvantages of a peripheral interface bus are the additional latency incurred because of the extra CPU-to-PCI hop and a possible limitation on the available bus bandwidth.

The MEMORY CHANNEL 1 hub is a broadcast-only shared bus capable of interconnecting up to eight nodes. The MEMORY Channel 1 adapters and the hub are interconnected in a star topology via 37-bit-wide (32 bits of data plus sideband signals) half-duplex channels. The cables can be up to 4 meters long, and the signaling level is 5-volt TTL. A two-node cluster can be formed without employing a hub, by direct node-to-node interconnection. This configuration is also known as virtual hub configuration.

The current release of the MEMORY CHANNEL 1 hardware achieves a sustained point-to-point bandwidth of 66 megabytes per second (MB/s) (from user process to user process). Maximum sustained broadcast bandwidth is also 66 MB/s (from a user process to many user processes). The cross-section MEMORY CHANNEL 1 hub bandwidth is 77 MB/s. Small message latency is 2.9 microseconds (μs) (from a sender process STORE instruction to a message LOAD by a receiver process). The processor overhead is less than 150 nanoseconds (ns) for a 32-byte packet (which is also the largest packet size).

As demonstrated in the literature, standard message-passing application programming interfaces (APIs) benefit greatly from these MEMORY CHANNEL communication capabilities.[12,17,25] MPI, PVM, and HPF on MEMORY CHANNEL 1 all have one-way message latencies of less than 10 μs. These latency numbers are more than a factor of five lower than those for traditional MPP architectures (52 to 190 μs).[11] Communication performance improvements of this magnitude translate into cluster performance gains of 25 to 500 percent.[12]

MEMORY CHANNEL 2 Architecture

Based on the experience with the first-generation product, the design goals for MEMORY CHANNEL 2 were twofold: (1) yield a significant performance improvement over MEMORY CHANNEL 1, and (2) provide functional enhancements to extend hardware support to new operating systems and programming paradigms.

The MEMORY CHANNEL 2 performance/hardware enhancement goals were

- Network bisection bandwidth scalable with the number of nodes: 1,000 MB/s for an 8-node cluster and 2,000 MB/s for a 16-node cluster
- Improved point-to-point bandwidth, exploiting the maximum capability of the 32-bit PCI bus: 97 MB/s for 32-byte packets and 127 MB/s for 256-byte packets
- Full-duplex channels to allow simultaneous bidirectional transfers
- Maximum copper cable length of 10 meters (increased from 4 meters on MEMORY CHANNEL 1) and fiber support up to 3 kilometers
- A link layer communication protocol compatible with future generations of MEMORY CHANNEL hardware and optical fiber interconnections
- Enhanced degree of error detection

The MEMORY CHANNEL 2 functional/software enhancement goals were

- Software compatible with the first-generation MEMORY CHANNEL hardware
- Receive-side address remapping and variable page size to better support new operating systems, such as Windows NT, and non-Alpha microprocessors
- Remote read capabilities
- Global time synchronization mechanism
- Conditional write access to support a faster recoverable messaging

These two sets of requirements translate into architectural and technological constraints that define the MEMORY CHANNEL 2 design space. To increase the bisection bandwidth, the hub had to implement an architecture that supported concurrent transfers. On MEMORY CHANNEL 1, all senders must arbitrate for the same hub resource (the bus) on every data transfer. Every data transmission occupies the entire MEMORY CHANNEL 1 hub for the duration of its transfer, and all message filtering is performed by the receivers. Substantial network traffic causes congestion because all sender nodes fight for the same resource. This congestion results in a decrease in the communication speed and thus an increase in the effective q-ratio as seen by the applications.

On MEMORY CHANNEL 2, the hub has been designed as an *N*-by-*N* nonblocking full-duplex crossbar with broadcast capabilities, with $N = 8$ or $N = 16$. Such an architecture provides a bisection bandwidth that scales with the number of nodes and thus remains matched to the point-to-point bandwidth of the individual channels while avoiding congestion among independent communication paths. Therefore, an increase in network traffic will have little effect on the effective q-ratio.

The MEMORY CHANNEL ordering rules are easily met on a crossbar of this type, as follows:

1. The single-sender ordering rule is naturally obeyed by the fact that the architecture provides a single path from any source to any destination.
2. The multisender ordering rule is enforced by taking over all the crossbar routing resources during broadcast. Although less efficient than broadcast by packet replication, this technique ensures a strict common ordering for all destinations.

Finally, crossbar switches are practical to implement for a modest number of nodes (8 to 32), but given the availability of medium-size SMPs, they provide a satisfactory degree of scaling for the great majority of practical clustering applications. For instance, cluster technology can easily provide a 1,000-processor system simply by connecting 32 nodes, each one a 32-way SMP.

The requirement for a higher point-to-point bandwidth called for a shift from half-duplex to full-duplex links. A longer cable length imposed the choice of a signaling technique other than the TTL employed in the MEMORY CHANNEL 1 network. The design team adopted low-voltage differential signaling (LVDS)[26] as the signaling technique for the second and future generations of the MEMORY CHANNEL network on copper. One of the major decisions that faced the team was whether to maintain the parallel channel of MEMORY CHANNEL 1 or to adopt a serial channel to minimize skew transmission problems for large communication distances. The bandwidth demands of future cluster nodes indicated that serial links would not provide sufficient bandwidth expansion capabilities at reasonable cost. Thus, the channel data path width was chosen to be 16 bits, a suitable compromise that would offer a manageable channel-to-channel skew while providing the required bandwidth. Figure 3 illustrates the distinctions between the first- and second-generation MEMORY CHANNEL architectures.

MEMORY CHANNEL 2 Link Protocol

The MEMORY CHANNEL 2 communication protocol was engineered with the goal of ensuring compatibility with optical fiber's unidirectional medium. The interconnection substrate consists of a pair of unidirectional channels, one incoming and one outgoing. Each channel consists of a 16-bit data path, a framing signal, and a clock. The channel carries two types of packets: data and control. Data packets vary in size and carry application data. Control packets are used to exchange flow control, port state, and global clock information. Control packets take priority over data packets. They are inserted immediately when flow control state change is needed and, otherwise, are generated on a regular interval (millisecond) to update less time-critical state. The MEMORY CHANNEL 2 data packet format is shown in Figure 4a. The header of the data packet contains a packet type (TP), a destination identifier (DNID), a remote command (CMD), and a sender identifier (SID). The data payload starts with the destination address and can vary in length from 4 to 256 bytes (two to one hundred twenty-eight 16-bit cycles). It is followed by two 16-bit cycles of Reed-Solomon error detection code.

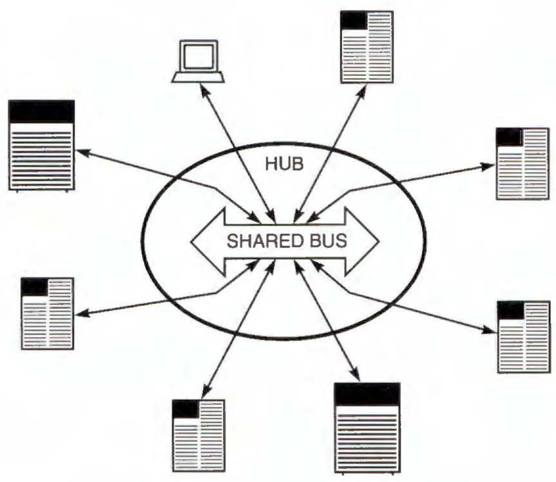
The control packet format is shown in Figure 4b. The packet is identified by a distinct TP and carries network and flow control information such as port status (PSTAT), configuration (CFG), DNID, hub status, and global status.

Similar to MEMORY CHANNEL 1, MEMORY CHANNEL 2 uses a clock-forwarding technique in which the transmit clock is sent along with the data and is used at the receiver to recover the data. Data is transmitted on both edges of the forwarded clock, and a novel dynamic retiming technique is used to synchronize the incoming packets to the node's local clock. The retiming circuit locks onto a good sample of the incoming data at the start of every packet and ensures accurate synchronization for the packet duration, as long as predefined conditions on maximum packet size and clock drifts are maintained.

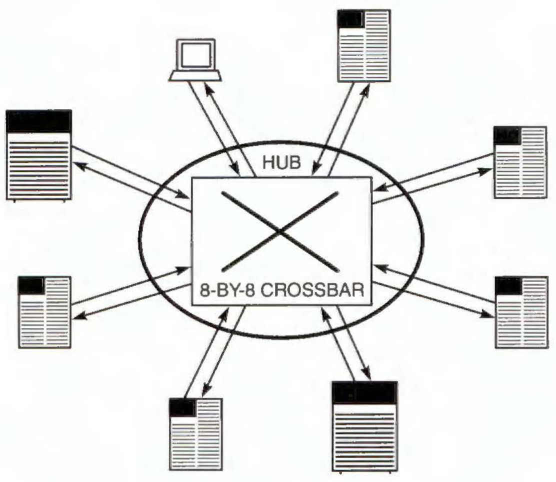
The MEMORY CHANNEL 2 link protocol has an embedded autoconfiguration mechanism that is invoked whenever a node goes on-line. The hub port and the adapter use this autoconfiguration mechanism to negotiate the mode of operation (link frequency, data path width, etc.). The same mechanism allows a two-node hubless system (a virtual hub configuration) to consistently assign node identifiers without any operator intervention or module jumpers.

MEMORY CHANNEL 2 Enhanced Software Support

MEMORY CHANNEL 2 provides four major additions to application and operating system support: (1) receive-side address remapping, (2) remote reads, (3) a global clock synchronization mechanism, and (4) conditional writes.



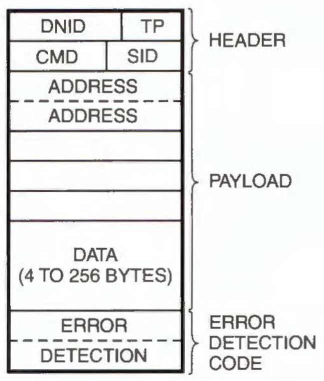
(a) MEMORY CHANNEL 1 Network



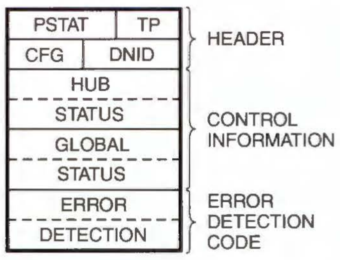
(b) MEMORY CHANNEL 2 Network

Characteristics	MEMORY CHANNEL 1	MEMORY CHANNEL 2
Channel data path width	37 bits	16 bits
Channel communication	Half duplex	Full duplex
Electrical signaling	TTL	LVDS
Optical fiber compatible	No	Yes
Link operating frequency	33 MHz	66 MHz
Peak raw data transfer rate	133 MB/s	133 + 133 MB/s
Sustained point-to-point bandwidth	66 MB/s	100 MB/s
Maximum packet size	32 bytes	256 bytes
Remote read support	No	Yes
Packet error detection	Horizontal and vertical parity	32-bit Reed-Solomon
Address space remapping	None	Receive
Supported page sizes	8 KB	4 KB and 8 KB
Hub architecture	Shared bus	Crossbar
Network bisection bandwidth	77 MB/s	800 to 1,600 MB/s

Figure 3
Comparison of First- and Second-generation MEMORY CHANNEL Architectures



(a) Data Packet



(b) Control Packet

Figure 4
MEMORY CHANNEL 2 Packet Format

On MEMORY CHANNEL 1 clusters, the network address is mapped to a local page of physical memory using remapping resources contained in the system's PCI-to-host memory bridge. All AlphaServer systems implement these remapping resources. Other systems, particularly those with 32-bit addresses, do not implement this PCI-to-host memory remapping resource. On MEMORY CHANNEL 2, software has the option to enable remapping in the receiver side of the MEMORY CHANNEL 2 adapter on a per-network-page basis. When configured for remapping, a section of the PCT is used to store the upper address bits needed to map any network page to any 32-bit address on the PCI bus. Such enhanced mapping capability will also be used to support remote access to PCI peripherals across the MEMORY CHANNEL network.

A simple remote read primitive was added to MEMORY CHANNEL 2 to support research into software-assisted shared memory. The primitive allows a node to complete a read request to another node without software intervention. It is implemented by a new remote read–on–write attribute in the receive page control table. The requesting node generates a write with the appropriate remote address (a read-request write). When the packet arrives at the receiver, its address maps in the PCT to a page marked as remote read. After remapping (if enabled), the address is converted to a PCI read command. The read data is returned as a MEMORY CHANNEL write to the same address as the original read-request write. Since read access to a page of memory in a remote node is provided by a unique network address, privileges to write or read cluster memory remain completely independent.

A global clock mechanism has been introduced to provide support for clusterwide synchronization. Global clocks, which are highly accurate, are extremely useful in many distributed applications, such as parallel databases or distributed debugging. The MEMORY CHANNEL 2 hub implements this global clock by periodically sending synchronization packets to all nodes in the cluster. The reception of such a pulse can be made to trigger an interrupt or, on future MEMORY CHANNEL–to–CPU direct-interface systems, may be used to update a local counter. The interrupt service software updates the offset between the local time and the global time. This synchronization mechanism allows a unique clusterwide time to be maintained with an accuracy equal to twice the range (max – min) of the MEMORY CHANNEL network latency, plus the interrupt service routine time.

Conditional write transactions have been introduced in MEMORY CHANNEL 2 to improve the speed of a recoverable messaging system. On MEMORY CHANNEL 1, the simplest implementation of general-purpose recoverable messaging requires a round-trip acknowledge delay to validate the message transfer, which adds to the communication latency. The MEMORY CHANNEL 2's newly introduced conditional write transaction provides a more efficient implementation that requires a single acknowledge packet, thus practically reducing the associated latency by more than a factor of two.

Memory Channel 2 Hardware

As suggested in the previous architectural description, MEMORY CHANNEL 2 hardware components are similar to those in MEMORY CHANNEL 1, namely a PCI adapter card (one per node), a cable, and a central hub.

The MEMORY CHANNEL 2 PCI Adapter Card The PCI adapter card is the hardware interface of a node to the MEMORY CHANNEL network. A block diagram of the adapter is shown in Figure 5. The adapter card is functionally partitioned into two subsystems: the PCI interface and the link interface. First in, first out (FIFO) queues are placed between the two subsystems. The PCI interface communicates with the host system, feeds the link interface with data packets to be sent, and forwards received packets on to the PCI bus. The link interface manages the link protocol and data flow: It formats data packets, generates control packets, and handles error code generation and detection. It also multiplexes the data path from the PCI format (32 bits at 33 megahertz [MHz]) to the link protocol (16 bits at 66 MHz). In addition, the link interface implements the conversion to and from LVDS signaling.

The transmit (TX) and receive (RX) data paths, both heavily pipelined, are kept completely separate from each other, and there is no resource conflict other than the PCI bus access. A special case occurs when a packet is received with the acknowledge request bit or the loopback bit set: the paths in both directions are coordinated to transmit back the response packet while still receiving the original one (employing the gray path in Figure 5). During a normal MEMORY CHANNEL 2 transaction, the transmit pipeline processes a transmit request from the PCI bus. The transmit PCT is addressed with a subset of the PCI address bits and is used to determine the intended destination of the packet and its attributes. The transmit pipeline feeds the link interface with data packets and appropriate commands through the transmit FIFO queue. The link interface formats the packets and sends them on the link cable. At the receiver, the link interface disassembles the packet in an intermediate format and stores it into the receive FIFO queue. The PCI interface performs a lookup in the

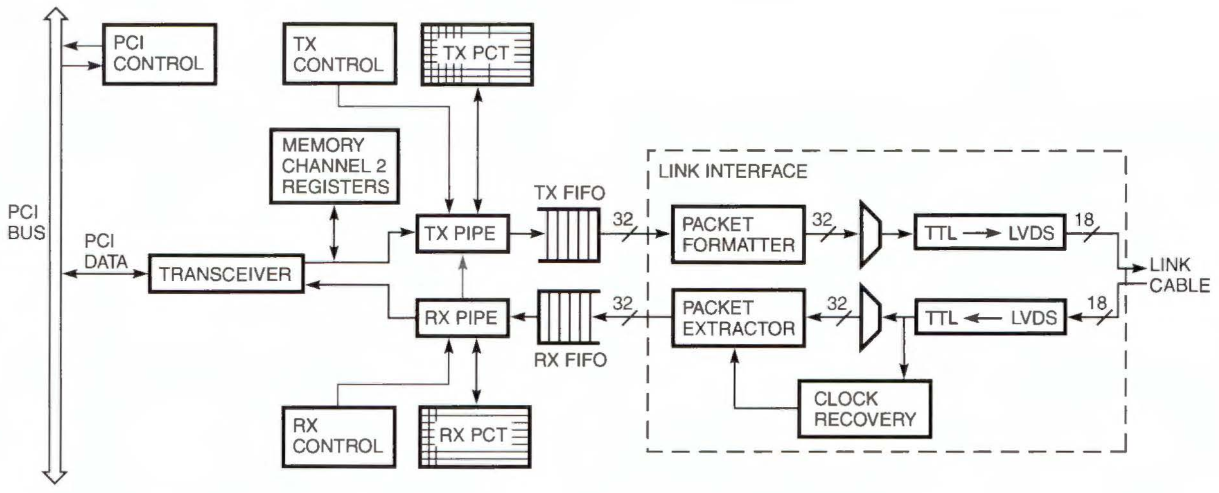


Figure 5
Block Diagram of a MEMORY CHANNEL 2 Adapter

receiver PCT to ensure that the page has been enabled for reception and to determine the local destination address.

In the simplest implementation, packets are subject to two store-and-forward delays—one on the transmit path and one on the receive path. Because of the atomicity of packets, the transmit path must wait for the last data word to be correctly taken in from the PCI bus before forwarding the packet to the link interface. The receive path experiences a delay because the error detection protocol requires the checking of the last cycle before the packet can be declared error-free. A set of control/status MEMORY CHANNEL 2 registers, addressable through the PCI, is used to set various modes of operation and to read local status of the link and global cluster status.

The MEMORY CHANNEL 2 Hub The hub is the central resource that interconnects all nodes to form a cluster. Figure 6 is a block diagram of an 8-by-8 MEMORY CHANNEL 2 hub. The hub implements a nonblocking 8-by-8 crossbar and interfaces to eight 16-bit-wide full-duplex links by means of a link interface similar to that used in the adapter. The actual crossbar has eight input ports and eight output ports, all 16 bits wide. Each output port has an 8-to-1 multiplexer, which is able to choose from one of eight input ports. Each multiplexer is controlled by a local arbiter, which is fed decoded destination requests from the eight input ports. The port arbitration is based on a fixed-priority, request-sampling algorithm. All requests that arrive within a sampling interval are considered of equal age and are serviced before any new requests. This algorithm, while not enforcing absolute arrival-time ordering among packets sent from different nodes, assures no starvation and a fair age-driven priority across sampling intervals.

When a broadcast request arrives at the hub, the otherwise independent arbiters synchronize themselves to transfer the broadcast packet. The arbiters wait for the completion of the packet currently being transferred, disable point-to-point arbitration, signal that they are ready for broadcast, and then wait for all other ports to arrive at the same synchronization point. Once all output ports are ready for broadcast, port 0 proceeds to read from the appropriate input port, and all other ports (including port 0) select the same input source. The maximum synchronization wait time, assuming no output queue blocking, is equal to the time it takes to transfer the largest size packets (256 bytes), about 4 μs, and is independent of the number of ports. As in any crossbar architecture with a single point of coherency, such broadcast operation is more costly than a point-to-point transfer. Our experience has been that some critical but relatively low-frequency operations (primarily fast locks) exploit the broadcast circuit.

MEMORY CHANNEL 2 Design Process and Physical Implementation

Figure 7 illustrates the main MEMORY CHANNEL physical components. As shown in Figure 7a, two-node clusters can be constructed by directly connecting two MEMORY CHANNEL PCI adapters and a cable. This configuration is called the virtual hub configuration. Figure 7b shows clusters interconnected by means of a hub.

The MEMORY CHANNEL adapter is implemented as a single PCI card. The hub consists of a mother-

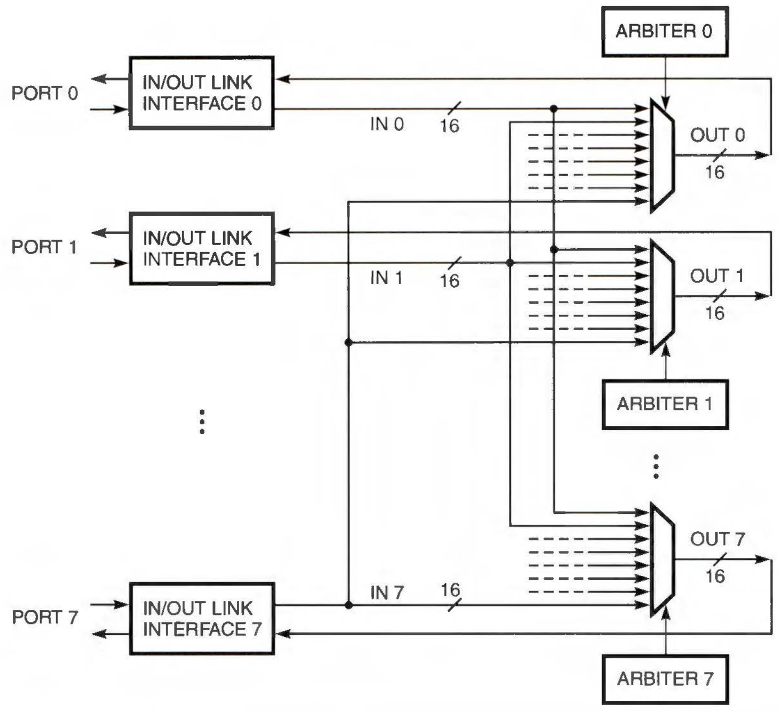
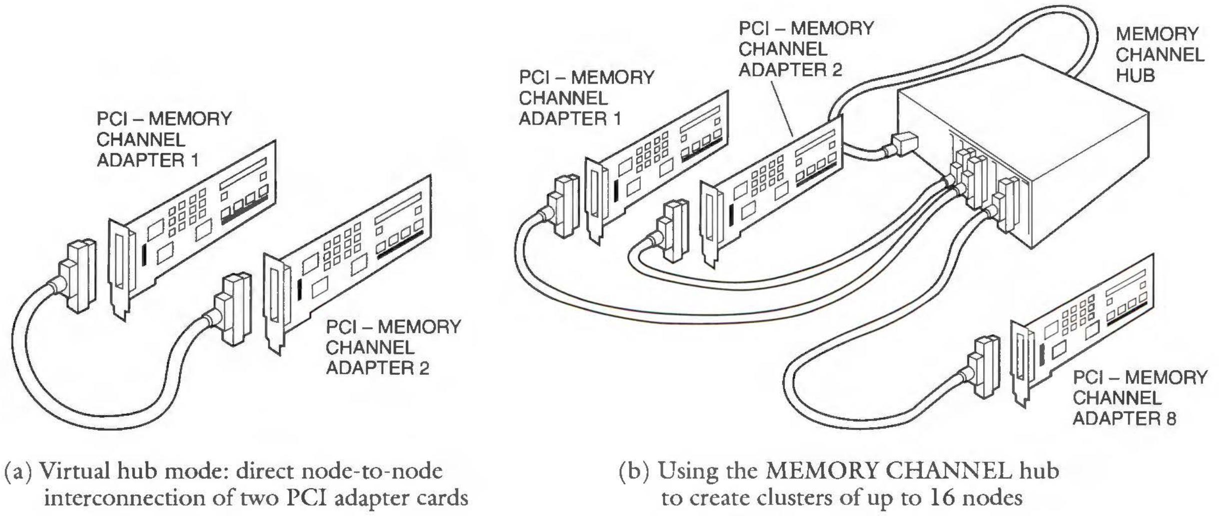


Figure 6
Block Diagram of an 8-by-8 MEMORY CHANNEL 2 Hub

board that holds the switch and a set of linecards, one per port, that provides the interface to the link cable.

The adapter and hub implementations use a combination of programmable logic devices and off-the-shelf components. This design was preferred to an application-specific integrated circuit (ASIC) implementation because of the short time-to-market requirements. In addition, some of the new functionality will evolve as software is modified to take advantage of the new features. The MEMORY CHANNEL 2 design was developed entirely in Verilog at the register transfer level (RTL). It was simulated using the Viewlogic VCS event-driven simulator and synthesized with the Synopsys tool. The resulting netlist


PCI – MEMORY CHANNEL ADAPTER 1
PCI – MEMORY CHANNEL ADAPTER 2
PCI – MEMORY CHANNEL ADAPTER 1
PCI – MEMORY CHANNEL ADAPTER 2
MEMORY CHANNEL HUB
PCI – MEMORY CHANNEL ADAPTER 8

(a) Virtual hub mode: direct node-to-node interconnection of two PCI adapter cards

(b) Using the MEMORY CHANNEL hub to create clusters of up to 16 nodes

Figure 7
MEMORY CHANNEL Hardware Components

was fed through the appropriate vendor tools for placing and routing to the specific devices. Once the device was routed, the vendor tools provided a gate-level Verilog netlist with timing information, which was then simulated to verify the correctness of the synthesized design. Boardwide static timing analysis was run using the Viewlogic MOTIVE tool. The link interface was fitted to a single Lucent Technologies Optimized Reconfigurable Cell Array (ORCA) Series field-programmable gate array (FPGA) device. The PCI interface was implemented with one ORCA FPGA device and several high-speed AMD programmable array logic devices (PALs). Thanks to the in-system programmability of PALs and FPGAs, the MEMORY CHANNEL 2 adapter board is designed to be completely reprogrammable in the field from the system console through the PCI interface.

MEMORY CHANNEL 2 Performance

This section presents MEMORY CHANNEL 2 performance data configured in virtual hub mode (direct node-to-node connection). Wherever possible actual measured results are presented. A two-node AlphaServer 4100 5/300 cluster was used for all hardware measurements.

Network Throughput

The MEMORY CHANNEL 2 network has a raw data rate of 2 bytes every 15 ns or 133.3 MB/s. Messages are packetized by the interface into one or more MEMORY CHANNEL packets. Packets with data payloads of 4 to 256 bytes are supported. Figure 8 compares, for various packet sizes, the maximum bandwidth the MEMORY CHANNEL 2 network is capable of sustaining with the effective process-to-process bandwidth achieved using a pair of AlphaServer 4100 systems. With 256-byte packets, MEMORY CHANNEL 2 achieves 127 MB/s or about 96 percent of the raw wire bandwidth.

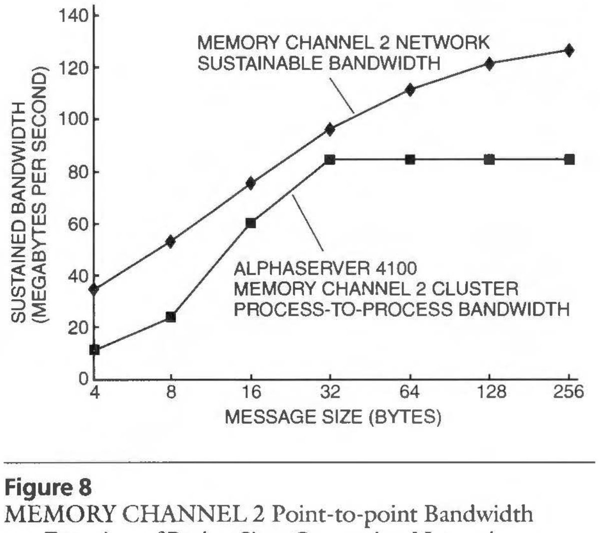


Figure 8
MEMORY CHANNEL 2 Point-to-point Bandwidth as a Function of Packet Size, Comparing Network Theoretical Limit and Sustained Process-to-process Measured Performance

For PCI writes of less than or equal to 256 bytes, the MEMORY CHANNEL 2 interface simply converts the PCI write to a similar-size MEMORY CHANNEL packet. The current design does not aggregate multiple PCI write transactions into a single MEMORY CHANNEL packet and automatically breaks PCI writes larger than 256 bytes into a sequence of 256-byte packets.

As Figure 8 shows, the bandwidth capability of the MEMORY CHANNEL 2 network exceeds the sustainable data rate of the AlphaServer 4100 system. The AlphaServer system is capable of generating 32-byte packets to the MEMORY CHANNEL 2 interface at 88 MB/s or about 10 percent less than the maximum network bandwidth at a 32-byte packet size. This represents a 33 percent bandwidth improvement over the previous-generation MEMORY CHANNEL, whose effective bandwidth was 66 MB/s. An ideal PCI host interface would achieve the full 97 MB/s, but the current AlphaServer 4100 design inserts an extra PCI stall cycle on sustained 32-byte writes to the PCI. The 32-byte packet size is a limitation of the Alpha 21164 microprocessor; future versions of the Alpha microprocessor will be able to generate larger writes to the PCI bus.

Latency

Figure 9 shows the latency contributions along a point-to-point path from a sending process on node 1 to a receiving process on node 2. Using a simple 8-byte ping-pong test, we determined that the one-way latency of this path is 2.17 μs. In the test, a user process on node 1 sends an 8-byte message to node 2. Node 2 is polling its memory waiting for the message. After node 2 sees the message, it sends a similar message back to node 1. (Node 1 started polling its memory after it sent the previous message.) One-way latency is calculated by dividing by two the time it takes to complete a ping-pong exchange. Approximately 330 ns elapse from the time a sending processor issues a store instruction until the store propagates to the sender's PCI bus. The latency from the sender's PCI to the receiver's PCI over the MEMORY CHANNEL 2 network is about 1.1 μs. Writing the main memory on the receiver node takes an additional 330 ns. Finally, the poll loop takes an average of about 400 ns to read the flag value from memory.

Table 2 shows the process-to-process one-way message latency for different types of communications

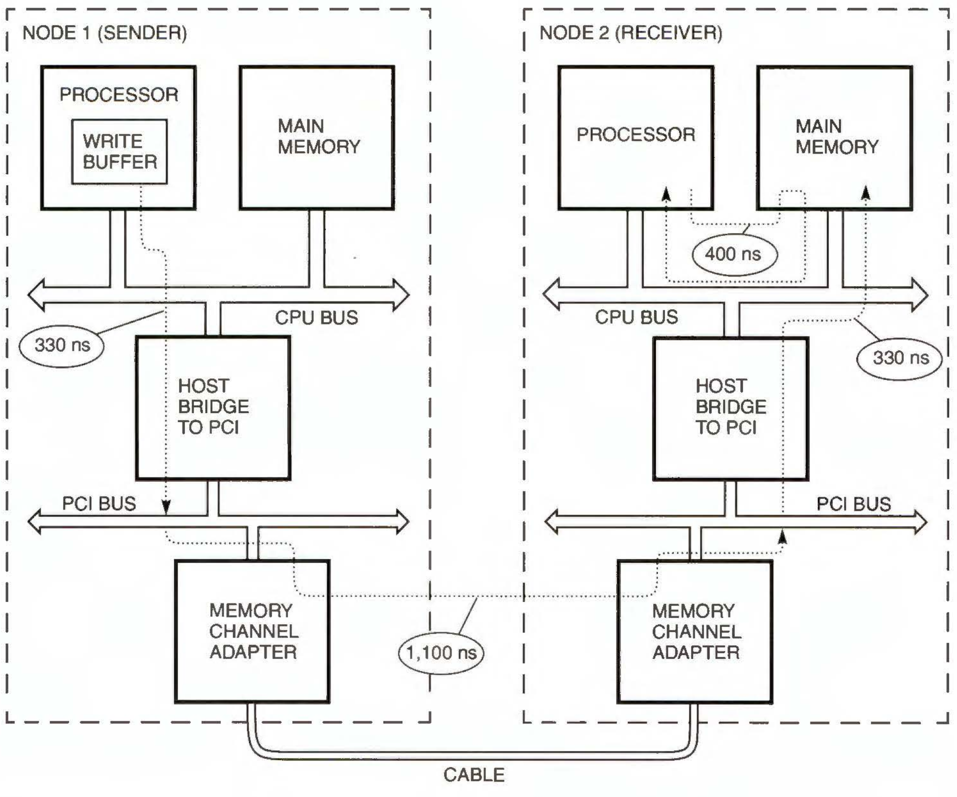


Figure 9
Latency Contributions along the Path from a Sender to a Receiver

at a fixed 8-byte message size. The first row contains the result of the ping-pong experiment previously described. For comparison, the previous generation of MEMORY CHANNEL had a ping-pong latency of 2.60 μs. The second row represents the latency for the simplest implementation of variable-length messaging. The latencies of standard communication interfaces are shown in the last two rows, namely, High Performance Fortran and Message Passing Interface. The results shown in this table are only between two and three times slower than the latencies measured for the same communication interfaces over the SMP bus of the AlphaServer 4100 system.

Table 2
MEMORY CHANNEL 2 One-way Message Latency in Virtual Hub Mode for Different Communication Interfaces

Communication Type	One-way Message Latency (Microseconds)
Ping-pong 8-byte message	2.17
8-byte message plus 8-byte flag	2.60
HPF 8-byte message	5.10
MPI 8-byte message	6.40

The latency of the MEMORY CHANNEL 2 network increases with the size of the message because of the presence of store-and-forward delays in the path. As discussed in the previous hardware description, all packets are subject to two store-and-forward delays, one on the outgoing buffer and one on the incoming buffer (required for error checking). These delays also play a role in the effective bandwidth of a stream of packets. On the one hand, smaller packets are less efficient than larger ones in term of overhead. On the other hand, smaller packets incur a shorter store-and-forward delay per packet, which can then be overlapped with the transfer of previous packets on the link, making the overall transfer more efficient. The hub performs cut-through packet routing with an additional delay of about 0.5 μs.

Summary and Future Work

This paper presents an overview of the second-generation MEMORY CHANNEL network, MEMORY CHANNEL 2. The rationale behind the major design decisions are discussed in light of the experience gained from MEMORY CHANNEL 1. A description of the MEMORY CHANNEL 2 hardware components led to the presentation of measured performance results.

Compared to other more traditional interconnection networks, MEMORY CHANNEL 1 provides unparalleled performance in terms of latency and bandwidth. MEMORY CHANNEL 2 further enhances performance by providing point-to-point bandwidth of 97 MB/s per second for 32-byte packets, an application-to-application latency of less than 2.2 microseconds, and a cross-section bandwidth of 1,000 MB/s for 8 nodes and 2,000 MB/s for 16 nodes. It also provides enhanced software support to improve the performance of the most common operations in a cluster environment, e.g., global synchronization, and reduces the complexity of the software layer by providing a more flexible address mapping. In addition, the MEMORY CHANNEL 2 network has been designed to be both hardware and software compatible with future generations on either copper or fiber-optic communication up to a distance of 3 kilometers. Future generations of the MEMORY CHANNEL architecture will benefit from the MEMORY CHANNEL 2 experience and will continue to provide enhancements to communication performance and to further refine those mechanisms introduced to support parallel cluster software.

Acknowledgments

Many thanks to Ed Benson for providing the detailed message-passing performance data contained in this paper. DIGITAL's MEMORY CHANNEL 2 team designed and implemented the second-generation MEMORY CHANNEL system described. Wayne Bortman, Robert Dickson, Marco Fillo, Richard Gillett, John Grooms, Michael McNamara, Jonathan Mooty, and Dave Pimm were the MEMORY CHANNEL 2 designers. Dale Keck, Edward Tulloch, and Ron Carn were responsible for the design verification. Brian McQuain was responsible for the printed circuit board layouts. Steve Campbell was the mechanical engineer on the hub enclosure. Special thanks go to John Grooms and Michael McNamara for their constructive comments and for proofreading the manuscript. The authors also thank the anonymous referees for their comments and suggestions, which considerably improved this paper.

References and Notes

1. G. Pfister, *In Search of Clusters: The Coming Battle in Lowly Parallel Computing* (Englewood Cliffs, N.J.: Prentice Hall, 1995).

2. M. Fillo, "Architectural Support for Scientific Applications on Multicomputers," *Series in Microelectronics,* vol. 27 (Konstanz, Germany: Hartung-Gorre Verlag, 1993).

3. D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods* (Englewood Cliffs, N.J.: Prentice Hall, 1989).

4. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal,* vol. 7, no. 3 (1995): 5–23.

5. R. Kaufmann and T. Reddin, "Digital's Clusters and Scientific Parallel Applications," *Proceedings of COMPCON '96,* San Jose, Calif. (February 1996).

6. M. Annaratone, C. Pommerell, and R. Ruehl, "Interprocessor Communication Speed and Performance in Distributed-Memory Parallel Processors," *Proceedings of the 16th International Symposium on Computer Architecture,* Jerusalem, Israel (May 1989).

7. C. Pommerell, M. Annaratone, and W. Fichtner, "A Set of New Mapping and Coloring Heuristics for Distributed Memory Parallel Processors," *SIAM Journal on Scientific and Statistical Computing* (January 1992).

8. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proceedings of the 17th International Symposium on Computer Architecture,* Seattle, Wash. (May 1990).

9. M. Fillo et al., "The M-Machine Multicomputer," *Proceedings of the XXVII Symposium on Microarchitecture,* Ann Arbor, Mich. (1995).

10. J. Dongarra, "Performance of Various Computers Using Standard Linear Equation Software," Technical Report CS-89-85 (Knoxville, Tenn.: University of Tennessee, Computer Science Department, December 19, 1996).

11. H. Cassanova, J. Dongarra, and W. Jiang, "The Performance of PVM on MPP Systems," Technical Report UT-CS-95-301 (Knoxville, Tenn.: University of Tennessee, Computer Science Department, 1995).

12. R. Gillett and R. Kaufmann, "Experience Using the First Generation Memory Channel for PCI Network," *Proceedings of the 4th Hot Interconnects Conference* (1996): 205–214.

13. R. Martin et al., "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proceedings of the 25th International Symposium on Computer Architecture* (May 1997): 85–97.

14. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proceedings of the 19th International Symposium on Computer Architecture,* Gold Coast, Australia (May 1992): 256–266.

15. K. Keeton, T. Anderson, and D. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Proceedings of Hot Interconnects III: A Symposium on High Performance Interconnects,* Stanford, Calif. (August 1995). Also available at http://http.cs.berkeley.edu/~kkeeton/Papers/paper.html.

16. R. Gillett, "Memory Channel Network for PCI," *IEEE Micro* (February 1996): 12–18.

17. J. Brosnan, J. Lawton, and T. Reddin, "A High-Performance PVM for Alpha Clusters," *Second European PVM Conference* (1995).

18. W. Gropp and E. Lusk, "The MPI Communication Library: Its Design and a Portable Implementation," http://www.mcs.anl.gov/Papers/Lusk/mississippi/paper.html (Argonne, Ill.: Mathematics and Computer Science Division, Argonne National Laboratory).

19. W. Cardoza, F. Glover, and W. Snaman, Jr., "Design of the TruCluster Multicomputer System for the Digital UNIX Environment," *Digital Technical Journal,* vol. 8, no. 1 (1996): 5–17.

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

21. M. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proceedings of the 21st International Symposium on Computer Architecture* (April 1994): 142–153.

22. R. Gillett, M. Collins, and D. Pimm, "Overview of Network Memory Channel for PCI," *Proceedings of COMPCON '96,* San Jose, Calif. (1996).

23. Information about the Scalable Coherent Interface is available at http://www.SCIzzL.com.

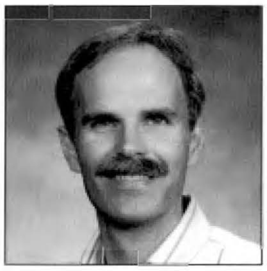
24. N. Boden et al., "Myrinet—A Gigabit-per-Second Local Area Network," *IEEE Micro,* vol. 15, no. 1 (February 1995): 29–36.

25. J. Lawton et al., "Building a High Performance Message Passing System for Memory Channel Clusters,"*Digital Technical Journal,* vol. 8, no. 2 (1996): 96–116.

26. IEEE Draft Standard for Low Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI). Draft IEEE P1596.3-1995.

Biographies

Marco Fillo
Marco Fillo is a principal engineer on the MEMORY CHANNEL 2 team in the AlphaServer Engineering Group. He is responsible for the design of the MEMORY CHANNEL 2 link protocol and hub. Before joining DIGITAL in September 1995, Marco held a position as research associate at M.I.T. in the Artificial Intelligence Laboratory, where he was one of the architects of the M-Machine, an experimental multithreaded parallel computer. Marco obtained a Ph.D. in electrical engineering from the Swiss Institute of Technology, Zurich, in 1993. He is a member of the IEEE and ACM, and his research interests are parallel computer architectures and interprocessor communication networks.



Richard B. Gillett
Rick Gillett is a corporate consulting engineer in Digital Equipment Corporation's AlphaServer Engineering Group, where he designs and develops custom VLSI chips, I/O systems, and SMP systems. As DIGITAL's parallel cluster architect, he defined and led the MEMORY CHANNEL project. He holds 17 patents on inventions in SMP architectures and high-performance communication and has patents pending on the MEMORY CHANNEL for PCI network. His primary interests are high-speed local and distributed shared-memory architectures. Rick has a B.S. in electrical engineering from the University of New Hampshire. He is a member of the IEEE and the IEEE Computer Society.

John H. Parodi
Fred W. Burgher

Integrating ObjectBroker and DCE Security

The integration of the ObjectBroker software product with the Distributed Computing Environment (DCE) Security Service makes ObjectBroker the most secure object request broker (ORB) in the industry. ObjectBroker and DCE Security together allow client-to-server, server-to-client, and mutual authentication. The integrated software provides these security functions, as well as message integrity protection, transparently to the applications. Integration has been accomplished in a way that allows plug-in replacement of the ObjectBroker security subsystem by DCE Security, Kerberos, or any third-party software security product that supports the DCE's Generic Security Service Application Programming Interface (GSS-API). This approach supports future GSS-API–compliant third-party security products based on Kerberos and also products that may address other security technologies such as biometrics and smart cards. In addition, the approach places responsibility for compliance with International Traffic in Arms Regulations in the hands of the purveyors and owners of GSS libraries rather than with the ORB vendor. Note that the ObjectBroker product is middleware jointly developed and distributed by DIGITAL and BEA Systems, who have formed a worldwide technology and distribution partnership.

An object request broker (ORB) is a distributed software layer that translates abstract service requests from a client application into requests for specific servers, regardless of where those servers actually reside on the network.[1] In this way, ORBs provide a middle tier in multitiered client-server systems. The ObjectBroker software, developed and distributed by strategic partners DIGITAL and BEA Systems, is an implementation of the Common Object Request Broker Architecture (CORBA) specified by the Object Management Group (OMG).[2]

Security is a growing concern for those who manage distributed computing systems, and the security options available to the CORBA community have been quite limited until recently. In the past year, OMG has adopted a specification for a CORBA Security Service, although few commercially available implementations exist at the time of this writing.

Outside the CORBA community, one widely accepted standard for security in distributed, heterogeneous systems is the Generic Security Service Application Programming Interface (GSS-API),[3,4] as specified by The Open Group (which was formed by the merger of the Open Software Foundation and X/Open Company Ltd.).[5] The GSS-API provides the ability for software entities in a distributed application to authenticate one another and to protect ongoing communication between them. The Distributed Computing Environment (DCE) Security Service provides an implementation of the GSS-API as one way to access its security services.

Plans are under way to implement the CORBA Security Service in the ObjectBroker software, but the implementation specifications were not complete when ObjectBroker version 2.6 was designed. At present, by integrating support for GSS-API implementations, the ObjectBroker software provides its customers state-of-the-art distributed system security with the widest choice of security technologies and products. The first commercially available GSS-API implementation was the Kerberos-based DCE Security Service itself, but other implementations, which use a variety of security technologies and are produced by various independent software vendors, are expected to follow soon.

Security

Ensuring secure communication among entities in a distributed computer system is a challenging task. The term security normally includes three broad classes of system requirements:[6]

1. Secrecy/privacy—the ability to protect information from unauthorized access
2. Integrity—the ability to protect information from unauthorized alteration or destruction
3. Availability—the ability to ensure that valid access to information can be accomplished in a timely manner

Enforcement of a security policy is accomplished by way of the following security functions:

- Authentication—the verification of the identity of a security principal
- Authorization—the determination of which principals can perform which actions
- Access control—the enforcement of the security policy, based on authentication and authorization information, to determine whether to allow or disallow a particular action

The Distributed Computing Environment

The Open Group's Distributed Computing Environment is an integrated, standard set of technologies, tools, and services that enables the development and deployment of distributed applications in a heterogeneous, multivendor computing environment.[7] Typically, system vendors implement the DCE on their own platforms. The DCE has been endorsed by virtually all system vendors, including IBM, HP, DIGITAL, NCR, Stratus, Cray, HAL, Hitachi, Siemens Nixdorf, NEC, Data General, Bull, Tandem, Transarc, SCO, Gradient, Siemens Pyramid, and Olivetti.

The DCE provides the following six technology components:

1. Remote Procedure Call (RPC), which facilitates distributed communication
2. Directory Service, which provides a single naming model throughout the distributed environment
3. Security Service, which provides reliable authentication, authorization, and data protection
4. Distributed Time Service, which synchronizes the network system clocks
5. Distributed File Service, which provides access to networkwide files
6. Threads Service (The DCE uses POSIX threads where available; on operating systems where POSIX is not available, the DCE supplies a threads package that provides the same interface as POSIX threads.)

DCE users can be characterized by their need to deploy and/or integrate large-scale applications on multiple heterogeneous platforms. The most common reasons given for choosing the DCE are its security features, its scalability, and its robustness.

DCE Security provides the following services:

- The DCE Authentication Service allows users and resources to prove their identity to each other. This service is currently based on Kerberos, which requires that all users and resources possess a secret key.
- The DCE Authorization Service verifies operations that users may perform on resources. A DCE Registry contains a list of valid users. An access control list associated with each resource determines valid users and the types of operations a user may perform.
- The DCE Data Integrity Service protects network data from tampering. Automatically generated cryptographic checksums are appended to network transmissions, allowing the DCE to determine if data has been corrupted in transit. The encrypted checksum is a message authentication code (MAC) based on the Data Encryption Standard (DES).

ObjectBroker uses the DCE Authentication and Data Integrity services.

ObjectBroker Security

Although DCE Security provides three basic levels of protection (None, Data Integrity, and Privacy), ObjectBroker uses only the Data Integrity level. This level provides a mechanism that computes an encrypted, time-stamped checksum and attaches it to the message so that any attempt to change or replay the information can be detected. In addition, ObjectBroker uses explicit calls to the DCE Security library's GSS-API to accomplish authentication but maintains its own access control lists and authorization database and mediates access control itself.[8]

Note that within a DCE cell, it is possible to use the DCE RPC with the DCE Security Service to protect communication at the wire protocol level. However, because ObjectBroker does not use the DCE RPC wire protocol, its use of the DCE Security Service is accomplished by means of explicit calls by ObjectBroker to the GSS-API implementation.

ObjectBroker's use of the DCE Security Service provides data integrity protection, authentication of clients to servers and servers to clients, and protection against replay and sequencing attacks. Although encryption is used to create the digital signatures that provide these protections at the network Data Integrity level, ObjectBroker does not directly support the capability to encrypt data, even on nodes that have Privacy-level DCE Security Service support. ObjectBroker provides no protection from denial of service attacks either.

Of course, a customer's use of DCE Security is entirely optional, and the security mechanism used in previous versions of the ObjectBroker software is still supported. With this mechanism, called trusted security, the node/username associated with a request from a remote node is accepted to be as claimed. For trusted security, ObjectBroker uses a proxy approach in which the node/username associated with a remote request is mapped to a proxy identity on the server's system. An access control decision is thus based on the authorization information for the proxy identity. The proxy approach to the trusted security mechanism was necessary because there was no concept of global identity for a user, that is, an identity known to all computer nodes in a distributed system.

To implement DCE Security on a particular platform, a Security Integration Architecture accomplishes the mapping of a globally understood username (e.g., a user or a security principal defined within a DCE cell or a Kerberos realm) to a login of a local user on a particular system. Some implementations of DCE Security and some systems (for example, the OpenVMS operating system) use the notion of integrated or global login, in which a local user login also causes a global user login to be performed. For the OpenVMS system, the global realm is the cluster. For the implementation of DCE Security on the DIGITAL UNIX system, the global realm is the DCE cell.

Because an ObjectBroker configuration can include platforms that have no implementation of the DCE, and because the Security Integration Architecture is different on every DCE platform, there was no common mechanism for ObjectBroker to use to implement an integrated global login across all supported platforms. Thus, ObjectBroker is limited by the integrated login capabilities available on other platforms' implementations of the DCE.

For this reason, ObjectBroker retains a proxy mechanism, even for use by nodes that support the DCE. For authentication between such nodes, a generic remote host definition (called SecGlobalName) is mapped to a local user on the local system. Should a server receive a request that requires authentication from a client node, the server uses SecGlobalName to attempt to match the corresponding global principal name to a local user name.

In other words, because there is no common global identity mechanism, ObjectBroker's proxy implementation maps either a trusted remote user or a global user identity to a local system identity to accomplish a generic mapping between global and local users. Rather than map multiple host/username pairs to the local proxy, the ObjectBroker software maps a single SecGlobalName, known to all nodes in the DCE cell, to that proxy whenever possible.

Mechanism for Global Authentication

The DCE Security Service provides the mechanism for global identity. The mechanism is based on Kerberos encryption, which is a private or symmetric key scheme (as opposed to a public or asymmetric key scheme). A private key scheme requires some trusted third-party node to act as a distribution center for encryption keys or credentials. Each node or user has a key that is known only to the user and the distribution center. In DCE Security, the distribution center is known as a privilege server.[9]

The following is a simplified description of the encryption key protocol between the privilege server and a client. The actual key exchange protocol, which uses three exchanges and conversion keys, results in a Privileged Access Certificate (PAC) in the possession of a client. The PAC, which is appended to each request, contains the authorization information to be compared with the access control information stored with the application server.

When a client wishes to communicate with a server, each must acquire a time-stamped session key for secure communication. The session key is protected in several ways. The time stamp means that the key is only valid for a limited time (the amount of time is configurable), which protects against brute-force attempts to break the key and reuse it. Also, each key is host-specific and can only be used from the node for which it is issued. Finally, the session key is never sent over the network in unencrypted form.

For a user to initiate a DCE_login, the client must enter its DCE_login password. To register as an initiator and acceptor of security contexts, a server uses a SERVTAB key file. This file contains an encrypted key that permits the server to obtain a set of credentials similar to those given to a user. These credentials allow the server to accept security contexts from clients or to initiate requests (that is, become a client) to other servers. The reason for having servers acquire credentials through the SERVTAB mechanism is that servers may be started on demand by the ObjectBroker Agent (the component that locates the appropriate server to satisfy a client request) or by system administrators who do not want to be burdened by having to know a server password.

In either case, the client or the server specifies the principal name to be authenticated. The node sends the specified principal's name to the privilege server. The privilege server returns a session key that is encrypted using the principal's password or SERVTAB key. The DCE run-time software running on the local system decrypts the session key using the password or SERVTAB key. Once the client and the server have decrypted session keys, they can use the keys to initiate secure communication with each other.

Thus, if a server is configured to require authentication, then before invoking a method on that server, a client must successfully perform a DCE_login and obtain the credentials needed to establish a security context with that server. A client may also require authentication from the server to ensure that some malicious software is not masquerading as a real server.

Note that the operations for acquiring credentials are accomplished outside the server executable. The operations are performed by the ObjectBroker runtime software, based on configuration settings in the ObjectBroker Security Registry. The goal is to avoid burdening applications with the knowledge of security mechanisms.

Authentication requirements can apply to the ObjectBroker Agent as well as to clients and servers. The Agent is in fact a separate security principal, and one can require client-to-Agent, Agent-to-client, Agent-to-server, and server-to-Agent authentication in an ObjectBroker configuration—in addition to authentication between the client and the server. The client or the server can independently set these modes, or the ObjectBroker system can require that modes be set nodewide.

Security Design Issues for ObjectBroker

The security issues associated with the design of ObjectBroker versions 2.6, 2.7, and 3.0 were primarily those of increasing the security capabilities and preserving upward compatibility with previous ObjectBroker versions. While compatibility is always a concern when upgrading software, ObjectBroker's requirements in this area are particularly stringent because customers have mission-critical applications running in very large configurations. In some cases, it is difficult or impossible to upgrade all ObjectBroker nodes at one time, so it must be possible to do a rolling upgrade that minimizes the disturbance to the configuration and allows uninterrupted operation of applications.

The need for dynamic, plug-in replaceability of the security subsystem was an important issue for two reasons. First, to provide standards-based solutions to computing problems, the ObjectBroker design had to allow the integration of any security product that implements the GSS-API. The second reason has to do with export controls.

United States government export regulations specify that hardware, software, and documentation for cryptographic products may be exported by license only. Specifically, the Department of State's International Traffic in Arms Regulations (22 Code of Federal Regulations Subchapter M) require that an export license be obtained from the department before any cryptographic hardware, software, or documentation is exported from the United States. An ObjectBroker design goal was not to encumber the product with export restrictions. Therefore, ObjectBroker itself does not include any cryptographic security mechanism. An ObjectBroker customer must provide an appropriate GSS library; whatever package is available on the system is the one ObjectBroker will use.

ObjectBroker Security Features

The security features that have been successfully implemented in the ObjectBroker software include

- Client-to-server, server-to-client, and mutual authentication
- Protection from replay and sequencing attacks and integrity protection
- Fine-grain control over the authentication mechanism (per-host, per-server, or per-method)
- Ability to demand a new security context for an invocation
- Ability to apply new security features to applications without rebuilding them
- Dynamically loadable security libraries

Usage

One of the most important characteristics of a secure ORB is that applications (clients and servers) need not be aware of security operations undertaken on their behalf. For ORBs, as well as for other support software, the goal is to avoid burdening applications with the need to deal with the complexities of a distributed system so that they can concentrate on the application problem at hand.

Therefore, most of ObjectBroker's security-relevant operations are invisible to applications. ObjectBroker's management utilities are used to specify the rules for authenticating clients and servers. These rules are stored in the ObjectBroker Security Registry, and the required authentications are performed automatically.

There are two exceptions to the general rule of keeping security operations invisible to the application. The first is that a client or a server (when acting as a client) can explicitly make a call to an ObjectBroker API to toggle mutual authentication on or off. This operation is allowed as long as it does not diminish the security level specified for the ObjectBroker node as a whole. In other words, a client can demand mutual authentication on a node that does not require such authentication but cannot disable mutual authentication if the node does require it. This feature was implemented to make it possible for clients to enable mutual authentication for specific operations that have security relevance.

The second exception is that a server can demand the creation of a new security context for an invocation, which immediately tests the authentication of the principal making the request. This is important because the GSS-API allows the initiation of a security context that has no expiration. Clearly, if a security context exists for a long enough period, there may be a concern that it is no longer valid. For example, when a user's account is revoked from the DCE Security Registry, it is possible that the user's credentials are still valid in some existing security context. Establishing a new security context forces the DCE run-time software to go back to the security server and verify the validity of the principal.

Figure 1 illustrates the interaction of ObjectBroker and the DCE Security Service components in the establishment of a security context. Once the security context is established, it is used in the verification of MAC-sealed messages between the server and the client. In this illustration, access to the DCE security subsystem is depicted as a local call, though accessing these services could also be done remotely.

The sequence of operations in Figure 1 is as follows:

1. A method invocation (a client request for a remote operation) results in a call to ObjectBroker's security subsystem.
2. The ObjectBroker security subsystem in turn invokes a GSS routine in the DCE Security library. This call determines whether a new security context needs to be established, which can happen for one of two reasons: either it is the first invocation of this server from this client or the context refresh rate has been specified as per-invocation.
3. The DCE Security library executes the call, which sets up the security context. (Note that the process of deleting an existing security context is not shown.)
4. The security subsystem checks the return status of the GSS routine to determine whether the resulting token is to be passed to the invocation layer.
5. If so, the token is passed to the transport layer for marshaling.
6. The client communicates with the server node through the normal ObjectBroker channel.
7. The transport layer in the receiving node unmarshals the message, examines the transport message header, and passes control to a dispatcher in the invocation layer.
8. Depending on the message type, the message may then be passed to a special dispatcher, in this case the security dispatcher in the security subsystem.
9. The security subsystem determines that the message should be handled by the GSS implementation and passes the message there.
10. The DCE Security layer checks the received token and if it is valid, accepts the security context. The routine generates a context establishment token to be passed to the client. The call also returns the server's context handle for the security context the server shares with the client.
11. The security layer passes the token to the invocation layer for marshaling.
12. The invocation layer marshals the information and sends it as an argument to the low-level transport routine call.
13. This message is sent to the client.
14. The data is unmarshaled.
15. The message is sent to the security subsystem.
16. The token is passed to the GSS implementation to initialize the security context, with the server-supplied token as an argument. The routine returns the client's context handle, which is used to sign subsequent messages.

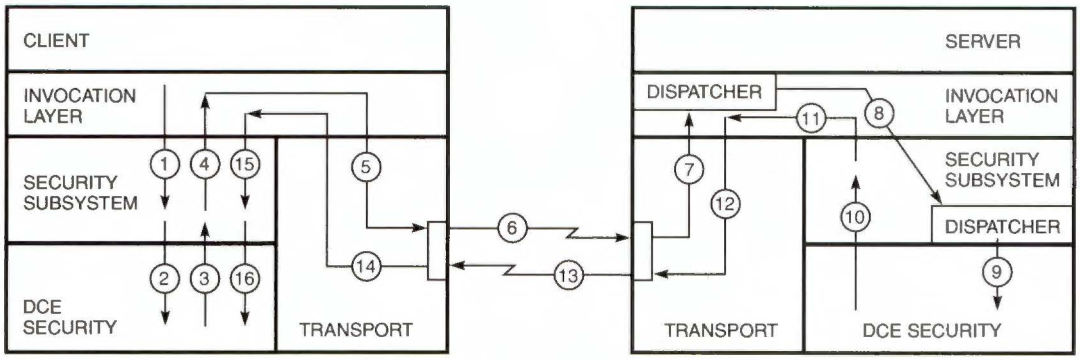


Figure 1
Establishment of a Security Context

Performance Considerations

The benefits of a secure ORB are not free. If authentication is required when a client and server establish a connection through a binding, part of that binding involves the establishment of a security context. Establishment of a security context requires a round-trip on the network, during which a token from the client is passed to the server, and a token is returned from the server to the client in the mutual authentication case.

Once established, the security context is used in subsequent requests (provided that the configuration does not require security context deletion after every method invocation). If the same security context is reused, the only additional overhead considerations are (1) the signing and verification of requests and responses in the client and server, and (2) the security context handle (32 additional bytes of information) appended to each message passed between the client and the server.

The signing and verification of a signature on a request or response is different from the verification of the privileges used when the security context is first set up, in that verification of a signature does not require a network round-trip. In contrast, when you first set up a security context, a network round-trip to the privilege server is required, and its overhead is significantly more costly than that of the verification and signature operations.

Note that when a client has multiple object references to a single method implementation in a server, a single security context can still be used. For example, a derived object reference does not require a new security context. This is both an optimization and a functional requirement, since only one security context is allowed between a client process and a server implementation.

Future Work

The OMG specifies a number of object services in addition to the CORBA specification itself. One of the most important specifications is for the CORBA Security Service. ObjectBroker's integration with DCE Security was designed and implemented before the OMG's CORBA Security Service specification was complete. Thus, even though ObjectBroker is the most secure ORB available today, it is reasonable to ask when and how its security features will be made compliant with the latest specifications from the OMG.

Given sufficient resources, ObjectBroker engineering could investigate supporting CORBA2 interoperability by implementing the OMG's General Inter-ORB Protocol (GIOP). The GIOP architecture supports both the Internet Inter-ORB Protocol (IIOP) and the DCE-based Common Inter-ORB Protocol (DCE-CIOP). Today, ObjectBroker uses a wire protocol based on the CORBA version 1.2 specification.

Security for the IIOP is governed by the Secure Inter-ORB Protocol (SECIOP) specification[10], although few commercially available implementations of the SECIOP are available at the time of this writing. Also, as mentioned previously, security for the DCE-CIOP is accomplished by protecting the RPC connections at the wire protocol level. For the DCE RPC, the DCE does its own authentication for RPC sessions; here the RPC connection between the client and the server, rather than the client and the server themselves, is authenticated. This approach provides the same potential for security management in the ORB configuration; it simply accomplishes the security functions at a level in the protocol stack that does not require the use of the GSS-API. By building in support for the GIOP, ObjectBroker gains the capability to provide the security features for both the IIOP and the DCE-CIOP protocols in future releases.

The SECIOP and the DCE-CIOP both follow the usage model of minimizing the need for applications to be aware of security. In the SECIOP, the OMG has specified APIs for security functions, and these functions are entirely separate from any mechanism that implements them. ORB vendors will be free to provide security features in much the same way that ObjectBroker provides security today, i.e., by working from security-related information kept by the ORB. The SECIOP also provides for administrative objects and operations that perform security management functions by means of APIs.

Conclusion

ObjectBroker provides state-of-the-art distributed system security today. Its security features provide upward compatibility, as well as the least possible disturbance to existing ObjectBroker applications and configurations. In addition, ObjectBroker's implementation of security by means of the DCE's Generic Security Service Application Programming Interface provides the greatest possible choice among security mechanisms and security implementation providers.

References and Notes

1. R. Otte, P. Patrick, and M. Roy, *Understanding CORBA* (Upper Saddle River, N.J.: Prentice Hall, 1996).

2. Information about the Object Management Group is available at http://www.omg.org.

3. J. Linn, *Generic Security Service Application Program Interface,* Internet RFC 1508, 1993.

4. J. Wray, *Generic Security Service API: Overview and C-bindings,* Internet RFC 1509, 1993.

5. Information about The Open Group is available at http://www.opengroup.org.

6. M. Gasser, *Building a Secure Computer System* (New York, N.Y.: Van Nostrand Reinhold, 1988).

7. *X/Open DCE: Authentication and Security Services,* X/Open Preliminary Specification P315, ISBN 1-85912-013-X, electronic version (Reading, U.K.: X/Open Company Limited, 1995).

8. *ObjectBroker—Designing and Building Applications,* Part No. AA-QX1LA-TK (Maynard, Mass.: Digital Equipment Corporation, 1996).

9. S. Miller, B. Neuman, J. Schiller, and J. Saltzer, *Kerberos Authentication and Authorization System* (Cambridge, Mass.: Massachusetts Institute of Technology, Project Athena, 1987).

10. *CORBA Security,* Document Number 95-12-01 (Framingham, Mass.: Object Management Group, 1995). The OMG members who contributed to the document were AT&T Global Information Solutions Co., Digital Equipment Corporation, Expersoft Corporation, Groupe Bull, Hewlett-Packard Company, International Business Machines Corporation (in collaboration with Taligent Inc.), International Computers Limited, Novell Inc., Siemens Nixdorf Informationssysteme AG, Sunsoft Inc., Tandem Computer Incorporated (in collaboration with Odyssey Research Associates Inc.), and Tivoli Systems Inc.

Biographies

John H. Parodi
John Parodi is a consulting technical writer in the Multiplatform Engineering group. His primary work involves customer communications and evangelism for object technology. In earlier work, John provided technical writing support for the Compound Document Architecture group and Architectural Engineering of Systems and Software Technical Office. John joined DIGITAL in 1979 after working in computer operations at Hendrix Electronics and at the University of New Hampshire. He has received two awards from the Society for Technical Communication and has more than 30 publications on various computer science topics, including compound documents, object technology, computer security, and BASIC.

Fred W. Burgher
Principal engineer Fred Burgher is employed by BEA Systems as a member of the ObjectBroker Engineering team. He is currently involved in ObjectBroker IIOP development. Previously, Fred worked at DIGITAL on integrating DCE Security and Naming for the OpenVMS operating system. Earlier in his career, he was employed as a principal engineer at Wang Laboratories, where he worked in the Imaging Engineering Group. Fred studied computer science at Boston University.

A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor

James Montanaro
Richard T. Witek
Krishna Anne
Andrew J. Black
Elizabeth M. Cooper
Daniel W. Dobberpuhl
Paul M. Donahue
Jim Eno
Gregory W. Hoeppner
David Kruckemyer
Thomas H. Lee
Peter C. M. Lin
Liam Madden
Daniel Murray
Mark H. Pearce
Sribalan Santhanam
Kathryn J. Snyder
Ray Stephany
Stephen C. Thierauf

This paper describes a 160 MHz 500 mW StrongARM microprocessor designed for low-power, low-cost applications. The chip implements the ARM V4 instruction set[1] and is bus compatible with earlier implementations. The pin interface runs at 3.3 V but the internal power supplies can vary from 1.5 to 2.2 V, providing various options to balance performance and power dissipation. At 160 MHz internal clock speed with a nominal Vdd of 1.65 V, it delivers 185 Dhrystone 2.1 MIPS while dissipating less than 450 mW. The range of operating points runs from 100 MHz at 1.65 V dissipating less than 300 mW to 200 MHz at 2.0 V for less than 900 mW. An on-chip PLL provides the internal clock based on a 3.68 MHz clock input. The chip contains 2.5 million transistors, 90% of which are in the two 16 kB caches. It is fabricated in a 0.35-μm three-metal CMOS process with 0.35 V thresholds and 0.25 μm effective channel lengths. The chip measures 7.8 mm × 6.4 mm and is packaged in a 144-pin plastic thin quad flat pack (TQFP) package.

Introduction

As personal digital assistants (PDA's) move into the next generation, there is an obvious need for additional processing power to enable new applications and improve existing ones. While enhanced functionality such as improved handwriting recognition, voice recognition, and speech synthesis are desirable, the size and weight limitations of PDA's require that microprocessors deliver this performance without consuming additional power. The microprocessor described in this paper—the Digital Equipment Corporation SA-110, the first microprocessor in the StrongARM family—directly addresses this need by delivering 185 Dhrystone 2.1 MIPS while dissipating less than 450 mW. This represents a significantly higher performance than is currently available at this power level.

CMOS Process Technology

The chip is fabricated in a 0.35 μm three-metal CMOS process with 0.35 V thresholds and 0.25 μm effective channel lengths. Process characteristics are shown in Table 1. The process is the result of several generations of development efforts directed toward high-performance microprocessors. It is identical to the one used in Digital Equipment Corporation's current generation of Alpha chips[2] except for the removal of the fourth layer of metal and the addition of a final nitride passivation required for plastic packaging.

The factors which drive process development for low-power design are similar to those which drive the process for pure high-performance although the motivation sometimes differs. For example, while both types of designs benefit from maximizing Idsat of the transistors at the lowest acceptable Vdd, the motivation for a pure high-performance design is reducing power distribution and thermal problems rather than extending battery life. Similar arguments apply to minimizing transistor leakage and on-chip variation of transistor parameters. This convergence of goals has been essential to our ability to develop one process to satisfy the requirements of both low-power and high-performance families.

Table 1
Process Features

Feature size	0.35 μm
Channel length	0.25 μm
Gate oxide	6.0 nm
Vtn/Vtp	0.35 V/-0.35 V
Power supply	2.0 V (nominal)
Substrate	P-epi with n-well
Salicide	Cobalt-disilicide in diffusions and gates
Metal 1	0.7 μm AlCu, 1.225 μm pitch (contacted)
Metal 2	0.7 μm AlCu, 1.225 μm pitch (contacted)
Metal 3	1.4 μm AlCu, 2.8 μm pitch (contacted)
RAM cell	6 transistor, 25.5 μm^2

Power Dissipation Tradeoffs

RISC microprocessors operating at 160 MHz are fairly common using current CMOS process technology. The novel aspect of this design is the ability to achieve this operating frequency at power levels which are low enough for handheld applications. Several design tradeoffs were made to achieve the desired power dissipation. In order to illustrate their effect on the design, it is interesting to imagine applying these tradeoffs to an earlier design whose power dissipation occupies the opposite end of the power spectrum, the first reported Alpha microprocessor.[3] This Alpha chip was fabricated in a 0.75-μm CMOS process and operated at 200 MHz dissipating 26 W at 3.45 V. The impact of these tradeoffs is summarized in Table 2.

The first decision is to reduce the internal power supply to 1.5 V. This change cuts the power by a factor of 5.3. While this has the desired effect, it has implications for the cycle time which are considered in the section Circuit Implementation.

The next step is to reduce the functionality. As compared to the early Alpha chip, the most obvious sections missing in this design are the floating point unit and the branch history table. Floating point is not required in the target applications and the low branch latency of this design eliminates the need for the branch history table. Less obvious, but very important, is reduced control complexity. This is a simple machine and we have worked hard to keep it so. We estimated that the reduced functionality would cut power by a factor of three.

Process scaling reduces node capacitances and therefore chip power. Note that although the area components of the capacitance will decrease as the square of the scale factor, the total capacitance change with scaling will be less dramatic primarily due to the effect of periphery capacitance. We estimate that scaling from 0.75 μm of the early Alpha chip to our current 0.35 μm process results in a power reduction of about a factor of two, a linear reduction with scale factor. Once again, coupled with this positive effect of process scaling are a host of other issues. Some of those issues are considered in the section Power Down Modes.

Next, consider the clock power. The clock power of the Alpha chips is fairly large and while that clocking strategy works well for Alpha machines, it is not appropriate for a low-power chip. Our clocking strategy and our latch circuits are described in some detail later. One major change from the Alpha design was to reject the pair of transparent latches per cycle used on the Alpha design. Instead, on this design, we switched to a single edge-triggered latch per cycle to reduce clock load and latch delay. Our estimate is that the changes in the clocking reduced the clock power by a factor of two. Since the clock power was about 65% of the total power on the first Alpha chip, this results in a reduction of about 1.3.

Finally, the reduction in clock frequency from 200 MHz to 160 MHz drops the power by 1.25.

Clearly, this analysis is not rigorous, but it suggests that it is reasonable to build a 160 MHz processor chip that dissipates around half a watt. A similar analysis was performed at the beginning of the project to select the power supply voltage and operating frequency and to determine whether significant changes in design method would be required to meet the performance and power goals. It is interesting to note that with the exception of the clocking changes, the design methods and philosophy used on this design were very similar to that used on the Alpha chips.

Table 2
Power Dissipation Tradeoffs

Start with Alpha 21064: 200 MHz @ 3.45 V. Power dissipation = 26W			
Vdd reduction:	Power reduction =	5.3x	⇛ 4.9 W
Reduce functions:	Power reduction =	3x	⇛ 1.6 W
Scale process:	Power reduction =	2x	⇛ 0.8 W
Reduce clock load:	Power reduction =	1.3x	⇛ 0.6 W
Reduce clock rate:	Power reduction =	1.25x	⇛ 0.5 W

Instruction Set

The microprocessor implements the ARM V4[1] instruction set. The architecture defines thirty 32-b general purpose registers and a program counter (PC). Registers are specified by a 4-b field where registers 0 to 14 are general purpose registers (GPR) and register 15 is the PC. The current processor status register contains a current mode field which selects either an unprivileged user mode or one of six privileged modes. The current mode selects which set of GPR's is visible.

In addition to basic RISC features of fixed length instructions and simple load/store architecture, the architecture implemented includes several features to improve code density. These include conditional execution of all instructions, load and store multiple instructions, auto-increment and auto-decrement for loads and stores, and a shift of one operand in every ALU operation. The architecture supports loads and stores of 8-, 16-, and 32-b data values. In addition to the standard 32-b computations, there is a 32-b × 32-b multiply accumulate with a 64-b product and accumulator.

Chip Microarchitecture

As shown in Figure 1, the chip is functionally partitioned into the following major sections: the instruction unit (IBOX), integer execution unit (EBOX), integer multiplier (MUL), memory management unit for data (DMMU), memory management unit for instructions (IMMU), write buffer (WB), bus interface unit (BIU), phase locked loop (PLL), and caches for data (Dcache) and instructions (Icache). To minimize pin power and support the high-speed internal core, one half of the chip area is devoted to the two 16 K caches. The pad ring occupies one-third of the chip area and the processor core fills the remaining one-sixth of the chip area.

The processor is a single issue design with a classic five-stage pipeline—Fetch, Issue, Execute, Buffer, and Register File Write (Figure 2). All arithmetic logic unit (ALU) results can be forwarded to the ALU input and there is a one-cycle bubble for dependent loads.

For example, the pipeline diagram in Figure 2 shows a SUBTRACT followed by a dependent LOAD. Note that at the end of cycle 3, we bypass the result from the SUBTRACT back into the ALU to compute the load address in cycle 4 without stalling the pipe.

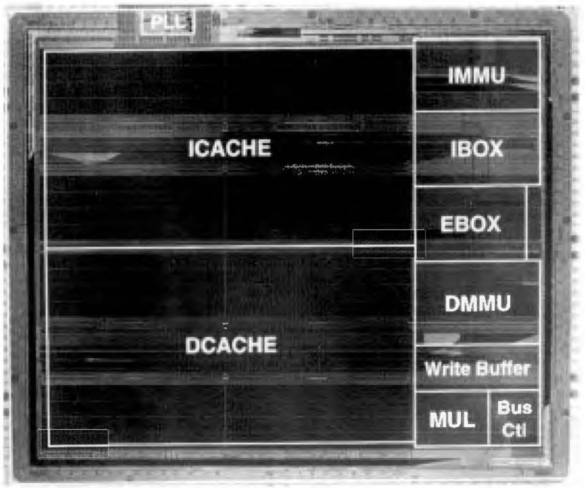


Figure 1
Chip Photo with Overlay

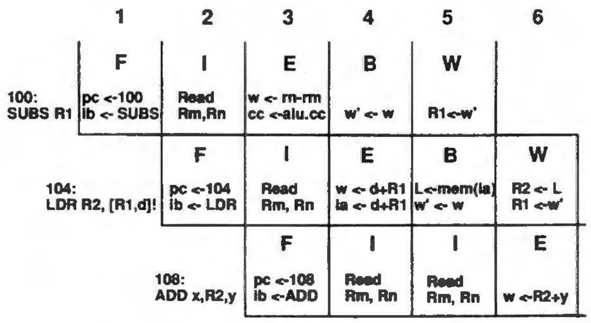


Figure 2
Basic Pipeline Diagram

The third instruction is an ADD which uses the result of the previous LOAD. The ADD is held in the Issue stage for one additional cycle until the LOAD data is available at the end of cycle 5.

The IBOX can resolve conditional branches in the Issue stage even when the condition codes are being updated in the current Execute cycle. By providing this optimized path, the IBOX incurs only a one-cycle penalty for branches taken, so the chip does not require branch prediction hardware. For example, in the pair of instructions shown in Figure 3, the BRANCH and LINK instruction at the (program counter) PC of 104 depends on the condition codes which are being generated by the SUBTRACT in the previous instruction. The condition codes from the Execute stage of the SUBTRACT are available at the end of cycle 3, in time to swing the PC multiplexer in the IBOX to point at the branch target PC during the next Fetch cycle.

The optimization of the branch path represents a power versus performance tradeoff in which perfor-

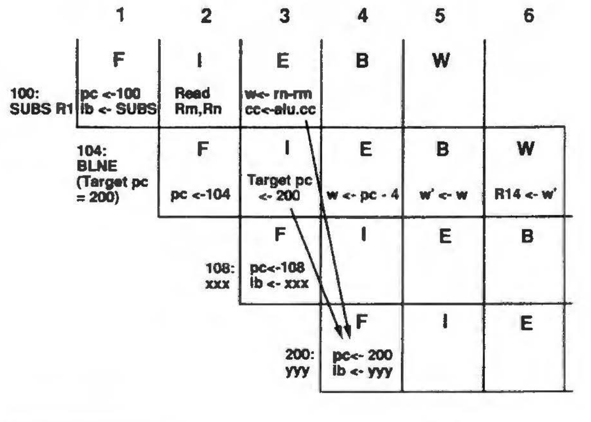


Figure 3
Pipeline Diagram of a Branch

mance won. In our effort to hold the one cycle branch penalty, we included a dedicated adder in the IBOX to calculate the branch target address and consumed additional power in the EBOX adder to meet the critical speed path to control the PC multiplexer. Due to critical path constraints, the adder in the IBOX must run every cycle, even if the instruction is not a branch.

In the early stage of the design, one of our concerns was whether the decision to pursue this optimized branch path would increase our cycle time. As the design turned out, our best efforts in this ALU path and in the cache access path resulted in nearly identical delays for these two longest critical speed paths.

Data for integer operations comes from a 31-entry register file with three read and two write ports. Sixteen of the registers are visible at any time with 15 additional shadow registers specified by the architecture to minimize the overhead associated with initiating exceptions. The EBOX contains an ALU with a full 32-b bidirectional shifter on one of the input operands. It includes bypassing circuitry to forward the data from the Dcache or the ALU output to any of the read ports. Figure 4 shows the circuit blocks involved in the branch path. The path starts at a latch in the bypassers and, in a single cycle, includes a 0-to 32-b shift, a 32-b ALU operation, and a condition code computation to swing the PC multiplexer for the next cycle. The registers to hold the condition codes were implemented in the EBOX so that this path could be locally optimized. Analysis of code traces indicated that most ALU operations included a shift of zero, so for this case, the shifter is disabled and bypassed to reduce power.

The EBOX also contains a 32-b multiply/accumulate unit. The multiplier consists of a 12- by 32-b carry-save multiplier array which is used for one to three cycles depending on the value of multiplicand and a 32-b final adder to reduce the carry-save result.

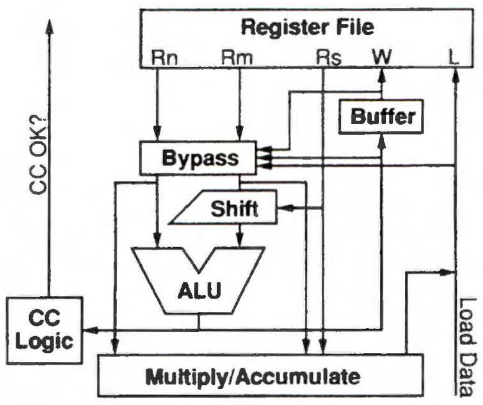


Figure 4
EBOX Block Diagram

For multiply accumulate operations, the accumulate value is inserted into the array so that an additional cycle is not required for the Multiplies with Accumulate. Multiply Long instructions require one additional cycle. This results in a MULTIPLY or MULTIPLY/ACCUMULATE in two to four cycles and MUL LONG or MUL LONG/ACCUMULATE in three to five cycles.

The Wallace tree implementation was chosen to minimize the delay through the array. This implementation required careful floor planning and custom layout to keep the wiring under control. The decision to perform 12 b of multiply per cycle was based on wiring tradeoffs made during the physical planning phase of the design rather than critical path concerns. When the multiplier is not in use, all clocks to the section stop and the input operands do not toggle.

The chip features separate 16 kByte, 32-way set associative virtual caches for instructions and data. Each cache is implemented as 16 fully associative blocks. Each cache is accessed in a single cycle for both reads and writes, providing a two-cycle latency for return data to the register file. One eighth of each cache is enabled for a cache access.

The Dcache is writeback with no write allocation. The block size is 32 bytes with dirty bits provided for each half block to minimize the data which needs to be castout in the event of a dirty victim. The physical address is stored with the data to avoid address translation during castouts.

Given the size of the caches and the low power target for the chip, it was important that we have fine granularity of bank selection. In addition, we required associativity of at least four-way for cache efficiency and it was important to performance that we maintain a single cycle access. We considered several solutions to this problem, including traditional four-way set associative caches, and decided that the simplest approach which satisfied all the requirements was to implement the caches as smaller, bank-addressed, fully associative caches. This resulted in 32-way associativity but this level of associativity was a side effect of the implementation used, not the result of a goal to get associativity significantly above four-way.

The chip includes separate memory management units (MMU) for instructions and data. Each MMU contains a 32-entry fully associative translation look-aside buffer (TLB) with entries which can map either 4 kB, 64 kB, or 1 MB pages. TLB fills are implemented in hardware. In addition to the standard memory management protection mechanisms, the ARM architecture defines an orthogonal memory protection scheme to allow the operating system easy access to large sections of memory without manipulating the page tables. This functionality requires a set of additional checks which must be performed after the TLB lookup. The resulting critical path was sufficiently long that we self-timed the RAM access in the TLB to allow us to perform the lookup and complex protection checks in a single cycle.

A write buffer with eight 16-byte entries handles stores and castouts from the Dcache. The write buffer includes a single-entry merge latch to pack up sequential stores to the same entry.

During normal operations, an external load request takes priority over stores on the pin bus. However, in the event of a load which hits in the write buffer, the chip executes a series of priority stores which raises the priority of the Write Buffer on the external bus above that of any loads. External stores occur and the write buffer empties until the store which was pending at the load address completes. At this point, top priority reverts back to loads.

Power Down Modes

There are two power down modes supported by the chip—Idle and Sleep.

Idle mode is intended for short periods of inactivity and is appropriate for situations in which rapid resumption of processing is required. In Idle mode, the on-chip PLL continues to run but the internal clock grid and the bus clock stop toggling. This eliminates most activity in the chip and the power dissipation drops from 450 mW to 20 mW. Return from Idle to normal mode is accomplished with essentially no delay by simply restarting the bus clock.

Sleep mode is designed for extended periods of inactivity which require the lowest power consumption. The current in Sleep mode is 50 μA which is achieved by turning off the internal power to the chip. The 3.3 V I/O circuitry remains powered and the chip is well behaved on the bus, maintaining specified levels if required by the drive enable inputs. Return from Sleep to normal operation takes approximately 140 μs.

As was noted earlier, a low voltage process is key to the design of a microprocessor which will run at 160 MHz while dissipating less than 450 mW. However, the same low device thresholds which allow the reduction of Vdd also result in significant device leakage. While this leakage is not large enough to cause a problem for normal operation, it does pose problems for standby current, especially if the process skews toward short channel devices. Our initial analysis indicated that the chip would dissipate over 100 mW in Idle mode with the clocks stopped. To reduce this leakage, we lengthened devices in the cache arrays, the pad drivers, and certain other areas. This brought the leakage power to within the required value of 20 mW in the fastest process corner. As a backup, we relaxed our design rules to allow the remaining gate regions, which are drawn with a standard 0.35 μm gate length, to be biased up algorithmically without violating design rules in case it was necessary to meet the leakage requirements.

The requirement for standby power in Sleep is more than two orders of magnitude lower than the Idle power. To meet the power limit in Sleep, we considered a variety of options including integrated power supply switches and substrate biasing schemes before choosing the simple approach of turning off the internal supply. This approach is reasonable for this generation of parts since they have a dedicated low voltage supply. As more parts of the system shift to the low voltage supply, this may no longer be acceptable. The conflicting requirements of high performance at low voltage and low standby current promise to create interesting challenges in future designs.

The power switch to turn off the internal power supply during Sleep is implemented off-chip as part of the power supply circuit for the low voltage supply. No state is stored internally during Sleep since in typical PDA systems, the Sleep state corresponds to the user turning the system off. Therefore the time associated with reloading the cache upon return from Sleep is acceptable.

The requirements in Idle and Sleep complicated the design of the bus interface circuits. This section includes the level-shifting interface between the internal low voltage (1.5 to 2.2 V) signals and the 3.3 V external pin bus. The bus interface circuits must drive and receive signals which are higher voltage than those nominally supported by the 0.35-μm process without using circuits which would cause us to exceed the current limit specified by the Idle spec. In addition, during Sleep the pads must be able to sustain the value on the output pins despite the loss of internal Vdd (Vddi) from the low voltage supply which is powered off by the system. The circuitry used to implement this function is shown in Figure 5.

Since Vddi will be driven to zero by the system during Sleep, it is used not only as a power supply but also as a logic signal. All circuitry which must be active in Sleep is driven from the external, 3.3 V supply (Vddx) which has been dropped through diode-connected PMOS devices to reduce the stress on the oxide of these devices. Before signaling the chip to enter Sleep, the system asserts the nRESET pin (active low) which drives all enabled outputs to a specified state—disabled for control signals and zero for addresses and data. It then asserts nPWRSLP (active low) which is ANDed with the appropriate output enable control to turn on small leaker devices which will hold the output pin in the appropriate state during Sleep. In the circuit shown in Figure 5, the output is an address. Therefore, the address bus enable (ABE) pin is the control pin on the lower NMOS leaker and a

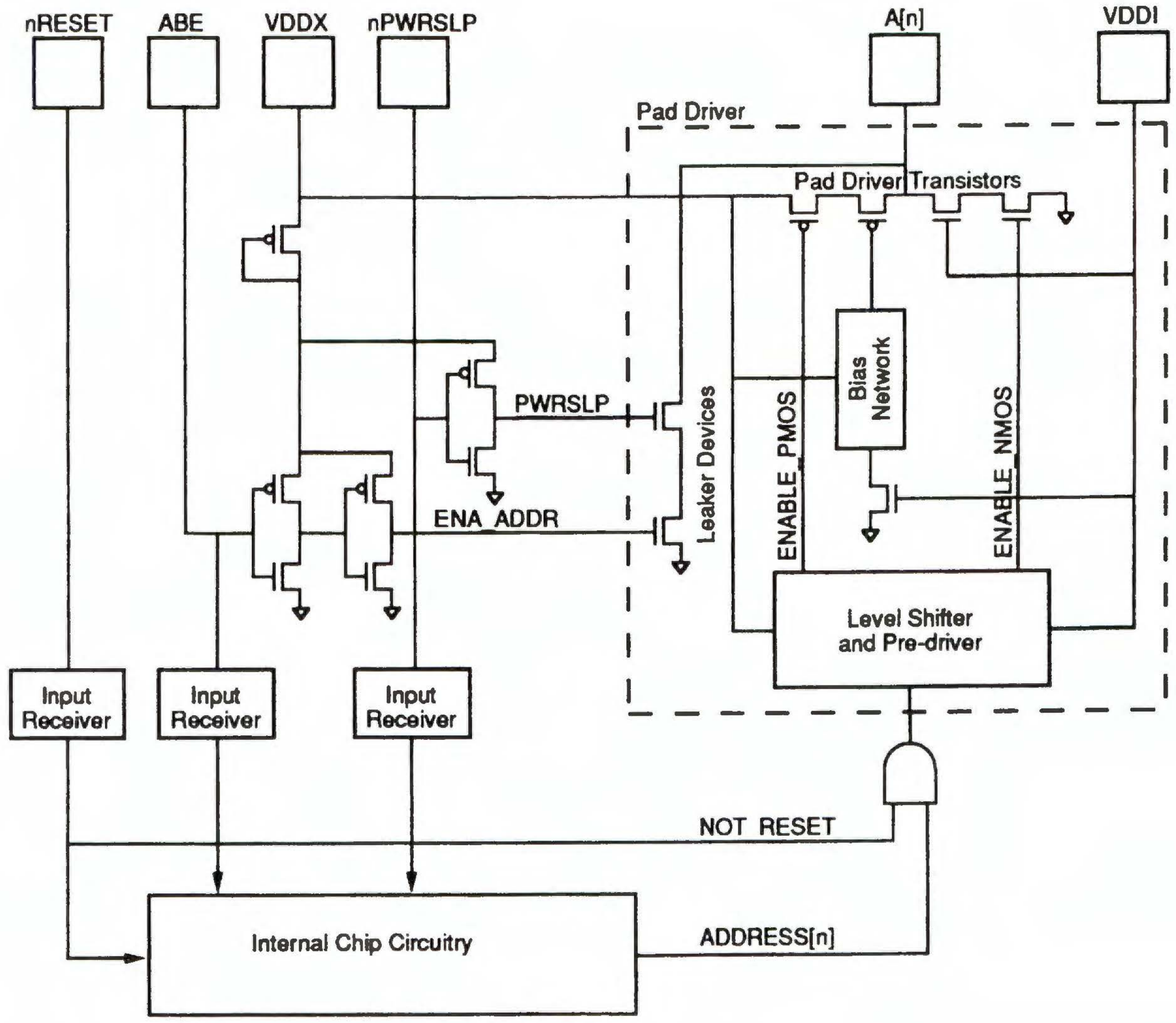


Figure 5
Pad Circuitry

buffered version of nPWRSLP controls the top device. Finally, the Vddi pins are actively driven to zero by the system. This action disables the output stage of the pad driver circuit by turning off the transistors closest to the pad—the NMOS directly and the PMOS via the bias network whose output goes to Vddx when its path to Vss is cut off. Note that for any input whose value is required during Sleep (ABE and nPWRSLP in the example described), a separate parallel input receiver must be implemented since the normal input receiver requires Vddi.

Circuit Implementation

The circuit implementation is pseudostatic and allows the internal clock to be stopped indefinitely in either state. Use of circuits which might limit low voltage operation was strictly controlled and the design was simulated to ensure operation significantly below the nominal 1.5 V level of the low voltage supply. The values of the internal supply and operating frequency were optimized to achieve maximum performance for less than half a watt.

The vast majority of the design is purely static, composed of either complementary CMOS gates or static differential logic. In certain situations, wide NOR functions were required and these were implemented in a pseudostatic fashion using either static weak feedback circuits or self-timed circuits to latch the output data and return the dynamic node to its precharged state.

The register file (RF) uses the self-timed approach to return the bit lines to the precharged state after an access (Figure 6). In this circuit, an extra self-timing column of bit cells with a dynamic bit line was implemented to mimic the timing of the data bit lines.

Figure 6 shows one cell from a column of register file data bit cells and one cell from the extra self-timing column (only one read port is shown). The bit cells in this extra column are all tied off so that the SELF_BITLINE signal will always discharge when the READ_WORDLINE goes high. When the SELF_BITLINE falls, it will set an RS latch causing the SELF_ENABLE signal to fall. This will disable the READ_WORDLINE and cause the bit lines to be precharged high when the read access is complete. Since the DATA_BITLINE's are received by low sensitive RS latches, the output data will be held when the bit line is precharged high. The self-timing RS latch is cleared when CLOCK_L goes low. This causes the SELF_ENABLE signal to go high, enabling the read port for the access in the next clock cycle. A separate SELF_BITLINE signal is implemented for each of the three register file ports so that the clocks for the three ports can be enabled independently.

The transistor leakage associated with the low threshold voltages is problematic for these pseudo-static circuits. If a weak feedback circuit is used in a NOR structure which is precharged high, excessive leakage in the parallel NMOS pulldowns would require that the feedback be fairly strong, which in turn would reduce the speed of the circuit. In the limit of very wide NOR's, it may not be possible to size a PMOS leaker so that it can supply the leakage of all the off NMOS pulldowns without making the leaker too large to be overpowered by a single active pulldown. In the case of a self-timed approach, a similar problem exists but it usually is manifested as a vanishingly small timing margin for the self-timed circuit to fire before the data on the dynamic node decays away. In either case, we addressed this issue by requiring the length of pulldowns on dynamic nodes to be slightly larger than minimum. Transistor leakage current is a strong function of channel length so a 12% increase in device length results in a leakage reduction in the worst case of about a factor of 20. The resulting leakage makes implementation of either weak feedback or a self-timed approach very reasonable.

The operating frequency at 1.5 V can be roughly derived by starting with the frequency of the Alpha

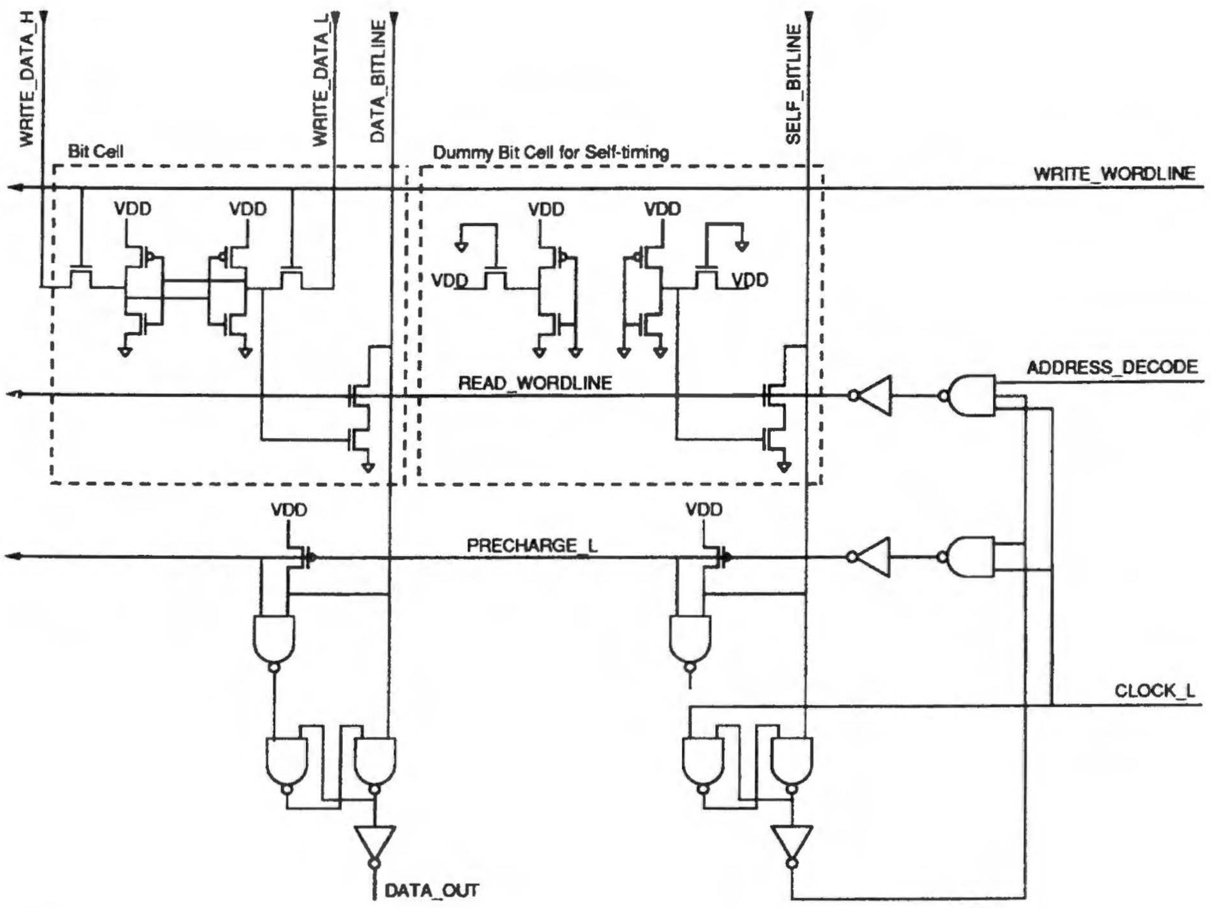


Figure 6
Self-timed RF Precharge

processor in the same process technology[2] and scaling for the use of a longer tick model and then Vdd. Since the long tick design requires the chip to perform a full SHIFT and a full ADD in a single cycle, this approximately doubles the cycle time required. The effect of Vdd scaling is roughly linear for this range of Vdd. Combining these effects results in an operating frequency at 1.5 V given by

$$433 \text{ MHz} * 0.5 * (1.5 \text{ V}/2.0 \text{ V}) = 162 \text{ MHz}.$$

This pair of voltage and frequency values agrees well with the power estimate outlined in the section Power Dissipation Tradeoffs. Note that for power supply voltages much lower than 1.5 V, the operating frequency decreases with voltage in a manner which is significantly stronger than linear. This fact sets a practical lower limit on the power supply voltage in most applications.

Power estimates made early in the design are prone to errors in either direction. In the case of this design, the power dissipated at 1.5 V was lower than the 450 mW target, so we shifted the nominal internal Vdd to 1.65 V to increase the yield in the 160 MHz bin.

Clock Generation

An on-chip PLL[4] generates the internal clock at one of 16 frequencies ranging from 88 to 287 MHz based on a fixed 3.68 MHz input clock. Due to internal resource constraints and our limited experience with low-power analog circuits, we contracted with Centre Suisse d'Electronique et de Microtechnique (CSEM) from Neuchâtel, Switzerland, to design the PLL and engaged Professor T. Lee from Stanford as a consultant on the project. Our initial feasibility work resulted in several design tradeoffs.

First, while there was a system requirement that the chip return quickly from the Idle state to normal operation, there was no such constraint on returning from the Sleep state. Based on this determination and our 20 mW power budget in Idle, we concluded that if we could keep the PLL power below 2 mW, we could leave the PLL running in Idle and remove the requirements on the PLL lock time. Thus, the need for a very low power PLL is dictated by the power budget in Idle, not in normal operation.

Next, we had specified a large frequency multiplication factor to allow the use of a common and cheap low frequency crystal clock source for consumer products. Early investigations indicated that this would make tight phase locking difficult. However, when we looked at target systems, we found no pressing need for phase locking. Consequently, we removed phase locking as a design criteria and concentrated our efforts and design tradeoffs on minimizing phase compression.

Finally, while the PLL was designed to handle the noise expected on the chip power supplies, we discovered toward the end of the design that the PLL was under its area budget and there was additional space available in the vicinity. We took advantage of this opportunity to provide cleaner power to the PLL by RC filtering our internal supply and we dedicated 1 nF of on-chip decoupling cap to this purpose.

CSEM performed the circuit and layout design and we placed the completed block into the microprocessor. Since we anticipated that the characterization of the PLL integrated in the chip would present some difficulties, we reserved one of the six die sites on our first pass reticle set for a test chip which contained several variants of the full PLL and interesting sub-blocks. These circuits allowed access to a variety of nodes in the PLL without compromising the design of the PLL instantiated in the chip. The results of the PLL characterization are reported in Reference 4.

Clock Distribution

The chip operates from two clocks as shown in Figure 7. An internal clock, called DCLK, is usually generated by the PLL. The second clock is a bus clock, known as MCLK which operates up to 66 MHz. MCLK can be supplied by an external asynchronous source or by the chip based on a division of the PLL clock signal.

There are five clock regimes in the chip. The first two regimes are sourced by MCLK and consist of the pad ring which receives MCLK directly and the bus interface unit (BIU) and part of the write buffer which receive MCLK through conditional clock buffers. The last three regimes are sourced by the internal DCLK clock tree and contain the Dcache, the Icache, and the

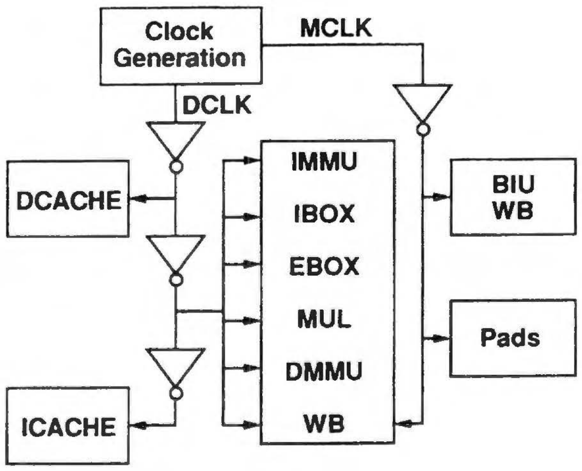


Figure 7
Clock Regimes

core. In this case, the core includes the IBOX, EBOX, MUL, IMMU, DMMU, and part of the write buffer.

Both MCLK and DCLK are distributed by buffered H-trees to conditional clock buffers in the various sections of the chip. The buffers in the H-tree allow the use of smaller lines for distribution and result in lower clock power. Although the three internal clock regimes are all sourced by the same H-tree, the topology of the chip did not allow corresponding sections of the H-tree to be routed in the same metal. This resulted in an increase in the expected skew between the caches and the core. In addition, we discovered that we could squeeze a bit more performance from the chip if we intentionally offset the clock in the caches relative to the clock in the core. Consequently, we used the clock buffers in the H-tree to tune the clock so that the Dcache receives a clock which is one gate delay earlier than the core and the Icache receives a clock which is one gate delay later than the core.

Figure 8 shows the physical routing of the internal clock tree. The buffer stages are not shown but they exist in the center of the chip and in four symmetric locations—two in the center of the I and D caches and two in locations at the cache/core interface. The final leg of the H-tree is tied to conditional clock buffers in the caches and the core. The problems associated with clock skew within the caches are reduced by the fact that only a single bank in each cache is enabled. This limits the physical distance over which tightly controlled clocks need to be delivered in the cache regions.

The clocks in the core present a more interesting problem. The final leg of the clock tree in the core stretches the full height of the chip and tight control of skew along this node is required for speed and functionality. It is implemented as a vertical, metal 2 line driven from four nominally equidistant points. The clock buffers are standard cells of varying drive strength built directly under this M2 line to minimize local variation in delay.

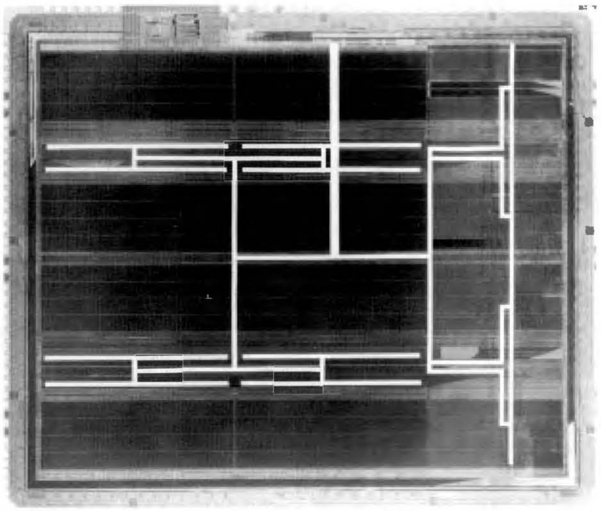


Figure 8
Physical Routing of Clock Tree

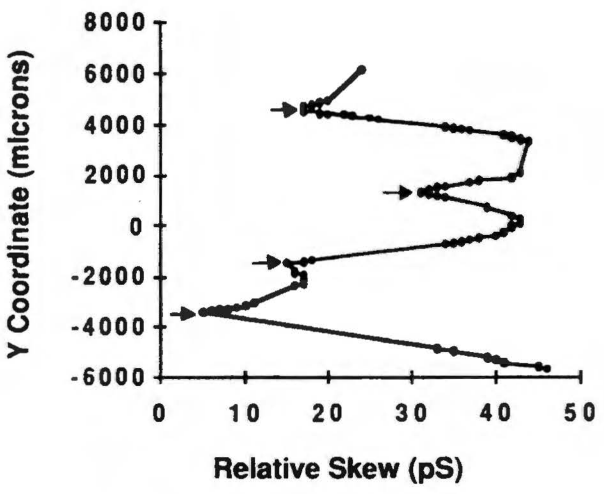


Figure 9
Clock Arrival Time in the Core

Circuit simulations of the H-tree were performed using SPICE to determine the skew between clock regions and within each of the clock regions. The nodes in the grid were extracted from layout and contained more than 30,000 R and C elements. Figure 9 shows the relative clock arrival time versus the Y coordinate for each conditional clock buffer on the vertical leg of the clock tree in the core. The four arrows on the graph indicate the points from which the final leg is driven. The data points are the relative arrival times of the clock input to the conditional clock buffers sourced by the clock tree. The total simulated skew is 41 pS assuming maximum metal resistance.

Clock Switching
One additional complication related to the internal clock tree is that it is not always driven by the clock from the PLL, known as CCLK. During cache fills, the clock source for the internal sections of the chip switches over to MCLK so that the whole chip is running synchronous to the bus (Figure 10). This simplifies fills and it reduces power since the bus clock is significantly slower than CCLK. Note that since this machine has a blocking cache, not much happens while waiting for a cache fill. Therefore, running on the slower bus clock during fills has essentially no performance impact.

Since MCLK and CCLK might be asynchronous, switching the driver of DCLK quickly between the two clock sources is difficult. Careful attention must be paid to the synchronization of the Mux control signals to prevent glitch pulses on the clock during the transition between the clock sources.

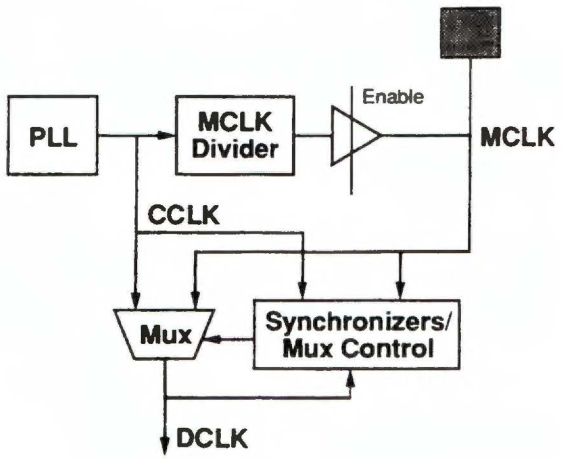


Figure 10
Clock Switching Circuit

Clock switching is only used during fills. Stores which miss in the cache and castouts are written to memory through the write buffer without switching the internal clock over to MCLK. The write buffer receives both DCLK and MCLK and passes the data for external stores across the DCLK/MCLK interface with proper attention to synchronization issues between the two clock regimes. One interesting characteristic of clock switching is that it gives the system designer another option to save power in situations for which the full performance of the chip is not required. By disabling clock switching on the fly, you can configure the chip to run off the bus clock. There is no limit on asymmetry or maximum pulse width of the bus clock, so the chip can be operated at very low frequencies if desired.

Conditional Clock Buffers

Conditional clock buffers are simple NAND/invert structures with an integral latch on the condition input. The buffers must be matched to their load to minimize skew. Since adding dummy clock loads is contrary to the low-power design philosophy, we created scaled clock buffers which would produce matched clocks for a wide range of loads and only needed to add dummy clock loads for a small number of very lightly loaded clock nodes. The task of matching the clock buffers to the load was greatly simplified by the fact the clock load presented by our standard latches is largely data-independent.

While the use of conditional clock buffers is central to the design method used on the chip, it should be noted that the critical paths to generate the condition input to these buffers represent some of the most difficult design problems in the chip. In this case, we decided that the power saving associated with the conditional clocking was worth the additional design effort and possible performance reduction.

Latch Circuits

The standard latches used in the design are differential edge-triggered latches (Figure 11). The circuit structure is a precharged differential sense amp followed by a pair of cross-coupled NAND gates. The sense amp need not be particularly well balanced because the inputs to the latch are full CMOS levels. The NMOS shorting device between nodes L3 and L4 provides a dc path to ground for leakage currents on nodes L1 and L2 in case the inputs to the latch switch after the latch evaluates. At normal operating frequencies, this device is not particularly important but it is required for the latch to be static. Note that since the dc current flowing is due only to device leakage, the magnitude of the current is insignificant to the power in normal operation.

Testability

The chip supports IEEE 1149.1 boundary scan for continuity testing. In addition, it has two hardware features to aid in manufacturing testing. The first is a bypass to allow CCLK to be driven from a pin synchronous to MCLK. This allows the tester to control the timing between CCLK and MCLK to make the asynchronous sections appear to be deterministic. The second test feature provides a linear feedback shift register (LFSR) that can be loaded with instruction data from the Icache. Loading the LFSR can be conditioned based on the value of address bit 2 and the Icache hit signal. The LFSR is loaded after the Fetch stage to allow the instruction following a branch to be read from the Icache and loaded into the LFSR. This feature allows any random pattern to be loaded into the

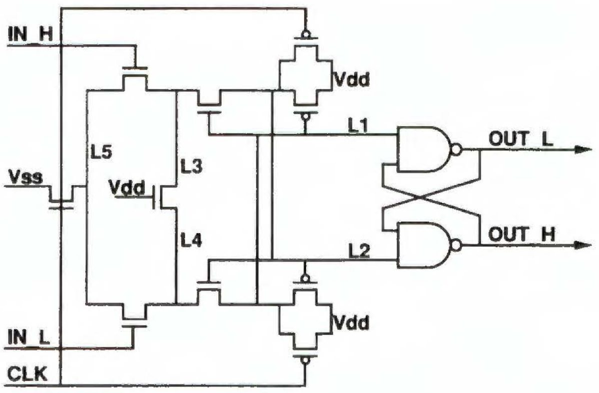


Figure 11
Latch Circuit

Icache and then read out by alternating branch instructions with data patterns words.

Power Dissipation Results

Measured Results

Power dissipation data was collected on an evaluation board running Dhrystone 2.1 with the bus clock running at one-third of the PLL clock frequency. Dhrystone fits entirely in the internal caches so, after the first pass through the loop, pin activity is limited. This is the highest power case because cache misses cause the internal clocks to run at the bus speed and result in a lower total power. For both sets of measurements, external Vdd is fixed at 3.3 V. For an internal Vdd of 1.5 V, the total power is 2.1 mW/MHz. If the internal supply is set to 2.0 V, the total power is 3.3 mW/MHz. Note that the ratio of the power at 1.5 and 2.0 V does not track Vdd^2 because it contains a component of external power and the external Vdd is fixed.

Simulated Power Dissipation by Section

An analysis of node transitions based on simulation was performed to estimate the power dissipation associated with the various major sections of the chip (Table 3). Toggle information was collected based on 160,000 cycles of Dhrystone and combined with extracted node capacitances to estimate power dissipation by node and this data was further grouped by section. The clock power listed in Table 3 is due only to the global clock circuits.

A few points are worth noting.

- First, the power is dominated by the caches as you might expect given their size. This is despite our efforts to reduce their power through bank selection and other means. The Icache burns more power than the Dcache because it runs every cycle.

Table 3
Simulated Power Dissipation by Section

ICACHE	27%
IBOX	18%
DCACHE	16%
CLOCK	10%
IMMU	9%
EBOX	8%
DMMU	8%
Write buffer	2%
Bus interface unit	2%
PLL	<1%

- Next, the PLL power is insignificant in normal operation. As was noted earlier, its low power characteristics are only important in Idle.
- Finally, since reduction in clock power was one of our explicit goals, it is interesting to consider the total clock power. If you extract the local clock power from the nonclock sections and sum it, you get a total clock power, including the global clock trees, the local clock buffers and the local clock loads. This power is 25% of the total chip power, significantly less than the 65% consumed by the clocks in the Alpha microprocessor used in our initial feasibility studies.

Conditional clocking was an integral part of the design method, so it is difficult to determine the power saving associated with it. However, the power associated with driving the conditional clocks is 15% of the chip power and if the conditions on all the conditional clock buffers were always true, this power would quadruple. This does not account for the additional power savings that has been achieved by blocking spurious data transitions.

CAD Tools

The CAD tools used on this chip were largely the same as those used on our Alpha designs.[5] This is not surprising since the performance target of the chip roughly parallels that of the Alpha family as noted in the section Circuit Implementation. The most significant departure was in the area of static timing verification and race analysis where the adoption of edge-triggered latching required significant modifications to the tools used in the Alpha designs.

Project Organization

One of the challenging aspects of this project was geographical. The detailed design was performed at four sites across a nine hour time zone range. The initial feasibility work and architectural definition was done at Digital Semiconductor's design center in Austin with on-site participation by personnel from Advanced RISC Machines Limited (ARM). The implementation was more widely distributed with the caches, MMU's, write buffer, and bus interface unit at Digital Semiconductor's design center in Palo Alto, the instruction unit, execution unit, and clocks in Austin, the pad driver and ESD protection circuits at Digital Semiconductor's main facility in Hudson, MA, and the PLL at the CSEM design center in Neuchâtel, Switzerland. In addition, we consulted with Hudson for CAD and process issues, with ARM in Cambridge, England, for all manner of architectural issues and implementation tradeoffs associated with ARM designs and with T. Lee from Stanford on the PLL. The implementation phase of the project took less than nine months with about 20 design engineers.

Conclusion

The microprocessor described uses traditional high performance custom circuit design, an intentionally simple architectural design, and advanced CMOS process technology to produce a 160 MHz microprocessor which dissipates less than 450 mW. The internal supplies can vary from 1.5 to 2.2 V while the pin interface runs at 3.3 V. The chip implements the ARM V4 instruction set and delivers 185 Dhrystone 2.1 MIPS at 160 MHz. The chip contains 2.5 million transistors and is fabricated in a 0.35-µm three-metal CMOS process. It measures 7.8 mm × 6.4 mm and is packaged in a 144-pin plastic thin quad flat pack (TQFP) package.

Acknowledgments

The authors would like to acknowledge the contributions of the following people:

F. Aires, M. Bazor, G. Cheney, K. Chui, M. Culbert, T. Daum, K. Fielding, J. Gee, J. Grodstein, L. Hall, J. Hancock, H. Horovitz, C. Houghton, L. Howarth, D. Jaggar, G. Joe, R. Kaye, J. Kapp, I. Kim, Y. Lou, S. Lum, D. Noorlag, L. O'Donnell, K. Patton, J. Reinschmidt, S. Roberts, A. Silveria, P. Skerry, D. Souyadalay, E. Supnet, L. Tran, D. Zoehrer, and the PLL design team at CSEM.

The support which they received on many aspects of the design from the people at Advanced RISC Machines, Ltd. was very important and keenly appreciated.

References

1. *ARM Architecture Reference* (Cambridge, England: Advanced RISC Machines, Ltd., 1995).

2. P. Gronowski et al., "A 433 MHz 64b Quad-Issue RISC Microprocessor," *ISSCC Digest of Technical Papers* (February, 1996): 222–223.

3. D. Dobberpuhl et al., "A 200 MHz 64b Dual-Issue CMOS Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11 (1992).

4. V. von Kaenel et al., "A 320 MHz, 1.5 mW CMOS PLL for Microprocessor Clock Generation," *ISSCC Digest of Technical Papers* (February, 1996): 132–133.

5. T. Fox, "The Design of High-Performance Microprocessors at Digital," *31st ACM/IEEE Design Automation Conference*, San Diego, Calif. (June 1994): 586–591.

Biographies

James Montanaro
James Montanaro received the B.S.E.E. and M.S.E.E. degree from the Massachusetts Institute of Technology, Cambridge, MA, in 1980. He joined Digital Equipment Corporation in 1982 and worked as a circuit designer on several RISC microprocessor chips including the first two Alpha designs. In 1992, he joined Apple Computer as a circuit designer on the PowerPC 603 chip. In 1993, he returned to Digital, working in the Austin Research and Design Center on the design of the first StrongARM microprocessor chip.

Richard T. Witek
Rich Witek received a B.S. in computer science from Aurora College, Aurora, IL, in 1976. He is the lead architect on the StrongARM microprocessors at Digital's Austin design center. He was co-architect of the Digital Alpha architecture and lead architect on the first Alpha microprocessor. Rich was one of the lead designers on the MicroVAX II microprocessor, the first single chip VAX. At Digital, Rich also worked on Phase 2 and Phase 3 DECnet architecture and implementation along with other PDP11 and VAX software projects. Rich was part of the Apple PowerPC architecture team at Somerset in Austin. His current professional interests include processor architecture and implementations. Rich has numerous patents and technical publications on microprocessors and caches.

Krishna Anne
Krishna Anne received the B.E. degree in electronics engineering in 1991 from Andhra University, Vizag, India, and the M.S.E.E. degree from the University of Texas at Arlington in 1993. After a brief stay at Tensleep Design, Inc., Austin, TX, in 1994, he joined Austin Research and Design Center of Digital Equipment Corporation as a design engineer responsible for the full-custom design and development of high-performance low-power processors. He worked on the design and implementation of the multiplier on the StrongARM project and is currently working on another low-power chip.

Andrew J. Black
Andy Black received a B.S.E.E. from Pennsylvania State University and an M.S.E.E. from the University of Southern California. He joined Digital in 1992 after working for International Solar Electric Technology. He was a senior hardware engineer in Digital's Palo Alto Design Center, where he led the bus interface unit design for the StrongARM SA-110 microprocessor chip. During his work on the Alpha 21164 CPU, he was a member of the design team for the memory management unit and contributed to the chip's clock design. He is currently with Silicon Graphics Inc. as a member of the technical staff in the MIPS Technology Division where he is working on high-performance consumer-oriented products. Andy is a member of I.E.E.E., Tau Beta Pi, and Eta Kappa Nu.

Elizabeth M. Cooper
Elizabeth Cooper received the B.S. degrees (summa cum laude) in electrical engineering and computer science from Washington University in St. Louis in 1985. She received the M.S. degree in computer science from Stanford University in 1995. She joined Digital Equipment Corporation in 1985. Her previous responsibilities include design contributions to several CMOS VAX and Alpha CPUs. She was responsible for the design of the memory management unit on the SA-110 StrongARM chip. She is currently employed at Silicon Graphics MIPS Technology Division.

Daniel W. Dobberpuhl
Daniel Dobberpuhl received the B.S.E.E. degree from the University of Illinois in 1967. He joined Digital Equipment Corporation in 1976 and has been responsible for five generations of microprocessor designs including the initial Alpha CPUs. Most recently he has been the Technical Director of the Low Power Microprocessor Group with Digital's Palo Alto Design Center. He is the co-author of *The Design and Analysis of VLSI Circuits* (Addison-Wesley, 1985).

Paul M. Donahue
Paul Donahue received the B.S. degree in computer science from Cornell University, Ithaca, NY, in 1994. Upon graduation he joined Digital Semiconductor's Palo Alto Design Center and worked on the SA-110. He is currently working on the microarchitecture and verification of a StrongARM variant.

Jim Eno
Jim Eno received the B.S.E.E degree from North Carolina State University, Raleigh, in 1989. He is employed as a senior engineer at Digital Equipment Corporation's Austin Research and Design Center in Austin, TX, working most recently on the microarchitecture of the SA-110 StrongARM microprocessor. Before his employment with Digital, he was with the Somerset Design Center in Austin, working on the microarchitecture and design of the PowerPC 603 microprocessor. Previous to this, Jim was involved in ASIC design support and tool development at Compaq Computer Corporation. His research interests include low-power microprocessor design and the propagation of acoustic waves in various materials, enhanced by interaction with selected organic compounds.

Gregory W. Hoeppner
Gregory Hoeppner graduated with distinction from Purdue University, West Lafayette, IN, in 1979. In 1980 he worked at General Telephone and Electronics Research Laboratory, Waltham, MA, performing basic properties research on GaAs. From 1981 to 1992 he held a number of positions with Digital Equipment Corporation, Hudson, MA, including CMOS process development, device characterization and modeling, circuit design, chip implementation, and finally co-led the 21064 Alpha chip implementation team. In 1992 he joined IBM's Advanced Workstation Division before returning to Digital Equipment Corporation in 1993 to co-found their Austin Research and Design Center, Austin, TX. Here he contributed to the microarchitecture, implementation and verification of Digital's first StrongARM processor.

David Kruckemyer
David Kruckemyer received the B.S. degree in computer engineering from the University of Illinois at Urbana-Champaign in 1993 and received the M.S. degree from Stanford University in 1995. After graduation, he joined Digital Equipment Corporation's Palo Alto Design Center to work on the implementation of the Instruction Memory Management Unit for the SA-110, the first StrongARM microprocessor. He is currently involved in the microarchitecture and implementation of a next-generation StrongARM variant.

Thomas H. Lee
Thomas Lee received the S.B., S.M., and Sc.D. degrees in electrical engineering, all from the Massachusetts Institute of Technology, Cambridge, MA, in 1983, 1985, and 1990, respectively. He joined Analog Devices in Wilmington, MA, in 1990 where he was primarily engaged in the design of high-speed clock recovery devices. In 1992, he joined Rambus, Inc. in Mountain View, CA, where he developed high-speed analog circuitry for 500 megabyte/s DRAMs. Since 1994, he has been an Assistant Professor of Electrical Engineering at Stanford University where his research interests are in low-power, high-speed analog circuits and systems, with a focus on gigahertz-speed wireless integrated circuits built in conventional silicon technologies, particularly CMOS. He has twice received the "Outstanding Paper" award at the International Solid-State Circuits Conference.

Peter C. M. Lin
Peter Lin was born in Taichung, Taiwan, on March 17, 1960. He received the B.S.E.E. degree from Feng Chia University, Taichung, Taiwan, in 1982 and the M.E. and E.E. degrees from University of Utah, Salt Lake City, in 1987 and 1989, respectively. From 1990 to 1993 he designed 2M VRAM and 8M WRAM for Samsung Semiconductor, San Jose, CA. From 1994 to 1995 he worked for Digital Equipment Corporation, Palo Alto, CA, where he contributed to the design of low power Alpha and StrongARM microprocessors. He is currently working for C-Cube Microsystems, Milpitas, CA. He holds one patent in output buffer design.

Liam Madden
Liam Madden received the B.E. degree from University College, Dublin, Ireland, in 1979 and the M.E. degree from Cornell University, Ithaca, NY, in 1990. Over the past 15 years he has designed CMOS CISC and RISC microprocessors, including the 21064 Alpha processor. He led the design team in Palo Alto which delivered the caches, write buffer, memory management, and bus interface units for the SA-110 StrongARM microprocessor. He is currently employed at Silicon Graphics, Mountain View, CA, where he is Director of Circuit Design and Technology.

Daniel Murray
Daniel Murray received the B.S. degree in electrical engineering in 1994 from the University of California, Berkeley. In 1994, he joined Digital Semiconductor's low power microprocessor group in Palo Alto, CA. He contributed as a circuit designer on the first StrongARM CPU and is currently involved in the implementation of another high-performance, low-power microprocessor.

Mark H. Pearce
Mark Pearce was born in Geneva, Switzerland, on June 12, 1969. He received the B.S.E.E. degree from University of Pennsylvania, Philadelphia, in 1992, and the M.S.E.E. degree from Stanford University, Stanford, CA, in 1994. In 1994 he joined Digital Equipment Corporation, at their Palo Alto Design Center, working initially on a low power Alpha processor prototype. He designed the write buffer on SA-110, the StrongARM processor. He is currently working on another high-performance, low-power processor.

Sribalan Santhanam
Sribalan Santhanam received the M.S.E. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1989. He joined Digital Equipment Corporation, in Hudson, MA, where he worked on the design of the floating-point unit of the 21064 CPU and subsequently on the design of the cache control unit of the Alpha 21164 CPU. He then moved to Digital's Palo Alto Design Center where he was responsible for the design of the caches for the SA-110 StrongARM microprocessor. He is currently a principal hardware engineer working on the implementation of a follow-on StrongARM microprocessor.

Kathryn J. Snyder
Kathryn Snyder (formerly Hoover) received the B.S. and M.S. degrees from the University of Michigan, Ann Arbor, in 1990 and 1992, respectively. She is a circuit designer with Digital Equipment Corporation working on low-power microprocessor designs in Austin, TX. She designed a variety of custom circuits for the SA-110 StrongARM microprocessor. Prior to employment with Digital, she worked for IBM in Austin, doing custom array design for PowerPC microprocessors.

Ray Stephany
Ray Stephany received the B.S.E.E. from Rensellaer Polytechnic Institute, Troy, NY, and an M.B.A. from Worcester Polytechnic Institute, Worcester, MA. He joined Digital's Austin Research and Design Center in July, 1993. Since that time, he has been one of the project leads on the StrongARM line of microprocessors. He has contributed to the development of low power circuit design techniques, CAD tools, verification, and overall methodology. He is currently leading the implementation of a next-generation StrongARM CPU and looking at SOI as a potential lower power process for future generations of microprocessors.

Stephen C. Thierauf
Stephen Thierauf is a consulting hardware engineer at Digital Equipment Corporation's Digital Semiconductor Group, located in Hudson, MA, and is responsible for I/O circuit design, on- and off-chip signal integrity, and I/O modeling for Alpha microprocessors, PCI peripherals, and other ULSI/VLSI devices. His previous work includes system level signal integrity analysis, micropackaging analysis and micropackaging design for numerous high-performance microprocessors and peripherals.

Referees, February 1995 to February 1997

The editors acknowledge and thank the referees who have participated in a peer review of the papers submitted for publication in the *Digital Technical Journal.* The referees' detailed reports have helped ensure that papers published in the *Journal* offer relevant and informative discussions of computer technologies and products. The referees are computer science and engineering professionals from academia and industry, including DIGITAL consulting engineers. Affiliations reflect referee status at the time of review. Note that independent consultants and DIGITAL employees are listed without company affiliation.

Mark R. Abbott, *Oregon State University*
Charles N. Abernethy
Jackie Albrecht, *Monitor Company*
Brian R. Allison
Dimitri A. Antoniadis, *Massachusetts Institute of Technology*
William Atkins, *Semiconductor Research Corporation*
Klaus J. Bachmann, *North Carolina State University*
Edward E. Balkovich
Prithviraj Banerjee, *University of Illinois at Urbana-Champaign*
Patrick Baudelaire
Carl J. Beckmann, *Dartmouth College*
Robert J. Bell
Walter Bender, *MIT Media Laboratory*
Anthony N. Berent
Kenneth P. Birman, *Cornell University*
Verell D. Boaen
Vladimir Bolkhovsky
Jean C. Bonney
V. Michael Bove, *MIT Media Laboratory*
William J. Bowhill
Scott O. Bradner, *Harvard University*
Mark Bramhall
Colin E. Brench
Karen Brouillette
Marc H. Brown
Stewart F. Bryant
David R. Butenhof
Fred C. Canter
Luca Cardelli
Wayne M. Cardoza
Donald R. Chand, *Bentley College*
J. Bradeley Chen, *Harvard University*
Peter M. Chen, *University of Michigan*
Wai-Mee Ching, *T. J. Watson Research Center*
James E. Chung, *Massachusetts Institute of Technology*
Matthew J. Conway, *University of Virginia*
W. Bruce Croft, *University of Massachusetts Amherst*
Christopher L. Cromer, *NIST*
Mark E. Crovella, *Boston University*
Zarka Cvetanovic
David Cyganski, *Worcester Polytechnic Institute*
Nathaniel J. Davis IV, *Virginia Tech*
John DeTreville
David J. DeWitt, *University of Wisconsin*
John C. Eck
John C. Egolf
Stephen G. Eick, *AT&T Bell Laboratories*
John Ellenberger
David C. Ellis
Joel S. Emer
Nicholas Emery
William E. Farrell, *Science Application International Corporation*
W. Burns Fisher
Jose A. B. Fortes, *Purdue University*
Tryggve Fossum
Michael J. Franklin, *University of Maryland*
Ko Fujimura, *NTT Information and Communications Laboratories*
Bruce Gitton, *Monterey Bay Aquarium Research Institute*
Michael Glantz, *Rank Xerox Research Centre, Grenoble*
William Goldenthal
Paul M. Goodwin
James F. Grochmal
Greg J. Grula
Dirk Grunwald, *University of Colorado*
Jonathan Harris
Jeffrey R. Harrow
Paul K. Harter
Mark D. Hayter
Denise Heagerty, *CERN*
George T. Heineman, *Columbia University*
Daniel Herr, *Semiconductor Research Corporation*
F. S. (Sandy) Hill, *University of Massachusetts Amherst*
Stephen R. Hoffman
Timothy A. Howes, *University of Michigan*
Henry G. Jakiela
Allan L. Jennings
Christopher F. Joerg

Douglas W. Jones, *University of Iowa*
Richard S. Kaufmann
James W. Keeley
Keith A. Kimball
James Jay Kistler
Wilfred L. Kling
Charles Koelbel, *Rice University*
Vijaya K. Konangi, *Cleveland State University*
Thomas E. Kopec
Nancy P. Kronenberg, *Avid Technology, Inc.*
Charles D. Kukla
Riva Ladkin
William A. Laing
Richard F. Lary
Mark E. Law, *University of Florida*
Alvin R. Lebeck, *Duke Universtiy*
Michael Lee, *Open Engineering Inc.*
Yann-Hang Lee, *University of Florida*
Robert D. Lembree
William H. Lenharth, *University of New Hampshire*
Norbert Leser, *The Open Group*
Donald M. Leskiw, *Syracuse University*
Roy Levin
Michael Levine, *Pittsburgh Supercomputing Center, Carnegie Mellon University*
Thomas D. Little, *Boston University*
David B. Lomet, *Microsoft Corporation*
Paula Long
P. Geoffrey Lowney
Mark W. Maier, *University of Alabama in Huntsville*
François Martzloff, *NIST*
Barry A. Maskas
Alan L. Matthews, *Trevecca Nazarene College*
Robert N. Mayo
Paul R. McJones
William M. McKeeman
John Mellor-Crummey, *Rice University*
Guiseppe Menga, *Politecnico di Torino, Dipartimento di Automatica e Informatica*
Scott F. Midkiff, *Virginia Tech*
Tom Miller, *Microsoft Corporation*
Jeffrey C. Mogel
Charles Robert Morgan
Ethan V. Munson, *University of Wisconsin*
Andre I. Nasr
Charles Gregory Nelson
Alan G. Nemeth
William G. Nichols
Nigel Norris
William B. Noyce
David R. Oran
Ricky S. Palmer
Sharon E. Perl
Mark Pesce, *Enterprise Integration Technologies*
Russell W. Quong, *Purdue University*
Mustafizur Rahman
T. V. Raman
Satish L. Rege
Steven K. Reinhardt, *University of Wisconsin*
Steven P. Reiss, *Brown University*
Llanda M. Richardson
Paul I. Rubinfeld
Alexander I. Rudnicky, *Carnegie Mellon University*
Joel H. Saltz, *University of Maryland*
Daniel Scales
Christopher Schmandt, *MIT Media Laboratory*
Michael D. Schroeder
Wayne Schroeder, *San Diego Supercomputer Center*
Robert W. Seidel, *Charles Babbage Institute*
Margo Seltzer, *Harvard University*
I. Michael C. Shand
John Shen, *Carnegie Mellon University*
Adam Shepela
Will H. Sherwood
Jieh-Hwa Shyu, *Millipore Corporation*
Robert J. Simcoe
Allen K. Simons
Michael D. Smith, *Harvard University*
Thomas R. Smith III
Robert J. Souza
Amitabh Srivastava
Simon C. Steely
Brian M. Stevens
Richard E. Stockdale
Alan L. Sussman, *University of Maryland*
Mark Swartout
Thomas A. Sweeney
Mark W. Sylor
Daniel Tabak, *George Mason University*
Owen H. Tallman
Charles P. Thacker
Kurt M. Thaller
Chandramohan A. Thekkath
David W. Thiel
Carl V. Thompson, *Massachusetts Institute of Technology*
Leo P. Treggiari
Jonathan S. Turner, *Washington University*
Reha M. Uzsoy, *Purdue University*
Edward F. Vogel
Theodore V. Vorburger, *NIST*
Richard F. Walters, *Univerisity of California, Davis*
Keith Waters
William Weihl
Thomas M. Wenners
Stanley J. Whitlock
John C. S. Whytock, *BAeSENA Ltd.*
Rebecca Will
Douglas D. Williams
David A. Wood, *University of Wisconsin*

Call for Papers Programming Languages, Tools, and Technologies

The *Digital Technical Journal* seeks technical papers in all areas of programming languages and tools for an issue to be published in the fall of 1998. DIGITAL engineers and industry partners interested in participating in the special issue should send topics and brief abstracts (100 words) by December 12, 1997, to

Jane Blake, Managing Editor
Digital Technical Journal
Digital Equipment Corporation
50 Nagog Park, AKO2-3/B3
Acton, MA 01720-9843
Email: jane.blake@digital.com
Tel: 508-264-7552

Notice of the topics accepted will be sent to all authors by January 9, 1997. The manuscript-submission date for accepted topics is March 2, 1998.

For information on topics published in the *Journal*, the audience, writing guidelines, and the peer-review process, see http://www.digital.com/info/dtj/dtj-guide.htm or contact the managing editor at jane.blake@digital.com.

ISSN 0898-901X

Printed in U.S.A. EC-N7963-18/97 08 14 27.5 Copyright © Digital Equipment Corporation